

z/OS Communications Server



IP Programmer's Guide and Reference

Version 1 Release 9

z/OS Communications Server



IP Programmer's Guide and Reference

Version 1 Release 9

Note:

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 741.

Tenth Edition (September 2007)

This edition applies to Version 1 Release 9 of z/OS (5694-A01) and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. You may send your comments to the following address.

International Business Machines Corporation
Attn: z/OS Communications Server Information Development
Department AKCA, Building 501
P.O. Box 12195, 3039 Cornwallis Road
Research Triangle Park, North Carolina 27709-2195

You can send us comments electronically by using one of the following methods:

Fax (USA and Canada):

1+919-254-1258

Send the fax to “Attn: z/OS Communications Server Information Development”

Internet e-mail:

comsvrcf@us.ibm.com

World Wide Web:

<http://www.ibm.com/servers/eserver/zseries/zos/webqs.html>

If you would like a reply, be sure to include your name, address, telephone number, or FAX number. Make sure to include the following in your comment or note:

- Title and order number of this document
- Page number or topic related to your comment

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 1989, 2007. All rights reserved.

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

Contents

Figures	xvii

Tables	xix

About this document	xxiii

Who should read this document	xxiii
How this document is organized	xxiii
How to use this document	xxv
Determining whether a publication is current	xxv
How to contact IBM service	xxv
Conventions and terminology used in this document	xxv
Clarification of notes	xxvi
How to read a syntax diagram	xxvi
Symbols and punctuation	xxvi
Commands	xxvii
Parameters.	xxvii
Syntax examples	xxvii
Longer than one line	xxviii
Required operands	xxviii
Optional values	xxviii
Selecting more than one operand	xxviii
Nonalphanumeric characters	xxviii
Blank spaces in syntax diagrams.	xxix
Default operands	xxix
Variables.	xxix
Syntax fragments	xxix
Prerequisite and related information	xxix
Required information	xxix
Related information	xxx
Softcopy information.	xxx
Other documents	xxx
Redbooks	xxxi
Where to find related information on the Internet.	xxxii
DNS Web sites	xxxiii
Using LookAt to look up message explanations	xxxiii
Using IBM Health Checker for z/OS	xxxiv
How to send your comments	xxxiv

Summary of changes	xxxv

Chapter 1. General programming information	1

Overview of Distributed Protocol Interface (DPI) versions 1.1 and 2.0	1

Chapter 2. SNMP agent Distributed Protocol Interface version 1.1.	3

SNMP agents and subagents	3
Processing DPI requests	4
Processing a GET request	4
Processing a SET request	4
Processing a GET-NEXT request	4
Processing a REGISTER request	5
Processing a TRAP request	5
SNMP agent DPI header files.	5
Compiling and linking	5
Sample compile cataloged procedure additions	6

Sample link-edit cataloged procedure additions.	6
SNMP DPI library routines	6
mkDPList()	6
fDPIparse()	7
mkDPIregister()	7
mkDPIresponse()	8
mkDPIset()	9
mkDPItrap()	10
mkDPItrape()	10
pDPIpacket()	11
query_DPI_port()	12
Sample SNMP DPI client program for C sockets for version 1.1	13
Using the DPISAMPL program	13
DPISAMPN NCCFLST for the SNMP manager	14
Compiling and linking the DPISAMPL.C source code	15
dpiSample table MIB descriptions	16
The DPISAMPL.C source code	16

Chapter 3. SNMP agent Distributed Protocol Interface version 2.0 41

SNMP agents and subagents.	41
DPI agent requests	42
SNMP DPI version 2.0 library	43
SNMP DPI Version 2.0 API	43
Compiling and linking	44
From a UNIX System Services environment	44
From an MVS environment	44
DPI Version 1.x base code considerations	45
SNMP DPI API Version 1.1 considerations	45
Migrating your SNMP DPI subagent to Version 2.0	45
SNMP DPI environment variables	48
Subagent programming concepts	48
Related information	49
Specifying the SNMP DPI API	49
Connect processing	49
OPEN request	49
REGISTER request	50
GET processing	51
SET processing	52
GETNEXT processing	53
GETBULK processing request	54
TRAP request.	55
ARE_YOU_THERE request	55
UNREGISTER request.	55
CLOSE request	56
Multithreading programming considerations	56
Functions, data structures, and constants	57
Basic DPI API functions	58
The DPIDebug() function	59
The DPI_PACKET_LEN() macro	60
The fDPIparse() function	61
The fDPIset() function	62
The mkDPIAreYouThere() function	63
The mkDPIclose() function	64
The mkDPIopen() function	65
The mkDPIregister() function	67
The mkDPIresponse() function	69
The mkDPIset() function	71
The mkDPItrap() function	73
The mkDPIunregister() function	75
The pDPIpacket() function	76
Transport-related DPI API functions	77

The DpIawait_packet_from_agent() function	78
The DpIconnect_to_agent_TCP() function	80
The DpIconnect_to_agent_UNIXstream() function.	82
The DpIdisconnect_from_agent() function	84
The DpIget_fd_for_handle() function	85
The DpIsend_packet_to_agent() function.	86
The lookup_host() function	88
The lookup_host6() function.	89
DPI structures	90
The snmp_dpi_close_packet structure.	91
The snmp_dpi_get_packet structure	92
The snmp_dpi_hdr structure.	93
The snmp_dpi_next_packet structure	95
The snmp_dpi_resp_packet structure	96
The snmp_dpi_set_packet structure	97
The snmp_dpi_ureg_packet structure.	99
The snmp_dpi_u64 structure	100
Character set selection	101
Related information	101
Constants, values, return codes, and include file.	101
DPI CLOSE reason codes	102
Related information	102
DPI packet types	102
Related information	102
DPI RESPONSE error codes	102
Related information	103
DPI UNREGISTER reason codes	103
Related information	103
DPI SNMP value types	103
Related information	104
Value representation	104
Related information	105
Value ranges and limits	105
Return codes from DPI transport-related functions	105
Related information	106
The snmp_dpi.h include file	106
Parameters	106
Description	106
Related information	106
A DPI subagent example	107
Overview of subagent processing.	107
Connecting to the agent	109
Registering a subtree with the agent.	111
Processing requests from the agent	113
Processing a GET request	116
Processing a GETNEXT request	119
Processing a SET/COMMIT/UNDO request	122
Processing an UNREGISTER request	125
Processing a CLOSE request	126
Generating a TRAP	126
Chapter 4. Running the sample SNMP DPI client program for version 2.0	129
Using the sample program	129
Compiling and linking the dpi_mvs_sample.c source code	129
DPISimple-MIB descriptions	130
 Chapter 5. SNMP manager API	131
SNMP protocol.	131
The SNMP manager API overview	132
The SNMP notification API overview	132

SNMP manager API functions	132
snmpAddVarBind – Adds a VarBind to the SnmpVarBinds structure	133
snmpBuildPDU – Builds an SNMP PDU	133
snmpBuildSession – Creates a session	135
snmpCreateVarBinds – Creates a VarBind structure.	136
snmpFreeDecodedPDU - Free the decoded PDU.	136
snmpFreeOID - Free an OID string	137
snmpFreePDU – Frees the resources of a PDU	137
snmpFreeVarBinds – Frees the VarBinds structure	137
snmpGetErrorInfo - Get the error information from the PDU response	138
snmpGetNumberOfVarBinds – Get the number of VarBinds attached to the PDU.	138
snmpGetOID – Get the OID from the VarBind structure	139
snmpGetRequestId – Get the PDU's requestId value	139
snmpGetSockFd – Get the socket's file descriptor	140
snmpGetValue – Get the value from the VarBind structure	140
snmpGetVarbind – Get a VarBind attached to the PDU	141
snmpInitialize – Initialize the manager environment	141
snmpSendRequest – Send the snmpPDU request to an agent	142
snmpSetLogFunction – Set the logging level	144
snmpSetLogLevel – Set the logging level	144
snmpSetRequestId – Set the PDU's requestId value.	145
snmpTerminate – Release the resources.	146
snmpTerminateSession – Terminate a session.	146
snmpValueCreateBits – Create an smiValue of type Bits	147
snmpValueCreateCounter32 – Create an smiValue of type Counter32.	147
snmpValueCreateCounter64 – Create an smiValue of type Counter64.	147
snmpValueCreateGauge32 – Create an smiValue of type Gauge32	148
snmpValueCreateInteger – Create an smiValue of type Integer	148
snmpValueCreateInteger32 – Create an smiValue of type Integer32	149
snmpValueCreateIPAddr – Create an smiValue of type IPAddr	149
snmpValueCreateNSApAddr – Create an smiValue of type NSApAddr	150
snmpValueCreateNull – Create an smiValue of type Null.	150
snmpValueCreateOctet – Create an smiValue of type Octet	150
snmpValueCreateOID – Create an smiValue of type OID	151
snmpValueCreateOpaque – Create an smiValue of type Opaque	151
snmpValueCreateTimerTicks – Create an smiValue of type TimerTicks	152
snmpValueCreateUnsignedInteger32 – Create an smiValue of type UnsignedInteger32	152
SNMP notification API functions	153
snmpBuildV1TrapPDU – Builds an SNMP V1 trap PDU	153
snmpBuildV2TrapOrInformPDU – Builds a SNMP V2 trap or inform PDU	155
SNMP manager API configuration file	156
Statement syntax	156
Steps for compiling and linking SNMP manager API applications	158
Running your application	159
Debugging the SNMP manager API	159
Sample manager source code	160

Chapter 6. Resource Reservation Setup Protocol API (RAPI) 161

API outline	162
Compiling and linking RAPI applications	162
Running RAPI applications.	162
Event upcall.	163
rapi_event_rtn_t - Event upcall	163
Client library services	165
rapi_release - Remove a session	166
rapi_reserve - Make, modify, or delete a reservation	166
rapi_sender - Specify sender parameters	167
rapi_session - Create a session.	169
rapi_version - RAPI version	170
RAPI formatting routines	170
rapi_fmt_adspec - Format an adspec.	170

rapi_fmt_filtspec - Format a filtspec	171
rapi_fmt_flowspec - Format a flowspec	172
rapi_fmt_tspeg - Format a tspeg	173
RAPI objects	173
Flowspecs	174
Sender tspecs	174
Adspecs	174
Filter specs and sender templates	175
Asynchronous event handling	175
rapi_dispatch - Dispatch API event	176
rapi_getfd - Get file descriptor	177
Error handling	177
RAPI error codes	178
RSVP error codes	179
Header files	179
Integer and floating point types	179
The <rapi.h> header	180
Integrated services data structures and macros	186

Chapter 7. X Window System interface in the z/OS Communications Server

environment 195

X Window System and Motif	195
DLL support for the X Window System	196
How the X Window System interface works in the MVS environment	197
Programming considerations	198
Running an X Window System or Motif DLL-enabled application	199
X Window System environment variables	200
Motif environment variables	201
EBCDIC/ASCII translation in the X Window System	202
Standard clients supplied with MVS z/OS UNIX X Window System support	202
Demonstration programs supplied with MVS z/OS UNIX X Window System support	203
Where files are located	203

Chapter 8. Remote procedure calls in the z/OS Communications Server environment 207

The RPC interface	207
Portmapper and rpcbind	209
Contacting portmapper or rpcbind	210
Target assistance	210
Registering with rpcbind	211
Obtaining address lists from the rpcbind server	211
RPC servers in a CINET environment	212
RPCGEN command	213
clnt_stat enumerated type	215
Porting	215
Remapping file names with MANIFEST.H.	215
Accessing system return messages	216
Printing system return messages	216
Enumerations	216
Header files for remote procedure calls	216
Compiling and linking RPC applications	216
Compatibility considerations	216
Sample compile cataloged procedure additions	217
Nonreentrant modules	217
Reentrant modules	217
RPC global variables	217
rpc_createerr	218
svc_fds	219
svc_fdset	220
Remote procedure and external data representation calls	221
auth_destroy()	222

authnone_create()	223
authunix_create()	224
authunix_create_default()	225
callrpc()	226
clnt_broadcast()	228
clnt_call()	230
clnt_control()	232
clnt_create()	233
clnt_destroy()	234
clnt_freeres()	235
clnt_geterr()	236
clnt_pcreateerror()	237
clnt_perrno()	238
clnt_perror()	239
clnt_spcreateerror()	240
clnt_sperrno()	241
clnt_sperror()	242
clntraw_create()	243
clnttcp_create()	244
clntudp_create()	246
get_myaddress()	248
getrpcport()	249
pmap_getmaps()	250
pmap_getport()	251
pmap_rmtcall()	252
pmap_set()	254
pmap_unset()	255
registerrpc()	256
svc_destroy()	257
svc_freeargs()	258
svc_getargs()	259
svc_getcaller()	260
svc_getreq()	261
svc_getreqset()	262
svc_register()	263
svc_run()	264
svc_sendreply()	265
svc_unregister()	266
svcerr_auth()	267
svcerr_decode()	268
svcerr_noproc()	269
svcerr_noprog()	270
svcerr_progvers()	271
svcerr_systemerr()	272
svcerr_weakauth()	273
svccraw_create()	274
svctcp_create()	275
svcudp_create()	276
xdr_accepted_reply()	277
xdr_array()	278
xdr_authunix_parms()	279
xdr_bool()	280
xdr_bytes()	281
xdr_callhdr()	282
xdr_callmsg()	283
xdr_char()	284
xdr_destroy()	285
xdr_double()	286
xdr_enum()	287
xdr_float()	289
xdr_free()	290

xdr_getpos()	291
xdr_inline()	292
xdr_int()	293
xdr_long()	294
xdr_opaque()	295
xdr_opaque_auth()	296
xdr_pmap()	297
xdr_pmaplist()	298
xdr_pointer()	299
xdr_reference()	300
xdr_rejected_reply()	301
xdr_replymsg()	302
xdr_setpos()	303
xdr_short()	304
xdr_string()	305
xdr_text_char()	306
xdr_u_char()	307
xdr_u_int()	308
xdr_u_long()	309
xdr_u_short()	310
xdr_union()	311
xdr_vector()	313
xdr_void()	314
xdr_wrapstring()	315
xdrmem_create()	316
xdrrec_create()	317
xdrrec_endofrecord()	318
xdrrec_eof()	319
xdrrec_skiprecord()	320
xdrstdio_create()	321
xprt_register()	322
xprt_unregister()	323
Sample RPC programs	324
Running RPC sample programs	324
RPC client	324
RPC server	326
RPC raw data stream.	328
RPCGEN sample programs.	331
Generating your own sequential data sets	331
Building client and server executable modules	332
Running RPCGEN sample programs	332
Chapter 9. Remote procedure calls in the z/OS UNIX System Services environment	333
Deviations from Sun RPC 4.0	333
Source margins.	333
Functions.	333
Using z/OS UNIX System Services RPC	334
Support for 64-bit integers	334
UDP transport protocol CLIENT handles	335
Restrictions	335
Chapter 10. Network Computing System.	337
NCS and the Network Computing Architecture	337
NCS components	337
Remote procedure call runtime library	337
Location broker.	338
Network interface definition language compiler	338
MVS implementation of NCS	338
NCS system IDL data sets	339
NCS C header data sets and the Pascal include data set	340

NCS RPC run-time library	340
Portability issues	340
NCS defines NCSDEFS.H	340
Required user-defined USERDEFS.H	341
Preprocessing, compiling, and linking	342
NCS preprocessor programs	342
Compiling and linking NCS programs	346
Running UUID@GEN	348
NCS sample programs	348
The NCSSMP sample program	349
NCS sample redefines	349
Compiling, linking, and running the sample BINOP program	349
Setup	350
Compile	351
Link	352
Run	353
Compiling, linking, and running the NCSSMP program	353
Setup	354
Compile	355
Link	356
Run	357
Compiling, linking, and running the sample BANK program	358
Setup	358
Compile	360
Link	361
Run	362

Chapter 11. Running the Sample Mail Filter program 363

Compiling and linking the lf_smpl.c source code	363
Specifying filters in the sendmail configuration file	363
Running the sample program	363
Library control functions	364
smfi_register	364
smfi_setconn	365
smfi_settimeout	365
smfi_main	366
Data access functions.	366
smfi_getsymval.	366
smfi_getpriv.	367
smfi_setpriv	367
smfi_setreply	368
Message modification functions	368
smfi_addheader	368
smfi_chgheader.	369
smfi_addrcpt	370
smfi_delrcpt	371
smfi_replacebody	371
Callbacks.	372
xxfi_connect - connection information	372
xxfi_helo - SMTP HELO/EHLO command	373
xxfi_envfrom - envelope sender	373
xxfi_envrcpt - envelope recipient	374
xxfi_header - header	374
xxfi_eoh - end of header.	375
xxfi_body - body block	375
xxfi_eom - end of message	376
xxfi_abort - message aborted	376
xxfi_close - connection cleanup	377

Chapter 12. Policy API (PAPI) 379

API outline for retrieving data from Policy Agent	379
Compiling and linking PAPI applications	379
Running PAPI applications	380
PAPI return codes	380
Client library services	381
Connecting and retrieving data	382
papi_connect - Connect to Policy Agent	382
papi_debug - Set debug capability	382
papi_disconnect - Disconnect from the Policy Agent	383
papi_free_perf_data - Free retrieved QoS performance data	384
papi_get_perf_data - Retrieve QoS performance data	384
Helper functions	387
papi_get_action_perf_by_id- Obtain performance information on the action specified by the action ID.	387
papi_get_action_perf_info - Obtain performance information on a particular action	387
papi_get_actions_count - Obtain number of actions in the policy performance data	388
papi_get_policy_instance - Obtain policy instance number for policies in the policy performance data.	389
papi_get_rule_perf_by_id - Obtain performance information on the rule specified by the rule ID	389
papi_get_rule_perf_info - Obtain performance information on a particular rule	390
papi_get_rules_count - Obtain number of rules in the policy performance data	391
papi_strerror - Return string describing PAPI return code value	391
Chapter 13. FTP Client Application Programming Interface (API)	393
Compatibility considerations	394
Guidelines and requirements for the FTP Client API	394
COBOL, C, REXX, Assembler, and PL/I call formats	396
Converting parameter descriptions	397
z/OS FTP client behavior when invoked from the FTP Client API.	397
FTP Client Application Interface (FCAI) control block	399
FTP Client Application Interface (FCAI) stem variables	407
Predefined REXX variables	408
Sending requests to the FTP Client API.	413
INIT	413
SCMD.	417
POLL	420
GETL	422
TERM.	428
FTP Client API C functions.	430
FAPI_INIT	430
FAPI_SCMD.	431
FAPI_POLL	431
FAPI_GETL_COPY	432
FAPI_GETL_FIND.	432
FAPI_TERM.	433
FTP Client API REXX function	433
Handling of SIGCHLD signals.	434
REXX FTP Client API trace	434
FTP Client API requests	436
REXX FTP Client API trace return codes	455
Output register information for the FTP Client API.	455
Other output that is returned to the application	456
Prompts from the client	457
Prompts not used by the FTP Client API	457
Prompts returned in FCAI-Status.	457
Command prompt.	458
Messages and replies	458
Interpreting results from an interface request.	459
FCAI request completion values	459
Considerations when evaluating request completion values	461
Programming notes for the FTP Client API	461
FCAI_Status_TraceFailed and FCAI_TraceStatus — reporting failures in the interface trace function	462
FCAI_IE_LengthInvalid — improper lengths passed to the interface	462

FCAI_ReqTimer — controlling requests that retrieve results from the spawned z/OS FTP client process . . .	463
FCAI_PollWait — specifying a wait time before POLL.	463
FCAI_IE_InternalErr — unanticipated exceptional conditions in the interface	464
Exceptional conditions in the z/OS FTP client	464
Using the FTP Client API trace	465
FTP Client API sample programs.	469

Chapter 14. Network management interfaces (NMIs). 471

Local IPsec NMI	472
Configuring the interface	473
Connecting to the server	473
IPsec NMI request/response format.	474
Request messages	479
Monitoring request format	481
Control request formats	487
Response messages	491
Initialization and termination messages.	517
Return and reason codes	517
Network security services NMI	520
Overview.	521
Configuring the interface	521
Connecting to the server	522
Network security services NMI request/response format.	522
Request format.	522
Response format	523
Initialization and termination messages.	524
Return and reason codes	525
Real-time TCP/IP network monitoring NMI	528
Overview.	529
Configuration and enablement	531
Connecting to the server	532
Interacting with the servers.	532
Common record header	533
Requests sent by the client to the server	533
Records sent by the server to the client.	534
Copying the trace buffer.	536
EZBTMIC1 - Copy TCP/IP Management Interface Data Buffer	536
TMI_CopyBuffer - Copy TCP/IP Management Interface Data Buffer	538
Understanding the common buffer output of the TMI copy buffer service	539
Format of service-specific data.	540
Real-time SMF data NMI (SYSTCPSM) record formats.	545
Packet and data trace formatting NMI	566
Overview.	567
Configuration and enablement	568
EZBCTAPI Network Management interface for formatting packet trace records	568
Passing options to the packet trace formatter.	581
Using the formatter	582
TCP/IP callable NMI (EZBNMIFR)	583
Poll-type requests	583
Action-type requests	583
Overview.	583
Configuration and enablement	584
EZBNMIFR — TCP/IP network management request	584
TCP/IP network management interface request format	587
TCP/IP network management interface response format	593
Request and response data structures	596
Examples.	596
SNA network monitoring NMI	600
Overview.	601
Configuration	601
Enabling and disabling the interface.	601

Communicating with the server	602
SNA network management interface (NMI) request/response format	602
NMI request errors	612
Diagnosis.	613
File storage locations	614
Chapter 15. Application Transparent Transport Layer Security (AT-TLS)	617
CICS transaction considerations	618
Using the SIOCTLCTL ioctl	619
Starting AT-TLS on a connection	619
Stopping AT-TLS on a connection	619
Requesting AT-TLS queries and additional functions	620
Steps for implementing an aware server application	620
Steps for implementing a controlling server application	620
Coding the SIOCTLCTL ioctl	623
General coding guidelines	623
SIOCTLCTL (X'C038D90B')	624
Return values	631
Coding examples	633
Chapter 16. Interfacing with the Digital Certificate Access Server (DCAS)	637
Overview.	637
Understanding how clients interface to DCAS	637
Defining the format for request and response specifications	638
Configuring the DCAS server to work with your solution	641
Chapter 17. Miscellaneous programming interfaces	643
SIOCSAPPLDATA IOCTL	643
Input	643
Output	644
C language example	644
TCP_KeepAlive socket option	645
Appendix A. TCP/IP in the sysplex	647
Appendix B. Well-known port assignments	651
Well-known UDP port assignments	652
Appendix C. Programming interfaces for providing classification data to be used in differentiated services policies	655
Passing application classification data on SENDMSG	656
Additional considerations	659
Appendix D. X Window System interface V11R4 and Motif version 1.1.	661
What is provided	661
Software requirements	662
How the X Window System interface works in the MVS environment	662
Identifying the target display	664
Application resource file.	664
Creating an application	665
X Window System header files	665
Compiling and linking	667
Nonreentrant modules	667
Reentrant modules	669
Using sample X Window System programs	671
Running a sample program.	671
Environment variables	671
Standard X client applications	672
Building X client modules	674

X Window System routines	677
Opening and closing a display	677
Creating and destroying windows	677
Manipulating windows	677
Changing window attributes	678
Obtaining window information	678
Obtaining properties and atoms	678
Manipulating window properties.	678
Setting window selections	679
Manipulating colormaps.	679
Manipulating color cells	679
Creating and freeing pixmaps	680
Manipulating graphics contexts	680
Clearing and copying areas.	681
Drawing lines	681
Filling areas	681
Loading and freeing fonts	682
Querying character string sizes	682
Drawing text	683
Transferring images	683
Manipulating cursors.	683
Handling window manager functions	683
Manipulating keyboard settings	684
Controlling the screen saver	685
Manipulating hosts and access control	685
Handling events	685
Enabling and disabling synchronization	686
Using default error handling	686
Communicating with window managers	686
Manipulating keyboard event functions	688
Manipulating regions.	688
Using cut and paste buffers	689
Querying visual types	689
Manipulating images	689
Manipulating bit maps	690
Using the resource manager	690
Manipulating display functions	691
Extension routines.	693
MIT extensions to X	694
Associate table functions	695
Miscellaneous utility routines	695
X authorization routines.	698
X Window System toolkit	698
Xt Intrinsics routines	700
Application resources.	708
Athena widget support	709
Motif-based widget support	712
z/OS UNIX System Services support	713
What is provided with z/OS UNIX System Services	713
z/OS UNIX System Services software requirements	714
z/OS UNIX System Services application resource file	714
Identifying the target display in z/OS UNIX System Services	714
Compiling and linking with z/OS UNIX System Services	715
Compiling and linking with z/OS UNIX System Services using c89	716
Standard X client applications for z/OS UNIX System Services.	717
Application resources for z/OS UNIX System Services	717

Appendix E. Related protocol specifications 719

Internet drafts	734

Appendix F. Information APARs and technotes 735

Information APARs for IP documents	735
Information APARs for SNA documents	736
Other information APARs	736
Appendix G. Accessibility	739
Using assistive technologies	739
Keyboard navigation of the user interface	739
z/OS information	739
Notices	741
Programming interface information	749
Trademarks	750
Bibliography.	753
z/OS Communications Server information	753
z/OS Communications Server library	753
Planning	753
Resource definition, configuration, and tuning	753
Operation	754
Customization	755
Writing application programs	755
Diagnosis.	756
Messages and codes	756
Index	757
Communicating Your Comments to IBM	769

Figures

1.	SNMP Dist Prog Interface subagent sample	17
2.	Remote procedure call (client)	208
3.	Remote procedure call (server).	209
4.	RPC client program sample	325
5.	RPC server program sample	327
6.	RPC raw data stream program sample	329
7.	Macro to maintain IBM System/370 portability	341
8.	NCSDEFS.H and USERDEFS.H include statements	341
9.	Message header and records	477
10.	NMI monitoring request format	481
11.	NMsec_ACTIVATE_IPTUNMANUAL request form	488
12.	NMsec_ACTIVATE_IPTUNDYN request format	489
13.	NMsec_DEACTIVATE_IPTUNMANUAL request format	489
14.	NMsec_DEACTIVATE_IPTUNDYN request format	489
15.	NMsec_DEACTIVATE_IKETUN request format	490
16.	NMsec_REFRESH_IPTUNDYN request format	490
17.	NMsec_REFRESH_IKETUN request format	490
18.	NMsec_GET_STACKINFO response format	491
19.	NMsec_GET_SUMMARY response format	492
20.	NMsec_GET_IPFLTCURR, NMsec_GET_IPFLTDEFAULT, and NMsec_GET_IPFLTPOLICY response format	496
21.	NMsec_GET_PORTTRAN response format.	501
22.	NMsec_GET_IPTUNMANUAL response format	502
23.	NMsec_GET_IPTUNDYNSTACK response format	504
24.	NMsec_GET_IPTUNDYNIKE response format	509
25.	NMsec_GET_IKETUN response format	510
26.	NMsec_GET_IKETUNCASCADE response format	514
27.	NMsec_GET_IPINTERFACES response format	514
28.	NMsec_GET_IKENSINFO response format.	515
29.	Tunnel control response format	517
30.	NMsec_GET_CLIENTINFO response format	523
31.	CTE layout	568
32.	SIOCTLCTL with TTLS_Query_Only	622
33.	SIOCTLCTL with TTLS_Init_Connection.	623
34.	SIOCTLCTL with TTLS_Reset_Session or TTLS_Reset_Cipher	623
35.	MVS X Window System application to server.	663
36.	Resources specified for a typical X Window System application	665

Tables

1.	Components of DPI version 2.0	43
2.	Environment variables for the SNMP DPI	48
3.	SNMP manager API debug levels	159
4.	Environment variables for the X Window System interface	200
5.	Environment variables for Motif	201
6.	Callback return values	372
7.	PAPI function return codes	380
8.	Programming requirements for the FTP Client API	394
9.	FCAI control block.	399
10.	FCAI_Version field value	401
11.	FCAI_TraceIt field value	401
12.	FCAI_TraceCAPI field value	402
13.	FCAI_TraceStatus field value	402
14.	FCAI_Result field value	402
15.	FCAI_Status field values	403
16.	FCAI_IE field values	403
17.	FCAI stem variables	407
18.	Predefined REXX variables	408
19.	REXX FTP Client API return codes	437
20.	FTP client CREATE request return codes	438
21.	FTP client INIT request return codes	441
22.	FTP client SCMD request return codes	442
23.	FTP client POLL request return codes	444
24.	FTP client GETL_FIND request return codes	447
25.	FTP client GETL_COPY request return codes	448
26.	FTP client SET_TRACE request return codes	450
27.	FTP client SET_REQUEST_TIMER request return codes	451
28.	FCAI_Map structure elements	452
29.	FTP client GET_FCAI_MAP request return codes	453
30.	FTP client TERM request return codes	454
31.	REXX FTP Client API trace return codes	455
32.	NMsecMessageHeader structure	474
33.	Input record descriptor	476
34.	Output record descriptor.	476
35.	NMsecRecordHdr structure	477
36.	NMsecSecDesc structure	478
37.	NMsecCascadingSecDesc structure	478
38.	Valid input filter specifications for request types	482
39.	NMsecInFilter structure	483
40.	NMsecTunnel field descriptions	487
41.	NMsecPolicySource data	491
42.	NMsecStack structure	491
43.	NMsec_GET_SUMMARY structure	493
44.	NMsecIPFilter structure	496
45.	NMsec_GET_PORTTRAN structure	501
46.	NMsecIPTunnel structure	502
47.	NMsecIPManualTunnel structure	504
48.	NMsecIPDynTunnel structure	504
49.	NMsecIPDynamicStack structure	508
50.	NMsecIPDynamicIKE structure	509
51.	NMsecIKETunnel structure	511
52.	IKE tunnel statistics	513
53.	NMsecInterface structure.	514
54.	NMsec_GET_IKENSINFO structure	515
55.	NMsecTunCntlResponse structure.	517

	56.	Return and reason codes	517
	57.	NMsecNSClient structure	523
	58.	Request return and reason codes	525
	59.	z/OS Communications Server network management interfaces	528
	60.	Interface descriptions	529
	61.	TCP/IP identification section format	545
	62.	FTP server transfer initialization self-defining section	546
	63.	FTP server transfer initialization record section	547
	64.	FTP server hostname section	548
	65.	FTP server MVS data set name section	549
	66.	FTP server z/OS UNIX file name section	549
	67.	FTP server security section	549
	68.	FTP server session record	550
	69.	Server session section	551
	70.	FTP server security section	552
	71.	FTP client transfer initialization record section	555
	72.	FTP client associated data set name section	556
	73.	FTP client SOCKS section	556
	74.	FTP client security section	557
	75.	FTP client user name section	558
	76.	FTP client login failure self-defining section	558
	77.	Client login failure session section	559
	78.	FTP client SOCKS section	560
	79.	FTP client login failure security section	560
	80.	FTP client user name section	562
	81.	FTP client session record self-defining section	562
	82.	FTP client session section	563
	83.	FTP client SOCKS section	564
	84.	FTP client security section	565
	85.	FTP client session user name section	566
	86.	EZBCTAPI return and reason codes	576
	87.	EZBCTAPI formatter return and reason codes	577
	88.	Available EZBYPTO options	581
	89.	NWMDropConnEntry description	590
	90.	Filter items applicable to each request type	593
	91.	Return code values	594
	92.	File storage locations	614
	93.	Application types	617
	94.	TTLShHeader control block structure	628
	95.	TTLShHeader fixed section	628
	96.	TTLShQuadruplet structure	628
	97.	Get request structure	629
	98.	Example TTLShHeader structure	630
	99.	Example returned TTLShHeader structure	631
	100.	Format 1 request	638
	101.	Format 1 response	638
	102.	Format 2 request	639
	103.	Understanding return codes in the response	639
	104.	DCAS client and server coordination	641
	105.	SetApplData	643
	106.	SetADcontainer	644
	107.	SIOCSAPPLDATA IOCTL return and reason codes	644
	108.	TCP_KeepAlive time values	645
	109.	SO_CLUSTERCONNTYPE socket option benefits	647
	110.	TCP well-known port assignments	651
	111.	Well-known UDP port assignments	652
	112.	Environment variables for X Window System Interface V11r4	671
	113.	Building X client modules based on X11 functions	675
	114.	Building X client modules based on Xt Intrinsics and Athena Toolkit functions	676
	115.	Opening and closing display	677
	116.	Creating and destroying windows	677

117.	Manipulating windows	677
118.	Changing window attributes	678
119.	Obtaining window information	678
120.	Properties and atoms	678
121.	Manipulating window properties	679
122.	Setting window selections	679
123.	Manipulating colormaps	679
124.	Manipulating color cells	679
125.	Creating and freeing pixmaps	680
126.	Manipulating graphics contexts	680
127.	Clearing and copying areas	681
128.	Drawing lines	681
129.	Filling areas	681
130.	Loading and freeing fonts	682
131.	Querying character string sizes	682
132.	Drawing text.	683
133.	Transferring images	683
134.	Manipulating cursors	683
135.	Handling window manager functions	684
136.	Manipulating keyboard settings	684
137.	Controlling the screen saver.	685
138.	Manipulating hosts and access control	685
139.	Handling events	685
140.	Enabling and disabling synchronization.	686
141.	Using default error handling	686
142.	Communicating with window managers	687
143.	Manipulating keyboard event functions.	688
144.	Manipulating regions	688
145.	Using cut and paste buffers.	689
146.	Querying visual types.	689
147.	Manipulating images	690
148.	Manipulating bit maps	690
149.	Using the resource manager.	690
150.	Manipulating display functions	691
151.	Extension routines	693
152.	MIT extensions to X	694
153.	Associate table functions.	695
154.	Miscellaneous utility routines	695
155.	Authorization routines	698
156.	X Intrinsic header file names	699
157.	Xt Intrinsics routines	700
158.	Athena widget routines	709
159.	Athena header file names	711
160.	Motif header file names	713
161.	IP information APARs for z/OS Communications Server	735
162.	SNA information APARs for z/OS Communications Server	736
163.	Non-document information APARs	736

About this document

This document describes the syntax and semantics of a set of high-level application functions that you can use to program your own applications in a TCP/IP environment. These functions provide support for application facilities, such as user authentication, distributed databases, distributed processing, network management, and device sharing. The information in this document supports both IPv6 and IPv4. Unless explicitly noted, information describes IPv4 networking protocol. IPv6 support is qualified within the text.

Who should read this document

This document is intended for use by an experienced programmer familiar with the IBM® z/OS® operating system and commands, and with the TCP/IP protocols.

This document is written for programmers interested in high-level application functions that can be used to program applications in a TCP/IP environment. These functions involve user authentication, distributed databases, distributed processing, network management, and device sharing.

Before using this document, you should be familiar with the MVS™ operating system and the IBM Time Sharing Option (TSO).

Depending on the design and function of your application, you should be familiar with the C programming language.

In addition, z/OS Communications Server and any required programming products should already be installed and customized for your network.

How this document is organized

The *z/OS Communications Server: IP Programmer's Guide and Reference* contains the following topics:

- Chapter 1, "General programming information," on page 1 provides an overview of Distributed Protocol Interface (DPI®) versions 1.1 and 2.0.
- Chapter 2, "SNMP agent Distributed Protocol Interface version 1.1," on page 3 provides information about SNMP agent DPI version 1.1 agents and subagents, compile and link samples, descriptions of DPI library routines, and a sample client program for C sockets.
- Chapter 3, "SNMP agent Distributed Protocol Interface version 2.0," on page 41 provides SNMP agent DPI version 2.0 programming information.
- Chapter 4, "Running the sample SNMP DPI client program for version 2.0," on page 129 explains how to run the sample SNMP DPI client program, `dpi_mvs_sample.c`, installed in `/usr/lpp/tcpip/samples`.
- Chapter 5, "SNMP manager API," on page 131 describes how use this API to build SNMP management applications to retrieve SNMP management data.
- Chapter 6, "Resource Reservation Setup Protocol API (RAPI)," on page 161 describes the calls provided through a set of C language bindings that provide an API for requesting enhanced Quality of Service (QoS).
- Chapter 7, "X Window System interface in the z/OS Communications Server environment," on page 195 describes the X Window System API.

- Chapter 8, “Remote procedure calls in the z/OS Communications Server environment,” on page 207 describes the high-level remote procedure calls (RPCs) implemented in TCP/IP, including the RPC programming interface to the C language and communication between processes.
- Chapter 9, “Remote procedure calls in the z/OS UNIX System Services environment,” on page 333 provides information on use of UNIX[®] System Services RPC and deviations from Sun RPC 4.0.
- Chapter 10, “Network Computing System,” on page 337 describes the NCS tools used for heterogeneous distributed computing.
- Chapter 11, “Running the Sample Mail Filter program,” on page 363 explains how to run the sample mail filter program, `lf_smpl.c`.
- Chapter 12, “Policy API (PAPI),” on page 379 describes the Policy Agent API (PAPI).
- Chapter 13, “FTP Client Application Programming Interface (API),” on page 393 describes the callable application programming interface to the z/OS FTP client.
- Chapter 14, “Network management interfaces (NMIs),” on page 471 describes the interfaces that allow network monitor and management applications to obtain information about their network operations, for both TCP/IP and VTAM[®].
- Chapter 15, “Application Transparent Transport Layer Security (AT-TLS),” on page 617 describes Application Transparent Transport Layer Security (AT-TLS), which creates a secure session at the TCP/IP layer on behalf of an application.
- Chapter 16, “Interfacing with the Digital Certificate Access Server (DCAS),” on page 637 documents the programming interface specifications for the Digital Certificate Access Server (DCAS) that runs on the z/OS operating system.
- Chapter 17, “Miscellaneous programming interfaces,” on page 643 describes programming interfaces including the TCP_KeepAlive function.
- Appendix A, “TCP/IP in the sysplex,” on page 647 introduces the `SO_CLUSTERCONNTYPE` socket option for obtaining sysplex connection endpoint characteristics.
- Appendix B, “Well-known port assignments,” on page 651 lists the well-known port assignments for transport protocols TCP and UDP.
- Appendix C, “Programming interfaces for providing classification data to be used in differentiated services policies,” on page 655 provides information on the Differentiated Services (DS) aspect of QoS and the passing of application classification data on `SENDMSG`.
- Appendix D, “X Window System interface V11R4 and Motif version 1.1,” on page 661 describes the X Window System application programming interface (API).
- Appendix E, “Related protocol specifications,” on page 719 lists the related protocol specifications for TCP/IP.
- “Information APARs and technotes,” lists information APARs for IP and SNA documents.
- “Accessibility” describes accessibility features to help users with physical disabilities.
- “Notices” contains notices and trademarks used in this document.
- “Bibliography” contains descriptions of the documents in the z/OS Communications Server library.

How to use this document

To use this document, you should be familiar with z/OS TCP/IP Services and the TCP/IP suite of protocols.

Determining whether a publication is current

As needed, IBM updates its publications with new and changed information. For a given publication, updates to the hardcopy and associated BookManager® softcopy are usually available at the same time. Sometimes, however, the updates to hardcopy and softcopy are available at different times. The following information describes how to determine if you are looking at the most current copy of a publication:

- At the end of a publication's order number there is a dash followed by two digits, often referred to as the dash level. A publication with a higher dash level is more current than one with a lower dash level. For example, in the publication order number GC28-1747-07, the dash level 07 means that the publication is more current than previous levels, such as 05 or 04.
- If a hardcopy publication and a softcopy publication have the same dash level, it is possible that the softcopy publication is more current than the hardcopy publication. Check the dates shown in the Summary of Changes. The softcopy publication might have a more recently dated Summary of Changes than the hardcopy publication.
- To compare softcopy publications, you can check the last two characters of the publication's file name (also called the book name). The higher the number, the more recent the publication. Also, next to the publication titles in the CD-ROM booklet and the readme files, there is an asterisk (*) that indicates whether a publication is new or changed.

How to contact IBM service

For immediate assistance, visit this Web site:

<http://www.software.ibm.com/network/commserver/support/>

Most problems can be resolved at this Web site, where you can submit questions and problem reports electronically, as well as access a variety of diagnosis information.

For telephone assistance in problem diagnosis and resolution (in the United States or Puerto Rico), call the IBM Software Support Center anytime (1-800-IBM-SERV). You will receive a return call within 8 business hours (Monday – Friday, 8:00 a.m. – 5:00 p.m., local customer time).

Outside of the United States or Puerto Rico, contact your local IBM representative or your authorized IBM supplier.

If you would like to provide feedback on this publication, see “Communicating Your Comments to IBM” on page 769.

Conventions and terminology used in this document

Commands in this book that can be used in both TSO and z/OS UNIX environments use the following conventions:

- When describing how to use the command in a TSO environment, the command is presented in uppercase (for example, NETSTAT).
- When describing how to use the command in a z/OS UNIX environment, the command is presented in bold lowercase (for example, **netstat**).
- When referring to the command in a general way in text, the command is presented with an initial capital letter (for example, Netstat).

All of the exit routines described in this document are *installation-wide exit routines*. You will see the installation-wide exit routines also called installation-wide exits, exit routines, and exits throughout this document.

The TPF logon manager, although shipped with VTAM, is an application program. Therefore, the logon manager is documented separately from VTAM.

Samples used in this book might not be updated for each release. Evaluate a sample carefully before applying it to your system.

For definitions of the terms and abbreviations used in this document, you can view the latest IBM terminology at the IBM Terminology Web site.

Clarification of notes

Information traditionally qualified as **Notes** is further qualified as follows:

Note Supplemental detail

Tip Offers shortcuts or alternative ways of performing an action; a hint

Guideline
Customary way to perform a procedure

Rule Something you must do; limitations on your actions

Restriction
Indicates certain conditions are not supported; limitations on a product or facility

Requirement
Dependencies, prerequisites

Result Indicates the outcome

How to read a syntax diagram

This syntax information applies to all commands and statements that do not have their own syntax described elsewhere.

The syntax diagram shows you how to specify a command so that the operating system can correctly interpret what you type. Read the syntax diagram from left to right and from top to bottom, following the horizontal line (the main path).

Symbols and punctuation

The following symbols are used in syntax diagrams:

Symbol	Description
►►	Marks the beginning of the command syntax.

- You must include all punctuation such as colons, semicolons, commas, quotation marks, and minus signs that are shown in the syntax diagram.

Commands that can be used in both TSO and z/OS UNIX environments use the following conventions in syntax diagrams:

- ## Parameters

Required

Optional

Default

Parameters are classified as keywords or variables. For the TSO and MVS console commands, the keywords are not case sensitive. You can code them in uppercase or lowercase. If the keyword appears in the syntax diagram in both uppercase and lowercase, the uppercase portion is the abbreviation for the keyword (for example, OPERand).

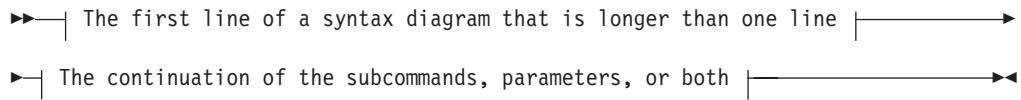
Variables are italicized, appear in lowercase letters, and represent names or values you supply. For example, a data set is a variable.

In the following example, the `USER` command is a keyword. The required variable parameter is `user_id`, and the optional variable parameter is `password`. Replace the variable parameters with your own values.



Longer than one line

If a diagram is longer than one line, the first line ends with a single arrowhead and the second line begins with a single arrowhead.



Required operands

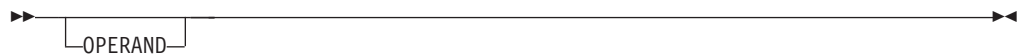
Required operands and values appear on the main path line.



You must code required operands and values.

Optional values

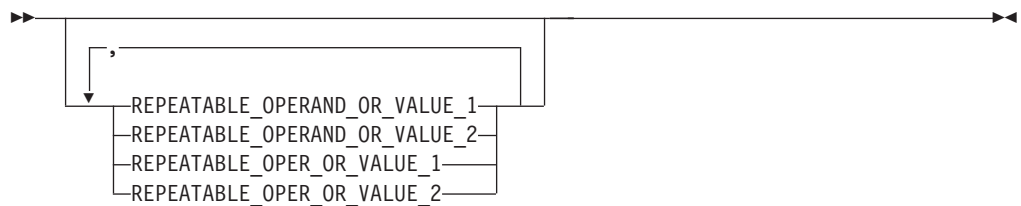
Optional operands and values appear below the main path line.



You can choose not to code optional operands and values.

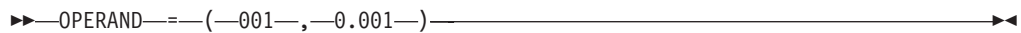
Selecting more than one operand

An arrow returning to the left above a group of operands or values means more than one can be selected, or a single one can be repeated.



Nonalphanumeric characters

If a diagram shows a character that is not alphanumeric (such as parentheses, periods, commas, and equal signs), you must code the character as part of the syntax. In this example, you must code OPERAND=(001,0.001).



Blank spaces in syntax diagrams

If a diagram shows a blank space, you must code the blank space as part of the syntax. In this example, you must code OPERAND=(001 FIXED).

▶▶—OPERAND—=(—001— —FIXED—)—▶▶

Default operands

Default operands and values appear above the main path line. TCP/IP uses the default if you omit the operand entirely.

▶▶—

DEFAULT
OPERAND

—▶▶

Variables

A word in all lowercase italics is a *variable*. Where you see a variable in the syntax, you must replace it with one of its allowable names or values, as defined in the text.

▶▶—*variable*—▶▶

Syntax fragments

Some diagrams contain syntax fragments, which serve to break up diagrams that are too long, too complex, or too repetitious. Syntax fragment names are in mixed case and are shown in the diagram and in the heading of the fragment. The fragment is placed below the main diagram.

▶▶—| Syntax fragment |—▶▶

Syntax fragment:

|—1ST_OPERAND—,—2ND_OPERAND—,—3RD_OPERAND—|

Prerequisite and related information

z/OS Communications Server function is described in the z/OS Communications Server library. Descriptions of those documents are listed in “z/OS Communications Server information” on page 753, in the back of this document.

Required information

Before using this product, you should be familiar with TCP/IP, VTAM, MVS, and UNIX System Services.

Related information

This section contains subsections on:

- “Softcopy information”
- “Other documents”
- “Redbooks” on page xxxi
- “Where to find related information on the Internet” on page xxxii
- “Using LookAt to look up message explanations” on page xxxiii
- “Using IBM Health Checker for z/OS” on page xxxiv

Softcopy information

Softcopy publications are available in the following collections:

Titles	Order Number	Description
<i>z/OS V1R9 Collection</i>	SK3T-4269	This is the CD collection shipped with the z/OS product. It includes the libraries for z/OS V1R9, in both BookManager and PDF formats.
<i>z/OS Software Products Collection</i>	SK3T-4270	This CD includes, in both BookManager and PDF formats, the libraries of z/OS software products that run on z/OS but are not elements and features, as well as the <i>Getting Started with Parallel Sysplex</i> [®] bookshelf.
<i>z/OS V1R9 and Software Products DVD Collection</i>	SK3T-4271	This collection includes the libraries of z/OS (the element and feature libraries) and the libraries for z/OS software products in both BookManager and PDF format. This collection combines SK3T-4269 and SK3T-4270.
<i>z/OS Licensed Product Library</i>	SK3T-4307	This CD includes the licensed documents in both BookManager and PDF format.
<i>IBM System z Redbooks Collection</i>	SK3T-7876	The Redbooks selected for this CD series are taken from the IBM Redbooks inventory of over 800 books. All the Redbooks that are of interest to the zSeries platform professional are identified by their authors and are included in this collection. The zSeries subject areas range from e-business application development and enablement to hardware, networking, Linux, solutions, security, parallel sysplex, and many others.

Other documents

For information about z/OS products, refer to *z/OS Information Roadmap* (SA22-7500). The Roadmap describes what level of documents are supplied with each release of z/OS Communications Server, as well as describing each z/OS publication.

Relevant RFCs are listed in an appendix of the IP documents. Architectural specifications for the SNA protocol are listed in an appendix of the SNA documents.

The following table lists documents that might be helpful to readers.

Title	Number
<i>DNS and BIND</i> , Fourth Edition, O'Reilly and Associates, 2001	ISBN 0-596-00158-4

Title	Number
<i>Routing in the Internet</i> , Christian Huitema (Prentice Hall PTR, 1995)	ISBN 0-13-132192-7
<i>sendmail</i> , Bryan Costales and Eric Allman, O'Reilly and Associates, 2002	ISBN 1-56592-839-3
<i>SNA Formats</i>	GA27-3136
<i>TCP/IP Illustrated, Volume I: The Protocols</i> , W. Richard Stevens, Addison-Wesley Publishing, 1994	ISBN 0-201-63346-9
<i>TCP/IP Illustrated, Volume II: The Implementation</i> , Gary R. Wright and W. Richard Stevens, Addison-Wesley Publishing, 1995	ISBN 0-201-63354-X
<i>TCP/IP Illustrated, Volume III</i> , W. Richard Stevens, Addison-Wesley Publishing, 1995	ISBN 0-201-63495-3
<i>TCP/IP Tutorial and Technical Overview</i>	GG24-3376
<i>Understanding LDAP</i>	SG24-4986
<i>z/OS Cryptographic Service System Secure Sockets Layer Programming</i>	SC24-5901
<i>z/OS Integrated Security Services LDAP Client Programming</i>	SC24-5924
<i>z/OS Integrated Security Services LDAP Server Administration and Use</i>	SC24-5923
<i>z/OS JES2 Initialization and Tuning Guide</i>	SA22-7532
<i>z/OS Problem Management</i>	G325-2564
<i>z/OS MVS Diagnosis: Reference</i>	GA22-7588
<i>z/OS MVS Diagnosis: Tools and Service Aids</i>	GA22-7589
<i>z/OS MVS Using the Subsystem Interface</i>	SA22-7642
<i>z/OS Program Directory</i>	GI10-0670
<i>z/OS UNIX System Services Command Reference</i>	SA22-7802
<i>z/OS UNIX System Services Planning</i>	GA22-7800
<i>z/OS UNIX System Services Programming: Assembler Callable Services Reference</i>	SA22-7803
<i>z/OS UNIX System Services User's Guide</i>	SA22-7801
<i>z/OS XL C/C++ Run-Time Library Reference</i>	SA22-7821
<i>System z9 and zSeries OSA-Express Customer's Guide and Reference</i>	SA22-7935

Redbooks

The following Redbooks™ might help you as you implement z/OS Communications Server.

Title	Number
<i>Communications Server for z/OS V1R8 TCP/IP Implementation, Volume 1: Base Functions, Connectivity, and Routing</i>	SG24-7339
<i>Communications Server for z/OS V1R8 TCP/IP Implementation, Volume 2: Standard Applications</i>	SG24-7340
<i>Communications Server for z/OS V1R8 TCP/IP Implementation, Volume 3: High Availability, Scalability, and Performance</i>	SG24-7341
<i>Communications Server for z/OS V1R8 TCP/IP Implementation, Volume 4: Policy-Based Network Security</i>	SG24-7342
<i>IBM Communication Controller Migration Guide</i>	SG24-6298
<i>IP Network Design Guide</i>	SG24-2580
<i>Managing OS/390® TCP/IP with SNMP</i>	SG24-5866

Title	Number
<i>Migrating Subarea Networks to an IP Infrastructure Using Enterprise Extender</i>	SG24-5957
<i>SecureWay Communications Server for OS/390 V2R8 TCP/IP: Guide to Enhancements</i>	SG24-5631
<i>SNA and TCP/IP Integration</i>	SG24-5291
<i>TCP/IP in a Sysplex</i>	SG24-5235
<i>TCP/IP Tutorial and Technical Overview</i>	GG24-3376
<i>Threadsafe Considerations for CICS</i>	SG24-6351

Where to find related information on the Internet

z/OS

This site provides information about z/OS Communications Server release availability, migration information, downloads, and links to information about z/OS technology

<http://www.ibm.com/servers/eserver/zseries/zos/>

z/OS Internet Library

Use this site to view and download z/OS Communications Server documentation

<http://www.ibm.com/servers/eserver/zseries/zos/bkserv/>

IBM Communications Server product

The primary home page for information about z/OS Communications Server

<http://www.software.ibm.com/network/commserver/>

IBM Communications Server product support

Use this site to submit and track problems and search the z/OS Communications Server knowledge base for Technotes, FAQs, white papers, and other z/OS Communications Server information

<http://www.software.ibm.com/network/commserver/support/>

IBM Systems Center publications

Use this site to view and order Redbooks, Redpapers, and Technotes

<http://www.redbooks.ibm.com/>

IBM Systems Center flashes

Search the Technical Sales Library for Techdocs (including Flashes, presentations, Technotes, FAQs, white papers, Customer Support Plans, and Skills Transfer information)

<http://www.ibm.com/support/techdocs/atmastr.nsf>

RFCs

Search for and view Request for Comments documents in this section of the Internet Engineering Task Force Web site, with links to the RFC repository and the IETF Working Groups Web page

<http://www.ietf.org/rfc.html>

Internet drafts

View Internet-Drafts, which are working documents of the Internet Engineering Task Force (IETF) and other groups, in this section of the Internet Engineering Task Force Web site

<http://www.ietf.org/ID.html>

Information about Web addresses can also be found in information APAR II11334.

Note: Any pointers in this publication to Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

DNS Web sites

For more information about DNS, see the following USENET news groups and mailing addresses:

USENET news groups

`comp.protocols.dns.bind`

BIND mailing lists

<http://www.isc.org/ml-archives/>

BIND Users

- Subscribe by sending mail to `bind-users-request@isc.org`.
- Submit questions or answers to this forum by sending mail to `bind-users@isc.org`.

BIND 9 Users (This list might not be maintained indefinitely.)

- Subscribe by sending mail to `bind9-users-request@isc.org`.
- Submit questions or answers to this forum by sending mail to `bind9-users@isc.org`.

Using LookAt to look up message explanations

LookAt is an online facility that lets you look up explanations for most of the IBM messages you encounter, as well as for some system abends and codes. Using LookAt to find information is faster than a conventional search because in most cases LookAt goes directly to the message explanation.

You can use LookAt from these locations to find IBM message explanations for z/OS elements and features, z/VM[®], VSE/ESA[™], and Clusters for AIX[®] and Linux[™]:

- The Internet. You can access IBM message explanations directly from the LookAt Web site at www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/.
- Your z/OS TSO/E host system. You can install code on your z/OS systems to access IBM message explanations using LookAt from a TSO/E command line (for example: TSO/E prompt, ISPF, or z/OS UNIX System Services).
- Your Microsoft[®] Windows[®] workstation. You can install LookAt directly from the z/OS Collection (SK3T-4269) or the *z/OS and Software Products DVD Collection* (SK3T-4271) and use it from the resulting Windows graphical user interface (GUI). The command prompt (also known as the DOS > command line) version can still be used from the directory in which you install the Windows version of LookAt.
- Your wireless handheld device. You can use the LookAt Mobile Edition from www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookatm.html with a

handheld device that has wireless access and an Internet browser (for example: Internet Explorer for Pocket PCs, Blazer or Eudora for Palm OS, or Opera for Linux handheld devices).

You can obtain code to install LookAt on your host system or Microsoft Windows workstation from:

- A CD-ROM in the z/OS Collection (SK3T-4269).
- The *z/OS and Software Products DVD Collection* (SK3T-4271).
- The LookAt Web site (click **Download** and then select the platform, release, collection, and location that suit your needs). More information is available in the LOOKAT.ME files available during the download process.

Using IBM Health Checker for z/OS

IBM Health Checker for z/OS is a z/OS component that installations can use to gather information about their system environment and system parameters to help identify potential configuration problems before they impact availability or cause outages. Individual products, z/OS components, or ISV software can provide checks that take advantage of the IBM Health Checker for z/OS framework. This book might refer to checks or messages associated with this component.

For additional information about checks and about IBM Health Checker for z/OS, see *IBM Health Checker for z/OS: User's Guide*. Starting with z/OS V1R4, z/OS users can obtain the IBM Health Checker for z/OS from the z/OS Downloads page at <http://www.ibm.com/servers/eservers/zseries/zos/downloads/>.

SDSF also provides functions to simplify the management of checks. See *z/OS SDSF Operation and Customization* for additional information.

How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this document or any other z/OS Communications Server documentation:

- Go to the z/OS contact page at:
<http://www.ibm.com/servers/eserver/zseries/zos/webqs.html>
There you will find the feedback page where you can enter and submit your comments.
- Send your comments by e-mail to comsvrcf@us.ibm.com. Be sure to include the name of the document, the part number of the document, the version of z/OS Communications Server, and, if applicable, the specific location of the text you are commenting on (for example, a section number, a page number or a table number).

Summary of changes

Summary of changes for SC31-8787-09 z/OS Version 1 Release 9

This document contains information previously presented in SC31-8787-08, which support z/OS Version 1 Release 8.

The information in this document includes descriptions of support for both IPv4 and IPv6 networking protocols. Unless explicitly noted, descriptions of IP protocol support concern IPv4. IPv6 support is qualified within the text.

This document refers to Communications Server data sets by their default SMP/E distribution library name. Your installation might, however, have different names for these data sets where allowed by SMP/E, your installation personnel, or administration staff. For instance, this document refers to samples in SEZAINST library as simply in SEZAINST. Your installation might choose a data set name of SYS1.SEZAINST, CS390.SEZAINST or other high-level qualifiers for the data set name.

New information

- SNMP manager API, see Chapter 5, “SNMP manager API,” on page 131.
- IPSec network management interface support and network security services, see “Local IPSec NMI” on page 472.
- FTP Kerberos single sign on support, see Table 67 on page 549 and Table 70 on page 552.
- Enable application identifier in NMI, SMF, and Netstat, see “SIOCSAPPLDATA IOCTL” on page 643.

Changed information

- Enable AT-TLS for FTP client and server, see “Real-time SMF data NMI (SYSTCPSM) record formats” on page 545.
- Enable AT-TLS for the TN3270E Telnet server, see “Using the SIOCTTLCTL ioctl” on page 619.

Deleted information

- The TN3270E Telnet server running in the TCPIP address space is removed.
- The APPC Application Suite is removed from the z/OS V1R9 Communications Server product and therefore documentation describing APPC Application Suite support has been deleted.

This document contains terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

This document has been enabled for the following types of advanced searches in the online z/OS Library Center: commands.

You might notice changes in the style and structure of some content in this document—for example, headings that use uppercase for the first letter of initial

words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our documents.

Summary of changes for SC31-8787- 08 z/OS Version 1 Release 8

This document contains information previously presented in SC31-8787-07, which support z/OS Version 1 Release 7.

The information in this document includes descriptions of support for both IPv4 and IPv6 networking protocols. Unless explicitly noted, descriptions of IP protocol support concern IPv4. IPv6 support is qualified within the text.

This document refers to Communications Server data sets by their default SMP/E distribution library name. Your installation might, however, have different names for these data sets where allowed by SMP/E, your installation personnel, or administration staff. For instance, this document refers to samples in SEZAINST library as simply in SEZAINST. Your installation might choose a data set name of SYS1.SEZAINST, CS390.SEZAINST or other high-level qualifiers for the data set name.

New information

- Support for RPC Binding Protocol 3 and 4. See Chapter 8, “Remote procedure calls in the z/OS Communications Server environment,” on page 207.
- FTP Client API - add REXX™ support, see “FTP Client Application Interface (FCAI) stem variables” on page 407 and “FTP Client API REXX function” on page 433.
- Network management interface support for dropping socket endpoints, see “TCP/IP callable NMI (EZBNMIFR)” on page 583.
- Network management interface for TN3270 performance data, see “TCP/IP callable NMI (EZBNMIFR)” on page 583.
- Allow wildcards on Enterprise Extender network management interface filters, see “SNA network management interface (NMI) request/response format” on page 602.
- FTP network management interface enhancements, see “Real-time SMF data NMI (SYSTCPSM) record formats” on page 545.
- Sysplex partitioning, see Appendix A, “TCP/IP in the sysplex,” on page 647.
- Interfacing with the Digital Certificate Access Server (DCAS), see Chapter 16, “Interfacing with the Digital Certificate Access Server (DCAS),” on page 637.

Changed information

- TN3270 SMF record enhancement, see “TCP/IP callable NMI (EZBNMIFR)” on page 583.

This document contains terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

This document has been enabled for the following types of advanced searches in the online z/OS Library Center: commands.

You might notice changes in the style and structure of some content in this document—for example, headings that use uppercase for the first letter of initial words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our documents.

**Summary of changes
for SC31-8787- 07
z/OS Version 1 Release 7**

This document contains information previously presented in SC31-8787-05 and SC31-8787-06, which support z/OS Version 1 Release 6.

The information in this document includes descriptions of support for both IPv4 and IPv6 networking protocols. Unless explicitly noted, descriptions of IP protocol support concern IPv4. IPv6 support is qualified within the text.

This document refers to Communications Server data sets by their default SMP/E distribution library name. Your installation might, however, have different names for these data sets where allowed by SMP/E, your installation personnel, or administration staff. For instance, this document refers to samples in SEZAINST library as simply in SEZAINST. Your installation might choose a data set name of SYS1.SEZAINST, CS390.SEZAINST or other high-level qualifiers for the data set name.

New information

- Chapter 15, “Application Transparent Transport Layer Security (AT-TLS),” on page 617 - AT-TLS creates a secure session at the TCP/IP layer on behalf of an application.
- “FTP Client API C functions” on page 430.
- QDIO OSA Express Segmentation Offload, see “Processing the CTE records for SYSTCPDA” on page 540.

Changed information

- The title of this publication, previously *z/OS Communications Server: IP Programmer's Reference*, has changed.

Deleted information

- References to OROUTED.

This document contains terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

This document has been enabled for the following types of advanced searches in the online z/OS Library Center: commands.

You might notice changes in the style and structure of some content in this document—for example, headings that use uppercase for the first letter of initial words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our documents.

Chapter 1. General programming information

The information presented in this reference applies only to IPv4, AF_INET sockets unless specified as IPv6.

For the fundamental technical information you need to know before you attempt to work with the application programming interfaces (APIs) that are provided with TCP/IP, see the *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*.

The modules generated by the new compiler are similar to those produced by the AD/Cycle® compiler.

Overview of Distributed Protocol Interface (DPI) versions 1.1 and 2.0

Two levels of Distributed Protocol Interface (DPI) are supported by z/OS Communications Server. The following shows some support differences between the two versions:

- Support provided by DPI Version 1.1
 - Was supported on earlier releases of TCP/IP and continues to be supported by the SNMP agent; existing subagents written with DPI Version 1.1 still run with no changes required.
 - Supports SNMP Version 1 protocols, but not SNMP Version 2.
 - Is intended for C socket API users, not z/OS UNIX C socket users.
 - Supports connections from subagents using TCP sockets.
 - Is documented in RFC 1228.
- Support provided by DPI Version 2.0:
 - Is supported in TCP/IP z/OS UNIX and above.
 - Contains more functions that make writing a subagent easier.
 - Supports both SNMP Version 1 and Version 2 protocols.
 - Is used by z/OS UNIX C socket users but not standard C socket users.
 - Supports connections from subagents using TCP sockets and UNIX Stream sockets.
 - Is documented in RFC 1592.

While DPI Version 1.1 can continue to be used by existing subagents, IBM recommends that users who are writing new subagents or modifying old ones consider upgrading to DPI Version 2.0 to take advantage of the SNMP Version 2 protocols and the greater functionality of DPI Version 2.0.

Although the SNMP agent shipped with z/OS Communications Server is now enabled to support SNMP Version 3 (SNMPv3), no changes are required to subagents written with either DPI Version 1.1 or Version 2.0. SNMPv3 did not introduce any new protocol data unit (PDU) types. Support for the SNMPv3 framework is handled by the SNMP agent.

Users of DPI Version 1.1 must compile using the DPI library routines provided in SEZADPIL and the version of the header file, `snmp_dpi.h`, provided in SEZACMAC. See Chapter 2, “SNMP agent Distributed Protocol Interface version 1.1,” on page 3 for additional details.

Users of DPI Version 2.0 must compile using the DPI library routines provided in the directory `/usr/lpp/tcpip/snmp/build/libdpi20` and the DPI Version 2.0 copy of the header file, `snmp_dpi.h` in `/usr/lpp/tcpip/snmp/include`. Additional details are in Chapter 3, “SNMP agent Distributed Protocol Interface version 2.0,” on page 41.

For information about migrating an existing subagent from DPI Version 1.1 to DPI Version 2.0, see “Migrating your SNMP DPI subagent to Version 2.0” on page 45.

Chapter 2. SNMP agent Distributed Protocol Interface version 1.1

The simple network management protocol (SNMP) agent Distributed Protocol Interface (DPI) permits you to dynamically add, delete, or replace management variables in the local management information base (MIB) without recompiling the SNMP agent. The DPI protocol is also supported by SNMP agents on other IBM platforms. This makes it easier to port subagents between those platforms.

For more information about the DPI interface, see RFC 1228. See Appendix E, "Related protocol specifications," on page 719 for information about accessing RFCs.

SNMP agents and subagents

To allow the subagents to perform their functions, the SNMP agent binds to an arbitrarily chosen TCP port and listens for connection requests from subagents. A well-known port is not used. Every invocation of the SNMP agent potentially results in a different TCP port being used.

Agents, or SNMP servers, are responsible for performing the network management functions requested by the network management stations.

A subagent provides an extension to the functionality provided by the SNMP agent. The subagent allows you to define your own MIB variables, which are useful in your environment, and register them with the SNMP agent. When requests for these variables are received by the SNMP agent, the agent passes the request to the subagent and returns a response to the agent. The SNMP agent creates an SNMP response packet and sends the response to the remote network management station that initiated the request. The existence of the subagent is transparent to the network management station.

A subagent of the SNMP agent determines the port number by sending a GET request for an MIB variable, which represents the value of the TCP port. The subagent is not required to create and parse SNMP packets, because the DPI application programming interface (API) has a library routine `query_DPI_port()`. After the subagent obtains the value of the DPI TCP port, it should make a TCP connection to the appropriate port. After a successful `socket connect()` call, the subagent registers the set of variables it supports with the SNMP agent. For information about the `connect()` call, see the `connect()` information in the *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*. When all variable classes are registered, the subagent waits for requests from the SNMP agent.

If connections to the SNMP agent are restricted by the security product, then the security product user ID associated with the subagent must be permitted to the agent's security product resource name for the connection to be accepted. See the Simple Network Management Protocol (SNMP) information in the *z/OS Communications Server: IP Configuration Guide* for more information about security product access between subagents and the z/OS Communications Server SNMP agent.

Processing DPI requests

The SNMP agent can initiate three DPI requests: GET, SET, and GET-NEXT. These requests correspond to the three SNMP requests that a network management station can make. The subagent responds to a request with a response packet. The response packet can be created using the `mkDPIresponse()` library routine, which is part of the DPI API library.

The SNMP subagent can only initiate two requests: REGISTER and TRAP. A REGISTER request indicates to the SNMP agent which MIB variables are supported by the subagent. A TRAP request notifies the SNMP agent of an asynchronous event that should be sent to network management stations.

Processing a GET request

The DPI packet is parsed to get the object ID of the requested variable. If the specified object ID of the requested variable is not supported by the subagent, the subagent returns an error indication of `SNMP_NO_SUCH_NAME`. Name, type, or value information is not returned. For example:

```
unsigned char *cp;

cp = mkDPIresponse(SNMP_NO_SUCH_NAME,0);
```

If the object ID of the variable is supported, an error is not returned and the name, type, and value of the object ID are returned using the `mkDPIset()` and `mkDPIresponse()` routines. The following is an example of an object ID, whose type is string, being returned.

```
char *obj_id;

unsigned char *cp;
struct dpi_set_packet *ret_value;
char *data;

data = "a string to be returned";
ret_value = mkDPIset(obj_id,SNMP_TYPE_STRING,
                    strlen(data)+1,data);
cp = mkDPIresponse(0,ret_value);
```

Processing a SET request

Processing a SET request is similar to processing a GET request, but the SNMP agent passes additional information to the subagent. This additional information consists of the type, length, and value to be set.

If the object ID of the variable is not supported, the subagent returns an error indication of `SNMP_NO_SUCH_NAME`. If the object ID of the variable is supported, but cannot be set, an error indication of `SNMP_READ_ONLY` is returned. If the object ID of the variable is supported, and is successfully set, the message `SNMP_NO_ERROR` is returned.

Processing a GET-NEXT request

Parsing a GET-NEXT request yields two parameters: the object ID of the requested variable and the reason for this request. This allows the subagent to return the name, type, and value of the next supported variable, whose name lexicographically follows that of the passed object ID.

Subagents can support several different groups of the MIB tree. However, the subagent cannot jump from one group to another. You must determine the reason

for the request to then determine the path to traverse in the MIB tree. The second parameter contains this reason and is the group prefix of the MIB tree that is supported by the subagent.

If the object ID of the next variable supported by the subagent does not match this group prefix, the subagent must return `SNMP_NO_SUCH_NAME`. If required, the SNMP agent calls on the subagent again and passes a different group prefix.

For example, if you have two subagents, the first subagent registers two group prefixes, A and C, and supports variables A.1, A.2, and C.1. The second subagent registers the group prefix B, and supports variable B.1.

When a remote management station begins dumping the MIB, starting from A, the following sequence of queries is performed:

Subagent 1 gets called:

```
get-next(A,A) == A.1
get-next(A.1,A) == A.2
get-next(A.2,A) == error(no such name)
```

Subagent 2 is then called:

```
get-next(A.2,B) == B.1
get-next(B.1,B) == error(no such name)
```

Subagent 1 is then called:

```
get-next(B.1,C) == C.1
get-next(C.1,C) == error(no such name)
```

Processing a REGISTER request

A subagent must register the variables that it supports with the SNMP agent. Packets can be created using the `mkDPiRegister()` routine.

For example:

```
unsigned char *cp;

cp = mkDPiRegister("1.3.6.1.2.1.1.2.");
```

Note: Object IDs are registered with a trailing period (.).

Processing a TRAP request

A subagent can request that the SNMP agent generate a TRAP. The subagent must provide the desired values for the generic and specific parameters of the TRAP. The subagent can optionally provide a name, type, and value parameter. The DPI API library routine `mkDPiTrap()` can be used to generate the TRAP packet.

SNMP agent DPI header files

The following header is required to run SNMP DPI applications:

```
snmp_dpi.h
```

Compiling and linking

You can use several methods to compile, link-edit, and execute your TCP/IP C source program in MVS. This section contains information about the data sets that you must include to run your C source program under MVS batch, using IBM-supplied cataloged procedures.

The following list contains partitioned data set names, which are used as examples in the following JCL statements:

USER.MYPROG.C

Contains user C source programs

USER.MYPROG.C(PROGRAM1)

Member PROGRAM1 in USER.MYPROG.C partitioned data set

USER.MYPROG.H

Contains user #include data sets

USER.MYPROG.OBJ

Contains object code for the compiled versions of user C programs in USER.MYPROG.C

USER.MYPROG.LOAD

Contains link-edited versions of user programs in USER.MYPROG.OBJ

Sample compile cataloged procedure additions

Include the following in the compile step of your cataloged procedure. Cataloged procedures are included in the IBM-supplied samples for your MVS system.

- Add the following statement as the first //SYSLIB DD statement:

```
//SYSLIB DD DSN=SEZACMAC,DISP=SHR
```

- Add the following //USERLIB DD statement:

```
//USERLIB DD DSN=USER.MYPROG.H,DISP=SHR
```

Sample link-edit cataloged procedure additions

Include the following in the link-edit step of your cataloged procedure.

Add the following statements after the //SYSLIB DD statement:

```
// DD DSN=SEZACMTX,DISP=SHR
// DD DSN=SEZADPIL,DISP=SHR
```

Note: For more information about compiling and linking, see the *z/OS XL C/C++ User's Guide*.

SNMP DPI library routines

This section provides the syntax, parameters, and other appropriate information for each DPI routine supported by TCP/IP.

mkDPIlist()

```
#include <snmp_dpi.h>
#include <types.h>

struct dpi_set_packet *mkDPIlist(packet, oid_name, type, len, value)
struct dpi_set_packet *packet;
char *oid_name;
int type;
int len;
char *value;
```

Parameters

<i>packet</i>	A pointer to a structure dpi_set_packet, or NULL
<i>oid_name</i>	The object identifier of the variable

<i>type</i>	The type of the value
<i>len</i>	The length of the value
<i>value</i>	A pointer to the value

Description: The `mkDPIlist()` routine can be used to create the portion of the parse tree that represents a list of name and value pairs. Each entry in the list represents a name and value pair (as would normally be returned in a response packet). If the pointer *packet* is NULL, a new `dpi_set_packet` structure is dynamically allocated and the pointer to that structure is returned. The structure will contain the new name and value pair. If the pointer *packet* is not NULL, a new `dpi_set_packet` structure is dynamically allocated and chained to the list. The new structure will contain the new name and value pair. The pointer *packet* will be returned to the caller. If an error is detected, a NULL pointer is returned.

The value of *type* can be the same as for `mkDPIset()`. These are defined in the `snmp_dpi.h` header file.

The `dpi_set_packet` structure has a next pointer [0 in case of a `mkDPIset()` call and is also 0 upon the first `mkDPIlist()` call]. The structure looks like this:

```
struct dpi_set_packet {
    char          *object_id;
    unsigned char  type;
    unsigned short value_len;
    char          *value;
    struct dpi_set_packet *next;
};
```

fDPIparse()

```
#include <snmp_dpi.h>
#include <bsdtypes.h>

void fDPIparse(hdr)
struct snmp_dpi_hdr *hdr;
```

Parameters

hdr Specifies a parse tree.

Description: The `fDPIparse()` routine frees a parse tree that was previously created by a call to `pDPIpacket()`. After calling `fDPIparse()`, you cannot make additional references to the parse tree.

Return Values: None.

mkDPIregister()

```
#include <snmp_dpi.h>
#include <bsdtypes.h>

unsigned char *mkDPIregister(oid_name)
char *oid_name;
```

Parameters

oid_name Specifies the object identifier of the variable to be registered. Object identifiers are registered with a trailing period (.).

Description: The `mkDPIregister()` routine creates a register request packet and returns a pointer to a static buffer, which holds the packet contents. The length of the remaining packet is stored in the first 2 bytes of the packet.

Return Values: If successful, returns a pointer to a static buffer containing the packet contents. A NULL pointer is returned if an error is detected during the creation of the packet.

Example: The following is an example of the `mkDPIregister()` call.

```
unsigned char *packet;
int len;

packet = mkDPIregister("1.3.6.1.2.1.1.1.");

len = *packet * 256 + *(packet + 1);
```

mkDPIresponse()

```
#include <snmp_dpi.h>
#include <bsdtypes.h>

unsigned char *mkDPIresponse(ret_code, value_list)
int ret_code;
struct dpi_set_packet *value_list;
```

Parameters

<i>ret_code</i>	Specifies the error code to be returned.
<i>value_list</i>	Indicates a pointer to a parse tree containing the name, type, and value information to be returned.

Description: The `mkDPIresponse()` routine creates a response packet. The first parameter, *ret_code*, is the error code to be returned. Zero indicates no errors. Possible errors include the following:

- `SNMP_BAD_VALUE`
- `SNMP_GEN_ERR`
- `SNMP_NO_ERROR`
- `SNMP_NO_SUCH_NAME`
- `SNMP_READ_ONLY`
- `SNMP_TOO_BIG`

See the `snmp_dpi.h` header file for a description of these messages.

If *ret_code* does not indicate an error, the second parameter is a pointer to a parse tree created by `mkDPIset()`, which represents the name, type, and value of the information being returned. If an error is indicated, the second parameter is passed as a NULL pointer.

The length of the remaining packet is stored in the first 2 bytes of the packet.

Note: `mkDPIresponse()` always frees the passed parse tree.

Return Values: If successful, `mkDPIresponse()` returns a pointer to a static buffer containing the packet contents. This is the same buffer used by `mkDPIregister()`. A NULL pointer is returned if an error is detected during the creation of the packet.

Example: The following is an example of the `mkDPISet()` call.

```
unsigned char *packet;

int error_code;
struct dpi_set_packet *ret_value;

packet = mkDPISet(error_code, ret_value);

len = *packet * 256 + *(packet + 1);
```

mkDPISet()

```
#include <snmp_dpi.h>
#include <bsdtypes.h>

struct dpi_set_packet *mkDPISet(oid_name, type, len, value)
char *oid_name;
int type;
int len;
char *value;
```

Parameters

<i>oid_name</i>	Specifies the object identifier of the variable.
<i>type</i>	Specifies the type of the object identifier.
<i>len</i>	Indicates the length of the value.
<i>value</i>	Indicates the pointer to the first byte of the value of the object identifier.

Description: The `mkDPISet()` routine can be used to create the portion of a parse tree that represents a name and value pair (as would normally be returned in a response packet). It returns a pointer to a dynamically allocated parse tree representing the name, type, and value information. If an error is detected while creating the parse tree, a NULL pointer is returned.

The value of *type* can be one of the following, which are defined in the `snmp_dpi.h` header file:

- `SNMP_TYPE_COUNTER`
- `SNMP_TYPE_GAUGE`
- `SNMP_TYPE_INTERNET`
- `SNMP_TYPE_NUMBER`
- `SNMP_TYPE_OBJECT`
- `SNMP_TYPE_STRING`
- `SNMP_TYPE_TICKS`

The *value* parameter is always a pointer to the first byte of the object ID value.

Note: The parse tree is dynamically allocated, and copies are made of the passed parameters. After a successful call to `mkDPISet()`, the application can dispose of the passed parameters without affecting the contents of the parse tree.

Return Values: Returns a pointer to a parse tree containing the name, type, and value information.

mkDPITrap()

```
#include <snmp_dpi.h>
#include <bsdtypes.h>

unsigned char *mkDPITrap(generic, specific, value_list)
int generic;
int specific;
struct dpi_set_packet *value_list;
```

Parameters

<i>generic</i>	Specifies the generic field in the SNMP TRAP packet.
<i>specific</i>	Specifies the specific field in the SNMP TRAP packet.
<i>value_list</i>	Used to pass the name and value pair to be placed into the SNMP packet.

Description: The mkDPITrap() routine creates a TRAP request packet. The information contained in *value_list* is passed as the set_packet portion of the parse tree.

The length of the remaining packet is stored in the first 2 bytes of the packet.

Note: mkDPITrap() always frees the passed parse tree.

Return Values: If the packet can be created, a pointer to a static buffer containing the packet contents is returned. This is the same buffer that is used by mkDPIDregister(). If an error is encountered while creating the packet, a NULL pointer is returned.

Example: The following is an example of the mkDPITrap() call.

```
struct dpi_set_packet *if_index_value;
unsigned long data;
unsigned char *packet;
int len;

if_index_value = mkDPISet("1.3.6.1.2.1.2.2.1.1", SNMP_TYPE_NUMBER,
    sizeof(unsigned long), &data);
packet = mkDPITrap(2, 0, if_index_value);
len = *packet * 256 + *(packet + 1);
write(fd, packet, len);
```

mkDPITrape()

```
#include <snmp_dpi.h>
#include <types.h>

unsigned char *mkDPITrape(generic, specific, value_list, enterprise_oid)
long int generic; /* 4 octet integer */
long int specific;
struct dpi_set_packet *value_list;
char *enterprise_oid;
```

Parameters

<i>generic</i>	The generic field for the SNMP TRAP packet.
<i>specific</i>	The specific field for the SNMP TRAP packet.

<i>value_list</i>	A pointer to a structure <code>dpi_set_packet</code> , which contains one or more variables to be sent with the SNMP TRAP packet. Or NULL if no variables are to be sent.
<i>enterprise_oid</i>	A pointer to a character string representing the enterprise object ID (in ASN.1 notation, for example, 1.3.6.1.4.1.2.2.1.4). Or NULL if you want the SNMP agent to use its own enterprise object ID.

Description: The `mkDPITrap()` routine can be used to create an *extended* trap. It is basically the same as the `mkDPITrap()` routine, but allows you to pass a list of variables, and also an enterprise object ID.

pDPIpacket()

```
#include <snmp_dpi.h>
#include <bsdtypes.h>

struct snmp_dpi_hdr *pDPIpacket(packet)
unsigned char *packet;
```

Parameters

packet Specifies the DPI packet to be parsed.

Description: The `pDPIpacket()` routine parses a DPI packet and returns a parse tree representing its contents. The parse tree is dynamically allocated and contains copies of the information within the DPI packet. After a successful call to `pDPIpacket()`, the packet can be disposed of in any manner the application chooses, without affecting the contents of the parse tree.

Return Values: If `pDPIpacket()` is successful, a parse tree is returned. If an error is encountered during the parse, a NULL pointer is returned.

Note: The parse tree structures are defined in the `snmp_dpi.h` header file.

Example: The following is an example of the `mkDPIpacket()` call.

The root of the parse tree is represented by an `snmp_dpi_hdr` structure.

```
struct snmp_dpi_hdr {
    unsigned char proto_major;
    unsigned char proto_minor;
    unsigned char proto_release;

    unsigned char packet_type;
    union {
        struct dpi_get_packet      *dpi_get;
        struct dpi_next_packet    *dpi_next;
        struct dpi_set_packet      *dpi_set;
        struct dpi_resp_packet     *dpi_response;
        struct dpi_trap_packet     *dpi_trap;
    } packet_body;
};
```

The `packet_type` field can have one of the following values, which are defined in the `snmp_dpi.h` header file:

- `SNMP_DPI_GET`
- `SNMP_DPI_GET_NEXT`
- `SNMP_DPI_SET`

The `packet_type` field indicates the request that is made of the DPI client. For each of these requests, the remainder of the `packet_body` is different. If a GET request is indicated, the object ID of the desired variable is passed in a `dpi_get_packet` structure.

```
struct dpi_get_packet {
    char *object_id;
};
```

A GET-NEXT request is similar, but the `dpi_next_packet` structure also contains the object ID prefix of the group that is currently being traversed.

```
struct dpi_next_packet {
    char *object_id;
    char *group_id;
};
```

If the next object, whose object ID lexicographically follows the object ID indicated by *object_id*, does not begin with the suffix indicated by the *group_id*, the DPI client must return an error indication of `SNMP_NO_SUCH_NAME`.

A SET request has the most data associated with it, and this is contained in a `dpi_set_packet` structure.

```
struct dpi_set_packet {
    char            *object_id;
    unsigned char    type;
    unsigned short   value_len;
    char            *value;
    struct dpi_set_packet *next;
};
```

The object ID of the variable to be modified is indicated by *object_id*. The type of the variable is provided in *type* and can have one of the following values:

- `SNMP_TYPE_COUNTER`
- `SNMP_TYPE_EMPTY`
- `SNMP_TYPE_GAUGE`
- `SNMP_TYPE_INTERNET`
- `SNMP_TYPE_NUMBER`
- `SNMP_TYPE_OBJECT`
- `SNMP_TYPE_STRING`
- `SNMP_TYPE_TICKS`

The length of the value to be set is stored in *value_len* and *value* contains a pointer to the value.

Note: The storage pointed to by *value* is reclaimed when the parse tree is freed. The DPI client must make provision for copying the value contents.

query_DPI_port()

```
#include <snmp_dpi.h>
#include <bsdtypes.h>

int query_DPI_port (host_name, community_name)
char *host_name;
char *community_name;
```

Parameters

host_name Specifies a pointer to the SNMP agent host name or internet address.

community_name Specifies a pointer to the community name to be used when making a request. The *community_name* constant must be specified in ASCII.

Description: The `query_DPI_port()` routine is used by a DPI client to determine the TCP port number that is associated with the DPI. This port number is needed to connect() to the SNMP agent. The port number is obtained through an SNMP GET request.

Return Values: An integer representing the TCP port number is returned if successful; a -1 is returned if the port cannot be determined.

Sample SNMP DPI client program for C sockets for version 1.1

This section contains an example of an SNMP DPI client program. The DPISAMPL program can be run using the SNMP agents that support the SNMP-DPI interface as described in RFC 1228. See Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs.

It can be used to test agent DPI implementations because it provides variables of all types and allows you to generate traps of all types.

DPISAMPL implements a set of variables in the `dpiSample` table, which consists of a set of objects in the IBM Research tree (1.3.6.1.2.2.1.4). See “`dpiSample` table MIB descriptions” on page 16 for the objectID and type of each object.

Using the DPISAMPL program

The DPISAMPL program accepts the following arguments:

- ? Explains the usage.
- d *n* Sets the debug at level *n*. The range is from 0 (for no messages) to 4 (for the most verbose). The default is 0. If a number greater than 4 is specified, tracing is set to level 4.
- trap *gtype stype data* Generates a trap of the generic type *gtype*, of the specific type *stype*, and pass *data* as an additional value for the variable `dpiSample.stype.0`. The values for *gtype* are from 0 through 5. The values for *stype* indicate how *data* is interpreted. The following values are valid for *stype*:
 - 1 number
 - 2 octet string
 - 3 object ID
 - 4 empty (ignored)
 - 5 internet address
 - 6 counter
 - 7 gauge
 - 8 time ticks

	9	display string
	10	octet string
-std_traps	Generates or simulates the standard SNMP traps, which are the generic types 0 through 5. This includes a link down trap.	
-ent_traps	Generates extended enterprise-specific traps, which are specific types 1 through 9, using the internal dpiSample variables.	
ent_trapse	Generates extended enterprise-specific traps, which are specific types 11 through 19.	
-all_traps	Generates std_traps, ent_traps, and ent_trapse.	
-iucv	Uses an AF_IUCV socket to connect to the SNMP agent. This is the default.	

Note: Although the IUCV API is no longer supported, use of the IUCV interaddress space communication mechanism is supported.

-u agent_userid	Specifies the user ID where the SNMP agent is running. The default is SNMPD.
-inet	Uses an AF_INET socket to connect to the SNMP agent.
agent_hostname	Specifies the host name of the system where an SNMP DPI-capable agent is running. The default is localhost.

Note: The localhost value is not defined by default on z/OS. Ensure localhost is defined to the name server or in the host name resolution file as the local IP address if the *agent_hostname* parameter is not explicitly specified.

community_name	Specifies the community name, which is required to get the dpiPort. The default is public.

DPISAMPN NCCFLST for the SNMP manager

The DPISAMPN NCCFLST allows you to exercise the DPISAMPL subagent from a Tivoli® NetView® SNMP management station. The DPISAMPL subagent must be running. This sample allows you to specify which test function you want to run.

You can specify the following on Tivoli NetView:

agent_host name	Specifies the host name or IP address of the system where the SNMP agent is running.
community_name	Specifies the community name. The CLIST makes the community name uppercases so the SNMP agent must be configured to accept the community name in uppercase.
function	Specifies the test function to be performed. Valid test functions are: ALL Runs all of the tests. This is the default. GET Retrieves the dpiSample variables one at a time.

GETNEXT

Retrieves all the dpiSample variables.

ONEGET

Retrieves all the dpiSample variables with one GET.

ONESET

Sets all the dpiSample variables at once.

QUIT Causes the DPISAMPLE subagent to terminate.

SET Sets the dpiSample variables one at a time with one SET.

TRAPS

Instructs the DPISAMPLE subagent to generate nine enterprise-specific traps.

The NCCFLST assumes that the definitions for the dpiSample table (see “dpiSample table MIB descriptions” on page 16) have been added to the MIBDESC.DATA file. You can also GET, GETNEXT, or SET dpiSample variables with regular SNMP GET/GETNEXT/SET commands.

The DPISAMPL subagent recognizes a few special values in the variable dpiSampleCommand. The following are the special values and their associated subagent actions.

all_traps

Generates std_traps, ent_traps, and ent_trapse.

ent_traps

Generates extended enterprise-specific traps, which are specific types 1 through 9, using the internal dpiSample variables.

ent_trapse

Generates extended enterprise-specific traps, which are specific types 11 through 19.

quit Causes the subagent to terminate.

std_traps

Generates or simulates the standard SNMP traps, which are the generic types 0 through 5. This includes a link down trap.

Compiling and linking the DPISAMPL.C source code

The source code for the sample DPI program can be found in the SEZAINST data set, member DPISAMPL.

You can specify the following compile time flags:

_NO_PROTO

The DPISAMPL.C code assumes that it is compiled with an ANSI-C compliant compiler. It can be compiled without ANSI-C by defining this flag.

MVS Indicates that compilation is for MVS and uses MVS-specific includes. Some MVS/VM-specific code is compiled.

When linking the DPISAMPL code, you must use the SEZADPIL data set. It contains the SNMP-DPI interface routines as described in RFC 1228. See Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs.

dpiSample table MIB descriptions

The following shows the MIB descriptions for the dpiSample table.

```
# DPISAMPLE.C supports these variables as an SNMP DPI sample sub-agent
# it also generates enterprise specific traps via DPI with these objects
dpiSample          1.3.6.1.4.1.2.2.1.4.    table          0
dpiSampleNumber    1.3.6.1.4.1.2.2.1.4.1.    number          10
# next one is to be able to send a badValue with a SET request
dpiSampleNumberString 1.3.6.1.4.1.2.2.1.4.1.1. string        10
dpiSampleOctetString  1.3.6.1.4.1.2.2.1.4.2.    string          10
dpiSampleObjectID     1.3.6.1.4.1.2.2.1.4.3.    object          10
# XGMON/SQESERV does not allow to specify empty (so use empty string)
dpiSampleEmpty        1.3.6.1.4.1.2.2.1.4.4.    string          10
dpiSampleInetAddress  1.3.6.1.4.1.2.2.1.4.5.    internet        10
dpiSampleCounter      1.3.6.1.4.1.2.2.1.4.6.    counter         10
dpiSampleGauge        1.3.6.1.4.1.2.2.1.4.7.    gauge           10
dpiSampleTimeTicks    1.3.6.1.4.1.2.2.1.4.8.    ticks           10
dpiSampleDisplayString 1.3.6.1.4.1.2.2.1.4.9.    display         10
dpiSampleCommand      1.3.6.1.4.1.2.2.1.4.10.   display         1
```

Notes:

1. dpiSample object is not accessible.
2. dpiSampleNumber object is only accessible for the SNMP GET command.
3. dpiSampleNumberString object is only accessible for the SNMP GET command.
4. dpiSampleEmpty object is not accessible for the SNMP SET command.

The DPISAMPL.C source code

The following is the source code for the DPISAMPL.C program.

Note: The characters shown below might vary due to differences in character sets.
This code is included as an example only.

```

/*****/
/*
/* TCP/IP for MVS
/* SMP/E Distribution Name: EZAEC02Z
/* File name: tcpip.SEZAINST(DPISAMPL)
/*
/*
/* SNMP-DPI - SNMP Distributed Programming Interface
/*
/* May 1991 - Version 1.0 - SNMP-DPI Version 1.0 (RFC1228)
/* Created by IBM Research.
/* Feb 1992 - Version 1.1 - Allow enterpriseID to be passed with
/* a (enterprise specific) trap
/* - allow multiple variables to be passed
/* - Use 4 octets (INTEGER from RFC1157)
/* for generic and specific type.
/* Jun 1992 - Make it run on OS/2 as well
/*
/* Copyright None
/*
/* dpisamp1.c - a sample SNMP-DPI subagent
/* - can be used to test agent DPI implementations.
/*
/* $P1= MV11816 TCPV3R2 960524 jab: zero siucv fields for connect
/* $Q1= MV27495 D316 030218 SHAGGAR: Include current header file
/*
/*****/
/* For testing with XGMON and/or SQESERV (SNMP Query Engine)
/* it is best to keep the following define for OID in sync
/* with the dpiSample objectID in the MIB description file
/* (mib_desc for XGMON, MIBDESC DATA for SQESERV on VM and
/* MIBDESC.DATA for SQESERV on MVS).
/*****/

#define OID "1.3.6.1.4.1.2.2.1.4."
#define ENTERPRISE_OID "1.3.6.1.4.1.2.2.1.4" /* dpiSample */
#define ifIndex "1.3.6.1.2.1.2.2.1.1.0"
#define egpNeighAddr "1.3.6.1.2.8.5.1.2.0"
#define PUBLIC_COMMUNITY_NAME "public"

#if defined(VM) || defined(MVS)

#define SNMPAGENTUSERID "SNMPD"
#define SNMPIUCVNAME "SNMP_DPI"
#pragma csect(CODE, "$DPISAMP")
#pragma csect(STATIC, "#DPISAMP")
#include <manifest.h> /* VM specific things */
#include "snmpnms.h" /* short external names for VM/MVS */
#include "snmp@vm.h" /* more of those short names */
#include <saiucv.h>
#include <bsdtime.h>
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 1 of 23)

```

#include <string.h>                                /*@Q1C*/
#include <inet.h>
#define  asciitoebcdic asciitoe
#define  ebcdictoascii ebcdicto
extern char  ebcdicto[], asciitoe[];
#pragma linkage(cmxlte,OS)
#define DO_ETOA(a) cmxlte((a),ebcdictoascii,strlen((a)))
#define DO_ATOE(a) cmxlte((a),asciitoebcdic,strlen((a)))
#define DO_ERROR(a) tcperror((a))
#define LOOPBACK "loopback"
#define IUCV TRUE
#define max(a,b) (((a) > (b)) ? (a) : (b))
#define min(a,b) (((a) < (b)) ? (a) : (b))

#else /* we are not on VM or MVS */

#ifdef OS2
#include <stdlib.h>
#include <types.h>
#include <doscalls.h>
#ifndef sleep
#define sleep(a) DOSSLEEP(1000 * (a))
#endif
#define close soclose
#endif

#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
// #include <arpa/inet.h>
#define DO_ETOA(a) ; /* no need for this */
#define DO_ATOE(a) ; /* no need for this */
#define DO_ERROR(a) perror((a))
#define LOOPBACK "localhost"
#define IUCV FALSE
#ifdef AIX221
#define isdigit(c) (((c) >= '0') && ((c) <= '9'))
#else
// #include <sys/select.h>
#endif /* AIX221 */

#endif /* defined(VM) || defined(MVS) */

#include <stdio.h>
#include "snmp@dpi.h"

#define WAIT_FOR_AGENT 3 /* time to wait before closing agent fd */

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 2 of 23)

```

#ifdef _NO_PROTO                                /* for classic K&R C */
static void check_arguments();
static void send_packet();
static void print_val();
static void usage();
static void init_connection();
static void init_variables();
static void await_and_read_packet();
static void handle_packet();
static void do_get();
static void do_set();
static void issue_traps();
static void issue_one_trap();
static void issue_one_trap();
static void issue_std_traps();
static void issue_ent_traps();
static void issue_ent_trapse();
static void do_register();
static void dump_bfr();
static struct dpi_set_packet *addtoset();
extern unsigned long lookup_host();

#else /* _NO_PROTO */                            /* for ANSI-C compiler */

static void check_arguments(const int argc, char *argv[]);
static void send_packet(const char * packet);
static void print_val(const int index);
static void usage(const char *progname, const int exit_rc);
static void init_connection(void);
static void init_variables(void);
static void await_and_read_packet(void);
static void handle_packet(void);
static void do_get(void);
static void do_set(void);
static void issue_traps(void);
static void issue_one_trap(void);
static void issue_one_trap(void);
static void issue_std_traps(void);
static void issue_ent_traps(void);
static void issue_ent_trapse(void);
static void do_register(void);
static void dump_bfr(const char *buf, const int len);
static struct dpi_set_packet *addtoset(struct dpi_set_packet *data,
                                       int stype);
extern unsigned long lookup_host(const char *hostname);

#endif /* _NO_PROTO */

#define OSTRING          "hex01-04:"
#define DSTRING          "Initial Display String"
#define COMMAND          "None"
#define BUFSIZE          4096
#define TIMEOUT          3
#define PACKET_LEN(packet) (((unsigned char)*(packet)) * 256 + \
                             ((unsigned char)*((packet) + 1)) + 2)

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 3 of 23)

```

/* We have the following instances for OID.x variables */
/* 0 - table */
static long number = 0; /* 1 - a number */
static unsigned char *ostring = 0; /* 2 - octet string */
static int ostring_len = 0; /* and its length */
static unsigned char *objectID = 0; /* 3 - objectID */
static int objectID_len = 0; /* and its length */
/* 4 - some empty variable */
static unsigned long ipaddr = 0; /* 5 - ipaddress */
static unsigned long counter = 1; /* 6 - a counter */
static unsigned long gauge = 1; /* 7 - a gauge */
static unsigned long ticks = 1; /* 8 - time ticks */
static unsigned char *dstring = 0; /* 9 - display string */
static unsigned char *command = 0; /* 10 - command */

static char *DPI_var[] = {
    "dpiSample",
    "dpiSampleNumber",
    "dpiSampleOctetString",
    "dpiSampleObjectID",
    "dpiSampleEmpty",
    "dpiSampleInetAddress",
    "dpiSampleCounter",
    "dpiSampleGauge",
    "dpiSampleTimeTicks",
    "dpiSampleDisplayString",
    "dpiSampleCommand"
};

static short int valid_types[] = { /* SNMP_TYPEs accepted on SET */
    -1, /* 0 do not check type */
    SNMP_TYPE_NUMBER, /* 1 number */
    SNMP_TYPE_STRING, /* 2 octet string */
    SNMP_TYPE_OBJECT, /* 3 object identifier */
    -1, /* 4 do not check type */
    SNMP_TYPE_INTERNET, /* 5 internet address */
    SNMP_TYPE_COUNTER, /* 6 counter */
    SNMP_TYPE_GAUGE, /* 7 gauge */
    SNMP_TYPE_TICKS, /* 8 time ticks */
    SNMP_TYPE_STRING, /* 9 display string */
    SNMP_TYPE_STRING /* 10 command (display string) */
};

#define OID_COUNT_FOR_TRAPS 9
#define OID_COUNT 10

static char *packet = NULL; /* ptr to send packet. */
static char inbuf[BUFSIZE]; /* buffer for receive packets */
static int dpi_fd; /* fd for socket to DPI agent */
static short int dpi_port; /* DPI_port at agent */
static unsigned long dpi_ipaddress; /* IP address of DPI agent */
static char *dpi_hostname; /* hostname of DPI agent */
static char *dpi_userid; /* userid of DPI agent VM/MVS */
static char *var_gid; /* groupID received */
static char *var_oid; /* objectID received */

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 4 of 23)


```

static int      var_index;      /* OID variable index */
static unsigned char var_type;  /* SET value type */
static char     *var_value;     /* SET value */
static short int var_value_len; /* SET value length */
static int      debug_lvl = 0; /* current debug level */
static int      use_iucv = IUCV; /* optional use of AF_IUCV */
static int      do_quit = FALSE; /* Quit in await loop */
static int      trap_gtype = 0; /* trap generic type */
static int      trap_stype = 0; /* trap specific type */
static char     *trap_data = NULL; /* trap data */
static int      do_trap = 0; /* switch for traps */
#define ONE_TRAP 1
#define ONE_TRAPE 2
#define STD_TRAPS 3
#define ENT_TRAPS 4
#define ENT_TRAPSE 5
#define ALL_TRAPS 6
#define MAX_TRAPE_DATA 10 /* data for extended trap */
static long     trape_gtype = 6; /* trap generic type */
static long     trape_stype = 11; /* trap specific type */
static char     *trape_eprise = NULL; /* enterprise id */
static char     *trape_data[MAX_TRAPE_DATA]; /* pointers to data values */
static int      trape_datacnt; /* actual number of values */

#ifdef _NO_PROTO /* for classic K&R C */
main(argc, argv) /* main line */
int  argc;
char *argv[];
#else /* _NO_PROTO */ /* for ANSI-C compiler */
main(const int argc, char *argv[]) /* main line */
#endif /* _NO_PROTO */
{
    check_arguments(argc, argv); /* check callers arguments */
    dpi_ipaddress = lookup_host(dpi_hostname); /* get ip address */
    init_connection(); /* connect to specified agent */
    init_variables(); /* initialize our variables */
    if (do_trap) { /* we just need to do traps */
        issue_traps(); /* issue the trap(s) */
        sleep(WAIT_FOR_AGENT); /* sleep a bit, so agent can */
        close(dpi_fd); /* read data before we close */
        exit(0); /* and that's it */
    } /* end if (do_trap) */
    do_register(); /* register our objectIDs */
    printf("%s ready and awaiting queries from agent\n", argv[0]);
    while (do_quit == FALSE) { /* forever until quit or error */
        await_and_read_packet(); /* wait for next packet */
        handle_packet(); /* handle it */
        if (do_trap) issue_traps(); /* request to issue traps */
    } /* while loop */
    sleep(WAIT_FOR_AGENT); /* allow agent to read response */
    printf("Quitting, %s set to: quit\n", DPI_var[10]);
    exit(2); /* sampleDisplayString == quit */
}

#ifdef _NO_PROTO /* for classic K&R C */

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 5 of 23)

```

static void issue_traps()
#else /* _NO_PROTO */                               /* for ANSI-C compiler */
static void issue_traps(void)
#endif /* _NO_PROTO */
{
    switch (do_trap) {                               /* let's see which one(s) */
    case ONE_TRAP:                                   /* only need to issue one trap */
        issue_one_trap();                           /* go issue the one trap */
        break;
    case ONE_TRAPE:                                  /* only need to issue one trape */
        issue_one_trapse();                           /* go issue the one trape */
        break;
    case STD_TRAPS:                                   /* only need to issue std traps */
        issue_std_traps();                             /* standard traps gtypes 0-5 */
        break;
    case ENT_TRAPS:                                   /* only need to issue ent traps */
        issue_ent_traps();                             /* enterprise specific traps */
        break;
    case ENT_TRAPSE:                                  /* only need to issue ent trapse */
        issue_ent_trapse();                             /* enterprise specific trapse */
        break;
    case ALL_TRAPS:                                   /* only need to issue std traps */
        issue_std_traps();                             /* standard traps gtypes 0-5 */
        issue_ent_traps();                             /* enterprise specific traps */
        issue_ent_trapse();                             /* enterprise specific trapse */
        break;
    default:
        break;
    }
    do_trap = 0;                                     /* end switch (do_trap) */
                                                /* reset do_trap switch */
}

#ifdef _NO_PROTO                                     /* for classic K&R C */
static void await_and_read_packet() /* await packet from DPI agent */
#else /* _NO_PROTO */                               /* for ANSI-C compiler */
static void await_and_read_packet(void) /* await packet from DPI agent */
#endif /* _NO_PROTO */
{
    int len, rc, bytes_to_read, bytes_read = 0;
#ifdef OS2
    int socks[5];
#else
    fd_set read_mask;
#endif
    struct timeval timeout;

#ifdef OS2
    socks[0] = dpi_fd;
    rc = select(socks, 1, 0, 0, -1L);
#else
    FD_ZERO(&read_mask);
    FD_SET(dpi_fd, &read_mask);                       /* wait for data */
    rc = select(dpi_fd+1, &read_mask, NULL, NULL, NULL);
#endif
    if (rc != 1) {                                     /* exit on error */

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 6 of 23)

```

        DO_ERROR("await_and_read_packet:  select");
        close(dpi_fd);
        exit(1);
    }
#ifdef OS2
    len = recv(dpi_fd, inbuf, 2, 0);          /* read 2 bytes first */
#else
    len = read(dpi_fd, inbuf, 2);             /* read 2 bytes first */
#endif
    if (len <= 0) {                          /* exit on error or EOF */
        if (len < 0) DO_ERROR("await_and_read_packet:  read");
        else printf("Quitting, EOF received from DPI-agent\n");
        close(dpi_fd);
        exit(1);
    }
    bytes_to_read = (inbuf[0] << 8) + inbuf[1]; /* bytes to follow */
    if (BUFSIZE < (bytes_to_read + 2)) {      /* exit if too much */
        printf("Quitting, packet larger than %d byte buffer\n",BUFSIZE);
        close(dpi_fd);
        exit(1);
    }
    while (bytes_to_read > 0) {               /* while bytes to read */
#ifdef OS2
        socks[0] = dpi_fd;
        len = select(socks, 1, 0, 0, 3000L);
#else
        timeout.tv_sec = 3;                  /* wait max 3 seconds */
        timeout.tv_usec = 0;
        FD_SET(dpi_fd, &read_mask);         /* check for data */
        len = select(dpi_fd+1, &read_mask, NULL, NULL, &timeout);
#endif
        if (len == 1) {                     /* select returned OK */
#ifdef OS2
            len = recv(dpi_fd, &inbuf[2] + bytes_read, bytes_to_read, 0);
#else
            len = read(dpi_fd, &inbuf[2] + bytes_read, bytes_to_read);
#endif
        } /* end if (len == 1) */
        if (len <= 0) {                     /* exit on error or EOF */
            if (len < 0) DO_ERROR("await_and_read_packet:  read");
            printf("Can't read remainder of packet\n");
            close(dpi_fd);
            exit(1);
        } else {                           /* count bytes_read */
            bytes_read += len;
            bytes_to_read -= len;
        }
    } /* while (bytes_to_read > 0) */
}

#ifdef _NO_PROTO                                /* for classic K&R C */
static void handle_packet() /* handle DPI packet from agent */
#else /* _NO_PROTO */                                /* for ANSI-C compiler */
static void handle_packet(void) /* handle DPI packet from agent */
#endif /* _NO_PROTO */

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 7 of 23)

```

{
    struct    snmp_dpi_hdr    *hdr;

    if (debug_lvl > 2) {
        printf("Received following SNMP-DPI packet:\n");
        dump_bfr(inbuf, PACKET_LEN(inbuf));
    }
    hdr = pDPIpacket(inbuf);                /* parse received packet */
    if (hdr == 0) {                          /* ignore if can't parse */
        printf("Ignore received packet, could not parse it!\n");
        return;
    }
    packet    = NULL;
    var_type  = 0;
    var_oid   = "";
    var_gid   = "";
    switch (hdr->packet_type) {
        /* extract pointers and/or data from specific packet types, */
        /* such that we can use them independent of packet type. */
        case SNMP_DPI_GET:
            if (debug_lvl > 0) printf("SNMP_DPI_GET for ");
            var_oid   = hdr->packet_body.dpi_get->object_id;
            break;
        case SNMP_DPI_GET_NEXT:
            if (debug_lvl > 0) printf("SNMP_DPI_GET_NEXT for ");
            var_oid   = hdr->packet_body.dpi_next->object_id;
            var_gid   = hdr->packet_body.dpi_next->group_id;
            break;
        case SNMP_DPI_SET:
            if (debug_lvl > 0) printf("SNMP_DPI_SET for ");
            var_value_len = hdr->packet_body.dpi_set->value_len;
            var_value     = hdr->packet_body.dpi_set->value;
            var_oid       = hdr->packet_body.dpi_set->object_id;
            var_type      = hdr->packet_body.dpi_set->type;
            break;
        default: /* Return a GEN_ERROR */
            if (debug_lvl > 0) printf("Unexpected packet_type %d, genErr\n",
                                     hdr->packet_type);
            packet = mkDPIresponse(SNMP_GEN_ERR, NULL);
            fDPIparse(hdr); /* return storage allocated by pDPIpacket() */
            send_packet(packet);
            return;
            break;
    } /* end switch(hdr->packet_type) */
    if (debug_lvl > 0) printf("objectID: %s \n", var_oid);

    if (strlen(var_oid) <= strlen(OID)) { /* not in our tree */
        if (hdr->packet_type == SNMP_DPI_GET_NEXT) var_index = 0; /* OK */
        else { /* cannot handle */
            if (debug_lvl > 0) printf("...Ignored %s, noSuchName\n", var_oid);
            packet = mkDPIresponse(SNMP_NO_SUCH_NAME, NULL);
            fDPIparse(hdr); /* return storage allocated by pDPIpacket() */
            send_packet(packet);
            return;
        }
    }
}

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 8 of 23)

```

} else { /* Extract our variable index (from OID.index.instance) */
/* We handle any instance the same (we only have one instance) */
var_index = atoi(&var_oid[strlen(OID)]);
}
if (debug_lvl > 1) {
printf("...The groupID=%s\n",var_gid);
printf("...Handle as if objectID=%s%d\n",OID,var_index);
}
switch (hdr->packet_type) {
case SNMP_DPI_GET:
do_get(); /* do a get to return response */
break;
case SNMP_DPI_GET_NEXT:
{ char toid[256]; /* space for temporary objectID */
var_index++; /* do a get for the next variable */
sprintf(toid,"%s%d",OID,var_index); /* construct objectID */
var_oid = toid; /* point to it */
do_get(); /* do a get to return response */
} break;
case SNMP_DPI_SET:
if (debug_lvl > 1) printf("...value_type=%d\n",var_type);
do_set(); /* set new value first */
if (packet) break; /* some error response was generated */
do_get(); /* do a get to return response */
break;
}
fDPIparse(hdr); /* return storage allocated by pDPIpacket() */
}

#ifdef _NO_PROTO /* for classic K&R C */
static void do_get() /* handle SNMP_GET request */
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void do_get(void) /* handle SNMP_GET request */
#endif /* _NO_PROTO */
{
struct dpi_set_packet *data = NULL;

switch (var_index) {
case 0: /* table, cannot be queried by itself */
printf("...Should not issue GET for table %s.0\n", OID);
break;
case 1: /* a number */
data = mkDPISet(var_oid,SNMP_TYPE_NUMBER,sizeof(number),&number);
break;
case 2: /* an octet_string (can have binary data) */
data = mkDPISet(var_oid,SNMP_TYPE_STRING,ostring_len,ostring);
break;
case 3: /* object id */
data = mkDPISet(var_oid,SNMP_TYPE_OBJECT,objectID_len,objectID);
break;
case 4: /* some empty variable */
data = mkDPISet(var_oid,SNMP_TYPE_EMPTY,0,NULL);
break;
case 5: /* internet address */
data = mkDPISet(var_oid,SNMP_TYPE_INTERNET,sizeof(ipaddr),&ipaddr);
}
}

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 9 of 23)

```

        break;
case 6: /* counter (unsigned) */
    data = mkDPIset(var_oid,SNMP_TYPE_COUNTER,sizeof(counter),&counter);
    break;
case 7: /* gauge (unsigned) */
    data = mkDPIset(var_oid,SNMP_TYPE_GAUGE,sizeof(gauge),&gauge);
    break;
case 8: /* time ticks (unsigned) */
    data = mkDPIset(var_oid,SNMP_TYPE_TICKS,sizeof(ticks),&ticks);
    break;
case 9: /* a display_string (printable ascii only) */
    DO_ETOA(dstring);
    data = mkDPIset(var_oid,SNMP_TYPE_STRING,strlen(dstring),dstring);
    DO_ATOE(dstring);
    break;
case 10: /* a command request (command is a display string) */
    DO_ETOA(command);
    data = mkDPIset(var_oid,SNMP_TYPE_STRING,strlen(command),command);
    DO_ATOE(command);
    break;
default: /* Return a NoSuchName */
    if (debug_lvl > 1)
        printf("...GET[NEXT] for %s, not found\n", var_oid);
    break;
} /* end switch (var_index) */

if (data) {
    if (debug_lvl > 0) {
        printf("...Sending response oid: %s type: %d\n",
            var_oid, data->type);
        printf(".....Current value: ");
        print_val(var_index); /* prints \n at end */
    }
    packet = mkDPIresponse(SNMP_NO_ERROR,data);
} else { /* Could have been an error in mkDPIset though */
    if (debug_lvl > 0) printf("...Sending response noSuchName\n");
    packet = mkDPIresponse(SNMP_NO_SUCH_NAME,NULL);
} /* end if (data) */
if (packet) send_packet(packet);
}

#ifdef _NO_PROTO
/* for classic K&R C */
static void do_set() /* handle SNMP_SET request */
#else /* _NO_PROTO */
/* for ANSI-C compiler */
static void do_set(void) /* handle SNMP_SET request */
#endif /* _NO_PROTO */
{
    unsigned long *ulp;
    long *lp;

    if (valid_types[var_index] != var_type &&
        valid_types[var_index] != -1) {
        printf("...Ignored set request with type %d, expect type %d,",
            var_type, valid_types[var_index]);
        printf(" Returning badValue\n");
    }
}

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 10 of 23)

```

    packet = mkDPIresponse(SNMP_BAD_VALUE, NULL);
    if (packet) send_packet(packet);
    return;
}
switch (var_index) {
case 0: /* table, cannot set table. */
    if (debug_lvl > 0) printf("...Ignored set TABLE, noSuchName\n");
    packet = mkDPIresponse(SNMP_NO_SUCH_NAME, NULL);
    break;
case 1: /* a number */
    lp = (long *)var_value;
    number = *lp;
    break;
case 2: /* an octet_string (can have binary data) */
    free(ostring);
    ostring = (char *)malloc(var_value_len + 1);
    bcopy(var_value, ostring, var_value_len);
    ostring_len = var_value_len;
    ostring[var_value_len] = '\0'; /* so we can use it as a string */
    break;
case 3: /* object id */
    free(objectID);
    objectID = (char *)malloc(var_value_len + 1);
    bcopy(var_value, objectID, var_value_len);
    objectID_len = var_value_len;
    if (objectID[objectID_len - 1]) {
        objectID[objectID_len++] = '\0'; /* a valid one needs a null */
        if (debug_lvl > 0)
            printf("...added a terminating null to objectID\n");
    }
    break;
case 4: /* an empty variable, cannot set */
    if (debug_lvl > 0) printf("...Ignored set EMPTY, readOnly\n");
    packet = mkDPIresponse(SNMP_READ_ONLY, NULL);
    break;
case 5: /* Internet address */
    ulp = (unsigned long *)var_value;
    ipaddr = *ulp;
    break;
case 6: /* counter (unsigned) */
    ulp = (unsigned long *)var_value;
    counter = *ulp;
    break;
case 7: /* gauge (unsigned) */
    ulp = (unsigned long *)var_value;
    gauge = *ulp;
    break;
case 8: /* time ticks (unsigned) */
    ulp = (unsigned long *)var_value;
    ticks = *ulp;
    break;
case 9: /* a display_string (printable ascii only) */
    free(dstring);
    dstring = (char *)malloc(var_value_len + 1);
    bcopy(var_value, dstring, var_value_len);

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 11 of 23)

```

    dstring[var_value_len] = '\0'; /* so we can use it as a string */
    DO_ATOE(dstring);
    break;
case 10: /* a request to execute a command */
    free(command);
    command = (char *)malloc(var_value_len + 1);
    bcopy(var_value, command, var_value_len);
    command[var_value_len] = '\0'; /* so we can use it as a string */
    DO_ATOE(command);
    if (strcmp("all_traps",command) == 0) do_trap = ALL_TRAPS;
    else if (strcmp("std_traps",command) == 0) do_trap = STD_TRAPS;
    else if (strcmp("ent_traps",command) == 0) do_trap = ENT_TRAPS;
    else if (strcmp("ent_trapse",command) == 0) do_trap = ENT_TRAPSE;
    else if (strcmp("all_traps",command) == 0) do_trap = ALL_TRAPS;
    else if (strcmp("quit",command) == 0) do_quit = TRUE;
    else break;
    if (debug_lvl > 0)
        printf("...Action requested: %s set to: %s\n",
            DPI_var[10], command);
    break;
default: /* NoSuchName */
    if (debug_lvl > 0)
        printf("...Ignored set for %s, noSuchName\n", var_oid);
    packet = mkDPIresponse(SNMP_NO_SUCH_NAME,NULL);
    break;
} /* end switch (var_index) */
if (packet) send_packet(packet);
}

#ifdef _NO_PROTO /* for classic K&R C */
static void issue_std_traps()
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void issue_std_traps(void)
#endif /* _NO_PROTO */
{
    trap_stype = 0;
    trap_data = dpi_hostname;
    for (trap_gtype=0; trap_gtype<6; trap_gtype++) {
        issue_one_trap();
        if (trap_gtype == 0) sleep(10); /* some managers purge cache */
    }
}

#ifdef _NO_PROTO /* for classic K&R C */
static void issue_ent_traps()
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void issue_ent_traps(void)
#endif /* _NO_PROTO */
{
    char temp_string[256];

    trap_gtype = 6;
    for (trap_stype = 1; trap_stype < 10; trap_stype++) {
        trap_data = temp_string;
        switch (trap_stype) {

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 12 of 23)


```

    case 1 :
        sprintf(temp_string,"%ld",number);
        break;
    case 2 :
        sprintf(temp_string,"%s",ostring);
        break;
    case 3 :
        trap_data = objectID;
        break;
    case 4 :
        trap_data = "";
        break;
    case 5 :
        trap_data = dpi_hostname;
        break;
    case 6 :
        sleep(1); /* give manager a break */
        sprintf(temp_string,"%lu",counter);
        break;
    case 7 :
        sprintf(temp_string,"%lu",gauge);
        break;
    case 8 :
        sprintf(temp_string,"%lu",ticks);
        break;
    case 9 :
        trap_data = dstring;
        break;
} /* end switch (trap_stype) */
issue_one_trap();
}

/* issue a set of extended traps, pass enterprise ID and multiple
 * variable (assume octet string) as passed by caller
 */
#ifdef _NO_PROTO /* for classic K&R C */
static void issue_ent_trapse()
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void issue_ent_trapse(void)
#endif /* _NO_PROTO */
{
    int i, n;
    struct dpi_set_packet *data = NULL;
    unsigned char *packet = NULL;
    unsigned long ipaddr, ulnum;
    char oid[256];
    char *cp;

    trape_gtype = 6;
    trape_eprise = ENTERPRISE_OID;
    for (n=11; n < (11+OID_COUNT_FOR_TRAPS); n++) {
        data = 0;
        trape_stype = n;
        for (i=1; i<=(n-10); i++)

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 13 of 23)

```

        data = addtoset(data, i);
    if (data == 0) {
        printf("Could not make dpi_set_packet\n");
        return;
    }
    packet = mkDPITrape(trape_gtype, trape_stype, data, trape_eprise);
    if ((debug_lvl > 0) && (packet)) {
        printf("sending trape packet: %lu %lu enterprise=%s\n",
            trape_gtype, trape_stype, trape_eprise);
    }
    if (packet) send_packet(packet);
    else printf("Could not make trape packet\n");
}

/* issue one extended trap, pass enterprise ID and multiple
 * variable (assume octet string) as passed by caller
 */
#ifdef _NO_PROTO                                /* for classic K&R C */
static void issue_one_trape()
#else /* _NO_PROTO */                          /* for ANSI-C compiler */
static void issue_one_trape(void)
#endif /* _NO_PROTO */
{
    struct dpi_set_packet *data = NULL;
    unsigned char *packet = NULL;
    char oid[256];
    char *cp;
    int i;

    for (i=0; i<trape_datacnt; i++) {
        sprintf(oid, "%s2.%d", OID, i);
        /* assume an octet_string (could have hex data) */
        data = mkDPIList(data, oid, SNMP_TYPE_STRING,
            strlen(trape_data[i]), trape_data[i]);
        if (data == 0) {
            printf("Could not make dpiset_packet\n");
        } else if (debug_lvl > 0) {
            printf("Preparing: [oid=%s] value: ", oid);
            printf("'");
            for (cp = trape_data[i]; *cp; cp++) /* loop through data */
                printf("%2.2X", *cp);          /* hex print one byte */
            printf("'H\n");
        }
    }
    packet = mkDPITrape(trape_gtype, trape_stype, data, trape_eprise);
    if ((debug_lvl > 0) && (packet)) {
        printf("sending trape packet: %lu %lu enterprise=%s\n",
            trape_gtype, trape_stype, trape_eprise);
    }
    if (packet) send_packet(packet);
    else printf("Could not make trape packet\n");
}

#ifdef _NO_PROTO                                /* for classic K&R C */

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 14 of 23)

```

static void issue_one_trap()
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void issue_one_trap(void)
#endif /* _NO_PROTO */
{
    long int num; /* must be 4 bytes */
    struct dpi_set_packet *data = NULL;
    unsigned char *packet = NULL;
    unsigned long ipaddr, ulnum;
    char oid[256];
    char *cp;

    switch (trap_gtype) {
        /* all traps are handled more or less the same sofar. */
        /* could put specific handling here if needed/wanted. */
        case 0: /* simulate cold start */
        case 1: /* simulate warm start */
        case 4: /* simulate authentication failure */
            strcpy(oid,"none");
            break;
        case 2: /* simulate link down */
        case 3: /* simulate link up */
            strcpy(oid,ifIndex);
            num = 1;
            data = mkDPISet(oid, SNMP_TYPE_NUMBER, sizeof(num), &num);
            break;
        case 5: /* simulate EGP neighbor loss */
            strcpy(oid,egpNeighAddr);
            ipaddr = lookup_host(trap_data);
            data = mkDPISet(oid, SNMP_TYPE_INTERNET, sizeof(ipaddr), &ipaddr);
            break;
        case 6: /* simulate enterprise specific trap */
            sprintf(oid,"%s%d.0",OID, trap_stype);
            switch (trap_stype) {
                case 1: /* a number */
                    num = strtol(trap_data,(char **)0,10);
                    data = mkDPISet(oid, SNMP_TYPE_NUMBER, sizeof(num), &num);
                    break;
                case 2: /* an octet_string (could have hex data) */
                    data = mkDPISet(oid,SNMP_TYPE_STRING,strlen(trap_data),trap_data);
                    break;
                case 3: /* object id */
                    data = mkDPISet(oid,SNMP_TYPE_OBJECT,strlen(trap_data) + 1,
                                   trap_data);
                    break;
                case 4: /* an empty variable value */
                    data = mkDPISet(oid, SNMP_TYPE_EMPTY, 0, 0);
                    break;
                case 5: /* internet address */
                    ipaddr = lookup_host(trap_data);
                    data = mkDPISet(oid, SNMP_TYPE_INTERNET, sizeof(ipaddr), &ipaddr);
                    break;
                case 6: /* counter (unsigned) */
                    ulnum = strtoul(trap_data,(char **)0,10);
                    data = mkDPISet(oid, SNMP_TYPE_COUNTER, sizeof(ulnum), &ulnum);
            }
    }
}

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 15 of 23)

```

        break;
case 7: /* gauge (unsigned) */
    ulnum = strtoul(trap_data, (char **)0, 10);
    data = mkDPIset(oid, SNMP_TYPE_GAUGE, sizeof(ulnum), &ulnum);
    break;
case 8: /* time ticks (unsigned) */
    ulnum = strtoul(trap_data, (char **)0, 10);
    data = mkDPIset(oid, SNMP_TYPE_TICKS, sizeof(num), &ulnum);
    break;
case 9: /* a display_string (ascii only) */
    DO_ETOA(trap_data);
    data = mkDPIset(oid, SNMP_TYPE_STRING, strlen(trap_data), trap_data);
    DO_ATOE(trap_data);
    break;
default: /* handle as string */
    printf("Unknown specific trap type: %s, assume octet_string\n",
           trap_stype);
    data = mkDPIset(oid, SNMP_TYPE_STRING, strlen(trap_data), trap_data);
    break;
} /* end switch (trap_stype) */
break;
default: /* unknown trap */
    printf("Unknown general trap type: %s\n", trap_gtype);
    return;
break;
} /* end switch (trap_gtype) */

packet = mkDPItrap(trap_gtype, trap_stype, data);
if ((debug_lvl > 0) && (packet)) {
    printf("sending trap packet: %u %u [oid=%s] value: ",
           trap_gtype, trap_stype, oid);
    if (trap_stype == 2) {
        printf("");
        for (cp = trap_data; *cp; cp++) /* loop through data */
            printf("%2.2x", *cp); /* hex print one byte */
        printf("\n");
    } else printf("%s\n", trap_data);
}
if (packet) send_packet(packet);
else printf("Could not make trap packet\n");
}

#ifdef _NO_PROTO /* for classic K&R C */
static void send_packet(packet) /* DPI packet to agent */
char *packet;
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void send_packet(const char *packet) /* DPI packet to agent */
#endif /* _NO_PROTO */
{
    int rc;

    if (debug_lvl > 2) {
        printf("...Sending DPI packet:\n");
        dump_bfr(packet, PACKET_LEN(packet));
    }
}

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 16 of 23)

```

#ifdef OS2
    rc = send(dpi_fd,packet,PACKET_LEN(packet),0);
#else
    rc = write(dpi_fd,(unsigned char *)packet,PACKET_LEN(packet));
#endif
    if (rc != PACKET_LEN(packet)) DO_ERROR("send_packet: write");
    /* no need to free packet (static buffer in mkDPI.... routine) */
}

#ifdef _NO_PROTO                                /* for classic K&R C */
static void do_register() /* register our objectIDs with agent */
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void do_register(void) /* register our objectIDs with agent */
#endif /* _NO_PROTO */
{
    int i, rc;
    char toid[256];

    if (debug_lvl > 0) printf("Registering variables:\n");
    for (i=1; i<=OID_COUNT; i++) {
        sprintf(toid,"%s%d.",OID,i);
        packet = mkDPIregister(toid);
#ifdef OS2
        rc = send(dpi_fd, packet, PACKET_LEN(packet),0);
#else
        rc = write(dpi_fd, packet, PACKET_LEN(packet));
#endif
        if (rc <= 0) {
            DO_ERROR("do_register: write");
            printf("Quitting, unsuccessful register for %s\n",toid);
            close(dpi_fd);
            exit(1);
        }
        if (debug_lvl > 0) {
            printf("...Registered: %-25s oid: %s\n",DPI_var[i],toid);
            printf(".....Initial value: ");
            print_val(i); /* prints \n at end */
        }
    }
}

/* add specified variable to list of variable in the dpi_set_packet
*/
#ifdef _NO_PROTO                                /* for classic K&R C */
struct dpi_set_packet *addtoset(data, stype)
struct dpi_set_packet *data;
int stype;
#else /* _NO_PROTO */ /* for ANSI-C compiler */
struct dpi_set_packet *addtoset(struct dpi_set_packet *data, int stype)
#endif /* _NO_PROTO */
{
    char var_oid[256];

    sprintf(var_oid,"%s%d.0",OID, stype);
    switch (stype) {

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 17 of 23)

```

case 1: /* a number */
    data = mkDPIList(data, var_oid, SNMP_TYPE_NUMBER,
                     sizeof(number), &number);
    break;
case 2: /* an octet_string (can have binary data) */
    data = mkDPIList(data, var_oid, SNMP_TYPE_STRING,
                     ostring_len, ostring);
    break;
case 3: /* object id */
    data = mkDPIList(data, var_oid, SNMP_TYPE_OBJECT,
                     objectID_len, objectID);
    break;
case 4: /* some empty variable */
    data = mkDPIList(data, var_oid, SNMP_TYPE_EMPTY, 0, NULL);
    break;
case 5: /* internet address */
    data = mkDPIList(data, var_oid, SNMP_TYPE_INTERNET,
                     sizeof(ipaddr), &ipaddr);
    break;
case 6: /* counter (unsigned) */
    data = mkDPIList(data, var_oid, SNMP_TYPE_COUNTER,
                     sizeof(counter), &counter);
    break;
case 7: /* gauge (unsigned) */
    data = mkDPIList(data, var_oid, SNMP_TYPE_GAUGE,
                     sizeof(gauge), &gauge);
    break;
case 8: /* time ticks (unsigned) */
    data = mkDPIList(data, var_oid, SNMP_TYPE_TICKS,
                     sizeof(ticks), &ticks);
    break;
case 9: /* a display_string (printable ascii only) */
    DO_ETOA(dstring);
    data = mkDPIList(data, var_oid, SNMP_TYPE_STRING,
                     strlen(dstring), dstring);
    DO_ATOE(dstring);
    break;
} /* end switch (stype) */
return(data);
}

#ifdef _NO_PROTO                                /* for classic K&R C */
static void print_val(index)
int index;
#else /* _NO_PROTO */                          /* for ANSI-C compiler */
static void print_val(const int index)
#endif /* _NO_PROTO */
{
    char *cp;
    struct in_addr temp_ipaddr;

    switch (index) {
case 1 :
    printf("%ld\n", number);
    break;

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 18 of 23)

```

case 2 :
    printf("");
    for (cp = ostring; cp < ostring + ostring_len; cp++)
        printf("%2.2x",*cp);
    printf("H\n");
    break;
case 3 :
    printf("%s\n", objectID_len, objectID);
    break;
case 4 :
    printf("no value (EMPTY)\n");
    break;
case 5 :
    temp_ipaddr.s_addr = ipaddr;
    printf("%s\n",inet_ntoa(temp_ipaddr));
/* This worked on VM, MVS and AIX, but not on OS/2
* printf("%d.%d.%d.%d\n", (ipaddr >> 24), ((ipaddr << 8) >> 24),
* ((ipaddr << 16) >> 24), ((ipaddr << 24) >> 24));
*/
    break;
case 6 :
    printf("%lu\n",counter);
    break;
case 7 :
    printf("%lu\n",gauge);
    break;
case 8 :
    printf("%lu\n",ticks);
    break;
case 9 :
    printf("%s\n",dstring);
    break;
case 10 :
    printf("%s\n",command);
    break;
} /* end switch(index) */
}

#ifdef _NO_PROTO
static void check_arguments(argc, argv) /* for classic K&R C */
int argc;
char *argv[];
/* check arguments */
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void check_arguments(const int argc, char *argv[])
#endif /* _NO_PROTO */
{
    char *hname, *cname;
    int i, j;

    dpi_userid = hname = cname = NULL;
    for (i=1; argc > i; i++) {
        if (strcmp(argv[i],"-d") == 0) {
            i++;
            if (argc > i) {
                debug_lvl = atoi(argv[i]);
            }
        }
    }
}

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 19 of 23)

```

        if (debug_lvl >= 5) {
            DPIdebug(1);
        }
    }
} else if (strcmp(argv[i], "-trap") == 0) {
    if (argc > i+3) {
        trap_gtype = atoi(argv[i+1]);
        trap_stype = atoi(argv[i+2]);
        trap_data = argv[i+3];
        i = i + 3;
        do_trap = ONE_TRAP;
    } else usage(argv[0], 1);
} else if (strcmp(argv[i], "-trape") == 0) {
    if (argc > i+4) {
        trape_gtype = strtoul(argv[i+1], (char**)0, 10);
        trape_stype = strtoul(argv[i+2], (char**)0, 10);
        trape_eprise = argv[i+3];
        for (i = i + 4, j = 0;
            (argc > i) && (j < MAX_TRAPE_DATA);
            i++, j++) {
            trape_data[j] = argv[i];
        }
        trape_datacnt = j;
        do_trap = ONE_TRAPE;
        break; /* -trape must be last option */
    } else usage(argv[0], 1);
} else if (strcmp(argv[i], "-all_traps") == 0) {
    do_trap = ALL_TRAPS;
} else if (strcmp(argv[i], "-std_traps") == 0) {
    do_trap = STD_TRAPS;
} else if (strcmp(argv[i], "-ent_traps") == 0) {
    do_trap = ENT_TRAPS;
} else if (strcmp(argv[i], "-ent_trapse") == 0) {
    do_trap = ENT_TRAPSE;
}
#ifdef VM || defined(MVS)
} else if (strcmp(argv[i], "-inet") == 0) {
    use_iucv = 0;
} else if (strcmp(argv[i], "-iucv") == 0) {
    use_iucv = TRUE;
} else if (strcmp(argv[i], "-u") == 0) {
    use_iucv = TRUE; /* -u implies -iucv */
    i++;
    if (argc > i) {
        dpi_userid = argv[i];
    }
}
#endif
} else if (strcmp(argv[i], "?") == 0) {
    usage(argv[0], 0);
} else {
    if (hname == NULL) hname = argv[i];
    else if (cname == NULL) cname = argv[i];
    else usage(argv[0], 1);
}
}
if (hname == NULL) hname = LOOPBACK; /* use default */

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 20 of 23)


```

    if (cname == NULL) cname = PUBLIC_COMMUNITY_NAME; /* use default */
#ifdef(VM) || defined(MVS)
    if (dpi_userid == NULL) dpi_userid = SNMPAGENTUSERID;
    if (debug_lvl > 2)
        printf("hname=%s, cname=%s, userid=%s\n",hname,cname,dpi_userid);
#else
    if (debug_lvl > 2)
        printf("hname=%s, cname=%s\n",hname,cname);
#endif
    if (use_iucv != TRUE) {
        DO_ETOA(cname); /* for VM or MVS */
        dpi_port = query_DPI_port(hname,cname);
        DO_ATOE(cname); /* for VM or MVS */
        if (dpi_port == -1) {
            printf("No response from agent at %s(%s)\n",hname,cname);
            exit(1);
        }
    } else dpi_port == -1;
    dpi_hostname = hname;
}

#ifdef _NO_PROTO /* for classic K&R C */
static void usage(pname, exit_rc)
char *pname;
int exit_rc;
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void usage(const char *pname, const int exit_rc)
#endif /* _NO_PROTO */
{
    printf("Usage: %s [-d debug_lvl] [-trap g_type s_type data]", pname);
    printf(" [-all_traps]\n");
    printf("%s[-trape g_type s_type enterprise data1 data2 .. datan]\n",
        strlen(pname)+8,"");
    printf("%s[-std_traps] [-ent_traps] [-ent_trapse]\n",
        strlen(pname)+8,"");
#ifdef(VM) || defined(MVS)
    printf("%s[-iucv] [-u agent_userid]\n",strlen(pname)+8,"");
    printf("%s", strlen(pname)+8,"");
    printf("[-inet] [agent_hostname [community_name]]\n");
    printf("default: -d 0 -iucv -u %s\n", SNMPAGENTUSERID);
    printf("          -inet %s %s\n", LOOPBACK, PUBLIC_COMMUNITY_NAME);
#else
    printf("%s[agent_hostname [community_name]]\n",strlen(pname)+8,"");
    printf("default: -d 0 %s %s\n", LOOPBACK, PUBLIC_COMMUNITY_NAME);
#endif
    exit(exit_rc);
}

#ifdef _NO_PROTO /* for classic K&R C */
static void init_variables() /* initialize our variables */
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void init_variables(void) /* initialize our variables */
#endif /* _NO_PROTO */
{
    char ch, *cp;

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 21 of 23)

```

ostring = (char *)malloc(strlen(OSTRING) + 4 + 1 );
bcopy(OSTRING,ostring,strlen(OSTRING));
ostring_len = strlen(OSTRING);
for (ch=1;ch<5;ch++)          /* add hex data 0x01020304 */
    ostring[ostring_len++] = ch;
ostring[ostring_len] = '\0';    /* so we can use it as a string */
objectID = (char *)malloc(strlen(OID));
objectID_len = strlen(OID);
bcopy(OID,objectID,strlen(OID));
if (objectID[objectID_len - 1] == '.') /* if trailing dot, */
    objectID[objectID_len - 1] = '\0'; /* remove it */
else objectID_len++;             /* length includes null */
dstring = (char *)malloc(strlen(DSTRING)+1);
bcopy(DSTRING,dstring,strlen(DSTRING)+1);
command = (char *)malloc(strlen(COMMAND)+1);
bcopy(COMMAND,command,strlen(COMMAND)+1);
ipaddr = dpi_ipaddress;
}

#ifdef _NO_PROTO                /* for classic K&R C */
static void init_connection()    /* connect to the DPI agent */
#else /* _NO_PROTO */          /* for ANSI-C compiler */
static void init_connection(void) /* connect to the DPI agent */
#endif /* _NO_PROTO */
{
    int rc;
    int sasize;                /* size of socket structure */
    struct sockaddr_in sin;     /* socket address AF_INET */
    struct sockaddr *sa;        /* socket address general */
#ifdef VM || defined(MVS)
    struct sockaddr_iucv siu;    /* socket address AF_IUCV */

    if (use_iucv == TRUE) {
        printf("Connecting to %s userid %s (TCP, AF_IUCV)\n",
            dpi_hostname,dpi_userid); /* @P1C*/
        bzero(&siu,sizeof(siu));
        siu.siucv_family = AF_IUCV;
        siu.siucv_addr = 0; /* @P1C*/
        siu.siucv_port = 0; /* @P1C*/
        memset(siu.siucv_nodeid, ' ', sizeof(siu.siucv_nodeid));
        memset(siu.siucv_userid, ' ', sizeof(siu.siucv_userid));
        memset(siu.siucv_name, ' ', sizeof(siu.siucv_name));
        bcopy(dpi_userid, siu.siucv_userid, min(8,strlen(dpi_userid)));
        bcopy(SNMP_IUCVNAME, siu.siucv_name, min(8,strlen(SNMP_IUCVNAME)));
        dpi_fd = socket(AF_IUCV, SOCK_STREAM, 0);
        sa = (struct sockaddr *) &siu;
        sasize = sizeof(struct sockaddr_iucv);
    } else {
#endif
        printf("Connecting to %s DPI_port %d (TCP, AF_INET)\n",
            dpi_hostname,dpi_port);
        bzero(&sin,sizeof(sin));
        sin.sin_family = AF_INET;

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 22 of 23)

```

        sin.sin_port      = htons(dpi_port);
        sin.sin_addr.s_addr = dpi_ipaddress;
        dpi_fd = socket(AF_INET, SOCK_STREAM, 0);
        sa      = (struct sockaddr *) &sin;
        sasize = sizeof(struct sockaddr_in);
#ifdef VM || defined (MVS)
    }
#endif
    if (dpi_fd < 0) {
        DO_ERROR("init_connection: socket");
        exit(1);
    }
    rc = connect(dpi_fd, sa, sasize);
    if (rc != 0) {
        DO_ERROR("init_connection: connect");
        close(dpi_fd);
        exit(1);
    }
}

#ifdef _NO_PROTO
static void dump_bfr(buf, len)
char *buf;
int len;
/* for classic K&R C */
/* hex dump buffer */
#else /* _NO_PROTO */
static void dump_bfr(const char *buf, const int len)
/* for ANSI-C compiler */
#endif /* _NO_PROTO */
{
    register int i;

    if (len == 0) printf("    empty buffer\n"); /* buffer is empty */
    for (i=0; i<len; i++) {
        if ((i&15) == 0) printf("    "); /* indent new line */
        printf("%2.2x", (unsigned char)buf[i]); /* hex print one byte */
        if ((i&15) == 15) printf("\n"); /* nl every 16 bytes */
        else if ((i&3) == 3) printf(" "); /* space every 4 bytes */
    }
    if (i&15) printf("\n"); /* always end with nl */
}

```

Figure 1. SNMP Dist Prog Interface subagent sample (Part 23 of 23)

Chapter 3. SNMP agent Distributed Protocol Interface version 2.0

The simple network management protocol (SNMP) agent Distributed Protocol Interface (DPI) permits you to dynamically add, delete, or replace management variables in the local management information base (MIB). The SNMP DPI protocol is also supported with the SNMP agent on OS/2®, VM, and AIX. This makes it easier to port subagents between those platforms and z/OS, as well as connect agents and subagents across these platforms.

The SNMP agent DPI Application Programming Interface (API) is for the DPI subagent programmer.

The following RFCs are related to SNMP and will be helpful when you are programming an SNMP API (see Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs):

- RFC 1592 is the SNMP DPI 2.0 RFC.
- RFC 1901 through RFC 1908 are the SNMP Version 2 RFCs.

The primary goal of RFC 1592 is to specify the SNMP DPI. This is a protocol by which subagents can exchange SNMP related information with an agent.

To provide an environment that is generally platform independent, RFC 1592 strongly suggests that you also define a DPI API. There is a sample DPI API available in the RFC. The document describes the same sample API as the IBM supported DPI Version 2.0 API. See “A DPI subagent example” on page 107.

SNMP agents and subagents

SNMP agents are primarily responsible for responding to SNMP operation requests. An operation request can originate from any entity that supports the management portion of the SNMP protocol. An example of this is z/OS UNIX SNMP command, `osnmp`, shipped with this version of TCP/IP. Examples of SNMP operations are GET, GETNEXT, and SET. An operation is performed on an MIB object.

A subagent extends the set of MIB objects provided by the SNMP agent. With the subagent, you define MIB objects useful in your own environment and register them with the SNMP agent.

When the agent receives a request for an MIB object, it passes the request to the subagent. The subagent then returns a response to the agent. The agent creates an SNMP response packet and sends the response to the remote network management station that initiated the request. The existence of the subagent is transparent to the network management station.

To allow the subagents to perform these functions, the agent provides for subagent connections through:

- A TCP connection
- An AF_UNIX streams connection

For the TCP connections, the agent binds to an arbitrarily chosen TCP port and listens for connection requests. A well-known port is not used. Every invocation of the SNMP agent could potentially use a different TCP port.

For UNIX streams connections, the agent is within the same machine. AF_UNIX connections should be used if possible, because they do not pass into TCP/IP, but flow only within UNIX System Services and hence require fewer system resources.

A DPI SNMP Subagent does not have to directly retrieve a dpiMIB object or objects, but instead uses either `DPIconnect_to_agent_TCP()` or `DPIconnect_to_agent_UNIXstream()`. `DPIconnect_to_agent_TCP` automatically retrieves the object `dpiPortForTCP` from the dpiMIB through an SNMP agent. `DPIconnect_to_agent_TCP` then establishes an AF_INET6 or AF_INET TCP socket connection with the SNMP agent.

The `query_DPI_port()` function issued in Version 1.1 is implicitly run by the `DPIconnect_to_agent_TCP()` function. The DPI subagent programmer would normally use the `DPIconnect_to_agent_TCP()` function to connect to the agent, and hence does not need to explicitly retrieve the value of the DPI TCP port.

Conversely, `DPIconnect_to_agent_UNIXstream` retrieves the value of the object `dpiPathNameForUnixStream` from the dpiMIB to establish an AF_UNIX connection with the SNMP agent.

After a successful connection to the SNMP agent the subagent registers the MIB trees for the set of variables it supports with the SNMP agent. When all variable classes are registered, the subagent waits for requests from the SNMP agent.

If connections to the SNMP agent are restricted by the security product, then the security product user ID associated with the subagent must be permitted to the agent's security product resource name for the connection to be accepted. See the Simple Network Management Protocol (SNMP) information in the information in the *z/OS Communications Server: IP Configuration Guide* for more information about security product access between subagents and the z/OS Communications Server SNMP agent.

DPI agent requests

The SNMP agent can initiate several DPI requests:

- CLOSE
- COMMIT
- GET
- GETBULK
- GETNEXT
- SET
- UNDO
- UNREGISTER

The GET, GETNEXT, and SET requests correspond to the SNMP requests that a network management station can make. The subagent responds to a request with a response packet. The response packet can be created using the `mkDPIresponse()` library routine, which is part of the DPI API library.

The GETBULK requests are translated into multiple GETNEXT requests by the agent. According to RFC 1592, a subagent can request that the GETBULK be

passed to it, but the z/OS version of DPI does not yet support that request. (See Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs.)

The COMMIT, UNDO, UNREGISTER, and CLOSE are specific SNMP DPI requests.

The subagent normally responds to a request with a RESPONSE packet. For the CLOSE and UNREGISTER request, the subagent does not need to send a RESPONSE.

Related information

- “GETNEXT processing” on page 53
- “UNREGISTER request” on page 55
- “TRAP request” on page 55
- “CLOSE request” on page 56
- “Overview of subagent processing” on page 107
- “Connecting to the agent” on page 109
- “Registering a subtree with the agent” on page 111
- “Processing requests from the agent” on page 113
- “Processing a GET request” on page 116
- “Processing a SET/COMMIT/UNDO request” on page 122

SNMP DPI version 2.0 library

z/OS Communications Server provides the following DPI library routines:

Table 1. Components of DPI version 2.0

Name	Contents	Location
snmp_dpi.h	header file	/usr/lpp/tcpip/snmp/include
snmp_IDPI.o snmp_mDPI.o snmp_qDPI.o	• z/OS UNIX System Services object files • DPI Version 2.0 library functions	/usr/lpp/tcpip/snmp/build/libdpi20
dpi_mvs_sample.c	SNMP DPI Version 2.0 C sample source	/usr/lpp/tcpip/samples
dpiSimpl.mi2	SNMP DPI Version 2.0 sample MIB definitions	/usr/lpp/tcpip/samples

SNMP DPI Version 2.0 API

DPI Version 2.0 is intended for use with UNIX System Services sockets and is not for use with other socket libraries. A DPI subagent must include the snmp_dpi.h header in any C part that intends to use DPI. The path for snmp_dpi.h is /usr/lpp/tcpip/snmp/include. By default, when you include the snmp_dpi.h include file, you will be exposed to the DPI Version 2.0 API. For a list of the functions provided, read more about the “The snmp_dpi.h include file” on page 106. This is the recommended use of the SNMP DPI API.

When you prelink your object code into an executable file, you must use the DPI Version 2.0 functions as provided in the snmp_IDPI.o, snmp_mDPI.o, and snmp_qDPI.o object files in /usr/lpp/tcpip/snmp/build/libdpi20.

Notes:

1. The object files are located only in a z/OS UNIX file system. Files in a z/OS UNIX file system can be accessed from JCL using the path parameter on an explicit DD definition.
2. Together the `snmp_dpi.h` include file and the `dpi_mvs_sample.c` file comprise an example of the DPI Version 2.0 API.
3. Debugging information (resulting from the `DPIdebug` function) is routed to `SYSLOGD`. Ensure the `SYSLOG` daemon is active.
4. Compile your subagent code using the `DEF(MVS)` compiler option.
5. Waiting for a DPI packet depends on the platform and how the chosen transport protocol is implemented. In addition, some subagents want to control the sending of and waiting for packets themselves, because they might need to be driven by other interrupts as well.
1. There is a set of DPI transport-related functions that are implemented on all platforms to hide the platform-dependent issues for those subagents that do not need detailed control for the transport themselves.

For more information about SNMP, see the Simple Network Management Protocol (SNMP) information in the information in the *z/OS Communications Server: IP Configuration Reference* or the Managing TCP/IP network resources with SNMP information in the *z/OS Communications Server: IP System Administrator's Commands*.

Compiling and linking

DPI Version 2.0 is installed in a z/OS UNIX file system only. You can build a subagent for either the UNIX System Services shell (using a z/OS UNIX file system and `c89`) or MVS (using `JCL`).

See the documentation provided by your C compiler for exact details of building a C application. The information provided in the following sections is intended as general guidance.

From a UNIX System Services environment

Use `c89` to compile a DPI subagent under the UNIX System Services shell. Every C file using DPI functions must include the DPI header file (`snmp_dpi.h`) from `/usr/lpp/tcpip/snmp/include`. Also include the three DPI library object files (`snmp_qDPI.o`, `snmp_lDPI.o`, and `snmp_mDPI.o`) from `/usr/lpp/tcpip/snmp/build/libdpi20`.

The following is an example of how `c89` is called to compile and build `dpi_mvs_sample.c`:

```
c89 -o dpi_mvs_sample -I /usr/lpp/tcpip/snmp/include \  
/usr/lpp/tcpip/samples/dpi_mvs_sample.c \  
/usr/lpp/tcpip/snmp/build/libdpi20/snmp_lDPI.o \  
/usr/lpp/tcpip/snmp/build/libdpi20/snmp_mDPI.o \  
/usr/lpp/tcpip/snmp/build/libdpi20/snmp_qDPI.o
```

Use the `-I` option to add the z/OS UNIX file system directory where `snmp_dpi.h` resides to the compiler include search path.

From an MVS environment

C programs that use DPI must:

- Compile with the `longname` compiler option

- Include `snmp_dpi.h` from `/usr/lpp/tcpip/snmp/include`

Add `#include` to the source code. You must inform the compiler that `/usr/lpp/tcpip/snmp/include` should be searched for include files. Use either a SYSLIB DD with a PATH parameter pointing to the z/OS UNIX file system directory, or use the SEARCH compiler parameter.

Prelink DPI subagent to resolve longnames. In the prelink JCL, define three DDs pointing to each DPI object file, and then include each, such as:

```

DPI1 DD PATH='/usr/lpp/tcpip/snmp/build/libdpi20/snmp_1DPI.o'
DPI2 DD PATH='/usr/lpp/tcpip/snmp/build/libdpi20/snmp_mDPI.o'
DPI3 DD PATH='/usr/lpp/tcpip/snmp/build/libdpi20/snmp_qDPI.o'

INCLUDE DPI1
INCLUDE DPI2
INCLUDE DPI3

```

Then, link edit the prelink output as usual.

DPI Version 1.x base code considerations

Use the DPI Version 1.1 API as described in Chapter 2, “SNMP agent Distributed Protocol Interface version 1.1,” on page 3.

The DPI Version 2.0 API provided with z/OS is for UNIX System Services sockets use only. Earlier versions of DPI were supported on C sockets.

See “Migrating your SNMP DPI subagent to Version 2.0” for more detail about the changes that you must make to your DPI Version 1.x source.

If you want to convert to DPI Version 2.0, which prepares you also for SNMP Version 2, you must make changes to your code.

You can keep your existing DPI Version 1.1 subagent and communicate with a DPI-capable agent that supports DPI Version 1.1 in addition to DPI Version 2.0. For example, the z/OS SNMP agent provides support for multiple versions of DPI, including Version 1.0, Version 1.1, and Version 2.0.

SNMP DPI API Version 1.1 considerations

The information presented in this section *must be understood as guidelines and not exact procedures*. Your specific implementation will vary from the guidelines presented.

Migrating your SNMP DPI subagent to Version 2.0

When you want to change your DPI Version 1.x-based subagent code to the DPI Version 2.0 level, use these guidelines for the required actions and the recommended actions.

Required actions

The following actions are required to migrate SNMP DPI subagent to Version 2.0:

- Add an `mkDPIopen()` call and send the created packet to the agent. This opens your DPI connection with the agent. Wait for the response and ensure that the open is accepted. You need to pass a subagent ID (object identifier), which must be a unique ASN.1 OID.

See “The `mkDPIopen()` function” on page 65 for more information.

- Change your `mkDPRegister()` calls and pass the parameters according to the new function prototype. You must also expect a RESPONSE to the REGISTER request.
See “The `mkDPRegister()` function” on page 67 for more information.
- Change `mkDPSet()` and `mkDPList()` calls to the new `mkDPSet()` call. Basically all `mkDPSet()` calls are now of the DPI Version 1.1 `mkDPList()` form.
See “The `mkDPSet()` function” on page 71 for more information.
- Change `mkDPTrap()` and `mkDPTrap()` calls to the new `mkDPTrap()` call. Basically all `mkDPTrap()` calls are now of the DPI Version 1.1 `mkDPTrap()` form.
See “The `mkDPTrap()` function” on page 73 for more information.
- Add code to recognize DPI RESPONSE packets, which should be expected as a result of OPEN, REGISTER, and UNREGISTER requests.
- Add code to expect and handle the DPI UNREGISTER packet from the agent. It might send such packets if an error occurs or if a higher priority subagent registers the same subtree as you have registered.
- Add code to unregister your subtrees and close the DPI connection when you want to terminate the subagent.
See “The `mkDPUnregister()` function” on page 75 and “The `mkDPIClose()` function” on page 64 for more information.
- Change your code to use the new SNMP Version 2 error codes as defined in the `snmp_dpi.h` include file.
- When migrating DPI Version 1.1 subagents to DPI Version 2.0, remove the include for `manifest.h`.
- Change your code that handles a GET request. It should return a `varBind` with `SNMP_TYPE_noSuchObject` value or `SNMP_TYPE_noSuchInstance` value instead of an error `SNMP_ERROR_noSuchName` if the object or the instance do not exist. This is not considered an error any more. Therefore, you should return an `SNMP_ERROR_noError` with an error index of 0.

Note: A `varBind` (variable binding) is the group ID, instance ID, type, length, and value that completely describes a variable in the MIB.
- Change your code that handles a GETNEXT request. It should return a `varBind` with `SNMP_TYPE_endOfMibView` value instead of an error `SNMP_ERROR_noSuchName` if you reach the end of your MIB or subtree. This is not considered an error any more. Therefore, you should return an `SNMP_ERROR_noError` with an error index of 0.
- Change your code that handles SET requests to follow the two-phase SET/COMMIT scheme as described in “SET processing” on page 52.
See the sample handling of SET/COMMIT/UNDO in “Processing a SET/COMMIT/UNDO request” on page 122.

Recommended actions

The following actions are recommended:

- Do not refer to the object ID pointer (`object_p`) in the `snmp_dpi_xxxx_packet` structures any more. Instead start using the `group_p` and `instance_p` pointers. The `object_p` pointer might be removed in a future version of the DPI API.
- Check “Transport-related DPI API functions” on page 77 to see if you want to use those functions instead of using your own code for those functions.

- Consider using more than one varBind per DPI packet. You can specify this on the REGISTER request. You must then be prepared to handle multiple varBinds per DPI packet. The varBinds are chained through the various snmp_dpi_xxxx_packet structures.
See “The mkDPIopen() function” on page 65 for more information.
- Consider specifying a timeout when you issue a DPI OPEN or DPI REGISTER.
See “The mkDPIopen() function” on page 65 and “The mkDPIregister() function” on page 67 for more information.
- Ensure SYSLOGD is active. The result of using DPIdebug is routed to SYSLOGD. For information on how to configure SYSLOGD, see the Syslog daemon information in the *z/OS Communications Server: IP Configuration Reference*.

DPI Version 2.0 recognizes mkDPIlist; however, Version 2.0 subagents should use mkDPIset instead.

Name changes

A number of field names in the snmp_dpi_xxxx_packet structures have changed so that the names are now more consistent throughout the DPI code.

The new names indicate if the value is a pointer (_p) or a union (_u). The names that have changed and that affect the subagent code are listed in the table below.

Old name	New name	Data structure (XXXX)
group_id	group_p	getnext
object_id	object_p	get, getnext, set
value	value_p	set
type	value_type	set
next	next_p	set
enterprise	enterprise_p	trap
packet_body	data_u	dpi_hdr
dpi_get	get_p	hdr (packet_body)
dpi_getnext	next_p	hdr (packet_body)
dpi_set	set_p	hdr (packet_body)
dpi_trap	trap_p	hdr (packet_body)

There is no clean approach to make this change transparent. You probably will need to change the names in your code. You could try a simple set of defines like:

```
#define packet_body    data_u
#define dpi_get        get_p
#define dpi_set        set_p
#define dpi_next       next_p
#define dpi_response   resp_p
#define dpi_trap       trap_p
#define group_id       group_p
#define object_id      object_p
#define value          value_p
#define type           value_type
#define next           next_p
#define enterprise     enterprise_p
```

If the names conflict with other definitions, change your code.

SNMP DPI environment variables

Table 2 provides a list of environment variables for the SNMP DPI.

Table 2. Environment variables for the SNMP DPI

Environment variable	Description
SNMP_PORT	Specifies the port to which a DPI subagent will direct a connection query. This variable defaults to 161, which is the default port on which the SNMP agent listens for queries.

Subagent programming concepts

When implementing a subagent, use the DPI Version 2 approach and keep the following in mind:

- Use the SNMP Version 2 error codes only, even though there are definitions for the SNMP Version 1 error codes.
- Implement the SET, COMMIT, UNDO processing properly.
- Use the SNMP Version 2 approach for GET requests, and pass back noSuchInstance value or noSuchObject value if appropriate. Continue to process all remaining varBinds.

More than one varBind can be specified in the SNMP PDU for the requested operation. For example, using the SNMP network manager, a user can request the retrieval of multiple objects in the same request (GET or GETNEXT). The varBind portion of the PDU sent would include multiple object identifiers (OIDs). The subagent limitations are passed to the agent through the max_varBinds parameter on the mkDPIopen call. When the subagent receives a request from the agent, it needs to handle multiple OIDs per request if it specified a max_varBinds value other than 1.

- Use the SNMP Version 2 approach for GETNEXT, and pass back endOfMibView value if appropriate. Continue to process all remaining varBinds.
- Specify the timeout period in the OPEN and REGISTER packets, when you are processing a request from the agent (GET, GETNEXT, SET, COMMIT, or UNDO).

If you fail to respond within the timeout period, the agent will probably close your DPI connection and discard your RESPONSE packet if it comes in later. If you can detect that the response is not going to be received in the time period, then you might decide to stop the request and return an SNMP_ERROR_genErr in the RESPONSE.

- Issue an SNMP DPI ARE_YOU_THERE request periodically to ensure that the agent is still connected and still knows about you.
- OS/2 runs on an ASCII based machine. However, when you are running a subagent on an EBCDIC based machine and you use the (default) native character set, all OID strings and all variable values of type OBJECT_IDENTIFIER or DisplayString objects that are known by the agent (in its compiled MIB) will be passed to you in EBCDIC format. OID strings include the group ID, instance ID, enterprise ID, and subagent ID. You should structure your response with the EBCDIC format.
- If you receive an error RESPONSE on the OPEN packet, you will also receive a DPI CLOSE packet with an SNMP_CLOSE_openError code. In this situation, the agent closes the connection.
- The DisplayString is only a textual convention. In the SNMP PDU (SNMP packet), the type is an OCTET_STRING.

When the type is OCTET_STRING, it is not clear if this is a DisplayString or any arbitrary data. This means that the agent can only know about an object being a DisplayString if the object is included in some sort of a compiled MIB. If it is, the agent will use SNMP_TYPE_DisplayString in the type field of the varBind in a DPI SET packet. When you send a DisplayString in a RESPONSE packet, the agent will handle it as such.

Related information

“A DPI subagent example” on page 107

Specifying the SNMP DPI API

The following sections describe each type of DPI processing in this order:

- Connect processing
- OPEN request
- REGISTER request
- GET, SET, GETNEXT, GETBULK, TRAP, and ARE_YOU_THERE processing
- UNREGISTER request
- CLOSE request

Connect processing

There are various connect functions that allow connections through either TCP or UNIXstream. Determine which is appropriate for you by evaluating whether you are connecting to the same machine or a different machine. If the agent and the subagent are using the same machine, use the UNIXstream connection for better performance. If the agent and the subagent are using different machines, you must use the TCP connection. There are two connect processing parameters:

- Hostname—name or the IP address of the agent
- Community name—password that allows the DPI connect function to obtain the port (for TCP) or pathname (for UNIX) that allows the socket connect to occur.

Related information

“Connecting to the agent” on page 109

OPEN request

Next, the DPI subagent must open a connection with the agent. To do so, it must send a DPI OPEN packet in which these parameters must be specified:

- The maximum timeout value in seconds. The agent is requested to wait this long for a response to any request for an object being handled by this subagent.

The agent can have an absolute maximum timeout value which will be used if the subagent asks for too large a timeout value. A value of 0 can be used to indicate that the agent default timeout value should be used. A subagent is advised to use a reasonably short interval of a few seconds or so. If a specific subtree needs longer time, a specific REGISTER can be done for that subtree with a longer timeout value.

- The maximum number of varBinds that the subagent is prepared to handle per DPI packet. Specifying 1 would result in DPI Version 1 behavior of one varBind per DPI packet that the agent sends to the subagent. A value of 0 means the agent will try to combine up to as many varBinds as are present in the SNMP packet that belongs to the same subtree.

- The character set you want to use. The default 0 value is the native character set of the machine platform where the agent runs. Because the subagent and agent normally run on the same system or platform, use the native character set, which is EBCDIC on MVS.

If your platform is EBCDIC-based, using the native character set of EBCDIC makes it easy to recognize the string representations of the fields, such as the group ID and instance ID. At the same time, the agent translates the value from ASCII NVT to EBCDIC and vice versa for objects that it knows from a compiled MIB to have a textual convention of DisplayString. This fact cannot be determined from the SNMP PDU encoding because, in the PDU, the object is only known to be an OCTET_STRING.

If your subagent runs on an ASCII-based platform and the agent runs on an EBCDIC-based platform (or the other way around), you can specify that you want to use the ASCII character set. The agent and subagent programmers know how to handle the string-based data in this situation.

- The subagent ID. This is an ASN.1 object identifier that uniquely identifies the subagent. This OID is represented as a null-terminated string using the selected character set.

For example: 1.3.5.1.2.3.4.5

- The subagent description. This is a DisplayString describing the subagent. This is a character string using the selected character set.

For an example see “A DPI subagent example” on page 107.

After a subagent has sent a DPI OPEN packet to an agent, it should expect a DPI RESPONSE packet that informs the subagent about the result of the request. The packet ID of the RESPONSE packet should be the same as that of the OPEN request to which the RESPONSE packet is the response. See “DPI RESPONSE error codes” on page 102 for a list of valid codes that can be expected.

If you receive an error RESPONSE on the OPEN packet, you will also receive a DPI CLOSE packet with an SNMP_CLOSE_openError code. In this situation, the agent closes the connection.

If the OPEN is accepted, the next step is to REGISTER one or more MIB subtrees.

Related information

“Connecting to the agent” on page 109

REGISTER request

Before a subagent will receive any requests for MIB objects, it must first register the variables or subtree it supports with the SNMP agent. The subagent must specify the following parameters in the REGISTER request:

- The subtree to be registered.

Object level registration: This is a null-terminated string in the selected character set specifying the subtree to be registered. Object level registration requires a trailing period following the object number, indicating a register request to support all instances of an object (for example, ifDescr). Object level registration requires the subtree must have a trailing period. For example: 1.3.6.1.2.1.2.2.1.2.

Instance level registration: Instance level registration does not require a trailing period for the subtree. Instance level registration can be used to allow different subagents to support separate instances of a particular MIB object. Registration by subagents at the instance level rather than the object level is accomplished by

simply adding the instance number after the object number when building the registration packet using the `mkDPIregister` call. For example, passing an object number such as 1.3.6.1.2.1.2.2.1.2. (*note the ending period*) would support all instances of `ifDescr`. However, a subagent could pass an object or instance number like 1.3.6.1.2.1.2.2.1.2.8 (*note the addition of the 8 after the period*) to support only `ifDescr.8` (instance 8).

- The requested priority for the registration. The values are:
 - 1 Request for the best available priority
 - 0 Request for the next best available priority than the highest (best) priority currently registered for this subtree
 - NNN Any other positive value requests a specific priority, if available, or the next best priority that is available.
- The maximum timeout value in seconds. The agent is requested to wait this long for a response to any request for an object in this subtree. The agent can have an absolute maximum timeout value which will be used if the subagents ask for too large a timeout value. A value of 0 can be used to indicate that the DPI OPEN value should be used for timeout.

After a subagent has sent a DPI REGISTER packet to the agent, it should expect a DPI RESPONSE packet that informs the subagent about the result of the request. The packet ID of the RESPONSE packet should be the same as that of the REGISTER packet to which the RESPONSE packet is the response.

If the response is successful, the `error_index` field in the RESPONSE packet contains the priority that the agent assigned to the subtree registration. See “DPI RESPONSE error codes” on page 102 for a list of valid codes that can be expected.

Error Code: `higherPriorityRegistered`: The response to a REGISTER request might return the error code `higherPriorityRegistered`. This might be caused by one of the following:

- Another subagent already registered the same subtree at a better priority than what you are requesting.
- Another subagent already registered a subtree at a higher level (at any priority). For instance, if a registration already exists for subtree 1.2.3.4.5.6 and you try to register for subtree 1.2.3.4.5.6.<anything> then you will get `higherPriorityRegistered` error code.

If you receive this error code, your subtree will be registered, but you will not see any requests for the subtree. They will be passed to the subagent that registered with a better priority. If you stay connected, and the other subagent goes away, you will get control over the subtree at that point in time.

Related information

“Registering a subtree with the agent” on page 111

GET processing

The DPI GET packet holds one or more `varBinds` that the subagent has taken responsibility for.

If the subagent encounters an error while processing the request, it creates a DPI RESPONSE packet with an appropriate error indication in the `error_code` field and sets the `error_index` to the position of the `varBind` at which the error occurs. The first `varBind` is index 1, the second `varBind` is index 2, and so on. No name, type,

length, or value information needs to be provided in the packet because, by definition, the varBind information is the same as in the request to which this is a response and the agent still has that information.

If there are no errors, the subagent creates a DPI RESPONSE packet in which the error_code is set to SNMP_ERROR_noError (0) and error_index is set to 0. The packet must also include the name, type, length, and value of each varBind requested.

When you get a request for a nonexistent object or a nonexistent instance of an object, you must return a NULL value with a type of SNMP_TYPE_noSuchObject or SNMP_TYPE_noSuchInstance respectively. These two values are not considered errors, so the error_code and error_index should be 0.

The DPI RESPONSE packet is then sent back to the agent.

Related information

“Processing a GET request” on page 116

“The mkDPIresponse() function” on page 69

SET processing

A DPI SET packet contains the name, type, length, and value of each varBind requested, plus the value type, value length, and value to be set.

If the subagent encounters an error while processing the request, it creates a DPI RESPONSE packet with an appropriate error indication in the error_code field and an error_index listing the position of the varBind at which the error occurs. The first varBind is index 1, the second varBind is index 2, and so on. No name, type, length, or value information needs to be provided in the packet because, by definition, the varBind information is the same as in the request to which this is a response and the agent still has that information.

If there are no errors, the subagent creates a DPI RESPONSE packet in which the error_code is set to SNMP_ERROR_noError (0) and error_index is set to 0. No name, type, length, or value information is needed because the RESPONSE to a SET should contain exactly the same varBind data as the data present in the request. The agent can use the values it already has.

This suggests that the agent must keep state information, and that is the case. It needs to do that anyway to be able to later pass the data with a DPI COMMIT or DPI UNDO packet. Because there are no errors, the subagent must have allocated the required resources and prepared itself for the SET. It does not yet carry out the SET, which will be done at COMMIT time.

The subagent sends a DPI RESPONSE packet, indicating success or failure for the preparation phase, back to the agent. The agent will issue a SET request for all other varBinds in the same original SNMP request it received. This can be to the same subagent or to one or more different subagents.

After all SET requests have returned a “no error” condition, the agent starts sending DPI COMMIT packets to the subagents. If any SET request returns an error, the agent sends DPI UNDO packets to those subagents that indicated successful processing of the SET preparation phase.

When the subagent receives the DPI COMMIT packet, all the varBind information will again be available in the packet. The subagent can now carry out the SET request.

If the subagent encounters an error while processing the COMMIT request, it creates a DPI RESPONSE packet with value `SNMP_ERROR_commitFailed` in the `error_code` field and an `error_index` that lists at which varBind the error occurs. The first varBind is index 1, the second varBind is 2, and so on. No name, type, length, or value information is needed. The fact that a `commitFailed` error exists does not mean that this error should be returned easily. A subagent should do all that is possible to make a COMMIT succeed.

If there are no errors and the SET and COMMIT have been carried out with success, the subagent creates a DPI RESPONSE packet in which the `error_code` is set to `SNMP_ERROR_noError` (0) and `error_index` is set to 0. No name, type, length, or value information is needed.

So far discussion has focused on successful SET and COMMIT sequences. However, after a successful SET, the subagent might receive a DPI UNDO packet. The subagent must now undo any preparations it made during the SET processing, such as free allocated memory.

Even after a COMMIT, a subagent might still receive a DPI UNDO packet. This occurs if some other subagent could not complete a COMMIT request. Because of the SNMP requirement that all varBinds in a single SNMP SET request must be changed *as if simultaneous*, all committed changes must be undone if any of the COMMIT requests fail. In this case the subagent must try and undo the committed SET operation.

If the subagent encounters an error while processing the UNDO request, it creates a DPI RESPONSE packet with value `SNMP_ERROR_undoFailed` in the `error_code` field and an `error_index` that lists at which varBind the error occurs. The first varBind is index 1, the second varBind is 2, and so on. No name, type, length, or value information is needed. The fact that an `undoFailed` error exists does not mean that this error should be returned easily. A subagent should do all that is possible to make an UNDO succeed.

If there are no errors and the UNDO has been successful, the subagent creates a DPI RESPONSE packet in which the `error_code` is set to `SNMP_ERROR_noError` (0) and `error_index` is set to 0. No name, type, length, or value information is needed.

Related information

“Processing a SET/COMMIT/UNDO request” on page 122

GETNEXT processing

The DPI GETNEXT packet contains the objects on which the GETNEXT operation must be performed. For this operation, the subagent is to return the name, type, length, and value of the next variable it supports whose (ASN.1) name lexicographically follows the one passed in the group ID (subtree) and instance ID.

In this case, the instance ID might not be present (NULL) in the incoming DPI packet, implying that the NEXT object must be the first instance of the first object in the subtree that was registered.

It is important to realize that a given subagent might support several discontinuous sections of the MIB tree. In that situation, it would be incorrect to jump from one section to another. This problem is correctly handled by examining the group ID in the DPI packet. This group ID represents the reason why the subagent is being called. It holds the prefix of the tree that the subagent had indicated it supported (registered).

If the next variable supported by the subagent does not begin with that prefix, the subagent must return the same object instance as in the request, for example the group ID and instance ID with a value of `SNMP_TYPE_endOfMibView` (implied NULL value). This `endOfMibView` is not considered an error, so the `error_code` and `error_index` should be 0. If required, the SNMP agent will call upon the subagent again, but pass it a different group ID (prefix). This is illustrated in the discussion below.

Assume there are two subagents. The first subagent registers two distinct sections of the tree: A and C. In reality, the subagent supports variables A.1 and A.2, but it correctly registers the minimal prefix required to uniquely identify the variable class it supports.

The second subagent registers section B, which appears between the two sections registered by the first agent.

If a management station begins browsing the MIB, starting from A, the following sequence of queries of the form GET-NEXT (group ID, instance ID) would be performed:

```
Subagent 1 gets called:
  get-next(A,none) = A.1
  get-next(A,1)   = A.2
  get-next(A,2)   = endOfMibView
```

```
Subagent 2 is then called:
  get-next(B,none) = B.1
  get-next(B,1)   = endOfMibView
```

```
Subagent 1 gets called again:
  get-next(C,none) = C.1
```

Related information

None

GETBULK processing request

You must ask the agent to translate GETBULK requests into multiple GETNEXT requests. This is basically the default and is specified in the DPI REGISTER packet. The majority of DPI subagents will run on the same machine as the agent, or on the same physical network. Therefore, repetitive GETNEXT requests remain local, and, in general, should not be a problem.

Note: Currently, z/OS SNMP does not support GETBULK protocol between agent and subagent. These requests are translated into multiple GETNEXT requests.

Related information

“GETNEXT processing” on page 53

TRAP request

A subagent can request that the SNMP agent generates a trap. The subagent must provide the desired values for the generic and specific parameters of the trap. It can optionally provide a set of one or more name, type, length, or value parameters that will be included in the trap packet.

It can optionally specify an enterprise ID (object identifier) for the trap to be generated. If a NULL value is specified for the enterprise ID, the agent will use the subagent identifier from the DPI OPEN packet as the enterprise ID to be sent with the trap.

Related information

“Generating a TRAP” on page 126

ARE_YOU_THERE request

A subagent can send an ARE_YOU_THERE packet to the agent. If the connection is in a healthy state, the agent responds with a RESPONSE packet with SNMP_ERROR_DPI_noError. If the connection is not in a healthy state, the agent might respond with a RESPONSE packet with an error indication, but the agent might not react at all. In this situation, you would time out while waiting for a response.

UNREGISTER request

A subagent can unregister a previously registered subtree. The subagent must specify the following parameters in the UNREGISTER request:

- The subtree to be unregistered.

Object level unregistration: This is a null-terminated string in the selected character set specifying the subtree to be unregistered. Object level unregistration requires a trailing period following the object number, indicating an unregister request to all supported instances of an object (for example, ifDescr). Object level unregistration requires the subtree must have a trailing period. For example: 1.3.6.1.2.1.2.2.1.2.

Instance level unregistration: Instance level unregistration does not require a trailing period for the subtree.

Note: Unregistration at the instance level can only be done if the original registration was done using instance level registration.

Unregistration by subagent at the instance level rather than the object level is accomplished by simply adding the instance number after the object number when building the unregistration packet using the mkDPIunregister call. For example, passing an object number such as 1.3.6.1.2.1.2.2.1.2. (*note the ending period*) would support all instances of ifDescr. However, a subagent could pass an object or instance number like 1.3.6.1.2.1.2.2.1.2.8 (*note the addition of the 8 after the period*) to support only ifDescr.8 (instance 8).

- The reason for the unregister. See “DPI UNREGISTER reason codes” on page 103 for a list of valid reason codes.

After a subagent has sent a DPI UNREGISTER packet to the agent, it should expect a DPI RESPONSE packet that informs the subagent about the result of the request. The packet ID of the RESPONSE packet should be the same as that of the REGISTER packet to which the RESPONSE packet is the response. See “DPI RESPONSE error codes” on page 102 for a list of valid codes that can be expected.

A subagent should also be prepared to handle incoming DPI UNREGISTER packets from the agent. In this situation, the DPI packet will contain a reason code for the UNREGISTER. A subagent does not have to send a response to an UNREGISTER request. The agent assumes that the subagent will handle it appropriately. The registration is removed regardless of what the subagent returns.

Related information

“Processing an UNREGISTER request” on page 125

CLOSE request

When a subagent is finished and wants to end processing, it should first UNREGISTER its subtrees and then close the connection with the agent. To do so, it must send a DPI CLOSE packet, which specifies a reason for the closing. See “DPI CLOSE reason codes” on page 102 for a list of valid codes. You should not expect a response to the CLOSE request.

A subagent should also be prepared to handle an incoming DPI CLOSE packet from the agent. In this case, the packet will contain a reason code for the CLOSE request. A subagent does not have to send a response to a CLOSE request. The agent assumes that the subagent will handle it appropriately. The close takes place regardless of what the subagent does with it.

Related information

“Processing a CLOSE request” on page 126

Multithreading programming considerations

The DPI Version 2.0 program does not support multithreaded subagents.

There are several static buffers in the DPI code. For compatibility reasons, that cannot be changed. Real multithread support will probably mean several potentially incompatible changes to the DPI Version 2.0 API.

Use a locking mechanism: Because the DPI API is not reentrant, to use your subagent in a multithreaded process you should use some locking mechanism of your own around the static buffers. Otherwise, one thread might be writing into the static buffer while another is writing into the same buffer at the same time. There are two static buffers. One buffer is for building the serialized DPI packet before sending it out and the other buffer is for receiving incoming DPI packets.

Basically, all DPI functions that return a pointer to an unsigned character are the DPI functions that write into the static buffer to create a serialized DPI packet:

```
mkDPIAreYouThere()
mkDPIopen()
mkDPIregister()
mkDPIunregister()
mkDPItrap()
mkDPIresponse()
mkDPIpacket()
mkDPIclose ()
```

After you have called the `DPIsend_packet_to_agent()` function for the buffer, which is pointed to by the pointer returned by one of the preceding functions, the buffer is free to use again.

There is one function that reads the static input buffer:

```
pDPIpacket()
```

The input buffer gets filled by the `DPIawait_packet_from_agent()` function. Upon return from the await, you receive a pointer to the static input buffer. The `pDPIpacket()` function parses the static input buffer and returns a pointer to dynamically allocated memory. Therefore, after the `pDPIpacket()` call the buffer is available for use again.

The DPI internal handle structures and control blocks used by the underlying code to send and receive data to and from the agent are also static data areas. Ensure that you use your own locking mechanism around the functions that add, change, or delete data in those static structures. The functions that change those internal static structures are:

```
DPIconnect_to_agent_TCP()           /* everyone has this one    */
DPIconnect_to_agent_UNIXstream()    /* supported */
DPIdisconnect_from_agent()          /* everyone has this one    */
```

Other functions will access the static structures. These other functions must be assured that the structure is not being changed while they are referencing it during their execution. The other functions are:

```
DPIawait_packet_from_agent()
DPIsend_packet_to_agent()
DPIget_fd_for_handle()
```

While the last three functions can be executed concurrently in different threads, you must ensure that no other thread is adding or deleting handles in these static structures during this process.

Functions, data structures, and constants

Use these lists to locate the descriptions for the functions, data structures, and constants.

Basic DPI Functions:

- “The `DPIdebug()` function” on page 59
- “The `DPI_PACKET_LEN()` macro” on page 60
- “The `fDPIparse()` function” on page 61
- “The `fDPIset()` function” on page 62
- “The `mkDPIAreYouThere()` function” on page 63
- “The `mkDPIclose()` function” on page 64
- “The `mkDPIopen()` function” on page 65
- “The `mkDPIregister()` function” on page 67
- “The `mkDPIresponse()` function” on page 69
- “The `mkDPIset()` function” on page 71
- “The `mkDPItrap()` function” on page 73
- “The `mkDPIunregister()` function” on page 75
- “The `pDPIpacket()` function” on page 76

DPI Transport-Related Functions:

- “The `DPIawait_packet_from_agent()` function” on page 78
- “The `DPIconnect_to_agent_TCP()` function” on page 80
- “The `DPIconnect_to_agent_UNIXstream()` function” on page 82
- “The `DPIdisconnect_from_agent()` function” on page 84
- “The `DPIget_fd_for_handle()` function” on page 85
- “The `DPIsend_packet_to_agent()` function” on page 86
- “The `lookup_host()` function” on page 88
- “The `lookup_host6()` function” on page 89

Data Structures:

- “The snmp_dpi_close_packet structure” on page 91
- “The snmp_dpi_get_packet structure” on page 92
- “The snmp_dpi_hdr structure” on page 93
- “The snmp_dpi_next_packet structure” on page 95
- “The snmp_dpi_resp_packet structure” on page 96
- “The snmp_dpi_set_packet structure” on page 97
- “The snmp_dpi_ureg_packet structure” on page 99
- “The snmp_dpi_u64 structure” on page 100

Constants and Values:

- “DPI CLOSE reason codes” on page 102
- “DPI packet types” on page 102
- “DPI RESPONSE error codes” on page 102
- “DPI UNREGISTER reason codes” on page 103
- “DPI SNMP value types” on page 103
- “Value representation” on page 104

Related Information:

- “Character set selection” on page 101
- “The snmp_dpi.h include file” on page 106

Basic DPI API functions

This section describes each of the basic DPI functions that are available to the DPI subagent programmer.

The Basic DPI Functions are:

- “The DPIddebug() function” on page 59
- “The DPI_PACKET_LEN() macro” on page 60
- “The fDPIparse() function” on page 61
- “The fDPIset() function” on page 62
- “The mkDPIAreYouThere() function” on page 63
- “The mkDPIclose() function” on page 64
- “The mkDPIopen() function” on page 65
- “The mkDPIregister() function” on page 67
- “The mkDPIresponse() function” on page 69
- “The mkDPIset() function” on page 71
- “The mkDPItrap() function” on page 73
- “The mkDPIunregister() function” on page 75
- “The pDPIpacket() function” on page 76

The DPIDebug() function

Format

```
#include <snmp_dpi.h>

void DPIDebug(int level);
```

Parameters

level If this value is 0, tracing is turned off. If it has any other value, tracing is turned on at the specified level. The higher the value, the more detail. A higher level includes all lower levels of tracing. Currently there are two levels of detail:

- 1 Display packet creation and parsing.
- 2 Display hex dump of incoming and outgoing DPI packets.

Usage

The DPIDebug() function turns DPI internal debugging or tracing on or off.

The trace output is sent to the SYSLOG Daemon. See the IBM 3172 Enterprise-specific MIB variables information in *z/OS Communications Server: IP System Administrator's Commands* for more information.

Examples

```
#include <snmp_dpi.h>

DPIDebug(2);
```

Context

“The snmp_dpi.h include file” on page 106

The DPI_PACKET_LEN() macro

Format

```
#include <snmp_dpi.h>

int DPI_PACKET_LEN(unsigned char *packet_p)
```

Parameters

packet_p
A pointer to a serialized DPI packet

Return codes

An integer representing the total DPI packet length

Usage

The DPI_PACKET_LEN macro generates C code that returns an integer representing the length of a DPI packet. It uses the first two octets in network byte order of the packet to calculate the length.

Examples

```
#include <snmp_dpi.h>
unsigned char *pack_p;
int          length;

pack_p = mkDPIClose(SNMP_CLOSE_goingDown);
if (pack_p) {
    length = DPI_PACKET_LEN(pack_p);
    /* send packet to agent */
} /* endif */
```


The fDPIparse() function

Format

```
#include <snmp_dpi.h>

void fDPIparse(snmpp_dpi_hdr *hdr_p);
```

Parameters

hdr_p A pointer to the parse tree. The parse tree is represented by an snmp_dpi_hdr structure.

Usage

The fDPIparse() function frees a parse tree that was previously created by a call to pDPIpacket(). The parse tree might have been created in other ways too. After calling fDPIparse(), no further references to the parse tree can be made.

A complete or partial DPI parse tree is also implicitly freed by a call to a DPI function that serializes a parse tree into a DPI packet. The section that describes each function tells you if this is the case. An example of such a function is mkDPIresponse().

Examples

```
#include <snmp_dpi.h>
snmp_dpi_hdr *hdr_p;
unsigned char *pack_p;          /* assume pack_p points to */
                                /* incoming DPI packet      */

hdr_p = pDPIpacket(pack_p);

/* handle the packet and when done do the following */
if (hdr_p) fDPIparse(hdr_p);
```

Context

“The snmp_dpi_hdr structure” on page 93

“The pDPIpacket() function” on page 76

“The snmp_dpi.h include file” on page 106

The fDPISet() function

Format

```
#include <snmp_dpi.h>

void fDPISet(snmp_dpi_set_packet *packet_p);
```

Parameters

packet_p

A pointer to the first snmp_dpi_set_packet structure in a chain of such structures.

Usage

The fDPISet() function is typically used if you must free a chain of one or more snmp_dpi_set_packet structures. This might be the case if you are in the middle of preparing a chain of such structures for a DPI RESPONSE packet, but then run into an error before you can actually make the response.

If you get to the point where you make a DPI response packet to which you pass the chain of snmp_dpi_set_packet structures, the mkDPISet() function will free the chain of snmp_dpi_set_packet structures.

Examples

```
#include <snmp_dpi.h>
unsigned char    *pack_p;
snmp_dpi_hdr     *hdr_p;
snmp_dpi_set_packet *set_p, *first_p;
long int         num1 = 0, num2 = 0;

hdr_p = pDPISet(pack_p);          /* assume pack_p    */
/* analyze packet and assume all OK */ /* points to the    */
/* now prepare response; 2 varBinds */ /* incoming packet */

set_p = mkDPISet(snmp_dpi_NULL_p, /* create first one */
                 "1.3.6.1.2.3.4.5.", "1.0", /* OID=1, instance=0 */
                 SNMP_TYPE_Integer32,
                 sizeof(num1), &num1);
if (set_p) {                      /* if success, then */
    first_p = set_p;              /* save ptr to first */
    set_p = mkDPISet(set_p,       /* chain next one    */
                     "1.3.6.1.2.3.4.5.", "1.1", /* OID=1, instance=1 */
                     SNMP_TYPE_Integer32,
                     sizeof(num2), &num2);
    if (set_p) {                  /* success 2nd one   */
        pack_p = mkDPISet(hdr_p, /* make response     */
                           SNMP_ERROR_noError, /* It will also free */
                           0L, first_p); /* the set_p tree    */
        /* send DPI response to agent */
    } else {                      /* 2nd mkDPISet fail */
        fDPISet(first_p);        /* must free chain   */
    } /* endif */
} /* endif */
```

Context

“The fDPISet() function” on page 61

“The snmp_dpi_set_packet structure” on page 97

“The mkDPISet() function” on page 69

The mkDPIAreYouThere() function

Format

```
#include <snmp_dpi.h>

unsigned char *mkDPIAreYouThere(void);
```

Parameters

None

Return codes

If successful, a pointer to a static DPI packet buffer is returned. The first 2 bytes of the buffer in network byte order contain the length of the remaining packet. The macro `DPI_PACKET_LEN` can be used to calculate the total length of the DPI packet.
If not successful, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other `mkDPIxxxx()` functions that create a serialized DPI packet.

Usage

The `mkDPIAreYouThere()` function creates a serialized DPI `ARE_YOU_THERE` packet that can be sent to the DPI peer, which is normally the agent.

A subagent connected through TCP or UNIXstream probably does not need this function because, normally when the agent breaks the connection, you will receive an EOF on the file descriptor.

If your connection to the agent is still healthy, the agent will send a DPI `RESPONSE` with `SNMP_ERROR_DPI_noError` in the error code field and 0 in the error index field. The `RESPONSE` will have no `varBind` data. If your connection is not healthy, the agent might send a response with an error indication, or might not send a response at all.

Examples

```
#include <snmp_dpi.h>
unsigned char *pack_p;

pack_p = mkDPIAreYouThere();
if (pack_p) {
    /* send the packet to the agent */
} /* endif */
/* wait for response with DPIawait_packet_from_agent() */
/* normally the response should come back pretty quickly, */
/* but it depends on the load of the agent */
```

Context

“The `snmp_dpi_resp_packet` structure” on page 96

“The `DPIawait_packet_from_agent()` function” on page 78

The mkDPIClose() function

Format

```
#include <snmp_dpi.h>

unsigned char *mkDPIClose(char reason_code);
```

Parameters

reason_code

The reason for closing the DPI connection. See “DPI CLOSE reason codes” on page 102 for a list of valid reason codes.

Return codes

If successful, a pointer to a static DPI packet buffer is returned. The first 2 bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If not successful, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPIxxxx() functions that create a serialized DPI packet.

Usage

The mkDPIClose() function creates a serialized DPI CLOSE packet that can be sent to the DPI peer. As a result of sending the packet, the DPI connection will be closed.

Sending a DPI CLOSE packet to the agent implies an automatic DPI UNREGISTER for all registered subtrees on the connection being closed.

Examples

```
#include <snmp_dpi.h>
unsigned char *pack_p;

pack_p = mkDPIClose(SNMP_CLOSE_goingDown);
if (pack_p) {
    /* send the packet to the agent */
} /* endif */
```

Context

“The snmp_dpi_close_packet structure” on page 91

“DPI CLOSE reason codes” on page 102

The mkDPlopen() function

Format

```
#include <snmp_dpi.h>

unsigned char *mkDPlopen(      /* Make a DPI open packet */
    char      *oid_p,         /* subagent Identifier (OID) */
    char      *description_p, /* subagent descriptive name */
    unsigned long timeout,     /* requested default timeout */
    unsigned long max_varBinds, /* max varBinds per DPI packet*/
    char      character_set,   /* selected character set */
    #define DPI_NATIVE_CSET 0 /* 0 = native character set */
    #define DPI_ASCII_CSET 1 /* 1 = ASCII character set */

    unsigned long password_len, /* length of password (if any)*/
    unsigned char *password_p); /* ptr to password (if any) */
```

Parameters

oid_p A pointer to a null-terminated character string representing the object identifier which uniquely identifies the subagent. The OID valued pointed to by oid_p must be in the EBCDIC character set when communicating with a TCP/IP UNIX System Services SNMP agent. The agent will add the OID passed in the mkDPlopen call to the sysORTable as sysORID in a corresponding new entry. By convention, sysORID should match a capabilities statement OID to refer to the MIBs supported by the subagent.

For a list of MIB variables, see the *z/OS Communications Server: IP System Administrator's Commands*.

description_p

A pointer to a null-terminated character string, which is a descriptive name for the subagent. This can be any DisplayString.

timeout

The requested timeout for this subagent. An agent often has a limit for this value and it will use that limit if this value is larger. A timeout of 0 has a special meaning in the sense that the agent will use its own default timeout value.

max_varBinds

The maximum number of varBinds per DPI packet that the subagent is prepared to handle. It must be a positive number or 0.

- If a value greater than 1 is specified, the agent will try to combine as many varBinds that belong to the same subtree per DPI packet as possible up to this value.
- If a value of 0 is specified, the agent will try to combine up to as many varBinds as are present in the SNMP packet and belong to the same subtree; there is no limit on the number of varBinds present in the DPI packet.

character_set

The character set that you want to use for string-based data fields in the DPI packets and structures. See “Character set selection” on page 101 for more information.

DPI_NATIVE_CSET

Specifies that you want to use the native character set of the platform on which the agent that you connect to is running.

password_len

The length in octets of an optional password. It depends on the agent implementation if a password is needed.

If coded, this parameter is ignored with the z/OS SNMP agent.

password_p

A pointer to an octet string representing the password for this subagent. A password might include any character value, including the NULL character. If the password_len is 0, this can be a NULL pointer.

If coded, this parameter is ignored with the SNMP agent.

Return codes

If successful, a pointer to a static DPI packet buffer is returned. The first 2 bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If not successful, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPIxxxx() functions that create a serialized DPI packet.

Usage

The mkDPIopen() function creates a serialized DPI OPEN packet that can then be sent to the DPI peer that is a DPI-capable SNMP agent.

Normally you will want to use the native character set, which is the easiest for the subagent programmer. However, if the agent and subagent each run on their own platforms and those platforms use different native character sets, you must select the ASCII character set, so that you both know exactly how to represent string-based data that is being sent back and forth.

Currently, if you specify a password parameter, it will be ignored. You do not need to specify a password to connect to the SNMP agent; you can pass a length of 0 and a NULL pointer for the password.

Examples

```
#include <snmp_dpi.h>
unsigned char *pack_p;

pack_p = mkDPIopen("1.3.6.1.2.3.4.5",
                  "Sample DPI subagent"
                  0L, 2L, DPI_NATIVE_CSET, /* max 2 varBinds */
                  0, (char *)0);

if (pack_p) {
    /* send packet to the agent */
} /* endif */
```

Context

“Character set selection” on page 101

The mkDPIregister() function

Format

```
#include <snmp_dpi.h>

unsigned char *mkDPIregister( /* Make a DPI register packet */
    unsigned short timeout, /* in seconds (16-bit) */
    long int priority, /* requested priority */
    char *group_p, /* ptr to group ID (subtree) */
    char bulk_select); /* Bulk selection (GETBULK) */
#define DPI_BULK_NO 0 /* map GETBULK into GETNEXTs */
/*
```

Parameters

timeout

The requested timeout in seconds. An agent often has a limit for this value and it will use that limit if this value is larger. The value 0 has special meaning in the sense that it tells the agent to use the timeout value that was specified in the DPI OPEN packet.

priority

The requested priority. This field can contain any of these values:

- 1** Requests the best available priority.
- 0** Requests a better priority than the highest priority currently registered. Use this value to obtain the SNMP DPI Version 1 behavior.
- nnn** Any positive value. You will receive that priority if available; otherwise, you will receive the next best priority that is available.

group_p

A pointer to a null-terminated character string that represents the subtree to be registered. For object level registration, this group ID must have a trailing period. For instance level registration, this group ID would simply have the instance number follow the object number subtree.

bulk_select

Specifies if you want the agent to pass GETBULK on to the subagent or to map them into multiple GETNEXT requests. The choices are:

DPI_BULK_NO

Do not pass any GETBULK requests, but instead map a GETBULK request into multiple GETNEXT requests.

Return codes

If successful, a pointer to a static DPI packet buffer is returned. The first 2 bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If not failure, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPIxxxx() functions that create a serialized DPI packet.

Usage

The mkDPIregister() function creates a serialized DPI REGISTER packet that can then be sent to the DPI peer that is a DPI-capable SNMP agent.

Normally, the SNMP agent sends a DPI RESPONSE packet back. This packet identifies if the register was successful or not.

The agent returns the assigned priority in the error index field of the response packet.

Examples

```
#include <snmp_dpi.h>
unsigned char *pack_p;

pack_p = mkDPIregister(0,0L,"1.3.6.1.2.3.4.5."
                      DPI_BULK_NO);
if (pack_p) {
    /* send packet to agent and await response */
} /* endif */
```

Context

“The snmp_dpi_resp_packet structure” on page 96

The mkDPiresponse() function

Format

```
#include <snmp_dpi.h>

unsigned char    *mkDPiresponse( /* Make a DPI response packet*/
    snmp_dpi_hdr    *hdr_p,      /* ptr to packet to respnd to*/
    long int        error_code,   /* error code: SNMP_ERROR_****/
    long int        error_index, /* index to varBind in error */
    snmp_dpi_set_packet *packet_p); /* ptr to varBinds, a chain */
                                   /* of dpi_set_packets      */
```

Parameters

hdr_p A pointer to the parse tree of the DPI request to which this DPI packet will be the response. The function uses this parse tree to copy the packet_id and the DPI version and release, so that the DPI packet is correctly formatted as a response.

error_code

The error code.

See “DPI RESPONSE error codes” on page 102 for a list of valid codes.

error_index

Specifies the first varBind in error. Counting starts at 1 for the first varBind. This field should be 0 if there is no error.

packet_p

A pointer to a chain of snmp_dpi_set_packet structures. This partial parse tree will be freed by the mkDPiresponse() function, so upon return you cannot refer to it anymore. Pass a NULL pointer if there are no varBinds to be returned.

Return codes

If successful, a pointer to a static DPI packet buffer is returned. The first 2 bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If not successful, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPixxxx() functions that create a serialized DPI packet.

Usage

The mkDPiresponse() function is used at the subagent side to prepare a DPI RESPONSE packet to a GET, GETNEXT, SET, COMMIT, or UNDO request. The resulting packet can be sent to the DPI peer, which is normally a DPI-capable SNMP agent.

Examples

```
#include <snmp_dpi.h>
unsigned char    *pack_p;
snmp_dpi_hdr    *hdr_p;
snmp_dpi_set_packet *set_p;
long int        num;

hdr_p = pDPiPacket(pack_p);    /* parse incoming packet */
                                /* assume it's in pack_p */
if (hdr_p) {
```

```

/* analyze packet, assume GET, no error */
set_p = mkDPISet(snmpp_dpi_set_packet_NULL_p,
                 "1.3.6.1.2.3.4.5.", "1.0",
                 SNMP_TYPE_Integer32,
                 sizeof(num), &num);

if (set_p) {
    pack_p = mkDPISet(snmpp_hdr_p,
                      SNMP_ERROR_noError, 0L, set_p);
    if (pack_p) {
        /* send packet to agent */
    } /* endif */
} /* endif */
} /* endif */

```

Context

- “The pDPISet() function” on page 76
- “The snmpp_hdr structure” on page 93
- “The snmpp_set_packet structure” on page 97

The mkDPIset() function

Format

```
#include <snmp_dpi.h>

snmp_dpi_set_packet *mkDPIset( /* Make DPI set packet tree */
    snmp_dpi_set_packet *packet_p, /* ptr to SET structure */
    char *group_p, /* ptr to group ID(subtree)*/
    char *instance_p, /* ptr to instance OIDstring*/
    int value_type, /* value type: SNMP_TYPE_****/
    int value_len, /* length of value */
    void *value_p); /* ptr to value */
```

Parameters

packet_p

A pointer to a chain of snmp_dpi_set_packet structures. Pass a NULL pointer if this is the first structure to be created.

group_p

A pointer to a null-terminated character string that represents the registered subtree that caused this GET request to be passed to this DPI subagent. The subtree must have a trailing period.

instance_p

A pointer to a null-terminated character string that represents the rest, which is the piece following the subtree part, of the object identifier of the variable instance being accessed. Use of the term *instance_p* here should not be confused with an OBJECT instance because this string can consist of a piece of the object identifier plus the INSTANCE IDENTIFIER.

value_type

The type of the value.

See “DPI SNMP value types” on page 103 for a list of currently defined value types.

value_len

This is the value that specifies the length in octets of the value pointed to by the *value* field. The length can be 0 if the value is of type SNMP_TYPE_NULL.

The maximum value is 64KB minus 1. However, the implementation often makes the length significantly less.

value_p

A pointer to the actual value. This field can contain a NULL pointer if the value is of implicit or explicit type SNMP_TYPE_NULL.

Return codes

If successful and a chain of one or more packets was passed in the *packet_p* parameter, the same pointer that was passed in *packet_p* is returned. A new dynamically allocated structure has been added to the end of that chain of snmp_dpi_get_packet structures.

If successful and a NULL pointer was passed in the *packet_p* parameter, a pointer to a new dynamically allocated structure is returned.

If not successful, a NULL pointer is returned.

Usage

The mkDPIset() function is used at the subagent side to prepare a chain of one or more snmp_dpi_set_packet structures. This chain is used to create a DPI

RESPONSE packet by a call to `mkDPIresponse()` that can be sent to the DPI peer, which is normally a DPI-capable SNMP agent.

The chain of `snmp_dpi_set_packet` structures can also be used to create a DPI TRAP packet that includes `varBinds` as explained in “The `mkDPItrap()` function” on page 73.

For the `value_len`, the maximum value is 64KB minus 1. However, the implementation often makes the length significantly less. For example, the SNMP PDU size might be limited to 484 bytes at the SNMP manager or agent side. In this case, the total response packet cannot exceed 484 bytes, so a `value_len` is limited to 484 bytes. You can send the DPI packet to the agent, but the manager will never see it.

Examples

```
#include <snmp_dpi.h>
unsigned char    *pack_p;
snmp_dpi_hdr    *hdr_p;
snmp_dpi_set_packet *set_p;
long int        num;

hdr_p = pDPIpacket(pack_p)    /* parse incoming packet */
                                /* assume it's in pack_p */

if (hdr_p) {
    /* analyze packet, assume GET, no error */
    set_p = mkDPIset(snmp_dpi_set_packet_NULL_p,
                    "1.3.6.1.2.3.4.5.", "1.0",
                    SNMP_TYPE_Integer32,
                    sizeof(num), &num);

    if (set_p) {
        pack_p = mkDPIresponse(hdr_p,
                                SNMP_ERROR_noError,
                                0L, set_p);

        if (pack_p)
            /* send packet to agent */
        } /* endif */
    } /* endif */
} /* endif */
```

If you must chain many `snmp_dpi_set_packet` structures, be sure to note that the packets are chained only by forward pointers. It is recommended that you use the last structure in the existing chain as the *packet_p* parameter. Then, the underlying code does not have to scan through a possibly long chain of structures to chain the new structure at the end.

Context

- “The `pDPIpacket()` function” on page 76
- “The `mkDPIresponse()` function” on page 69
- “The `mkDPItrap()` function” on page 73
- “The `snmp_dpi_hdr` structure” on page 93
- “The `snmp_dpi_set_packet` structure” on page 97
- “DPI SNMP value types” on page 103
- “Value representation” on page 104

The mkDPITrap() function

Format

```
#include <snmp_dpi.h>

unsigned char      *mkDPITrap( /* Make a DPI trap packet */
    long int      generic, /* generic traptype (32 bit)*/
    long int      specific, /* specific traptype (32 bit)*/
    snmp_dpi_set_packet *packet_p, /* ptr to varBinds, a chain */
                                /* of dpi_set_packets */
    char          *enterprise_p); /* ptr to enterprise OID */
```

Parameters

generic

The generic trap type. The range of this value is 0-6, where 6, which is enterprise specific, is the type that is probably used most by DPI subagent programmers. The values in the range 0-5 are well defined standard SNMP traps.

specific

The enterprise specific trap type. This can be any value that is valid for the MIB subtrees that the subagent implements.

packet_p

A pointer to a chain of snmp_dpi_set_structures, representing the varBinds to be passed with the trap. This partial parse tree will be freed by the mkDPITrap() function so you cannot refer to it anymore upon completion of the call. A NULL pointer means that there are no varBinds to be included in the trap.

enterprise_p

A pointer to a null-terminated character string representing the enterprise ID (object identifier) for which this trap is defined. A NULL pointer can be used. In this case, the subagent identifier, as passed in the DPI OPEN packet, will be used when the agent receives the DPI TRAP packet.

Return codes

If successful, a pointer to a static DPI packet buffer is returned. The first 2 bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If not successful, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPIxxxx() functions that create a serialized DPI packet.

Usage

The mkDPITrap() function is used at the subagent side to prepare a DPI TRAP packet. The resulting packet can be sent to the DPI peer, which is normally a DPI-capable SNMP agent.

Examples

```
#include <snmp_dpi.h>
unsigned char      *pack_p;
snmp_dpi_set_packet *set_p;
long int          num;

set_p = mkDPIset(snm_dpi_set_packet_NULL_p,
```

```

        "1.3.6.1.2.3.4.5.", "1.0",
        SNMP_TYPE_Integer32,
        sizeof(num), &num);
if (set_p) {
    pack_p = mkDPITrap(6,1,set_p, (char *)0);
    if (pack_p) {
        /* send packet to agent */
    } /* endif */
} /* endif */

```

Context

“The mkDPISet() function” on page 71

The mkDPIunregister() function

Format

```
#include <snmp_dpi.h>

unsigned char *mkDPIunregister( /* Make DPI unregister packet */
    char        reason_code; /* unregister reason code */
    char        *group_p);    /* ptr to group ID (subtree) */
```

Parameters

reason_code

The reason for the unregister.

See “DPI UNREGISTER reason codes” on page 103 for a list of the currently defined reason codes.

group_p

A pointer to a null-terminated character string that represents the subtree to be unregistered. For object level registration, this group ID must have a trailing period. For instance level registration, this group ID would simply have the instance number follow the object number subtree.

Return codes

If successful, a pointer to a static DPI packet buffer is returned. The first 2 bytes of the buffer in network byte order contain the length of the remaining packet. The macro DPI_PACKET_LEN can be used to calculate the total length of the DPI packet.

If not successful, a NULL pointer is returned.

Note: The static buffer for the DPI packet is shared by other mkDPIxxxx() functions that create a serialized DPI packet.

Usage

The mkDPIunregister() function creates a serialized DPI UNREGISTER packet that can be sent to the DPI peer, which is a DPI-capable SNMP agent.

Normally, the SNMP peer then sends a DPI RESPONSE packet back. This packet identifies if the unregister was successful or not.

Examples

```
#include <snmp_dpi.h>
unsigned char *pack_p;

pack_p = mkDPIunregister(
    SNMP_UNREGISTER_goingDown,
    "1.3.6.1.2.3.4.5.");
if (pack_p) {
    /* send packet to agent and await response */
} /* endif */
```

Context

“The snmp_dpi_ureg_packet structure” on page 99

The pDPIpacket() function

Format

```
#include <snmp_dpi.h>

snmp_dpi_hdr *pDPIpacket(unsigned char *packet_p);
```

Parameters

packet_p

A pointer to a serialized DPI packet.

Return codes

If successful, a pointer to a DPI parse tree (snmp_dpi_hdr) is returned. Memory for the parse tree has been dynamically allocated, and it is the callers responsibility to free it when no longer needed. You can use the fDPIparse() function to free the parse tree.

If not successful, a NULL pointer is returned.

Usage

The pDPIpacket() function parses the buffer pointed to by the *packet_p* parameter. It ensures that the buffer contains a valid DPI packet and that the packet is for a DPI version and release that is supported by the DPI functions in use.

Examples

```
#include <snmp_dpi.h>
unsigned char      *pack_p;
snmp_dpi_hdr      *hdr_p;

hdr_p = pDPIpacket(pack_p);          /* parse incoming packet */
                                      /* assume it's in pack_p */

if (hdr_p) {
    /* analyze packet, and handle it */
}
```

Context

“The snmp_dpi_hdr structure” on page 93

“The snmp_dpi.h include file” on page 106

“The fDPIparse() function” on page 61

Transport-related DPI API functions

This section describes each of the DPI transport-related functions that are available to the DPI subagent programmer. These functions try to hide any platform specific issues for the DPI subagent programmer so that the subagent can be made as portable as possible. If you need detailed control for sending and awaiting DPI packets, you might have to do some of the transport-related code yourself.

The transport-related functions are basically the same for any platform, except for the initial call to set up a connection. SNMP currently supports the TCP transport type, as well as UNIXstream.

The Transport-Related DPI API Functions are:

- “The `DPIawait_packet_from_agent()` function” on page 78
- “The `DPIconnect_to_agent_TCP()` function” on page 80
- “The `DPIconnect_to_agent_UNIXstream()` function” on page 82
- “The `DPIdisconnect_from_agent()` function” on page 84
- “The `DPIget_fd_for_handle()` function” on page 85
- “The `DPIsend_packet_to_agent()` function” on page 86
- “The `lookup_host()` function” on page 88
- “The `lookup_host6()` function” on page 89

The DPlawait_packet_from_agent() function

Format

```
#include <snmp_dpi.h>

int DPlawait_packet_from_agent( /* await a DPI packet */
    int handle, /* on this connection */
    int timeout, /* timeout in seconds */
    unsigned char **message_p, /* receives ptr to data */
    unsigned long *length); /* receives length of data */
```

Parameters

handle

A handle as obtained with a DPIconnect_to_agent_xxxx() call.

timeout

A timeout value in seconds. There are two special values:

- 1 Causes the function to wait forever until a packet arrives.
- 0 Means that the function will only check if a packet is waiting. If not, an immediate return is made. If there is a packet, it will be returned.

message_p

The address of a pointer that will receive the address of a static DPI packet buffer or, if there is no packet, a NULL pointer.

length The address of an unsigned long integer that will receive the length of the received DPI packet or, if there is no packet, a 0 value.

Return codes

If successful, a 0 (DPI_RC_OK) is returned. The buffer pointer and length of the caller will be set to point to the received DPI packet and to the length of that packet.

If not successful, a negative integer is returned, which indicates the kind of error that occurred. See “Return codes from DPI transport-related functions” on page 105 for a list of possible error codes.

DPI_RC_NOK

This is a return code indicating the DPI code is out of sync or has a bug.

DPI_RC_EOF

End of file on the connection. The connection has been closed.

DPI_RC_IO_ERROR

An error occurred with an underlying select() or recvfrom() call, or a DPI packet was read that was less than 2 bytes. DPI uses the first 2 bytes to get the packet length.

DPI_RC_INVALID_HANDLE

A bad handle was passed as input. Either the handle is not valid, or it describes a connection that has been disconnected.

DPI_RC_TIMEOUT

No packet was received during the specified timeout period.

DPI_RC_PACKET_TOO_LARGE

The packet received was too large.

Usage

The `DPIawait_packet_from_agent()` function is used at the subagent side to await a DPI packet from the DPI-capable SNMP agent. The programmer can specify how long to wait.

Examples

```
#include <snmp_dpi.h>
int             handle;
unsigned char    *pack_p;
unsigned long    length;

handle = DPIconnect_to_agent_TCP("127.0.0.1", "public");
if (handle < 0) {
    printf("Error %d from connect\n",handle);
    exit(1);
} /* endif */
/* do useful stuff */
rc = DPIawait_packet_from_agent(handle, -1,
                                &pack_p, &length);
if (rc) {
    printf("Error %d from await packet\n");
    exit(1);
} /* endif */
/* handle the packet */
```

Context

“The `DPIconnect_to_agent_TCP()` function” on page 80

“The `DPIconnect_to_agent_UNIXstream()` function” on page 82

The DPlconnect_to_agent_TCP() function

Format

```
#include <snmp_dpi.h>

int DPlconnect_to_agent_TCP( /* Connect to DPI TCP port */
    char *hostname_p, /* target hostname/IP address */
    char *community_p); /* community name */
```

Parameters

hostname_p

A pointer to a null-terminated character string representing the host name or IP address in IPv4 dotted-decimal or IPv6 colon-hexadecimal notation of the host where the DPI-capable SNMP agent is running.

community_p

A pointer to a null-terminated character string representing the community name that is required to obtain the dpiPort from the SNMP agent through an SNMP GET request.

Note: For z/OS Communications Server, the SNMP community passed by the subagent must be in ASCII only.

Return codes

If successful, a nonnegative integer that represents the connection is returned. It is to be used as a handle in subsequent calls to DPI transport-related functions.

If not successful, a negative integer is returned, which indicates the kind of error that occurred. See “Return codes from DPI transport-related functions” on page 105 for a list of possible error codes.

DPI_RC_NO_PORT

Unable to obtain the dpiPort number. There are many reasons for this, for example: bad host name, bad community name, or default timeout (9 seconds) before a response from the agent.

DPI_RC_IO_ERROR

An error occurred with an underlying select(), or DPI was not able to set up a socket (could be due to an error on a socket(), bind(), connect() call, or other internal errors).

Usage

The DPlconnect_to_agent_TCP() function is used at the subagent side to set up a TCP connection to the DPI-capable SNMP agent.

As part of the connection processing, the DPlconnect_to_agent_TCP() function sends an SNMP GET request to the SNMP agent to retrieve the port number of the DPI port to be used for the TCP connection. By default, this SNMP GET request is sent to the well-known SNMP port 161. If the SNMP agent is listening on a port other than well-known port 161, the SNMP_PORT environment variable can be set to the port number of the SNMP agent prior to issuing the DPlconnect_to_agent_TCP(). Use setenv() to override port 161 before using this function.

Examples

```
#include <snmp_dpi.h>
int          handle;

handle = DPIconnect_to_agent_TCP("127.0.0.1", "public");
if (handle < 0) {
    printf("Error %d from connect\n",handle);
    exit(1);
} /* endif */
```

Context

“Return codes from DPI transport-related functions” on page 105

“The DPIconnect_to_agent_UNIXstream() function” on page 82

The `DPIconnect_to_agent_UNIXstream()` function

Format

```
#include <snmp_dpi.h>

int DPIconnect_to_agent_UNIXstream(    /* Connect to DPI UNIXstream */
    char        *hostname_p,          /* target hostname/IP address */
    char        *community_p);        /* community name */
```

Parameters

`hostname_p`

A pointer to a null-terminated character string representing the local host name or IP address in IPv4 dotted-decimal or IPv6 colon-hexadecimal notation of the local host where the DPI-capable SNMP agent is running.

`community_p`

A pointer to a null-terminated character string representing the community name that is required to obtain the UNIX[®] pathname from the SNMP agent through an SNMP GET request.

Note: For z/OS Communications Server, the SNMP community passed by the subagent must be in ASCII only.

Return codes

If successful, a nonnegative integer that represents the connection is returned. It is to be used as a handle in subsequent calls to DPI transport-related functions. If not successful, a negative integer is returned, which indicates the kind of error that occurred. See “Return codes from DPI transport-related functions” on page 105 for a list of possible error codes.

`DPI_RC_NO_PORT`

Unable to obtain the UNIX pathname. There are many reasons for this, for example: bad host name, bad community name, or default timeout (9 seconds) before a response from the agent.

`DPI_RC_IO_ERROR`

An error occurred with an underlying `select()`, or DPI was not able to set up a socket (could be due to an error on a `socket()`, `bind()`, `connect()` call, or other internal errors).

Usage

The `DPIconnect_to_agent_UNIXstream()` function is used at the subagent side to set up an `AF_UNIX` connection to the DPI-capable SNMP agent.

As part of the connection processing, the `DPIconnect_to_agent_UNIXstream()` function will send an SNMP GET request to the SNMP agent to retrieve the pathname to be used for the UNIX streams connection. By default, this SNMP GET request is sent to the well-known SNMP port 161. If the SNMP agent is listening on a port other than well-known port 161, the `SNMP_PORT` environment variable can be set to the port number of the SNMP agent prior to issuing the `DPIconnect_to_agent_UNIXstream()`. Use `setenv()` to override port 161 before using this function.

Establishing Permission uses a pathname in the z/OS UNIX file system as the name of the socket for connect. This pathname is available at the SNMP agent through the MIB object 1.3.6.1.4.1.2.2.1.1.3, which has the name `dpiPathNameForUnixStream`. The SNMP agent has a default name that it uses

(/tmp/dpi_socket) if you do not supply another name in the agent startup parameter (-s) or in the OSNMPD.DATA file. Whatever name is chosen, it has to live in the z/OS UNIX file system as a character special file.

To run a user-written subagent from a nonprivileged user ID, set the permission bits for the character special file to *write* access. Otherwise, a subagent using this function will have to be run from a superuser or other user with appropriate privileges.

Examples

```
#include <snmp_dpi.h>
int          handle;

handle = DPIconnect_to_agent_UNIXstream("127.0.0.1", "public");
if (handle < 0) {
    printf("Error %d from connect\n",handle);
    exit(1);
} /* endif */
```

Context

“Return codes from DPI transport-related functions” on page 105

“The DPIconnect_to_agent_TCP() function” on page 80

The DPIdisconnect_from_agent() function

Format

```
#include <snmp_dpi.h>

void DPIdisconnect_from_agent( /* disconnect from DPI (agent)*/
    int handle); /* close this connection */
```

Parameters

handle

A handle as obtained with a DPIconnect_to_agent_xxxx() call.

Usage

The DPIdisconnect_from_agent() function is used at the subagent side to terminate a connection to the DPI-capable SNMP agent.

Examples

```
#include <snmp_dpi.h>
int handle;

handle = DPIconnect_to_agent_TCP("127.0.0.1", "public");
if (handle < 0) {
    printf("Error %d from connect\n",handle);
    exit(1);
} /* endif */
/* do useful stuff */
DPIdisconnect_from_agent(handle);
```

Context

“The DPIconnect_to_agent_TCP() function” on page 80

“The DPIconnect_to_agent_UNIXstream() function” on page 82

The `DPIget_fd_for_handle()` function

Format

```
#include <snmp_dpi.h>

int DPIget_fd_for_handle(    /* get the file descriptor */
    int handle);           /* for this handle */
```

Parameters

handle

A handle that was obtained with a `DPIconnect_to_agent_xxxx()` call.

Return codes

If successful, a positive integer representing the file descriptor associated with the specified handle.

If not successful, a negative integer is returned, which indicates the error that occurred. See “Return codes from DPI transport-related functions” on page 105 for a list of possible error codes.

DPI_RC_INVALID_HANDLE

A bad handle was passed as input. Either the handle is not valid, or it describes a connection that has been disconnected.

Usage

The `DPIget_fd_for_handle` function is used to obtain the file descriptor for the handle, which was obtained with a `DPIconnect_to_agent_TCP()` call or a `DPIconnect_to_agent_UNIXstream()` call.

Using this function to retrieve the file descriptor associated with your DPI connections enables you to use either the `select` or `selectex` socket calls. Using `selectex` enables your program to wait for event control blocks (ECBs), in addition to a read condition. This is one example of how an MVS application can wait for notification of the receipt of a modify command (through an ECB post) or DPI packet at the same time.

Examples

```
#include <snmp_dpi.h>
#include /* other include files for BSD sockets and such */
int handle;
int fd;

handle = DPIconnect_to_agent_TCP("127.0.0.1","public");
if (handle < 0) {
    printf("Error %d from connect\n",handle);
    exit(1);
}
fd = DPIget_fd_for_handle(handle);
if (fd < 0) {
    printf("Error %d from get_fd\n",fd);
    exit(1);
}
```

Context

“The `DPIconnect_to_agent_TCP()` function” on page 80

“The `DPIconnect_to_agent_UNIXstream()` function” on page 82

The DPlsend_packet_to_agent() function

Format

```
#include <snmp_dpi.h>

int DPlsend_packet_to_agent(    /* send a DPI packet    */
    int            handle,    /* on this connection    */
    unsigned char  *message_p, /* ptr to the packet data */
    unsigned long   length);  /* length of the packet  */
```

Parameters

handle

A handle as obtained with a DPIconnect_to_agent_xxxx() call.

message_p

A pointer to the buffer containing the DPI packet to be sent.

length The length of the DPI packet to be sent. The DPI_PACKET_LEN macro is a useful macro to calculate the length.

Return codes

If successful, a 0 (DPI_RC_OK) is returned.

If not successful, a negative integer is returned, which indicates the kind of error that occurred. See "Return codes from DPI transport-related functions" on page 105 for a list of possible error codes.

DPI_RC_NOK

This is a return code, but it really means the DPI code is out of sync or has a bug.

DPI_RC_IO_ERROR

An error occurred with an underlying send(), or the send() failed to send all of the data on the socket (incomplete send).

DPI_RC_INVALID_ARGUMENT

The message_p parameter is NULL or the length parameter has a value of 0.

DPI_RC_INVALID_HANDLE

A bad handle was passed as input. Either the handle is not valid, or it describes a connection that has been disconnected.

Usage

The DPlsend_packet_to_agent() function is used at the subagent side to send a DPI packet to the DPI-capable SNMP agent.

Examples

```
#include <snmp_dpi.h>
int            handle;
unsigned char  *pack_p;

handle = DPIconnect_to_agent_TCP("127.0.0.1", "public");
if (handle < 0) {
    printf("Error %d from connect\n",handle);
    exit(1);
} /* endif */
pack_p = mkDPIopen("1.3.6.1.2.3.4.5",
                  "Sample DPI subagent",
                  0L,2L,,DPI_NATIVE_CSET,
```

```

                                0,(char *)0);
if (pack_p) {
    rc = DPISend_packet_to_agent(handle,pack_p,
                                DPI_PACKET_LEN(pack_p));
    if (rc) {
        printf("Error %d from send packet\n");
        exit(1);
    } /* endif */
} else {
    printf("Can't make DPI OPEN packet\n");
    exit(1);
} /* endif */
/* await the response */

```

Context

“The DPISend_packet_to_agent_TCP() function” on page 80

“The DPISend_packet_to_agent_UNIXstream() function” on page 82

“The DPI_PACKET_LEN() macro” on page 60

The lookup_host() function

Format

```
#include <snmp_dpi.h>

unsigned long lookup_host( /* find IP address in network */
    char *hostname_p); /* byte order for this host */
```

Parameters

hostname_p

A pointer to a null-terminated character string representing the host name or IP address in dotted-decimal notation of the host where the DPI-capable SNMP agent is running.

Return codes

If successful, the IP address is returned in network byte order, so it is ready to be used in a sockaddr_in structure.

If not successful, a value of 0 is returned.

Usage

The lookup_host() function is used to obtain the IP address in network byte order of a host or IP address in dotted decimal notation. This function is implicitly executed by both DPIconnect_to_agent_TCP and DPIconnect_to_agent_UNIXstream.

Context

“The DPIconnect_to_agent_TCP() function” on page 80

The lookup_host6() function

Format

```
#include <snmp_dpi.h>

struct sockaddr_in6 *lookup_host6( /* find IPv6 address in network */
    char *hostname_p); /* byte order for this host */
```

Parameters

hostname_p

A pointer to a null-terminated character string representing the host name or IPv6 address in colon-hexadecimal notation of the host where the DPI-capable SNMP agent is running.

Return codes

If successful, a pointer to a `sockaddr_in6` structure is returned. The structure is filled in with the IPv6 address of the specified host in network byte order.

If not successful, a NULL pointer is returned.

Usage

The `lookup_host6()` function is used to obtain an IPv6 address in network byte order of a host specified by host name or IPv6 address in colon-hexadecimal notation. This function can be implicitly executed by `DPIconnect_to_agent_TCP` and `DPIconnect_to_agent_UNIXstream`.

If the function is successful, the caller does not own the `sockaddr_in6` structure pointed to by the return value. If the caller needs to store the IPv6 address or the entire structure, it should do so immediately after `lookup_host6()` returns, because subsequent calls to `lookup_host6()` will cause the contents of the `sockaddr_in6` to be overwritten.

Context

“The `DPIconnect_to_agent_TCP()` function” on page 80

DPI structures

This section describes each data structure that is used in the SNMP DPI API:

- “The snmp_dpi_close_packet structure” on page 91
- “The snmp_dpi_get_packet structure” on page 92
- “The snmp_dpi_hdr structure” on page 93
- “The snmp_dpi_next_packet structure” on page 95)
- “The snmp_dpi_resp_packet structure” on page 96
- “The snmp_dpi_set_packet structure” on page 97
- “The snmp_dpi_ureg_packet structure” on page 99
- “The snmp_dpi_u64 structure” on page 100

The snmp_dpi_close_packet structure

Format

```
struct dpi_close_packet {  
    char          reason_code;    /* reason for closing */  
};  
typedef struct dpi_close_packet    snmp_dpi_close_packet;  
#define snmp_dpi_close_packet_NULL_p ((snmp_dpi_close_packet*)0)
```

Parameters

reason_code

The reason for the close.

See “DPI CLOSE reason codes” on page 102 for a list of valid reason codes.

Usage

The snmp_dpi_close_packet structure represents a parse tree for a DPI CLOSE packet.

The snmp_dpi_close_packet structure might be created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_CLOSE. The snmp_dpi_hdr structure then contains a pointer to an snmp_dpi_close_packet structure.

An snmp_dpi_close_packet_structure is also created as a result of an mkDPIclose() call, but the programmer never sees the structure because mkDPIclose() immediately creates a serialized DPI packet from it and then frees the structure.

It is recommended that DPI subagent programmer uses mkDPIclose() to create a DPI CLOSE packet.

Context

“The pDPIpacket() function” on page 76

“The mkDPIclose() function” on page 64

“The snmp_dpi_hdr structure” on page 93

The snmp_dpi_get_packet structure

Format

```
struct dpi_get_packet {  
    char      *object_p; /* ptr to OID string */  
    char      *group_p; /* ptr to subtree(group) */  
    char      *instance_p; /* ptr to rest of OID */  
    struct dpi_get_packet *next_p; /* ptr to next in chain */  
};  
typedef struct dpi_get_packet      snmp_dpi_get_packet;  
#define snmp_dpi_get_packet_NULL_p ((snmp_dpi_get_packet *)0)
```

Parameters

object_p

A pointer to a null-terminated character string that represents the full object identifier of the variable instance that is being accessed. It basically is a concatenation of the fields *group_p* and *instance_p*. Using this field is not recommended because it is only included for DPI Version 1 compatibility and it might be withdrawn in a later version.

group_p

A pointer to a null-terminated character string that represents the registered subtree that caused this SET request to be passed to this DPI subagent. The subtree must have a trailing period.

instance_p

A pointer to a null-terminated character string that represents the rest, which is the piece following the subtree part, of the object identifier of the variable instance being accessed.

Use of the term *instance_p* here should not be confused with an OBJECT instance because this string might consist of a piece of the object identifier plus the INSTANCE IDENTIFIER.

next_p

A pointer to a possible next snmp_dpi_get_packet structure. If this next field contains the NULL pointer, this is the end of the chain.

Usage

The snmp_dpi_get_packet structure represents a parse tree for a DPI GET packet.

At the subagent side, the snmp_dpi_get_packet structure is normally created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_GET. The snmp_dpi_hdr structure then contains a pointer to a chain of one or more snmp_dpi_get_packet structures.

The DPI subagent programmer uses this structure to find out which variable instances are to be returned in a DPI RESPONSE.

Context

“The pDPIpacket() function” on page 76

“The snmp_dpi_hdr structure” on page 93

The snmp_dpi_hdr structure

Format

```
struct snmp_dpi_hdr {
    unsigned char  proto_major; /* always 2: SNMP_DPI_PROTOCOL*/
    unsigned char  proto_version; /* DPI version */
    unsigned char  proto_release; /* DPI release */
    unsigned short packet_id; /* 16-bit, DPI packet ID */
    unsigned char  packet_type; /* DPI packet type */
    union {
        snmp_dpi_reg_packet *reg_p;
        snmp_dpi_ureg_packet *ureg_p;
        snmp_dpi_get_packet *get_p;
        snmp_dpi_next_packet *next_p;
        snmp_dpi_next_packet *bulk_p;
        snmp_dpi_set_packet *set_p;
        snmp_dpi_resp_packet *resp_p;
        snmp_dpi_trap_packet *trap_p;
        snmp_dpi_open_packet *open_p;
        snmp_dpi_close_packet *close_p;
        unsigned char *any_p;
    } data_u;
};
typedef struct snmp_dpi_hdr snmp_dpi_hdr;
#define snmp_dpi_hdr_NULL_p ((snmp_dpi_hdr *)0)
```

Parameters

proto_major

The major protocol. For SNMP DPI, it is always 2.

proto_version

The DPI version.

proto_release

The DPI release.

packet_id

This field contains the packet ID of the DPI packet. When you create a response to a request, the packet ID must be the same as that of the request. This is taken care of if you use the `mkDPIresponse()` function.

packet_type

The type of DPI packet (parse tree) that you are dealing with.

See “DPI packet types” on page 102 for a list of currently defined DPI packet types.

data_u

A union of pointers to the different types of data structures that are created based on the *packet_type* field. The pointers themselves have names that are self-explanatory.

The fields *proto_major*, *proto_version*, *proto_release*, and *packet_id* are basically for DPI internal use, so the DPI programmer normally does not need to be concerned about them.

Usage

The `snmp_dpi_hdr` structure is the anchor of a DPI parse tree. At the subagent side, the `snmp_dpi_hdr` structure is normally created as a result of a call to `pDPIpacket()`.

The DPI subagent programmer uses this structure to interrogate packets. Depending on the *packet_type*, the pointer to the chain of one or more *packet_type* specific structures that contain the actual packet data can be picked.

The storage for a DPI parse tree is always dynamically allocated. It is the responsibility of the caller to free this parse tree when it is no longer needed. You can use the `fDPIparse()` function to do that.

Note: Some `mkDPIxxxx` functions do free the parse tree that is passed to them. An example is the `mkDPIresponse()` function.

Context

- “The `fDPIparse()` function” on page 61
- “The `pDPIpacket()` function” on page 76
- “The `snmp_dpi_close_packet` structure” on page 91
- “The `snmp_dpi_get_packet` structure” on page 92
- “The `snmp_dpi_next_packet` structure” on page 95
- “The `snmp_dpi_resp_packet` structure” on page 96
- “The `snmp_dpi_set_packet` structure” on page 97
- “The `snmp_dpi_ureg_packet` structure” on page 99

The snmp_dpi_next_packet structure

Format

```
struct dpi_next_packet {
    char      *object_p; /* ptr to OID (string) */
    char      *group_p;  /* ptr to subtree(group)*/
    char      *instance_p; /* ptr to rest of OID */
    struct dpi_next_packet *next_p; /* ptr to next in chain */
};
typedef struct dpi_next_packet      snmp_dpi_next_packet;
#define snmp_dpi_next_packet_NULL_p ((snmp_dpi_next_packet *)0)
```

Parameters

object_p

A pointer to a null-terminated character string that represents the full object identifier of the variable instance that is being accessed. It basically is a concatenation of the fields *group_p* and *instance_p*. Using this field is not recommended because it is only included for DPI Version 1 compatibility and it might be withdrawn in a later version.

group_p

A pointer to a null-terminated character string that represents the registered subtree that caused this GETNEXT request to be passed to this DPI subagent. This subtree must have a trailing period.

instance_p

A pointer to a null-terminated character string that represents the rest, which is the piece following the subtree part, of the object identifier of the variable instance being accessed.

Use of the term *instance_p* here should not be confused with an OBJECT instance because this string might consist of a piece of the object identifier plus the INSTANCE IDENTIFIER.

next_p

A pointer to a possible next snmp_dpi_next_packet structure. If this next field contains the NULL pointer, this is the end of the chain.

Usage

The snmp_dpi_next_packet structure represents a parse tree for a DPI GETNEXT packet.

At the subagent side, the snmp_dpi_next_packet structure is normally created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_GETNEXT. The snmp_dpi_hdr structure then contains a pointer to a chain of one or more snmp_dpi_next_packet structures.

The DPI subagent programmer uses this structure to find out which variables instances are to be returned in a DPI RESPONSE.

Context

“The pDPIpacket() function” on page 76

“The snmp_dpi_hdr structure” on page 93

The snmp_dpi_resp_packet structure

Format

```
struct dpi_resp_packet {
    char          error_code; /* like: SNMP_ERROR_xxx */
    unsigned long int error_index; /* 1st varBind in error */
    #define resp_priority error_index /* if responds to register*/
    struct dpi_set_packet *varBind_p; /* ptr to varBind, chain */
                                         /* of dpi_set_packets */
};
typedef struct dpi_resp_packet      snmp_dpi_resp_packet;
#define snmp_dpi_resp_packet_NULL_p ((snmp_dpi_resp_packet *)0)
```

Parameters

error_code

The return code or the error code.

See “DPI RESPONSE error codes” on page 102 for a list of valid codes.

error_index

Specifies the first varBind in error. Counting starts at 1 for the first varBind. This field should be 0 if there is no error.

resp_priority

This is a redefinition of the *error_index* field. If the response is a response to a DPI REGISTER request and the *error_code* is equal to *SNMP_ERROR_DPI_noError* or *SNMP_ERROR_DPI_higherPriorityRegistered*, then this field contains the priority that was actually assigned. Otherwise, this field is set to 0 for responses to a DPI REGISTER.

varBind_p

A pointer to the chain of one or more *snmp_dpi_set_structures*, representing varBinds of the response. This field contains a NULL pointer if there are no varBinds in the response.

Usage

The *snmp_dpi_resp_packet* structure represents a parse tree for a DPI RESPONSE packet.

The *snmp_dpi_resp_packet* structure is normally created as a result of a call to *pDPIpacket()*. This is the case if the DPI packet is of type *SNMP_DPI_RESPONSE*. The *snmp_dpi_hdr* structure then contains a pointer to an *snmp_dpi_resp_packet* structure.

At the DPI subagent side, a DPI RESPONSE should only be expected at initialization and termination time when the subagent has issued a DPI OPEN, DPI REGISTER, or DPI UNREGISTER request.

The DPI programmer is advised to use the *mkDPIresponse()* function to prepare a DPI RESPONSE packet.

Context

- “The *pDPIpacket()* function” on page 76
- “The *mkDPIresponse()* function” on page 69
- “The *snmp_dpi_set_packet* structure” on page 97
- “The *snmp_dpi_hdr* structure” on page 93

The snmp_dpi_set_packet structure

Format

```
struct dpi_set_packet {  
    char      *object_p;    /* ptr to Object ID (string) */  
    char      *group_p;     /* ptr to subtree (group) */  
    char      *instance_p;  /* ptr to rest of OID */  
    unsigned char  value_type; /* value type: SNMP_TYPE_xxx */  
    unsigned short value_len; /* value length */  
    char      *value_p;     /* ptr to the value itself */  
    struct dpi_set_packet *next_p; /* ptr to next in chain */  
};  
typedef struct dpi_set_packet      snmp_dpi_set_packet;  
#define snmp_dpi_set_packet_NULL_p ((snmp_dpi_set_packet *)0)
```

Parameters

object_p

A pointer to a null-terminated character string that represents the full object identifier of the variable instance that is being accessed. It basically is a concatenation of the fields *group_p* and *instance_p*. Using this field is not recommended because it is only included for DPI Version 1 compatibility and it might be withdrawn in a later version.

group_p

A pointer to a null-terminated character string that represents the registered subtree that caused this SET, COMMIT, or UNDO request to be passed to this DPI subagent. The subtree must have a trailing period.

instance_p

A pointer to a null-terminated character string that represents the rest, which is the piece following the subtree part, of the object identifier of the variable instance being accessed.

Use of the term *instance_p* here should not be confused with an OBJECT instance because this string might consist of a piece of the object identifier plus the INSTANCE IDENTIFIER.

value_type

The type of the value.

See “DPI SNMP value types” on page 103 for a list of currently defined value types.

value_len

This is an unsigned 16-bit integer that specifies the length in octets of the value pointed to by the *value* field. The length can be 0 if the value is of type SNMP_TYPE_NULL.

value_p

A pointer to the actual value. This field can contain a NULL pointer if the value is of type SNMP_TYPE_NULL.

See “Value representation” on page 104 for information on how the data is represented for the various value types.

next_p

A pointer to a possible next snmp_dpi_set_packet structure. If this next field contains the NULL pointer, this is the end of the chain.

Usage

The snmp_dpi_set_packet structure represents a parse tree for a DPI SET request.

The `snmp_dpi_set_packet` structure might be created as a result of a call to `pDPIpacket()`. This is the case if the DPI packet is of type `SNMP_DPI_SET`, `SNMP_DPI_COMMIT`, or `SNMP_DPI_UNDO`. The `snmp_dpi_hdr` structure then contains a pointer to a chain of one or more `snmp_dpi_set_packet` structures.

This structure can also be created with an `mkDPIset()` call, which is typically used when preparing `varBinds` for a DPI RESPONSE packet.

Context

- “The `pDPIpacket()` function” on page 76
- “The `mkDPIset()` function” on page 71
- “DPI SNMP value types” on page 103
- “Value representation” on page 104
- “The `snmp_dpi_hdr` structure” on page 93

The snmp_dpi_ureg_packet structure

Format

```
struct dpi_ureg_packet {
    char          reason_code; /* reason for unregister */
    char          *group_p;    /* ptr to subtree(group) */
    struct dpi_ureg_packet *next_p; /* ptr to next in chain */
};
typedef struct dpi_ureg_packet      snmp_dpi_ureg_packet;
#define snmp_dpi_ureg_packet_NULL_p ((snmp_dpi_ureg_packet *)0)
```

Parameters

reason_code

The reason for the unregister.

See “DPI UNREGISTER reason codes” on page 103 for reason codes.

group_p

A pointer to a null-terminated character string that represents the subtree to be unregistered. This subtree must have a trailing period.

next_p

A pointer to a possible next snmp_dpi_ureg_packet structure. If this next field contains the NULL pointer, this is the end of the chain. Currently, multiple unregister requests are not supported in one DPI packet, so this field should always be 0.

Usage

The snmp_dpi_ureg_packet structure represents a parse tree for a DPI UNREGISTER request.

The snmp_dpi_ureg_packet structure is normally created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_UNREGISTER. The snmp_dpi_hdr structure then contains a pointer to an snmp_dpi_ureg_packet structure.

The DPI programmer is advised to use the mkDPIunregister() function to create a DPI UNREGISTER packet.

Context

“The pDPIpacket() function” on page 76

“The mkDPIunregister() function” on page 75

“The snmp_dpi_hdr structure” on page 93

The snmp_dpi_u64 structure

Format

```
struct snmp_dpi_u64 {           /* for unsigned 64-bit int */
    unsigned long high;         /* - high order 32 bits   */
    unsigned long low;          /* - low order  32 bits   */
};
typedef struct snmp_dpi_u64     snmp_dpi_u64;
```

Note: This structure is supported only in SNMP Version 2.

Parameters

high The high order, most significant, 32 bits.

low The low order, least significant, 32 bits.

Usage

The snmp_dpi_u64 structure represents an unsigned 64-bit integer as needed for values with a type of SNMP_TYPE_Counter64.

The snmp_dpi_u64 structure might be created as a result of a call to pDPIpacket(). This is the case if the DPI packet is of type SNMP_DPI_SET and one of the values has a type of SNMP_TYPE_Counter64. The value_p pointer of the snmp_dpi_set_packet structure will then point to an snmp_dpi_u64 structure.

The DPI programmer must also use an snmp_dpi_u64 structure as the parameter to an mkDPIset() call if you want to create a value of type SNMP_TYPE_Counter64.

Context

“The pDPIpacket() function” on page 76

“The snmp_dpi_set_packet structure” on page 97

“DPI SNMP value types” on page 103

“Value representation” on page 104

Character set selection

The version of DPI Version 2.0 shipped with SNMP requires use of the EBCDIC character set. Any DisplayString MIB objects known to the agent (in its compiled MIB) supplied with SNMP will have ASCII conversion handled by the agent. The subagent will always deal with the values of these objects in EBCDIC. Any portion of an instance identifier that is a DisplayString must be in ASCII. The agent does not handle instance IDs.

When the DPI subagent sends a DPI OPEN packet, it must specify the character set that it wants to use. The subagent here needs to know or determine in an implementation dependent manner if the agent is running on a system with the same character set as the subagent. If you connect to the agent at loopback or your own machine, you might assume that you are using the same character set.

The DPI subagent has two choices:

DPI_NATIVE_CSET

Specifies that you want to use the native character set of the platform on which the agent that you connect to is running.

DPI_ASCII_CSET

Specifies that you want to use the ASCII character set. The agent will not translate between ASCII and the native character set.

Although you can specify ASCII, the SNMP agent does not support it.

The DPI packets have a number of fields that are represented as strings. The fields that must be represented in the selected character set are:

- The null-terminated string pointed to by the *description_p*, *enterprise_p*, *group_p*, *instance_p*, and *oid_p* parameters in the various *mkDPIxxx(...)* functions.
- The string pointed to by the *value_p* parameter in the *mkDPIset(...)* function, that is if the *value_type* parameter specifies that the value is an *SNMP_TYPE_DisplayString* or an *SNMP_TYPE_OBJECT_IDENTIFIER*.
- The null-terminated string pointed to by the *description_p*, *enterprise_p*, *group_p*, *instance_p*, and *oid_p* pointers in the various *snmp_dpi_xxxx_packet* structures.
- The string pointed to by the *value_p* pointer in the *snmp_dpi_set_packet* structure, that is if the *value_type* field specifies that the value is an *SNMP_TYPE_DisplayString* or an *SNMP_TYPE_OBJECT_IDENTIFIER*.

Related information

“The *mkDPIopen()* function” on page 65

Constants, values, return codes, and include file

This section describes all the constants and names for values as they are defined in the *snmp_dpi.h* include file (see “The *snmp_dpi.h* include file” on page 106):

“DPI CLOSE reason codes” on page 102

“DPI packet types” on page 102

“DPI RESPONSE error codes” on page 102

“DPI UNREGISTER reason codes” on page 103

“DPI SNMP value types” on page 103

“Value representation” on page 104

“Value ranges and limits” on page 105

“Return codes from DPI transport-related functions” on page 105

DPI CLOSE reason codes

The currently defined DPI CLOSE reason codes as defined in the `snmp_dpi.h` include file are:

```
#define SNMP_CLOSE_otherReason      1
#define SNMP_CLOSE_goingDown        2
#define SNMP_CLOSE_unsupportedVersion 3
#define SNMP_CLOSE_protocolError    4
#define SNMP_CLOSE_authenticationFailure 5
#define SNMP_CLOSE_byManager        6
#define SNMP_CLOSE_timeout          7
#define SNMP_CLOSE_openError        8
```

These codes are used in the *reason_code* parameter for the *mkDPIClose()* function and in the *reason_code* field in the *snmp_dpi_close_packet* structure.

Related information

“The *snmp_dpi_close_packet* structure” on page 91

“The *mkDPIClose()* function” on page 64

DPI packet types

The currently defined DPI packet types as defined in the `snmp_dpi.h` include file are:

```
#define SNMP_DPI_GET      1
#define SNMP_DPI_GET_NEXT 2 /* old DPI Version 1.x style */
#define SNMP_DPI_GETNEXT 2
#define SNMP_DPI_SET      3
#define SNMP_DPI_TRAP     4
#define SNMP_DPI_RESPONSE 5
#define SNMP_DPI_REGISTER 6
#define SNMP_DPI_UNREGISTER 7
#define SNMP_DPI_OPEN     8
#define SNMP_DPI_CLOSE    9
#define SNMP_DPI_COMMIT   10
#define SNMP_DPI_UNDO     11
#define SNMP_DPI_GETBULK  12
#define SNMP_DPI_TRAPV2   13 /* reserved, not implmented */
#define SNMP_DPI_INFORM   14 /* reserved, not implemented */
#define SNMP_DPI_ARE_YOU_THERE 15
```

These packet types are used in the *type* parameter for the *packet_type* field in the *snmp_dpi_hdr* structure.

Related information

“The *snmp_dpi_hdr* structure” on page 93

DPI RESPONSE error codes

In case of an error on an SNMP request like GET, GETNEXT, SET, COMMIT, or UNDO, the RESPONSE can have one of these currently defined error codes. They are defined in the `snmp_dpi.h` include file:

```
#define SNMP_ERROR_noError      0
#define SNMP_ERROR_tooBig       1
#define SNMP_ERROR_noSuchName   2
#define SNMP_ERROR_badValue     3
#define SNMP_ERROR_readOnly     4
#define SNMP_ERROR_genErr       5
```

```

#define SNMP_ERROR_noAccess          6
#define SNMP_ERROR_wrongType         7
#define SNMP_ERROR_wrongLength       8
#define SNMP_ERROR_wrongEncoding     9
#define SNMP_ERROR_wrongValue        10
#define SNMP_ERROR_noCreation         11
#define SNMP_ERROR_inconsistentValue  12
#define SNMP_ERROR_resourceUnavailable 13
#define SNMP_ERROR_commitFailed       14
#define SNMP_ERROR_undoFailed         15
#define SNMP_ERROR_authorizationError 16
#define SNMP_ERROR_notWritable        17
#define SNMP_ERROR_inconsistentName   18

```

In case of an error on a DPI only request (OPEN, REGISTER, UNREGISTER, ARE_YOU_THERE), the RESPONSE can have one of these currently defined error codes. They are defined in the `snmp_dpi.h` include file:

```

#define SNMP_ERROR_DPI_noError          0
#define SNMP_ERROR_DPI_otherError       101
#define SNMP_ERROR_DPI_notFound         102
#define SNMP_ERROR_DPI_alreadyRegistered 103
#define SNMP_ERROR_DPI_higherPriorityRegistered 104
#define SNMP_ERROR_DPI_mustOpenFirst    105
#define SNMP_ERROR_DPI_notAuthorized     106
#define SNMP_ERROR_DPI_viewSelectionNotSupported 107
#define SNMP_ERROR_DPI_getBulkSelectionNotSupported 108
#define SNMP_ERROR_DPI_duplicateSubAgentIdentifier 109
#define SNMP_ERROR_DPI_invalidDisplayString 110
#define SNMP_ERROR_DPI_characterSetSelectionNotSupported 111

```

These codes are used in the *error_code* parameter for the *mkDPInresponse()* function and in the *error_code* field in the *snmp_dpi_resp_packet* structure.

Related information

“The *snmp_dpi_resp_packet* structure” on page 96

“The *mkDPInresponse()* function” on page 69

DPI UNREGISTER reason codes

These are the currently defined DPI UNREGISTER reason codes. They are defined in the `snmp_dpi.h` include file:

```

#define SNMP_UNREGISTER_otherReason    1
#define SNMP_UNREGISTER_goingDown      2
#define SNMP_UNREGISTER_justUnregister 3
#define SNMP_UNREGISTER_newRegistration 4
#define SNMP_UNREGISTER_higherPriorityRegistered 5
#define SNMP_UNREGISTER_byManager      6
#define SNMP_UNREGISTER_timeout        7

```

These codes are used in the *reason_code* parameter for the *mkDPInunregister()* function and in the *reason_code* field in the *snmp_dpi_ureg_packet* structure.

Related information

“The *snmp_dpi_ureg_packet* structure” on page 99

“The *mkDPInunregister()* function” on page 75

DPI SNMP value types

These are the currently defined value types as defined in the `snmp_dpi.h` include file:

```

#define SNMP_TYPE_MASK          0x7f /* mask to isolate type*/
#define SNMP_TYPE_Integer32     (128|1) /* 32-bit INTEGER */
#define SNMP_TYPE_OCTET_STRING  2 /* OCTET STRING */
#define SNMP_TYPE_OBJECT_IDENTIFIER 3 /* OBJECT IDENTIFIER */
#define SNMP_TYPE_NULL          4 /* NULL, no value */
#define SNMP_TYPE_IpAddress      5 /* IMPLICIT OCTETSTRING*/
#define SNMP_TYPE_Counter32     (128|6) /* 32-bit Counter */
#define SNMP_TYPE_Gauge32       (128|7) /* 32-bit Gauge */
#define SNMP_TYPE_TimeTicks     (128|8) /* 32-bit TimeTicks in */
                                   /* hundredths of a sec */
#define SNMP_TYPE_DisplayString  9 /* DisplayString (TC) */
#define SNMP_TYPE_BIT_STRING    10 /* BIT STRING */
#define SNMP_TYPE_NsapAddress    11 /* IMPLICIT OCTETSTRING*/
#define SNMP_TYPE_UInteger32    (128|12) /* 32-bit INTEGER */
#define SNMP_TYPE_Counter64     13 /* 64-bit Counter */
#define SNMP_TYPE_Opaque        14 /* IMPLICIT OCTETSTRING*/
#define SNMP_TYPE_noSuchObject  15 /* IMPLICIT NULL */
#define SNMP_TYPE_noSuchInstance 16 /* IMPLICIT NULL */
#define SNMP_TYPE_endOfMibView  17 /* IMPLICIT NULL */

```

These value types are used in the *value_type* parameter for the *mkDPIset()* function and in the *value_type* field in the *snmp_dpi_set_packet* structure.

Related information

- “The *snmp_dpi_set_packet* structure” on page 97
- “The *mkDPIset()* function” on page 71
- “Value representation” on page 104
- “Value ranges and limits” on page 105

Value representation

Values in the *snmp_dpi_set_packet* structure are represented as follows:

- 32-bit integers are defined as long int or unsigned long int. A long int is assumed to be 4 bytes.
- 64-bit integers are represented as an *snmp_dpi_u64*.
Unsigned 64 bit integers are only dealt with in SNMP. In a structure that has two fields, the high order piece and the low order piece, each is of type unsigned long int. These are assumed to be 4 bytes.
- Object identifiers are null-terminated strings in the selected character set, representing the OID in ASN.1 dotted-decimal notation. The length includes the terminating NULL.

An ASCII example:

```
'312e332e362e312e322e312e312e312e3000'h
```

represents "1.3.6.1.2.1.1.0" which is sysDescr.0.

An EBCDIC example:

```
'f14bf34bf64bf14bf24bf14bf14bf14bf000'h
```

represents "1.3.6.1.2.1.1.0" which is sysDescr.0.

- DisplayStrings are in the selected character set. The length specifies the length of the string.

An ASCII example:

```
'6162630d0a'h
```

represents "abc\r\n", no NULL.

An EBCDIC example:

```
'8182830d25'h
```

represents "abc\r\n", no NULL.

- IpAddress and Opaque are implicit OCTET_STRING, so they are a sequence of octets or bytes. This means, for instance, that the IP address is in network byte order.
- NULL has a 0 length for the value, no value data, so a NULL pointer is returned in the *value_p* field.
- noSuchObject, noSuchInstance, and endOfMibView are implicit NULL and are represented as such.
- BIT_STRING is an OCTET_STRING of the form uubbbb...bb, where the first octet (uu) is 0x00-0x07 and indicates the number of unused bits in the last octet (bb). The bb octets represent the bit string itself, where bit 0 comes first and so on.

Related information

“Value ranges and limits” on page 105

Value ranges and limits

The following rules apply to object IDs in ASN.1 notation:

- The object ID consists of 1 to 128 subIDs, which are separated by periods.
- Each subID is a positive number. No negative numbers are allowed.
- The value of each number cannot exceed 4294967295. This value is 2 to the power of 32 minus 1.
- The valid values of the first subID are 0, 1, or 2.
- If the first subID has a value of 0 or 1, the second subID can only have a value of 0 through 39.

The following rules apply to DisplayString:

- A DisplayString (Textual Convention) is basically an OCTET STRING in SNMP terms.
- The maximum size of a DisplayString is 255 octets or bytes.

More information on the DPI SNMP value types can be found in the SNMP Structure of Management Information (SMI) and SNMP Textual Conventions (TC) RFCs. These two RFCs are RFC 1902 and RFC 1903. See Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs.

Return codes from DPI transport-related functions

These are the currently defined values for the return codes from DPI transport-related functions. They are defined in the `snmp_dpi.h` include file:

```
#define DPI_RC_OK                0 /* all OK, no error          */
#define DPI_RC_NOK               -1 /* some other error         */
#define DPI_RC_NO_PORT           -2 /* can't determine DPIport */
#define DPI_RC_NO_CONNECTION     -3 /* no connection to DPIagent*/
#define DPI_RC_EOF               -4 /* EOF received on connection*/
#define DPI_RC_IO_ERROR          -5 /* Some I/O error on connect*/
#define DPI_RC_INVALID_HANDLE    -6 /* unknown/invalid handle  */
#define DPI_RC_TIMEOUT           -7 /* timeout occurred         */
#define DPI_RC_PACKET_TOO_LARGE -8 /* packed too large, dropped*/
#define DPI_RC_UNSUPPORTED_DOMAIN -9 /* unsupported domain for connect*/
#define DPI_RC_INVALID_ARGUMENT -10 /* invalid argument passed*/
```

These values are used as return codes for the transport-related DPI functions.

Related information

- “The `DPIconnect_to_agent_TCP()` function” on page 80
- “The `DPIconnect_to_agent_UNIXstream()` function” on page 82
- “The `DPIawait_packet_from_agent()` function” on page 78
- “The `DPIsend_packet_to_agent()` function” on page 86

The `snmp_dpi.h` include file

```
#include <snmp_dpi.h>
```

Parameters

None

Description

The `snmp_dpi.h` include file defines the SNMP DPI API to the DPI subagent programmer. It has all the function prototype statements, and it also has the definitions for the `snmp_dpi` structures.

The same include file is used at the agent side, so you will see some definitions that are unique to the agent side. Also, other functions or prototypes of functions not implemented on SNMP might exist. Therefore, only use the API as it is documented in this manual.

Related information

Macros, functions, structures, constants, and values defined in the `snmp_dpi.h` include file are:

- “The `DPIawait_packet_from_agent()` function” on page 78
- “The `DPIconnect_to_agent_TCP()` function” on page 80
- “The `DPIconnect_to_agent_UNIXstream()` function” on page 82
- “The `DPIdebug()` function” on page 59
- “The `DPIdisconnect_from_agent()` function” on page 84
- “The `DPI_PACKET_LEN()` macro” on page 60
- “The `DPIsend_packet_to_agent()` function” on page 86
- “The `fDPIparse()` function” on page 61
- “The `fDPIset()` function” on page 62
- “The `mkDPIAreYouThere()` function” on page 63
- “The `mkDPIclose()` function” on page 64
- “The `mkDPIopen()` function” on page 65
- “The `mkDPIregister()` function” on page 67
- “The `mkDPIresponse()` function” on page 69
- “The `mkDPIset()` function” on page 71
- “The `mkDPItrap()` function” on page 73
- “The `mkDPIunregister()` function” on page 75
- “The `pDPIpacket()` function” on page 76
- “The `snmp_dpi_close_packet` structure” on page 91
- “The `snmp_dpi_get_packet` structure” on page 92
- “The `snmp_dpi_next_packet` structure” on page 95

- “The `snmp_dpi_hdr` structure” on page 93
- “The `lookup_host()` function” on page 88
- “The `snmp_dpi_resp_packet` structure” on page 96
- “The `snmp_dpi_set_packet` structure” on page 97
- “The `snmp_dpi_ureg_packet` structure” on page 99
- “DPI CLOSE reason codes” on page 102
- “DPI packet types” on page 102
- “DPI RESPONSE error codes” on page 102
- “DPI UNREGISTER reason codes” on page 103
- “DPI SNMP value types” on page 103
- “Character set selection” on page 101

A DPI subagent example

This is an example of a DPI version 2.0 subagent. The code is called `dpi_mvs_sample.c` in the `/usr/lpp/tcpip/samples` directory.

Note: The example code in this document was copied from the sample file at the time of the publication. There may be differences in the code presented and the code that is shipped with the product. Always use the code provided in the `/usr/lpp/tcpip/samples` directory as the authoritative sample code.

The DPI subagent example includes:

- “Overview of subagent processing” on page 107
- “Connecting to the agent” on page 109
- “Registering a subtree with the agent” on page 111
- “Processing requests from the agent” on page 113
- “Processing a GET request” on page 116
- “Processing a GETNEXT request” on page 119
- “Processing a SET/COMMIT/UNDO request” on page 122
- “Processing an UNREGISTER request” on page 125
- “Processing a CLOSE request” on page 126
- “Generating a TRAP” on page 126

Related information

“Subagent programming concepts” on page 48

Overview of subagent processing

This overview assumes that the subagent communicates with the agent over a TCP connection. Other connection implementations are possible and, in that case, the processing approach may be a bit different.

In this overview, the agent will be requested to send at most one `varBind` per DPI packet, so there will be no need to loop through a list of `varBinds`. Potentially, you may gain performance improvements if you allow for multiple `varBinds` per DPI packet on GET, GETNEXT, SET requests, but to do so, your code will have to loop through the `varBind` list and so it becomes more complicated. The DPI subagent programmer can handle that once you understand the basics of the DPI API.

The following are the supported MIB variable definitions for `DPI_SIMPLE`:

```

DPISimple-MIB DEFINITIONS ::= BEGIN

    IMPORTS
        MODULE-IDENTITY, OBJECT-TYPE, snmpModules, enterprises
            FROM SNMPv2-SMI
        DisplayString
            FROM SNMPv2-TC

    ibm      OBJECT IDENTIFIER ::= { enterprises 2 }
    ibmDPI   OBJECT IDENTIFIER ::= { ibm 2 }
    dpi20MIB OBJECT IDENTIFIER ::= { ibmDPI 1 }

    -- dpiSimpleMIB MODULE-IDENTITY
    --   LAST-UPDATED "9401310000Z"

    --   DESCRIPTION
    --       "The MIB module describing DPI Simple Objects for
    --       the dpi_samp.c program"
    --   ::= { snmpModules x }

    dpiSimpleMIB OBJECT IDENTIFIER ::= { dpi20MIB 5 }

    dpiSimpleInteger      OBJECT-TYPE
        SYNTAX  INTEGER
        ACCESS   read-only
        STATUS   mandatory
        DESCRIPTION
            "A sample integer32 value"
        ::= { dpiSimpleMIB 1 }

    dpiSimpleString      OBJECT-TYPE
        SYNTAX  DisplayString
        ACCESS   read-write
        STATUS   mandatory
        DESCRIPTION
            "A sample Display String"
        ::= { dpiSimpleMIB 2 }

    dpiSimpleCounter32    OBJECT-TYPE
        SYNTAX  Counter    -- Counter32 is SNMPv2
        ACCESS   read-only
        STATUS   mandatory
        DESCRIPTION
            "A sample 32-bit counter"
        ::= { dpiSimpleMIB 3 }

    dpiSimpleCounter64    OBJECT-TYPE
        SYNTAX  Counter    -- Counter64 is SNMPv2,
                           -- No SMI support for it yet
        ACCESS   read-only
        STATUS   mandatory
        DESCRIPTION
            "A sample 64-bit counter"
        ::= { dpiSimpleMIB 4 }

END

```

To make the code more readable, the following names have been defined in our `dpi_mvs_sample.c` source file.

```

#define DPI_SIMPLE_SUBAGENT "1.3.6.1.4.1.2.2.1.5"
#define DPI_SIMPLE_MIB      "1.3.6.1.4.1.2.2.1.5."
#define DPI_SIMPLE_INTEGER  "1.0"      /* dpiSimpleInteger.0 */
#define DPI_SIMPLE_STRING   "2.0"      /* dpiSimpleString.0 */
#define DPI_SIMPLE_COUNTER32 "3.0"      /* dpiSimpleCounter32.0 */
#define DPI_SIMPLE_COUNTER64 "4.0"      /* dpiSimpleCounter64.0 */

```


In addition, the following variables have been defined as global variables in our `dpi_mvs_sample.c` source file.

```
static int                                     /*handle has global scope */
int global_role=0;                             /*flag for debug macros */
static int      instance_level = 0;
static long int value1      = 5;
#define         value2_p     cur_val_p  /* writable object */
#define         value2_len   cur_val_len /* writable object */
static char     *cur_val_p   = (char *)0;
static char     *new_val_p   = (char *)0;
static char     *old_val_p   = (char *)0;
static unsigned long cur_val_len = 0;
static unsigned long new_val_len = 0;
static unsigned long old_val_len = 0;
static unsigned long value3     = 1;
#ifdef EXCLUDE_SNMP_SMIV2_SUPPORT
static snmp_dpi_u64 value4      = {0x80000000,1L};
#endif/*ndef EXCLUDE_SNMP_SMIV2_SUPPORT*/
static int      unix_sock =0; /*default use TCP */
static unsigned short timeout = 3; /*default timeout */
```

Connecting to the agent

Before a subagent can receive or send any DPI packets from or to the SNMP DPI-capable agent, it must connect to the agent and identify itself to the agent.

The following example code returns a response. It is assumed that there are no errors in the request, but proper code should do the checking for that. Proper checking is done for lexicographic next object, but no checking is done for `ULONG_MAX`, or making sure that the instance ID is indeed valid (digits and periods). If the code gets to the end of our `dpiSimpleMIB`, an `endOfMibView` must be returned as defined by the SNMP Version 2 rules. You will need to specify:

- A host name or IP address in dotted decimal notation that specifies where the agent is running. Often the name *loopback* can be used if the subagent runs on the same system as the agent.
- A community name that is used to obtain the dpi TCP port from the agent. Internally that is done by sending a regular SNMP GET request to the agent. In an open environment, the well-known community name *public* can probably be used.

The function returns a negative error code if an error occurs. If the connection setup is successful, it returns a handle that represents the connection and that must be used on subsequent calls to send or await DPI packets.

The second step is to identify the subagent to the agent. This is done by making a DPI-OPEN packet, sending it to the agent, and then awaiting the response from the agent. The agent may accept or deny the OPEN request. Making a DPI-OPEN packet is done by calling `mkDPIopen()`, which expects the following parameters:

- A unique subagent identification (an object identifier).
- A description, which can be the NULL string (`""`).
- Overall subagent timeout in seconds. The agent uses this value as a timeout value for a response when it sends a request to the subagent. The agent may have a maximum value for this timeout that will be used if you exceed it.
- The maximum number of varBinds per DPI packet that the subagent is willing or is able to handle.
- The desired character set. In most cases you want to use the native character set.
- Length of a password. A 0 means no password.

- Pointer to the password or NULL if no password. It depends on the agent if subagents must specify a password to open up a connection.

The function returns a pointer to a static buffer holding the DPI packet if successful. If it fails, it returns a NULL pointer.

When the DPI-OPEN packet has been created, you must send it to the agent. You can use the `DPIsend_packet_to_agent()` function, which expects the following parameters:

- The handle of a connection from `DPIconnect_to_agent_TCP`.
- A pointer to the DPI packet from `mkDPIopen`.
- The length of the packet. The `snmp_dpi.h` include file provides a macro `DPI_PACKET_LEN` that calculates the packet length of a DPI packet.

This function returns `DPI_RC_OK` (value 0) if successful. Otherwise, an appropriate `DPI_RC_xxxx` error code as defined in `snmp_dpi.h` is returned.

Now wait for a response to the DPI-OPEN. To await such a response, you call the `DPIawait_packet_from_agent()` function, which expects the following parameters:

- The handle of a connection from `DPIconnect_to_agent_TCP`.
- A timeout in seconds, which is the maximum time to wait for response.
- A pointer to a pointer, which will receive a pointer to a static buffer containing the awaited DPI packet. If the system fails to receive a packet, a NULL pointer is stored.
- A pointer to a long integer (32-bit), which will receive the length of the awaited packet. If it fails, it will be set to 0.

This function returns `DPI_RC_OK` (value 0) if successful. Otherwise, an appropriate `DPI_RC_xxxx` error code as defined in `snmp_dpi.h` is returned.

The last step is to ensure that you received a DPI-RESPONSE back from the agent. If so, ensure that the agent accepted you as a valid subagent. This will be shown by the `error_code` field in the DPI response packet.

The following example code establishes a connection and opens it by identifying you to the agent.

```
static void do_connect_and_open(char *hostname_p, char *community_p)
{
    unsigned char *packet_p;
    int rc;
    unsigned long length;
    snmp_dpi_hdr *hdr_p;

#ifdef MVS
    __etoa(community_p);          /* Translate to ASCII */
#endif /* MVS */

#ifdef DPI_MINIMAL_SUBAGENT
#ifdef INCLUDE_UNIX_DOMAIN_FOR_DPI
    if (unix_sock) {
        handle =
            DPIconnect_to_agent_UNIXstream( /* (UNIX) connect to */
                hostname_p,                /* agent on this host */
                community_p);              /* snmp community name */
    } else
#endif /* def INCLUDE_UNIX_DOMAIN_FOR_DPI */
#endif /* def DPI_MINIMAL_SUBAGENT */
    handle =
```

```

        DPIconnect_to_agent_TCP( /* (TCP) connect to agent */
            hostname_p,          /* on this host */
            community_p);        /* snmp community name */

    if (handle < 0) exit(1);      /* If it failed, exit */

    packet_p = mkDPIopen(        /* Make DPI-OPEN packet */
        DPI_SIMPLE_SUBAGENT,    /* Our identification */
        "Simple DPI subAgent",  /* description */
        10L,                    /* Our overall timeout */
        1L,                      /* max varBinds/packet */
        DPI_NATIVE_CSET,        /* native character set */
        0L,                      /* password length */
        (unsigned char *)0);    /* ptr to password */

    if (!packet_p) exit(1);      /* If it failed, exit */

    rc = DPIsend_packet_to_agent( /* send OPEN packet */
        handle,                  /* on this connection */
        packet_p,                /* this is the packet */
        DPI_PACKET_LEN(packet_p)); /* and this is its length */

    if (rc != DPI_RC_OK) exit(1); /* If it failed, exit */

    rc = DPIawait_packet_from_agent( /* wait for response */
        handle,                  /* on this connection */
        10,                      /* timeout in seconds */
        packet_p,                /* receives ptr to packet */
        length);                 /* receives packet length */

    if (rc != DPI_RC_OK) exit(1); /* If it failed, exit */

    hdr_p = pDPIpacket(packet_p); /* parse DPI packet */
    if (hdr_p == snmp_dpi_hdr_NULL_p) /* If we fail to parse it */
        exit(1);                  /* then exit */

    if (hdr_p->packet_type != SNMP_DPI_RESPONSE) exit(1);

    rc = hdr_p->data_u.resp_p->error_code;
    if (rc != SNMP_ERROR_DPI_noError) exit(1);

} /* end of do_connect_and_open() */

```

Registering a subtree with the agent

After setting up a connection to the agent and identifying yourself, register one or more MIB subtrees or instances for which you want to be responsible to handle SNMP requests.

To do so, the subagent must create a DPI-REGISTER packet and send it to the agent. The agent will then send a response to indicate success or failure of the register request.

To create a DPI-REGISTER packet, the subagent uses a call to the `mkDPIregister()` function, which expects these parameters:

- A timeout value in seconds for this subtree. If you specify 0, your overall timeout value that was specified in DPI-OPEN is used. You can specify a different value if you expect longer processing time for a specific subtree.
- A requested priority. Multiple subagents may register the same subtree at different priorities. For example, 0 is better than 1 and so on. The agent considers the subagent with the best priority to be the active subagent for the

subtree. If you specify -1, you are asking for the best priority available. If you specify 0, you are asking for a better priority than any existing subagent may already have.

- The MIB subtree or instance that you want to control. For object level registration, this group ID must have a trailing dot. For instance level registration, this group ID would simply have the instance number follow the object number subtree.
- You have no choice in GETBULK processing. You must ask the agent to map a GETBULK into multiple GETNEXT packets.

The function returns a pointer to a static buffer holding the DPI packet if successful. If it fails, it returns a NULL pointer.

Now send this DPI-REGISTER packet to the agent with the `DPIsend_packet_to_agent()` function. This is similar to sending the DPI_OPEN packet. Then wait for a response from the agent. Again, use the `DPIawait_packet_from_agent()` function in the same way as you awaited a response on the DPI-OPEN request. Once you have received the response, check the return code to ensure that registration was successful.

The following code example demonstrates how to register one MIB subtree with the agent.

```
static void do_register(void)
{
    unsigned char *packet_p;
    int rc;
    unsigned long length;
    snmp_dpi_hdr *hdr_p;
    int i;
    char buf 512 ;

    for (i=0; i<4; i++) {

        strcpy(buf,DPI_SIMPLE_MIB);
        if (instance_level) {
            switch (i) {
                case 0:
                    strcat(buf,DPI_SIMPLE_INTEGER);
                    break;
                case 1:
                    strcat(buf,DPI_SIMPLE_STRING);
                    break;
                case 2:
                    strcat(buf,DPI_SIMPLE_COUNTER32);
                    break;
                case 3:
                    strcat(buf,DPI_SIMPLE_COUNTER64);
                    break;
            } /* endswitch */
        }
        packet_p = mkDPIregister(           /* Make DPIregister packet */
            timeout,                       /* timeout in seconds */
            0,                             /* requested priority */
            buf,                           /* ptr to the subtree */
            DPI_BULK_NO);                  /* Map GetBulk into GetNext*/

        if (!packet_p) exit(1);             /* If it failed, exit */

        rc = DPIsend_packet_to_agent(      /* send REGISTER packet */
            handle,                        /* on this connection */
            packet_p,                      /* this is the packet */
            DPI_PACKET_LEN(packet_p));     /* and this is its length */
    }
}
```

```

        if (rc != DPI_RC_OK) exit(1);          /* If it failed, exit */

        rc = DPIawait_packet_from_agent( /* wait for response */
            handle,                       /* on this connection */
            10,                          /* timeout in seconds */
            &packet_p,                   /* receives ptr to packet */
            &length);                   /* receives packet length */

        if (rc != DPI_RC_OK) exit(1);          /* If it failed, exit */

        hdr_p = pDPIpacket(packet_p);        /* parse DPI packet */
        if (hdr_p == snmp_dpi_hdr_NULL_p) /* If we fail to parse it */
            exit(1);                      /* then exit */

        if (hdr_p->packet_type != SNMP_DPI_RESPONSE) exit(1);

        rc = hdr_p->data_u.resp_p->error_code;
        if (rc != SNMP_ERROR_DPI_noError) exit(1);

        if (!instance_level) break;

    } /* endfor */

} /* end of do_register() */

```

Processing requests from the agent

After registering your sample MIB subtree with the agent, expect that SNMP requests for that subtree will be passed back to you for processing. Since the requests will arrive in the form of DPI packets on the connection that you have established, go into a While loop to await DPI packets from the agent.

Because the subagent cannot know in advance which kind of packet arrives from the agent, await a DPI packet (forever), then parse the packet, check the packet type, and process the request based on the DPI packet type. A call to `pDPIpacket`, which expects as parameter a pointer to the encoded or serialized DPI packet, returns a pointer to a DPI parse tree. The pointer points to an `snmp_dpi_hdr` structure which looks as follows:

```

struct snmp_dpi_hdr {
    unsigned char  proto_major;
    unsigned char  proto_version;
    unsigned char  proto_release;
    unsigned short packet_id;
    unsigned char  packet_type;
    union {
        snmp_dpi_reg_packet      *reg_p;
        snmp_dpi_ureg_packet     *ureg_p;
        snmp_dpi_get_packet      *get_p;
        snmp_dpi_next_packet     *next_p;
        snmp_dpi_next_packet     *bulk_p;
        snmp_dpi_set_packet      *set_p;
        snmp_dpi_resp_packet     *resp_p;
        snmp_dpi_trap_packet     *trap_p;
        snmp_dpi_open_packet     *open_p;
        snmp_dpi_close_packet    *close_p;
        unsigned char            *any_p;
    } data_u;
};
typedef struct snmp_dpi_hdr      snmp_dpi_hdr;
#define snmp_dpi_hdr_NULL_p    ((snmp_dpi_hdr *)0)

```

With the DPI parse tree, you decide how to process the DPI packet. The following code example demonstrates the high-level process of a DPI subagent.

```
main(int argc, char *argv[], char *envp)[{} [] []]
{
    unsigned char *packet_p;
    int i = 0;
    int rc = 0;
#ifdef DPI_VERY_MINIMAL_SUBAGENT /* with VERY minimal agent */
    int debug = 0;
#endif /* ndef DPI_VERY_MINIMAL_SUBAGENT */
    unsigned long length;
    snmp_dpi_hdr *hdr_p;
    char *hostname_p = NULL; /* @L1C*/
    char *community_p = SNMP_COMMUNITY;
    char *cmd_p = "";
    char hostname[MAX_HOSTNAME_LEN+1]; /* @L1A*/

    if (argc >= 1) cmd_p = argv[0];

    for (i=1; i < argc; i++) {
        if (strcmp(argv[i], "-h") == 0) {
            if (i+1 >= argc) {
                printf("Need hostname\n\n");
                usage(cmd_p);
            } /* endif */
            hostname_p = argv[++i];
#ifdef DPI_VERY_MINIMAL_SUBAGENT /* with VERY minimal agent */
        } else if (strcmp(argv[i], "-c") == 0) {
            if (i+1 >= argc) {
                printf("Need community name\n\n");
                usage(cmd_p);
            } /* endif */
            community_p = argv[++i];
#endif INCLUDE_UNIX_DOMAIN_FOR_DPI
        } else if (strcmp(argv[i], "-unix") == 0) {
            unix_sock = 1;
#ifdef INCLUDE_UNIX_DOMAIN_FOR_DPI /* def INCLUDE_UNIX_DOMAIN_FOR_DPI */
        } else if (strcmp(argv[i], "-ireg") == 0) {
            instance_level = 1;
        } else if (strcmp(argv[i], "-d") == 0) {
            if (i+1 >= argc) {
                debug = 1;
                continue;
            }
            if ((strlen(argv[i+1]) == 1) && isdigit(*argv[i+1])) {
                i++;
                debug = atoi(argv[i]);
            } else {
                debug = 1;
            } /* endif */
#endif /* ndef DPI_VERY_MINIMAL_SUBAGENT */
        } else {
            usage(cmd_p);
        } /* endif */
    } /* endfor */

#ifdef DPI_VERY_MINIMAL_SUBAGENT
    if (debug) {
        printf("\n%s - %s\n", __FILE__, VERSION);
        DPIdbg(debug); /* turn on DPI debugging */
        timeout += 6; /* longer timeout please */
    } /* endif */
#endif /* ndef DPI_VERY_MINIMAL_SUBAGENT */

    if (hostname_p == NULL) { /* -h not specified. Try to
                             obtain local host name
```

```

                                                                    @L1A*/
if (gethostname(hostname, MAX_HOSTNAME_LEN) != 0) {
    printf("\ngethostname failed. "
           "Restart with -h parameter.\n\n");
    exit(1);
}
else {
    hostname_p = hostname;
}
                                                                    /* @L1A*/
                                                                    /* @L1A*/
                                                                    /* @L1A*/
                                                                    /* -h not specified @L1A*/

/* first init value2_p, our dpiSimpleString (DisplayString) */
/* since we treat it as display string keep terminating NULL */
value2_p = (char *) malloc(strlen("Initial String")+1);
if (value2_p) {
    memcpy(value2_p, "Initial String", strlen("Initial String")+1);
    value2_len = strlen("Initial String")+1;
} /* endif */

do_connect_and_open(hostname_p,
                    community_p); /* connect and DPI-OPEN */

do_register(); /* register our subtree */

do_trap(); /* issue a trap as sample */

while (rc == 0) { /* do forever */
    rc = DPIawait_packet_from_agent( /* wait for a DPI packet */
        handle, /* on this connection */
        -1, /* wait forever */
        &packet_p, /* receives ptr to packet */
        &length); /* receives packet length */

    if (rc != DPI_RC_OK) exit(1); /* If it failed, exit */

    hdr_p = pDPIpacket(packet_p); /* parse DPI packet */
    if (hdr_p == snmp_dpi_hdr_NULL_p) /* If we fail to parse it */
        exit(1); /* then exit */

    switch(hdr_p->packet_type) { /* handle by DPI type */
    case SNMP_DPI_GET:
        rc = do_get(hdr_p,
                    hdr_p->data_u.get_p);
        break;
    case SNMP_DPI_GETNEXT:
        rc = do_next(hdr_p,
                    hdr_p->data_u.next_p);
        break;
    case SNMP_DPI_SET:
    case SNMP_DPI_COMMIT:
    case SNMP_DPI_UNDO:
        rc = do_set(hdr_p,
                    hdr_p->data_u.set_p);
        break;
    case SNMP_DPI_CLOSE:
        rc = do_close(hdr_p,
                    hdr_p->data_u.close_p);
        break;
    case SNMP_DPI_UNREGISTER:
        rc = do_unreg(hdr_p,
                    hdr_p->data_u.ureg_p);
        break;
    default:
        printf("Unexpected DPI packet type %d\n",
                hdr_p->packet_type);
        rc = -1;
    } /* endswitch */
}

```

```

        if (rc) exit(1);
    } /* endwhile */

    return(0);
} /* end of main() */

```

Processing a GET request

When the DPI packet is parsed, the `snmp_dpi_hdr` structure will show in the *packet_type* that this is an `SNMP_DPI_GET` packet. In that case, the *packet_body* contains a pointer to a GET-varBind, which is represented in an `snmp_dpi_get_packet` structure:

```

struct dpi_get_packet {
    char          *object_p; /* ptr to OIDstring */
    char          *group_p; /* ptr to sub-tree */
    char          *instance_p; /* ptr to rest of OID */
    struct dpi_get_packet *next_p; /* ptr to next in chain */
};
typedef struct dpi_get_packet      snmp_dpi_get_packet;
#define snmp_dpi_get_packet_NULL_p ((snmp_dpi_get_packet *)0)

```

Assuming you have registered subtree 1.3.6.1.4.1.2.2.1.5 and a GET request comes in for one variable (1.3.6.1.4.1.2.2.1.5.1.0) that is object 1 instance 0 in the subtree, the fields in the `snmp_dpi_get_packet` would have pointers to:

```

object_p  -> "1.3.6.1.4.1.2.2.1.5.1.0"
group_p   -> "1.3.6.1.4.1.2.2.1.5."
instance_p -> "1.0"
next_p    -> snmp_dpi_get_packet_NULL_p

```

If there are multiple varBinds in a GET request, each one is represented in an `snmp_dpi_get_packet` structure and all the `snmp_dpi_get_packet` structures are chained using the next pointer. As long as the next pointer is not the `snmp_dpi_get_packet_NULL_p` pointer, there are more varBinds in the list.

Now you can analyze the varBind structure for whatever checking you want to do. When you are ready to make a response that contains the value of the variable, you prepare a SET-varBind, which is represented in an `snmp_dpi_set_packet` structure:

```

struct dpi_set_packet {
    char          *object_p; /* ptr to OIDstring */
    char          *group_p; /* ptr to sub-tree */
    char          *instance_p; /* ptr to rest of OID */
    unsigned char  value_type; /* SNMP_TYPE_xxxx */
    unsigned short value_len; /* value length */
    char          *value_p; /* ptr to value itself */
    struct dpi_set_packet *next_p; /* ptr to next in chain */
};
typedef struct dpi_set_packet      snmp_dpi_set_packet;
#define snmp_dpi_set_packet_NULL_p ((snmp_dpi_set_packet *)0)

```

You can use the `mkDPIset()` function to prepare such a structure. This function expects the following parameters:

- A pointer to an existing `snmp_dpi_set_packet` structure if the new varBind must be added to an existing chain of varBinds. If this is the first or the only varBind in the chain, pass the `snmp_dpi_set_packet_NULL_p` pointer to indicate this.
- A pointer to the subtree that you registered.
- A pointer to the rest of the OID; in other words, the piece that follows the subtree.

- The value type of the value to be bound to the variable name. This must be one of the SNMP_TYPE_xxxx values as defined in the snmp_dpi.h include file.
- The length of the value. For integer type values, this must be a length of 4. Work with 32-bit signed or unsigned integers except for the Counter64 type. For the Counter64 type, point to an snmp_dpi_u64 structure and pass the length of that structure.
- A pointer to the value.

Memory for the varBind is dynamically allocated and the data itself is copied. So upon return you can dispose of our own pointers and allocated memory as you please. If the call is successful, a pointer is returned as follows:

- To a new snmp_dpi_set_packet if it is the first or only varBind.
- To the existing snmp_dpi_set_packet that you passed on the call. In this case, the new packet has been chained to the end of the varBind list.

If the mkDPISet() call fails, a NULL pointer is returned.

When you have prepared the SET-varBind data, you can create a DPI RESPONSE packet using the mkDPISresponse() function that expects these parameters:

- A pointer to an snmp_dpi_hdr. You should use the header of the parsed incoming packet. It is used to copy the *packet_id* from the request into the response, such that the agent can correlate the response to a request.
- A return code which is an SNMP error code. If successful, this should be SNMP_ERROR_noError (value 0). If failure, it must be one of the SNMP_ERROR_xxxx values as defined in the snmp_dpi.h include file.

A request for a nonexisting object or instance is not considered an error. Instead, you must pass a value type of SNMP_TYPE_noSuchObject or SNMP_TYPE_noSuchInstance respectively. These two value types have an implicit value of NULL, so you can pass a 0 length and a NULL pointer for the value in this case.

- The index of the varBind in error starts counting at 1. Pass 0 if no error occurred, or pass the proper index of the first varBind for which an error was detected.
- A pointer to a chain of snmp_dpi_set_packets (varBinds) to be returned as response to the GET request. If an error was detected, an snmp_dpi_set_packet_NULL_p pointer may be passed.

The following code example returns a response. You assume that there are no errors in the request, but proper code should do the checking for that. For instance, you return a noSuchInstance if the instance is not exactly what you expect and a noSuchObject if the object instance_ID is greater than 3. However, there might be no instance_ID at all and you should check for that, too.

```
static int do_get(snmpp_dpi_hdr *hdr_p, snmpp_dpi_get_packet *pack_p)
{
    unsigned char    *packet_p;
    int              rc;
    snmpp_dpi_set_packet *varBind_p;
    char             *i_p;

    varBind_p =
        snmpp_dpi_set_packet_NULL_p;    /* init the varBind chain */
                                        /* to a NULL pointer      */

    if (instance_level) {
        if (pack_p->instance_p) {
            printf("unexpected INSTANCE ptr \n");
            return(-1);
        }
    }
}
```

```

    }
    i_p = pack_p->group_p + strlen(DPI_SIMPLE_MIB);
} else {
    i_p = pack_p->instance_p;
}

if (i_p && (strcmp(i_p,"1.0") == 0)) {
    varBind_p = mkDPIset(
        varBind_p,          /* Make DPI set packet */
        varBind_p,          /* ptr to varBind chain */
        pack_p->group_p,     /* ptr to subtree */
        pack_p->instance_p,  /* ptr to rest of OID */
        SNMP_TYPE_Integer32, /* value type Integer 32 */
        sizeof(value1),     /* length of value */
        value1);            /* ptr to value */
} else if (i_p && (strcmp(i_p,"2.0") == 0)) {
    varBind_p = mkDPIset(
        varBind_p,          /* Make DPI set packet */
        varBind_p,          /* ptr to varBind chain */
        pack_p->group_p,     /* ptr to subtree */
        pack_p->instance_p,  /* ptr to rest of OID */
        SNMP_TYPE_DisplayString, /* value type */
        value2_len,         /* length of value */
        value2_p);          /* ptr to value */
} else if (i_p && (strcmp(i_p,"3.0") == 0)) {
    varBind_p = mkDPIset(
        varBind_p,          /* Make DPI set packet */
        varBind_p,          /* ptr to varBind chain */
        pack_p->group_p,     /* ptr to subtree */
        pack_p->instance_p,  /* ptr to rest of OID */
        SNMP_TYPE_Counter32, /* value type */
        sizeof(value3),     /* length of value */
        value3);            /* ptr to value */

#ifdef EXCLUDE_SNMP_SMIv2_SUPPORT
} else if (i_p && (strcmp(i_p,"4.0") == 0)) {
    varBind_p = mkDPIset(
        varBind_p,          /* Make DPI set packet */
        varBind_p,          /* ptr to varBind chain */
        pack_p->group_p,     /* ptr to subtree */
        pack_p->instance_p,  /* ptr to rest of OID */
        SNMP_TYPE_Counter64, /* value type */
        sizeof(value4),     /* length of value */
        value4);            /* ptr to value *Apr23*/
} else if (i_p && (strcmp(i_p,"4") > 0)) {
#else
} else if (i_p && (strcmp(i_p,"3") > 0)) {
#endif /* ndef EXCLUDE_SNMP_SMIv2_SUPPORT */
    varBind_p = mkDPIset(
        varBind_p,          /* Make DPI set packet */
        varBind_p,          /* ptr to varBind chain */
        pack_p->group_p,     /* ptr to subtree */
        pack_p->instance_p,  /* ptr to rest of OID */
        SNMP_TYPE_noSuchObject, /* value type */
        0L,                 /* length of value */
        (unsigned char *)0); /* ptr to value */
} else {
    varBind_p = mkDPIset(
        varBind_p,          /* Make DPI set packet */
        varBind_p,          /* ptr to varBind chain */
        pack_p->group_p,     /* ptr to subtree */
        pack_p->instance_p,  /* ptr to rest of OID */
        SNMP_TYPE_noSuchInstance, /* value type */
        0L,                 /* length of value */
        (unsigned char *)0); /* ptr to value */
} /* endif */

if (!varBind_p) return(-1); /* If it failed, return */

packet_p = mkDPIresponse(
    hdr_p,          /* Make DPIresponse packet */
    SNMP_ERROR_noError, /* ptr parsed request */
    0L,             /* all is OK, no error */
    0L,             /* index is zero, no error */

```

```

        varBind_p);          /* varBind response data */

    if (!packet_p) return(-1); /* If it failed, return */

    rc = DPISend_packet_to_agent( /* send RESPONSE packet */
        handle,                  /* on this connection */
        packet_p,                /* this is the packet */
        DPI_PACKET_LEN(packet_p)); /* and this is its length */

    return(rc);                  /* return retcode */
} /* end of do_get() */

```

Processing a GETNEXT request

When a DPI packet is parsed, the `snmp_dpi_hdr` structure shows in the `packet_type` that this is an `SNMP_DPI_GETNEXT` packet, and so the `packet_body` contains a pointer to a `GETNEXT-varBind`, which is represented in an `snmp_dpi_next_packet` structure:

```

struct dpi_next_packet {
    char      *object_p; /* ptr to OIDstring */
    char      *group_p;  /* ptr to sub-tree */
    char      *instance_p; /* ptr to rest of OID */
    struct dpi_next_packet *next_p; /* ptr to next in chain */
};
typedef struct dpi_next_packet      snmp_dpi_next_packet;
#define snmp_dpi_next_packet_NULL_p ((snmp_dpi_next_packet *)0)

```

Assuming you have registered subtree `dpiSimpleMIB` and a `GETNEXT` arrives for one variable (`dpiSimpleInteger.0`) that is object 1 instance 0 in the subtree, the fields in the `snmp_dpi_get_packet` structure would have pointers to:

```

object_p    -> "1.3.6.1.4.1.2.2.1.5.1.0"
group_p     -> "1.3.6.1.4.1.2.2.1.5."
instance_p  -> "1.0"
next_p      -> snmp_dpi_next_packet_NULL_p

```

If there are multiple `varBinds` in a `GETNEXT` request, each one is represented in an `snmp_dpi_next_packet` structure and all the `snmp_dpi_next_packet` structures are chained by the next pointer. As long as the next pointer is not the `snmp_dpi_next_packet_NULL_p` pointer, there are more `varBinds` in the list.

Now you can analyze the `varBind` structure for whatever checking you want to do. You must find out which `OID` is the one that lexicographically follows the one in the request. It is that `OID` with its value that you must return as a response. Therefore, you must now also set the proper `OID` in the response. When you are ready to make a response that contains the new `OID` and the value of that variable, you must prepare a `SET-varBind` which is represented in an `snmp_dpi_set_packet`:

```

struct dpi_set_packet {
    char      *object_p; /* ptr to OIDstring */
    char      *group_p;  /* ptr to sub-tree */
    char      *instance_p; /* ptr to rest of OID */
    unsigned char value_type; /* SNMP_TYPE_xxxx */
    unsigned short value_len; /* value length */
    char      *value_p; /* ptr to value itself */
    struct dpi_set_packet *next_p; /* ptr to next in chain */
};
typedef struct dpi_set_packet      snmp_dpi_set_packet;
#define snmp_dpi_set_packet_NULL_p ((snmp_dpi_set_packet *)0)

```

You can use the `mkDPIset()` function to prepare such a structure. This function expects the following parameters:

- A pointer to an existing `snmp_dpi_set_packet` structure if the new `varBind` must be added to an existing chain of `varBinds`. If this is the first or only `varBind` in the chain, pass the `snmp_dpi_set_packet_NULL_p` pointer to indicate this.
- A pointer to the desired subtree.
- A pointer to the rest of the OID, in other words the piece that follows the subtree.
- The value type of the value to be bound to the variable name. This must be one of the `SNMP_TYPE_xxxx` values as defined in the `snmp_dpi.h` include file.
- The length of the value. For integer type values, this must be a length of 4. Work with 32-bit signed or unsigned integers except for the Counter64 type. For Counter 64 type, point to an `snmp_dpi_u64` structure and pass the length of that structure.
- A pointer to the value.

Memory for the `varBind` is dynamically allocated and the data itself is copied. Upon return, you can dispose of your own pointers and allocated memory as you please. If the call is successful, a pointer is returned as follows:

- A new `snmp_dpi_set_packet` if it is the first or only `varBind`.
- The existing `snmp_dpi_set_packet` that you passed on the call. In this case, the new packet has been chained to the end of the `varBind` list.

If the `mkDPISet()` call fails, a NULL pointer is returned.

When you have prepared the SET-`varBind` data, create a DPI RESPONSE packet using the `mkDPISresponse()` function, which expects these parameters:

- A pointer to an `snmp_dpi_hdr`. Use the header of the parsed incoming packet. It is used to copy the `packet_id` from the request into the response, such that the agent can correlate the response to a request.
- A return code that is an SNMP error code. If successful, this should be `SNMP_ERROR_noError` (value 0). If failure, it must be one of the `SNMP_ERROR_xxxx` values as defined in the `snmp_dpi.h` include file.

A request for a nonexisting object or instance is not considered an error. Instead, pass the OID and value of the first OID that lexicographically follows the nonexisting object or instance.

Reaching the end of the subtree is not considered an error. For example, if there is no NEXT OID, this is not an error. In this situation, return the original OID as received in the request and a `value_type` of `SNMP_TYPE_endOfMibView`. This `value_type` has an implicit value of NULL, so you can pass a 0 length and a NULL pointer for the value.

- The index of the first `varBind` in error starts counting at 1. Pass 0 if no error occurred, or pass the proper index of the first `varBind` for which an error was detected.
- A pointer to a chain of `snmp_dpi_set_packets` (`varBinds`) to be returned as response to the GETNEXT request. If an error was detected, an `snmp_dpi_set_packet_NULL_p` pointer may be passed.

The following code example returns a response. It is assumed that there are no errors in the request, but proper code should do the checking for that. Proper checking is done for lexicographic next object, but no checking is done for `ULONG_MAX`, or making sure that the instance ID is indeed valid (digits and periods). If the code gets to the end of our `dpiSimpleMIB`, an `endOfMibView` is returned as defined by the SNMP Version 2 rules.

```

static int do_next(snmp_dpi_hdr *hdr_p, snmp_dpi_next_packet *pack_p)
{
    unsigned char    *packet_p;
    int              rc;
    unsigned long    subid;          /* subid is unsigned */
    unsigned long    instance;       /* same with instance */
    char             *cp;
    snmp_dpi_set_packet *varBind_p;

    varBind_p =
        snmp_dpi_set_packet_NULL_p; /* init the varBind chain */
                                   /* to a NULL pointer */

    /* If we have done instance level registration, then we should */
    /* never get a getNext. Anyway, if we do, then we skip this and */
    /* return an endOfMibView. */
    if (instance_level) {

        varBind_p = mkDPISet(
            varBind_p,          /* Make DPI set packet */
            varBind_p,          /* ptr to varBind chain */
            pack_p->group_p,     /* ptr to subtree */
            pack_p->instance_p,  /* ptr to rest of OID */
            SNMP_TYPE_endOfMibView, /* value type */
            0L,                 /* length of value */
            (unsigned char *)0); /* ptr to value */

    } else {

        if (pack_p->instance_p) { /* we have an instance ID */
            cp = pack_p->instance_p; /* pick up ptr */
            subid = strtoul(cp, cp, 10); /* convert subid (object) */
            if (*cp == '.') { /* followed by a dot ? */
                cp++; /* point after it if yes */
                instance=strtoul(cp, cp, 10); /* convert real instance */
                /* not that we need it, we */
                subid++; /* only have instance 0, */
                /* so NEXT is next object */
                instance = 0; /* and always instance 0 */
            } else { /* no real instance passed */
                instance = 0; /* so we can use 0 */
                if (subid == 0) subid++; /* if object 0, start at 1 */
            } /* endif */
        } else { /* no instance ID passed */
            subid = 1; /* so do first object */
            instance = 0; /* instance 0 (all we have) */
        } /* endif */

        /* we have set subid and instance such that we can basically */
        /* process the request as a GET now. Actually, we don't even */
        /* need instance, because all out object instances are zero. */

        if (instance != 0) printf("Strange instance: %lu\n", instance);

        switch (subid) {
        case 1:
            varBind_p = mkDPISet(
                varBind_p, /* Make DPI set packet */
                varBind_p, /* ptr to varBind chain */
                pack_p->group_p, /* ptr to subtree */
                DPI_SIMPLE_INTEGER, /* ptr to rest of OID */
                SNMP_TYPE_Integer32, /* value type Integer 32 */
                sizeof(value1), /* length of value */
                value1); /* ptr to value */
            break;
        case 2:
            varBind_p = mkDPISet(
                varBind_p, /* Make DPI set packet */
                varBind_p, /* ptr to varBind chain */
                pack_p->group_p, /* ptr to subtree */
                DPI_SIMPLE_STRING, /* ptr to rest of OID */
            );
        }
    }
}

```

```

        SNMP_TYPE_DisplayString, /* value type          */
        value2_len,             /* length of value */
        value2_p);              /* ptr to value     */
    break;
case 3:
    varBind_p = mkDPIset(        /* Make DPI set packet */
        varBind_p,              /* ptr to varBind chain */
        pack_p->group_p,        /* ptr to subtree      */
        DPI_SIMPLE_COUNTER32,   /* ptr to rest of OID  */
        SNMP_TYPE_Counter32,    /* value type          */
        sizeof(value3),         /* length of value     */
        value3);                /* ptr to value        */
    break;
#ifdef EXCLUDE_SNMP_SMIV2_SUPPORT
case 4:                          /* *Apr23*/
    varBind_p = mkDPIset(        /* Make DPI set packet */
        varBind_p,              /* ptr to varBind chain */
        pack_p->group_p,        /* ptr to subtree      */
        DPI_SIMPLE_COUNTER64,   /* ptr to rest of OID  */
        SNMP_TYPE_Counter64,    /* value type          */
        sizeof(value4),         /* length of value     */
        value4);                /* ptr to value        */
    break;
#endif /* ndef EXCLUDE_SNMP_SMIV2_SUPPORT */
default:
    varBind_p = mkDPIset(        /* Make DPI set packet */
        varBind_p,              /* ptr to varBind chain */
        pack_p->group_p,        /* ptr to subtree      */
        pack_p->instance_p,     /* ptr to rest of OID  */
        SNMP_TYPE_endOfMibView, /* value type          */
        0L,                    /* length of value     */
        (unsigned char *)0);    /* ptr to value        */
    break;
} /* endswitch */

} /* endif */

if (!varBind_p) return(-1);      /* If it failed, return */

packet_p = mkDPIresponse(        /* Make DPIresponse packet */
    hdr_p,                       /* ptr parsed request */
    SNMP_ERROR_noError,          /* all is OK, no error */
    0L,                          /* index is zero, no error */
    varBind_p);                 /* varBind response data */

if (!packet_p) return(-1);      /* If it failed, return */

rc = DPIsend_packet_to_agent(    /* send RESPONSE packet */
    handle,                      /* on this connection */
    packet_p,                   /* this is the packet */
    DPI_PACKET_LEN(packet_p)); /* and this is its length */

return(rc);                     /* return retcode */
} /* end of do_next() */

```

Processing a SET/COMMIT/UNDO request

These three requests can come in one of these sequences:

- SET, COMMIT
- SET, UNDO
- SET, COMMIT, UNDO

The normal sequence is SET and then COMMIT. When a SET request is received, preparations must be made to accept the new value. For example, check that

request is for an existing object and instance, check the value type and contents to be valid, and allocate memory, but do not yet make the change.

If there are no SET errors, the next received request will be a COMMIT request. It is then that the change must be made, but keep enough information such that you can UNDO the change later if you get a subsequent UNDO request. The latter may happen if the agent discovers any errors with other subagents while processing requests that belong to the same original SNMP SET packet. All the varBinds in the same SNMP request PDU must be processed as if atomic.

When the DPI packet is parsed, the `snmp_dpi_hdr` structure shows in the *packet_type* that this is an `SNMP_DPI_SET`, `SNMP_DPI_COMMIT`, or `SNMP_DPI_UNDO` packet. In that case, the `packet_body` contains a pointer to a SET-varBind, represented in an `snmp_dpi_set_packet` structure. COMMIT and UNDO have same varBind data as SET upon which they follow:

```
struct dpi_set_packet {
    char          *object_p;    /* ptr to OIDstring    */
    char          *group_p;     /* ptr to sub-tree     */
    char          *instance_p; /* ptr to rest of OID  */
    unsigned char  value_type;  /* SNMP_TYPE_xxxx      */
    unsigned short value_len;   /* value length        */
    char          *value_p;     /* ptr to value itself */
    struct dpi_set_packet *next_p; /* ptr to next in chain */
};
typedef struct dpi_set_packet      snmp_dpi_set_packet;
#define snmp_dpi_set_packet_NULL_p ((snmp_dpi_set_packet *)0)
```

Assuming we have a registered subtree `dpiSimpleMIB` and a SET request comes in for one variable (`dpiSimpleString.0`) that is object 1 instance 0 in the subtree, and also assuming that the agent knows about our compiled `dpiSimpleMIB` so that it knows this is a `DisplayString` (as opposed to just an arbitrary `OCTET_STRING`), the pointers in the `snmp_dpi_set_packet` structure would have pointers and values, such as:

```
object_p    -> "1.3.6.1.4.1.2.2.1.5.2.0"
group_p     -> "1.3.6.1.4.1.2.2.1.5."
instance_p  -> "2.0"
value_type  -> SNMP_TYPE_DisplayString
value_len   -> 8
value_p     -> pointer to the value to be set
next_p      -> snmp_dpi_get_packet_NULL_p
```

If there are multiple varBinds in a SET request, each one is represented in an `snmp_dpi_set_packet` structure and all the `snmp_dpi_set_packet` structures are chained by the next pointer. As long as the next pointer is not the `snmp_dpi_set_packet_NULL_p` pointer, there are more varBinds in the list.

Now you can analyze the varBind structure for whatever checking you want to do. When you are ready to make a response that contains the value of the variable, you can prepare a new SET-varBind. However, by definition, the response to a successful SET is exactly the same as the SET request. So there is no need to return any varBinds. A response with `SNMP_ERROR_noError` and an index of zero will do. If there is an error, a response with the `SNMP_ERROR_xxxx` error code and an index pointing to the varBind in error (counting starts at 1) will do.

The following code example returns a response. It is assumed that there are no errors in the request, but proper code should do the checking for that. The code also does not check if the varBind in the COMMIT or UNDO is the same as that in the SET request. A proper agent would make sure that that is the case, but a

proper subagent may want to verify that for itself. Only one check is done that this is `dpiSimpleString.0`, and if it is not, a `noCreation` is returned.

```
static int do_set(snmp_dpi_hdr *hdr_p, snmp_dpi_set_packet *pack_p)
{
    unsigned char    *packet_p;
    int               rc;
    int               index    = 0;
    int               error    = SNMP_ERROR_noError;
    snmp_dpi_set_packet *varBind_p;
    char              *i_p;

    varBind_p =
        snmp_dpi_set_packet_NULL_p;    /* init the varBind chain */
                                        /* to a NULL pointer      */

    if (instance_level) {
        i_p = pack_p->group_p + strlen(DPI_SIMPLE_MIB);
    } else {
        i_p = pack_p->instance_p;
    }

    if (!i_p || (strcmp(i_p,"2.0") != 0))
    {
        if (i_p &&
            (strncmp(i_p,"1.",2) == 0))
        {
            error = SNMP_ERROR_notWritable;
        } else if (i_p &&
            (strncmp(i_p,"2.",2) == 0))
        {
            error = SNMP_ERROR_noCreation;
        } else if (i_p &&
            (strncmp(i_p,"3.",2) == 0))
        {
            error = SNMP_ERROR_notWritable;
        } else {
            error = SNMP_ERROR_noCreation;
        } /* endif */

        packet_p = mkDPIresponse(    /* Make DPIresponse packet */
            hdr_p,                  /* ptr parsed request      */
            error,                  /* all is OK, no error     */
            1,                      /* index is 1, 1st varBind */
            varBind_p);            /* varBind response data  */

        if (!packet_p) return(-1);    /* If it failed, return    */

        rc = DPIsend_packet_to_agent( /* send RESPONSE packet   */
            handle,                  /* on this connection     */
            packet_p,                /* this is the packet      */
            DPI_PACKET_LEN(packet_p)); /* and this is its length  */

        return(rc);                /* return retcode         */
    }

    switch (hdr_p->packet_type) {
    case SNMP_DPI_SET:
        if ((pack_p->value_type != SNMP_TYPE_DisplayString) &&
            (pack_p->value_type != SNMP_TYPE_OCTET_STRING))
        { /* check octet string in case agent has no compiled MIB */
            error = SNMP_ERROR_wrongType;
            break; /* from switch */
        } /* endif */
        if (new_val_p) free(new_val_p); /* free these memory areas */
        if (old_val_p) free(old_val_p); /* if we allocated any    */
        new_val_p = (char *)0;
    }
}
```



```

old_val_p = (char *)0;
new_val_len = 0;
old_val_len = 0;

new_val_p = malloc(pack_p->value_len); /* allocate memory for */
/* new value to set */
if (new_val_p) { /* If success, then also */
    memcpy(new_val_p, /* copy new value to our */
           pack_p->value_p, /* own and newly allocated */
           pack_p->value_len); /* memory area. */
    new_val_len = pack_p->value_len;
} else { /* Else failed to malloc, */
    error = SNMP_ERROR_genErr; /* so that is a genErr */
    index = 1; /* at first varBind */
} /* endif */
break;
case SNMP_DPI_COMMIT:
    old_val_p = cur_val_p; /* save old value for undo */
    cur_val_p = new_val_p; /* make new value current */
    new_val_p = (char *)0; /* keep only 1 ptr around */
    old_val_len = cur_val_len; /* and keep lengths correct*/
    cur_val_len = new_val_len;
    new_val_len = 0;
    /* may need to convert from ASCII to native if OCTET_STRING */
    break;
case SNMP_DPI_UNDO:
    if (new_val_p) { /* free allocated memory */
        free(new_val_p);
        new_val_p = (char *)0;
        new_val_len = 0;
    } /* endif */
    if (old_val_p) {
        if (cur_val_p) free(cur_val_p);
        cur_val_p = old_val_p; /* reset to old value */
        cur_val_len = old_val_len;
        old_val_p = (char *)0;
        old_val_len = 0;
    } /* endif */
    break;
} /* endswitch */

packet_p = mkDPIresponse( /* Make DPIresponse packet */
    hdr_p, /* ptr parsed request */
    error, /* all is OK, no error */
    index, /* index is zero, no error */
    varBind_p); /* varBind response data */

if (!packet_p) return(-1); /* If it failed, return */

rc = DPIsend_packet_to_agent( /* send RESPONSE packet */
    handle, /* on this connection */
    packet_p, /* this is the packet */
    DPI_PACKET_LEN(packet_p)); /* and this is its length */

return(rc); /* return retcode */
} /* end of do_set() */

```

Processing an UNREGISTER request

An agent can send an UNREGISTER packet if some other subagent does a register for the same subtree at a higher priority. An agent can also send an UNREGISTER if, for example, an SNMP manager tells the agent to make the subagent connection or the registered subtree not valid.

Here is an example of how to handle such a packet.

```

static int do_unreg(snmp_dpi_hdr *hdr_p, snmp_dpi_ureg_packet *pack_p)
{
    printf("DPI UNREGISTER received from agent, reason=%d\n",
           pack_p->reason_code);
    printf("    subtree=%s\n", pack_p->group_p);
    if (pack_p->reason_code ==
        SNMP_UNREGISTER_higherPriorityRegistered)
    {
        return(0); /* keep waiting, we may regain subtree later */
    } /* endif */

    DPIdisconnect_from_agent(handle);
    return(-1); /* causes exit in main loop */
} /* end of do_unreg() */

```

Processing a CLOSE request

An agent can send a CLOSE packet if it encounters an error or for some other reason. It can also do so if an SNMP MANAGER tells it to make the subagent connection not valid.

Here is an example of how to handle such a packet.

```

static int do_close(snmp_dpi_hdr *hdr_p, snmp_dpi_close_packet *pack_p)
{
    printf("DPI CLOSE received from agent, reason=%d\n",
           pack_p->reason_code);

    DPIdisconnect_from_agent(handle);
    return(-1); /* causes exit in main loop */
} /* end of do_close() */

```

Generating a TRAP

Issue a trap any time after a DPI OPEN was successful. To do so, you must create a trap packet and send it to the agent. With the TRAP, you can pass different kinds of varBinds, if you want. In this example, three varBinds are passed; one with integer data, one with an octet string, and one with a counter. You can also pass an Enterprise ID, but with DPI 2.0, the agent will use your subagent ID as the enterprise ID if you do not pass one with the trap. In most cases, that will probably not cause problems.

You must first prepare a varBind list chain that contains the three variables that you want to pass along with the trap. To do so, prepare a chain of three `snmp_dpi_set_packet` structures, which looks like:

```

struct dpi_set_packet {
    char          *object_p; /* ptr to OIDstring */
    char          *group_p; /* ptr to sub-tree */
    char          *instance_p; /* ptr to rest of OID */
    unsigned char value_type; /* SNMP_TYPE_xxxx */
    unsigned short value_len; /* value length */
    char          *value_p; /* ptr to value itself */
    struct dpi_set_packet *next_p; /* ptr to next in chain */
};
typedef struct dpi_set_packet snmp_dpi_set_packet;
#define snmp_dpi_set_packet_NULL_p ((snmp_dpi_set_packet *)0)

```

You can use the `mkDPISet()` function to prepare such a structure. This function expects the following parameters:

- A pointer to an existing `snmp_dpi_set_packet` structure if the new `varBind` must be added to an existing chain of `varBinds`. If this is the first or the only `varBind` in the chain, pass the `snmp_dpi_set_packet_NULL_p` pointer to indicate this.
- A pointer to the desired subtree.
- A pointer to the rest of the OID, in other words, the piece that follows the subtree.
- The value type of the value to be bound to the variable name. This must be one of the `SNMP_TYPE_xxxx` values as defined in the `snmp_dpi.h` include file.
- The length of the value. For integer type values, this must be a length of 4. Always work with 32-bit signed or unsigned integers except for the `Counter64` type. For the `Counter64` type, point to an `snmp_dpi_u64` structure and pass the length of that structure.
- A pointer to the value.

Memory for the `varBind` is dynamically allocated and the data itself is copied. Upon return, you can dispose of your own pointers and allocated memory as you please. If the call is successful, a pointer is returned as follows:

- To a new `snmp_dpi_set_packet` if it is the first or only `varBind`.
- To the existing `snmp_dpi_set_packet` that you passed on the call. In this case, the new packet has been chained to the end of the `varBind` list.

If the `mkDPISet()` call fails, a `NULL` pointer is returned.

When you have prepared the SET-`varBind` data, create a DPI TRAP packet. To do so, use the `mkDPITrap()` function, which expects these parameters:

- The generic trap code. Use 6 for enterprise specific trap type.
- The specific trap type. This is a type that is defined by the MIB that you are implementing. In our example you just use a 1.
- A pointer to a chain of `varBinds` or the `NULL` pointer if no `varBinds` need to be passed with the trap.
- A pointer to the enterprise OID if you want to use a different enterprise ID than the OID you used to identify yourself as a subagent at DPI-OPEN time.

The following code creates an enterprise-specific trap with specific type 1 and passes 3 `varBinds`. The first `varBind` with object 1, instance 0, Integer32 value; the second `varBind` with object 2, instance 0, Octet String; the third with `Counter32`. You pass no enterprise ID.

```
static int do_trap(void)
{
    unsigned char    *packet_p;
    int              rc;
    snmp_dpi_set_packet *varBind_p, *set_p;

    varBind_p =
        snmp_dpi_set_packet_NULL_p;    /* init the varBind chain */
                                        /* to a NULL pointer */

    varBind_p = mkDPISet(
        varBind_p,          /* Make DPI set packet */
        DPI_SIMPLE_MIB,     /* ptr to varBind chain */
        DPI_SIMPLE_INTEGER, /* ptr to subtree */
        SNMP_TYPE_Integer32, /* ptr to rest of OID */
        sizeof(value1),     /* value type Integer 32 */
        value1);            /* length of value */
                                /* ptr to value */

    if (!varBind_p) return(-1);    /* If it failed, return */
}
```

```

        set_p      = mkDPIset(                /* Make DPI set packet */
                                varBind_p,    /* ptr to varBind chain */
                                DPI_SIMPLE_MIB, /* ptr to subtree      */
                                DPI_SIMPLE_STRING, /* ptr to rest of OID */
                                SNMP_TYPE_DisplayString, /* value type          */
                                value2_len,    /* length of value      */
                                value2_p);     /* ptr to value         */

        if (!set_p) {                        /* if we failed... then */
            fDPIset(varBind_p);              /* free earlier varBinds */
            return(-1);                      /* If it failed, return */
        }

        set_p      = mkDPIset(                /* Make DPI set packet */
                                varBind_p,    /* ptr to varBind chain */
                                DPI_SIMPLE_MIB, /* ptr to subtree      */
                                DPI_SIMPLE_COUNTER32, /* ptr to rest of OID */
                                SNMP_TYPE_Counter32, /* value type          */
                                sizeof(value3), /* length of value      */
                                value3);       /* ptr to value         */

        if (!set_p) {                        /* if we failed... then */
            fDPIset(varBind_p);              /* free earlier varBinds */
            return(-1);                      /* If it failed, return */
        }

#ifdef EXCLUDE_SNMP_SMIV2_SUPPORT
        set_p      = mkDPIset(                /* Make DPI set packet */
                                varBind_p,    /* ptr to varBind chain */
                                DPI_SIMPLE_MIB, /* ptr to subtree      */
                                DPI_SIMPLE_COUNTER64, /* ptr to rest of OID */
                                SNMP_TYPE_Counter64, /* value type          */
                                sizeof(value4), /* length of value      */
                                value4);       /* ptr to value         */

        if (!set_p) {                        /* if we failed... then */
            fDPIset(varBind_p);              /* free earlier varBinds */
            return(-1);                      /* If it failed, return */
        }
#endif /* ndef EXCLUDE_SNMP_SMIV2_SUPPORT */

        packet_p = mkDPITrap(                /* Make DPITrap packet */
                                6,            /* enterpriseSpecific   */
                                1,           /* specific type = 1    */
                                varBind_p,   /* varBind data, and use */
                                (char *)0);  /* default enterpriseID */

        if (!packet_p) return(-1);           /* If it failed, return */

        rc = DPIsend_packet_to_agent(        /* send TRAP packet    */
                                handle,      /* on this connection  */
                                packet_p,    /* this is the packet  */
                                DPI_PACKET_LEN(packet_p)); /* and this is its length */

        return(rc);                         /* return retcode      */
    } /* end of do_trap() */

```

Chapter 4. Running the sample SNMP DPI client program for version 2.0

This section explains how to run the sample SNMP DPI client program, `dpi_mvs_sample.c`, installed in `/usr/lpp/tcpip/samples`. It can be run using the SNMP agents that support the SNMP-DPI interface as described in RFC 1592. (See Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs.)

The sample implements a set of variables described by the `DPISimple-MIB`, a set of objects in the IBM Research tree (under the 1.3.6.1.4.1.2.2.1.5 object ID prefix). See “`DPISimple-MIB` descriptions” on page 130 for the object ID and type of each object.

Using the sample program

The `dpi_mvs_sample.c` program accepts the following arguments:

? Explains the usage

-d *n* Sets the debug at level *n*. For levels that cause DPI API debug messages to be created, the messages are written to the syslog daemon under the daemon facility.

The range is 0 (for no messages) to 2 (for the most verbose). The default value is 1 if you specify `-d` with no value.

0 No debug messages

1 Informational processing debug messages are written to stdout; DPI packet creation debug messages are written to the syslog daemon by the DPI API.

2 Informational processing debug messages are written to stdout; DPI packet creation debug messages and traces of packets sent and received are written to the syslog daemon by the DPI API.

-h *hostname*

Specifies the host name or IP address where an SNMP DPI-capable agent is running; the default is the local host.

-c *community_name*

Specifies the community name for the SNMP agent that is required to get the `dpiPort`; the default is `public`.

-ireg Specifies that the subagent should do instance-level registration of MIB objects.

-unix Specifies that the subagent should connect to the SNMP agent using a UNIX stream socket instead of a TCP socket. You must also define `INCLUDE_UNIX_DOMAIN_FOR_DPI` when compiling the subagent.

Compiling and linking the `dpi_mvs_sample.c` source code

The `dpi_mvs_sample.c` program is located in `/usr/lpp/tcpip/samples`.

You can specify the following compile-time flags:

- INCLUDE_UNIX_DOMAIN_FOR_DPI** Indicates that the sample subagent should be compiled to connect to the agent using a UNIX Stream socket instead of a TCP connection.
- MVS** Indicates that compilation is for MVS, and uses MVS-specific includes. Some MVS/VM-specific code is compiled.

DPISimple-MIB descriptions

The following shows the MIB descriptions for DPISimple-MIB implemented by the sample subagent.

```
# dpi_mvs_sample.c supports these variables as an SNMP DPI
sample sub-agent
# it also generates enterprise specific traps via DPI with these objects
```

Name	OID	Type	Value
-----	-----	-----	-----
dpiSimpleInteger	1.3.6.1.4.1.2.2.1.5.1.0	integer	5
dpiSimpleString	1.3.6.1.4.1.2.2.1.5.2.0	string	"Initial String"
dpiSimpleCounter32	1.3.6.1.4.1.2.2.1.5.3.0	counter32	1
dpiSimpleCounter64	1.3.6.1.4.1.2.2.1.5.4.0	counter64	
X'8000000000000001'			

Of the above, only dpiSimpleString can be changed with an SNMP SET request.

Chapter 5. SNMP manager API

z/OS Communications Server provides the SNMP manager application programming interface (API) for writing SNMP managers. Application developers can use this API to build SNMP management applications to retrieve SNMP management data.

SNMP protocol

SNMP is a set of Internet Engineering Task Force (IETF) standards for network management, including a protocol, a database structure specification, a set of data objects, and controls for using the protocol. The SNMP protocol is based on the TCP/IP protocol. SNMP has evolved over many years resulting in three major versions of the protocol: SNMPv1, SNMPv2c, and SNMPv3.

Elements of an SNMP model for a managed network are as follows:

Agent This entity implements the SNMP protocol stack (sometimes called the engine). The agent's role is to receive and respond to requests using the SNMP protocol. It routes requests from managers to the appropriate subagents. It communicates with managers using the SNMP protocol. For z/OS Communications Server, the agent is the osnmpd daemon.

SubAgents

These entities are sometimes called the monitoring agents. Subagents provide the data that represents the managed objects. They communicate with the agents. An example in z/OS Communications Server is the TCP/IP subagent.

Manager

The role of the manager is to generate requests to retrieve and modify management information. The manager uses the SNMP protocol stack to receive responses from these requests and can also receive notifications, which are unsolicited events. The manager uses the SNMP protocol to communicate with the agent.

Management Information Base (MIB)

The MIB defines a set of managed objects. Each managed object has a unique identifier, which is sometimes referred to as an object identifier (OID).

SNMP Messages

Messages are exchanged between the manager and agent entities over the UDP transport of TCP/IP. This facilitates the exchange of SNMP operations. The messages, called PDUs, have formats that are defined by the SNMPv1, SNMPv2c, and SNMPv3 protocols; the types are not interoperable. The messages that are sent and received depend on the role of the entity.

The SNMP manager API overview

The SNMP manager API simplifies management application development by hiding SNMP protocol stack complexities, which enables an application to focus on management.

The SNMP manager API provides the following:

- A set of C functions and a header file that your application can use to build an SNMP manager. These functions are 31-bit DLLs and a 64-bit DLL. See “Steps for compiling and linking SNMP manager API applications” on page 158 for more information about compiling and linking the SNMP manager API.
- The ability to build, send, and receive messages for SNMPv1, SNMPv2c, and SNMPv3 using the functions provided by the API.
- Helper functions to perform operations on the decoded PDU response.

The SNMP notification API overview

The SNMP notification API is an extension of the SNMP manager API. It leverages the SNMP manager API’s functionality to send notifications to SNMP agents, the SNMP notification receivers, or both. Available notifications include informs and both Version 1 and 2 traps.

The SNMP notification API provides the following:

- A set of C functions and header files that your application can use to build an SNMP notification originator. These functions are included within the SNMP manager API libraries.
- The ability to build and send notifications for SNMPv1, SNMPv2c, and SNMPv3 using the functions provided by the API.

SNMP manager API functions

Several functions, data structures, and constants are defined in the `snmpmgr.h` file in the `/usr/include` directory. To build, send, and receive an SNMP message, your SNMP manager needs to call certain functions. After each call to one of these functions, your SNMP manager should verify that a successful return code, `SNMP_MGR_RC_OK`, was passed back. If an error occurs during the function call, an invalid return code is passed back. In addition to the error code, your SNMP manager API log file contains helpful information regarding the specific cause of the error. See “Debugging the SNMP manager API” on page 159 for information on using the debugging features.

By calling the following functions from your SNMP manager, you build the data structures necessary to send an SNMP message to an SNMP agent or subagent. The packet your SNMP manager receives as a response from the target agent can then be decoded and parsed with the defined set of helper functions. These helper functions provide your SNMP manager the ability to extract those same data structures used to create the outgoing packet.

A sample manager implementation, `snmpSMgr.c`, is found in `usr/lpp/tcpip/samples`.

snmpAddVarBind – Adds a VarBind to the SnmpVarBinds structure

```
#include <snmpmgr.h>
int snmpAddVarBind(SnmpVarBinds **varbinds, const char *oid, const smiValue *value)
```

Description

This function adds a VarBind structure (OID and value) to the input SnmpVarBinds structure. If space is available in the SnmpVarBinds structure, that space is used; if not, storage space is reallocated and the SnmpVarBinds structure is updated.

Parameters

varbinds

This input and output parameter is a pointer to the address of the SnmpVarBinds structure into which the new VarBind structure needs to be created. If there is sufficient storage for this new VarBind structure in the array of VarBind structures pointed to from this SnmpVarBinds structure, then the output SnmpVarBinds parameter is not changed. Otherwise, storage is reallocated for this new VarBind structure and the new pointer to the beginning of the VarBind array is stored in this output parameter. This parameter is required.

oid

This input parameter is the string representation of the OID of the VarBind structure that is to be created. This parameter is required.

value

This input parameter is the address of the smiValue structure of the VarBind structure that is to be created. If this address is NULL, the created VarBind structure will contain an empty smiValue structure.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if either the *varbinds* parameter or the *oid* parameter is NULL
- SNMP_MGR_RC_OUT_OF_MEMORY if sufficient storage could not be allocated for the new VarBind, the OID container, or the value container
- SNMP_MGR_RC_INVALID_OID if an OID container cannot be created using this function's *oid* parameter
- SNMP_MGR_RC_INVALID_VALUE if a value container cannot be created using this function's value parameter

Rule: You must call the snmpCreateVarBinds() function before you call this function, because the *varbinds* parameter that is used on this function is returned from the snmpCreateVarBinds() function.

snmpBuildPDU – Builds an SNMP PDU

```
#include <snmpmgr.h>
int snmpBuildPDU(SnmpPDU **pdu, const SnmpSession *snmpSession, const int pduType,
                 const SnmpVarBinds *varbinds,
                 const smiUINT32 non_repeaters, const smiUINT32 max_repetitions,
                 smiUINT32 *req_id)
```

Description

This function creates and initializes an SNMP PDU. The PDU is built using the input parameters. The security-related information stored in the PDU is obtained from the session parameter. This function encodes the PDU using Basic Encoding Rules (BER), which are used by SNMP.

Parameters

pdu

This output parameter is a pointer to the variable where the address of the SnmpPDU structure that is created is stored. This parameter is required.

snmpSession

This input parameter is the address of the SNMP session for which the SNMP PDU needs to be built. This parameter is the output parameter of the snmpBuildSession() function and is required.

pduType

This input parameter specifies the type of the PDU. This parameter is required. The valid values are:

SNMP_PDU_NULL

If used, the RFC-defined default PDU type SNMP_PDU_GETNEXT is built and returned

SNMP_PDU_GET

SNMP_PDU_GETNEXT

SNMP_PDU_SET

SNMP_PDU_GETBULK

varbinds

This input parameter is a pointer to the array of VarBind structures built using the snmpCreateVarBinds() and snmpAddVarBind() functions. This parameter is required.

non_repeaters

This input parameter is the number of non-repeaters for an SNMP_PDU_GETBULK request. This parameter is required, but is ignored if the *pduType* value is not SNMP_PDU_GETBULK.

max_repetitions

This input parameter specifies the maximum number of repetitions for an SNMP_PDU_GETBULK request. This parameter is required, but is ignored if the *pduType* value is not SNMP_PDU_GETBULK

req_id

This input parameter is a pointer to the request ID to be stored in the SnmpPDU field. If the value of this parameter is greater than or equal to 0, then this value is used. Otherwise, a random request ID is generated and stored in the SnmpPDU field.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *pdu*, *snmpSession*, or *varbinds* parameter is NULL
- SNMP_MGR_RC_INVALID_PDU_TYPE if this function's *pduType* parameter is not valid.
- SNMP_MGR_RC_INVALID_PARAMETERS if this function's *non_repeaters* or *max_repetitions* parameter is not valid, or if this function's *req_id* parameter value is less than 0
- SNMP_MGR_RC_OUT_OF_MEMORY if storage could not be allocated for either the SnmpPDU structure or the encoded PDU string
- SNMP_MGR_RC_ENCODE_ERROR if an error was encountered while encoding the PDU

- `SNMP_MGR_RC_INTERNAL_ERROR` if an error occurred in an internal function. See your SNMP manager API's log file for more information about the error, including the internal return code value. Your IBM service representative uses this internal return code to help solve your error.

Rule: You must call the `snmpBuildSession()` and `snmpCreateVarBinds()` functions before you call this function because the outputs from these functions are required as input for the `snmpBuildPDU` function. After you are finished using the `SnmpPDU` created by this function, you must free the storage allocated by calling the `snmpFreePDU()` function.

snmpBuildSession – Creates a session

```
#include <snmpmgr.h>
int snmpBuildSession(const SnmpConfigEntry *configEntry, SnmpSession **snmpSession ,
                    const snmpSockAddr *localAddr)
```

Description

This function creates an SNMP session for a particular configuration entry, which represents a session with a target agent. Sessions are supported for communication using SNMPv1, SNMPv2c, and SNMPv3. This function opens a socket and returns the handle to the session, which is used as input to the `snmpBuildPDU()`, `snmpBuildV1TrapPDU()`, `snmpBuildV2TrapOrInformPDU()` and `snmpSendRequest()` functions.

Parameters

configEntry

This input parameter specifies the configuration information that is to be used to create this session. The format of the `SnmpConfigEntry` structure is specified in the `snmpmgr.h` file. This parameter is typically obtained from the `snmpInitialize()` call, but alternatively you can manually create an `SnmpConfigEntry` structure. This parameter is required.

snmpSession

This output parameter is a pointer to the variable where the address of the `SnmpSession` parameter is stored. This parameter is required.

localAddr

This input parameter specifies the local address and port that are to be used for sending the SNMP message. If this parameter is set to `NULL`, the TCP/IP stack selects the source address and port.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_NULL_PTR` if this function's *configEntry* or *snmpSession* parameter is `NULL`
- `SNMP_MGR_RC SOCK_ERROR` if there was an error encountered when opening the socket, setting the socket to be non-blocking, or attempting to bind the socket to a local address or port
- `SNMP_MGR_RC_OUT_OF_MEMORY` if storage for the session could not be allocated

Rule: Typically, you should call the `snmpInitialize()` function before you call this function, because the `SnmpConfigEntry` input parameter is returned from the `snmpInitialize()` function. However, alternatively you can manually create an `SnmpConfigEntry` structure. After you are finished with the SNMP session

returned by this function, you must close the socket and free the storage that was allocated by calling the `snmpTerminateSession()` function.

snmpCreateVarBinds – Creates a VarBind structure

```
#include <snmpmgr.h>
int snmpCreateVarBinds(SnmpVarBinds **varbinds, const int numVarbinds,
                      const char *oid, const smiValue *value)
```

Description

This function creates an `SnmpVarBinds` structure, which is used as input to the `snmpBuildPDU()` function. Optionally, the number of `VarBind` structures that are added to this `VarBinds` structure can be specified along with the first `OID`, `value` pair.

Parameters

varbinds

This output parameter is a pointer to the address of the `SnmpVarBinds` structure that is created and returned. This parameter is required.

numVarbinds

This input parameter specifies the number of variable bindings that will be part of the created `SnmpVarBinds` structure. This value must be greater than or equal to 0. If this value is 0, the `oid` and `value` parameters are ignored.

oid

This input parameter specifies the `OID` value for the first `VarBind` structure. If this parameter set to `NULL`, no `VarBind` structure is created within the `SnmpVarBinds` structure.

value

This input parameter is the address of the `smiValue` value of the `VarBind` structure that is to be created. This parameter is valid only if the `oid` parameter is specified. If an `oid` value is specified and the `smiValue` address is `NULL`, the created `VarBind` structure will contain an empty `smiValue`.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_NULL_PTR` if this function's *varbinds* parameter is `NULL`
- `SNMP_MGR_RC_INVALID_PARAMETERS` if this function's *numVarBinds* parameter is less than 0 or if the *oid* parameter is `NULL` but the *value* parameter is not `NULL`
- `SNMP_MGR_RC_OUT_OF_MEMORY` if storage cannot be allocated for the `SnmpVarBinds` structure

Tip: Use the `snmpAddVarBind()` function to add more `VarBind` structures to the `SnmpVarBinds` structure.

Rule: After you are finished using the `SnmpVarBinds` structure and the array of `VarBind` structures it contains, you must free the storage that was allocated for these structures by calling the `snmpFreeVarBinds()` function.

snmpFreeDecodedPDU - Free the decoded PDU

```
#include <snmpmgr.h>
int snmpFreeDecodedPDU(SnmpDecodedPDU *decodedPDU)
```

Description

This function frees the storage that was allocated for the decoded PDU by the `snmpSendRequest()` function.

Parameters

decodedPDU

This input parameter is a pointer to the decoded PDU that is returned by the `snmpSendRequest()` function. This parameter is required.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_NULL_PTR` if this function's *decodedPDU* parameter is NULL

snmpFreeOID - Free an OID string

```
#include <snmpmgr.h>
int snmpFreeOID(char *oidString)
```

Description

This function frees the storage that was allocated for the OID string by the `snmpGetOID()` function.

Parameters

oidString

This input parameter is a pointer to the OID string that is returned by the `snmpGetOID()` function. This parameter is required.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_NULL_PTR` if this function's *oidString* parameter is NULL

snmpFreePDU – Frees the resources of a PDU

```
#include <snmpmgr.h>
int snmpFreePDU(SnmpPDU *pdu)
```

Description

Frees the storage for the SNMP PDU, which was allocated by the `snmpBuildPDU()` function.

Parameters

pdu

This input parameter is the address of the PDU that is to be freed. This parameter is required.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_NULL_PTR` if this function's *pdu* parameter is NULL

snmpFreeVarBinds – Frees the VarBinds structure

```
#include <snmpmgr.h>
int snmpFreeVarBinds(SnmpVarBinds *varbinds)
```

Description

This function frees the storage for the `SnmpVarBinds` structure and its array of `VarBind` structures that was created by the `snmpCreateVarBinds()` and `snmpAddVarBind()` functions.

Parameters

varbinds

This input parameter specifies the address of the SnmpVarBinds structure that is to be freed. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *varbinds* parameter is NULL

snmpGetErrorInfo - Get the error information from the PDU response

```
#include <snmpmgr.h>
int snmpGetErrorInfo(const SnmpDecodedPDU *decodedPDU, int *error_index,
                    int *error_status)
```

Description

This function is used to retrieve the *error_index* and *error_status* values from a decoded response PDU.

Parameters

decodedPDU

This input parameter is a pointer to the decoded PDU that the error information is retrieved from. This parameter is required and is returned from a call to the snmpSendRequest() function.

error_index

This output parameter is a pointer to an integer where the error index from the PDU is stored. This parameter is required.

error_status

This output parameter is a pointer to an integer where the error status from the PDU is stored. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *pdu*, *error_index*, or *error_status* parameter is NULL

Rule: You must call the snmpSendRequest() function before you call this function because the decodedPDU value used as an input parameter with this function is returned from the snmpSendRequest() function.

snmpGetNumberOfVarBinds – Get the number of VarBinds attached to the PDU

```
#include <snmpmgr.h>
int snmpGetNumberOfVarBinds(const SnmpDecodedPDU *decodedPDU,
                           int *numVarbinds)
```

Description

This function retrieves the number of VarBind structures that are attached to the decoded PDU that is returned from the snmpSendRequest() function. Your SNMP manager can use this number to loop through the array of VarBind structures in the PDU, retrieving each of them by calling the snmpGetVarBind() function.

Parameters

decodedPDU

This input parameter is a pointer to the decoded PDU from which the number of VarBind structures is retrieved. This parameter is required and is returned from a call to the `snmpSendRequest()` function.

numVarbinds

This output parameter is a pointer to an integer where the number of VarBind structures is stored. This parameter is required.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_NULL_PTR` if this function's *decodedPDU* or *numVarbinds* parameter is NULL

Rule: You must call the `snmpSendRequest()` function before you call this function, because the *decodedPDU* input parameter used by this function is returned from the `snmpSendRequest()` function.

snmpGetOID – Get the OID from the VarBind structure

```
#include <snmpmgr.h>
int snmpGetOid(const VarBind *varbind, char **oidString)
```

Description

This function retrieves the OID value from a VarBind structure and returns the OID value as a string.

Parameters

varbind This input parameter is a pointer to the VarBind structure from which the OID value is retrieved. This parameter is required and is returned from a call to the `snmpGetVarBind()` function.

oidString

This output parameter is a pointer to the address of the OID value, in string format, that is to be returned. This parameter is required.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_NULL_PTR` if this function's *varbind* or *oidString* parameter is NULL
- `SNMP_MGR_RC_OUT_OF_MEMORY` if storage could not be allocated for the OID string

Rule: Your SNMP manager must call the `snmpGetVarBind()` function before calling this function, because the required *varbind* input parameter is returned from a call to the `snmpGetVarBind()` function. After you finish using the *oidString* value that is returned from this function, you must free the storage that was allocated for this string by calling the `snmpFreeOID()` function.

snmpGetRequestId – Get the PDU's requestId value

```
#include <snmpmgr.h>
int snmpGetRequestId(const SnmpDecodedPDU *decodedPDU, int *req_id)
```

Description

This function is used to retrieve the *requestId* field from a decoded response PDU.

Parameters

decodedPDU

This input parameter is a pointer to the decoded PDU that the *requestId* value will be retrieved from. This parameter is required and is returned from a call to the `snmpSendRequest()` function.

req_id This output parameter is the address where the retrieved *requestId* value is stored.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *decodedPDU* or *req_id* parameter is NULL

Rule: You must call the `snmpSendRequest()` function before you call this function because the required *decodedPDU* input parameter is returned from the `snmpSendRequest()` function.

snmpGetSockFd – Get the socket's file descriptor

```
#include <snmpmgr.h>
int snmpGetSockFd(const SnmpSession *snmpSession)
```

Description

This function retrieves the socket's file descriptor, which is stored in the session.

Parameters

snmpSession

This input parameter is a pointer to the session that the socket's file descriptor is retrieved from. This parameter is required and is returned from a call to the `snmpBuildSession()` function.

Result

Return the integer value of the socket's file descriptor.

- SNMP_MGR_RC_NULL_PTR if the parameter is NULL

Rule: You must call the `snmpBuildSession()` function before you call this function because the required *snmpSession* input parameter is returned by the `snmpBuildSession()` function.

snmpGetValue – Get the value from the VarBind structure

```
#include <snmpmgr.h>
int snmpGetValue(const VarBind *varbind, smiValue *value)
```

Description

This function retrieves the value from a VarBind structure and returns the value in an *smiValue* structure.

Parameters

varbind

This input parameter is a pointer to the VarBind structure that the value is to be retrieved from. This parameter is required and is returned from a call to the `snmpGetVarBind()` function.

value This output parameter is a pointer to the completed *smiValue* structure.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *varbind* or *value* parameter is NULL
- SNMP_MGR_RC_INVALID_VALUE if the *smi_type* value of the *varbind* parameter is not valid

Rule: Your SNMP manager must call the `snmpGetVarBind()` function before calling this function because the required *varbind* input parameter is returned by a call to the `snmpGetVarBind()` function.

snmpGetVarbind – Get a VarBind attached to the PDU

```
#include <snmpmgr.h>
int snmpGetVarBind(const SnmpDecodedPDU *decodedPDU, const int varbindNum,
                  VarBind *varbind)
```

Description

Given an index into a PDU's array of VarBind structures, this function completes the contents of the input VarBind structure.

Parameters

decodedPDU

This input parameter is a pointer to the decoded PDU that the VarBind structure is retrieved from. This parameter is required and is returned by a call to the `snmpSendRequest()` function.

varbindNum

This input parameter is an index into the array of VarBind structures in the decoded response PDU.

varbind

This output parameter is a pointer to the VarBind structure that is to be completed.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_INVALID_PARAMETERS if the *varbindNum* value is less than 0
- SNMP_MGR_RC_NULL_PTR if this function's *pdu* or *varbind* parameter is NULL

Rule: You must call the `snmpSendRequest()` function before you call this function because the required *decodedPDU* input parameter is returned by the `snmpSendRequest()` function.

snmpInitialize – Initialize the manager environment

```
#include <snmpmgr.h>
int snmpInitialize(const int functionsRequested,
                  const char *configFileName,
                  SnmpConfigEntry **configList)
```

Description

This function initializes the SNMP manager API. Optionally, your SNMP manager can pass a configuration file to this call. The format and syntax of this

configuration file is described in “SNMP manager API configuration file” on page 156. Your SNMP manager passes a single `SnmplibConfigEntry` structure as input to the `snmpBuildSession()` function.

Parameters

functionsRequested

This input parameter specifies the functions that might have to be performed by this SNMP engine. This parameter helps the SNMP engine to initialize the required components. Currently the only supported value is 0.

configFileName

This input parameter is the name of the configuration file that contains the configuration entries. A configuration entry is used on the `snmpBuildSession()` call. The file name can be a z/OS UNIX file name or an MVS data set name. If this parameter is set to NULL, no configuration file processing is performed. You must manually create the `SnmplibConfigEntry` structures that are required by the other SNMP manager API functions.

configList

This output parameter is a pointer that is used to return the configuration list that has been read from the configuration file during the initialization process. This list is a linked list of `SnmplibConfigEntry` structures, as defined in the `snmpmgr.h` file. Each `SnmplibConfigEntry` value represents a target agent defined in the configuration file. This parameter is used only if the `configFileName` parameter is provided. Otherwise, the value NULL should be specified.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_CONFIG_ERROR` if there was an error in the configuration file
- `SNMP_MGR_RC_FILE_ERROR` if the configuration file could not be opened.
- `SNMP_MGR_RC_NULL_PTR` if this function’s *configList* parameter is NULL and the *configFileName* parameter is not NULL
- `SNMP_MGR_RC_INTERNAL_ERROR` if an error occurred in an internal function. See your SNMP manager API’s log file for more information about the error, including the internal return code value. Your IBM service representative uses this internal return code to help solve your error

Rule: Because an `SnmplibConfigEntry` structure is required to create an SNMP session, your SNMP manager must call this function before calling the `snmpBuildSession()` function. After you are finished using the list of `SnmplibConfigEntry` structures, you must free their storage by calling the `snmpTerminate()` function.

snmpSendRequest – Send the snmpPDU request to an agent

```
#include <snmpmgr.h>
int snmpSendRequest(const SnmpSession *snmpSession, const SnmpPDU *pdu,
                   const int waitInterval, SnmpDecodedPDU **decodedResponse,
                   const int receiveOnly)
```

Description

This function sends the request PDU to the agent and waits for a response PDU. This function is a blocking function and after the function sends the request, it waits for the length of time specified by the *waitInterval* value, until it receives a response from the agent.

Parameters

snmpSession

This input parameter is a pointer to the SNMP session that was created by the `snmpBuildSession()` function. This parameter is required.

pdu

This input parameter is a pointer to the BER-encoded PDU value that is to be sent to the agent. This parameter is required.

waitInterval

This input parameter specifies the number of seconds that this call waits to receive a return. If the response does not return within this specified time, this function returns an error code that specifies the timeout period. If this parameter's value is 0, this call waits until a response is received. This parameter's value must be non-negative.

decodedResponse

This output parameter is a pointer to the address of the decoded response PDU that is returned from the agent. The value might be NULL if the PDU is not received before the *waitInterval* period expires. This decoded PDU is used as input to several of the helper functions. This parameter is required.

receiveOnly

This input parameter specifies whether or not to retry receiving the response PDU if the first attempt timed out. The assumption is that the request has been successfully sent but that there is no response from the SNMP agent. This parameter can be set to 0 if your SNMP manager is sending a PDU, or it can be set to 1 if you want to retry receiving a response. If you specify a value of 1, a PDU is not sent to the target agent.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_NULL_PTR` if this function's *snmpSession*, *pdu*, or *decodedResponse* parameter value is NULL
- `SNMP_MGR_RC_TIMEOUT` if there is a timeout while waiting for a response from the target agent
- `SNMP_MGR_RC_IO_ERROR` if there is an error while waiting for a response from the target agent that is not the result of a timeout
- `SNMP_MGR_RC_INVALID_PARAMETERS` if the *waitInterval* parameter value is less than 0 or if the *receiveOnly* parameter value is not 0 or 1
- `SNMP_MGR_RC_ENCODE_ERROR` if an error was encountered while encoding a PDU. If your SNMP manager sends an SNMPv3 request to a target agent without the agent's *engineID* value, this function handles the receipt of the agent's report PDU. In doing so, a new request PDU is built and is sent to the target agent, using the agent's *engineID* value.
- `SNMP_MGR_RC_INTERNAL_ERROR` if an error occurred in an internal function. See your SNMP manager API log file for more information about the error, including the internal return code value. Your IBM service representative uses this internal return code to help solve your error.
- `SNMP_MGR_RC_USM_UNKNOWN_USERNAME` if the target agent responded with a report PDU, indicating that a user name that was not valid was sent in your request
- `SNMP_MGR_RC_USM_UNSUPPORTED_SECLEVEL` if the target agent responded with a report PDU, indicating that a security level that was not valid was sent in your request

- `SNMP_MGR_RC_USM_WRONG_DIGEST` if the target agent responded with a report PDU, indicating that the message digest (created with the authentication key you defined) is not valid
- `SNMP_MGR_RC_USM_NOT_IN_WINDOW` if the target agent responded with a report PDU, indicating that your request did not fall within the target agent's accepted time range
- `SNMP_MGR_RC_USM_DECRYPTION_ERROR` if the target agent responded with a report PDU, indicating that the target agent could not successfully decrypt your encrypted request

Rule: Since this function requires an SNMP session and an SNMP PDU as input parameters, your SNMP manager must call the `snmpBuildSession()` and `snmpBuildPDU()` functions before calling this function. Your SNMP manager is responsible for allocating storage for the response PDU. After you are finished using this PDU, ensure that you free its storage.

snmpSetLogFunction – Set the logging level

```
#include <snmpmgr.h>
int snmpSetLogFunction(SnmpLogFunc funcName)
```

Description

Use this function to define an external function to be used for logging SNMP manager API messages. You should define such a function if you want the SNMP manager API log messages to be logged to the same location as other applications on your system. Your logging function must have only two parameters: an integer to define the level of the log message, and a string to define the log message itself. After you have called the `snmpSetLogFunction()` function from your SNMP manager, all of the SNMP manager API log messages are sent to your defined logging function. An example definition for your logging function is as follows:

```
void myLogger(int logLevel, char *logMsg);
```

Based on that example, you would then need to set the parameter for this function to point to your logging function as follows:

```
SnmpLogFunc funcName = myLogger;
rc = snmpSetLogFunc(funcName);
```

If you choose not to define your own logging function, you can log messages to a file defined by the `SNMP_MGR_LOG_FILE` environment variable, or have messages logged to syslog (the default). For more information about your logging options, see “Debugging the SNMP manager API” on page 159.

Parameters

funcName

This input parameter specifies the SNMP manager's preferred logging function.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_NULL_PTR` if the *funcName* parameter is NULL

snmpSetLogLevel – Set the logging level

```
#include <snmpmgr.h>
int snmpSetLogLevel(const int logLevel)
```

Description

This function sets the log level for messages. See “Debugging the SNMP manager API” on page 159 for more information about logging.

Parameters

logLevel

This input parameter specifies the preferred message logging level. Valid values for the *logLevel* parameter, as defined in `snmpmgr.h`, are defined as follows:

- `SNMP_LOG_NONE` (0)
- `SNMP_LOG_ERROR` (1)
- `SNMP_LOG_TRACE` (2)
- `SNMP_LOG_DUMP` (4)
- `SNMP_LOG_ALL` (7)

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_INVALID_PARAMETERS` if the *logLevel* parameter is not valid

Guideline: Because the `SNMP_MGR_LOG_LEVEL` environment variable is not read until the `snmpInitialize()` function is called, you should call this function prior to calling the `snmpInitialize()` function, to make sure that all possible messages are successfully logged.

Rule: The value of the `SNMP_MGR_LOG_LEVEL` environment variable, when set, cannot be changed by calling this function with a new log level. You must unset the environment variable for this function to operate correctly.

snmpSetRequestId – Set the PDU’s requestId value

```
#include <snmpmgr.h>
int snmpSetRequestId(const SnmpSession *snmpSession,
                    SnmpPDU *pdu, const int req_id)
```

Description

Use this function to set the *requestId* field in an SNMP PDU. This function rebuilds the encoded PDU that is returned by the `snmpBuildPDU()` function.

Parameters

snmpSession

This input parameter is a pointer to the SNMP session information required to reconstruct a PDU with the new request ID. This input is required and is returned by the `snmpBuildSession()` function.

pdu

This input parameter is a pointer to the encoded PDU where the request ID is to be set. This parameter is required and is returned by the `snmpBuildPDU()` function.

req_id

This input parameter specifies the integer *requestId* value to set in the PDU. The value of this parameter must be greater than or equal to 0.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_NULL_PTR` if either the *snmpSession* or *pdu* pointer is `NUL.L`
- `SNMP_MGR_RC_INVALID_PARAMETERS` if the *req_id* parameter is less than 0

- SNMP_MGR_RC_ENCODE_ERROR if an error was encountered while encoding the new PDU
- SNMP_MGR_RC_INTERNAL_ERROR if an error occurred in an internal function. See your SNMP manager API log file for more information about the error, including the internal return code value. Your IBM service representative uses this internal return code to help solve your error.

Rules:

- Because this function requires an SNMP session and an *Snmppdu* value as input, your SNMP manager must call the `snmpBuildSession()` and `snmpBuildPDU()` functions before calling this function.
- SNMPv1 traps do not use a request ID. Therefore, your SNMP manager must not call this function for a PDU type `SNMP_PDU_TRAPV1`

snmpTerminate – Release the resources

```
#include <snmpmgr.h>
int snmpTerminate(Snmppdu *headEntry)
```

Description

This function releases the resources that were allocated by the `snmpInitialize()` function.

Parameters

headEntry

This input parameter points to the first entry in the linked list of `Snmppdu` objects returned by the `snmpInitialize()` function.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *headEntry* parameter is NULL

snmpTerminateSession – Terminate a session

```
#include <snmpmgr.h>
int snmpTerminateSession(Snmppdu *snmpSession)
```

Description

This function terminates an SNMP session and releases all of the resources held by that session.

Parameters

snmpSession

This input parameter is the address of the session that was created by the `snmpBuildSession()` call.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *snmpSession* parameter is NULL
- SNMP_MGR_RC_SOCKET_ERROR if the socket that was created by the `snmpBuildSession()` function cannot be closed

Tip: After a session is terminated, it cannot be used again.

snmpValueCreateBits – Create an smiValue of type Bits

```
#include <snmpmgr.h>
int snmpValueCreateBits(smiValue *value, char *inBits, smiUINT32 inLen)
```

Description

This function completes an smiValue structure of type SNMP_SYNTAX_BITS, based on the input parameter values. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

Parameters

- value* This output parameter is the address of the smiValue structure to be completed. This parameter is required.
- inBits* This input parameter is a pointer to the value to be stored in the smiValue structure. This parameter is required.
- inLen* This input parameter is the length, in bytes, of the *inBits* parameter value. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* or *inBits* parameter is NULL
- SNMP_MGR_RC_OUT_OF_MEMORY if storage could not be allocated for the string representation of the value in the smiValue structure

snmpValueCreateCounter32 – Create an smiValue of type Counter32

```
#include <snmpmgr.h>
int snmpValueCreateCounter32(smiValue *value, smiUINT32 inInt)
```

Description

This function completes an smiValue structure of type SNMP_SYNTAX_CNTR32, based on the input integer. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

Parameters

- value* This output parameter is the address of the smiValue structure to be completed. This parameter is required.
- inInt* This input parameter is the unsigned integer to be stored in the smiValue structure. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter is NULL

snmpValueCreateCounter64 – Create an smiValue of type Counter64

```
#include <snmpmgr.h>
int snmpValueCreateCounter64(smiValue *value, smiUINT32 hiPart, smiUINT32 loPart)
```

Description

This function completes an smiValue structure of type SNMP_SYNTAX_CNTR64, based on the input integers. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

Parameters

- value* This output parameter is the address of the smiValue structure to be completed. This parameter is required.
- hiPart* This input parameter is the high-order 32 bits to be stored in the smiValue structure. This parameter is required.
- loPart* This input parameter is the low-order 32 bits to be stored in the smiValue structure. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter is NULL

snmpValueCreateGuage32 – Create an smiValue of type Guage32

```
#include <snmpmgr.h>
int snmpValueCreateGuage32(smiValue *value, smiUINT32 inInt)
```

Description

This function completes an smiValue structure of type SNMP_SYNTAX_GUAGE32, based on the input integer. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

Parameters

- value* This output parameter is the address of the smiValue structure to be completed. This parameter is required.
- inInt* This input parameter is the unsigned integer to be stored in the smiValue structure. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter is NULL

snmpValueCreateInteger – Create an smiValue of type Integer

```
#include <snmpmgr.h>
int snmpValueCreateInteger(smiValue *value, smiINT32 inInt)
```

Description

This function completes an smiValue structure of type SNMP_SYNTAX_INT, based on the input integer. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

Parameters

- value* This output parameter is the address of the smiValue structure to be completed. This parameter is required.
- inInt* This input parameter is the integer to be stored in the smiValue structure. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter is NULL

- SNMP_MGR_RC_INVALID_PARAMETERS if this function's *inInt* parameter value is less than 0

snmpValueCreateInteger32 – Create an smiValue of type Integer32

```
#include <snmpmgr.h>
int snmpValueCreateInteger32(smiValue *value, smiINT32 inInt)
```

Description

This function completes an smiValue structure of type SNMP_SYNTAX_INT32, based on the input integer. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

Parameters

- value* This output parameter is the address of the smiValue structure to be completed. This parameter is required.
- inInt* This input parameter is the integer to be stored in the smiValue structure. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter is NULL
- SNMP_MGR_RC_INVALID_PARAMETERS if this function's *inInt* parameter is less than 0

snmpValueCreateIPAddr – Create an smiValue of type IPAddr

```
#include <snmpmgr.h>
int snmpValueCreateIPAddr(smiValue *value, char *inAddr, smiUINT32 inLen)
```

Description

This function completes an smiValue structure of type SNMP_SYNTAX_IPADDR, based on the input parameters. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

Parameters

- value* This output parameter is the address of the smiValue structure to be completed. This parameter is required.
- inAddr* This input parameter is a pointer to the value to be stored in the smiValue structure. This parameter is required.
- inLen* This input parameter is the length, in bytes, of the *inAddr* parameter value. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter or *inAddr* parameter is NULL
- SNMP_MGR_RC_OUT_OF_MEMORY if storage could not be allocated for the string representation of the value in the smiValue structure

snmpValueCreateNSApAddr – Create an smiValue of type NSApAddr

```
#include <snmpmgr.h>
int snmpValueCreateNSApAddr(smiValue *value, char *inAddr, smiUINT32 inLen)
```

Description

This function completes an smiValue structure of type SNMP_SYNTAX_NSAPADDR, based on the input parameters. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

Parameters

- value* This output parameter is the address of the smiValue structure to be completed. This parameter is required.
- inAddr* This input parameter is a pointer to the value to be stored in the smiValue structure, as a binary network address. This parameter is required.
- inLen* This input parameter is the length, in bytes, of the *inAddr* parameter value. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter or *inAddr* parameter is NULL
- SNMP_MGR_RC_OUT_OF_MEMORY if storage could not be allocated for the string representation of the value in the smiValue structure

snmpValueCreateNull – Create an smiValue of type Null

```
#include <snmpmgr.h>
int snmpValueCreateNull(smiValue *value)
```

Description

This function completes an smiValue structure of type SNMP_SYNTAX_NULL, based on the input parameters. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

Parameters

- value* This output parameter is the address of the smiValue structure to be completed. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter is NULL

snmpValueCreateOctet – Create an smiValue of type Octet

```
#include <snmpmgr.h>
int snmpValueCreateOctet(smiValue *value, char *inOctet, smiUINT32 inLen)
```

Description

This function completes an smiValue structure of type SNMP_SYNTAX_OCTET, based on the input parameters. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

Parameters

value This output parameter is the address of the *smiValue* structure to be completed. This parameter is required.

inOctet

This input parameter is a pointer to the value to be stored in the *smiValue* structure. This parameter is required.

inLen This input parameter is the length, in bytes, of the *inOctet* parameter value. This parameter is required.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_bMGR_RC_NULL_PTR` if this function's *value* parameter or *inOctet* parameter is `NULL`
- `SNMP_MGR_RC_OUT_OF_MEMORY` if storage could not be allocated for the string representation of the value in the *smiValue* structure

snmpValueCreateOID – Create an *smiValue* of type `OID`

```
#include <snmpmgr.h>
int snmpValueCreateOID(smiValue *value, char *inOID, smiUINT32 inLen)
```

Description

This function completes an *smiValue* structure of type `SNMP_SYNTAX_OID`, based on the input parameters. This *smiValue* structure can then be used as input to the `snmpCreateVarBinds()` or `snmpAddVarBind()` function.

Parameters

value This output parameter is the address of the *smiValue* structure to be completed. This parameter is required.

inOID

This input parameter is a pointer to the value to be stored in the *smiValue* structure. This parameter is required.

inLen

This input parameter is the length, in bytes, of the *inOID* parameter value. This parameter is required.

Result

- `SNMP_MGR_RC_OK` if successful
- `SNMP_MGR_RC_NULL_PTR` if this function's *value* parameter or *inOID* parameter is `NULL`
- `SNMP_MGR_RC_OUT_OF_MEMORY` if storage could not be allocated for the string representation of the value in the *smiValue* structure

snmpValueCreateOpaque – Create an *smiValue* of type `Opaque`

```
#include <snmpmgr.h>
int snmpValueCreateOpaque(smiValue *value, char *inOpaque, smiUINT32 inLen)
```

Description

This function completes an *smiValue* structure of type `SNMP_SYNTAX_OPAQUE`, based on the input parameters. This *smiValue* structure can then be used as input to the `snmpCreateVarBinds()` or `snmpAddVarBind()` function.

Parameters

value This output parameter is the address of the smiValue structure to be completed. This parameter is required.

inOpaque

This input parameter is a pointer to the value to be stored in the smiValue structure. This parameter is required.

inLen This input parameter is the length, in bytes, of the *inOpaque* parameter value. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter or *inOpaque* parameter is NULL
- SNMP_MGR_RC_OUT_OF_MEMORY if storage could not be allocated in the smiValue structure for the string representation of the value

snmpValueCreateTimerTicks – Create an smiValue of type TimerTicks

```
#include <snmpmgr.h>
int snmpValueCreateTimerTicks(smiValue *value, smiUINT32 inInt)
```

Description

This function completes an smiValue structure of type SNMP_SYNTAX_TIMETICKS, based on the input integer. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

Parameters

value This output parameter is the address of the smiValue structure to be completed. This parameter is required.

inInt This input parameter is the unsigned integer to be stored in the smiValue structure. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter is NULL

snmpValueCreateUnsignedInteger32 – Create an smiValue of type UnsignedInteger32

```
#include <snmpmgr.h>
int snmpValueCreateUnsignedInteger32(smiValue *value, smiUINT32 inInt)
```

Description

This function completes an smiValue structure of type SNMP_SYNTAX_UINT32, based on the input integer. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

Parameters

value This output parameter is the address of the smiValue structure to be completed. This parameter is required.

inInt This input parameter is the unsigned integer to be stored in the *smiValue* structure. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter is NULL

SNMP notification API functions

The SNMP manager API can also be used to send notifications. When sending a notification, you can build the PDU using either the `snmpBuildV1TrapPDU()` or `snmpBuildV2TrapOrInformPDU()` function, which are defined in the `snmpntfy.h` file. Use the following SNMP manager API functions to use the SNMP notification API. See “SNMP manager API functions” on page 132 for descriptions of these functions.

- `snmpAddVarBind`
- `snmpBuildSession`
- `snmpCreateVarBinds`
- `snmpFreePDU`
- `snmpFreeVarBinds`
- `snmpInitialize`
- `snmpSendRequest`
- `snmpSetLogFunction`
- `snmpSetLogLevel`
- `snmpSetRequestId`
- `snmpTerminate`
- `snmpTerminateSession`

After each call to one of these functions, your SNMP manager should verify that a successful return code, `SNMP_MGR_RC_OK`, was returned. If an error occurred during the function call, an invalid return code is returned. In addition to the error code, your SNMP manager API log file contains information about the specific cause of the error. See “Debugging the SNMP manager API” on page 159 for more information about how to use the debugging features.

By calling the following functions from your SNMP manager, you build the data structures necessary to send an SNMP notification to an SNMP agent or subagent.

snmpBuildV1TrapPDU – Builds an SNMP V1 trap PDU

```
#include <snmpntfy.h>
int snmpBuildV1TrapPDU(SnmpPDU **pdu, const SnmpSession *snmpSession,
                      const SnmpVarBinds *varbinds, char *ent_p,
                      char *local_ip, int generic, int specific,
                      unsigned int timestamp);
```

Description

This function creates and initializes an SNMP PDU of type `SNMP_PDU_TRAPV1`. The PDU is built using the input parameters. The security-related information stored in the PDU is obtained from the *session* parameter. This function encodes the PDU using Basic Encoding Rules (BER), which are used by SNMP.

Parameters

- pdu* This output parameter is a pointer to the variable into which to store the address of the SnmpPDU structure that is created. This parameter is required.
- snmpSession* This input parameter is the address of the SNMP session for which the SNMP PDU needs to be built. This parameter is the output parameter of the snmpBuildSession() function and is required.
- varbinds* This input parameter is a pointer to the array of VarBind structures that was built using the snmpCreateVarBinds() and snmpAddVarBind() functions. This parameter is required.
- ent_p* This input parameter is a pointer to the enterprise OID that generates the trap. This parameter is required.
- local_ip* This input parameter is the address of the system that generates the trap (a character string). This parameter is required.
- generic* This input parameter indicates the generic trap type. This parameter is required.
- specific* This input parameter indicates the specific trap type. This parameter is required.
- timestamp* This input parameter specifies the amount of time that has elapsed between the last network re-initialization and generation of the trap. This parameter is required.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *pdu*, *snmpSession*, *varbinds*, *ent_p*, or *local_ip* parameter is NULL
- SNMP_MGR_RC_INVALID_PARAMETERS if one of the following is true for this function:
 - The *generic* parameter value is not in the range 0 - 6
 - The *generic* parameter value is in the range 0 - 5 and the *specific* parameter value is not 0
 - The *specific* parameter value is less than 0
- SNMP_MGR_RC_OUT_OF_MEMORY if storage could not be allocated for either the SnmpPDU structure or the encoded PDU string
- SNMP_MGR_RC_ENCODE_ERROR if an error was encountered while encoding the PDU
- SNMP_MGR_RC_INTERNAL_ERROR if an error occurred in an internal function. See your SNMP manager API log file for more information about the error, including the internal return code value. Your IBM service representative uses this internal return code to help solve your error

Rule: The snmpBuildSession() and snmpCreateVarBinds() functions must have been successfully called before calling this function, because output from those functions is required as input for this function. After you are finished using the SnmpPDU structure created by this function, you must free the storage that was allocated by calling the snmpFreePDU() function.

snmpBuildV2TrapOrInformPDU – Builds a SNMP V2 trap or inform PDU

```
#include <snmpntfy.h>
int snmpBuildPDU(SnmpPDU **pdu, const SnmpSession *snmpSession,
                 const int pduType, const SnmpVarBinds *varbinds,
                 smiUINT32 *req_id)
```

Description

This function creates and initializes either a v2 trap or an inform SNMP PDU. The PDU is built using the input parameters. The security-related information stored in the PDU is obtained from the *session* parameter. This function encodes the PDU using Basic Encoding Rules (BER), which are used by SNMP.

Parameters

pdu This output parameter is a pointer to the variable into which to store the address of the SnmpPDU structure that is created. This parameter is required.

snmpSession This input parameter is the address of the SNMP session for which the SNMP PDU needs to be built. This parameter is the output parameter of the snmpBuildSession() function and is required.

pduType This input parameter specifies the type of the PDU. This parameter is required. The valid values are:

- SNMP_PDU_TRAPV2
- SNMP_PDU_INFORM

varbinds This input parameter is a pointer to the array of VarBind structures built using the snmpCreateVarBinds() and snmpAddVarBind() functions. This parameter is required.

req_id This input parameter is a pointer to the request ID to be stored in the SnmpPDU structure. If the value of this parameter is greater than or equal to 0, this value is used. Otherwise, a random request ID is generated and stored in the SnmpPDU structure.

Result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *pdu*, *snmpSession*, or *varbinds* parameter is NULL
- SNMP_MGR_RC_INVALID_PDU_TYPE if this function's *pduType* parameter is not valid
- SNMP_MGR_RC_OUT_OF_MEMORY if storage could not be allocated for either the SnmpPDU structure or for the encoded PDU string
- SNMP_MGR_RC_ENCODE_ERROR if an error was encountered while encoding the PDU
- SNMP_MGR_RC_INTERNAL_ERROR if an error occurred in an internal function. See your SNMP manager API log file for more information about the error, including the internal return code value. Your IBM service representative uses this internal return code to help solve your error.

Rules:

- The `snmpBuildSession()` and `snmpCreateVarBinds()` functions must have been successfully called before calling this function, because the output from the `snmpBuildSession()` and `snmpCreateVarBinds()` functions is required as input for this function. After you are finished using the `SnmpPDU` structure created by this function, you must free the storage allocated by calling the `snmpFreePDU()` function.
- SNMPv2 traps and informs require the following two `VarBind` structures in the `VarBind` array:
 - `sysUpTime`
 - `snmpTrapOid`

Your SNMP manager must call the `snmpCreateVarBinds()` and `snmpAddVarBind()` functions to create these `VarBind` structures, in this order, before calling this function

SNMP manager API configuration file

```
#-----
# Format of entries (SNMPv1 and SNMPv2c):
#
#   targetAddr targetPort version communityName
#
#-----
# Community-based security (SNMPv1 and SNMPv2c)
#-----
9.1.1.1    161      snmpv1    -
9.1.1.2    -       snmpv2c    public

#-----
# Format of entries (SNMPv3):
#
#   targetAddr targetPort version userName password secLevel authProto authKey privProto privKey
#
#-----
# User-based security (SNMPv3)
#-----
9.8.0.1    1161     snmpv3     userid - AuthPriv HMAC-SHA 7236726349623ADCDEFFF32223ACC DES 32472653
666765764674EEAECDDB
```

Statement syntax

targetAddr

IP address (IPv4 dotted decimal format or IPv6 colon hexadecimal format) of the node of the target agent (maximum 19 characters). There is no default value.

targetPort

Port number of the target agent, in the range 1 - 65535. Use a dash (-) to indicate the default value (161).

version

Specifies the administrative model that is supported by the target agent. Valid values are:

snmpv1

Community-based SNMPV1 security

snmpv2c

Community-based SNMPV2 security

snmpv3

User-based SNMPV3 security

There is no default value.

communityName

Specifies the community name for community-based security (SNMPV1 or SNMPV2c). A dash (-) can be used to indicate the default value (public).

userName

Specifies the security name of the principal using this configuration file entry. For user-based security, this is the user name. The user must be defined at the target agent. This field is ignored unless SNMPv3 is specified for the version keyword. A valid value is a user name that is 1 - 32 characters in length. There is no default value.

password

Specifies the password that is to be used in generating the authentication and privacy keys for this user. If a password is specified, it is used to automatically generate any needed keys and the authKey and privKey fields are ignored. This field is ignored unless SNMPv3 is specified for the version keyword. If you do not want to specify a password, set the field to a single dash (-). (The minimum number of characters that you can specify is eight, and the maximum number is 64 characters.)

Rule: If you define a password in your configuration entry, the authKey and privKey fields must be set to a dash (-), which specifies no key.

Guideline: You should not use the password instead of keys in this configuration file, because using keys is more secure than storing passwords in this file.

Tip: To use a different password for authentication and privacy, you can overwrite the authPassword or privPassword field in the SnmpConfigEntry structure. By default, both of these password fields contain the value defined in the configuration file.

secLevel

Specifies the security level to be used when communicating with the target SNMP agent when this entry is used. This field is ignored unless SNMPv3 is specified for the version keyword. Valid values are:

- noAuthNoPriv or none to indicate that no authentication or privacy is requested
- AuthNoPriv or auth to indicate that authentication is requested but privacy is not requested
- AuthPriv or priv to indicate that both authentication and privacy are requested
- Dash (-) to indicate the default value (noAuthNoPriv)

authProto

SNMP authentication protocol to be used when communicating with the target SNMP agent when this entry is used. This field is ignored unless SNMPv3 is specified for the version keyword. Valid values are:

- HMAC-MD5
- HMAC-SHA
- A single dash (-) for no authentication

authKey

Specifies the SNMP authentication key to be used when communicating with the target SNMP agent when this entry is used. This key must be the non-localized key. This field is ignored if the password keyword is used.

This field is ignored unless SNMPv3 is specified for the version keyword and a non-default value is specified for the authProto parameter. Valid values are:

- 16 bytes (32 hexadecimal digits) when the authProto value is HMAC-MD5
- 20 bytes (40 hexadecimal digits) when the authProto value is HMAC-SHA
- A dash (-) indicates the default value, which is no key

privProto

Specifies the SNMP privacy protocol to be used when communicating with the target SNMP agent when this entry is used. This field is ignored unless SNMPv3 is specified for the version keyword. Valid values are:

- DES for CBC-DES
- A dash (-) to indicate the default value, which is no privacy

privKey

Specifies the SNMP privacy key to be used when communicating with the target SNMP agent when this entry is used. This key must be the non-localized key. This field is ignored if the password keyword is used. The privacy and authentication keys are assumed to have been generated using the same authentication protocol (for example, both with HMAC-MD5 or both with HMAC-SHA). This field is ignored unless the value snmpv3 is specified for the admin keyword and a non-default value is specified for the privProto parameter. Valid values are:

- 16 bytes (32 hexadecimal digits) when the authProto value is HMAC-MD5
- 20 bytes (40 hexadecimal digits) when the authProto value is HMAC-SHA
- A dash (-) to indicate the default value (no key)

General rules

- All parameters for an entry must be contained on one line in the configuration file.
- A dash (-) indicates the default value for a keyword.
- Comments begin with a number sign character (#) in column 1.
- The userName and password parameters are case sensitive.
- IP addresses are checked for validity. Enumerated values for example authProto) are checked for validity. All other character strings are checked only to ensure that they are not too long.

Steps for compiling and linking SNMP manager API applications

To use the SNMP manager API or SNMP notification API, perform the following steps:

1. Write your SNMP manager source application.

Make sure to include the

<snmpmgr.h> header file, which is available in the /usr/include directory. If your SNMP manager will be sending notifications, make sure to include the SNMP notification header,

<snmpmgr.h> , which is also available in the /usr/include directory.

2. Compile your application using the DLL compiler option. See the *z/OS XL C/C++ User's Guide* for more information about how to specify compiler options.
3. Include the SNMP manager API definition side deck (/usr/lib/EZBSNMPA.x, /usr/lib/EZBSNMPX.x, or /usr/lib/EZBSNMP6.x) when prelinking or binding the application.

Running your application

The SNMP API provides the following DLLs for running your application:

- 31-bit DLL EZBSNMPA
- 31-bit DLL EZBSNMPX for applications compiled with XPLINK
- 64-bit DLL EZBSNMP6 for applications compiled with XPLINK

These DLLs are included in the SYS1.SIEALNKE data set and in z/OS UNIX, in the /usr/lib directory. Ensure that the SYS1.SIEALNKE file is in your LNKLIST statement before running your application

Debugging the SNMP manager API

You can debug problems with the SNMP manager API in two ways:

For manager applications, call the snmpSetLogLevel() routine using the following debug levels:

Table 3. SNMP manager API debug levels

Debug level	Description
SNMP_LOG_NONE (0)	No logging
SNMP_LOG_ERROR (1)	Log only errors
SNMP_LOG_TRACE (2)	Trace function upon entry and exit
SNMP_LOG_DUMP(4)	Dump the session object
SNMP_LOG_ALL (7)	Log all levels

When calling the snmpSetLogLevel() routine, you can use multiple trace levels by specifying either the numerical value of each desired item, or the logical name. For example, for both SNMP_LOG_ERROR (1) and SNMP_LOG_TRACE (2), you could issue one of the following:

- snmpSetLogLevel(SNMP_LOG_ERROR + SNMP_LOG_TRACE)
- snmpSetLogLevel(3)

Set the SNMP_MGR_LOG_LEVEL *debuglevel* environment variable to turn on debugging. This environment variable is read when the snmpInitialize() function is called. You can set multiple trace levels by adding the levels that you want to trace.

The SNMP manager API attempts to read the SNMP_MGR_LOG_LEVEL environment variable in the snmpInitialize() function. If your SNMP manager calls the snmpSetLogLevel() function before calling the snmpInitialize() function, all API-generated trace messages in the snmpInitialize() function are logged. If not, logging begins inside the snmpInitialize() function after the environment variable is validated. After the environment variable has been read in the snmpInitialize() function, the value of the environment variable SNMP_MGR_LOG_LEVEL (if set) is used as the log level for your SNMP manager application. After this point, calls

to the `snmpSetLogLevel()` function do not change the log level. You must unset the environment variable for this function to operate correctly.

The SNMP manager API, by default, uses the SYSLOG daemon, and uses the current SYSLOG configuration for the output location. However, by declaring the `SNMP_MGR_LOG_FILE` environment variable, the SNMP manager can also send the output stream to an individual file (in addition to SYSLOG or another logging function). Use the `SNMP_MGR_LOG_FILE` environment variable to test your application. If the environment variable is declared, the log messages generated by the SNMP manager API are sent to both the file specified by the environment variable and either SYSLOG or your own logging function.

You can use your own logging function, rather than the SYSLOG default. Using your own logging function has the benefit of providing the log messages from the SNMP manager API, and your application logging. Your logging function overrides the default, SYSLOG logging function, which means that after you have enabled your logging function in the SNMP manager API, log messages are no longer sent to SYSLOG. If you have declared the `SNMP_MGR_LOG_FILE` environment variable, log messages are sent to both your logging function and to the file specified by the environment variable.

To use your own logging function, your SNMP manager needs to pass the name of the function as a parameter to the SNMP manager API's `snmpSetLogFunction()` routine. Every message produced by the SNMP manager API is then sent to your logging function, along with the integer that specifies the level of the log message (for example, `SNMP_LOG_TRACE`). Your function definition must be defined as follows so that the SNMP manager API calls it correctly:

```
void myLogger(int logLevel, char *logMsg);
```

Sample manager source code

You can find a sample SNMP manager implementation, `snmpSMgr.c`, in the `usr/lpp/tcpip/samples` directory.

Chapter 6. Resource Reservation Setup Protocol API (RAPI)

The z/OS UNIX RSVP Agent includes an application programming interface (API) for the Resource ReSerVation Protocol (RSVP), known as RAPI.

The RAPI interface is one realization of the generic API contained in the RSVP functional specification (see RFC 2205; see Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs). RSVP describes a resource reservation setup protocol designed for an integrated services internet. RSVP provides receiver-initiated setup of resource reservations for multicast or unicast data flows. See the RSVP applicability statement in reference RFC 2210 for more information.

The RAPI interface is a set of C language bindings whose calls are defined in this topic. Applications use RAPI to request enhanced Quality of Service (QoS). The RSVP agent then uses the RSVP protocol to propagate the QoS request through the routers along the paths for the data flow. Each router may accept or deny the request, depending upon the availability of resources. In the case of failure, the RSVP agent will return the decision to the requesting application by way of RAPI.

RSVP is a receiver-oriented signaling protocol that enables applications to request Quality of Service on an IP network. The types of Quality of Service requested by those applications are defined by Integrated Services. RSVP signaling applies to simplex unicast or multicast data flows. Although RSVP distinguishes senders from receivers, the same application may act in both roles.

RSVP assigns QoS to specific IP data flows that can be either multipoint-to-multipoint or point-to-point data flows, known as *sessions*. A session is defined by a particular transport protocol, IP destination address, and destination port. To receive data packets for a particular multicast session, an application must join the corresponding IP multicast group.

A data source, or sender, is defined by an IP source address and a source port. A given session may have multiple senders (S1, S2, ... Sn), and if the destination is a multicast address, multiple receivers (R1, R2, ... Rn).

Under RSVP, QoS requests are made by the data receivers. A QoS request contains a flowspec, together with a filter spec. The flowspec includes an Rspec, which defines the desired QoS and is used to control the packet scheduling mechanism in the router or host, and also a Tspec, which defines the traffic expected by the receiver. The filter spec controls packet classification to determine which sender data packets receive the corresponding QoS.

The detailed manner in which reservations from different receivers are shared in the internet is controlled by a reservation parameter known as the reservation style. The RSVP Functional Specification (see RFC 2205; see Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs) contains a definition and explanation of the different reservation styles. Also see the *z/OS Communications Server: IP Configuration Guide* and *z/OS Communications Server: IP Diagnosis Guide* for more information about the RSVP agent.

API outline

Using the RAPI interface, an application uses the `rapi_session()` call to define an *API session* for sending a single simplex data flow or receiving such a data flow. The `rapi_sender()` call may then be used to register as a data sender, and the `rapi_reserve()` call may be used to make a QoS reservation as a data receiver.

The `rapi_sender()` or `rapi_reserve()` calls may be repeated with different parameters to dynamically modify the state at any time or they can be issued in null forms that retract the corresponding registration. The application can call `rapi_release()` to close the session and delete all of its resource reservations.

A single API session, defined by a single `rapi_session()` call, can define only one sender at a time. More than one API session may be established for the same RSVP session. For example, if an application sends multiple UDP data flows distinguished by source port, it will call `rapi_session()` and `rapi_sender()` separately for each of these flows.

The `rapi_session()` call allows the application to specify an *upcall* (or *callback*) routine that will be invoked to signal RSVP state change and error events. There are five types of events:

- `RAPI_PATH_EVENT` signals the arrival or change of path state.
- `RAPI_RESV_EVENT` signals the arrival or change of reservation state.
- `RAPI_PATH_ERROR` signals the corresponding path error.
- `RAPI_RESV_CONFIRM` signals the arrival of a CONFIRM message.
- `RAPI_RESV_ERROR` signals the corresponding reservation error.

A synchronous error in a RAPI routine returns an appropriate error code. Asynchronous RSVP errors are delivered to the application by way of the RAPI upcall routine.

Compiling and linking RAPI applications

To use the RAPI interface, an application must perform the following steps:

1. Include the `<rapi.h>` header file, which is available in the `/usr/include` directory.
2. Compile the application with the DLL compiler option. See the *z/OS XL C/C++ User's Guide* for more information about how to specify compiler options.
3. Include the RAPI definition side deck (`rapi.x`), which is available in the `/usr/lib` directory, when prelinking or binding the application.
4. If the Binder is used instead of the C Prelinker, specify the Binder `DYNAM=DLL` option. See the *z/OS MVS Program Management: User's Guide and Reference* for information about specifying Binder options.

Running RAPI applications

At execution time, the RAPI application must have access to the RAPI DLL (`rapi.dll`), which is available in the `/usr/lib` directory. Ensure that the `LIBPATH` environment variable includes this directory when running the application. The RAPI application must run with superuser authority to use RAPI.

Event upcall

An *upcall* is invoked by the asynchronous event mechanism. It executes the function whose address was specified in the *event_rtn* parameter in the *rapi_session()* call.

The event upcall function template is defined as follows:

rapi_event_rtn_t - Event upcall

```
#include <rapi.h>

typedef int      rapi_event_rtn_t(
    rapi_sid_t      Sid,           /* Session ID          */
    rapi_eventinfo_t EventType,    /* Event type           */
    rapi_styleid_t   Style,        /* Reservation style    */
    int             ErrorCode,     /* Error event: code    */
    int             ErrorValue,    /* Error event: value   */
    rapi_addr_t      *ErrorNode,   /* Node detecting error */
    unsigned int     ErrorFlags,   /* Error flags          */
    int             FilterspecNo,  /* number of filterspecs */
    rapi_filter_t     *Filterspec_list,
    int             FlowspecNo,    /* number of flowspecs */
    rapi_flowspec_t   *Flowspec_list,
    int             AdspecNo,     /* number of adspecs   */
    rapi_adspec_t     *Adspec_list,
    void             *Event_arg    /* application argument */
);
```

Description

This is the template for the function address supplied on the *rapi_session* call. The event upcall function is invoked from the asynchronous event mechanism when an event occurs.

Parameters

Sid This parameter is the session ID for the session initiated by a successful *rapi_session()* call.

EventType

This parameter contains the upcall event type. See the description of this parameter under “Result” on page 164.

Style This parameter contains the style of the reservation; it is nonzero only for a RAPI_RESV_EVENT or RAPI_RESV_ERROR event.

ErrorCode, ErrorValue

These values encode the error cause, and they are set only for a RAPI_PATH_ERROR or RAPI_RESV_ERROR event. See “Error handling” on page 177 for interpretation of these values.

ErrorNode

This is the IP address of the node that detected the error, and it is set only for a RAPI_PATH_ERROR or RAPI_RESV_ERROR event.

ErrorFlags

These error flags are set only for a RAPI_PATH_ERROR or RAPI_RESV_ERROR event.

RAPI_ERRF_InPlace

The reservation failed, but another (presumably smaller) reservation is still in place on the same interface.

RAPI_ERRF_NotGuilty

The reservation failed, but the request from this client was merged with a larger reservation upstream, so this client reservation might not have caused the failure.

FilterSpec_list, FilterSpecNo

The *FilterSpec_list* parameter is a pointer to an area containing a sequential vector of RAPI *filter spec* or *sender template* objects. The number of objects in this vector is specified in *FilterSpecNo*. If *FilterSpecNo* is 0, the *FilterSpec_list* parameter will be NULL.

Flowspec_list, FlowspecNo

The *Flowspec_list* parameter is a pointer to an area containing a sequential vector of RAPI *flowspec* or *Tspec* objects. The number of objects in this vector is specified in *FlowspecNo*. If *FlowspecNo* is 0, the *Flowspec_list* parameter will be NULL.

Adspec_list, AdspecNo

The *Adspec_list* parameter is a pointer to an area containing a sequential vector of RAPI *adspec* objects. The number of objects in this vector is specified in *AdspecNo*. If *AdspecNo* is 0, the *Adspec_list* parameter will be NULL.

Event_arg

This is the value supplied in the *razi_session()* call.

Result

When the application upcall function returns, any areas pointed to by *Flowspec_list*, *FilterSpec_list*, or *Adspec_list* become not valid for further reference. The upcall function must copy any values it wants to save.

The specific parameters depend upon *EventType*, which may have one of the following values:

RAPI_PATH_EVENT

A path event indicates that RSVP sender (Path) state from a remote node has arrived or changed at the local node. A RAPI_PATH_EVENT event containing the complete current list of senders (or possibly no senders, after a path teardown) in the path state for the specified session will be triggered whenever the path state changes.

FilterSpec_list, *Flowspec_list*, and *Adspec_list* will be of equal length, and corresponding entries will contain *sender templates*, *sender Tspecs*, and *Adspecs*, respectively, for all senders known at this node. A missing object will generally be indicated by an empty RAPI object.

RAPI_PATH_EVENT events are enabled by the initial *razi_session()* call.

RAPI_RESV_EVENT

A reservation event indicates that reservation state has arrived or changed at the node, implying (but not assuring) that reservations have been established or deleted along the entire data path to one or more receivers. RAPI_RESV_EVENT upcalls containing the current reservation state for the API session will be triggered whenever the reservation state changes.

Flowspec_list will either contain one *flowspec* object or be empty (if the state has been torn down), and *FilterSpec_list* will contain zero or more corresponding *filter spec* objects. *Adspec_list* will be empty.

RAPI_RESV_EVENT upcalls are enabled by a `rapi_sender()` call; the *sender template* from the latter call will match the *filter spec* returned in the upcall triggered by a reservation event.

RAPI_PATH_ERROR

A path error event indicates that an asynchronous error has been found in the sender information specified in a `rapi_sender()` call.

The *ErrorCode* and *ErrorValue* parameters will specify the error. *FilterSpec_list* and *Flowspec_list* will each contain one object, the *sender template* and corresponding *sender Tspec* (if any) in error, while *Adspec_list* will be empty. If there is no *sender Tspec*, the object in *Flowspec_list* will be an empty RAPI object. The *Adspec_list* will be empty.

RAPI_PATH_ERROR events are enabled by a `rapi_sender()` call, and the *sender Tspec* in that call will match the *sender Tspec* returned in a subsequent upcall triggered by a RAPI_PATH_ERROR event.

RAPI_RESV_ERROR

A *reservation error* upcall indicates that an asynchronous reservation error has occurred.

The *ErrorCode* and *ErrorValue* parameters will specify the error. *Flowspec_list* will contain one *flowspec*, while *FilterSpec_list* may contain zero or more corresponding filter specs. *Adspec_list* will be empty.

RAPI_RESV_ERROR events are enabled by a `rapi_reserve()` call.

RAPI_RESV_CONFIRM

A RAPI_RESV_CONFIRM event indicates that a reservation has been made at least up to an intermediate merge point, and probably (but not necessarily) all the way to at least one sender.

The parameters of a RAPI_RESV_CONFIRM event are the same as those for a RAPI_RESV_EVENT event upcall.

The accompanying table summarizes the upcalls. *n* is a nonnegative integer.

Upcall	Enabled by	FilterSpecNo	FlowspecNo	AdspecNo
RAPI_PATH_EVENT	<code>rapi_session</code>	<i>n</i>	<i>n</i>	<i>n</i>
RAPI_PATH_ERROR	<code>rapi_sender</code>	1	1	0
RAPI_RESV_EVENT	<code>rapi_sender</code>	<i>n</i>	1 or 0	0
RAPI_RESV_ERROR	<code>rapi_reserve</code>	<i>n</i>	1	0
RAPI_RESV_CONFIRM	<code>rapi_reserve</code>	1	1	0

Client library services

The RSVP API provides the following client library calls:

- `rapi_release()`
- `rapi_reserve()`
- `rapi_sender()`
- `rapi_session()`
- `rapi_version()`

To use these calls, the application must include the file <rapi.h>. See “Header files” on page 179 for more information on header files.

rapi_release - Remove a session

```
#include <rapi.h>
```

```
int rapi_release (rapi_sid_t Sid)
```

Description

The rapi_release() call removes the reservation, if any, and the state corresponding to a given session handle. This call will be made implicitly if the application terminates without closing its RSVP sessions.

Parameters

Sid This parameter is the session ID for the session initiated by a successful rapi_session() call.

Result

If the session handle is not valid, the call returns a corresponding RAPI error code; otherwise, it returns 0.

rapi_reserve - Make, modify, or delete a reservation

```
#include <rapi.h>
```

```
int rapi_reserve(
    rapi_sid_t    Sid,           /* Session ID          */
    int           Flags,         /* Flags                */
    rapi_addr_t   *RHost,        /* Receive host addr    */
    rapi_styleid_t StyleId,      /* Style ID             */
    rapi_stylex_t *Style_Ext,    /* Style extension      */
    rapi_policy_t *Rcvr_Policy,  /* Receiver policy      */
    int           FilterSpecNo,  /* Number of filter specs */
    rapi_filter_t *FilterSpec_list, /* List of filter specs */
    int           FlowspecNo,    /* Number of flowspecs  */
    rapi_flowspec_t *Flowspec_list /* List of flowspecs    */
)
```

Description

The rapi_reserve() function is called to make, modify, or delete a resource reservation for a session. The call may be repeated with different parameters, allowing the application to modify or remove the reservation; the latest call will take precedence.

Parameters

Sid This parameter is the session ID for the session initiated by a successful rapi_session() call.

Flags No flags are currently defined for this call.

RHost This parameter is used to define the interface address on which data will be received for multicast flows. It is useful for a multihomed host. If it is NULL or the host address is INADDR_ANY, the default interface will be chosen.

StyleId This parameter specifies the reservation style ID (see *Flowspec_list*, *FlowspecNo*).

Style_Ext This parameter must be NULL.

Rcvr_Policy This parameter must be NULL.

FilterSpec_list, FilterSpecNo

The *FilterSpec_list* parameter is a pointer to an area containing a sequential vector of RAPI filter spec objects. The number of objects in this vector is specified in *FilterSpecNo*. If *FilterSpecNo* is 0, the *FilterSpec_list* parameter is ignored and can be NULL.

Flowspec_list, FlowspecNo

The *Flowspec_list* parameter is a pointer to an area containing a sequential vector of RAPI flow spec objects. The number of objects in this vector is specified in *FlowspecNo*. If *FlowspecNo* is 0, the *Flowspec_list* parameter is ignored and can be NULL.

If *FlowspecNo* is 0, the call will remove the current reservations for the specified session, and *FilterSpec_list* and *Flowspec_list* will be ignored. Otherwise, the parameters depend upon the style, as follows:

Wildcard Filter (WF)

Use *StyleId* = RAPI_RSTYLE_WILDCARD. The *Flowspec_list* parameter may be NULL (to delete the reservation) or else point to a single flowspec. The *FilterSpec_list* parameter should be empty.

Fixed Filter (FF)

Use *StyleId* = RAPI_RSTYLE_FIXED. *FilterSpecNo* must equal *FlowspecNo*. Entries in *Flowspec_list* and *FilterSpec_list* parameters will correspond in pairs.

Shared Explicit (SE)

Use *StyleId* = RAPI_RSTYLE_SE. The *Flowspec_list* parameter should point to a single flowspec. The *FilterSpec_list* parameter may point to a list of any length.

Result

Depending upon the parameters, each call may or may not result in new *admission control* calls, which could fail asynchronously.

If there is a synchronous error in this call, *razi_reserve()* returns a RAPI error code; otherwise, it returns 0.

Applications measure success in the form of errors returned when making QoS requests. No final acknowledgment will occur.

An *admission control* failure (for example, refusal of the QoS request) is reported asynchronously by an upcall of type RAPI_RESV_ERROR. A RSVP_Err_NO_PATH error code indicates that RSVP state from one or more of the senders specified in *FilterSpec_list* has not (yet) propagated all the way to the receiver; it may also indicate that one or more of the specified senders has closed its API session and that its RSVP state has been deleted from the routers.

razi_sender - Specify sender parameters

```
#include <razi.h>
```

```
int razi_sender(  
    razi_sid_t    Sid,          /* Session ID          */
```

```

int          Flags,          /* Flags          */
rapi_addr_t  *LHost,         /* Local Host     */
rapi_filter_t *SenderTemplate, /* Sender template */
rapi_tspec_t *SenderTspec,   /* Sender Tspec   */
rapi_adspec_t *SenderAdspec, /* Sender Adspec  */
rapi_policy_t *SenderPolicy, /* Sender policy data */
int          TTL             /* Multicast data TTL */
)

```

Description

An application must issue a `rapi_sender()` call if it intends to send a flow of data for which receivers may make reservations. This call defines, redefines, or deletes the parameters of that flow. A `rapi_sender()` call may be issued more than once for the same API session; the most recent one takes precedence.

Once a successful `rapi_sender()` call has been made, the application may receive upcalls of type `RAPI_RESV_EVENT` or `RAPI_PATH_ERROR`.

Parameters

Sid This parameter is the session ID for the session initiated by a successful `rapi_session()` call.

Flags No flags are currently defined for this call.

LHost This parameter may point to a `rapi_addr_t` structure specifying the IP source address and, if applicable, the source port from which data will be sent, or it may be NULL.

If the IP source address is `INADDR_ANY`, the API will use the default IP address of the local host. This is sufficient unless the host is multihomed. The port number may be zero if the protocol for the session does not have ports.

A NULL *LHost* parameter indicates that the application wishes to withdraw its registration as a sender. In this case, the following parameters will all be ignored.

SenderTemplate

This parameter may be a pointer to a RAPI filter specification structure specifying the format of data packets to be sent, or it may be NULL.

If this parameter is NULL, a sender template will be created internally from the *Dest* and *LHost* parameters. The *Dest* parameter was supplied in an earlier `rapi_session()` call. If a *SenderTemplate* parameter is present, the (non-NULL) *LHost* parameter is ignored.

SenderTspec

This parameter is a pointer to a *Tspec* that defines the traffic that this sender will create and must not be NULL.

SenderAdspec

This parameter must be NULL or unpredictable results may occur.

SenderPolicy

This parameter must be NULL.

TTL This parameter specifies the IP TTL (Time-to-Live) value with which multicast data will be sent. It allows RSVP to send its control messages with the same TTL scope as the data packets.

Result

If there is a synchronous error, `rapi_sender()` returns a RAPI error code; otherwise, it returns 0.

rapi_session - Create a session

```
#include <rapi.h>
```

```
rapi_sid_t rapi_session(  
    rapi_addr_t    *Dest,      /* Session: (Dst addr, port) */  
    int            Protid,     /* Protocol Id                */  
    int            Flags,      /* Flags                      */  
    rapi_event_rtn_t Event_rtn, /* Address of upcall routine */  
    void           *Event_arg, /* App argument to upcall    */  
    int            *Errnop     /* Place to return error code*/  
)
```

Description

The `rapi_session()` call creates an API session.

After a successful `rapi_session()` call has been made, the application may receive upcalls of type `RAPI_PATH_EVENT` for the API session.

Parameters

The parameters are as follows:

Dest This parameter points to a `rapi_addr_t` structure defining the destination IP address and a port number to which data will be sent. The *Dest* and *Protid* parameters define an RSVP session. If the *Protid* specifies UDP or TCP transport, the port corresponds to the appropriate transport port number.

Protid The IP protocol ID for the session. If it is omitted (that is, zero), 17 (UDP) is assumed.

Flags The valid values for *Flags* are as follows:

RAPI_USE_INTSERV

If set, *IntServ* formats are used in upcalls; otherwise, the *Simplified* format is used.

Event_rtn

This parameter is a function typedef for an upcall function that will be invoked to notify the application of RSVP errors and state change events. Pending events cause the invocation of the *upcall* function. The application must supply an upcall routine for event processing.

Event_arg

This parameter is an argument that will be passed to any invocation of the upcall routine.

Errnop The address of an integer into which a RAPI error code will be returned. If *Errnop* is NULL, no error code is returned.

Result

If the call succeeds, the `rapi_session()` call returns a nonzero session handle for use in subsequent calls related to this API session.

If the call fails synchronously, it returns zero (`RAPI_NULL_SID`) and stores a RAPI error code into an integer variable pointed to by the *Errnop* parameter.

Extended description

An application can have multiple API sessions registered for the same or different RSVP sessions at the same time. There can be at most one sender associated with each API session; however, an application can announce multiple senders for a given RSVP session by announcing each sender in a separate API session.

Two API sessions for the same RSVP session, if they are receiving data, are assumed to have joined the same multicast group and will receive the same data packets.

rapi_version - RAPI version

```
#include <rapi.h>
```

```
int rapi_version(void)
```

Description

This call obtains the version of the interface. It may be used by an application to adapt to different versions.

Result

This call returns a single integer that defines the version of the interface. The returned value is composed of a major number and a minor number, encoded as $100 * \text{major} + \text{minor}$

The API described in this topic has major version number 6.

RAPI formatting routines

For convenience of applications, RAPI includes standard routines for displaying the contents of RAPI objects.

These standard formatting routines are:

- rapi_fmt_adspec()
- rapi_fmt_filtspec()
- rapi_fmt_flowspec()
- rapi_fmt_tspect()

rapi_fmt_adspec - Format an adspec

```
#include <rapi.h>
```

```
void rapi_fmt_adspec(  
    rapi_adspec_t *adspecp, /* Addr of RAPI adspec */  
    char *buffer, /* Addr of buffer */  
    int length /* Length of buffer */  
)
```

Description

The rapi_fmt_adspec() call formats a given RAPI adspec into a buffer of given address and length. The output is truncated if the length is too small. If it is NULL, this function returns without performing any formatting.

Parameters

adspecp

This parameter is a pointer to the adspec to be formatted. If it is NULL, this function returns without performing any formatting.

buffer This is a pointer to the user-supplied buffer into which the formatted output will be placed. If the buffer is too small to contain the output, then the formatted output is truncated. If this parameter is NULL, this function returns without performing any formatting.

length This is the length of the buffer pointed to with the buffer parameter. If this parameter is 0, this function returns without performing any formatting.

Result

If possible, the input object is formatted into the user-supplied buffer. There is no return value.

The following example shows possible adspec output:

```
[GEN AS[brk=y hop=0 BW=0 lat=0 mtu=0] ]
```

The output reflects the following code:

```
GEN    Generic adspec
```

rapi_fmt_filtspec - Format a filtspec

```
#include <rapi.h>
```

```
void rapi_fmt_filtspec(  
    rapi_filtspec_t *filtp, /* Addr of RAPI Filtspec */  
    char *buffer, /* Addr of buffer */  
    int length /* Length of buffer */  
)
```

Description

The rapi_fmt_filtspec() call formats a given RAPI filter spec into a buffer of given address and length. The output is truncated if the length is too small. If it is NULL, this function returns without performing any formatting.

Parameters

filtp This parameter is a pointer to the Filtspec to be formatted. If it is NULL, this function returns without performing any formatting.

buffer This is a pointer to the user-supplied buffer into which the formatted output will be placed. If the buffer is too small to contain the output, then the formatted output is truncated. If this parameter is NULL, this function returns without performing any formatting.

length This is the length of the buffer pointed to with the buffer parameter. If this parameter is 0, this function returns without performing any formatting.

Result

If possible, the input object is formatted into the user-supplied buffer. There is no return value.

The following example shows possible filtspec output:

```
9.67.200.2/8000
```

showing the IP address and port.

rapi_fmt_flowspec - Format a flowspec

```
#include <rapi.h>
```

```
void rapi_fmt_flowspec(  
    rapi_flowspec_t *specp, /* Addr of RAPI flowspec */  
    char *buffer, /* Addr of buffer */  
    int length /* Length of buffer */  
)
```

Description

The `rapi_fmt_flowspec()` call formats a given RAPI *flowspec* into a buffer of given address and length. The output is truncated if the length is too small.

Parameters

- specp** This parameter is a pointer to the flowspec to be formatted. If it is NULL, this function returns without performing any formatting.
- buffer** This is a pointer to the user-supplied buffer into which the formatted output will be placed. If the buffer is too small to contain the output, then the formatted output is truncated. If this parameter is NULL, this function returns without performing any formatting.
- length** This is the length of the buffer pointed to with the buffer parameter. If this parameter is 0, this function returns without performing any formatting.

Result

If possible, the input object is formatted into the user-supplied buffer. There is no return value.

The following example shows the formatted output for a Controlled Load flowspec.

```
[CL TS[r=90000 b=6000 p=5.5e+06 m=1024 M=2048] ]
```

Note: Many of the RAPI object values are floating point numbers. The formatting functions display large floating point values in a user-friendly way, such as that shown for the Tspec p value.

The output reflects the following codes:

- CL** Controlled load
- TS** Tspec, listing the Tspec values

The following example shows the formatted output for a guaranteed flowspec.

```
[GUAR TS[r=90000 b=6000 p=5.5e+06 m=1024 M=2048] RS[R=90000 S=1] ]
```

Note: Many of the RAPI object values are floating point numbers. The formatting functions display large floating point values in a user-friendly way, such as that shown for the Tspec p value.

The output reflects the following codes:

- GUAR** Guaranteed
- TS** Tspec, listing the Tspec values
- RS** Rspec, listing the Rspec values

rapi_fmt_tspec - Format a tspec

```
#include <rapi.h>

void rapi_fmt_tspec(
    rapi_tspec_t *tspecp, /* Addr of RAPI Tspec */
    char *buffer, /* Addr of buffer */
    int length /* Length of buffer */
)
```

Description

The `rapi_fmt_tspec()` call formats a given RAPI *Tspec* into a buffer of given address and length. The output is truncated if the length is too small.

Parameters

- tspecp** This parameter is a pointer to the *Tspec* to be formatted. If it is NULL, this function returns without performing any formatting.
- buffer** This is a pointer to the user-supplied buffer into which the formatted output will be placed. If the buffer is too small to contain the output, then the formatted output is truncated. If this parameter is NULL, this function returns without performing any formatting.
- length** This is the length of the buffer pointed to with the buffer parameter. If this parameter is 0, this function returns without performing any formatting.

Result

If possible, the input object is formatted into the user-supplied buffer. There is no return value.

The following example shows possible *Tspec* output:

```
[GEN TS[r=55000 b=6000 p=5.5e+06 m=1024 M=2048] ]
```

Note: Many of the RAPI object values are floating point numbers. The formatting functions display large floating point values in a user-friendly way, such as that shown for the *Tspec* *p* value.

The output reflects the following codes:

GEN Generic *Tspec*

TS *Tspec*, listing the *Tspec* values

RAPI objects

Flowspecs, *filter specs*, *sender templates*, and *sender Tspecs* are encoded as variable-length RAPI objects.

Every RAPI object begins with a header of type `rapi_hdr_t`, which contains:

- The total length of the object in bytes
- The type

An empty object consists only of a header, with type 0 and length *sizeof* (`rapi_hdr_t`).

Integrated services data structures are defined in RFC 2210, which describes the use of the RSVP with the Controlled-Load and Guaranteed services. (See Appendix E, “Related protocol specifications,” on page 719 for information about

accessing RFCs.) RSVP defines several data objects which carry resource reservation information but are opaque to RSVP itself. The usage and data format of those objects is given in RFC 2210.

Flowspecs

There are two formats for RAPI *flowspecs*. For further details, see “The <razi.h> header” on page 180.

RAPI_FLOWSTYPE_Simplified

This is a *simplified* format. It consists of a simple list of parameters needed for either *Guaranteed* or *Controlled Load* service, using the service type QOS_GUARANTEED or QOS_CNTR_LOAD, respectively.

The RAPI client library routines map this format to or from an appropriate Integrated Services data structure.

RAPI_FLOWSTYPE_Intserv

This *flowspec* must be a fully formatted Integrated Services flowspec data structure.

Upcalls

In an upcall, a *flowspec* is by default delivered in *simplified* format. However, if the RAPI_USE_INTSERV flag was set in the razi_session() call, then the *IntServ* format is used in upcalls.

Sender tspecs

There are two formats for RAPI *Sender Tspecs*. For further details, see “The <razi.h> header” on page 180.

RAPI_TSPECTYPE_Simplified

This is a *simplified* format consisting of a simple list of parameters with the service type QOS_TSPEC. The RAPI client library routines map this format to or from an appropriate Integrated Services data structure.

RAPI_TSPECTYPE_Intserv

This *Tspec* must be a fully formatted Integrated Services *Tspec* data structure.

Upcalls

In an upcall, a *sender Tspec* is by default delivered in *simplified* format. However, if the RAPI_USE_INTSERV flag was set in the razi_session() call, then the *IntServ* format is used in upcalls.

Adspecs

There are two formats for RAPI *adspecs*. For further details, see “The <razi.h> header” on page 180.

RAPI_ADSTYPE_Simplified

This is a *simplified* format, consisting of a list of *adspec* parameters for all possible services. The RAPI client library routines map this format to an appropriate Integrated Services data structure.

RAPI_ADSTYPE_Intserv

This *adspec* must be a fully formatted Integrated Services *Adspec* data structure.

Upcalls

In an upcall, an *adspec* is by default delivered in *simplified* format. However, if the `RAPI_USE_INTSERV` flag was set in the `rapi_session()` call, then the *IntServ* format is used in upcalls.

Filter specs and sender templates

These objects have the following format:

RAPI_FILTERFORM_BASE This object consists of a socket address structure defining the IP address and port.

Asynchronous event handling

The RAPI interface provides an asynchronous upcall mechanism using the `select()` function. The upcall mechanism is a cooperative effort between RAPI and the using application. The following shows the steps that must be taken by a RAPI application to receive asynchronous upcalls:

1. The upcall function pointer must be specified on the `rapi_session()` call that initiates the RAPI session. If the upcall function requires an argument, that also must be specified on `rapi_session()`. The argument is defined as a pointer to void.
2. The application must provide a means to be notified of asynchronous events. The best way to do this is to create a thread using `pthread_create()`.
3. The thread created above must issue the `rapi_getfd()` call to learn the file descriptor of the socket used by RAPI for asynchronous communication.
4. The thread should then enter an endless loop to detect asynchronous events using the `select()` call with the file descriptor learned using `rapi_getfd()`. When an event is detected, the thread should call `rapi_dispatch()`, which then in turn calls the upcall function synchronously.

The following example illustrates these steps. This example is for illustration purposes only. It is not a complete program.

```
/* ***** */
/* Issue a rapi_session() call to initialize RAPI. */
/* ***** */
rapi_sid = rapi_session(&destination,
                        protocol,
                        0,
                        rapi_async, /* upcall function pointer */
                        0,          /* no upcall argument */
                        &rc);

...
/* ***** */
/* Create a pthread to handle RAPI upcalls. */
/* ***** */
pthread_create(&thread_d,
              NULL,
              &rapi_th,
              NULL);

...
/* ***** */
/* Function: rapi_th() */
/* ***** */
void *rapi_th(void *arg)
{
    fd_set      fds;
    int         fd;
    struct timeval tv;
```

```

int                rc = SUCCESSFUL;

/*****
/* This is the pthread created to handle RAPI upcalls. First, get */
/* the rapi socket descriptor to use on select().                */
*****/
pthread_mutex_lock(&rapi_lock);
fd = rapi_getfd(rapi_sid);
pthread_mutex_unlock(&rapi_lock);

if (fd > 0) {
/*****
/* Loop as long as all is well, waiting via select() for an    */
/* asynchronous RAPI packet to arrive.                          */
*****/
while (rc == SUCCESSFUL) {
    tv.tv_sec = 1;
    tv.tv_usec = 0;

    FD_ZERO(&fds);
    FD_SET(fd, &fds);
    switch(select(FD_SETSIZE, &fds, (fd_set *) NULL,
                  (fd_set *) NULL, &tv)) {
/*****
/* Bad return from select(). Get out.                            */
*****/
case -1:
    rc = UNSUCCESSFUL;
    break;
/*****
/* Time out on select(). Ignore.                                  */
*****/
case 0:
    break;

/*****
/* Dispatch data have arrived. Call the upcall function via */
/* rapi_dispatch().                                          */
*****/
default:
    pthread_mutex_lock(&rapi_lock);
    rc = rapi_dispatch();
    pthread_mutex_unlock(&rapi_lock);
    break;
    }
    }
}

/*****
/* Error on rapi_getfd().                                        */
*****/
else {
    rc = UNSUCCESSFUL;
}

pthread_exit(NULL);
}

```

rapi_dispatch - Dispatch API event

```

#include <rapi.h>

int rapi_dispatch(void)

```

Description

The application should call this routine whenever a read event is signaled on a file descriptor returned by `rapi_getfd()`. The `rapi_dispatch()` routine may be called at any time, but it will generally have no effect unless there is a pending event.

Parameters

There are no parameters to this call.

Result

Calling this routine may result in one or more upcalls to the application from any of the open API sessions known to this instance of the library.

If this call encounters an error, `rapi_dispatch()` returns a RAPI error code; otherwise, it returns 0. See “RAPI error codes” on page 178 for a list of error codes.

rapi_getfd - Get file descriptor

```
#include <rapi.h>
```

```
int rapi_getfd (rapi_sid_t Sid)
```

Description

After a `rapi_session()` call has completed successfully and before `rapi_release()` has been called, the application may call `rapi_getfd()` to obtain the file descriptor associated with that session. When a read event is signaled on this file descriptor, the application should call `rapi_dispatch()`.

Parameters

Sid This parameter is the session ID for the session initiated by a successful `rapi_session()` call.

Result

If *Sid* is illegal or undefined, this call returns -1; otherwise, it returns the file descriptor.

Error handling

Errors can be detected synchronously or asynchronously.

When an error is detected synchronously, a RAPI error code is returned in the *Errnop* argument of `rapi_session()`, or as the function return value of `rapi_sender()`, `rapi_reserve()`, `rapi_release()`, or `rapi_dispatch()`.

When an error is detected asynchronously, it is indicated by a `RAPI_PATH_ERROR` or `RAPI_RESV_ERROR` event. An RSVP error code and error value are then contained in the *ErrorCode* and *ErrorValue* arguments of the `event_upcall()` function. In case of an *API error* (RSVP error code 20), a RAPI error code is contained in the *ErrorValue* argument.

A description of RSVP error codes and values can be found in RFC 2205. See Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs.

RAPI error codes

[RAPI_ERR_OK]	No error
[RAPI_ERR_INVALID]	Parameter not valid
[RAPI_ERR_MAXSESS]	Too many sessions
[RAPI_ERR_BADSID]	Session identity out of legal range
[RAPI_ERR_N_FFS]	Wrong filter number or flow number for style
[RAPI_ERR_BADSTYLE]	Illegal reservation style
[RAPI_ERR_SYSCALL]	A system error has occurred; its nature may be indicated by <i>errno</i> .
[RAPI_ERR_OVERFLOW]	Parameter list overflow
[RAPI_ERR_MEMFULL]	Not enough memory
[RAPI_ERR_NORSVP]	The RSVP agent is not active or is unable to respond.
[RAPI_ERR_OBJTYPE]	Object type not valid
[RAPI_ERR_OBJLEN]	Object length not valid
[RAPI_ERR_NOTSPEC]	No sender Tspec
[RAPI_ERR_INTSERV]	Integrated Services parameter format not valid
[RAPI_ERR_GPI_CONFLICT]	IPSEC: Conflicting C-type
[RAPI_ERR_BADPROTO]	IPSEC: Protocol not AH or ESP
[RAPI_ERR_BADVDPOR]	IPSEC: vDstPort is 0.
[RAPI_ERR_GPISESS]	IPSEC: Parameters for GPI_SESSION flag not valid, or other parameter error
[RAPI_ERR_BADSEND]	Sender address not my interface
[RAPI_ERR_BADRECV]	Receiver address not my interface
[RAPI_ERR_BADSPORT]	Source port not valid: the source port is nonzero when the destination port is 0.

[RAPI_ERR_UNSUPPORTED]
Unsupported feature

[RAPI_ERR_UNKNOWN]
Unknown error

[RAPI_ERR_BADSEND], [RAPI_ERR_BADRECV] and [RAPI_ERR_BADSPORT]
occur only asynchronously, as the *ErrorValue* when the *ErrorCode* is 20 (API error).

RSVP error codes

Value	Symbol	Meaning
0	RSVP_Err_NONE	No error (confirmation)
1	RSVP_Err_ADMISSION	Admission control failure
2	RSVP_Err_POLICY	Policy control failure
3	RSVP_Err_NO_PATH	No path information
4	RSVP_Err_NO_SENDER	No sender information
5	RSVP_Err_BAD_STYLE	Conflicting style
6	RSVP_Err_UNKNOWN_STYLE	Unknown style
7	RSVP_Err_BAD_DSTPORT	Conflicting destination port in session
8	RSVP_Err_BAD_SNDPORT	Conflicting source port
9		Reserved
10		Reserved
11		Reserved
12	RSVP_Err_PREEMPTED	Service preempted
13	RSVP_Err_UNKN_OBJ_CLASS	Unknown object class
14	RSVP_Err_UNKNOWN_CTYPE	Unknown object C-Type
15		Reserved
16		Reserved
17		Reserved
18		Reserved
19		Reserved
20	RSVP_Err_API_ERROR	API error
21	RSVP_Err_TC_ERROR	Traffic control error
22	RSVP_Err_TC_SYS_ERROR	Traffic control system error
23	RSVP_Err_RSVP_SYS_ERROR	RSVP system error

Header files

Integer and floating point types

Types *u_int8_t*, *u_int16_t* and *u_int32_t*, which appear in the <rap.h> header file, are unsigned integer types of length 8, 16, and 32 bits, respectively.

Type *float32_t* is a floating-point type of length 32 bits. It is defined by including the <rap.h> header file.

The <rapi.h> header

This header file contains the definitions of the RSVP API (RAPI) library calls.

Inclusion of this header may make available other symbols in addition to those specified in this section.

General definitions

The following general definitions apply to the <rapi.h> header:

- Macro `RAPI_VERSION` is defined with value $100 * \text{major} + \text{minor}$, where *major* is the major version number and *minor* is the minor version number. The value of `RAPI_VERSION` is returned by `rapi_version()`.
- Type `rapi_addr_t` is defined for protocol addresses. It is defined to be *struct sockaddr*.
- Enumeration `qos_service_t` is defined by typedef and has at least the following members:

Member	Meaning
<code>QOS_CNTR_LOAD</code>	Controlled-load service
<code>QOS_GUARANTEED</code>	Guaranteed service
<code>QOS_TSPEC</code>	Generic Tspec

- Enumeration `rapi_format_t` is defined by typedef and has at least the following members:

Member	Meaning
<code>RAPI_ADSTYPE_Intserv</code>	Int-Serv format adspec
<code>RAPI_ADSTYPE_Simplified</code>	Simplified format adspec
<code>RAPI_EMPTY_OTYPE</code>	Empty object
<code>RAPI_FILTERFORM_BASE</code>	Simple V4: Only <i>sockaddr</i>
<code>RAPI_FLOWSTYPE_Intserv</code>	Int-Serv format flowspec
<code>RAPI_FLOWSTYPE_Simplified</code>	Simplified format flowspec
<code>RAPI_TSPECTYPE_Intserv</code>	Int-Serv format (sndr)Tspec
<code>RAPI_TSPECTYPE_Simplified</code>	Simplified format (sndr)Tspec

- Type `rapi_hdr_t` is defined by typedef as a structure to represent a generic RAPI object header. It has the following members, followed by type-specific contents:

Member	Type	Usage
<code>form</code>	int	Format
<code>len</code>	unsigned int	Actual length in bytes

Tspec definitions

The following Tspec definitions apply to the <rapi.h> header:

- Type `qos_Tspec_body` is defined by typedef as a structure with at least the following members:

Member	Type	Usage
<code>spec_Tspec_r</code>	float32_t	Token bucket average rate in bytes per second

Member	Type	Usage
spec_Tspec_b	float32_t	Token bucket depth in bytes
spec_Tspec_m	u_int32_t	Minimum policed unit in bytes
spec_Tspec_M	u_int32_t	Maximum packet size in bytes
spec_Tspec_p	float32_t	Peak data rate in bytes per second

- Type *qos_tspecx_t* is defined by typedef as a structure that contains the generic Tspec parameters, and has at least the following members:

Member	Type	Usage
spec_type	qos_service_t	QoS_service_type
xtspec_Tspec	qos_Tspec_body	Tspec

- The following macros are defined with the values given below:

Macro	Value
xtspec_r	xtspec_Tspec.spec_Tspec_r
xtspec_b	xtspec_Tspec.spec_Tspec_b
xtspec_m	xtspec_Tspec.spec_Tspec_m
xtspec_M	xtspec_Tspec.spec_Tspec_M
xtspec_p	xtspec_Tspec.spec_Tspec_p

- Type *rapi_tspec_t* is defined by typedef as a structure to represent a Tspec descriptor, and has at least the following members:

Member	Type	Usage
form	rapi_format_t	Tspec format
ISt	IS_tspbod_t	Int-serv format Tspec
len	unsigned int	Actual length in bytes
qosxt	qos_tspecx_t	Simplified format Tspec
tspecbody_u	union	

- The following macros are defined with the values given below:

Macro	Value
tspecbody_qosx	tspecbody_u.qosxt
tspecbody_IS	tspecbody_u.ISt

Flowspec definitions

The following flowspec definitions apply to the <rapi.h> header:

- Type *qos_flowspecx_t* is defined by typedef as a structure that contains the union of the parameters for *controlled-load service* and *guaranteed service* models, and has at least the following members:

Member	Type	Usage
spec_type	qos_service_t	QoS_service_type

Member	Type	Usage
xspec_R	float32_t	Rate in bytes per second
xspec_S	u_int32_t	Slack term in microseconds
xspec_Tspec	qos_Tspec_body	Tspec

- The following macros are defined with the values given below:

Macro	Value
xspec_r	xspec_Tspec.spec_Tspec_r
xspec_b	xspec_Tspec.spec_Tspec_b
xspec_m	xspec_Tspec.spec_Tspec_m
xspec_M	xspec_Tspec.spec_Tspec_M
xspec_p	xspec_Tspec.spec_Tspec_p

- Type *rapi_flowspec_t* is defined by typedef as a structure to represent a flowspec descriptor, and has at least the following members:

Member	Type	Usage
len	unsigned int	Actual length in bytes
form	rapi_format_t	Flowspec format
IS	IS_specbody_t	Int-serv format flowspec
specbody_u	union	
qosx	qos_flowspecx_t	Simplified format flowspec

- The following macros are defined with the values given below:

Macro	Value
specbody_qosx	specbody_u.qosx
specbody_IS	specbody_u.IS

Adspec definitions

The following adspec definitions apply to the <rapi.h> header:

- Type *qos_adspecx_t* is defined by typedef as a structure that contains the union of all *adspec* parameters for *controlled-load service* and *guaranteed service* models, and has at least the following members:

Member	Type	Usage
General path characterization parameters		
xaspec_flags	u_int8_t	Flags(1)
xaspec_hopcnt	u_int16_t	
xaspec_path_bw	float32_t	
xaspec_min_latency	u_int32_t	
xaspec_composed_MTU	u_int32_t	
Controlled-load service adspec parameters		
xClaspec_flags	u_int8_t	Flags
xClaspec_override	u_int8_t	See note (2)

Member	Type	Usage
xClaspec_hopcnt	u_int16_t	
xClaspec_path_bw	float32_t	
xClaspec_min_latency	u_int32_t	
xClaspec_composed_MTU	u_int32_t	
Guaranteed service adspec parameters		
xGaspec_flags	u_int8_t	Flags
xGaspec_Ctot	u_int32_t	
xGaspec_Dtot	u_int32_t	
xGaspec_Csum	u_int32_t	
xGaspec_Dsum	u_int32_t	
xGaspec_override	u_int8_t	See note (2)
xGaspec_hopcnt	u_int16_t	
xGaspec_path_bw	float32_t	
xGaspec_min_latency	u_int32_t	
xGaspec_composed_MTU	u_int32_t	

Notes:

- (1) FLG_IGN is not allowed; FLG_PARM is assumed.
 - (2) A value of 1 means "override all generic parameters."
- The following macros are defined with bitwise-distinct integral values for use in the *xaspec_flags* *xClaspec_flags* and *xGaspec_flags* fields:

Macro	Meaning
XASPEC_FLG_BRK	Break bit: service unsupported in some node.
XASPEC_FLG_IGN	Ignore flag: Do not include this service.
XASPEC_FLG_PARM	Parms-present flag: Include service parameters.

- Type *rapi_adspec_t* is defined by typedef as a structure to represent an adspec descriptor, and has at least the following members:

Member	Type	Usage
adsbody_u	union	
adsx	qos_adspecx_t	Simplified format adspec
form	rapi_format_t	adspec format
ISa	IS_adsbody_t	Int-serv format adspec
len	unsigned int	Actual length in bytes

- The following macros are defined with the values given below:

Macro	Value
adspecbody_IS	adsbody_u.ISa
adspecbody_qosx	adsbody_u.adsx

Filter spec definitions

The following filter spec definitions apply to the <rapi.h> header:

- Type *rapi_filter_base_t* is defined by typedef as a structure that contains at least the following member:

Member	Type
sender	struct sockaddr_in

- Type *rapi_filter_t* is defined by typedef as a structure that contains at least the following members:

Member	Type	Usage
base	rapi_filter_base_t	
filt_u	union	
form	rapi_format_t	Filterspec format
len	u_int32_t	actual length in bytes

- The following macros are defined with the values given below:

Macro	Value
rapi_filt4	filt_u.base.sender
rapi_filtbase4_addr	rapi_filt4.sin_addr
rapi_filtbase4_port	rapi_filt4.sin_port

Policy definitions

The following policy definitions apply to the <rapi.h> header:

Member	Type
form	rapi_format_t
len	u_int32_t
pol_u	union

Reservation style definitions

The following reservation style definitions apply to the <rapi.h> header:

- Enumeration *rapi_styleid_t* is defined by typedef for reservation style identifiers, and has at least the following members:

Member	Meaning
RAPI_RSTYLE_WILDCARD	Reservation will be shared among a wildcard selection of senders.
RAPI_RSTYLE_FIXED	Reservation will not be shared and will be dedicated to a particular sender.
RAPI_RSTYLE_SE	Reservation will be shared among an explicit list of senders.

- Type *rapi_stylex_t* is defined by typedef as *void*.

Function interface definitions

The following function interface definitions apply to the <rapi.h> header:

- Type *rapi_sid_t* is defined by typedef as *unsigned int* for RAPI client handles.

- Macro NULL_SID is defined for error returns from rapi_session().
- The following macro is defined and evaluated to a bitwise-distinct integral value:

Constant	Meaning
RAPI_USE_INTSERV	Use Int-Serv fmt in upcalls

Enumeration *rapi_eventinfo_t* is defined by typedef for RAPI event types, and has at least the following members:

Member
RAPI_PATH_ERROR
RAPI_PATH_EVENT
RAPI_RESV_CONFIRM
RAPI_RESV_ERROR
RAPI_RESV_EVENT

- The following macros are defined and evaluate to distinct integral values:

Constant	Meaning
RAPI_ERRF_InPlace	Left reservation in place
RAPI_ERRF_NotGuilty	This receiver not guilty

- Type rapi_event_rtn_t is defined by typedef as a function that conforms to the prototype defined in the definition for *event upcall*.
- The following macros are defined and evaluate to distinct integral values for use as RAPI error codes. Macro RAPI_ERR_OK (which indicates that there is no error) evaluates to 0.

Error code
RAPI_ERR_BADPROTO
RAPI_ERR_BADRECV
RAPI_ERR_BADSEND
RAPI_ERR_BADSID
RAPI_ERR_BADSPORT
RAPI_ERR_BADSTYLE
RAPI_ERR_BADVDPORT
RAPI_ERR_GPI_CONFLICT
RAPI_ERR_GPISESS
RAPI_ERR_INTSERV
RAPI_ERR_INVALID
RAPI_ERR_MAXSESS
RAPI_ERR_MEMFULL
RAPI_ERR_N_FFS
RAPI_ERR_NORSVP
RAPI_ERR_NOTSPEC
RAPI_ERR_OBJLEN

Error code
RAPI_ERR_OBJTYPE
RAPI_ERR_OK
RAPI_ERR_OVERFLOW
RAPI_ERR_SYSCALL
RAPI_ERR_UNKNOWN
RAPI_ERR_UNSUPPORTED

- The following macros are defined and evaluate to the RSVP error code values as defined in “RSVP error codes” on page 179:

Error code
RSVP_Err_ADMISSION
RSVP_Err_API_ERROR
RSVP_Err_BAD_DSTPORT
RSVP_Err_BAD_SNDPORT
RSVP_Err_BAD_STYLE
RSVP_Err_NONE
RSVP_Err_NO_PATH
RSVP_Err_NO_SENDER
RSVP_Err_POLICY
RSVP_Err_PREEMPTED
RSVP_Err_RSVP_SYS_ERROR
RSVP_Err_TC_ERROR
RSVP_Err_TC_SYS_ERROR
RSVP_Err_UNKN_OBJ_CLASS
RSVP_Err_UNKNOWN_STYLE
RSVP_Err_UNKNOWN_CTYPE

Integrated services data structures and macros

This section defines the integrated services (see RFC 2210) data formats. (See Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs.) The RAPI interface was designed to allow an application to specify either the *int-serv* format of a flowspec, Tspec, or adspec, or a simplified version of each.

The simplified versions allow almost any *int-serv* version to be generated, but there may be circumstances in which this is not adequate. For example, more general forms of flowspec, containing more than one service, may be defined in the future (so that in case the Resv message reaches a node that does not implement service A, it can drop back to service B). Allowing an application to specify the body of an arbitrary *int-serv* data object allows for such contingencies.

Future versions of this specification may change the definitions in this section. Application writers are advised not to use these definitions except when absolutely necessary.

Notes:

1. The values in the data structures defined in this section are in host byte order.
2. Inclusion of this header may make available other symbols in addition to those specified in this section.

General definitions

The following general definitions apply to the integrated services data structures and macros:

- The following macro is defined with the value given below:

Macro	Value	Usage
wordsof(x)	$((x+3)/4)$	number of 32-bit words

- The following macros are defined with the following integer values for service numbers:

Note: The values are protocol values defined in RFC 2211, RFC 2212, and RFC 2215. See Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs.

Macro	Value
GENERAL_INFO	1
GUARANTEED_SERV	2
CONTROLLED_LOAD_SERV	5

- Enumeration *int_serv_wkp* is defined for well-known parameter identities and has at least the following members with the following integer values:

Note: The values are protocol values defined in RFC 2215. See Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs.

Member	Value	Meaning
IS_WKP_HOP_CNT	4	Number of network nodes supporting Integrated Services along the flow path
IS_WKP_PATH_BW	6	Available bandwidth in bytes per second throughout the flow path
IS_WKP_MIN_LATENCY	8	Minimum end-to-end latency in microseconds
IS_WKP_COMPOSED_MTU	10	Maximum transmission unit without causing IP fragmentation along the flow path
IS_WKP_TB_TSPEC	127	Token-bucket TSPEC parameter

- The following macros are defined with the values given below:

Macro	Value
INTSERV_VERS_MASK	0xf0

Macro	Value
INTSERV_VERSION0	0
Intserv_Version(x)	((x)&ismh_version &INTSERV_VERS_MASK)>>4)
Intserv_Version_OK(x)	((x)->ismh_version &INTSERV_VERS_MASK)== \INTSERV_VERSION0)

- Type *IS_main_hdr_t* is defined by typedef as a structure to represent an Integrated Services main header, and has at least the following members:

Member	Type	Usage
ismh_len32b	u_int16_t	Number of 32-bit words excluding this header
ismh_unused	u_int8_t	
ismh_version	u_int8_t	Version

- Type *IS_serv_hdr_t* is defined by typedef as a structure to represent an Integrated Services service element header, and has at least the following members:

Member	Type	Usage
issh_flags	u_int8_t	Flag byte
issh_len32b	u_int16_t	Number of 32-bit words excluding this header
issh_service	u_int8_t	Service number

- The following macro is defined with the value given below to indicate the *break* bit in the *IS_serv_hdr_t* flag byte:

Macro	Value
ISSH_BREAK_BIT	0x80

- Type *IS_parm_hdr_t* is defined by typedef as a structure to represent an Integrated Services parameter element header, and has at least the following members:

Member	Type	Usage
isph_flags	u_int8_t	Flags
isph_len32b	u_int16_t	Number of 32-bit words excluding this header
isph_parm_num	u_int8_t	Parameter number

- The following macro is defined with the value given below to indicate the *not valid* bit in the *IS_parm_hdr_t* flag byte:

Macro	Value
ISPH_FLG_INV	0x80

- The following macros are defined with the values given below:

Macro	Value
Next_Main_Hdr(p)	(IS_main_hdr_t*)((u_int32_t) (p)+1+(p)->ismh_len32b)

Macro	Value
Next_Parm_Hdr(p)	(IS_parm_hdr_t *)((u_int32_t *) (p) + 1 + (p)->isph_len32b)
Next_Serv_Hdr(p)	(IS_serv_hdr_t *)((u_int32_t *) (p) + 1 + (p)->iss_h_len32b)
Non_Is_Hop	((IS_serv_hdr_t *) p)->iss_h_flags & ISSH_BREAK_BIT
Set_Break_Bit(p)	((IS_serv_hdr_t *) p)->iss_h_flags = ISSH_BREAK_BIT
Set_Main_Hdr(p, len)	{(p)->ismh_version = INTSERV_VERSION0; \ (p)->ismh_unused = 0; \ (p)->ismh_len32b = wordsof(len); }
Set_Parm_Hdr(p, id, len)	{(p)->isph_parm_num = (id); \ (p)->isph_flags = 0; \ (p)->isph_len32b = wordsof(len); }
Set_Serv_Hdr(p, s, len)	{(p)->iss_h_service = (s); \ (p)->iss_h_flags = 0; \ (p)->iss_h_len32b = wordsof(len); }

Generic tspec format

The following generic tspec formats apply to the integrated services data structures and macros:

- The following macros define constraints on the *token bucket* parameters for both the controlled-load and guaranteed service. These constraints are imposed by the respective service specifications and are not an indication of what minimum or maximum values a RAPI implementation will accept.

The following macros are defined with values of type *float32_t*:

Macro	Usage	Value
TB_MIN_RATE	Minimum token bucket rate	1 byte per second
TB_MAX_RATE	Maximum token bucket rate	40 terabytes per second
TB_MIN_DEPTH	Minimum token bucket depth	1 byte
TB_MAX_DEPTH	Maximum token bucket depth	250 gigabytes
TB_MAX_PEAK	Maximum peak rate	Positive infinity, defined as an IEEE single-precision floating-point number with an exponent of all ones (255) and a sign and mantissa of all zeros (see RFC 1832; see Appendix E, "Related protocol specifications," on page 719 for information about accessing RFCs)

- Type *TB_Tsp_parms_t* is defined by typedef as a structure to represent generic Tspec parameters, and has at least the following members:

Member	Type	Usage
TB_Tspec_b	float32_t	Token bucket depth in bytes
TB_Tspec_m	u_int32_t	Minimum policed unit in bytes

Member	Type	Usage
TB_Tspec_M	u_int32_t	Maximum packet size in bytes
TB_Tspec_p	float32_t	Peak data rate in bytes per second
TB_Tspec_r	float32_t	Token bucket rate in bytes per second

- Type *gen_Tspec_t* is defined by typedef as a structure to represent a generic Tspec, and has at least the following members:

Member	Type	Usage
gen_Tspec_parms	TB_Tsp_parms_t	
gen_Tspec_parm_hdr	IS_parm_hdr_t	(IS_WKP_TB_TSPEC,)
gen_Tspec_serv_hdr	IS_serv_hdr_t	(GENERAL_INFO, length)

- The following macros are defined with the values given below:

Macro	Value
gtspec_b	gen_Tspec_parms.TB_Tspec_b
gtspec_flags	gen_Tspec_parm_hdr.isph_flags
gtspec_len	(sizeof(gen_Tspec_t) - sizeof(IS_serv_hdr_t))
gtspec_len32b	gen_Tspec_parm_hdr.isph_len32b
gtspec_m	gen_Tspec_parms.TB_Tspec_m
gtspec_M	gen_Tspec_parms.TB_Tspec_M
gtspec_p	gen_Tspec_parms.TB_Tspec_p
gtspec_parmno	gen_Tspec_parm_hdr.isph_parm_num
gtspec_r	gen_Tspec_parms.TB_Tspec_r

Formats for controlled-load service

The following formats for controlled-load service apply to the integrated services data structures and macros:

- Type *CL_flowspec_t* is defined by typedef as a structure to represent a controlled-load flowspec, and has at least the following members:

Member	Type	Usage
CL_spec_parms	TB_Tsp_parms_t	
CL_spec_parm_hdr	IS_parm_hdr_t	(IS_WKP_TB_TSPEC)
CL_spec_serv_hdr	IS_serv_hdr_t	(CONTROLLED_LOAD_SERV, 0,len)

- The following macros are defined with the values given below:

Macro	Value
CLspec_b	CL_spec_parms.TB_Tspec_b
CLspec_flags	CL_spec_parm_hdr.isph_flags
CLspec_len	(sizeof(CL_flowspec_t) - sizeof(IS_serv_hdr_t))

Macro	Value
CLspec_len32b	CL_spec_parm_hdr.isph_len32b
CLspec_m	CL_spec_parms.TB_Tspec_m
CLspec_M	CL_spec_parms.TB_Tspec_M
CLspec_p	CL_spec_parms.TB_Tspec_p
CLspec_parmno	CL_spec_parm_hdr.isph_parm_num
CLspec_r	CL_spec_parms.TB_Tspec_r

Formats for guaranteed service

The following formats for guaranteed service apply to the integrated services data structures and macros:

- The following enumeration is defined for service-specific parameter identifiers and has at least the following members with the following values:

Member	Value
IS_GUAR_RSPEC	130
GUAR_ADSPARM_C	131
GUAR_ADSPARM_D	132
GUAR_ADSPARM_Ctot	133
GUAR_ADSPARM_Dtot	134
GUAR_ADSPARM_Csum	135
GUAR_ADSPARM_Dsum	136

- Type *guar_Rspec_t* is defined by typedef as a structure for guaranteed Rspec parameters, and has at least the following members:

Member	Type	Usage
Guar_R	float32_t	Guaranteed rate in bytes per second
Guar_S	u_int32_t	Slack term in microseconds

- Type *Guar_flowspec_t* is defined by typedef as a structure to represent a guaranteed flowspec, and has at least the following members:

Member	Type	Usage
Guar_Rspec	guar_Rspec_t	Guaranteed rate in Bytes per second
Guar_Rspec_hdr	IS_parm_hdr_t	(IS_GUAR_RSPEC)
Guar_serv_hdr	IS_serv_hdr_t	(GUARANTEED_SERV, 0, length)
Guar_Tspec_hdr	IS_parm_hdr_t	(IS_WKP_TB_TSPEC)
Guar_Tspec_parms	TB_Tsp_parms_t	GENERIC Tspec parameters

- The following macros are defined with the values given below:

Macro	Value
Gspec_b	Guar_Tspec_parms.TB_Tspec_b

Macro	Value
Gspec_len	(sizeof(Guar_flowspec_t) - sizeof(IS_serv_hdr_t))
Gspec_m	Guar_Tspec_parms.TB_Tspec_m
Gspec_M	Guar_Tspec_parms.TB_Tspec_M
Gspec_p	Guar_Tspec_parms.TB_Tspec_p
Gspec_r	Guar_Tspec_parms.TB_Tspec_r
Gspec_R	Guar_Rspec.Guar_R
Gspec_R_flags	Guar_Rspec_hdr.isph_flags
Gspec_R_len32b	Guar_Rspec_hdr.isph_len32b
Gspec_R_parmno	Guar_Rspec_hdr.isph_parm_num
Gspec_S	Guar_Rspec.Guar_S
Gspec_T_flags	Guar_Tspec_hdr.isph_flags
Gspec_T_len32b	Guar_Tspec_hdr.isph_len32b
Gspec_T_parmno	Guar_Tspec_hdr.isph_parm_num

- Type *Gads_parms_t* is defined by typedef as a structure for guaranteed adspec parameters, and has the following members, which may be followed by override general parameter values:

Member	Type	Usage
Gads_Csum	u_int32_t	
Gads_Csum_hdr	IS_parm_hdr_t	(GUAR_ADSPARM_Csum)
Gads_Ctot	u_int32_t	
Gads_Ctot_hdr	IS_parm_hdr_t	(GUAR_ADSPARM_Ctot)
Gads_Dsum	u_int32_t	
Gads_Dsum_hdr	IS_parm_hdr_t	(GUAR_ADSPARM_Dsum)
Gads_Dtot	u_int32_t	
Gads_Dtot_hdr	IS_parm_hdr_t	(GUAR_ADSPARM_Dtot)
Gads_serv_hdr	IS_serv_hdr_t	(GUARANTEED_SERV, x, len)

Basic adspec pieces

The following basic adspec pieces apply to the integrated services data structures and macros:

- Type *genparm_parms_t* is defined by typedef as a structure for general path characterization parameters, and has at least the following members:

Member	Type	Usage
gen_parm_compmtu_hdr	IS_parm_hdr_t	(IS_WKP_COMPOSED_MTU)
gen_parm_composed_MTU	u_int32_t	
gen_parm_hdr	IS_serv_hdr_t	(GENERAL_INFO, len)
gen_parm_hopcnt	u_int32_t	
gen_parm_hopcnt_hdr	IS_parm_hdr_t	(IS_WKP_HOP_CNT)
gen_parm_min_latency	u_int32_t	

Member	Type	Usage
gen_parm_minlat_hdr	IS_parm_hdr_t	(IS_WKP_MIN_LATENCY)
gen_parm_path_bw	float32_t	
gen_parm_pathbw_hdr	IS_parm_hdr_t	(IS_WKP_PATH_BW)

- Type *Min_adspec_t* is defined by typedef as a structure to represent a minimal adspec per-service fragment (an empty service header) and has at least the following member.

Member	Type	Usage
mads_hdr	IS_serv_hdr_t	(<service>, 1, len=0)

Integrated services flowspec

The following integrated services flowspecs apply to the integrated services data structures and macros:

- Type *IS_specbody_t* is defined by typedef as a structure to represent an integrated services flowspec, and has at least the following members:

Member	Type	Usage
CL_spec	CL_flowspec_t	Controlled-load service
G_spec	Guar_flowspec_t	Guaranteed service
spec_mh	IS_main_hdr_t	
spec_u	union	

- The following macros are defined with the values given below:

Macro	Value
ISmh_len32b	spec_mh.ismh_len32b
ISmh_unused	spec_mh.ismh_unused
ISmh_version	spec_mh.ismh_version

Integrated services tspec

The following integrated services tspecs apply to the integrated services data structures and macros:

- Type *IS_tsbody_t* is defined by typedef as a structure to represent an Integrated Services Tspec, and has at least the following members:

Member	Type	Usage
st_mh	IS_main_hdr_t	
tspec_u	union (1)	
gen_stspec	gen_Tspec_t	Generic Tspec

Note:

- (1) While service-dependent Tspecs are possible, there are none.

- The following macros are defined with the values given below:

Macro	Value
IStmh_len32b	st_mh.ismh_len32b
IStmh_unused	st_mh.ismh_unused
IStmh_version	st_mh.ismh_version

Integrated services adspec

The following integrated services adspecs apply to the integrated services data structures and macros:

Member	Type	Usage
adspec_genparms	genparm_parms_t	General char parameter fragment
adspec_mh	IS_main_hdr_t	Main header

Chapter 7. X Window System interface in the z/OS Communications Server environment

This topic describes the X Window System application programming interface (API). The X Window System API allows you to write applications in the MVS environment that can be displayed on X11 servers on a TCP/IP-based network, and provides the application with graphics capabilities as defined by the X Window System protocol.

X11 and Motif libraries are based on the X Window System Version 11 Release 6.6 and Motif Version 2.1.30. Applications are supported in 31- and 64-bit mode. For compatibility with applications written for prior releases, X11 R6.1 and Motif 1.2 libraries and corresponding header files are also provided.

X Window System and Motif

This section describes the X Window System API. The X Window System API allows you to write applications in the z/OS UNIX System Services (z/OS UNIX) MVS environment.

The X Window System support includes the following APIs from the X Window System Version 11 Release 6.6:

- X11 Core distribution routines (X11)
- Inter-Client Exchange routines (ICE)
- Session Manager routines (SM)
- X Window System extended routines (Xext) including:
 - XC-MISC: Allows clients to get back ID ranges from the server
 - Big-Requests: Allows large length value in protocol requests
 - Shape: Allows nonrectangular windows
 - Sync: Lets clients synchronize through the X Server
- Authentication functions (Xau)
- X10 compatibility routines (oldX)
- X Toolkit (Xt)
- Utility functions used by Xaw (Xmu)
- Athena Widget set (Xaw)
- Header files needed for compiling X clients
- Selection of standard MIT X clients
- Sample X demonstrations
- Sample Motif demonstrations

The X Window System support provided also includes the APIs based on Motif Release 2.1.30:

- Motif-based widget set (Xm library)
- Motif Resource Manager (Mrm library)
- Motif User Interface language (uil library)
- Motif User Interface Language Compiler
- Header files needed for compiling clients using the Motif-based widget set

DLL support for the X Window System

The X Window System and Motif functions are provided as a set of archive files for applications that are statically linked and as a set of DLLs for applications that are dynamically linked. Dynamic linkage is recommended; it results in application binaries that are much smaller. All applications linked using these DLLs must be compiled with the DLL option. The examples shown in “Compiling and linking Motif and X Window System applications” on page 199 assume that c89 is using the z/OS C/C ++ Compiler.

Three sets of DLLs are provided. The first set ensures compatibility with applications compiled with previous releases of the X Window System and Motif. For this set of DLLs applications must be compiled in 31-bit mode with the DLL option; applications cannot be compiled with XPLINK. These DLLs are unchanged from the previous release and are compiled with IBM hexadecimal floating point support. New or changed applications should be migrated to the new X11R6.6 and Motif 2.1.30 versions of the libraries.

The following DLLs are provided to support applications that require X11R6.1 and Motif 1.2 function. These libraries are provided to ensure compatibility of applications written for previous releases of z/OS.

- X11.dll (contains the contents of libX11.a, libXau.a, liboldX.a, and libXext.a)
- SM.dll (contains the contents of libSM.a)
- ICE.dll (contains the contents of libICE.a)
- PEX5.dll (contains the contents of libPEX5.a)
- Xaw.dll (contains the contents of libXaw.a, libXmu.a, and libXt.a)
- Xm.dll (contains the contents of libXm.a and libXt.a)
- Mrm.dll (contains the contents of libMrm.a)
- Uil.dll (contains the contents of libUil.a)

The second set of DLLs provides X11R6.6 and Motif 2.1 function. To use this set of DLLs, the application must be compiled in 31-bit mode, with the DLL option and the XPLINK option. This set of DLLs is compiled with IEEE floating point support. These DLLs do not support applications compiled with enhanced ASCII support. The PEX5 library is no longer supported with these DLLs.

- X11_31.dll (contains the contents of libX11.a, libXau.a, liboldX.a, libXext.a, and libXp.a)
- SM_31.dll (contains the contents of libSM.a)
- ICE_31.dll (contains the contents of libICE.a)
- Xaw_31.dll (contains the contents of libXaw.a, libXmu.a, and libXt.a)
- Xm_31.dll (contains the contents of libXm.a and libXt.a)
- Mrm_31.dll (contains the contents of libMrm.a)
- Uil_31.dll (contains the contents of libUil.a)

The third set of DLLs provides X11R6.6 and Motif 2.1 function in 64-bit addressing mode. To use this set of DLLs, the application must be compiled in 64-bit mode, with the DLL option and the XPLINK option.

- X11_64.dll (contains the contents of libX11.a, libXau.a, liboldX.a, libXext.a, and libXp.a)
- SM_64.dll (contains the contents of libSM.a)
- ICE_64.dll (contains the contents of libICE.a)
- Xaw_64.dll (contains the contents of libXaw.a, libXmu.a, and libXt.a)

- Xm_64.dll (contains the contents of libXm.a and libXt.a)
- Mrm_64.dll (contains the contents of libMrm.a)
- Uil_64.dll (contains the contents of libUil.a)

All DLLs, along with their sidedecks (.x), are symbolically linked from /usr/lib.

Rules:

- An application must use only one set of DLLs. You cannot mix 31- and 64-bit DLLs. An application should not attempt to mix old- and new-function DLLs.
- An application should use either the static libraries or the dynamic libraries, not both.

How the X Window System interface works in the MVS environment

The X Window System is a network-transparent protocol that supports windowing and graphics. The protocol is communicated between a client or application and an X server over a reliable bidirectional byte stream. This byte stream is provided by the TCP/IP communication protocol. In the MVS environment, X Window System support consists of a set of application calls that create the X protocol, as requested by the application. This application programming interface allows an application to be created, which uses the X Window System protocol to be displayed on an X server.

In an X Window System environment, the X server is generally located on the workstation, and distributes user input to and accepts requests from various client programs located either on the same system or elsewhere on a network. The X server provides access to the resources that are shared among many X applications, such as the screen, keyboard, mouse, fonts, and graphics contexts. A single X server can control more than one physical screen.

The application program that you create is the client part of a client-server relationship. The communication path from the MVS X Window System application to the server involves the client code and TCP/IP.

The X client code uses sockets to communicate with the X server. Each client can interact with multiple servers, and each server can interact with multiple clients.

If your application is written to the Xlib interface, it calls XOpenDisplay() to start communication with an X server on a workstation. The Xlib code opens a communication path called a socket to the X server, and sends the appropriate X protocol to initiate client-server communication.

The X protocol generated by the X Window System client code uses an ISO Latin-1 encoding for character strings, while the MVS encoding for character strings is EBCDIC. The X Window System client code in the MVS environment automatically transforms character strings from EBCDIC to ISO Latin-1 or from ISO Latin-1 to EBCDIC, as needed.

z/OS UNIX application resource file

The X Window System allows you to modify certain characteristics of an application at run time using application resources. Typically, application resources are set to tailor the appearance and possibly the behavior of an application. The application resources can specify information about an application's window sizes, placement, coloring, font usage, and other functional details.

In the z/OS UNIX environment, this information can be found in the file
`/u/user_id/.Xdefaults`

where
`/u/user_id`

is found from the environment variable *home*.

Identifying the target display in z/OS UNIX

The *DISPLAY* environment variable is used by the X Window System to identify the host name of the target display.

The following is the format of the *DISPLAY* environment variable:

`host_name:target_server.target_screen`

Value	Description
host_name	Specifies the host name or IP address of the host machine on which the X Window System server is running.
target_server	Specifies the display number on the host machine. This is usually 0, unless the host machine is running multiple X servers.
target_screen	Specifies the screen to be used on the target server. This is optional and defaults to 0.

For more information about resolving a host name to an IP address, see the z/OS XL C/C++ *Programming Guide*.

Programming considerations

Porting Motif applications to z/OS UNIX MVS

The X Window System toolkit includes files that define two macros for obtaining the offset of fields in an X Window System Toolkit structure, *XtOffset*, and *XtOffsetOf*. Programs written for, or ported to, z/OS UNIX MVS must use the *XtOffsetOf* macro for this purpose.

Some Motif widget and gadget resources have the type *KeySym*. In an ASCII-based system the *KeySym* is the same as the ASCII character value. For example, the character 'F' has the ASCII hexadecimal value 46 and a *KeySym* hexadecimal value of 46.

However, on z/OS UNIX MVS, the character value of 'F' is hexadecimal C6, while the *KeySym* hexadecimal value is still 46. Remember to use true *KeySym* values when specifying resources of type *KeySym*, whether in a defaults file or in a function call.

In some cases, an X Window System server may have clients that are not running on z/OS UNIX MVS. If a z/OS UNIX MVS X Window System application sends nonstandard properties that contain text strings to the X Window System server, and these properties might be accessed by clients that are not running on z/OS UNIX MVS, the strings should be translated. The translation should be to the server default character set before transmission to the server and to the appropriate host character set when retrieved from the server. This translation is an application responsibility.

Compiling and linking Motif and X Window System applications

The z/OS UNIX c89 or **make** commands should be used to compile and link X Window System and Motif programs. The following example shows how to use the c89 command to compile an X Window System program, xxx, which uses the Athena widget set, and create the executable file xxx. All code that uses the X Window System and Motif libraries must be compiled with the DLL option even if static linking is used.

```
c89 -o xxx -Wc,dll,xplink -Wl,xplink xxx.c /usr/lib/Xaw_31.x /usr/lib/SM_31.x /usr/lib/ICE_31.x /usr/lib/X11_31.x
```

The following example shows how to compile the program xxx for use with the 64-bit DLLs. LP64 also requires the use of XPLINK.

```
c89 -o xxx -Wc,dll,xplink,LP64 -Wl,xplink xxx.c /usr/lib/Xaw_64.x /usr/lib/SM_64.x /usr/lib/ICE_64.x /usr/lib/X11_64.x
```

The following example shows how to use the c89 command to compile an X Window System program, yyy, which uses the Motif widget set, and create an executable file yyy:

```
c89 -o yyy -Wc,dll,xplink -Wl,xplink yyy.c /usr/lib/Xm_31.x /usr/lib/SM_31.x /usr/lib/ICE_31.x /usr/lib/X11_31.x
```

The following example shows how to use the c89 command to compile an X Window System program, yyy, which uses the Motif widget set, and create an executable file yyy. This example links with the previous function libraries (X 6.1 and Motif 1.2). You must explicitly tell the compiler where to pick up the header files for the previous function libraries with the -I option.

```
c89 -o yyy -Wc,dll -Wl,xplink yyy.c -I/usr/include/lpp/tcpip/X11R6/include /usr/lib/Xm.x /usr/lib/SM.x /usr/lib/ICE.x /usr/lib/X11.x
```

For examples of the input to the **make** command, see the Makefile in each of these subdirectories:

```
/usr/lpp/tcpip/X11R6/Xamples/demos
/usr/lpp/tcpip/X11R6/Xamples/clients
/usr/lpp/tcpip/X11R66/Xamples/demos
/usr/lpp/tcpip/X11R66/Xamples/clients
/usr/lpp/tcpip/X11R66/Xamples/motif
```

To build the samples for X11 and Motif, set the following environment variables:

- export _C89_CCMODE=1
- export _CC_CCMODE=1

Setting these environment variables causes the c89 and cc commands to relax requirements on the order of options and operands and makes the porting of makefiles from other platforms easier.

For more information about the z/OS UNIX c89 and make commands, see the *z/OS UNIX System Services Command Reference*.

Running an X Window System or Motif DLL-enabled application

When running an X Window System or Motif DLL-enabled application, ensure that the LIBPATH environment variable includes /usr/lib.

X Window System environment variables

Table 4 provides a list of environment variables that can be set to affect the behavior of X Window System applications.

Table 4. Environment variables for the X Window System interface

Environment variable	Description
DISPLAY	Contains the name of the display to be used. There is no default value. See note 1.
ICEAUTHORITY	This variable identifies where the authentication information is located.
LANG	Determines the locale category for native language, local customs, and coded character set in the absence of the LC_ALL and other LC_* (LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC, LC_TIME) environment variables. See note 2.
LC_CTYPE	Determine the locale category for character handling functions, such as tolower(), toupper(), and isalpha(). This environment variable determines the interpretation of sequences of bytes of text data as characters (for example, single as opposed to multibyte characters), the classification of characters (for example, alpha, digit, graph), and the behavior of character classes.
SESSION_MANAGER	If defined, causes a Session Shell widget to connect to a session manager. There is no default value.
XAPPLRESDIR	Specifies the directory to search for files that contain application defaults.
XAUTHORITY	Specifies the name of the authority file on the local host.
XCMSDB	Specifies the name of a color name database file.
XENVIRONMENT	Contains the full path name of the file that contains resource defaults. There is no default value.
XFILESEARCHPATH	Used by XtResolvePathname as a default path. There is no default value.
XKEYSYMDB	Specifies the location of the XKEYSYMDB.
XLOCALEDIR	Specifies the directory to search for locale files. The default value is /usr/lib/X11/locale.
XMODIFIERS	Can be set to contain additional information important for the current locale setting. See note 3.
XUSERFILESEARCHPATH	Specifies where to find the personal X resources files used to configure an application.
XWTRACE	Controls the generation of socket-level communications traces between Xlib and the X Window System server. These traces are as follows: • XWTRACE undefined or 0: No trace generated • XWTRACE=1: Error messages • XWTRACE>=2: API function tracing for TRANS functions There is no default value. The output is sent to stderr.

Table 4. Environment variables for the X Window System interface (continued)

Environment variable	Description
XWTRACELC	If defined, causes a trace of locale-sensitive routines. Possible values are: • XWTRACELC undefined or 0: No trace generated • XWTRACELC=1: Error messages • XWTRACELC>=2: All available trace information There is no default value. The output is sent to stderr. See note 4.
Windows Notes: 1. In the following example, royal.csc.ibm.com is the name of the workstation running the X Window System server. The display is indicated by :0.0, and is specified this way in almost all cases. <pre>export DISPLAY=royal.csc.ibm.com:0.0</pre> 2. This can be used by applications to determine the language to use for error messages, instructions, collating sequences, date formats, and so on. 3. Typically set to @im=<input-method> to enable a particular input method. 4. If XWTRACELC is defined, a routine flow trace is generated. If XWTRACELC=2, more detailed information is provided.	

Motif environment variables

Table 5 provides a list of environment variables that can be set to affect the behavior of Motif applications.

Table 5. Environment variables for Motif

Environment variable	Description
DTICONBMSEARCHPATH	Contains the search path for icons on monochrome displays.
DTICONSEARCHPATH	Contains the search path for icons on color displays.
KBD_LANG	Specifies the value of LANG for applicable languages.
RESOURCE_NAME	Used by XtOpenDisplay as an alternative specification of an application name. There is no default value.
UILTRACE	Specifies whether UIL trace is on or off.
WMDPATH	Specifies the WMD path.
XAPPLRESDIR	Specifies the directory to search for files that contain application defaults.
XMBINDDIR	Specifies the location of the xmbind.alias file.
XMICONBMSEARCHPATH	Used to locate desktop icons.
XMICONSEARCHPATH	Used to locate bitmap (2-color) desktop icons.
XPROPFORMATS	Specifies the name of the file from which additional formats are to be obtained.

EBCDIC/ASCII translation in the X Window System

Because the X Window System was designed primarily for an ASCII-based environment and z/OS UNIX MVS uses EBCDIC, it is necessary to provide translations between various servers and MVS clients. Translations must also be provided between locale-based coded character sets in z/OS UNIX MVS and the coded character sets used on the X Window System server. The following sections describe how this is accomplished.

Locale independent translation

All arguments for X Window System functions that are specified to be in the Host Portable Character Set are translated between EBCDIC and ASCII by a translation between code page IBM-1047 and code page ISO8859-1. All single-byte character set string arguments to X Window System function calls that are not locale-dependent (do not have names starting with Xmb or Xwc) are also translated between EBCDIC and ASCII using code page IBM-1047 and ISO8859-1. In addition, properties of type STRING passed to XChangeProperty are translated to ASCII before transmission to the server.

These translations are performed on data being transmitted to the server and on data received from the server that is being returned to the application.

The arguments to X Window System functions of the type XChar2b are not translated. This includes such functions as XDraw16, XDrawText16, and XTextExtents16.

Locale dependent translation

The string arguments to X Window System functions with names starting with Xmb or Xwc are translated between the current MVS z/OS UNIX locale code set (the value returned by nl_info(CODESET)) and the current XLocale. The MVS z/OS UNIX locale is mapped to the XLocale by an entry in /usr/lib/X11/locale/locale.alias. Properties passed to XChangeProperty with a type of the locale-encoding name atom are translated from the MVS z/OS UNIX locale-coded character set to the XLocale coded character set.

XTextProperty with COMPOUND_TEXT encoding

The XTextProperty structure returned by XmbTextListToProperty and XwcTextListToProperty has its property data translated from the MVS z/OS UNIX locale coded character set to the XLocale coded character set if the XTextProperty encoding is COMPOUND_TEXT. Similarly the reverse translation is performed for XmbTextPropertyToTextList and XwcTextPropertyToTextList if the XTextProperty has the encoding COMPOUND_TEXT.

Standard clients supplied with MVS z/OS UNIX X Window System support

The following standard clients are provided in /usr/lpp/tcpip/X11R6/Xamples/clients:

Client	Description
appres	Lists application resource database
atobm	Bit map conversion utility
bitmap	Bit map editor
bmtoa	Bit map conversion utility
editres	Resource editor

Client	Description
iceauth	ICE authority file utility
oclock	Displays time of day
xauth	X authority file utility
xclipboard	Clipboard utility
xcutsel	Clipboard utility
clock	Analog and digital clock for X
xdpyinfo	Display information utility for X
xfd	X font display utility
xlogo	Displays X logo
xlsatoms	Lists interned atoms defined on server
xlsclients	Lists client applications running on a display
xmag	Magnifies part of screen
xlsfonts	Lists server fonts
xprop	Property displayer for X
xwininfo	Window information utility for X
xwd	Dumps an image of an X window
xwud	Displays dumped image for X
xfindproxy	Find an LBX proxy

Use the **man** command to display information about these clients as shown below:

```
man -M /usr/lpp/tcpip/X11R6/Xamples/man client
```

Demonstration programs supplied with MVS z/OS UNIX X Window System support

The following demonstration programs are supplied in /usr/lpp/tcpip/X11R6/Xamples/demos:

xsamp1

Uses only Xlib

xsamp2

Uses Athena widget set

xsamp3

Uses Motif widget set

pexsamp

Uses PEX5 library

Where files are located

The following sections provide X Window System and Motif locations.

Previous function X11R6.1 and Motif 1.2

- Previous function X11R6.1 and Motif 1.2 static libraries for 31-bit applications. Applications that want to link with these libraries must use the -L flag on the cc or c89 command to specify the library directory.

```
/usr/lpp/tcpip/X11R6/lib/libX11.a
/usr/lpp/tcpip/X11R6/lib/libXext.a
/usr/lpp/tcpip/X11R6/lib/liboldX.a
/usr/lpp/tcpip/X11R6/lib/libICE.a
/usr/lpp/tcpip/X11R6/lib/libSM.a
```



```

/usr/lpp/tcpip/X11R6/lib/libXt.a
/usr/lpp/tcpip/X11R6/lib/libXmu.a
/usr/lpp/tcpip/X11R6/lib/libXaw.a
/usr/lpp/tcpip/X11R6/lib/libXau.a
/usr/lpp/tcpip/X11R6/lib/libPEX5.a
/usr/lpp/tcpip/X11R6/lib/libXm.a
/usr/lpp/tcpip/X11R6/lib/libMrm.a
/usr/lpp/tcpip/X11R6/lib/libUil.a

```

- Previous function X11R6.1 and Motif 1.2 dynamic link libraries (DLLs); 31-bit, non-XPLINK:

```

/usr/lib/X11.dll -> symlink to /usr/lpp/tcpip/X11R6/lib/X11.dll
/usr/lib/ICE.dll -> symlink to /usr/lpp/tcpip/X11R6/lib/ICE.dll
/usr/lib/SM.dll -> symlink to /usr/lpp/tcpip/X11R6/lib/SM.dll
/usr/lib/Xaw.dll -> symlink to /usr/lpp/tcpip/X11R6/lib/Xaw.dll

```

- Header files for previous function X11R6.1 and Motif 1.2:

```

/usr/lpp/tcpip/X11R6/include/X11
/usr/lpp/tcpip/X11R6/include/X11/ICE
/usr/lpp/tcpip/X11R6/include/X11/PEX5
/usr/lpp/tcpip/X11R6/include/X11/SM
/usr/lpp/tcpip/X11R6/include/X11/Xaw
/usr/lpp/tcpip/X11R6/include/X11/Xmu
/usr/lpp/tcpip/X11R6/include/X11/bitmaps
/usr/lpp/tcpip/X11R6/include/X11/extensions

```

```

/usr/lpp/tcpip/X11R6/include/Mrm (motif header files)
/usr/lpp/tcpip/X11R6/include/Xm (motif header files)
/usr/lpp/tcpip/X11R6/include/Uil (Uil header files)

```

- Other utilities and data files for the previous function X11R6.1 and Motif 1.2:

```

/usr/lpp/tcpip/bin/X11/uil (uil compiler)

/usr/lpp/tcpip/X11R6/lib/X11/locale (locale data files)
/usr/lpp/tcpip/X11R6/lib/X11/XErrorDB (X Error message database)
/usr/lpp/tcpip/X11R6/lib/X11/XKeysymDB (X keysym Database)
/usr/lpp/tcpip/X11R6/lib/X11/app-defaults/ (application default files)

```

- Examples shipped for X11R6.1 and Motif 1.2:

```

/usr/lpp/tcpip/X11R6/Xamples/man/cat1/ (man pages for Xamples programs)
/usr/lpp/tcpip/X11R6/Xamples/demos/ (demonstration programs)
/usr/lpp/tcpip/X11R6/Xamples/clients/ (selected standard clients)

```

New function X11R6.6 and Motif 2.1.30

- New function X11R6.6 and Motif 2.1 static libraries for 31- and 64-bit applications (these libraries are all XPLINK):

Notes:

1. PEX is no longer supported in these libraries.
2. Xp is a new library.

```

/usr/lib/libX11.a -> /usr/lpp/tcpip/X11R66/lib/libX11.a
/usr/lib/libXext.a -> /usr/lpp/tcpip/X11R66/lib/libXext.a
/usr/lib/liboldX.a -> /usr/lpp/tcpip/X11R66/lib/liboldX.a
/usr/lib/libICE.a -> /usr/lpp/tcpip/X11R66/lib/libICE.a
/usr/lib/libSM.a -> /usr/lpp/tcpip/X11R66/lib/libSM.a
/usr/lib/libXt.a -> /usr/lpp/tcpip/X11R66/lib/libXt.a
/usr/lib/libXmu.a -> /usr/lpp/tcpip/X11R66/lib/libXmu.a
/usr/lib/libXaw.a -> /usr/lpp/tcpip/X11R66/lib/libXaw.a
/usr/lib/libXp.a -> /usr/lpp/tcpip/X11R66/lib/libXp.a
/usr/lib/libXau.a -> /usr/lpp/tcpip/X11R66/lib/libXau.a
/usr/lib/libXm.a -> /usr/lpp/tcpip/X11R66/lib/libXm.a
/usr/lib/libMrm.a -> /usr/lpp/tcpip/X11R66/lib/libMrm.a
/usr/lib/libUil.a -> /usr/lpp/tcpip/X11R66/lib/libUil.a

```

- New function X11R6.6 and Motif 2.1 31-bit dynamic link libraries (DLLs):


```

/usr/lib/X11_31.dll -> /usr/lpp/tcpip/X11R66/lib/X11_31.dll
/usr/lib/ICE_31.dll -> /usr/lpp/tcpip/X11R66/lib/ICE_31.dll
/usr/lib/SM_31.dll -> /usr/lpp/tcpip/X11R66/lib/SM_31.dll
/usr/lib/Xaw_31.dll -> /usr/lpp/tcpip/X11R66/lib/Xaw_31.dll
/usr/lib/Mrm_31.dll -> /usr/lpp/tcpip/X11R66/lib/Mrm_31.dll
/usr/lib/Uil_31.dll -> /usr/lpp/tcpip/X11R66/lib/Uil_31.dll
/usr/lib/Xm_31.dll -> /usr/lpp/tcpip/X11R66/lib/Xm_31.dll

```

- New function X11R6.6 and Motif 2.1 64-bit dynamic link libraries (DLLs):

```

/usr/lib/X11_64.dll -> /usr/lpp/tcpip/X11R66/lib/X11_64.dll
/usr/lib/ICE_64.dll -> /usr/lpp/tcpip/X11R66/lib/ICE_64.dll
/usr/lib/SM_64.dll -> /usr/lpp/tcpip/X11R66/lib/SM_64.dll
/usr/lib/Xaw_64.dll -> /usr/lpp/tcpip/X11R66/lib/Xaw_64.dll
/usr/lib/Mrm_64.dll -> /usr/lpp/tcpip/X11R66/lib/Mrm_64.dll
/usr/lib/Uil_64.dll -> /usr/lpp/tcpip/X11R66/lib/Uil_64.dll
/usr/lib/Xm_64.dll -> /usr/lpp/tcpip/X11R66/lib/Xm_64.dll

```

- Header files for X11R6.6 and Motif 2.1:

```

/usr/include/X11/ -> /usr/lpp/tcpip/X11R66/include/X11 (header files)
/usr/include/X11/ICE -> /usr/lpp/tcpip/X11R66/include/X11/ICE (ICE specific header files)
/usr/include/X11/SM -> /usr/lpp/tcpip/X11R66/include/X11/SM (SM specific header files)
/usr/include/X11/Xaw -> /usr/lpp/tcpip/X11R66/include/X11/Xaw (Xaw specific header files)
/usr/include/X11/Xmu -> /usr/lpp/tcpip/X11R66/include/X11/Xmu (Xmu specific header files)
/usr/include/X11/extensions -> /usr/lpp/tcpip/X11R66/include/X11/extensions (extensions specific header files)
/usr/include/X11/bitmaps -> /usr/lpp/tcpip/X11R66/include/X11/bitmaps (bitmaps for samples)
/usr/include/Mrm -> /usr/lpp/tcpip/X11R66/include/Mrm (motif header files)
/usr/include/Xm -> /usr/lpp/tcpip/X11R66/include/Xm (motif header files)
/usr/include/X11/uil -> /usr/lpp/tcpip/X11R66/include/uil (Uil header files)

```

- Other utilities and Data files for the new function X11R6.6 and Motif 2.1:

```

/bin/X11/uil -> /usr/lpp/tcpip/bin/X1166/uil (31-bit uil compiler)
/bin/X11/uil64 -> /usr/lpp/tcpip/bin/X1166/uil64 (64-bit uil compiler)
/usr/lib/X11 -> /usr/lpp/tcpip/X11R66/lib/X11
/usr/lib/X11/locale -> /usr/lpp/tcpip/X11R66/lib/X11/locale (locale data files)
/usr/lib/X11/XErrorDB -> /usr/lpp/tcpip/X11R66/lib/X11/XErrorDB (X Error message database)
/usr/lib/X11/XKeysymDB -> /usr/lpp/tcpip/X11R66/lib/X11/XKeysymDB (X keysym Database)
/usr/lib/X11/app-defaults -> /usr/lpp/tcpip/X11R66/lib/X11/app-defaults/ (application default files)

```

- Examples shipped for X11R6.6 and Motif 2.1:

```

/usr/lpp/tcpip/X11R66/Xamples/man/cat1/ (man pages for Xamples programs)
/usr/lpp/tcpip/X11R66/Xamples/demos/ (demonstration programs)
/usr/lpp/tcpip/X11R66/Xamples/clients/ (selected standard clients)
/usr/lpp/tcpip/X11R66/Xamples/motif (selected Motif examples)

```

Chapter 8. Remote procedure calls in the z/OS Communications Server environment

This topic describes the high-level remote procedure calls (RPCs) implemented in TCP/IP including the RPC programming interface to the C language and communication between processes.

The RPC protocol permits remote execution of subroutines across a TCP/IP network. RPC, together with the eXternal Data Representation (XDR) protocol, defines a standard for representing data that is independent of internal protocols or formatting. RPCs can communicate between processes on the same or different hosts.

For more information about the RPC and XDR protocols, see the following:

- Sun Microsystems publication, *Networking on the Sun Workstation: Remote Procedure Call Programming Guide*
- RFC 1831
RPC: Remote Procedure Call Protocol Specification Version 2, R. Srinivasan, August 1995
- RFC 1832
XDR: External Data Representation Standard, R. Srinivasan, August 1995

See Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs.

Tips:

- RPC is supported using the C/370 programming language and the TCP/IP C socket API. For more information about the C/370 socket API, see the *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*.
- For more information about z/OS UNIX System Services sockets, see the *z/OS XL C/C++ Run-Time Library Reference*.

The RPC interface

To use the RPC interface, you must be familiar with programming in the C language, and you should have a working knowledge of networking concepts.

The RPC interface enables programmers to write distributed applications using high-level RPCs rather than lower-level calls based on sockets.

When you use RPCs, the client communicates with a server. The client invokes a procedure to send a call message to the server. When the message arrives, the server calls a dispatch routine, and performs the requested service. The server sends back a reply message, after which the original procedure call returns to the client program with a value derived from the reply message.

See Sample RPC programs, for sample RPC client, server, and raw data stream programs. Figure 2 on page 208 and Figure 3 on page 209 provide an overview of the high-level RPC client and server processes from initialization through cleanup.

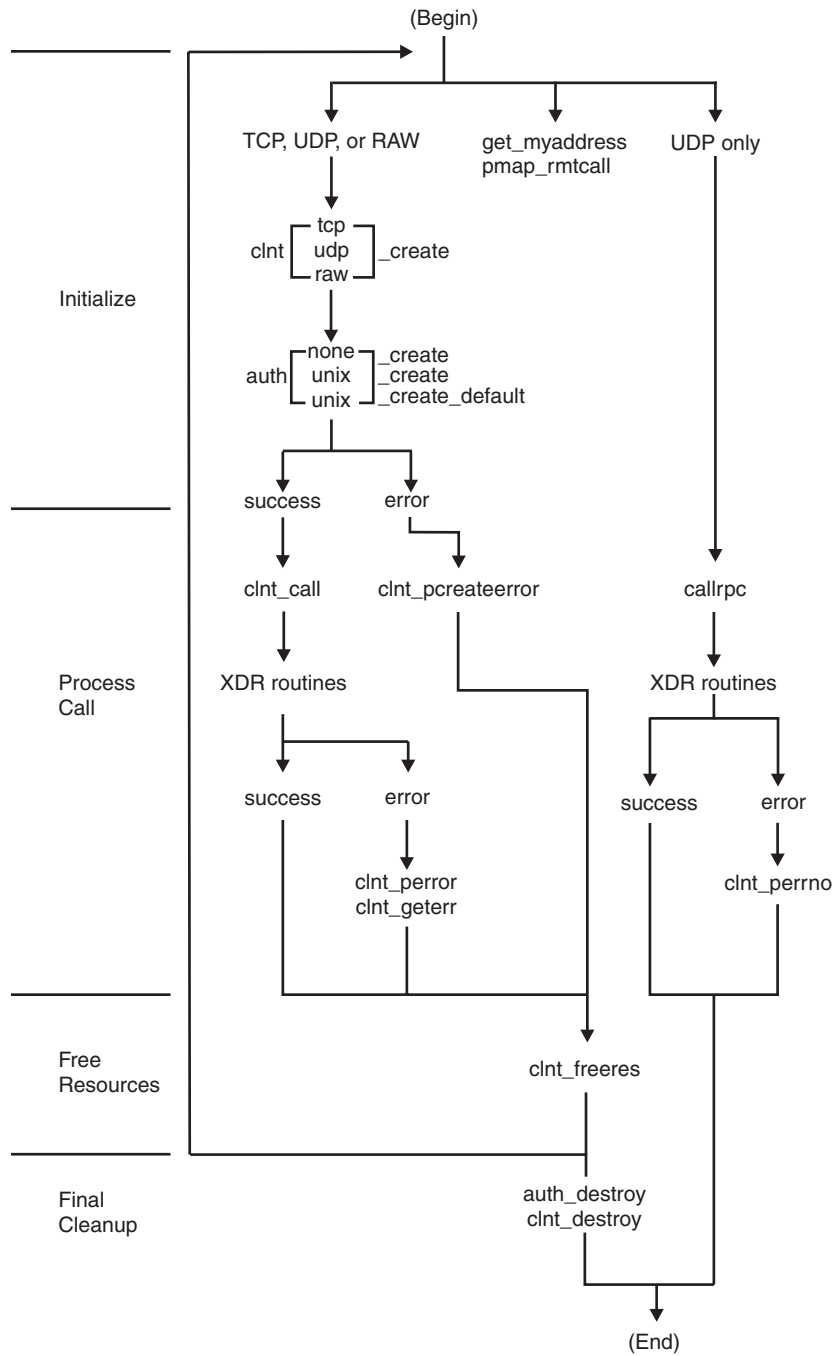


Figure 2. Remote procedure call (client)

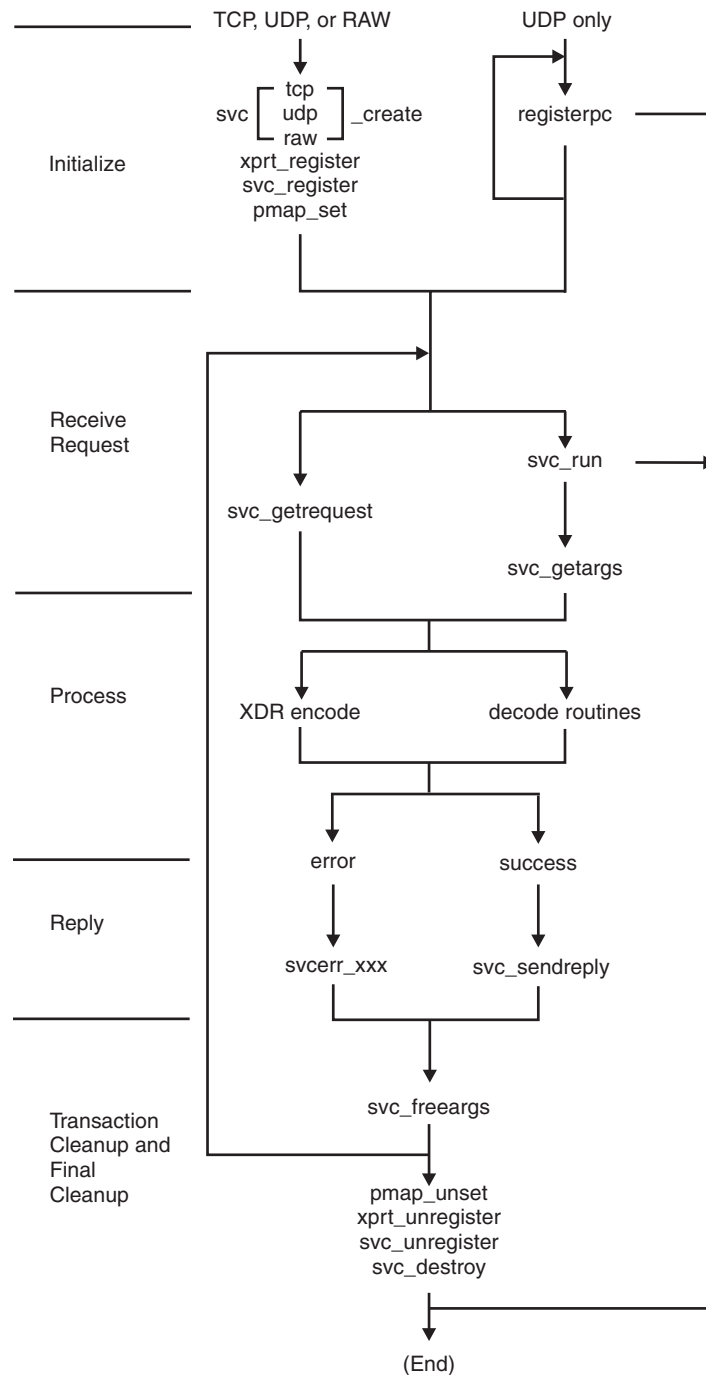


Figure 3. Remote procedure call (server)

Portmapper and rpcbind

Portmapper and rpcbind are the software that supply client programs with information about server programs. Portmapper returns port numbers of server programs and rpcbind returns universal addresses. A universal address is a text string representation of the transport dependent address. A universal address for rpcbind is defined in RFC 3530 as a text string of the IP address, a dot, then the

text string of the two octets of the port number. The following are examples of universal addresses for port 1024 (port 1024 = port 0x400):

- 9.1.1.1.4.0
- ::FFFF:9.1.1.1.4.0
- 2001:0DB8::10:1:1:1.4.0

You can communicate between different computer operating systems when messages are directed to port numbers or universal addresses rather than to targeted remote programs. Clients contact server programs by sending messages to the port numbers or universal addresses of remote processes. Because you make requests to the port number or universal address of a server rather than directly to a server program, client programs need a way to find this information about the server programs they are calling. Portmapper and rpcbind standardize the way clients locate information about the server programs that are supported on a network.

Portmapper and rpcbind use well-known port 111. See Appendix B, “Well-known port assignments,” on page 651, for other well-known TCP and UDP port assignments.

The port-to-program information maintained by portmapper is called the portmap. Clients ask portmapper or rpcbind about entries for servers on the network. Servers contact portmapper or rpcbind to add or update entries to the portmap.

Contacting portmapper or rpcbind

To find the port or universal addresses of a remote program, the client sends an RPC to well-known port 111 of the server’s host. If the server listening on port 111 (rpcbind or portmapper) has an entry for the remote program, it provides the port number or universal addresses in a return RPC. The client then contacts the remote program by sending an RPC to the port number or universal addresses provided.

Clients can save information about recently called remote programs to avoid having to contact portmapper or rpcbind for each request to a server. Some of the RPC function calls automatically contact portmapper or rpcbind on behalf of the client. This eliminates the need for the application code to perform this task.

To see all the servers currently registered with RPC binding protocol Version 2 with portmapper or rpcbind, use the RPCINFO command as follows:

```
RPCINFO -p host_name
```

For details about rpcbind, see the MODIFY command—RPCBIND information in the *z/OS Communications Server: IP System Administrator’s Commands*. For more information about rpcinfo and portmapper, see the Rpcinfo information in the *z/OS Communications Server: IP System Administrator’s Commands*.

Target assistance

Portmapper and rpcbind assist clients in contacting server programs. Either portmapper or rpcbind can be used on the same host, but not both. If the client sends an RPC with the target program number, version number, procedure number, and arguments to the server listening on port 111 (rpcbind or portmapper), that server locates the target server in its list of registered servers and passes the client’s message to the target server. When the target server returns information to portmapper or rpcbind, the information is passed to the client along

with the port number (or universal address, if rpcbind is being used) of the remote program. The client can then contact the server directly.

Registering with rpcbind

This section applies only to RPC servers that register using the binding protocol procedure `RPCBPROC_SET`.

RPC application servers register with rpcbind by sending a `PMAPPROC_SET` or `RPCBPROC_SET` request to the rpcbind server. Servers that register with `RPCBPROC_SET` specify the IP address that the server listens to as part of `RPCBPROC_SET` input.

Example 1

Requirement: If your server registers an IPv4 IP address, you must register the address as an IPv4 address rather than an IPv4-mapped IPv6 address.

The following example assumes that your server is listening on IP address 1.2.3.4 and port 1024 and that the server uses stream sockets. In the `rpcb` specified with the `RPCBPROC_SET` procedure, specify the following `rpcb` field values:

- `r_addr` = 1.2.3.4.0 instead of `::FFFF:1.2.3.4.0`
- `r_netid` = `tcp` instead of `tcp6`

Example 2

If the following conditions apply to your server, you should register your application with both the IPv4 address, `INADDR_ANY`, and the IPv6 unspecified address (`in6addr_any`):

- The server is listening on an `AF_INET6` socket bound to the IPv6 unspecified address (`in6addr_any`)
- The server host has both IPv4 and IPv6 interfaces
- The server will serve IPv4 clients as well as IPv6 clients

This example assumes that your server uses datagram sockets and is listening to an `AF_INET6` socket that is bound to the IPv6 unspecified address (`in6addr_any`) on port 2048. The server host has both IPv4 and IPv6 interfaces and the server intends to accept requests from both IPv4 and IPv6 clients.

Register your application twice. In the `rpcb` that is specified by `RPCBPROC_SET`, specify the following `rpcb` field values on the first registration:

```
r_addr = 0.0.0.0.8.0
r_netid = udp
```

Specify the following `rpcb` field values on the second registration:

```
r_addr = ::0.8.0
r_netid = udp6
```

Restriction: When processing an `RPCBPROC_SET` request, rpcbind ignores the `r_owner` field of the input `rpcb`.

Obtaining address lists from the rpcbind server

RPC binding protocol V4 provides a procedure, `RPCBPROC_GETADDRLIST`, for obtaining a list of addresses supported by a service. When a client queries the z/OS rpcbind server using the UDP protocol over IPv4 or IPv6 transport, the rpcbind server confines the reply to fit within one UDP IPv4 datagram. To obtain all addresses supported by the service, the client should use TCP protocol when

invoking the `RPCBPROC_GETADDRLIST` procedure. For more information on RPC binding protocol V4 and the `RPCBPROC_GETADDRLIST` procedure, see RFC 1833. See Appendix E, “Related protocol specifications,” on page 719 for information on accessing RFCs.

Result: Your client might not be able to reach every address returned by the `RPCBPROC_GETADDRLIST` procedure and possibly might not be able to reach the service at all with the information provided by the `RPCBPROC_GETADDRLIST` procedure. Following are some examples:

- The service might register a specific address that is not reachable from the client.
- If you use UDP to query the `rpcbind` server, the addresses returned within the span of a single datagram might be unreachable by the client.

Restriction: If the service supports private network addresses, `rpcbind` returns those addresses in an `RPCBPROC_GETADDRLIST` reply. If your client resides in the private network with the service, your client can use these addresses to contact the service. However, if the service and the client reside in different private networks, unpredictable results will occur. See RFC 1918 for more information about private network addresses. See Appendix E, “Related protocol specifications,” on page 719 for information on accessing RFCs.

RPC servers in a CINET environment

Tip: This section applies only if your RPC server application registers with `rpcbind`. The portmapper does not recognize new stacks that join a CINET environment, so RPC servers that register with the portmapper are not affected.

The `rpcbind` server recognizes stacks that are started after `rpcbind` itself is started. If your RPC server does not recognize stacks that are started after your server establishes its listening socket, your server will not accept calls from the new stack. This is true of all servers in a CINET environment, not just RPC servers. If an RPC client reaches the `rpcbind` server from a newly started stack to obtain the universal address of your server, it is possible that the client will be unable to contact your server (because your server is not accepting connections from the new stack).

To avoid this problem, do the following:

- Avoid starting a new stack after your RPC server is started.
- Always stop and start your RPC server after starting a new stack.
- Code your server to do the following:
 - Detect a stack starting
 - Deregister your application with `rpcbind`
 - Close your listening socket
 - Establish a new listening socket and ephemeral port
 - Register the new port using `rpcbind`

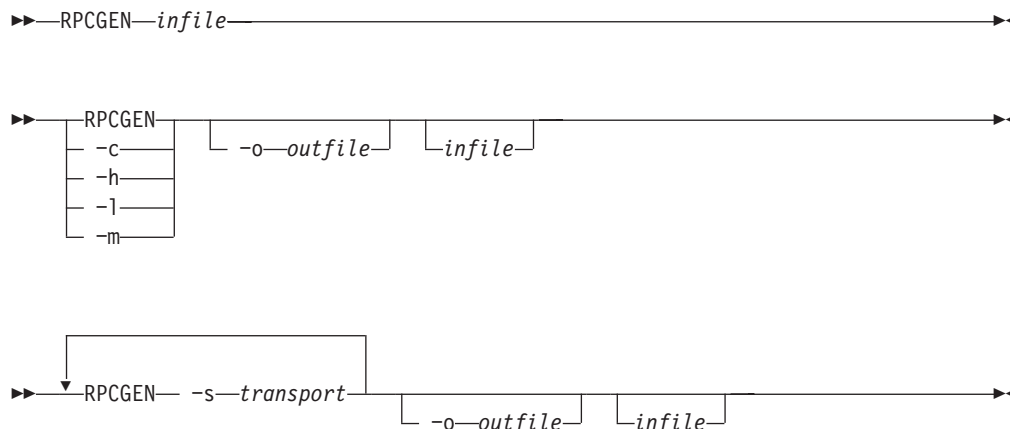
For more information about detecting a stack that is starting, see the `setibmsockopt()` -- Set IBM Specific Options Associated with a Socket information in the *z/OS XL C/C++ Run-Time Library Reference*.

RPCGEN command

Purpose

Use the `RPCGEN` command to generate the code to implement the RPC protocol.

Format



Parameters

- c Compiles into XDR routines.
- h Compiles into C data definitions (a header file).
- l Compiles into client-side stubs.
- m Compiles into server-side stubs without generating a main routine. This option is useful for callback routines and for writing a main routine for initialization.

-o outfile

Specifies the name of the output data set. If none is specified, standard output is used for *-c*, *-h*, *-l*, *-m*, and *-s* modes.

infile

Specifies the name of the input data set written in the RPC language. The default is the data specified by the SYSIN DD statement.

-s transport

Compiles into server-side stubs, using the given transport. TCP and UDP are the supported transports. You can invoke this option more than once to compile a server that serves multiple transports. By default, RPCGEN creates server stubs that support both TCP and UDP.

RPCGEN is a tool that generates C code to implement an RPC protocol. The input to RPCGEN is a language similar to C, known as RPC language.

RPCGEN *infile* is normally used when you want to generate all four of the following output data sets. For example:

- If the *infile* is named `proto.x`, `RPCGEN` generates:
 - A header file called `PROTO.H`
 - XDR routines called `PROTOX.C`
 - Server-side stubs called `PROTOS.C`
 - Client-side stubs called `PROTOS.C`
- If the *infile* is named `USERA.RPC.SOURCE(PROTO)`, `RPCGEN` generates:
 - A header file called `USERA.RPC.H(PROTO)`

- XDR routines called USERA.RPC.C(PROTOX)
- Server-side stubs called USERA.RPC.C(PROTOS)
- Client-side stubs called USERA.RPC.C(PROTOC)

RPCGEN obtains the file names for the C compiler for preprocessing input from the CCRPCGEN CLIST, which must be customized similar to the C installation procedure. For installation using the C/C++ compiler, the following would be an example of the values for the statements in CCRPCGEN that are used by RPCGEN:

```

SET CHD      = &STR(CBC)           /* PREFIX FOR SYSTEM FILES      */
SET CVER     = &STR(OSV2R5)        /* VERSION OF COMPILER          */
SET COMPL    = &STR(SCBCCMP)       /* C COMPILER MODULES           */
SET EDCMSGSG = &STR(SCBCDMSG)      /* C COMPILER MESSAGES          */
SET LANG     = &STR(CBCLMSGSG)     /* MESSAGE LANGUAGE             */
SET SCEEHDRS = &STR(SCEEH)         /* C SYSTEM HEADER FILES        */
SET CMOD     = &STR(CBCDRVR)       /* C COMPILER EXECUTABLE MODULE */
SET WORKDA   = &STR(SYSDA)         /* UNIT TYPE FOR WORK FILES     */
SET WRKSPC   = &STR(1,1)          /* CYLS ALLOCATED FOR WORK FILES */

```

The CCRPCGEN clist must reside in the SYSPROC concatenation.

Notes:

1. A temporary file called PROTO.EXPANDED is created by the RPCGEN command. During normal operation, this file is also subsequently erased by the RPCGEN command.
2. The code generated by RPCGEN is not suitable for input to a C++ compiler.

For more information about the RPCGEN command, see the Sun Microsystems publication, *Network Programming*.

clnt_stat enumerated type

The clnt_stat enumerated type is defined in the CLNT.H file.

RPCs frequently return information in the form of a clnt_stat enumerated value. The following is the format and a description of the clnt_stat enumerated type:

```
enum clnt_stat {
    RPC_SUCCESS=0,          /* call succeeded */
    /*
     * local errors
     */
    RPC_CANTENCODEARGS=1,   /* can't encode arguments */
    RPC_CANTDECODERES=2,    /* can't decode results */
    RPC_CANTSEND=3,         /* failure in sending call */
    RPC_CANTRECV=4,         /* failure in receiving result */
    RPC_TIMEDOUT=5,         /* call timed out */
    /*
     * remote errors
     */
    RPC_VERSIONMISMATCH=6,  /* RPC versions not compatible */
    RPC_AUTHERROR=7,        /* authentication error */
    RPC_PROGUNAVAIL=8,      /* program not available */
    RPC_PROGVERSIONMISMATCH=9, /* program version mismatched */
    RPC_PROCUNAVAIL=10,     /* procedure unavailable */
    RPC_CANTENCODEARGS=11,  /* decode arguments error */
    RPC_SYSTEMERROR=12,     /* generic "other problem" */
    /*
     * callrpc errors
     */
    RPC_UNKNOWNHOST=13,     /* unknown host name */
    /*
     * create errors
     */
    RPC_PMAPFAILURE=14,     /* the pmap failed in its call */
    RPC_PROGNOTREGISTERED=15, /* remote program is not registered */
    /*
     * unspecified error
     */
    RPC_FAILED=16
};
```

Porting

This section contains information about porting RPC applications.

Remapping file names with MANIFEST.H

To conform to the MVS requirement that MVS data set names be eight characters or less in length, a file called MANIFEST.H remaps the RPC long names to eight-character derived names for internal processing.

The MANIFEST.H header file must be the first include file in the application, and it must be present at compile time. If it is not included, the application will fail to link-edit. If the preprocessor macro MVS is defined when the RPC.H file is included, RPC.H will implicitly include MANIFEST.H.

Note: #define Resolve_Via_Lookup must be specified before #include manifest.h to enable the following socket calls: endhostent(), gethostent(), gethostbyaddr(), gethostbyname(), and sethostent().

Accessing system return messages

To access system return values, you need only use the `ERRNO.H` include statement supplied with the compiler. To access network return values, you must add the following include statement:

```
#include <tcperrno.h>
```

Printing system return messages

To print only system errors, use `perror()`, a procedure available in the C compiler run-time library. To print both system and network errors, use `tcperror()`, a procedure included with TCP/IP.

Enumerations

Both `xdr_enum()` and `xdr_union()` are macros to account for varying length enumerations. `xdr_enum()` and `xdr_union` cannot be referenced by `callrpc()`, `svc_freeargs()`, `svc_getargs()`, or `svc_sendreply()`. An XDR routine for the specific enumeration or union must be created. For more information, see “`xdr_enum()`” on page 287.

Header files for remote procedure calls

The following header files are provided with TCP/IP. To compile your program, you must include certain header files; however, not all of them are necessary for every RPC application program.

<code>auth.h</code>	<code>pmap@pro.h</code>
<code>auth@uni.h</code>	<code>rpc.h</code>
<code>bsdtime.h</code>	<code>rpc@msg.h</code>
<code>bsdtocms.h</code>	<code>svc.h</code>
<code>clnt.h</code>	<code>svc@auth.h</code>
<code>in.h</code>	<code>socket.h</code>
<code>inet.h</code>	<code>tcperrno.h</code>
<code>manifest.h</code>	<code>types.h</code>
<code>netdb.h</code>	<code>xdr.h</code>
<code>pmap@cln.h</code>	

Note: When you compile your application program using RPC, you must include the RPC header files before the X Window System include files.

Compiling and linking RPC applications

You can use several methods to compile, link-edit, and execute your TCP/IP C source program in MVS. This section contains information about the data sets that you must include to run your C source program under MVS batch, using IBM-supplied cataloged procedures.

The following data set name is used as an example in the sample JCL statements:

USER.MYPROG.H

Contains user `#include` files.

Compatibility considerations

Unless noted in *z/OS Communications Server: New Function Summary*, an application program compiled and link edited on a release of z/OS Communications Server IP can be used on higher level releases. That is, the API is upward compatible.

Application programs that are compiled and link edited on a release of z/OS Communications Server IP cannot be used on older releases. That is, the API is not downward compatible.

Sample compile cataloged procedure additions

Include the following in the compile step of your cataloged procedure. Cataloged procedures are included in the IBM-supplied samples for your MVS system.

- Add the following statement as the first //SYSLIB DD statement.

```
//SYSLIB DD DSN=SEZACMAC,DISP=SHR
```

- Add the following //USERLIB DD statement.

```
//USERLIB DD DSN=USER.MYPROG.H,DISP=SHR
```

Nonreentrant modules

To compile and link nonreentrant RPC applications, the procedure is similar to the procedure for nonreentrant C applications as described in the section on nonreentrant modules in the *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*.

One additional JCL statement is needed. Add the following SYSLIB statement after SEZACMTX statement in the link step:

```
// DD DSN=SEZARPCL,DISP=SHR
```

Reentrant modules

To compile and link reentrant RPC applications, the procedure is similar to the procedure for reentrant C applications as described in the section on reentrant modules in the *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*.

One additional JCL statement is needed. Add the following SYSLIB statement after the SEZARNT1 statement in the prelink-edit step:

```
// DD DSN=SEZARNT4,DISP=SHR
```

RPC global variables

These sections describe the three RPC global variables, *rpc_createerr*, *svc_fds*, and *svc_fdset*.

rpc_createerr

Format

```
#include <rpc.h>

struct rpc_createerr rpc_createerr;
```

Usage

rpc_createerr is a global variable that is set when any RPC client creation routine fails. Use clnt_pcreateerror() to print the message.

Context

- clntraw_create();
- clnttcp_create()
- clntudp_create()

svc_fds

Format

```
#include <rpc.h>
int svc_fds;
```

Usage

svc_fds is a global variable that specifies the read descriptor bit set on the service machine. This is of interest only if the service programmer decides to write an asynchronous event processing routine; otherwise svc_run() should be used. Writing asynchronous routines in the MVS environment is not simple, because there is no direct relationship between the descriptors used by the socket routines and the event control blocks commonly used by MVS programs for coordinating concurrent activities.

Rule: Do not modify this variable.

Context

- svc_getreq()

svc_fdset

Format

```
#include <rpc.h>

fd_set svc_fdset;
```

Usage

svc_fdset is a global variable that specifies the read descriptor bit set on the service machine. This is of interest only if the service programmer decides to write an asynchronous event processing routine; otherwise svc_run() should be used. Writing asynchronous routines in the MVS environment is not simple, because there is no direct relationship between the descriptors used by the socket routines and the event control blocks commonly used by MVS programs for coordinating concurrent activities.

Rule: Do not modify this variable.

Context

- svc_getreqset()

Remote procedure and external data representation calls

These sections provide the syntax, parameters, and other appropriate information for each remote procedure and external data representation call supported by z/OS Communications Server.

auth_destroy()

Format

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

Parameters

auth
Indicates a pointer to authentication information.

Usage

The `auth_destroy()` call deletes the authentication information for *auth*. Once this procedure is called, *auth* is undefined.

Context

- `authnone_create()`
- `authunix_create()`
- `authunix_create_default()`

authnone_create()

Format

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

Parameters

None.

Usage

The `authnone_create()` call creates and returns an RPC authentication handle. The handle passes the NULL authentication on each call.

Context

- `auth_destroy()`
- `authunix_create()`
- `authunix_create_default()`

authunix_create()

Format

```
#include <rpc.h>
AUTH *
authunix_create(host, uid, gid, len, aup_gids)
char *host;
int uid;
int gid;
int len;
int *aup_gids;
```

Parameters

host

Specifies a pointer to the symbolic name of the host where the desired server is located.

uid

Specifies the user's user ID.

gid

Specifies the user's group ID.

len

Indicates the length of the information pointed to by *aup_gids*.

aup_gids

Specifies a pointer to an array of groups to which the user belongs.

Usage

The `authunix_create()` call creates and returns an authentication handle that contains UNIX-based authentication information.

Context

- `auth_destroy()`
- `authnone_create()`
- `authunix_create_default()`

authunix_create_default()

Format

```
#include <rpc.h>

AUTH *
authunix_create_default()
```

Parameters

None

Usage

The `authunix_create_default()` call invokes `authunix_create()` with default parameters.

Context

- `auth_destroy()`
- `authnone_create()`
- `authunix_create()`

callrpc()

Format

```
#include <rpc.h>

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

Parameters

host

Specifies a pointer to the symbolic name of the host where the desired server is located.

prognum

Identifies the program number of the remote procedure.

versnum

Identifies the version number of the remote procedure.

procnum

Identifies the procedure number of the remote procedure.

inproc

Specifies the XDR procedure used to encode the arguments of the remote procedure.

in

Specifies a pointer to the arguments of the remote procedure.

outproc

Specifies the XDR procedure used to decode the results of the remote procedure.

out

Specifies a pointer to the results of the remote procedure.

Usage

The `callrpc()` calls the remote procedure described by *prognum*, *versnum*, and *procnum* running on the *host* system. `callrpc()` encodes and decodes the parameters for transfer.

Notes:

1. `clnt_perrno()` can be used to translate the return code into messages.
2. `callrpc()` cannot call the procedure `xdr_enum`. See “`xdr_enum()` ” on page 287 for more information.
3. This procedure uses UDP as its transport layer. See “`clntudp_create()`” on page 246 for more information.

Return codes

A value of `RPC_SUCCESS` (0) indicates success; otherwise, an error has occurred as indicated by the value returned. The results of the remote procedure call are returned to *out*.

Context

- `clnt_broadcast()`
- `clnt_call()`
- `clnt_perrno()`
- `clntudp_create()`
- `clnt_sperno()`
- `clnt_sperno()`
- `xdr_enum()`

clnt_broadcast()

Format

```
#include <rpc.h>
enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)
u_long prognum;
u_long versnum;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
resultproc_t eachresult;
```

Parameters

prognum

Identifies the program number of the remote procedure.

versnum

Identifies the version number of the remote procedure.

procnum

Identifies the procedure number of the remote procedure.

inproc

Identifies the XDR procedure used to encode the arguments of the remote procedure.

in

Specifies a pointer to the arguments of the remote procedure.

outproc

Specifies the XDR procedure used to decode the results of the remote procedure.

out

Specifies a pointer to the results of the remote procedure; however, the output of the remote procedure is decoded.

eachresult

Specifies the procedure called after each response.

Note: resultproc_t is a type definition.

```
#include <rpc.h>
typedef bool_t (*resultproc_t)
();
```

addr

Specifies the pointer to the address of the machine that sent the results.

Usage

The clnt_broadcast() call broadcasts the remote procedure described by *prognum*, *versnum*, and *procnum* to all locally connected broadcast networks. Each time clnt_broadcast() receives a response it calls eachresult(). The format of eachresult() is:

Format

```
#include <rpc.h>
bool_t eachresult(out, addr)
char *out;
struct sockaddr_in *addr;
```

Return codes

If eachresult() returns 0, clnt_broadcast() waits for more replies; otherwise, eachresult() returns the appropriate status.

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

Context

- callpc()
- clnt_call()

clnt_call()

Format

```
#include <rpc.h>

enum clnt_stat
clnt_call(clnt, procnum,
inproc, in, outproc, out, tout)
CLIENT *clnt;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
```

Parameters

clnt

Specifies the pointer to a client handle that was previously obtained using `clntraw_create()`, `clnttcp_create()`, or `clntudp_create()`.

procnum

Identifies the remote procedure number.

inproc

Specifies the XDR procedure used to encode *procnum* arguments.

in

Specifies a pointer to the arguments of the remote procedure.

outproc

Indicates the XDR procedure used to decode the remote procedure results.

out

Specifies a pointer to the results of the remote procedure.

tout

Indicates the time allowed for the server to respond.

Usage

The `clnt_call()` calls the remote procedure (*procnum*) associated with the client handle (*clnt*).

Return codes

A value of `RPC_SUCCESS` (0) indicates success; otherwise, an error has occurred as indicated by the value returned. The results of the remote procedure call are returned in *out*.

Context

- `callrpc()`
- `clnt_broadcast()`
- `clnt_geterr()`
- `clnt_perror()`
- `clnt_sperror()`
- `clntraw_create()`

- `clnttcp_create()`
- `clntudp_create()`

clnt_control()

Format

```
#include <rpc.h>

bool_t
clnt_control(clnt, request, info)
CLIENT *clnt;
int request;
void *info;
```

Parameters

clnt

Indicates the pointer to a client handle that was previously obtained using `clntraw_create()`, `clnttcp_create()`, or `clntudp_create()`.

request

Determines the operation (either `CLSET_TIMEOUT`, `CLGET_TIMEOUT`, `CLGET_SERVER_ADDR`, `CLSET_RETRY_TIMEOUT`, or `CLGET_RETRY_TIMEOUT`).

info

Indicates the pointer to information used by the request.

Usage

The `clnt_control()` call performs one of the following control operations:

- Control operations that apply to both UDP and TCP transports:

CLSET_TIMEOUT

Sets timeout (*info* points to the `timeval` structure).

CLGET_TIMEOUT

Gets timeout (*info* points to the `timeval` structure).

CLGET_SERVER_ADDR

Gets server's address (*info* points to the `sockaddr_in` structure).

- UDP only control operations:

CLSET_RETRY_TIMEOUT

Sets retry timeout (*info* points to the `timeval` structure).

CLGET_RETRY_TIMEOUT

Gets retry timeout (*info* points to the `timeval` structure). If you set the timeout using `clnt_control()`, the timeout parameter to `clnt_call()` is ignored in all future calls.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- `clnt_create()`
- `clnt_destroy()`
- `clntraw_create()`
- `clnttcp_create()`
- `clntudp_create()`

clnt_create()

Format

```
#include <rpc.h>
CLIENT *
clnt_create(host, prognum, versnum, protocol)
char *host;
u_long prognum;
u_long versnum;
char *protocol;
```

Parameters

host

Indicates the pointer to the name of the host at which the remote program resides.

prognum

Specifies the remote program number.

versnum

Specifies the version number of the remote program.

protocol

Indicates the pointer to the protocol, which can be either tcp or udp.

Usage

The `clnt_create()` call creates an RPC client transport handle for the remote program specified by (*prognum*, *versnum*). The client uses the specified protocol as the transport layer. Default timeouts are set, but they can be modified using `clnt_control()`.

Return codes

NULL indicates failure.

Context

- `clnt_control()`
- `clnt_destroy()`
- `clnt_pcreateerror()`
- `clnt_spccreateerror()`
- `clnt_sperror()`
- `clntraw_create()`
- `clnttcp_create()`
- `clntudp_create()`

clnt_destroy()

Format

```
#include <rpc.h>
void
clnt_destroy(clnt)
CLIENT *clnt;
```

Parameters

clnt

Specifies the pointer to a client handle that was previously created using `clntudp_create()`, `clnttcp_create()`, or `clntraw_create()`.

Usage

The `clnt_destroy()` call deletes a client RPC transport handle. This procedure involves the deallocation of private data resources, including *clnt*. Once this procedure is used, *clnt* is undefined. If the RPC library opened the associated sockets, it also closes them. Otherwise, the sockets remain open.

Context

- `clnt_control()`
- `clnt_create()`
- `clntraw_create()`
- `clnttcp_create()`
- `clntudp_create()`

clnt_freeres()

Format

```
#include <rpc.h>
bool_t
clnt_freeres(clnt, outproc, out)
CLIENT *clnt;
xdrproc_t outproc;
char *out;
```

Parameters

clnt

Indicates the pointer to a client handle that was previously obtained using `clnt_create()`, `clntraw_create()`, `clnttcp_create()`, or `clntudp_create()`.

outproc

Specifies the XDR procedure used to decode the remote procedure's results.

out

Specifies the pointer to the results of the remote procedure.

Usage

The `clnt_freeres()` call deallocates any resources that were assigned by the system to decode the results of an RPC.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- `clnt_create()`
- `clntraw_create()`
- `clnttcp_create()`
- `clntudp_create()`

clnt_geterr()

Format

```
#include <rpc.h>
void
clnt_geterr(clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;
```

Parameters

clnt

Indicates the pointer to a client handle that was previously obtained using `clnt_create()`, `clntraw_create()`, `clnttcp_create()`, or `clntudp_create()`.

errp

Indicates the pointer to the address into which the error structure is copied.

Usage

The `clnt_geterr()` call copies the error structure from the client handle to the structure at address *errp*.

Context

- `clnt_call()`
- `clnt_create()`
- `clnt_pcreateerror()`
- `clnt_perrno()`
- `clnt_perror()`
- `clnt_spcreateerror()`
- `clnt_sperrno()`
- `clnt_sperror()`
- `clntraw_create()`
- `clnttcp_create()`
- `clntudp_create()`

clnt_pcreateerror()

Format

```
#include <rpc.h>

void
clnt_pcreateerror(s)
char *s;
```

Parameters

- s** Indicates a null or null-terminated character string. If *s* is nonnull, `clnt_pcreateerror()` prints the string *s* followed by a colon, followed by a space, followed by the error message, and terminated with a new line. If *s* is null or points to a null string, just the error message and the new line are output.

Usage

The `clnt_pcreateerror()` call writes a message to the standard error device, indicating why a client handle cannot be created. This procedure is used after `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`, or `clnt_create()`, fails.

Context

- `clnt_create()`
- `clnt_geterr()`
- `clnt_perrno()`
- `clnt_perror()`
- `clnt_spcreateerror()`
- `clnt_sperrno()`
- `clnt_sperror()`
- `clntraw_create()`
- `clnttcp_create()`
- `clntudp_create()`

clnt_perrno()

Format

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

Parameters

stat

Indicates the client status.

Usage

The `clnt_perrno()` call writes a message to the standard error device corresponding to the condition indicated by *stat*. This procedure should be used after `callrpc()` if there is an error.

Context

- `callrpc()`
- `clnt_geterr()`
- `clnt_pcreateerror()`
- `clnt_perror()`
- `clnt_spcreateerror()`
- `clnt_sperrno()`
- `clnt_sperror()`

clnt_perror()

Format

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

Parameters

clnt

Specifies the pointer to a client handle that was previously obtained using `clnt_create()`, `clntudp_create()`, `clnttcp_create()`, or `clntraw_create()`.

- s** Indicates a null or null-terminated character string. If *s* is nonnull, `clnt_perrorerror()` prints the string *s* followed by a colon, followed by a space, followed by the error message, and terminated with a new line. If *s* is null or points to a null string, just the error message and the new line are output.

Usage

The `clnt_perror()` call writes a message to the standard error device, indicating why an RPC failed. This procedure should be used after `clnt_call()` if there is an error.

Context

- `clnt_call()`
- `clnt_create()`
- `clnt_geterr()`
- `clnt_pcreateerror()`
- `clnt_perrno()`
- `clnt_spcreateerror()`
- `clnt_sperrno()`
- `clnt_sperror()`
- `clntraw_create()`
- `clnttcp_create()`
- `clntudp_create()`

clnt_spcreateerror()

Format

```
#include <rpc.h>

char *
clnt_spcreateerror(s)
char *s;
```

Parameters

- s** Indicates a null or null-terminated character string. If *s* is nonnull, `clnt_spcreateerror()` prints the string *s* followed by a colon, followed by a space, followed by the error message, and terminated with a new line. If *s* is null or points to a null string, just the error message and the new line are output.

Usage

The `clnt_spcreateerror()` call returns the address of a message indicating why a client handle cannot be created. This procedure is used after `clnt_create()`, `clntraw_create()`, `clnttcp_create()`, or `clntudp_create()` fails.

Return codes

Pointer to a character string ending with a new line.

Context

- `callrpc()`
- `clnt_geterr()`
- `clnt_perrno()`
- `clnt_perror()`
- `clnt_pcreateerror()`
- `clnt_sperrno()`
- `clnt_sperror()`
- `clntraw_create()`
- `clnttcp_create()`
- `clntudp_create()`

clnt_sperrno()

Format

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

Parameters

stat
Indicates the client status.

Usage

The `clnt_sperrno()` call returns the address of a message corresponding to the condition indicated by *stat*. This procedure should be used after `callrpc()`, if there is an error.

Return codes

Pointer to a character string ending with a new line.

Context

- `clnt_call()`
- `clnt_geterr()`
- `clnt_pcreateerror()`
- `clnt_spccreateerror()`
- `clnt_sperror()`
- `clnt_perrno()`
- `clnt_perror()`

clnt_sperror()

Format

```
#include <rpc.h>

char *
clnt_sperror(clnt, s)
CLIENT *clnt;
char *s;
```

Parameters

clnt

Indicates the pointer to a client handle that was previously obtained using `clnt_create()`, `clntudp_create()`, `clnttcp_create()`, or `clntraw_create()`.

- s* Indicates a null or null-terminated character string. If *s* is nonnull, `clnt_sperror()` prints the string *s* followed by a colon, followed by a space, followed by the error message, and terminated with a new line. If *s* is null or points to a null string, just the error message and the new line are output.

Usage

The `clnt_sperror()` call returns the address of a message indicating why an RPC failed. This procedure should be used after `clnt_call()`, if there is an error.

Return codes

Pointer to a character string ending with a new line.

Context

- `clnt_call()`
- `clnt_create()`
- `clnt_geterr()`
- `clnt_pcreateerror()`
- `clnt_perrno()`
- `clnt_perror()`
- `clnt_spccreateerror()`
- `clnt_sperrno()`
- `clntraw_create()`
- `clnttcp_create()`
- `clntudp_create()`

clntraw_create()

Format

```
#include <rpc.h>
CLIENT *
clntraw_create(prognum, versnum)
u_long prognum;
u_long versnum;
```

Parameters

prognum

Specifies the remote program number.

versnum

Specifies the version number of the remote program.

Usage

The `clntraw_create()` call creates a dummy client for the remote double (*prognum*, *versnum*). Because messages are passed using a buffer within the address space of the local process, the server should also use the same address space, which simulates RPC programs within one address space. See “`svccraw_create()`” on page 274 for more information.

Return codes

NULL indicates failure.

Context

- `clnt_call()`
- `clnt_create()`
- `clnt_destroy()`
- `clnt_freeres()`
- `clnt_geterr()`
- `clnt_pcreateerror()`
- `clnt_perror()`
- `clnt_screateerror()`
- `clnt_sperror()`
- `clntudp_create()`
- `clnttcp_create()`
- `svccraw_create()`

clnttcp_create()

Format

```
#include <rpc.h>
CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
int *sockp;
u_int sendsz;
u_int recvsz;
```

Parameters

addr

Indicates the pointer to the Internet address of the remote program. If *addr* points to a port number of 0, *addr* is set to the port on which the remote program is receiving.

prognum

Specifies the remote program number.

versnum

Specifies the version number of the remote program.

sockp

Indicates the pointer to the socket. If **sockp* is `RPC_ANYSOCK`, then this routine opens a new socket and sets **sockp*.

sendsz

Specifies the size of the send buffer. Specify 0 to choose the default.

recvsz

Specifies the size of the receive buffer. Specify 0 to choose the default.

Usage

The `clnttcp_create()` call creates an RPC client transport handle for the remote program specified by (*prognum*, *versnum*). The client uses TCP as the transport layer.

Return codes

NULL indicates failure.

Context

- `clnt_call()`
- `clnt_control()`
- `clnt_create()`
- `clnt_destroy()`
- `clnt_freeres()`
- `clnt_geterr()`
- `clnt_pcreateerror()`
- `clnt_perror()`
- `clnt_spcreateerror()`

- `clnt_sperror()`
- `clntraw_create()`
- `clntudp_create()`

clntudp_create()

Format

```
#include <rpc.h>
CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
struct timeval wait;
int *sockp;
```

Parameters

addr

Indicates the pointer to the Internet address of the remote program. If *addr* points to a port number of 0, *addr* is set to the port on which the remote program is receiving. The remote portmap service is used for this.

prognum

Specifies the remote program number.

versnum

Specifies the version number of the remote program.

wait

Indicates that UDP resends the call request at intervals of *wait* time, until either a response is received or the call times out. The timeout length is set using the `clnt_call()` procedure.

sockp

Specifies the pointer to the socket. If **sockp* is `RPC_ANYSOCK`, this routine opens a new socket and sets **sockp*.

Usage

The `clntudp_create()` call creates a client transport handle for the remote program (*prognum*) with version (*versnum*). UDP is used as the transport layer.

Note: This procedure should not be used with procedures that use large arguments or return large results. While UDP packet size is configurable to a maximum of 64 - 1 kilobytes, the default UDP packet size is only 8 kilobytes.

Return codes

NULL indicates failure.

Context

- `call_rpc()`
- `clnt_call()`
- `clnt_control()`
- `clnt_create()`
- `clnt_destroy()`
- `clnt_freeres()`
- `clnt_geterr()`

- `clnt_pcreateerror()`
- `clnt_perror()`
- `clnt_spccreateerror()`
- `clnt_sperror()`
- `clntraw_create()`
- `clnttcp_create()`

get_myaddress()

Format

```
#include <rpc.h>
void
get_myaddress(addr)
struct sockaddr_in *addr;
```

Parameters

addr

Indicates the pointer to the location where the local Internet address is placed.

Usage

The `get_myaddress()` call puts the local host Internet address into *addr*. The port number (*addr* → *sin_port*) is set to `htons (PMAPPORT)`, which is 111.

Context

- `clnttcp_create()`
- `getpcport()`
- `pmap_getmaps()`
- `pmap_getport()`
- `pmap_rmtcall()`
- `pmap_set()`
- `pmap_unset()`

getrpcport()

Format

```
#include <rpc.h>
u_short
getrpcport(host, prognum, versnum, protocol)
char *host;
u_long prognum;
u_long versnum;
int protocol;
```

Parameters

host

Specifies the pointer to the name of the foreign host.

prognum

Specifies the program number to be mapped.

versnum

Specifies the version number of the program to be mapped.

protocol

Specifies the transport protocol used by the program (IPPROTO_TCP or IPPROTO_UDP).

Usage

The `getrpcport()` call returns the port number associated with the remote program (*prognum*), the version (*versnum*), and the transport protocol (*protocol*).

Return codes

The value 1 indicates that the mapping does not exist or that the remote portmap could not be contacted. If portmapper cannot be contacted, `rpc_createerr` contains the RPC status.

Context

- `get_myaddress()`
- `pmap_getmaps()`
- `pmap_getport()`
- `pmap_rmtcall()`
- `pmap_set()`
- `pmap_unset()`

pmap_getmaps()

Format

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

struct pmaplist *
pmap_getmaps(addr)
struct sockaddr_in *addr;
```

Parameters

addr

Indicates the pointer to the Internet address of the foreign host.

Usage

The pmap_getmaps() call returns a list of current program-to-port mappings on the foreign host specified by *addr*.

Return codes

Returns a pointer to a pmaplist structure, or NULL.

Context

- getrpcport()
- pmap_getport()
- pmap_rmtcall()
- pmap_set()
- pmap_unset()

pmap_getport()

Format

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>
u_short
pmap_getport(addr, prognum,
versnum, protocol)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
int protocol;
```

Parameters

addr

Indicates the pointer to the Internet address of the foreign host.

prognum

Specifies the program number to be mapped.

versnum

Specifies the version number of the program to be mapped.

protocol

Indicates the transport protocol used by the program (IPPROTO_TCP or IPPROTO_UDP).

Usage

The pmap_getport() call returns the port number associated with the remote program (*prognum*), the version (*versnum*), and the transport protocol (*protocol*).

Return codes

The value 1 indicates that the mapping does not exist or that the remote portmap could not be contacted. If portmapper cannot be contacted, rpc_createerr contains the RPC status.

Context

- getrpcport()
- pmap_getmaps()
- pmap_rmtcall()
- pmap_set()
- pmap_unset()

pmap_rmtcall()

Format

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>
enum clnt_stat
pmap_rmtcall(addr, prognum,
versnum, procnum, inproc, in, outproc, out, tout, portp)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
u_long *portp;
```

Parameters

addr

Indicates the pointer to the Internet address of the foreign host.

prognum

Specifies the remote program number.

versnum

Specifies the version number of the remote program.

procnum

Identifies the procedure to be called.

inproc

Specifies the XDR procedure used to encode the arguments of the remote procedure.

in

Specifies the pointer to the arguments of the remote procedure.

outproc

Specifies the XDR procedure used to decode the results of the remote procedure.

out

Indicates the pointer to the results of the remote procedure.

tout

Specifies the timeout period for the remote request.

portp

If the call from the remote portmap service is successful, *portp* contains the port number of the triple (*prognum*, *versnum*, *procnum*).

Usage

The `pmap_rmtcall()` call instructs portmapper, on the host at *addr*, to make an RPC call to a procedure on that host. This procedure should be used only for ping-type functions.

Return codes

clnt_stat enumerated type.

Context

- getrpcport()
- pmap_getmaps()
- pmap_getport()
- pmap_set()
- pmap_unset()

pmap_set()

Format

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

bool_t
pmap_set(prognum, versnum,
        protocol, port)
u_long prognum;
u_long versnum;
int protocol;
u_short port;
```

Parameters

prognum

Specifies the local program number.

versnum

Specifies the version number of the local program.

protocol

Indicates the transport protocol used by the local program.

port

Indicates the port to which the local program is mapped.

Usage

The pmap_set() call sets the mapping of the program (specified by *prognum*, *versnum*, and *protocol*) to *port* on the local machine. This procedure is automatically called by the svc_register() procedure.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- getrpcport()
- pmap_getmaps()
- pmap_getport()
- pmap_rmtcall()
- pmap_unset()

pmap_unset()

Format

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

bool_t
pmap_unset(prognum, versnum)
u_long prognum;
u_long versnum;
```

Parameters

prognum

Specifies the local program number.

versnum

Specifies the version number of the local program.

Usage

The pmap_unset() call removes the mappings associated with *prognum* and *versnum* on the local machine. All ports for each transport protocol currently mapping the *prognum* and *versnum* are removed from the portmap service.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- getrpcport()
- pmap_getmaps()
- pmap_getport()
- pmap_rmtcall()
- pmap_set()

registerrpc()

Format

```
#include <rpc.h>
int
registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
u_long prognum;
u_long versnum;
u_long procnum;
char *(*procname) ();
xdrproc_t inproc;
xdrproc_t outproc;
```

Parameters

prognum

Specifies the program number to register.

versnum

Specifies the version number to register.

procnum

Specifies the procedure number to register.

procname

Indicates the procedure that is called when the registered program is requested. *procname* must accept a pointer to its arguments, and return a static pointer to its results.

inproc

Specifies the XDR routine used to decode the arguments.

outproc

Specifies the XDR routine that encodes the results.

Usage

The `registerrpc()` call registers a procedure (*prognum*, *versnum*, *procnum*) with the local portmapper, and creates a control structure to remember the server procedure and its XDR routine. The control structure is used by `svc_run()`. When a request arrives for the program (*prognum*, *versnum*, *procnum*), the procedure *procname* is called. Procedures registered using `registerrpc()` are accessed using the UDP transport layer.

Note: `xdr_enum()` cannot be used as an argument to `registerrpc()`. See “`xdr_enum()`” on page 287 for more information.

Return codes

The value 1 indicates success; the value -1 indicates an error.

Context

- `svc_register()`
- `svc_run()`

svc_destroy()

Format

```
#include <rpc.h>
void
svc_destroy(xprt)
SVCXPRT *xprt;
```

Parameters

xprt

Specifies the pointer to the service transport handle.

Usage

The `svc_destroy()` call deletes the RPC service transport handle *xprt*, which becomes undefined after this routine is called.

Context

- `svccraw_create()`
- `svctcp_create()`
- `svcudp_create()`

svc_freeargs()

Format

```
#include <rpc.h>
bool_t
svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Parameters

xprt
Specifies the pointer to the service transport handle.

inproc
Specifies the XDR routine used to decode the arguments.

in
Indicates the pointer to the input arguments.

Usage

The `svc_freeargs()` call frees storage allocated to decode the arguments to a service procedure using `svc_getargs()`.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- `svc_getargs()`

svc_getargs()

Format

```
#include <rpc.h>
bool_t
svc_getargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Parameters

xprt
Specifies the pointer to the service transport handle.

inproc
Specifies the XDR routine used to decode the arguments.

in
Indicates the pointer to the decoded arguments.

Usage

The `svc_getargs()` call uses the XDR routine *inproc* to decode the arguments of an RPC request associated with the RPC service transport handle *xprt*. The results are placed at address *in*.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- `svc_freeargs()`

svc_getcaller()

Format

```
#include <rpc.h>
struct sockaddr_in *
svc_getcaller(xprt)
SVCXPRT *xprt;
```

Parameters

xprt
Specifies the pointer to the service transport handle.

Usage

Macro obtains the network address of the client associated with the service transport handle *xprt*.

Return codes

This is a pointer to a `sockaddr_in` structure.

Context

- `get_myaddress()`

svc_getreq()

Format

```
#include <rpc.h>
void
svc_getreq(rdfds)
int rdfds;
```

Parameters

rdfds

Specifies the read descriptor bit set.

Usage

The `svc_getreq()` call is used, rather than `svc_run()`, to implement asynchronous event processing. The routine returns control to the program when all sockets have been serviced.

`svc_getreq` limits you to 32 socket descriptors, of which 3 are reserved. Use `svc_getreqset` if you have more than 29 socket descriptors.

Context

- `svc_run()`

svc_getreqset()

Format

```
#include <rpc.h>
void
svc_getreqset(readfds)
fd_set readfds;
```

Parameters

readfds

Specifies the read descriptor bit set.

Usage

The `svc_getreqset()` call is used, rather than `svc_run()`, to implement asynchronous event processing. The routine returns control to the program when all sockets have been serviced.

A server would use a `select()` call to determine if there are any outstanding RPC requests at any of the sockets created when the programs were registered. The read bit descriptor set returned by `select()` is then used on the call to `svc_getreqset()`.

Note that you should not pass the global bit descriptor set `svc_fdset` on the call to `select()`, because `select()` changes the values. Instead, you should make a copy of `svc_fdset` before you call `select()`.

Context

- `svc_run()`

svc_register()

Format

```
#include <rpc.h>
bool_t
svc_register(xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
u_long prognum;
u_long versnum;
void (*dispatch) ();
int protocol;
```

Parameters

xprt

Specifies the pointer to the service transport handle.

prognum

Specifies the program number to be registered.

versnum

Specifies the version number of the program to be registered.

dispatch()

Indicates the dispatch routine associated with prognum and versnum.

The structure of the dispatch routine is:

```
#include <rpc.h>

dispatch(request, xprt)
struct svc_req *request;
SVCXPRT *xprt;
```

protocol

The protocol used. The value is generally one of the following:

- 0
- IPPROTO_UDP
- IPPROTO_TCP

When a value of 0 is used, the service is not registered with portmapper.

Rule: When using a toy RPC service transport created with `svccraw_create()`, a call to `xprt_register()` must be made immediately after a call to `svc_register()`.

Usage

The `svc_register()` call associates the program described by (prognum, versnum) with the service dispatch routine dispatch.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- `registerrpc()`
- `svc_unregister()`
- `xprt_register()`

svc_run()

Format

```
#include <rpc.h>
svc_run()
```

Parameters

None.

Usage

The `svc_run()` call does not return control. It accepts RPC requests and calls the appropriate service using `svc_getreqset()`.

Context

`svc_getreqset()`

svc_sendreply()

Format

```
#include <rpc.h>
bool_t
svc_sendreply(xprt, outproc, out)
SVCXPRT *xprt;
xdrproc_t outproc;
char *out;
```

Parameters

xprt

Indicates the pointer to the caller's transport handle.

outproc

Specifies the XDR procedure used to encode the results.

out

Specifies the pointer to the results.

Usage

The svc_sendreply() call is called by the service dispatch routine to send the results of the call to the caller.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_call()

svc_unregister()

Format

```
#include <rpc.h>
void
svc_unregister(prognum, versnum)
u_long prognum;
u_long versnum;
```

Parameters

prognum

Specifies the program number that is removed.

versnum

Specifies the version number of the program that is removed.

Usage

The svc_unregister() call removes all local mappings of prognum and versnum to dispatch routines and prognum, versnum, and * to port numbers.

svcerr_auth()

Format

```
#include <rpc.h>
void
svcerr_auth(xprt, why)
SVCXPRT *xprt;
enum auth_stat why;
```

Parameters

xprt

Specifies the pointer to the service transport handle.

why

Specifies the reason the call is refused.

Usage

The `svcerr_auth()` call is called by a service dispatch routine that refuses to execute an RPC request because of authentication errors.

Context

- `svcerr_noproc()`
- `svcerr_noprogram()`
- `svcerr_progvers()`
- `svcerr_systemerr()`
- `svcerr_weakauth()`

svcerr_decode()

Format

```
#include <rpc.h>
void
svcerr_decode(xprt)
SVCXPRT *xprt;
```

Parameters

xprt

Indicates the pointer to the service transport handle.

Usage

The `svcerr_decode()` call is called by a service dispatch routine that cannot decode its parameters.

Context

- `svcerr_noproc()`
- `svcerr_noprogram()`
- `svcerr_progvers()`
- `svcerr_systemerr()`
- `svcerr_weakauth()`

svcerr_noproc()

Format

```
#include <rpc.h>
void
svcerr_noproc(xprt)
SVCXPRT *xprt;
```

Parameters

xprt

Indicates the pointer to the service transport handle.

Usage

The `svcerr_noproc()` call is called by a service dispatch routine that does not implement the requested procedure.

Context

- `svcerr_decode()`
- `svcerr_noprogram()`
- `svcerr_progvers()`
- `svcerr_systemerr()`
- `svcerr_weakauth()`

svcerr_noprogram()

Format

```
#include <rpc.h>
void
svcerr_noprogram(xprt)
SVCXPRT *xprt;
```

Parameters

xprt
Indicates the pointer to the service transport handle.

Usage

The `svcerr_noprogram()` call is used when the desired program is not registered.

Context

- `svcerr_decode()`
- `svcerr_noprogram()`
- `svcerr_progvers()`
- `svcerr_systemerr()`
- `svcerr_weakauth()`

svcerr_progvers()

Format

```
#include <rpc.h>
void
svcerr_progvers(xprt, low_vers, high_vers)
SVCXPRT *xprt;
u_long low_vers;
u_long high_vers;
```

Parameters

xprt

Indicates the pointer to the service transport handle.

low_vers

Specifies the low version number that does not match.

high_vers

Specifies the high version number that does not match.

Usage

The `svcerr_progvers()` call is called when the version numbers of two RPC programs do not match. The low version number corresponds to the lowest registered version, and the high version corresponds to the highest version registered on the portmapper.

Context

- `svcerr_decode()`
- `svcerr_noproc()`
- `svcerr_noprog()`
- `svcerr_progvers()`
- `svcerr_systemerr()`
- `svcerr_weakauth()`

svcerr_systemerr()

Format

```
#include <rpc.h>
void
svcerr_systemerr(xprt)
SVCXPRT *xprt;
```

Parameters

xprt

Indicates the pointer to the service transport handle.

Usage

The svcerr_systemerr() call is called by a service dispatch routine when it detects a system error that is not handled by the protocol.

Context

- svcerr_decode()
- svcerr_noproc()
- svcerr_noprog()
- svcerr_progvers()
- svcerr_weakauth()

svcerr_weakauth()

Format

```
#include <rpc.h>
void
svcerr_weakauth(xprt)
SVCXPRT *xprt;
```

Parameters

xprt

Indicates the pointer to the service transport handle.

Note: This is the equivalent of `svcerr_auth(xprt, AUTH_T00WEAK)`.

Usage

The `svcerr_weakauth()` call is called by a service dispatch routine that cannot execute an RPC because of correct but weak authentication parameters.

Context

- `svcerr_decode()`
- `svcerr_noproc()`
- `svcerr_noprog()`
- `svcerr_progvers()`
- `svcerr_systemerr()`

svccraw_create()

Format

```
#include <rpc.h>
SVCPRT *
svccraw_create()
```

Parameters

None.

Usage

The svccraw_create() call creates a local RPC service transport used for timings, to which it returns a pointer. Messages are passed using a buffer within the address space of the local process; therefore, the client process must also use the same address space. This allows the simulation of RPC programs within one computer. See “clntcreate_create()” on page 243 for more information.

Return codes

NULL indicates failure.

Context

- svc_destroy()
- svctcp_create()
- svcudp_create()

svctcp_create()

Format

```
#include <rpc.h>
SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
int sock;
u_int send_buf_size;
u_int recv_buf_size;
```

Parameters

sock

Specifies the socket descriptor. If sock is RPC_ANYSOCK, a new socket is created. If the socket is not bound to a local TCP port, it is bound to an arbitrary port.

send_buf_size

Specifies the size of the send buffer. Specify 0 to choose the default.

recv_buf_size

Specifies the size of the receive buffer. Specify 0 to choose the default.

Usage

The svctcp_create() call creates a TCP-based service transport to which it returns a pointer. xprt—>xp_sock contains the transport socket descriptor. xprt—>xp_port contains the transport port number.

Return codes

NULL indicates failure.

Context

- svcraw_create()
- svcudp_create()

svcudp_create()

Format

```
#include <rpc.h>
SVCXPRT *
svcudp_create(sockp, send_buf_size, recv_buf_size)
int sock;
u_int send_buf_size;
u_int recv_buf_size;
```

Parameters

sock

Specifies the socket associated with the service transport handle. If sock is RPC_ANYSOCK, a new socket is created.

send_buf_size

Specifies the size of the send buffer. Specify 0 to choose the default.

recv_buf_size

Specifies the size of the receive buffer. Specify 0 to choose the default.

Usage

The svcudp_create() call creates a UDP-based service transport to which it returns a pointer. xprt->xp_sock contains the transport socket descriptor. xprt->xp_port contains the transport port number.

Return codes

NULL indicates failure.

Context

- svcraw_create()
- svctcp_create()

xdr_accepted_reply()

Format

```
#include <rpc.h>
bool_t
xdr_accepted_reply(xdrs, ar)
XDR *xdrs;
struct accepted_reply *ar;
```

Parameters

xdrs

Specifies the pointer to an XDR stream.

ar

Specifies the pointer to the reply to be represented.

Usage

The xdr_accepted_reply() call translates RPC reply messages.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_array()

Format

```
#include <rpc.h>
bool_t
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
XDR *xdrs;
char **arrp;
u_int *sizep;
u_int maxsize;
u_int elsize;
xdrproc_t elproc;
```

Parameters

- xdrs**
Specifies the pointer to an XDR stream.
- arrp**
Specifies the address of the pointer to the array.
- sizep**
Specifies the pointer to the element count of the array.
- maxsize**
Specifies the maximum number of elements accepted.
- elsize**
Specifies the size of each of the array's elements, found using sizeof().
- elproc**
Specifies the XDR routine that translates an individual array element.

Usage

The xdr_array() call translates between an array and its external representation.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_authunix_parms()

Format

```
#include <rpc.h>
bool_t
xdr_authunix_parms(xdrs, aupp)
XDR *xdrs;
struct authunix_parms *aupp;
```

Parameters

xdrs

Specifies the pointer to an XDR stream.

aupp

Indicates the pointer to the authentication information.

Usage

The xdr_authunix_parms() call translates UNIX-based authentication information.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_bool()

Format

```
#include <rpc.h>
bool_t
xdr_bool(xdrs, bp)
XDR *xdrs;
bool_t *bp;
```

Parameters

xdrs
Specifies the pointer to an XDR stream.

bp
Indicates the pointer to the Boolean.

Usage

The xdr_bool() call translates between booleans and their external representation.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_bytes()

Format

```
#include <rpc.h>
bool_t
xdr_bytes(xdrs, sp, sizep, maxsize)
XDR *xdrs;
char **sp;
u_int *sizep;
u_int maxsize;
```

Parameters

xdrs
Specifies the pointer to an XDR stream.

sp Specifies the pointer to the byte string.

sizep
Indicates the pointer to the byte string size.

maxsize
Specifies the maximum size of the byte string.

Usage

The `xdr_bytes()` call translates between byte strings and their external representations.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- `clnt_broadcast()`
- `clnt_call()`
- `clnt_freeres()`
- `pmap_rmtcall()`
- `registerrpc()`
- `svc_freeargs()`
- `svc_getargs()`
- `svc_sendreply()`

xdr_callhdr()

Format

```
#include <rpc.h>
void
xdr_callhdr(xdrs, chdr)
XDR *xdrs;
struct rpc_msg *chdr;
```

Parameters

xdrs
Specifies the pointer to an XDR stream.

chdr
Specifies the pointer to the call header.

Usage

The xdr_callhdr() call translates an RPC message header into XDR format.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_callmsg()

Format

```
#include <rpc.h>
bool_t
xdr_callmsg(xdrs, cmsg)
XDR *xdrs;
struct rpc_msg *cmsg;
```

Parameters

xdrs

Specifies the pointer to an XDR stream.

cmsg

Specifies the pointer to the call message.

Usage

The xdr_callmsg() call translates RPC messages (header and authentication, not argument data) to and from the XDR format.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_char()

Format

```
#include <rpc.h>

bool_t
xdr_char(xdrs, cp)
XDR *xdrs;
char *cp;
```

Parameters

xdrs
Specifies the pointer to an XDR stream.

cp Specifies the pointer to the C character.

Usage

The xdr_char() call is a filter that translates between C characters and their external representations.

Notes:

1. Encoded characters are not packed, and they occupy 4 bytes each.
2. xdr_string and xdr_text_char() are the only supported routines that convert ASCII to EBCDIC. The xdr_char routine does not support conversion.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()
- xdr_bytes()
- xdr_opaque()xdr_opaque()
- xdr_string()

xdr_destroy()

Format

```
#include <rpc.h>
void
xdr_destroy(xdrs)
XDR *xdrs;
```

Parameters

xdrs
Specifies the pointer to an XDR stream.

Usage

The `xdr_destroy()` call is a macro that invokes the destroy routine associated with the XDR stream, `xdrs`. Destruction usually involves freeing private data structures associated with the stream. Using `xdrs` after invoking `xdr_destroy()` is undefined.

xdr_double()

Format

```
#include <rpc.h>
bool_t
xdr_double(xdrs, dp)
XDR *xdrs;
double *dp;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

dp Indicates the pointer to a double-precision number.

Usage

The xdr_double() call translates between C double-precision numbers and their external representations.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_enum()

Format

```
#include <rpc.h>
bool_t
xdr_enum(xdrs, ep)
XDR *xdrs;
enum_t *ep;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

ep

Indicates the pointer to the enumerated number. enum_t can be any enumeration type, such as enum colors, with colors declared as enum colors (black, brown, red).

Usage

The xdr_enum() call translates between C-enumerated groups and their external representation. When calling the procedures callrpc() and registerrpc(), a stub procedure must be created for both the server and the client before the procedure of the application program using xdr_enum(). This procedure should look like the following:

```
#include <rpc.h>
enum colors (black, brown, red)
void
static xdr_enum_t(xdrs, ep)
XDR *xdrs;
enum colors *ep;
{
    xdr_enum(xdrs, ep)
}
```

The xdr_enum_t procedure is used as the inproc and outproc in both the client and server RPCs. For example:

- An RPC client would contain the following lines:

```
⋮
error = callrpc(argv[1],ENUMRCVPROG,VERSION,ENUMRCVPROC,
               xdr_enum_t,&innumber,xdr_enum_t,&outnumber);
⋮
```

- An RPC server would contain the following line:

```
⋮
registerrpc(ENUMRCVPROG,VERSION,ENUMRCVPROC,xdr_enum_t,xdr_enum_t);
⋮
```

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- `clnt_broadcast()`
- `clnt_call()`
- `clnt_freeres()`
- `pmap_rmtcall()`
- `registerrpc()`
- `svc_freeargs()`
- `svc_getargs()`
- `svc_sendreply()`

xdr_float()

Format

```
#include <rpc.h>
bool_t
xdr_float(xdrs, fp)
XDR *xdrs;
float *fp;
```

Parameters

xdrs

Specifies the pointer to an XDR stream.

fp

Indicates the pointer to the floating-point number.

Usage

The xdr_float() call translates between C floating-point numbers and their external representations.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_free()

Format

```
#include <rpc.h>
void
xdr_free(proc, objp)
xdrproc_t proc;
char *objp;
```

Parameters

proc
Specifies the XDR routine.

objp
Indicates the pointer to the object being freed.

Usage

The xdr_free() call is a generic freeing routine.

Note: The pointer passed to this routine is not freed, but what it points to is freed (recursively).

xdr_getpos()

Format

```
#include <rpc.h>
u_int
xdr_getpos(xdrs)
XDR *xdrs;
```

Parameters

xdrs

Specifies the pointer to an XDR stream.

Usage

The `xdr_getpos()` call is a macro that invokes the get-position routine associated with the XDR stream, `xdrs`. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances do not guarantee this.

Return codes

An unsigned integer, which indicates the position of the XDR byte stream.

Context

- `clnt_broadcast()`
- `clnt_call()`
- `clnt_freeres()`
- `pmap_rmtcall()`
- `registerrpc()`
- `svc_freeargs()`
- `svc_getargs()`
- `svc_sendreply()`

xdr_inline()

Format

```
#include <rpc.h>
long *
xdr_inline(xdrs, len)
XDR *xdrs;
int len;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

len

Specifies the byte length of the desired buffer.

Usage

The xdr_inline() call returns a pointer to a continuous piece of the XDR stream buffer. The value is long * rather than char *, because the external data representation of any object is always an integer multiple of 32 bits.

Note: xdr_inline() can return NULL if there is not sufficient space in the stream buffer to satisfy the request.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_int()

Format

```
#include <rpc.h>
bool_t
xdr_int(xdrs, ip)
XDR *xdrs;
int *ip;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

ip Indicates the pointer to the integer.

Usage

The xdr_int() call translates between C integers and their external representations.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_long()

Format

```
#include <rpc.h>
bool_t
xdr_long(xdrs, lp)
XDR *xdrs;
long *lp;
```

Parameters

xdrs
Indicates the pointer to an XDR stream.

lp Indicates the pointer to the long integer.

Usage

The xdr_long() call translates between C long integers and their external representations.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_opaque()

Format

```
#include <rpc.h>
bool_t
xdr_opaque(xdrs, cp, cnt)
XDR *xdrs;
char *cp;
u_int cnt;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

cp Indicates the pointer to the opaque object.

cnt

Specifies the size of the opaque object.

Usage

The xdr_opaque() call translates between fixed-size opaque data and its external representation.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_opaque_auth()

Format

```
#include <rpc.h>
bool_t
xdr_opaque_auth(xdrs, ap)
XDR *xdrs;
struct opaque_auth *ap;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

ap Indicates the pointer to the opaque authentication information.

Usage

The xdr_opaque_auth() call translates RPC message authentications.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_pmap()

Format

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>
bool_t
xdr_pmap(xdrs, regs)
XDR *xdrs;
struct pmap *regs;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

regs

Indicates the pointer to the portmap parameters.

Usage

The `xdr_pmap()` call translates an RPC procedure identification, such as is used in calls to `portmapper`.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- `clnt_broadcast()`
- `clnt_call()`
- `clnt_freeres()`
- `pmap_rmtcall()`
- `registerrpc()`
- `svc_freeargs()`
- `svc_getargs()`
- `svc_sendreply()`

xdr_pmaplist()

Format

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>
bool_t
xdr_pmaplist(xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

rp Indicates the pointer that points to a pointer to the portmap data array.

Usage

The xdr_pmaplist() call translates a variable number of RPC procedure identifications, such as portmapper creates.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_pointer()

Format

```
#include <rpc.h>
bool_t
xdr_pointer(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

pp Indicates the pointer that points to a pointer.

size

Specifies the size of the target.

proc

Indicates the XDR procedure that translates an individual element of the type addressed by the pointer.

Usage

The `xdr_pointer()` call provides pointer-chasing within structures. This differs from the `xdr_reference()` call in that it can serialize or deserialize trees correctly.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- `clnt_broadcast()`
- `clnt_call()`
- `clnt_freeres()`
- `pmap_rmtcall()`
- `registerrpc()`
- `svc_freeargs()`
- `svc_getargs()`
- `svc_sendreply()`

xdr_reference()

Format

```
#include <rpc.h>
bool_t
xdr_reference(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

pp Indicates the pointer that points to a pointer.

size

Specifies the size of the target.

proc

Specifies the XDR procedure that translates an individual element of the type addressed by the pointer.

Usage

The xdr_reference() call provides pointer-chasing within structures.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_rejected_reply()

Format

```
#include <rpc.h>
bool_t
xdr_rejected_reply(xdrs, rr)
XDR *xdrs;
struct rejected_reply *rr;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

rr Indicates the pointer to the rejected reply.

Usage

The xdr_rejected_reply() call translates rejected RPC reply messages.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_replymsg()

Format

```
#include <rpc.h>
bool_t
xdr_replymsg(xdrs, rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;
```

Parameters

xdrs
Indicates the pointer to an XDR stream.

rmsg
Indicates the pointer to the reply message.

Usage

The xdr_replymsg() call translates RPC reply messages.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_setpos()

Format

```
#include <rpc.h>
int
xdr_setpos(xdrs, pos)
XDR *xdrs;
u_int pos;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

pos

Indicates the pointer to a set position routine.

Usage

The xdr_setpos() call is a macro that invokes the set position routine associated with the XDR stream xdrs. The parameter pos is a position value obtained from xdr_getpos().

Return codes

The value 1 indicates that the XDR stream can be repositioned; the value 0 indicates otherwise.

Note: It is difficult to reposition some types of XDR streams; therefore, this routine might fail with one type of stream and succeed with another.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_short()

Format

```
#include <rpc.h>
bool_t
xdr_short(xdrs, sp)
XDR *xdrs;
short *sp;
```

Parameters

xdrs
Indicates the pointer to an XDR stream.

sp Indicates the pointer to the short integer.

Usage

The xdr_short() call translates between C short integers and their external representations.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_string()

Format

```
#include <rpc.h>
bool_t
xdr_string(xdrs, sp, maxsize)
XDR *xdrs;
char **sp;
u_int maxsize;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

sp Indicates the pointer that points to a pointer to the string.

maxsize

Indicates the maximum size of the string.

Usage

The xdr_string() call translates between C strings and their external representations.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_text_char()

Format

```
#include <rpc.h>
bool_t
xdr_text_char(xdrs, cp)
XDR *xdrs;
char *cp;
```

Parameters

xdrs
Specifies the pointer to an XDR stream.

cp Specifies the pointer to the C character.

Usage

The xdr_text_char() call is a filter primitive that translates between C characters and their external representations.

Notes:

1. Encoded characters are not packed, and they occupy 4 bytes each.
2. xdr_text_char() converts ASCII to EBCDIC.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()
- xdr_bytes()
- xdr_opaque()
- xdr_string()

xdr_u_char()

Format

```
#include <rpc.h>
bool_t
xdr_u_char(xdrs, ucp)
XDR *xdrs;
unsigned char *ucp;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

ucp

Indicates the pointer to an unsigned C character.

Usage

The xdr_u_char() call is a filter primitive that translates between unsigned C characters and their external representations.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_u_int()

Format

```
#include <rpc.h>
bool_t
xdr_u_int(xdrs, up)
XDR *xdrs;
u_int *up;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

up Indicates the pointer to the unsigned integer.

Usage

The xdr_u_int() call translates between C unsigned integers and their external representations.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_u_long()

Format

```
#include <rpc.h>
bool_t
xdr_u_long(xdrs, ulp)
XDR *xdrs;
u_long *ulp;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

ulp

Indicates the pointer to the unsigned long integer.

Usage

The xdr_u_long() call translates between C unsigned long integers and their external representations.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_u_short()

Format

```
#include <rpc.h>
bool_t
xdr_u_short(xdrs, usp)
XDR *xdrs;
u_short *usp;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

usp

Indicates the pointer to the unsigned short integer.

Usage

The xdr_u_short() call translates between C unsigned short integers and their external representations.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_union()

Format

```
#include <rpc.h>
bool_t
xdr_union(xdrs, dscmp, unp, choices, dfault)
XDR *xdrs;
enum_t *dscmp;
char *unp;
struct xdr_discrim *choices;
xdrproc_t dfault;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

dscmp

Indicates the pointer to the union discriminant. enum_t can be any enumeration type.

unp

Indicates the pointer to the union.

choices

Indicates the pointer to an array detailing the XDR procedure to use on each arm of the union.

dfault

Indicates the default XDR procedure to use.

Usage

The xdr_union() call translates between a discriminated C union and its external representation.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Examples

The following is an example of this call:

```
#include <rpc.h>
enum colors (black, brown, red);
bool_t
xdr_union(xdrs, dscmp, unp, choices, dfault)
XDR *xdrs;
enum colors *dscmp;
char *unp;
struct xdr_discrim *choices;
xdrproc_t dfault;
```

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()

- `registerrpc()`
- `svc_freeargs()`
- `svc_getargs()`
- `svc_sendreply()`

xdr_vector()

Format

```
#include <rpc.h>
bool_t
xdr_vector(xdrs, basep, nelem, elemsize, xdr_elem)
XDR *xdrs;
char *basep;
u_int nelem;
u_int elemsize;
xdrproc_t xdr_elem;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

basep

Indicates the base of the array.

nelem

Indicates the element count of the array.

elemsize

Specifies the size of each of array elements, found using sizeof().

xdr_elem

Specifies the XDR routine that translates an individual array element.

Usage

The xdr_vector() call translates between a fixed-length array and its external representation. Unlike variable-length arrays, the storage of fixed-length arrays is static and cannot be freed.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_void()

Format

```
#include <rpc.h>
bool_t
xdr_void()
```

Parameters

None.

Usage

The xdr_void call always returns 1. It may be passed to RPC routines that require a function parameter, where no action is required. This call can be placed in the inproc or outproc parameter of the clnt_call function when you do not need to move data.

Return codes

Always a value of 1.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdr_wrapstring()

Format

```
#include <rpc.h>
bool_t
xdr_wrapstring(xdrs, sp)
XDR *xdrs;
char **sp;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

sp Indicates the pointer that points to a pointer to the string.

Usage

The xdr_wrapstring() call is the same as calling xdr_string() with a maximum size of MAXUNSIGNED. It is useful, because many RPC procedures implicitly invoke two-parameter XDR routines, and xdr_string() is a three-parameter routine.

Return codes

The value 1 indicates success; the value 0 indicates an error.

Context

- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

xdrmem_create()

Format

```
#include <rpc.h>
void
xdrmem_create(xdrs, addr, size, op)
XDR *xdrs;
char *addr;
u_int size;
enum xdr_op op;
```

Parameters

- xdrs**
Indicates the pointer to an XDR stream.
- addr**
Indicates the pointer to the memory location.
- size**
Specifies the maximum size of addr.
- op** Determines the direction of the XDR stream (XDR_ENCODE, XDR_DECODE, or XDR_FREE).

Usage

The xdrmem_create() call creates an XDR stream in memory. It initializes the XDR stream pointed to by xdrs. Data is written to, or read from, addr.

xdrrec_create()

Format

```
#include <rpc.h>

void
xdrrec_create(xdrs, sendsize, recvsiz, handle, readit, writeit)
XDR *xdrs;
u_int sendsize;
u_int recvsiz;
char *handle;
int (*readit) ();
int (*writeit) ();
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

sendsize

Specifies the size of the send buffer. Specify 0 to choose the default.

recvsiz

Specifies the size of the receive buffer. Specify 0 to choose the default.

handle

Specifies the first parameter passed to readit() and writeit().

readit()

Called when a stream input buffer is empty.

writeit()

Called when a stream output buffer is full.

Usage

The xdrrec_create() call creates a record-oriented stream and initializes the XDR stream pointed to by xdrs.

Notes:

1. The caller must set the x_op field.
2. This XDR procedure implements an intermediate record string.
3. Additional bytes in the XDR stream provide record boundary information.

xdrrec_endofrecord()

Format

```
#include <rpc.h>
bool_t
xdrrec_endofrecord(xdrs, sendnow)
XDR *xdrs;
int sendnow;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

sendnow

Specify nonzero to write out data in the output buffer.

Usage

The xdrrec_endofrecord() call can be invoked only on streams created by xdrrec_create(). Data in the output buffer is marked as a complete record.

Return codes

The value 1 indicates success; the value 0 indicates an error.

xdrrec_eof()

Format

```
#include <rpc.h>
bool_t
xdrrec_eof(xdrs)
XDR *xdrs;
```

Parameters

xdrs

Indicates the pointer to an XDR stream.

Usage

The xdrrec_eof() call can be invoked only on streams created by xdrrec_create().

Return codes

The value 1 indicates the current record has been consumed; the value 0 indicates continued input on the stream.

xdrrec_skiprecord()

Format

```
#include <rpc.h>
bool_t
xdrrec_skiprecord(xdrs)
XDR *xdrs;
```

Parameters

xdrs
Indicates the pointer to an XDR stream.

Usage

The xdrrec_skiprecord() call can be invoked only on streams created by xdrrec_create(). The XDR implementation is instructed to discard the remaining data in the input buffer.

Return codes

The value 1 indicates success; the value 0 indicates an error.

xdrstdio_create()

Format

```
#include <rpc.h>
#include <stdio.h>
void
xdrstdio_create(xdrs, file, op)
XDR *xdrs;
FILE *file;
enum xdr_op op;
```

Parameters

- xdrs**
Indicates the pointer to an XDR stream.
- file**
Specifies the data set name for the input/output (I/O) stream.
- op** Determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

Usage

The xdrstdio_create() call creates an XDR stream connected to a file through standard I/O mechanisms. It initializes the XDR stream pointed to by xdrs. Data is written to, or read from, file.

xprt_register()

Format

```
#include <rpc.h>
void
xprt_register(xprt)
SVCXPRT *xprt;
```

Parameters

xprt
Indicates the pointer to the service transport handle.

Usage

The xprt_register() call registers service transport handles with the RPC service package. This routine also modifies the global variables svc_fds and svc_fdset.

Context

svc_fds

xprt_unregister()

Format

```
#include <rpc.h>
void
xprt_unregister(xprt)
SVCXPRT *xprt;
```

Parameters

xprt

Indicates the pointer to the service transport handle.

Usage

The `xprt_unregister()` call unregisters an RPC service transport handle. A transport handle should be unregistered with the RPC service package before it is destroyed. This routine also modifies the global variables `svc_fds` and `svc_fdset`.

Sample RPC programs

z/OS Communications Server provides sample RPC programs. The C source code can be found in the SEZAINST data set.

The following are sample C source modules:

Program	Description
SEZAINST(GENESEND)	RPC client
SEZAINST(GENESERV)	RPC server
RAWEX	RAW client/server

Running RPC sample programs

This section provides information needed to run the GENESERV, GENESEND, and RAWEX modules.

Starting the GENESERV server

To start the GENESERV server, run GENESERV on the other MVS address space (server).

Note: Portmapper must be running before you can run GENESERV.

Running GENESEND client

To start the GENESEND client, run GENESEND MVSX 4445 (MVSX is the name of the host machine where the GENESERV server is running, and 4445 is the integer to send and return).

The following output is displayed:

```
Value sent: 4445
Value received: 4445
```

Running the RAWEX module

To start RAWEX, run RAWEX 6667, (6667 is an integer chosen by you).

The following output is displayed:

```
Argument: 6667
Received: 6667
Sent: 6667
Result: 6667
```

RPC client

The following is an example of an RPC client program.

Note: The characters shown in this example might vary due to differences in character sets. This code is included as an example only.


```

/* GENESEND.C */
/* Send an integer to the remote host and receive the integer back */
/* PORTMAPPER AND REMOTE SERVER MUST BE RUNNING */

/** IBMCOPYR *****/
/*
/* Component Name: GENESEND.C
/*
/*
/* Copyright:    Licensed Materials - Property of IBM
/*
/*              "Restricted Materials of IBM"
/*
/*              5694-A01
/*
/*              (C) Copyright IBM Corp. 1977, 2006
/*
/*              US Government Users Restricted Rights -
/*              Use, duplication or disclosure restricted by
/*              GSA ADP Schedule Contract with IBM Corp.
/*
/* Status:      CSV2R6
/*
/* SMP/E Distribution Name: EZAEC00E
/*
/** IBMCOPYR *****/

static char ibmcopyr[] =
    "GENESEND - Licensed Materials - Property of IBM. "
    "This module is \"Restricted Materials of IBM\" "
    "5647-A01 (C) Copyright IBM Corp. 2006. "
    "See IBM Copyright Instructions.";

#define MVS
#include <stdio.h>
#include <rpc.h>
#include <socket.h>
#include <xdr.h>
#include <svc.h>

#define intrcvprog ((u_long)150000)
#define version    ((u_long)1)
#define intrcvproc ((u_long)1)

main(argc, argv)
    int argc;
    char *argv[];
{
    int innumber;
    int outnumber;
    int error;

    if (argc != 3) {

```

Figure 4. RPC client program sample (Part 1 of 2)

```

    fprintf(stderr,"usage: %s hostname integer\n", argv[0]);
    exit (-1);
} /* endif */
innumber = atoi(argv[2]);
/*
 * Send the integer to the server. The server should
 * return the same integer.
 */
error = callrpc(argv[1],intrcvprog,version,intrcvproc,xdr_int,
                (char *)&innumber,xdr_int,(char *)&outnumber);

if (error != 0) {
    fprintf(stderr,"error: callrpc failed: %d \n",error);
    fprintf(stderr,"intrcprog: %d version: %d intrcvproc: %d",
            intrcvprog, version,intrcvproc);
    exit(1);
} /* endif */

printf("value sent: %d   value received: %d\n", innumber, outnumber);
exit(0);
}

```

Figure 4. RPC client program sample (Part 2 of 2)

RPC server

The following is an example of an RPC server program.

```

/* GENERIC SERVER      */
/* RECEIVE AN INTEGER OR FLOAT AND RETURN THEM RESPECTIVELY */
/* PORTMAPPER MUST BE RUNNING */

/**** IBMCOPYR *****/
/*
/* Component Name: GENESERV.C
/*
/*
/* Copyright:    Licensed Materials - Property of IBM
/*
/*              "Restricted Materials of IBM"
/*
/*              5694-A01
/*
/*              (C) Copyright IBM Corp. 1977, 2006
/*
/*              US Government Users Restricted Rights -
/*              Use, duplication or disclosure restricted by
/*              GSA ADP Schedule Contract with IBM Corp.
/*
/* Status:      CSV1R8
/*
/* SMP/E Distribution Name: EZAEC00F
/*
/*
/**** IBMCOPYR *****/

static char ibmcopyr[] =
    "GENESERV - Licensed Materials - Property of IBM. "
    "This module is \"Restricted Materials of IBM\" "
    "5647-A01 (C) Copyright IBM Corp. 2006. "
    "See IBM Copyright Instructions.";

#ifdef MVS
#define MVS
#endif

#include <rpc.h>
#include <stdio.h>
#include <xdr.h>
#include <svc.h>

#define intrcvprog ((u_long)150000)
#define fltrcvprog ((u_long)150102)
#define intvers    ((u_long)1)
#define intrcvproc ((u_long)1)
#define fltrcvproc ((u_long)1)
#define fltvers    ((u_long)1)

main()
{
    int *intrcv();

```

Figure 5. RPC server program sample (Part 1 of 2)

```

float *floatrcv();

/*REGISTER PROG, VERS AND PROC WITH THE PORTMAPPER*/

    /*FIRST PROGRAM*/
    registerrpc(intrcvprog,intvers,intrcvproc,intrcv,xdr_int,xdr_int);
    printf("Intrcv Registration with Port Mapper completed\n");

    /*OR MULTIPLE PROGRAMS*/
    registerrpc(floatrcvprog,fltvers,flttrcvproc,
        floatrcv,xdr_float,xdr_float);
    printf("Floatrcv Registration with Port Mapper completed\n");

/*
 * svc_run will handle all requests for programs registered.
 */
svc_run();
printf("Error:svc_run returned!\n");
exit(1);
}

/*
 * Procedure called by the server to receive and return an integer.
 */
int *
intrcv(in)
    int *in;
{
    int *out;

    printf("integer received: %d\n",*in);
    out = in;
    printf("integer being returned: %d\n",*out);
    return (out);
}

/*
 * Procedure called by the server to receive and return a float.
 */
float *
floatrcv(in)
    float *in;
{
    float *out;

    printf("float received: %e\n",*in);
    out=in;
    printf("float being returned: %e\n",*out);
    return(out);
}

```

Figure 5. RPC server program sample (Part 2 of 2)

RPC raw data stream

The following is an example of an RPC raw data stream program.

```

/*RAWEX                                     */
/* AN EXAMPLE OF THE RAW CLIENT/SERVER USAGE */
/* PORTMAPPER MUST BE RUNNING               */

static char ibmcopyr[] =
    "RAWEX      - Licensed Materials - Property of IBM. "
    "This module is \"Restricted Materials of IBM\" "
    "5647-A01 (C) Copyright IBM Corp. 1994. "
    "See IBM Copyright Instructions.";

/**** IBMCOPYR *****/
/*
/* Component Name: RAWEX.C (alias EZAEC01H)
/*
/* Copyright:      Licensed Materials - Property of IBM
/*
/*                  "Restricted Materials of IBM"
/*
/*                  5647-A01
/*
/*                  (C) Copyright IBM Corp. 1977, 1998
/*
/*                  US Government Users Restricted Rights -
/*                  Use, duplication or disclosure restricted by
/*                  GSA ADP Schedule Contract with IBM Corp.
/*
/* Status:         CSV2R6
/*
/*
/* SMP/E Distribution Name: EZAEC01H
/*
/*
/**** IBMCOPYR *****/

/*
* This program does not access an external interface. It provides
* a test of the raw RPC interface allowing a client and server
* program to be in the same process.
*
*/
#ifdef MVS
#define MVS
#endif
#include <rpc.h>
#include <stdio.h>

#define rawprog ((u_long)150104)
#define rawvers ((u_long)1)
#define rawproc ((u_long)1)

extern enum clnt_stat clnt_raw_call();
extern void raw2();

main(argc,argv)

```

Figure 6. RPC raw data stream program sample (Part 1 of 3)

```

int argc;
char *argv[];
{
    SVCXPRT *transp;
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int bout,in;
    register CLIENT *clnt;
    enum clnt_stat cs;
    int addrlen;

    /*
     * The only argument passed to the program is an integer to
     * be transferred from the client to the server and back.
     */
    if(argc!=2) {
        printf("usage:  %s    integer\n", argv[0]);
        exit(-1);
    }
    in = atoi(argv[1]);

    /*
     * Create the raw transport handle for the server.
     */
    transp = svcraw_create();
    if (transp == NULL) {
        fprintf(stderr, "can't create an RPC server transport\n");
        exit(-1);
    }

    /* In case the program is already registered, deregister it */
    pmap_unset(rawprog, rawvers);

    /* Register the server program with PORTMAPPER */
    if (!svc_register(transp,rawprog,rawvers,raw2, 0)) {
        fprintf(stderr, "can't register service\n");
        exit(-1);
    }
    /*
     * The following registers the transport handle with internal
     * data structures.
     */
    xprt_register(transp);

    /*
     * Create the client transport handle.
     */
    if ((clnt = clntraw_create(rawprog, rawvers)) == NULL ) {
        clnt_pcreateerror("clntudp_create");
        exit(-1);
    }
    total_timeout.tv_sec = 60;
    total_timeout.tv_usec = 0;
    printf("Argument:  %d\n",in);
}

```

Figure 6. RPC raw data stream program sample (Part 2 of 3)

```

/*
 * Make the call from the client to the server.
 */
cs=clnt_call(clnt,rawproc,xdr_int,
             (char *)&in,xdr_int,(char *)&bout,total_timeout);

printf("Result: %d",bout);
if(cs!=0) {
    clnt_perror(clnt,"Client call failed");
    exit(1);
}
exit(0);
}

/*
 * Service procedure called by the server when it receives the client
 * request.
 */
void raw2(rqstp,transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    int in,out;
    if (rqstp->rq_proc==rawproc) {
        /*
         * Unpack the integer passed by the client.
         */
        svc_getargs(transp,xdr_int,&in);
        printf("Received: %d\n",in);
        /*
         * Send the integer back to the client.
         */
        out=in;
        printf("Sent: %d\n",out);
        if (!svc_sendreply(transp, xdr_int,&out)) {
            printf("Can't reply to RPC call.\n");
            exit(1);
        }
    }
}
}

```

Figure 6. RPC raw data stream program sample (Part 3 of 3)

RPCGEN sample programs

This section provides information about sample RPCGEN programs. The C source code can be found in the SEZAINST data set.

The following are sample C source files:

File	Description
RG	RPCGEN user-generated input
RGUC	RPCGEN user-generated client
RGUS	RPCGEN user-generated server

Generating your own sequential data sets

The following steps describe how to generate your own sequential data sets:

1. Execute RPCGEN RG from the TSO command line.

The following sequential data sets are generated in your user space:

- *user_id.RG.H*
 - *user_id.RGC.C*
 - *user_id.RGS.C*
 - *user_id.RGX.C*
2. Verify that the sample C source code modules RGUC and RGUS contain the `#include` statements found in *user_id.RGX.C*.
 3. Verify that *user_id.RG.H* is referenced by the compile procedure.

Building client and server executable modules

Complete the following steps to build client and server executable modules:

1. Compile the RGUS C source program.
2. Compile the RGUC C source program.
3. Compile the RGS.C C source program generated by RPCGEN.
4. Compile the RGC.C C source program generated by RPCGEN.
5. Link-edit the sample source modules RGS and RGUS.
6. Link-edit the sample source modules RGUC and RGC.

Running RPCGEN sample programs

This section provides information needed to run the sample programs in RPCGEN.

1. Execute RGS on the other MVS address space (server).
No message is displayed.
2. Execute RGUC MVSX 6504 (MVSX is the host machine where the RGS server is running, and 6504 is the integer chosen by you).

After executing the RGUC client, the following message is displayed:

Output on the server session: 6504

Chapter 9. Remote procedure calls in the z/OS UNIX System Services environment

The z/OS UNIX files used by z/OS UNIX System Services RPC and their location in the z/OS UNIX file system are as follows:

- /usr/include/rpc: All header files are contained here.
- /usr/lib/librpclib.a: RPC archive files.
- orpcgen: ONC RPC protocol compiler.
- orpcinfo: Utility program for looking at portmaps of networked machines.
- oportmap: Network service program that maps ONC RPC program and version numbers to transport-specific port numbers.

Deviations from Sun RPC 4.0

Source margins

The source was modified to fit into 72 columns.

Functions

xdr_enum()

In z/OS UNIX System Services `rpc` `xdr_enum()` is a macro. This is a change identical to the changes in TCP/IP Version 2 for MVS and VM, and Version 3.1 for MVS. It is necessary because enumerations in C/370™ may have a length of 1, 2, or 4 bytes. The `enum_t` is not defined and `xdr_enum()` is replaced first by a call to `_xdr_enum()` that returns the entry to the appropriate XDR routine (`xdr_char()`, `xdr_short()`, or `xdr_long()`), which is then followed by a call to that routine. The `xdr_union()` is also modified into a macro, which separates the call for the discriminant from the remainder. The discriminant is processed as an enumeration, and then passed as a value to `_xdr_union()` to process the remaining union.

xdr_string()

As with previous 370 versions of TCP/IP, `xdr_string()` translates from EBCDIC to ASCII or reverse. With z/OS UNIX System Services the `iconv()` call is used, and data is translated directly into or out of the XDR buffers if sufficient buffers are available as indicated by an `xdr_inline()` call. With previous versions (or with z/OS UNIX System Services if the entire string will not fit into the buffer) it is necessary to allocate an additional buffer. While encoding, if the length of the data changes in the translation, `xdr_setpos()` is used to adjust the XDR buffer to reflect the actual amount of translated data. `realloc()` is used while decoding or for the temporary buffer, which may be necessary while encoding.

The default translation is between ISO8859-1 and IBM-1047. This can be modified by `iconv_open()` calls during initialization, by specifying the external `iconv_t` variables `xdr_hton_cd` and `xdr_ntoh_cd`.

xdr_float(), xdr_double()

The format for S/370™ floating point data differs from the IEEE format specified for XDR. The `xdr_float()` and `xdr_double()` routines are modified to make the

necessary conversions. For z/OS UNIX System Services, these routines utilize the C/370 library routines `frexp()` and `ldexp()` to extract and restore the exponent from the floating point number, rather than private subroutines.

Using z/OS UNIX System Services RPC

For RPC, a Sun ONC sample program is provided in `/usr/lpp/tcpip/rpc/samples`. To run the sample, you can run the Makefile facility in the `rpc` samples directory. Running *make* produces three executable files.

- `printmsg`
The command **`printmsg text`** prints the message (text) on the local console. It can be displayed by viewing the system log.
- `msg_svc`
`msg_svc` is an RPC server that enables the user at a remote station to put a message on the console of the server. The command **`msg_svc &`** starts this server.
- `rprintmsg`
The command **`rprintmsg rhost text`** prints a message (text) on the console of host *rhost*.

Note: The `_C89_LIBDIRS` environmental variable must be set (for example, `export _C89_LIBDIRS=/usr/lpp/tcpip/lib`) before the make is executed.

A sample makefile is provided: `/usr/lpp/tcpip/rpc/samples/Makefile`. To run *make*, use `make -f /usr/lpp/tcpip/rpc/samples/Makefile` from a writable directory.

New cache call function for RPC

```
svcudp_enablecache(transp, size)
SVCXPRT *transp;
u_long size;
```

where:

- *svcudp_enablecache* enables the caching of replies to remote calls using UDP. When a request due to a retry is received, and there is a reply to an earlier attempt in the cache, the cached reply is immediately returned to the client without calling the remote procedure.
- *transp* is the UDP service transport for which caching is to be enabled.
- *size* is the number of entries to be provided in the cache.

When issuing `RPCGEN` for a specification file that contains a `%#`, the following compiler error message may be displayed: `ERROR EDC0401 abc.x:n` The character is not valid, where *abc.x* is the name of the file and *n* is the line number containing a `%#`. This combination of characters is not accepted by the compiler.

Support for 64-bit integers

Four XDR functions support 64-bit integers in the z/OS UNIX System Services RPC API.

The function `xdr_hyper()` is equivalent to `xdr_longlong_t()`. The function `xdr_u_hyper()` is equivalent to `xdr_u_longlong_t()`.

XDR Function	Description
<code>xdr_hyper()</code>	Translates between C long longs and their external representatives.
<code>xdr_u_hyper()</code>	Translates between C unsigned long longs and their external representatives.
<code>xdr_longlong_t()</code>	Translates between C long longs and their external representatives.
<code>xdr_u_longlong_t()</code>	Translates between C unsigned long longs and their external representatives.

UDP transport protocol CLIENT handles

The function of `clntudp_bufcreate()` is similar to `clnttcp_create()` but creates UDP transport protocol CLIENT handles. The wait time for retries and timeouts is specified for the UDP transport. The total time allowed for RPC completion can be specified by `clnt_call()`. Buffer sizes may be specified or defaulted. The same potential for version number mismatch exists. Success returns the CLIENT handle, failure NULL.

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

Restrictions

RPC does not support the Binary Floating Point Facility. If you install the BFP processor, you must compile your RPC applications to preclude use of the BFP hardware. You can do this by specifying compiler option `ARCH(0)`, (the default setting).

Chapter 10. Network Computing System

The Network Computing System (NCS) is a set of tools for heterogeneous distributed computing. These tools conform to the Network Computing Architecture. This topic introduces the Network Computing Architecture and NCS.

To use the NCS system calls, you must know C language programming. For more information about NCS, see the *NCS for IBM AIX/ESA Planning and Administration Guide* and the *NCS for IBM AIX/ESA Programming Reference*.

NCS and the Network Computing Architecture

NCS is an implementation of the Network Computing Architecture, an architecture for distributing software applications across heterogeneous collections of computers, networks, and programming environments. Programs based on NCS can take advantage of computing resources throughout a network or internet, with different parts of each program executing on the computers best suited for the tasks.

The Network Computing Architecture supports distributed programs of many kinds. For example, one program might perform graphical input and output on a workstation while it does intense computation on a supercomputer. Another program might perform many independent calculations on a large set of data; it could distribute these calculations among any number of available processors on the network or internet.

NCS components

The components of NCS are written in portable C wherever possible. They are available in source code and in several binary formats. Currently, the NCS components are:

- Remote procedure call (RPC) runtime library
- Location Broker
- Network Interface Definition Language (NIDL) compiler

The RPC runtime library and the Location Broker provide runtime support for network computing. These two components, along with various utilities and files, make up the Network Computing Kernel (NCK), which contains all the software you need to run a distributed application.

The Network Interface Definition Language (NIDL) compiler is a tool for developing distributed applications.

Remote procedure call runtime library

The RPC runtime library is the backbone of the Network Computing System. It provides the calls that enable local programs to execute procedures on remote hosts. These calls transfer requests and responses between clients (the programs calling the procedures) and servers (the programs executing the procedures).

When you write NCS applications, you usually do not use many RPC runtime library calls directly. Instead, you write interface definitions in NIDL and use the NIDL compiler to generate most of the required calls to the runtime library.

Location broker

A broker is a server that provides information about resources. The Location Broker enables clients to locate specific objects (for example, databases) or specific interfaces (for example, data retrieval interfaces). Location Broker software includes the Global Location Broker (GLB), the Local Location Broker (LLB), a client agent through which programs use GLB and LLB services, and administrative tools.

The GLB stores in a database the locations of objects and interfaces throughout a network or internet; clients can use the GLB to access an object or interface without knowing its location beforehand. The LLB stores in a local database similar information about resources on the local host; it also implements a forwarding facility that provides access by means of a single address to all of the objects and interfaces at the host.

Network interface definition language compiler

The NIDL compiler takes as input an interface definition written in NIDL. From this definition, the compiler generates source code in portable C for client and server stub modules. An interface definition specifies the interface between a user of a service and the provider of the service; it defines how a client *sees* a remote service and how a server *sees* requests for its service.

The stubs produced by the NIDL compiler contain nearly all of the *remoteness* in a distributed application. They perform data conversions, assemble and disassemble packets, and interact with the RPC run-time library. It's much easier to write an interface definition in NIDL than it would be to write the stub code that the NIDL compiler generates from your definition.

MVS implementation of NCS

The following list indicates the NCS components that are available in MVS or z/OS UNIX.

- Network Interface Definition Language (NIDL) compiler 1.0
- Network Computing Kernel (NCK) 1.1

The IBM MVS implementation of NCS differs from the Apollo Computer, Inc. implementation of NCS. The following list summarizes the differences between the two implementations:

- The IBM MVS implementation of NCS contains support for the Non-Replicated Global Location Broker daemon (nrglbd). It does not contain support for the Global Location Broker daemon (glbd), which can be replicated on multiple hosts in the network.
- The IBM MVS implementation of NCS does not contain support for the Data Replication Manager Administrative Tool (drm_admin). This tool works only with the replicated version of the Global Location Broker, which is not supported in MVS NCS.
- The IBM MVS implementation of NCS does not support multitasking. Neither does it support forking or spawning a task. It does not support Apollo's Concurrent Programming Support (CPS).
- The IBM MVS implementation of NCS supports AF_INET only.
- In NCS, the receiving machine (client or server) translates EBCDIC characters to ASCII and ASCII characters to EBCDIC. The IBM MVS implementation of NCS translates correctly, but the Apollo NCS Version 1.0 code has the following problems:

- The EBCDIC Null character 0x00 is incorrectly translated to the ASCII character 0x02. It should be translated to the ASCII character 0x00.
- The EBCDIC Delete character 0x07 is incorrectly translated to the ASCII character 0x10.
- The EBCDIC Line Feed character 0x25 is incorrectly translated to the ASCII character 0x3f.

These are the three significant errors in the EBCDIC to ASCII translation table that is part of NCS Version 1.0. EBCDIC to ASCII translation works correctly only if you do not use the previous characters or if the EBCDIC to ASCII translation table has already been fixed in the NCS program on the receiving side.

- NCS Version 1.0 does not correctly translate between IBM floating point and IEEE floating point. This includes both the translation from IEEE to IBM floating point and IBM to IEEE floating point. As with EBCDIC to ASCII translations, the receiver of the data performs the floating point conversion. Servers and clients can both act as receivers of data. Therefore, NCS programs on both sides need to contain correct support of IBM floating point if you pass floating point data to or from a system that uses IBM floating point.
- Apollo NCS Version 1.0 supports two enum data types: the short enum, which NCS assumes occupies 2 bytes in storage; and the regular enum, which occupies 4 bytes. The IBM C/370 Compiler dynamically determines the size required for an enum variable as 1 byte, 2 bytes, or 4 bytes.

The NCS short enum data type works correctly on MVS, but the NCS regular enum data type does not. If for some reason you cannot use the short enum data type on MVS and must use the regular enum data type, then you must force the C/370 compiler to allocate 4 bytes for all enum variables.

If your Interface Definition Language (IDL) contains enum typedefs as input to the NIDL Compiler, for example

```
typedef enum {low, medium, high} word;
typedef enum {red, green, blue} colors;
```

then you must modify the header data set that gets generated by the NIDL compiler. If the header data set is to be used on MVS with the C/370 compiler, you must force the compiler to use fullword enumeration types:

```
/* you should add the following define to the header data set */
#define INT_MAX (0xffffffff)

/* you need to modify the declares for the enum data type to */
/* force the compiler to use 4 bytes (word) for regular enum. */
enum word {low, medium, high, word_expand_to_fullword = INT_MAX};
enum colors {red, green, blue, colors_expand_to_fullword = INT_MAX};
```

If you do not force the compiler to use fullword enumeration types, the compiler assigns either 1 byte or 2 bytes to your enum variables and the enum variables are not transmitted correctly using NCS.

Note: MVS NCS does not support C language pragma statements.

NCS system IDL data sets

The NCS System Interface Definition Language (IDL) data sets consist of several interface definition data sets that are distributed with NCS. These data sets define types and constants, or local or remote interfaces. Some of these data sets can be imported by your own IDL data set. The import declaration is an NIDL statement

similar to the C `#include` directive, which causes other IDL data sets to be included by the NIDL compiler. You do not need to run NIDL against the data sets to be imported.

- `base.idl`
- `conv.idl`
- `glb.idl`
- `lb.idl`
- `llb.idl`
- `nbase.idl`
- `rpc.idl`
- `rrpc.idl`
- `socket.idl`
- `uuid.idl`

For more information on IDL files, see the *NCS for IBM AIX/ESA Planning and Administration Guide*.

NCS C header data sets and the Pascal include data set

The following is a list of the C header data sets that you might need to include in your C source programs to use NCS. These data sets can also be included by the NIDL-generated stub code. These data sets are located in SEZACMAC and must be copied to your user ID.

The following is a list of the headers used by NCS:

<code>base.h</code>	<code>ncsdefs.h</code>
<code>conv.h</code>	<code>ncssock.h</code>
<code>glb.h</code>	<code>pfm.h</code>
<code>bsdto cms.h</code>	<code>rpc.h</code>
<code>idl@base.h</code>	<code>rrpc.h</code>
<code>lb.h</code>	<code>socket.h</code>
<code>llb.h</code>	<code>uuid.h</code>
<code>nbase.h</code>	

IDL@BASE.COPY is the name of the Pascal include data set. This data set should be included in your client or server source code if it is written in Pascal.

NCS RPC run-time library

On MVS, all of the routines that make up the NCS RPC run-time library are stored in the SEZALIBN data set. This library must be specified on the SYSLIB DD statement of your link-edit job step.

Portability issues

There are several NCS-based portability issues of which you need to be aware.

NCS defines NCSDEFS.H

The linkage editor and loader on MVS restrict the number of characters in an external name to eight characters or less. This means that if you are porting an existing non-MVS program, and it contains external references that are longer than eight characters, you need to redefine these references into unique, eight-character

names. If you are writing new code on MVS and you create external references that are longer than eight characters, you also have to redefine these references into unique eight-character names.

A data set called NCSDEFS.H, contains the redefines of all the external references greater than eight characters in length that are part of the NCS RPC run-time library. This data set needs to be included in all of your code that uses NCS.

Figure 7 shows the lines of code that should be included in each NCS-based routine to maintain portability of your code.

```
#ifdef IBM370
# include "ncsdefs.h"    /* NCS redefines for IBM 370.*/
#endif
```

Figure 7. Macro to maintain IBM System/370 portability

To compile the code on MVS, define IBM370 to the compiler by using the compile option `DEFINE(IBM370)`. By isolating MVS-dependent sections of code, you can maintain code portability.

Required user-defined USERDEFS.H

The NIDL compiler generates stub code. For this stub code to compile correctly on MVS, the external references greater than eight characters must be redefined to eight characters or less. The data set USERDEFS.H contains a template for the information that needs to be redefined.

The following are considerations when using the USERDEFS.H data set.

- The data set should be copied to your user ID and be renamed to something appropriate for your NCS-based code (for example, *user_id.USERDEFS.H*).

This data set is a good place to put any code-specific external names longer than eight characters that need to be redefined.

- The data set must always contain the redefines for the server and client entry point vector (epv). See the example USERDEFS.H data set shown in this section for more information about USERDEFS.H.
- The data set should be included in all your NCS-based source code
- The data set must be included by the NIDL-generated stubs and switches.

To have NIDL automatically add this include, use the NIDL run-time option `-inc`.

Figure 8 shows the H data set in the stub and switch code. You should also follow this method for including the USERDEFS.H data set (or whatever you renamed it) in your NCS-based code.

```
#ifdef IBM370
# include "ncsdefs.h"
# include "userdefs.h"
#endif
```

Figure 8. NCSDEFS.H and USERDEFS.H include statements

The following provides an example of the USERDEFS.H data set:

```
/*****
 *      Template for User Redefines
 *      On IBM MVS or MVS operating systems external references longer
```

```

*   than 8 characters must be redefined to 8 characters
*   or less. This data set must be included in your Client or Server
*   code, and you must provide the nidl compiler with the name of
*   this data set when nidl is invoked so that the stub code can also
*   include it.
*****/
#define IDL_interface_name_server_epv xxxSEpv
#define IDL_interface_name_client_epv xxxCEpv

```

The following is a description of the elements shown in the preceding example.

Element	Description
IDL_interface_name	The interface name coded in your IDL data set. You must replace <i>IDL_interface_name</i> with this name.
xxx	A unique three-character sequence, starting with a letter, that makes this redefine name unique throughout your NCS-based programs. For example, the <i>xxx</i> could be replaced with the first 3 characters of the <i>IDL_interface_name</i> .

See “NIDL compiler options” on page 344 for a description of NIDL run-time options.

Preprocessing, compiling, and linking

The following sections provide information about how to compile and link-edit your program:

- NCS Preprocessor Programs
- Compiling and Linking NCS Programs

NCS preprocessor programs

The NIDL compiler translates an NIDL interface definition into the NCS client and server stub modules. Before the C/C++ for z/OS compiler can be run on NCS-based code, any \$ (such as those in the NCS RPC run-time library routines) must be converted to an underscore (_). You can use CPP to do this conversion. For more information about CPP, see “Converting C identifiers using the CPP program” on page 344.

NIDL compiler

The Network Interface Definition Language (NIDL) compiler is a member of SEZALOAD. MVS data sets written in NIDL must have the form *user_id.name.IDL*. The NIDL compiler generates a server stub data set, a client stub data set, a client switch data set, and a header data set.

For more information about NIDL, see the *NCS for IBM AIX/ESA Planning and Administration Guide*.

A command list (CLIST) called RUNNIDL is provided to assist you in invoking the NIDL compiler. RUNNIDL is a member of SEZAINST. The NIDL options specified in RUNNIDL CLIST are set to the most frequently used NIDL run-time options. If you do not want to run with these NIDL options, you can invoke the NIDL compiler directly.

The *NIDL* compiler does not support IDL include files that are members of a partitioned data set.

Member	Data set name
basei	<i>user_id</i> .base.idl
conv	<i>user_id</i> .conv.idl
glb	<i>user_id</i> .glb.idl
lbi	<i>user_id</i> .lb.idl
llbi	<i>user_id</i> .llb.idl
nbasei	<i>user_id</i> .nbase.idl
rpci	<i>user_id</i> .rpc.idl
rrpci	<i>user_id</i> .rrpc.idl
socketi	<i>user_id</i> .socket.idl
uuidi	<i>user_id</i> .uuid.idl

Timing diagram for IDL_d_s_n signal. The signal is high for RUNNIDL and low for IDL. A bracket labeled 'inc (d_s_n)' spans the low period. A bracket labeled 'C' spans the high period.

The following example invokes the NIDL compiler using the BANK.IDL data set as input. The header data set containing the redefines for BANK is in the data set BANKDEFS.H.

NIDL compiler limitations: You should be aware of the following limitations concerning the NIDL compiler options on MVS.

- You cannot invoke the NCS CPP routine from within the NIDL compiler. If you invoke NIDL directly, you must specify the `-no_cpp` option.

- The extension option is used to generate unique data set names for the NIDL output. The defaults for `-ext` on MVS are `@C.C@CSTUB`, `@S.C@SSTUB`, and `@W.C@CSWTCH`. The extension is appended to the data set name of the IDL data set to generate a unique data set name for the two stubs and the switch.

For example, the IDL data set name and default extension for a client switch are appended in the following format:

`IDL_data_set_name@W.C@CSWTCH`

Note: This default restricts the IDL data set name to 6 characters or less.

The following is a list of data set names and default low-level qualifiers for the NIDL generated output:

Data set name	Low-level qualifier	Description
<code>IDL_data_set_name@C</code>	<code>C@CSTUB</code>	Client stub
<code>IDL_data_set_name@W</code>	<code>C@CSWTCH</code>	Client switch
<code>IDL_data_set_name@S</code>	<code>C@SSTUB</code>	Server stub
<code>IDL_data_set_name</code>	<code>H</code>	C header data set
<code>IDL_data_set_name</code>	<code>COPY</code>	Pascal header data set (if the pascal option is used).

You can change this default by invoking NIDL directly and specifying your own `-ext` option. If you specify your own `-ext` option, the name of your data set is restricted to a maximum of 8 characters, and the extension is restricted to a maximum of 8 characters.

NIDL compiler options: The linkage editor and loader on MVS restricts the number of characters in an external name to 8 characters or less. For the code generated by the NIDL compiler to compile correctly on MVS, the external references greater than 8 characters need to be redefined to 8 characters or less. The data set `USERDEFS.H` contains a template for the information to be redefined.

The `-inc` option allows you to specify the data set name of a header data set that contains redefines specific to your programs and stubs. If the `-inc` option is specified, the NIDL compiler generates code to `#include` the user-specified `-inc` data set name in the stub and switch code that it generates.

For example, the `BANK` sample program has a `BANKDEFS.H` data set, where all of the `BANK` external names greater than 8 characters are redefined. When the NIDL compiler is run against the `BANK.IDL` data set, if you specified `-inc bankdefs`, the `#include` for this data set is automatically generated in the two stubs and switch programs. The following is an example of the code:

```
#ifdef IBM370
# include "ncsdefs.h"
# include "bankdefs.h"
#endif
```

Converting C identifiers using the CPP program

All of the NCS RPC run-time library routines and most of the NCS constants and data types contain a `$` character. For example, the routine you call to register your server with RPC run-time is `rpc_$register`. The routine you call to register your server with the location broker is `lb_$register`.

IBM C/370, based on ANSI standards, does not allow a `$` to be used as a correct character in a C identifier. The IBM C/370 preprocessor does not allow you to redefine a `$` to another character. NCS provides a routine called `CPP` for systems that do not allow a `$` in C identifiers. The NCS `CPP` program reads a C source data set, expands macros and include data sets, and writes an input for the C compiler. The most important function that the `CPP` program performs for MVS NCS users is that it converts every `$` to an underscore (`_`) when it occurs in a C identifier.

Before any of your code or the stub code can be compiled, all occurrences of a \$ in a C identifier must be converted to an underscore (_). NCS uses CPP to do this.

Note: Because CPP does not contain all the functions of the C/370 preprocessor, there can be times when you need to modify your code to make it acceptable to CPP, even though C/370 might have accepted it.

A CLIST called RUNCPP is provided to assist you in invoking the CPP program. You can use this CLIST, or invoke CPP directly. RUNCPP is a member of SEZAINST.

Use the RUNCPP CLIST command in the following format:

►►—RUNCPP—*data_set_name*—*data_set_type*—◄◄

Parameter	Description

<i>data_set_name</i>	Specifies the name of the data set used as input to NCS CPP.

<i>data_set_type</i>	Specifies the data set type.

To run CPP with the data set BANK.C@CSTUB as input, enter the following:

RUNCPP BANK C@CSTUB

The RUNCPP CLIST has the most frequently used CPP run-time options hard coded into it. IBM recommends using RUNCPP, but if you must use options that are not specified with RUNCPP, invoke CPP directly.

For portability reasons, you should leave the \$ in all the RPC run-time routines, constants, and data types. CPP should be run against your code after you run NIDL. In this way, the client stub and switch or server stub can be moved to a system that supports the \$. For portability to other systems, you should always maintain the version of your code that contains the \$.

For programs that are not run on any system other than IBM MVS, you can permanently change \$ to (_), so that you do not have to use CPP. Then, only the client stub and switch or the server stub has to be run through the CPP routine. In some cases, this is the preferred solution, especially if you need the full function of the C/C++ for z/OS preprocessor and compiler and CPP does not include this support. For example, many AD/Cycle C/370 header files contain preprocessor directives that CPP does not understand. If you are including AD/Cycle C/370 header files in your application, you should manually change \$ to underscore (_) in your application and any included header files so that you do not have to run CPP.

CPP does not support C include files that are members of a partitioned data set. Any NCS C header files that are included by your data set must be copied to your user ID. The following are the members of SEZACMAC that you might need to copy:

Member	Data set name
ncsock1	<i>user_id.socket.h</i>
ncsrpc	<i>user_id.rpc.h</i>
base	<i>user_id.base.h</i>
conv	<i>user_id.conv.h</i>
glb	<i>user_id.glb.h</i>
bsdtocms	<i>user_id.bsdtocms.h</i>

idl@base	<i>user_id.idl@base.h</i>
lb	<i>user_id.lb.h</i>
llb	<i>user_id.llb.h</i>
nbase	<i>user_id.nbase.h</i>
ncsdefs	<i>user_id.ncsdefs.h</i>
ncsock	<i>user_id.ncsock.h</i>
pfm	<i>user_id.pfm.h</i>
rrpc	<i>user_id.rrpc.h</i>
uuid	<i>user_id.uuid.h</i>

Any C/370 standard header files that are included by your data set must be copied from the C/370 product header partitioned data set (SEZACMAC).

Compiling and linking NCS programs

Following are the steps needed to create, build, and execute an NCS application:

1. Set up

Copy RUNNIDL and RUNCPP from SEZAINST to one of your system-supported CLIST libraries.

2. Write the IDL description of the client and server applications.

Write your NIDL interface program and client or server code, and your userdefs-type header file that redefines your long names.

3. Run NIDL

- Copy any imported NCS IDL files from SEZAINST to your user ID.
- Run the NIDL compiler using your IDL data set as input.

```
RUNNIDL middle_qualifier IDL INC(userdefs)
```

If your data set is *user_id.SAMPLE.IDL* and your header file is *user_id.USERDEFS.H*, the command to run is:

```
RUNNIDL SAMPLE IDL INC(userdefs)
```

4. Convert \$ to _

You can convert any identifiers containing a \$ either using CPP or manually.

• Run CPP

- Copy any included header files from the partitioned data set in which it resides to your user ID.
- Run CPP against all of your code, the client stub and switch, and the server stub.

```
RUNCPP middle_qualifier low_level_qualifier
```

If your data set is *user_id.SAMPLE.C*, run the following command:

```
RUNCPP SAMPLE C
```

• Manually convert \$ to underscore (_):

- Use an editor to convert all occurrences of \$ to _ in all of your code, the client stub and switch, and the server stub.
- Copy to a partitioned data set any C header files that contain a \$ and that are included by your code, the client stub or switch, or the server stub. Edit the C header files in the partitioned data set to convert all occurrences of \$ to _. During compilation, this partitioned data set must be specified on the SYSLIB statement ahead of SEZACMAC.

5. Compile and Link

You can use several methods to compile, link-edit, and execute your C/C++ for z/OS source program in MVS. This section contains information about the

additional data sets that you must include to run the C data sets generated by RUNCPP under MVS batch, using IBM-supplied cataloged procedures.

The following list contains data set names, which are used as examples in the following JCL statements:

Data set name
Contents

user_id.SAMPLE.CPPOUT

Sequential data set that contains the C program generated by RUNCPP.

user_id.OBJ

A partitioned data set that contains the compiled versions of C programs as its members.

user_id.LOADLIST

A partitioned data set that contains the loadlist as its members.

user_id.LOAD

A partitioned data set that contains the link-edited versions of C programs as its members.

user_id.HDRS

A partitioned data set that contains C header files as its members.

Sample compile cataloged procedure additions

Include the following in the compile step of your cataloged procedure. Cataloged procedures are included in the IBM-supplied samples for your z/OS system.

Add the following to the CPARM parameter:

```
CPARM='DEF(IBM390)'
```

Add the following statement as the first //SYSLIB DD statement.

```
//SYSLIB DD DSN=SEZACMAC,DISP=SHR
```

Note: If you do not run CPP and your C source file includes either socket.h or rpc.h, you must copy the NCS versions of these files (ncssock1 and ncsrpc) from SEZACMAC to *user_id*.HDRS and rename them to socket and rpc. *user_id*.HDRS must then be specified on the SYSLIB statement ahead of SEZACMAC.

```
//SYSLIB DD DSN=user_id.HDRS,DISP=SHR
           DD DSN=SEZACMAC,DISP=SHR
```

Sample link-edit cataloged procedure additions

Include the following in the link-edit step of your cataloged procedure.

- Add the following statements as the first //SYSLIB DD statement:

```
//      DD DSN=SEZALIBN,DISP=SHR
//      DD DSN=SEZACMTX,DISP=SHR
```

- Add the following //USERLIB DD statement:

```
//USERLIB DD DSN=user_id.OBJ,DISP=SHR
```

All entry points are not defined as external references in SEZALIBN. You must include the following when you link-edit your application code.

```
INCLUDE SYSLIB(RPC0S)
INCLUDE SYSLIB(RPC0SEQ)
INCLUDE SYSLIB(RPC0UTIL)
INCLUDE SYSLIB(SOCKET)
```

- Create a member SAMPLE of partitioned data set *user_id*.LOADLIST and add the necessary objects to link to SAMPLE.

For example, to create SAMPLE load module with three objects (SAMPLE, SAMPLE@C, SAMPLE@W), the corresponding contents of SAMPLE in *user_id.LOADLIST* would be:

```
INCLUDE SYSLIB(RPC@S)
INCLUDE SYSLIB(RPC@SEQ)
INCLUDE SYSLIB(RPC@UTIL)
INCLUDE SYSLIB(SOCKET)
INCLUDE USERLIB(SAMPLE)
INCLUDE USERLIB(SAMPLE@C)
INCLUDE USERLIB(SAMPLE@W)
MODE AMODE(31)
ENTRY CEESTART
```

Note: For more information about compiling and linking, see the *z/OS XL C/C++ User's Guide*.

Running UUID@GEN

The NCS program UUID@GEN generates universal unique identifiers. The UUID@GEN data set is a member of SEZALOAD.

For more information about using UUID@GEN, see the *NCS for IBM AIX/ESA Planning and Administration Guide*.

Use the following format to invoke the UUID@GEN.

►►—UUID@GEN—◄◄

NCS sample programs

The source code for the following NCS sample programs is included in SEZAINST:

- BANK
- NCSSMP
- BINOP

See “Compiling and linking NCS programs” on page 346 for step-by-step instructions on compiling, link-editing, and running the sample programs. For specific instructions on building and running each sample, see “Compiling, linking, and running the sample BINOP program” on page 349, “Compiling, linking, and running the NCSSMP program” on page 353, and “Compiling, linking, and running the sample BANK program” on page 358.

Implement the BINOP sample program on your system, then run either the NCSSMP program or BANK. BINOP uses a well-known port rather than the NCS location broker. The BINOP sample program can help verify NCS on your system.

When running the NIDL compiler against any of the sample program IDL data sets, ensure that you specify the include data set. For example, to run NIDL against the BANK.IDL data set, enter the following:

```
RUNNIDL BANK IDL inc (bankdefs)
```

The NCSSMP sample program

The following is an example of an NCS sample program. It includes the following program segments:

- NCS redefines for this sample program
- Instructions to compile and run the sample program on MVS

The source code for the following program segments are included in SEZAINST:

- NCSSERV1 (NCS server)
- NCSCCLNT1 (NCS client)
- NCSSMPI (NCS NIDL interface)

NCS sample redefines

The following is an example of a redefine data set that is needed if this NCS sample program is to run on MVS:

```
/******  
*                               Redefines for NCS Sample Program                               *  
*   On IBM VM or MVS operating systems external references longer than 8 characters must be *  
*   redefined to 8 characters or less. *  
*   This file must be included in the Sample Programs and stubs. *  
*****/  
  
#define binop_server_epv binSEpv  
#define binop_client_epv binCEpv  
#define binop_add binAdd  
#define getNCShandle binGtHnd
```

Compiling, linking, and running the sample BINOP program

The NCS sample program BINOP consists of the following data sets, which are members of SEZAINST:

Sample data set

	Description
BINOPR	Describes how to run the BINOP sample program.
BINOPSC	Contains C source code for the BINOP server program.
BINOPCC	Contains C source code for the BINOP client program.
BINOP	Contains C source code for the BINOP remote subroutine.
BINOPI	Contains the interface definition language data set for BINOP sample programs used as input to the NIDL compiler.
BINDEFS	Indicates the header data set containing the redefines of external references, greater than 8 characters in length, used in the BINOP sample programs.

The following sections describe steps required to run the sample BINOP program successfully.

- “Setup” on page 350
- “Compile” on page 351
- “Link” on page 352
- “Run” on page 353

Note: If you have a problem with any of these steps, you must resolve them before you can go on to the next step. If you encounter a problem, first ensure that TCP/IP for MVS or z/OS CS has been installed and is operational on your system.

Setup

Before you begin: You need to know how to access data sets and copy files.

Perform the following steps as prerequisites to compiling, linking, and running the sample BINOP program.

1. Copy the sample data sets from SEZAINST to your user ID.

From location	To location
SEZAINST(BINOP)	<i>user_id.binop.c</i>
SEZAINST(BINOPCC)	<i>user_id.binopc.c</i>
SEZAINST(BINOPSC)	<i>user_id.binops.c</i>
SEZAINST(BINDEFS)	<i>user_id.bindefs.h</i>
SEZAINST(BINOPI)	<i>user_id.binop.idl</i>

2. Copy the imported data sets from SEZAINST to your user ID.

From location	To location
SEZAINST(BASEI)	<i>user_id.base.idl</i>
SEZAINST(NBASEI)	<i>user_id.nbase.idl</i>
SEZAINST(RPCI)	<i>user_id.rpc.idl</i>

3. To generate stubs, run NIDL using the following command:

```
RUNNIDL BINOP IDL INC(BINDEFS)
```

4. Copy the included C header files to your user ID.

From location	To location
SEZACMAC(BASE)	<i>user_id.base.h</i>
SEZACMAC(NBASE)	<i>user_id.nbase.h</i>
SEZACMAC(NCSDEFS)	<i>user_id.ncsdefs.h</i>
SEZACMAC(TYPES)	<i>user_id.types.h</i>
SEZACMAC(BSDTIME)	<i>user_id.bsdtime.h</i>
SEZACMAC(BSDTOCMS)	<i>user_id.bsdtocms.h</i>
SEZACMAC(BSDTYPES)	<i>user_id.bsdtypes.h</i>
SEZACMAC(IDL@BASE)	<i>user_id.idl@base.h</i>
SEZACMAC(PFM)	<i>user_id.pfm.h</i>
SEZACMAC(NCSRPC)	<i>user_id.rpc.h</i>
'C' library	<i>user_id.setjmp.h</i>
'C' library	<i>user_id.stdio.h</i>
'C' library	<i>user_id.time.h</i>

Note: C library header files depend on the compiler you are using. For example:

- C370 2.2
- AD/Cycle C/370

-
5. You must run CPP to change \$ to _ before you can compile this code. To run CPP, enter the following commands:

```
RUNCPP BINOPS C
RUNCPP BINOPC C
RUNCPP BINOP@S C@SSTUB
RUNCPP BINOP@C C@CSTUB
RUNCPP BINOP@W C@CSWTCH
RUNCPP BINOP C
```

You know you are done when RUNCPP completes with no errors.

Compile

Before you begin: You need to have completed the steps in “Setup” on page 350.

You can use several methods to compile, link-edit, and execute your program in MVS. The following explains how to compile your C data sets generated by RUNCPP under MVS batch, using IBM-supplied cataloged procedures.

The following list contains data set names, which are used as examples in the following JCL statements:

Data set name Contents

user_id.OBJ A partitioned data set that contains the compiled versions of C programs as its members.

user_id.LOADLIST A partitioned data set that contains the loadlist as its members.

user_id.LOAD A partitioned data set that contains the link-edited versions of C programs as its members.

In order for the program to compile correctly, you must make changes to the EDCC cataloged procedure, which is supplied with IBM C for zSeries® Compiler Licensed Program (5688-187).

Perform the following steps to compile your program.

1. Remove the OUTFILE and OUTDCB parameters.

-
2. Add the following to the CPARM parameter:

```
CPARM= 'DEF(IBM CPP,IBM370) ',
```

3. Replace the //SYSIN DD statement and the //SYSLIN statement with the following:

```
//SYSIN DD DSN=user_id..&INFILE..CPPOUT,DISP=SHR
//SYSLIN DD DSN=user_id..OBJ(&MEM),DISP=SHR
```

4. Add the following //SYSLIB DD statement:

```
//SYSLIB DD DSN=SEZACMAC,DISP=SHR
```

-
5. Submit the compile job at the Spool Display and Search Facility (SDSF) command panel, by entering the following:

```
/s EDCC,INFILE=BINOPS
/s EDCC,INFILE=BINOPC
/s EDCC,INFILE=BINOP0S
/s EDCC,INFILE=BINOP0C
/s EDCC,INFILE=BINOP0W
/s EDCC,INFILE=BINOP
```

You know you are done when no errors are received.

Link

Before you begin: You need to have completed the steps in “Setup” on page 350 and “Compile” on page 351.

In order for the program to link correctly, you must make changes to the EDCL cataloged procedure, which is supplied with IBM C for zSeries Compiler Licensed Program (5688-187).

Perform the following steps to link-edit your program.

1. Remove the OUTFILE parameter.

-
2. Add the following statements after the //SYSLIB DD statement:

```
//      DD  DSN=SEZALIBN,DISP=SHR
//      DD  DSN=SEZACMTX,DISP=SHR
```

3. Add the following //USERLIB DD statement:

```
//USERLIB DD DSN=user_id.OBJ,DISP=SHR
```

4. Replace the //SYSLIN DD statement with the following:

```
//SYSLIN DD DSN=user_id.OBJ(&MEM),DISP=SHR
//      DD DSN=user_id.LOADLIST(&MEM),DISP=SHR
```

5. Include the following lines when you link-edit your application code, because not all entry points are defined as external references in SEZALIBN.

```
INCLUDE SYSLIB(RPC0S)
INCLUDE SYSLIB(RPC0SEQ)
INCLUDE SYSLIB(RPC0UTIL)
INCLUDE SYSLIB(SOCKET)
```

6. Replace the //SYSLMOD DD statement with the following:

```
//SYSLMOD DD DSN=user_id.LOAD(&MEM),DISP=SHR
```

7. Create one member of the partitioned data set *user_id.LOADLIST*, by adding the following lines to the data set BINOPC.

```
INCLUDE SYSLIB(RPC0S)
INCLUDE SYSLIB(RPC0SEQ)
INCLUDE SYSLIB(RPC0UTIL)
INCLUDE SYSLIB(SOCKET)
```

```

INCLUDE USERLIB(BINOP@C)
INCLUDE USERLIB(BINOP@W)
MODE AMODE(31)
ENTRY CEESTART

```

8. Create a second member of the partitioned data set *user_id.LOADLIST*, by adding the following lines to the data set BINOPS.

```

INCLUDE SYSLIB(RPC@S)
INCLUDE SYSLIB(RPC@SEQ)
INCLUDE SYSLIB(RPC@UTIL)
INCLUDE SYSLIB(SOCKET)
INCLUDE USERLIB(BINOP@S)
INCLUDE USERLIB(BINOP)
MODE AMODE(31)
ENTRY CEESTART

```

9. Submit the link-edit job at the SDSF command panel, by entering the following:

```

/s EDCL,MEM=BINOPC
/s EDCL,MEM=BINOPS

```

You know you are done when no errors are received.

Run

Before you begin: You need to have completed the steps in “Setup” on page 350, “Compile” on page 351, and “Link” on page 352.

Perform the following steps to run your program.

1. Start the NCS server sample program on one MVS user ID by entering the following command:

```
CALL 'user_id.LOAD(BINOPS)' '2'
```

2. Start the NCS client on a different MVS user ID by entering the following command:

```
CALL 'user_id.LOAD(BINOPC)' 'hostname 2 3'
```

where *hostname* is the name of the system that the server is running on.

You know you are done when the program runs successfully.

Compiling, linking, and running the NCSSMP program

The NCSSMP sample program consists of the following data sets, which are members of SEZAINST:

NCSSMPR	Describes the NCS sample program.
NCSSERV1	Contains C source code for the server for the NCS sample program.
NCSCCNT1	Contains C source code for the client for the NCS sample program.
NCSSMPI	Contains the interface definition language data set for the NCS sample program used as input to the NIDL compiler.

NSMPDEFS Indicates the header data set containing the redefines of external references, greater than 8 characters in length, used in the NCS sample program.

For an example of the source code, see “The NCSSMP sample program” on page 349.

The following sections describe steps required to run the NCSSMP program successfully.

- “Setup”
- “Compile” on page 355
- “Link” on page 356
- “Run” on page 357

Note: If you have a problem with any of these steps, you must resolve them before you can go on to the next step. If you encounter a problem, first ensure that TCP/IP for MVS or z/OS CS has been installed and is operational on your system. Also, ensure that the NCS Global Location Broker is running somewhere on your network.

Setup

Before you begin: You need to know how to access data sets and copy files.

Perform the following steps as prerequisites to compiling, linking, and running the NCSSMP program.

1. Copy the sample data sets from SEZAINST to your user ID.

From location	To location
SEZAINST(NCSSERV1)	<i>user_id.ncsserv1.c</i>
SEZAINST(NCSCSCLNT1)	<i>user_id.ncscsclnt1.c</i>
SEZAINST(NCSSMPI)	<i>user_id.ncssmp.idl</i>
SEZAINST(NSMPDEFS)	<i>user_id.nsmptdefs.h</i>

2. Copy the imported data sets from SEZAINST to your user ID.

From location	To location
SEZAINST(RPCI)	<i>user_id.rpc.idl</i>
SEZAINST(BASEI)	<i>user_id.base.idl</i>
SEZAINST(NBASEI)	<i>user_id.nbase.idl</i>

3. To generate stubs, run NIDL using the following command:

```
RUNNIDL NCSSMP IDL INC(nsmptdefs)
```

4. Copy the data sets included by CPP to your user ID.

From location	To location
SEZACMAC(NCSDEFS)	<i>user_id.ncsdefs.h</i>
SEZACMAC(BSDTOCMS)	<i>user_id.bsdtocms.h</i>

From location	To location
SEZACMAC(BASE)	<i>user_id</i> .base.h
SEZACMAC(IDL@BASE)	<i>user_id</i> .idl@base.h
SEZACMAC(NBASE)	<i>user_id</i> .nbase.h
SEZACMAC(LB)	<i>user_id</i> .lb.h
SEZACMAC(GLB)	<i>user_id</i> .glb.h
SEZACMAC(TYPES)	<i>user_id</i> .types.h
SEZACMAC(BSDTYPES)	<i>user_id</i> .bsdtypes.h
SEZACMAC(BSDTIME)	<i>user_id</i> .bsdtime.h
SEZACMAC(PFM)	<i>user_id</i> .pfm.h
C library	<i>user_id</i> .stdio.h
C library	<i>user_id</i> .setjmp.h
Note: C library header files depend on the compiler you are using. For example: • C370 2.2 • AD/Cycle C/370	

-
5. You must run CPP to change \$ to _ before you can compile this code. To run CPP, enter the following commands:

```
RUNCPP NCSSERV1 C
RUNCPP NCSCNT1 C
RUNCPP NCSSMP@S C@SSTUB
RUNCPP NCSSMP@C C@CSTUB
RUNCPP NCSSMP@W C@CSWTCH
```

You know you are done when RUNCPP completes with no errors.

Compile

Before you begin: You need to have completed the steps in “Setup” on page 354.

You can use several methods to compile, link-edit, and execute your program in MVS. This section explains how to compile your C data sets generated by RUNCPP under MVS batch, using IBM-supplied cataloged procedures.

The following list contains data set names, which are used as examples in the following JCL statements:

user_id.OBJ A partitioned data set that contains the compiled versions of C programs as its members.

user_id.LOADLIST A partitioned data set that contains the loadlist as its members.

user_id.LOAD A partitioned data set that contains the link-edited versions of C programs as its members.

In order for the program to compile correctly, you must make changes to the EDCC cataloged procedure, which is supplied with IBM C for zSeries, Compiler Licensed Program (5688-187).

Perform the following steps to compile your program.

1. Remove the OUTFILE and OUTDCB parameters.

2. Add the following to the CPARM parameter:

```
CPARM= 'DEF (IBMCPP,IBM370) ',
```

3. Replace the //SYSIN DD statement and the //SYSLIN statement with the following:

```
//SYSIN DD DSN=user_id..&MEM..CPPOUT,DISP=SHR
//SYSLIN DD DSN=user_id..OBJ(&MEM),DISP=SHR
```

4. Add the following //SYSLIB DD statement:

```
//SYSLIB DD DSN=SEZACMAC,DISP=SHR
```

5. Submit the compile job at the Spool Display and Search Facility (SDSF) command panel, by entering the following:

```
/s EDCC,MEM=NCSSERV1
/s EDCC,MEM=NCSCINT1
/s EDCC,MEM=NCSSMP@S
/s EDCC,MEM=NCSSMP@C
/s EDCC,MEM=NCSSMP@W
```

You know you are done when no errors are received.

Link

Before you begin: You need to have completed the steps in “Setup” on page 354 and “Compile” on page 355.

In order for the program to link correctly, you must make changes to the EDCL cataloged procedure, which is supplied with IBM C for zSeries, Compiler Licensed Program (5688-187).

Perform the following steps to link-edit your program.

1. Remove the OUTFILE parameter.

2. Add the following statements after the //SYSLIB DD statement:

```
// DD DSN=SEZALIBN,DISP=SHR
// DD DSN=SEZACMTX,DISP=SHR
```

3. Add the following //USERLIB DD statement:

```
//USERLIB DD DSN=user_id.OBJ,DISP=SHR
```

4. Replace the //SYSLIN DD statement with the following:

```
//SYSLIN DD DSN=user_id.OBJ(&MEM),DISP=SHR
// DD DSN=user_id.LOADLIST(&MEM),DISP=SHR
```

5. Include the following when you link-edit your application code, because not all entry points are defined as external references in SEZALIBN.


```

INCLUDE SYSLIB(RPC@S)
INCLUDE SYSLIB(RPC@SEQ)
INCLUDE SYSLIB(RPC@UTIL)
INCLUDE SYSLIB(SOCKET)

```

-
6. Replace the //SYSLMOD DD statement with the following:

```
//SYSLMOD DD DSN=user_id.LOAD(&MEM),DISP=SHR
```

7. Create one member of the partitioned data set *userid.LOADLIST* by adding the following lines to the data set NCSCSNT1.

```

INCLUDE SYSLIB(RPC@S)
INCLUDE SYSLIB(RPC@SEQ)
INCLUDE SYSLIB(RPC@UTIL)
INCLUDE SYSLIB(SOCKET)
INCLUDE USERLIB(NCSSMP@C)
INCLUDE USERLIB(NCSSMP@W)
MODE AMODE(31)
ENTRY CEESTART

```

8. Create a second member of the partitioned data set *userid.LOADLIST* by adding the following lines to the data set NCSSERV1.

```

INCLUDE SYSLIB(RPC@S)
INCLUDE SYSLIB(RPC@SEQ)
INCLUDE SYSLIB(RPC@UTIL)
INCLUDE SYSLIB(SOCKET)
INCLUDE USERLIB(NCSSMP@S)
MODE AMODE(31)
ENTRY CEESTART

```

9. Submit the link-edit job at the SDSF command panel, by entering the following:

```

/s EDCL,MEM=NCSCSNT1
/s EDCL,MEM=NCSSERV1

```

You know you are done when no errors are received.

Run

Before you begin: You need to have completed the steps in “Setup” on page 354, “Compile” on page 355, and “Link” on page 356.

Perform the following steps to run your program.

1. Make sure that the Local and Global Location Brokers are running.
-

2. Start the NCS server sample program on one MVS user ID by entering the following command:

```
CALL 'user_id.LOAD(NCSSERV1)'
```

3. Start the NCS client on a different MVS user ID by entering the following command:

```
CALL 'user_id.LOAD(NCSCSNT1)' '5 32'
```

You know you are done when the program runs successfully.

Compiling, linking, and running the sample BANK program

The NCS sample program BANK consists of the following data sets, which are members of SEZAINST:

Sample data set	Description
BANKR	Describes how to run the BANK sample program.
BANKDC	Contains C language source code for the BANK server program.
BANKC	Contains C language source code for the BANK client program.
UTILC	Contains utility routines used by the BANK server and client programs.
UTILH	Indicates a header data set used in the BANK sample program.
UIDBIND	Contains autobind and unbind source code routines used by the BANK server and client programs.
BANKIDL	Contains the interface definition language data set for the BANK sample programs used as input to the NIDL compiler.
SHAWMUT	Contains input data for BANK server program.
BAYBANKS	Contains input data for BANK server program.
BANKDEFS	Indicates a header data set containing the redefines of external references, greater than 8 characters in length, used in the BANK sample programs.

The following sections describe steps required to run the sample BANK program successfully.

- “Setup”
- “Compile” on page 360
- “Link” on page 361
- “Run” on page 362

Note: If you have a problem with any of these steps, you must resolve them before you can go on to the next step. If you encounter a problem, first ensure that TCP/IP for MVS or z/OS Communications Server has been installed and is operational on your system. Also, ensure that the NCS Global Location Broker is running somewhere on your network and the Local Location Broker is running on the client system.

Setup

Before you begin: You need to know how to access data sets and copy files.

Perform the following steps as prerequisites to compiling, linking, and running the BANK program.

1. Copy the sample data sets from SEZAINST to your user ID.

From location	To location
SEZAINST(BANKDC)	<i>user_id.bankd.c</i>
SEZAINST(BANKC)	<i>user_id.bank.c</i>
SEZAINST(UTILC)	<i>user_id.util.c</i>
SEZAINST(UUIDBIND)	<i>user_id.uuidbind.c</i>
SEZAINST(UTILH)	<i>user_id.util.h</i>
SEZAINST(BANKIDL)	<i>user_id.bank.idl</i>
SEZAINST(SHAWMUT)	<i>user_id.shawmut.bank</i>
SEZAINST(BAYBANK)	<i>user_id.baybank.bank</i>
SEZAINST(BANKDEFS)	<i>user_id.bankdefs.h</i>

-
2. Copy the data sets imported by IDL from SEZAINST to your user ID.

From location	To location
SEZAINST(BASEI)	<i>user_id.base.idl</i>
SEZAINST(NBASEI)	<i>user_id.nbase.idl</i>
SEZAINST(RPCI)	<i>user_id.rpc.idl</i>

-
3. To generate stubs, run NIDL using the following command:

```
RUNNIDL BANK IDL INC(bankdefs)
```

-
4. Copy the data sets included by CPP to your user ID.

From location	To location
SEZACMAC(NCSDEFS)	<i>user_id.ncsdefs.h</i>
SEZACMAC(BSDTOCMS)	<i>user_id.bsdtocms.h</i>
SEZACMAC(BASE)	<i>user_id.base.h</i>
SEZACMAC(IDL@BASE)	<i>user_id.idl@base.h</i>
SEZACMAC(NBASE)	<i>user_id.nbase.h</i>
SEZACMAC(LB)	<i>user_id.lb.h</i>
SEZACMAC(GLB)	<i>user_id.glb.h</i>
SEZACMAC(TYPES)	<i>user_id.types.h</i>
SEZACMAC(BSDTYPES)	<i>user_id.bsdtypes.h</i>
SEZACMAC(BSDTIME)	<i>user_id.bsvertime.h</i>
SEZACMAC(PFM)	<i>user_id.pfm.h</i>
SEZACMAC(UUID)	<i>user_id.uuid.h</i>
'C' library	<i>user_id.stdio.h</i>
'C' library	<i>user_id.setjmp.h</i>
'C' library(ERRNO)	<i>user_id.errno.h</i>
'C' library(TIME)	<i>user_id.time.h</i>

From location	To location
Note: 'C' library header files depend on the compiler you are using. For example: • C370 2.2 • AD/Cycle C/370	

5. You must run CPP to change \$ to _ before you can compile this code. To run CPP, enter the following commands:

```

RUNCPP UTIL C
RUNCPP UIDBIND C
RUNCPP BANKD C
RUNCPP BANK C
RUNCPP BANK@S C@SSTUB
RUNCPP BANK@C C@CSTUB
RUNCPP BANK@W C@CSWTCH

```

You know you are done when RUNCPP completes with no errors.

Compile

Before you begin: You need to have completed the steps in “Setup” on page 358.

You can use several methods to compile, link-edit, and execute your program in MVS. This section explains how to compile your C data sets generated by RUNCPP under MVS batch, using IBM-supplied cataloged procedures.

The following list contains data set names, which are used as examples in the following JCL statements:

Data set name	Contents
<i>user_id</i> .OBJ	A partitioned data set that contains the compiled versions of C programs as its members.
<i>user_id</i> .LOADLIST	A partitioned data set that contains the loadlist as its members.
<i>user_id</i> .LOAD	A partitioned data set that contains the link-edited versions of C programs as its members.

In order for the program to compile correctly, you must make changes to the EDCC cataloged procedure, which is supplied with IBM C for zSeries, Compiler Licensed Program (5688-187).

Perform the following steps to compile your program.

1. Remove the OUTFILE and OUTDCB parameters.

2. Add the following to the CPARM parameter:

```
CPARM='DEF(IBMCP,IBM370)',
```

3. Replace the //SYSIN DD statement and the //SYSLIN statement with the following:

```

//SYSIN DD DSN=user_id..&MEM..CPPOUT,DISP=SHR
//SYSLIN DD DSN=user_id..OBJ(&MEM),DISP=SHR

```

4. Add the following `//SYSLIB DD` statement:

```
//SYSLIB DD DSN=SEZACMAC,DISP=SHR
```

5. Submit the compile job at the Spool Display and Search Facility (SDSF) command panel, by entering the following:

```
/s EDCC,MEM=BANKD
/s EDCC,MEM=BANK
/s EDCC,MEM=BANK@S
/s EDCC,MEM=BANK@C
/s EDCC,MEM=BANK@W
/s EDCC,MEM=UTIL
/s EDCC,MEM=UUIDBIND
```

You know you are done when no errors are received.

Link

Before you begin: You need to have completed the steps in “Setup” on page 358 and “Compile” on page 360.

In order for the program to link correctly, you must make changes to the EDCL cataloged procedure, which is supplied with IBM C for zSeries, Compiler Licensed Program (5688-187).

Perform the following steps to link-edit your program.

1. Remove the `OUTFILE` parameter.

2. Add the following statements after the `//SYSLIB DD` statement:

```
//      DD DSN=SEZALIBN,DISP=SHR
//      DD DSN=SEZACMTX,DISP=SHR
```

3. Add the following `//USERLIB DD` statement:

```
//USERLIB DD DSN=user_id.OBJ,DISP=SHR
```

4. Replace the `//SYSLIN DD` statement with the following:

```
//SYSLIN DD DSN=user_id.OBJ(&MEM),DISP=SHR
//      DD DSN=user_id.LOADLIST(&MEM),DISP=SHR
```

5. Include the following when you link-edit your application code, because not all entry points are defined as external references in `SEZALIBN`.

```
INCLUDE SYSLIB(RPC@S)
INCLUDE SYSLIB(RPC@SEQ)
INCLUDE SYSLIB(RPC@UTIL)
INCLUDE SYSLIB(SOCKET)
```

6. Replace the `//SYSLMOD DD` statement with the following:

```
//SYSLMOD DD DSN=user_id.LOAD(&MEM),DISP=SHR
```

7. Create one member of the partitioned data set `userid.LOADLIST` by adding the following lines to the data set `BANK`:

```

INCLUDE SYSLIB(RPC@S)
INCLUDE SYSLIB(RPC@SEQ)
INCLUDE SYSLIB(RPC@UTIL)
INCLUDE SYSLIB(SOCKET)
INCLUDE USERLIB(BANK@C)
INCLUDE USERLIB(BANK@W)
INCLUDE USERLIB(UTIL)
INCLUDE USERLIB(UUIDBIND)
MODE AMODE(31)
ENTRY CEESTART

```

8. Create a second member of the partitioned data set *userid.LOADLIST* by adding the following lines to the data set *BANKD*:

```

INCLUDE SYSLIB(RPC@S)
INCLUDE SYSLIB(RPC@SEQ)
INCLUDE SYSLIB(RPC@UTIL)
INCLUDE SYSLIB(SOCKET)
INCLUDE USERLIB(BANK@S)
INCLUDE USERLIB(UTIL)
INCLUDE USERLIB(UUIDBIND)
MODE AMODE(31)
ENTRY CEESTART

```

9. Submit the link-edit job at the SDSF command panel, by entering the following:

```

/s EDCL,MEM=BANK
/s EDCL,MEM=BANKD

```

You know you are done when no errors are received.

Run

Before you begin: You need to have completed the steps in “Setup” on page 358, “Compile” on page 360, and “Link” on page 361.

Perform the following steps to run your program.

1. Make sure that the Local and Global Location Brokers are running.
-
2. Start the NCS server sample program on one MVS user ID. To do so, enter the following command:


```
CALL 'userid.LOAD(BANKD)' 'ip shawmut shawmut.bank' asis
```
-
3. Start the NCS client on a different MVS user ID. To do so, enter the following command:


```
CALL 'userid.LOAD(BANK)' 'inquire shawmut Leach' asis
```
-

You know you are done when the program runs successfully.

Chapter 11. Running the Sample Mail Filter program

This topic explains how to run the sample mail filter program, `lf_smpl.c`. A mail filter is designed to provide more functionality for a sendmail daemon. These functions might include adding a recipient, scanning for viruses, rejecting a disallowed recipient address, and so on. This sample mail filter creates a file in `/tmp` named `msg.XXXXXXXX` (where `X` represents any combination of letters and numbers) to log the message body and headers.

Compiling and linking the `lf_smpl.c` source code

The following source is needed to compile and link the sample filter:

- `/usr/lpp/tcpip/samples/sendmail/milter/lf_smpl.c` - sample filter program
- `/usr/include/libmilter/mfapi.h` - header file needed for `lf_smpl.c`
- `/usr/include/libmilter/mfdef.h` - header file needed for `lf_smpl.c`
- `/usr/lib/libmilter.a` - milter API library

The sample filter program is documented in the end of `/usr/lpp/tcpip/samples/sendmail/libmilter/README`.

Note: The milter API library `libmilter.a` is built in the IBM-1047 environment. The sample filter must be compiled and linked in IBM-1047 to assure correct data exchange between the sample filter and the milter API. The sample filter also must be executed in the environment with codepage IBM-1047.

The following example shows how to use the `cc` command to compile and link the sample filter.

```
cc -I. -o filter lf_smpl.c libmilter.a
```

Specifying filters in the sendmail configuration file

To use filters in sendmail, filters must be declared in the sendmail configuration file (`sendmail.cf`). For more information about this sendmail configuration file, see the *z/OS Communications Server: IP Configuration Guide*.

Running the sample program

Error messages for the sample filter are written to a log file. The log file is defined in `lf_smpl.c` as follows:

```
openlog(NULL, LOG_PID, LOG_LOCAL7)
```

To get error messages, first create a log file. The log reference in the source code can be modified to reference the log file you created. For more information about error messages, see *z/OS Communications Server: IP Diagnosis Guide*.

The `lf_smpl.c` sample program accepts the `-p` argument as follows:

-p socket_reference

Specifying the socket information of the filter, the parameter should be formatted according to the socket specification in the sendmail configuration file.

For example, use the command filter `-p inet:3333@localhost` with the following configuration:

```
Xfilter, S=inet:3333@localhost
0 InputMailFilters=filter
```

Library control functions

smfi_register

```
#include <libmilter/mfapi.h>
int smfi_register(
    smfiDesc_str descr
);
```

Description

Register a set of filter callbacks. When called, `smfi_register` creates a filter using the information given in the `smfiDesc_str` argument.

Notes:

1. `smfi_register` must be called before `smfi_main`.
2. Multiple calls to `smfi_register` within a single process are not allowed.

Parameters

descr A filter descriptor of type `smfiDesc_str` describing the filter's functions. The structure has the following members:

```
struct smfiDesc
{
    char *xxfi_name; /* filter name */
    int xxfi_version; /* version code -- do not change */
    unsigned long xxfi_flags; /* flags */

    /* connection info filter */
    sfsistat (*xxfi_connect)(SMFICTX *, char *, _SOCK_ADDR *);
    /* SMTP HELO command filter */
    sfsistat (*xxfi_helo)(SMFICTX *, char *);
    /* envelope sender filter */
    sfsistat (*xxfi_envfrom)(SMFICTX *, char **);
    /* envelope recipient filter */
    sfsistat (*xxfi_envrcpt)(SMFICTX *, char **);
    /* header filter */
    sfsistat (*xxfi_header)(SMFICTX *, char *, char *);
    /* end of header */
    sfsistat (*xxfi_eoh)(SMFICTX *);
    /* body block */
    sfsistat (*xxfi_body)(SMFICTX *, unsigned char *, size_t);
    /* end of message */
    sfsistat (*xxfi_eom)(SMFICTX *);
    /* message aborted */
    sfsistat (*xxfi_abort)(SMFICTX *);
    /* connection cleanup */
    sfsistat (*xxfi_close)(SMFICTX *);
};
```

A NULL value for any callback function indicates that the filter does not process the given type of information, simply returning `SMFIS_CONTINUE`.

Result

`smfi_register` can return `MI_FAILURE` for any of the following reasons:

- Memory allocation failed.

- Incompatible version or illegal flags value.

The `xxfi_flags` field should contain the bitwise OR of zero or more of the following values, describing the actions the filter might take:

SMFIF_ADDHDRS

This filter can add headers.

SMFIF_CHGHDRS

This filter can change and delete headers.

SMFIF_CHGBODY

This filter can replace the body during filtering. This can have significant performance impact if other filters do body filtering after this filter.

SMFIF_ADDRcpt

This filter can add recipients to the message.

SMFIF_DELRcpt

This filter can remove recipients from the message.

smfi_setconn

```
#include <libmilter/mfapi.h>
int smfi_setconn(
    char *oconn;
);
```

Description

The `smfi_setconn` API sets the socket through which the filter communicates with sendmail. The `smfi_setconn` API must be called once before `smfi_main`.

Parameters

oconn The address of the desired communication socket. The address should be a NULL-terminated string in `proto:address` format as follows:

- `{unix|local}:/path/to/file` - A named pipe
- `inet:port@{hostname|ip-address}` - An IPv4 socket
- `inet6:port@{hostname|ip-address}` - An IPv6 socket

Result

`smfi_setconn` does not fail on an address that is not valid. A failure is detected only in `smfi_main`.

Notes:

1. If possible, filters should not run as root when communicating over UNIX/local domain sockets.
2. UNIX/local sockets should have their permissions set to 0600 (read/write permission only for the socket's owner).

smfi_settimeout

```
#include <libmilter/mfapi.h>
int smfi_settimeout(
    int otimeout
);
```

Description

Sets the number of seconds libmilter will wait for an MTA connection before timing out a socket. If `smfi_settimeout` is not called, a default timeout of 1800 seconds is used.

Note: The `smfi_settimeout` API should be called only before `smfi_main`.

Parameters

`otimeout`

The number of seconds to wait before timing out (value must be greater than zero). Zero means no wait, rather than wait forever.

Result

`smfi_settimeout` always returns `MI_SUCCESS`.

`smfi_main`

```
#include <libmilter/mfapi.h>
int smfi_main(
);
```

Description

The `smfi_main` API is called after a filter's initialization is complete. `smfi_main` passes control to the milter event loop.

Parameters

`smfi_main` has no parameters.

Result

`smfi_main` returns `MI_FAILURE` if it fails to establish a connection. This can occur for a number of reasons (for instance, if an address that is not valid is passed to `smfi_setconn`). The reason for the failure is logged. Otherwise, `smfi_main` returns `MI_SUCCESS`.

Data access functions

`smfi_getsymval`

```
#include <libmilter/mfapi.h>
char* smfi_getsymval(
    SMFICTX *ctx,
    char *symname
);
```

Description

Get the value of a sendmail macro. `smfi_getsymval` can be called from within any of the `xxfi_*` callbacks. Which macros are defined depends on when `smfi_getsymval` is called.

Parameters

ctx The opaque context structure.

symname

The name of a sendmail macro, optionally enclosed in braces (`{` and `}`). See below for default macros.

Result

`smfi_getsymval` returns the value of the given macro as a null-terminated string, or returns `NULL` if the macro is not defined.

Notes:

1. By default, the following macros are valid in the given contexts:

Sent With	Macros
xxfi_connect	daemon_name, if_name, if_addr, j, _
xxfi_helo	tls_version, cipher, cipher_bits, cert_subject, cert_issuer
xxfi_envfrom	i, auth_type, auth_authen, auth_ssf, auth_author, mail_mailer, mail_host, mail_addr
xxfi_envrcpt	rcpt_mailer, rcpt_host, rcpt_addr

2. All macros remain in effect from the point they are received until the end of the connection for the first two sets, the end of the message for the third set (xxfi_envfrom), and for each recipient for the final set (xxfi_envrcpt).
3. The macro list can be changed using the confMILTER_MACROS_* options in sendmail.mc. The scopes of such macros are determined by when they are set by sendmail. For descriptions of macro values, see the *Sendmail Installation and Operation Guide* provided with the sendmail distribution.

smfi_getpriv

```
#include <libmilter/mfapi.h>
void* smfi_getpriv(
    SMFICTX *ctx
);
```

Description

Get the connection-specific data pointer for this connection. smfi_getpriv can be called in any of the xxfi_* callbacks.

Parameters

ctx Opaque context structure.

Result

smfi_getpriv returns the private data pointer stored by a prior call to smfi_setpriv, or returns NULL if none has been set.

smfi_setpriv

```
#include <libmilter/mfapi.h>
int smfi_setpriv(
    SMFICTX *ctx,
    void *privatedata
);
```

Description

Set the private data pointer for this connection.

Parameters

ctx Opaque context structure.

privatedata

Pointer to private data. This value is returned by subsequent calls to smfi_getpriv using ctx.

Result

smfi_setpriv returns MI_FAILURE if the context of ctx is not valid. Otherwise, it returns MI_SUCCESS.

Notes:

1. There is only one private data pointer per connection; multiple calls to smfi_setpriv with different values cause previous values to be lost.
2. Before a filter terminates it should release the private data and set the pointer to NULL.

smfi_setreply

```
#include <libmilter/mfapi.h>
int smfi_setreply(
    SMFICTX *ctx,
    char *rcode,
    char *xcode,
    char *message
);
```

Description

Directly set the SMTP error reply code for this connection. This code is used on subsequent error replies resulting from actions taken by this filter. smfi_setreply can be called from any of the xxfi_ callbacks.

Parameters

ctx Opaque context structure.

rcode The three-digit SMTP reply code, as a null-terminated string. rcode cannot be NULL and must be a valid reply code.

xcode The extended reply code. If xcode is NULL, no extended code is used.

message

The text part of the SMTP reply. If message is NULL, an empty message is used.

Result

smfi_setreply fails and returns MI_FAILURE if:

- The rcode or xcode argument is not valid.
- A memory-allocation failure occurs.

Otherwise, it returns MI_SUCCESS.

Notes:

1. Values passed to smfi_setreply are not checked for standards compliance.
2. For details about reply codes and their meanings, see the documentation provided with the sendmail distribution.

Message modification functions

smfi_addheader

```
#include <libmilter/mfapi.h>
int smfi_addheader(
    SMFICTX *ctx,
    char *headerf,
    char *headerv
);
```

Description

The `smfi_addheader` API adds a header to the current message. It is called only from `xxfi_eom`.

Parameters

ctx Opaque context structure.

headerf

The header name, a non-NULL, null-terminated string.

headerv

The header value to be added, a non-NULL, null-terminated string. This can be the empty string.

Result

`smfi_addheader` returns `MI_FAILURE` in the following cases:

- `headerf` or `headerv` is NULL.
- Adding headers in the current connection state is not valid.
- Memory allocation fails.
- A network error occurs.
- `SMFIF_ADDHDRS` was not set when `smfi_register` was called.

Otherwise, it returns `MI_SUCCESS`.

Notes:

1. `smfi_addheader` does not change a message's existing headers. To change a header's current value, use `smfi_chgheader`.
2. A filter that calls `smfi_addheader` must have set the `SMFIF_ADDHDRS` flag in the `smfiDesc_str` passed to `smfi_register`.
3. Filter order is important for `smfi_chgheader`; later filters see the header changes made by previous filters.
4. Neither the name nor the value of the header is checked for standards compliance. However, each line of the header must be less than 2048 characters in length and should be less than 998 characters in length. If longer headers are required, make them multiline. It is the filter writer's responsibility to ensure that no standards are violated.

`smfi_chgheader`

```
#include <libmilter/mfapi.h>
int smfi_chgheader(
    SMFICTX *ctx,
    char *headerf,
    mi_int32 hdridx,
    char *headerv
);
```

Description

The `smfi_chgheader` API changes a header's value for the current message. It is called only from `xxfi_eom`.

Parameters

ctx Opaque context structure.

headerf

The header name, a non-NULL, null-terminated string.

hdridx

Header index value (1-based). A `hdridx` value of 1 modifies the first occurrence of a header named `headerf`. If `hdridx` is greater than the number of times `headerf` appears, a new copy of `headerf` is added.

headerv

The new value of the given header. A `headerv` value of `NULL` implies that the header should be deleted.

Result

`smfi_chgheader` returns `MI_FAILURE` in the following cases:

- `headerf` is `NULL`.
- Modifying headers in the current connection state is not valid.
- Memory allocation fails.
- A network error occurs.
- `SMFIF_CHGHDRS` was not set when `smfi_register` was called.

Otherwise, it returns `MI_SUCCESS`.

Notes:

1. While `smfi_chgheader` can be used to add new headers, it is more efficient and safer to use `smfi_addheader` to add new headers.
2. A filter that calls `smfi_chgheader` must have set the `SMFIF_CHGHDRS` flag in the `smfiDesc_str` passed to `smfi_register`.
3. Filter order is important for `smfi_chgheader`; later filters see the header changes made by previous filters.
4. Neither the name nor the value of the header is checked for standards compliance. However, each line of the header must be less than 2048 characters in length and should be less than 998 characters in length. If longer headers are needed, make them multiline. It is the filter writer's responsibility to ensure that no standards are violated.

smfi_addrcpt

```
#include <libmilter/mfapi.h>
int smfi_addrcpt(
    SMFICTX *ctx,
    char *rcpt
);
```

Description

The `smfi_addrcpt` API adds a recipient to the message envelope. It is called only from `xxfi_eom`.

Parameters

ctx Opaque context structure.

rcpt The new recipient's address.

Result

`smfi_addrcpt` fails and returns `MI_FAILURE` if the following conditions occur:

- Adding headers in the current connection state is not valid.
- A network error occurs.
- `SMFIF_ADDRRCPT` was not set when `smfi_register` was called.
- `rcpt` is `NULL`.

Otherwise, `smfi_addrcpt` returns `MI_SUCCESS`.

Note: A filter that calls `smfi_addrcpt` must have set the `SMFIF_ADDRRCPT` flag in the `smfiDesc_str` passed to `smfi_register`.

smfi_delrcpt

```
#include <libmilter/mfapi.h>
int smfi_delrcpt(
    SMFICTX *ctx;
    char *rcpt;
);
```

Description

The `smfi_delrcpt` API removes the named recipient from the current message's envelope. It is called only from `xxfi_eom`.

Parameters

ctx Opaque context structure.

rcpt The recipient address to be removed, a non-NULL, null-terminated string.

Result

`smfi_delrcpt` fails and returns `MI_FAILURE` if any of the following conditions occur:

- `rcpt` is NULL.
- Adding headers in the current connection state is not valid.
- A network error occurs.
- `SMFIF_DELRCP` was not set when `smfi_register` was called.

Otherwise, it returns `MI_SUCCESS`.

Note: The addresses to be removed must match exactly (for example, an address and its expanded form must match).

smfi_replacebody

```
#include <libmilter/mfapi.h>
int smfi_replacebody(
    SMFICTX *ctx,
    unsigned char *bodyp,
    int bodylen
);
```

Description

The `smfi_replacebody` API replaces the body of the current message. It is called only from `xxfi_eom` and can be called more than once. If it is called multiple times, subsequent calls result in data being appended to the new body.

Parameters

ctx Opaque context structure.

bodyp A pointer to the start of the new body data, which does not have to be null-terminated. If `bodyp` is NULL, it is treated as having length equal to 0. Body data should be in CR/LF form.

bodylen

The number of data bytes `bodyp` points to.

Result

smfi_replacebody fails and returns MI_FAILURE if any of the following conditions occur:

- bodyp is equal to NULL and bodylen is greater than 0.
- Changing the body in the current connection state is not valid.
- A network error occurs.
- SMFIF_CHGBODY was not set when smfi_register was called.

Otherwise, it will return MI_SUCCESS.

Notes:

1. Since the message body can be very large, setting SMFIF_CHGBODY might significantly affect filter performance.
2. If a filter sets SMFIF_CHGBODY but does not call smfi_replacebody, the original body remains unchanged.
3. Filter order is important for smfi_replacebody; later filters see the new body contents created by previous filters.

Callbacks

Each of these callbacks should return one of the values defined in Table 6. Any value other than those listed constitutes an error and causes sendmail to terminate its connection to the offending filter.

Table 6. Callback return values

Return value	Description
SMFIS_CONTINUE	Continue processing the current connection, message, or recipient.
SMFIS_REJECT	For a connection-oriented routine, reject this connection; call xxfi_close. For a message-oriented routine (except xxfi_eom or xxfi_abort), reject this message. For a recipient-oriented routine, reject the current recipient (but continue processing the current message).
SMFIS_DISCARD	For a message or recipient-oriented routine, accept this message, but silently discard it. SMFIS_DISCARD should not be returned by a connection-oriented routine.
SMFIS_ACCEPT	For a connection-oriented routine, accept this connection without further filter processing; call xxfi_close. For a message or recipient-oriented routine, accept this message without further filtering.
SMFIS_TEMPFAIL	Return a temporary failure; the corresponding SMTP command will return an appropriate 4xx status code. For a message-oriented routine (except xxfi_envfrom), fail for this message. For a connection-oriented routine, fail for this connection; call xxfi_close. For a recipient-oriented routine, fail only for the current recipient; continue message processing.

xxfi_connect - connection information

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_connect)(
    SMFICTX *ctx,
    char *hostname,
    _SOCK_ADDR *hostaddr);
```


Description

Called once, at the start of each SMTP connection. Default behavior is to do nothing and return SMFIS_CONTINUE.

Parameters

ctx The opaque context structure.

hostname

The host name of the message sender, as determined by a reverse lookup on the host address. If the reverse lookup fails, hostname will contain the message sender's IP address enclosed in square brackets (for example, [a.b.c.d]).

hostaddr

The host address, as determined by a getpeername() call on the SMTP socket. NULL if the type is not supported in the current version.

Result

If a previous filter rejects the connection in its xxfi_connect() routine, this filter's xxfi_connect() is not called.

xxfi_helo - SMTP HELO/EHLO command

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_helo)(
    SMFICTX * ctx,
    char * helohost
);
```

Description

Handle the HELO/EHLO command. xxfi_helo is called whenever the client sends a HELO/EHLO command. It can therefore be called between zero and three times. Default is to do nothing and return SMFIS_CONTINUE.

Parameters

ctx Opaque context structure.

helohost

Value passed to HELO/EHLO command, which should be the domain name of the sending host (but is, in practice, anything the sending host wants to send).

xxfi_envfrom - envelope sender

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_envfrom)(
    SMFICTX * ctx,
    char ** argv
);
```

Description

Handle the envelope FROM command. xxfi_envfrom is called once at the beginning of each message, before xxfi_envrcpt. The default behavior is to do nothing and return SMFIS_CONTINUE.

Parameters

ctx Opaque context structure.

argv Null-terminated SMTP command arguments; argv[0] is guaranteed to be the sender address. Later arguments are the ESMTP arguments.

Result

Can return the following values:

SMFIS_TEMPFAIL

Reject this sender and message with a temporary error; a new sender (and associated new message) can subsequently be specified. `xxfi_abort` is not called.

SMFIS_REJECT

Reject this sender and message; a new sender and message can be specified. `xxfi_abort` is not called.

SMFIS_DISCARD

Accept and silently discard this message. `xxfi_abort` is not called.

SMFIS_ACCEPT

Accept this message. `xxfi_abort` is not called.

xxfi_envrcpt - envelope recipient

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_envrcpt)(
    SMFICTX * ctx,
    char ** argv
);
```

Description

Handle the envelope RCPT command. `xxfi_envrcpt` is called once per recipient (one or more times per message), immediately after `xxfi_envfrom`. The default behavior is to do nothing and return `SMFIS_CONTINUE`.

Parameters

ctx Opaque context structure.

argv Null-terminated SMTP command arguments; `argv[0]` is guaranteed to be the recipient address. Later arguments are the ESMTP arguments.

Result

Can return the following values:

SMFIS_TEMPFAIL

Temporarily fail for this particular recipient; further recipients can still be sent. `xxfi_abort` is not called.

SMFIS_REJECT

Reject this particular recipient; further recipients can still be sent. `xxfi_abort` is not called.

SMFIS_DISCARD

Accept and discard the message. `xxfi_abort` is called.

SMFIS_ACCEPT

Accept recipient. `xxfi_abort` is not called.

xxfi_header - header

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_header)(
    SMFICTX * ctx,
    char * headerf,
    char * headerv
);
```

Description

Handle a message header. `xxfi_header` is called zero or more times between `xxfi_envrcpt` and `xxfi_eoh`, once per message header. The default behavior is to do nothing and then return `SMFIS_CONTINUE`.

Parameters

ctx Opaque context structure.

headerf
Header field name.

headerv
Header field value. The content of the header can include folded white space (multiple lines with following white space). The trailing line terminator (CR/LF) is removed.

Notes:

1. Later filters see any header changes or additions made by previous filters.
2. For more detail about header format, see `sendmail` documentation.

xxfi_eoh - end of header

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_eoh)(
    SMFICTX * ctx
);
```

Description

Handle the end of a message header. `xxfi_eoh` is called once after all headers have been sent and processed. The default behavior is to do nothing and then return `SMFIS_CONTINUE`.

Parameters

ctx Opaque context structure.

xxfi_body - body block

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_body)(
    SMFICTX * ctx,
    unsigned char * bodyp,
    size_t len
);
```

Description

Handle a piece of a message's body. `xxfi_body` is called zero or more times between `xxfi_eoh` and `xxfi_eom`. The default behavior is to do nothing and then return `SMFIS_CONTINUE`.

Parameters

ctx Opaque context structure.

bodyp
Pointer to the start of this block of body data. `bodyp` is not valid outside this call to `xxfi_body`.

len The amount of data pointed to by `bodyp`.

Notes:

1. Since message bodies can be very large, defining `xxfi_body` can significantly impact filter performance.
2. End-of-lines are represented as received from SMTP (normally CR/LF).
3. Later filters see body changes made by previous filters.
4. Message bodies might be sent in multiple chunks, with one call to `xxfi_body` per chunk.

xxfi_eom - end of message

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_eom)(
    SMFICTX * ctx
);
```

Description

End of a message. `xxfi_eom` is called once for a given message, after all calls to `xxfi_body`. The default behavior is to do nothing and then return `SMFIS_CONTINUE`.

Parameters

`ctx` Opaque context structure.

Note: A filter is required to make all its modifications to the message headers, body, and envelope in `xxfi_eom`. Modifications are made using the `smfi_*` routines.

xxfi_abort - message aborted

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_abort)(
    SMFICTX * ctx
);
```

Description

Handle the current message being aborted. `xxfi_abort` can be called at any time during message processing (between some message-oriented routine and `xxfi_eom`).

Parameters

`ctx` Opaque context structure.

Notes:

1. `xxfi_abort` must reclaim any resources allocated on a per-message basis, and must be tolerant of being called between any two message-oriented callbacks.
2. Calls to `xxfi_abort` and `xxfi_eom` are mutually exclusive.
3. `xxfi_abort` is not responsible for reclaiming connection-specific data, since `xxfi_close` is always called when a connection is closed.
4. Since the current message is already being aborted, the return value is currently ignored.
5. `xxfi_abort` is called only if the message is aborted outside the filter's control and the filter has not completed its message-oriented processing. For example, if a filter has already returned `SMFIS_ACCEPT`, `SMFIS_REJECT` or `SMFIS_DISCARD` from a message-oriented routine, `xxfi_abort` is not called, even if the message is later aborted outside its control.

xxfi_close - connection cleanup

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_close)(
    SMFICTX * ctx
);
```

Description

Provides notification that the current connection is being closed. `xxfi_close` is always called once at the end of each connection. The default behavior is to do nothing and then return `SMFIS_CONTINUE`.

Parameters

`ctx` Opaque context structure.

Notes:

1. `xxfi_close` is called on close even if the previous mail transaction was aborted.
2. `xxfi_close` is responsible for freeing any resources allocated on a per-connection basis.
3. Since the connection is already closing, the return value is currently ignored.

Chapter 12. Policy API (PAPI)

The Policy Agent includes an application programming interface (API) known as the Policy API or PAPI.

The PAPI interface allows user applications to connect to the Policy Agent through a UNIX socket connection and access policy related data. Data returned from the Policy Agent are queued in the user's address space. A set of PAPI functions is defined to access specific portions of the returned data. The interface also provides for terminating the connection and cleaning up resources obtained while the API was in use.

Currently, the only function provided by PAPI is to retrieve policy performance data.

API outline for retrieving data from Policy Agent

Using the PAPI interface, an application uses the `papi_connect()` call to define an API connection and to register with the Policy Agent. The `papi_get_perf_data()` call is used to retrieve the policy performance data from the Policy Agent. An application can then use a set of helper functions to access performance information returned on the `papi_get_perf_data()` call. The set of helper functions is:

- `papi_get_policy_instance()` - Returns the policy instance number for the set of policies in the performance data returned.
- `papi_get_rules_count()` - Returns the number of policy rules in the performance data returned.
- `papi_get_actions_count()` - Returns the number of policy actions in the performance data returned.
- `papi_get_rule_perf_info()` - Returns a policy rule entry based on the rule number that contains the rule performance information.
- `papi_get_rule_perf_by_id()` - Returns a policy rule entry based on the rule ID that contains the rule performance information.
- `papi_get_action_perf_info()` - Returns a policy action entry based on the action number that contains the action performance information.
- `papi_get_action_perf_by_id()` - Returns a policy action entry based on the action ID that contains the action performance information.
- `papi_strerror()` - Returns a string describing a PAPI return code value, similar to the C `strerror()` function.

When the application is done using the data returned on the `papi_get_perf_data()` call, it can call `papi_free_perf_data()` to free the data. When the application no longer wants to retrieve policy performance data from the Policy Agent, it can call `papi_disconnect()` to end the connection.

Compiling and linking PAPI applications

To use the PAPI interface, an application must perform the following steps:

1. Include the `<papiuser.h>` header file, which is available in the `/usr/include` directory.

2. Compile the application with the DLL compiler option. See the *z/OS XL C/C++ User's Guide* for more information on how to specify compiler options.
3. Include the PAPI definition side deck (papiuser.x), which is available in the /usr/lib directory, when prelinking or binding the application.
4. If the Binder is used instead of the C prelinker, specify the Binder DYNAM=DLL option. See the *z/OS MVS Program Management: User's Guide and Reference* for information on specifying Binder options.

Running PAPI applications

At execution time, the PAPI application must have access to the PAPI DLL (papi.dll), which is available in the /usr/lib directory. Ensure that the LIBPATH environment variable includes this directory when running the application. The PAPI application must either run with superuser authority to use PAPI, or must have security product authority in the SERVAUTH class. These security product profiles can be defined by TCP/IP stack (TcpImage) and policy type (only ptype = QOS is applicable). Wildcarding of profile names is allowed. The security product profiles take the form:

EZB.PAGENT.<sysname>.<TcpImage>.<ptype> where:

- sysname - System name defined in sysplex
- TcpImage - TCP name for the requested policy information
- ptype - Policy Type that is being requested (QOS)

Note: Wildcarding is allowed on segments of the profile name.

See the EZARACF sample in SEZAINST for sample commands needed to create the profile name and permit users access to it.

PAPI return codes

The following return codes may be returned from PAPI functions.

Table 7. PAPI function return codes

Return Code	Value	Description
PAPI_OK	0	Success.
PAPI_HELPER_RETURN_NULL	NULL	NULL return from PAPI helper function.
PAPI_HELPER_RETURN_ZERO	0	Zero return from PAPI helper function.
PAPI_NOK	1	Generic error.
PAPI_INVALID_PARAMETER_VALUE	2	Parameter has an invalid value.
PAPI_CLIENT_ALREADY_INITIALIZED	4	User already issued papi_connect().
PAPI_CLIENT_ALREADY_REGISTERED	5	User already issued papi_connect().
PAPI_FUNC_NOT_READY	8	PAPI function not ready - try again later.
PAPI_INVALID_ACCEPTABLE_CACHED_TIME	10	The acceptableCachedTime input parameter is ignored because it is less than the MinimimSamplingInterval configured to the Policy Agent.
PAPI_PERF_COLL_TYPE_MISMATCH	11	Some or all of the requested type of performance data (rules, actions, or both) is not being collected by the Policy Agent.

Table 7. PAPI function return codes (continued)

Return Code	Value	Description
PAPI_MALLOC_FAILED	16	PAPI could not allocate memory in user's address space.
PAPI_MALLOC_IN_PAGENT_FAILED	17	Policy Agent could not allocate memory.
PAPI_PAGENT_INTERNAL_ERROR	18	Internal error encountered in Policy Agent.
PAPI_INTERNAL_ERROR	19	Internal error encountered in PAPI.
PAPI_CLIENT_NOT_REGISTERED	20	User did not issue papi_connect().
PAPI_NOT_VALID_AUTHORIZATION	21	User not authorized to issue PAPI function.
PAPI_VERSION_INCORRECT	22	Incompatibility between the version of the PAPI DLL (papi.dll) and the version of Policy Agent.
PAPI_CONNECT_FAILED	30	Connect to Policy Agent failed.
PAPI_READ_FAILED	31	Read from Policy Agent failed.
PAPI_SOCKET_NOT_READABLE	32	Socket for Policy Agent connection is not readable.
PAPI_WRITE_FAILED	33	Write to Policy Agent failed.
PAPI_READ_TIMEOUT	34	Read from Policy Agent timed out.
PAPI_SOCKET_FAILED	35	Could not open socket for Policy Agent connection.
PAPI_FCNTL_FAILED	36	fcntl() on connection socket failed.
PAPI_NULL_INPUT	50	A required parameter is not specified.
PAPI_TCPIMAGE_NOT_VALID	51	The specified kernel name is not known to the Policy Agent.
PAPI_TCPIMAGE_INVALID_LENGTH	52	The specified kernel name is too long.
PAPI_FILTERNAME_INVALID_LENGTH	53	The specified filter name is too long.
PAPI_KERNEL_NOT_AVAILABLE	54	The TCP/IP stack is not available to process a request, or an error occurred while obtaining data from the stack.

Client library services

The Policy Agent API provides the following client library calls to connect, disconnect, get, and free storage for policy performance data.

- papi_connect()
- papi_debug()
- papi_disconnect()
- papi_free_perf_data()
- papi_get_perf_data()

The Policy Agent API provides the following helper functions to access the policy performance data.

- papi_get_action_perf_by_id()
- papi_get_action_perf_info()
- papi_get_actions_count()
- papi_get_policy_instance()

- papi_get_rule_perf_by_id()
- papi_get_rule_perf_info()
- papi_get_rules_count()
- papi_strerror()

To use these calls, the application must include the file papiuser.h.

Connecting and retrieving data

papi_connect - Connect to Policy Agent

```
#include <papiuser.h>

extern int papi_connect(void **papiHandle, void *regReq);
```

Description

This function is used to open a connection and register with the Policy Agent. The parameters it takes are a pointer to a void pointer, which is used to return the handle, and a void pointer to pass in the registration information. The registration information is currently not used. All information about this connection is stored internally using the handle pointer as a reference. Most other PAPI functions require that this handle be passed in as input. A call should subsequently be made to papi_disconnect() to release the resources used by papi_connect().

Parameters

**papiHandle

This is an output parameter that points to the handle to identify this papi_connect().

*regReq

This is an input parameter that points to the registration information. This pointer should be NULL.

Result

If the connection is successful, the call returns a return code of PAPI_OK, and papiHandle is set.

If the connect fails, the call returns a non-PAPI_OK return code value.

Example

```
void *mainHandle;
int nRc;

nRc = papi_connect(&mainHandle, NULL);

if (nRc != PAPI_OK)
{
    printf("Error in papi_connect : %d\n", nRc );
}
else
{
    /* everything is ok so far ... */
}
```

papi_debug - Set debug capability

```
#include <papiuser.h>

extern int papi_debug(papiDebug_t *debugValue);
```

Description

This function allows debug information to be displayed for PAPI functions. This function can be called by the application to turn debug on or off anytime during the PAPI processing.

Parameters

***debugValue**

This is a pointer to an input parameter that is used to turn debug on or off.

If papi_debug() is not issued, then no debug information is displayed.

If debug is being used, then the application can pass in a user exit in the papiDebug_t papiLogFunc field.

If papiLogFunc is NULL, then all messages are logged using printf().

The following defines are located in papiuser.h.

```
typedef struct {
    unsigned int    papiDebugOpt; /* Debug On/Off */
    void            *papiUserValue; /* User Define value */
    papiLogUserExit_t papiLogFunc; /* Logging Function */
} papiDebug_t; /* Input papi_debug() */

Set papiDebugOpt:
#define PAPI_DEBUG_OFF    0 /* Debug Off */
#define PAPI_DEBUG_ON    1 /* Debug On */
```

Result

If the debug is successful, the call returns a return code of PAPI_OK.

If the debug fails, the call returns a non-PAPI_OK return code value.

papi_disconnect - Disconnect from the Policy Agent

```
#include <papiuser.h>
```

```
extern int papi_disconnect(void *papiHandle);
```

Description

This function is used to terminate a connection with the Policy Agent. The only parameter it takes is a pointer to the handle that was set in the papi_connect() call.

Parameters

***papiHandle**

This is an input parameter of type void. It is the responsibility of the caller to disconnect from the Policy Agent based on the papiHandle returned on the papi_connect() API.

Result

If the disconnect is successful, the call returns a return code of PAPI_OK.

If the disconnect fails, the call returns a non-PAPI_OK return code value.

Example

```
void *mainHandle;
int nRc;

nRc = papi_disconnect(mainHandle);
```

```

if (nRc != PAPI_OK)
{
    printf("Error in papi_disconnect : %d\n", nRc );
}
else
{
    /* everything is ok so far ... */
}

```

papi_free_perf_data - Free retrieved QoS performance data

```

#include <papiuser.h>

extern int papi_free_perf_data(void *perfDataHandle);

```

Description

This function is used to free the memory associated with the policy performance data returned by the `papi_get_perf_data()` API. This API should be invoked with the handle to free the memory allocated to hold the performance information.

Parameters

***perfDataHandle**

This is an input parameter of type void that points to the memory obtained from the `papi_get_perf_data()` API.

Result

If the free is successful, the call returns a return code of PAPI_OK.

If the free fails, the call returns a non-PAPI_OK return code value.

Example

```

int nRc;
void *perfDataHandle;

/* Initialization and obtaining data to be done here */

nRc = papi_free_perf_data(perfDataHandle);

if (nRc != PAPI_OK)
{
    printf("Error in papi_free_perf_data : %d\n", nRc );
}
else
{
    /* everything is ok so far ... */
}

```

papi_get_perf_data - Retrieve QoS performance data

```

#include <papiuser.h>

extern int papi_get_perf_data(void *papiHandle,
                             int typeFlag,
                             void *filter,
                             int *acceptableCachedTime,
                             char *kernelName,
                             void **perfDataHandle);

```

Description

This function is used to retrieve the policy performance data from the Policy Agent.

Policy performance metrics collected by the kernel are affected by the FLUSH/NOFLUSH parameter on the Policy Agent TcpImage configuration statement. FLUSH causes the metrics values to be reset to 0 at the following times:

- When a new TcpImage statement is processed for the first time, including Policy Agent starting. This should not be a concern in most cases.
- When a MODIFY REFRESH command is entered.

Metrics are never reset when NOFLUSH is specified. See the *z/OS Communications Server: IP Configuration Reference* and the policy-based networking information in the *z/OS Communications Server: IP Configuration Guide* for more information.

Note: Changes to policy definitions might not cause immediate changes in performance metrics, due to averaging and smoothing over several sampling intervals. A period of time must elapse after a policy change in order to reach a new steady state.

Parameters

***papiHandle**

This is an input parameter of type void that points to the handle to identify the associated papi_connect().

typeFlag

This is an input parameter of type int that specifies the type of performance data that is requested. This field is treated as a bit stream and multiple data types can be specified by turning on the required bits. Turning on the bits will return all the performance data of that type (for example, turning on the rules bit returns all rule performance data). The supported bit definitions are:

PAPI_RULES

Indicates to retrieve performance data from policy rules.

PAPI_ACTIONS

Indicates to retrieve performance data from policy actions.

PAPI_ALL

Indicates to retrieve performance data from policy rules and policy actions.

If this API is issued to get data for a type (rule and action) that has not been configured by the DataCollection parameter on the PolicyPerformanceCollection statement, this API is able to return only data that is configured to be collected by the Policy Agent. A PAPI_PERF_COLL_TYPE_MISMATCH return code indicates that the request type was not collected.

***filter**

This field of type void is reserved for future use. The only type of filtering that is supported is through the use of the typeFlag field. This parameter must be specified as NULL.

***acceptableCachedTime**

This is an input and output parameter of type int. This parameter is specified in seconds and is used to determine whether Policy Agent returns to the caller any performance data that has been cached. If the time that had elapsed after the data was retrieved and cached is greater than the acceptableCachedTime, Policy Agent retrieves new data from the stack and

returns this new data to the caller. The acceptableCachedTime is set to MinimumSamplingInterval. This new performance data will now be cached.

If this API specifies an acceptableCachedTime that is less than the MinimumSamplingInterval parameter on the PolicyPerformanceCollection statement, the MinimumSamplingInterval is used to determine whether data needs to be retrieved from the stack. An error (PAPI_INVALID_ACCEPTABLE_CACHED_TIME) is returned stating that the acceptableCachedTime has been ignored. Data will be returned and the MinimumSamplingInterval will be returned as the acceptableCachedTime. See the *z/OS Communications Server: IP Configuration Reference* for setting the MinimumSamplingInterval parameter on the PolicyPerformanceCollection statement in the Policy Agent configuration file.

Note: This is a required parameter.

***kernelName**

This is an input parameter. It is a pointer to a character string of the kernel name whose policy performance data will be returned. The kernel name must be eight characters or less in length. If kernelName is NULL, the default kernel name, as determined using the standard resolver search order, will be used. If kernelName is invalid, the return code of PAPI_TCPIIMAGE_NOT_VALID will be returned.

****perfDataHandle**

This is an output parameter that points to the handle to identify this papi_get_perf_data(). It is the responsibility of the caller to free this memory by calling the papi_free_perf_data() API. The values within this memory should be obtained using the helper functions provided.

Result

If the retrieve is successful, the call returns a return code of PAPI_OK, and perfDataHandle is set.

If the retrieve fails, the call returns a non-PAPI_OK return code value.

Example

```
void *mainHandle;
int nRc, type;
int cacheTime = 10;
void *perfDataHandle;
static char kernelName[9] = {'\0'};

/* initialization to be done here */

strcpy(kernelName, "TCPNAME");
type = PAPI_RULES_DATA | PAPI_ACTION_DATA;
nRc = papi_get_perf_data(mainHandle,
                        type,
                        NULL,
                        &cachetime,
                        kernelName,
                        &perfDataHandle);

if (nRc != PAPI_OK)
{
    printf("Error in papi_get_perf_data : %d\n", nRc );
}
```

```

else
{
    /* everything is ok so far ... */
}

```

Helper functions

papi_get_action_perf_by_id- Obtain performance information on the action specified by the action ID

```

#include <papiuser.h>

extern ActionPerfInfo *papi_get_action_perf_by_id( void *perfDataHandle,
                                                    int    actionId);

```

Description

This function is used to obtain the performance information on a particular action specified by the action ID.

The performance information is returned as an ActionPerfInfo structure, as described in “papi_get_action_perf_info - Obtain performance information on a particular action” (the recordId will match the actionId field).

Parameters

***perfDataHandle**

This is an input parameter of type void. This parameter points to the performance data returned by the papi_get_perf_data() API.

actionId

This is an input parameter of type int. This parameter identifies a particular action.

Result

If the function is successful, the ActionPerfInfo is returned. This pointer should not be freed.

If the function fails, it returns NULL.

papi_get_action_perf_info - Obtain performance information on a particular action

```

#include <papiuser.h>

extern ActionPerfInfo *papi_get_action_perf_info( void *perfDataHandle,
                                                    int    actionNum);

```

Description

This function is used to obtain the performance information on a particular action. The action number is specified by actionNum. When multiple policy rules refer to a given policy action, the performance information in the action is an aggregate of all the rules that refer to it.

The performance information is returned as an ActionPerfInfo structure, defined as:

```

typedef struct {
    char          name[MAX_POLICY_NAME]; /* Rule / Action name */
    Bit32         recordType;            /* Rule / Action      */
    Bit32         recordId;              /* Rule / Action Id   */
    time_t        firstActivated;        /* Time first activated */
}

```

```

time_t      lastMapped;          /* Time last mapped      */
Bit64      bytesXmitted;        /* Total bytes transmitted*/
Bit64      packetsXmitted;      /* Total packets
                                transmitted      */
Bit32      activeConnections;   /* Active connections
                                count            */
Bit32      reserved4;          /* Reserved              */
Bit64      acceptedConnections; /* Total accepted
                                connections      */
Bit32      smoothedRttAvg;      /* Average smoothed RTT  */
Bit32      smoothedRttMdev;     /* MDEV of smoothed RTT  */
Bit64      bytesRexmitted;      /* Total bytes
                                retransmitted    */
Bit64      packetsRexmitted;    /* Total packets
                                retransmitted    */
Bit32      smoothedConnDelayAvg; /* Average smoothed conn
                                delay            */
Bit32      smoothedConnDelayMdev; /* MDEV of smoothed conn
                                delay            */
Bit32      acceptQDelayAvg;     /* Average accept queue
                                delay            */
Bit32      acceptQDelayMDev;    /* MDEV of accept queue
                                delay            */
Bit64      packetsXmittedInProfile; /* Outbound in profile
                                packets count    */
Bit64      bytesXmittedInProfile; /* Outbound in profile
                                bytes count      */
Bit64      reserved2;          /* Reserved              @Q1A*/
Bit64      reserved3;          /* Reserved              @Q1A*/
Bit64      packetsReceived;     /* Total packets
                                received          @Q1A*/
Bit64      bytesReceived;       /* Total bytes
                                received          @Q1A*/
Bit64      packetsXmittedTimedOut; /* Total transmitted
                                packets timed out @Q1A*/
Bit64      deniedConnections;   /* Total denied
                                connections      @Q1A*/
} RulePerfInfo, ActionPerfInfo;

```

Parameters

*perfDataHandle

This is an input parameter of type void. This parameter points to the performance data returned by the papi_get_perf_data() API.

actionNum

This is an input parameter of type int. This parameter points to a particular action. The action number starts from 0. For example, if the number of actions returned by the papi_get_actions_count() function is 6, specify 0 through 5 for the actionNum parameter.

Result

If the function is successful, the ActionPerfInfo is returned. This pointer should not be freed.

If the function fails, it returns NULL.

papi_get_actions_count - Obtain number of actions in the policy performance data

```

#include <papiuser.h>

extern int papi_get_actions_count( void *perfDataHandle );

```


Description

This function is used to obtain the number of actions in the policy performance data returned by the `papi_get_perf_data()` function.

Parameters

***perfDataHandle**

This is an input parameter of type `void`. This parameter points to the performance data returned by the `papi_get_perf_data()` API.

Result

If the function is successful, the number of actions in the policy performance data is returned.

If the function fails, 0 is returned.

papi_get_policy_instance - Obtain policy instance number for policies in the policy performance data

```
#include <papiuser.h>
```

```
extern int papi_get_policy_instance( void *perfDataHandle );
```

Description

This function is used to obtain the policy instance number for the set of policies in the policy performance data returned by the `papi_get_perf_data()` function. The instance number is a value that applies to an entire set of policies, and changes only when a change has been made to the set of policies (for example, when policies are added or deleted).

Parameters

***perfDataHandle**

This is an input parameter of type `void`. This parameter points to the performance data returned by the `papi_get_perf_data()` API.

Result

If the function is successful, the policy instance number for the set of policies in the policy performance data is returned. The instance number is a positive integer.

If the function fails, 0 is returned.

papi_get_rule_perf_by_id - Obtain performance information on the rule specified by the rule ID

```
#include <papiuser.h>
```

```
extern RulePerfInfo *papi_get_rule_perf_by_id( void *perfDataHandle,  
                                              int    ruleId);
```

Description

This function is used to obtain the performance information on the rule that is specified by the rule ID.

The performance information is returned as a `RulePerfInfo` structure, as described in “`papi_get_rule_perf_info` - Obtain performance information on a particular rule” on page 390 (the `recordId` will match the `ruleId` field).

Parameters

***perfDataHandle**

This is an input parameter of type void. This parameter points to the performance data returned by the papi_get_perf_data() API.

ruleId This is an input parameter of type int. This parameter identifies a particular rule.

Result

If the function is successful, the RulePerfInfo is returned. This pointer should not be freed.

If the function fails, it returns NULL.

papi_get_rule_perf_info - Obtain performance information on a particular rule

```
#include <papiuser.h>
```

```
extern RulePerfInfo *papi_get_rule_perf_info( void *perfDataHandle,  
                                              int ruleNum);
```

Description

This function is used to obtain the performance information on a particular rule. The rule number is specified by ruleNum.

The performance information is returned as a RulePerfInfo structure, defined as:

```
typedef struct {  
    char          name[MAX_POLICY_NAME]; /* Rule / Action name */  
    Bit32         recordType;           /* Rule / Action */  
    Bit32         recordId;             /* Rule / Action Id */  
    time_t        firstActivated;       /* Time first activated */  
    time_t        lastMapped;          /* Time last mapped */  
    Bit64         bytesXmitted;         /* Total bytes transmitted*/  
    Bit64         packetsXmitted;      /* Total packets */  
                                         transmitted /*  
    Bit32         activeConnections;   /* Active connections */  
                                         count /*  
    Bit32         reserved4;           /* Reserved */ /*  
    Bit64         acceptedConnections; /* Total accepted */  
                                         connections /*  
    Bit32         smoothedRttAvg;      /* Average smoothed RTT */ /*  
    Bit32         smoothedRttMdev;     /* MDEV of smoothed RTT */ /*  
    Bit64         bytesRexmitted;      /* Total bytes */  
                                         retransmitted /*  
    Bit64         packetsRexmitted;    /* Total packets */  
                                         retransmitted /*  
    Bit32         smoothedConnDelayAvg; /* Average smoothed conn */  
                                         delay /*  
    Bit32         smoothedConnDelayMdev; /* MDEV of smoothed conn */  
                                         delay /*  
    Bit32         acceptQDelayAvg;     /* Average accept queue */  
                                         delay /*  
    Bit32         acceptQDelayMDev;    /* MDEV of accept queue */  
                                         delay /*  
    Bit64         packetsXmittedInProfile; /* Outbound in profile */  
                                         packets count /*  
    Bit64         bytesXmittedInProfile; /* Outbound in profile bytes */  
                                         count /*  
    Bit64         reserved2;           /* Reserved @Q1A*/ /*  
    Bit64         reserved3;           /* Reserved @Q1A*/ /*  
    Bit64         packetsReceived;     /* Total packets
```

```

        received          @Q1A*/
Bit64      bytesReceived; /* Total bytes
        received          @Q1A*/
Bit64      packetsXmittedTimedOut; /* Total transmitted
        packets timed out @Q1A*/
Bit64      deniedConnections; /* Total denied
        connections       @Q1A*/
} RulePerfInfo, ActionPerfInfo;

```

Parameters

***perfDataHandle**

This is an input parameter of type void. This parameter points to the performance data returned by the `papi_get_perf_data()` API.

ruleNum

This is an input parameter of type int. This parameter points to a particular rule. The rule number starts from 0. For example, if the number of rules returned by the `papi_get_rules_count()` function is 5, specify 0 through 4 for the `ruleNum` parameter.

Result

If the function is successful, the `RulePerfInfo` is returned. This pointer should not be freed.

If the function fails, it returns NULL.

papi_get_rules_count - Obtain number of rules in the policy performance data

```

#include <papiuser.h>

extern int papi_get_rules_count( void *perfDataHandle );

```

Description

This function is used to obtain the number of rules in the policy performance data returned by the `papi_get_perf_data()` function.

Parameters

***perfDataHandle**

This is an input parameter of type void. This parameter points to the performance data returned by the `papi_get_perf_data()` API.

Result

If the function is successful, the number of rules in the policy performance data is returned.

If the function fails, 0 is returned.

papi_strerror - Return string describing PAPI return code value

```

#include <papiuser.h>

extern char *papi_strerror( int papiReturnCode );

```

Description

This function is used to obtain a string describing a PAPI return code value. It is similar to the C `strerror()` function.

Parameters

papiReturnCode

This is an input parameter of type int. This parameter contains a PAPI return code value.

Result

If the return code is known, a string describing the return code value is returned.

If the return code is not known, a generic unknown error string is returned.

Chapter 13. FTP Client Application Programming Interface (API)

This topic describes the FTP Client Application Programming Interface (API) to the z/OS FTP client. This topic explains how to initialize the interface, how to use the interface to submit a subcommand to the client, how to retrieve results of a request, and how to terminate the interface.

The following terms apply:

- *Subcommand* refers to z/OS FTP client subcommands.
- *Request* refers to a request sent to the interface (see “Sending requests to the FTP Client API” on page 413).

A subcommand might result from a request, because a request can do the following:

- Invoke a specific subcommand (on an SCMD request)
- Result in an implicit subcommand (OPEN resulting from INIT)
- Automatically generate a subcommand (QUIT sent by TERM)
- Result in no subcommand (INIT with no host name or IP address included in start parameters; TERM issued after the user has explicitly issued SCMD QUIT; GETL; or POLL)

Guideline: Subcommands are processed by the z/OS FTP client. Some subcommands result in one or more FTP commands being sent to the FTP server. Examples of subcommands and commands are:

- LOCSTAT is a subcommand. No command is sent to the server for this subcommand.
- SYSTEM is a subcommand; a SYSTEM subcommand causes the client to send a SYST command to the server.
- GET is a subcommand. A data connection establishment command (PORT, PASV, or EPSV) might be sent to the server; then a RETR command is sent to the server.

Tip: FTP subcommands in *z/OS Communications Server: IP User's Guide and Commands* describes the subcommands that are supported by the z/OS FTP client.

The interface to the z/OS FTP client enables a user program to send subcommands for the client to process. The user program can multithread to different instances of the interface by requesting no-wait mode when processing a subcommand. The interface also enables the user program to retrieve output that includes the messages from the client, replies from the FTP server, and other data generated as the result of the request.

The interface requires the use of an FTP Client Application Interface (FCAI) control block that is created by the user program [see “FTP Client Application Interface (FCAI) control block” on page 399]. The FCAI is a parameter on all calls to the interface and it is used to pass information between the interface and the user program.

The following topics are included in this topic:

- “Compatibility considerations” on page 394

- “Guidelines and requirements for the FTP Client API”
- “COBOL, C, REXX, Assembler, and PL/I call formats” on page 396
- “Converting parameter descriptions” on page 397
- “z/OS FTP client behavior when invoked from the FTP Client API” on page 397
- “FTP Client Application Interface (FCAI) control block” on page 399
- “Sending requests to the FTP Client API” on page 413
- “FTP Client API C functions” on page 430
- “FTP Client API REXX function” on page 433
- “Output register information for the FTP Client API” on page 455
- “Other output that is returned to the application” on page 456
- “Prompts from the client” on page 457
- “Messages and replies” on page 458
- “Interpreting results from an interface request” on page 459
- “Programming notes for the FTP Client API” on page 461
- “Using the FTP Client API trace” on page 465
- “FTP Client API sample programs” on page 469

Compatibility considerations

Unless noted in *z/OS Communications Server: New Function Summary*, an application program compiled and link edited on a particular release of z/OS Communications Server IP can be used on higher level releases. Application programs that are compiled and link edited on a particular release of z/OS Communications Server IP cannot be used on older releases.

Guidelines and requirements for the FTP Client API

This section lists the usage guidelines, requirements, and restrictions for the FTP Client Application Programming Interface (API) for user application programs.

Table 8 describes the programming requirements that apply to the FTP Client API.

Table 8. Programming requirements for the FTP Client API

Function	Restriction
Authorization	Supervisor state or problem state, any PSW key
Dispatchable unit mode	Task
SRB mode	The API can be invoked only in TCB mode (task mode).
Cross-memory mode	The API can be invoked only in a non-cross-memory environment (PASN=SASN=HASN).
ASC mode	Primary address space control (ASC) mode
Interrupt status	Enabled for interrupts
Locks	No locks should be held when issuing these calls.
Control parameters	Parameter lists and the FCAI control block must reside in primary storage that is accessible by the API to prevent ABENDs in the EZAFTPKS interface program.
Functional Recovery Routine (FRR)	Do not invoke the API with an FRR set. This can cause system recovery routines to be bypassed and severely damage the system.

Table 8. Programming requirements for the FTP Client API (continued)

Function	Restriction
Storage	Storage acquired for the purpose of containing data returned from an FTP Client API call must be obtained in the same key as the application program status word (PSW) at the time of the call.
Nested FTP Client API calls	You cannot issue nested FTP Client API calls within the same task. If a request block (RB) issues an FTP Client API call and is interrupted by an interrupt request block (IRB) in an STIMER exit, no additional FTP Client API calls can be issued by the IRB.
Addressability mode (AMODE) considerations	The API must be invoked while the caller is in 31-bit addressability mode.

Guidelines:

- The FTP Client API is re-entrant.
- The user program can have more than one FTP Client Application Interface control block initialized and active in a single address space (see “FTP Client Application Interface (FCAI) control block” on page 399).
- The FTP Client API spawns a child process for the z/OS FTP client. If you have a signal handler, you might see the SIGCHILD signal raised when the FTP client terminates; no action is required.
- The z/OS FTP client contains handlers for various asynchronous signals. The FTP Client API does not contain any signal handlers, nor does it block or explicitly raise any signals. See “Programming notes for the FTP Client API” on page 461 for more information about errors in the z/OS FTP client process.

Requirements:

- The application must supply an accessible parameter list and FCAI in primary storage to the FTP Client API. ABENDs can occur in the interface if the application fails to comply with this requirement.
- Other ABENDS that occur due to inaccessible storage are trapped by the interface and returned to the application program as an interface error [see FCAI_IE and its associated values in “FTP Client Application Interface (FCAI) control block” on page 399]. To enable the interface to trap these ABENDs, specify TRAP(ON,NOSPIE) to disable invocation of the ESPIE macro when the application program executes within a Language Environment® enclave. For example, specify the following execution parameter for a COBOL application program:

```
PARM= '/TRAP(ON,NOSPIE)'
```

For instructions on specifying runtime options and parameters for Language Environment languages, see the information about using runtime options in *z/OS Language Environment Programming Guide*.
- All of the requests using the same FCAI control block must be made from the same thread.
- The user program must use a standard call interface. Samples are provided for COBOL, C, PL/I, and Assembler (see “FTP Client API sample programs” on page 469).
- The user program must execute in 31-bit addressing mode (AMODE 31). Other addressing modes are not supported by the interface. The program can reside at any location (RMODE can be 24 or ANY).

- The application must have an OMVS segment defined (or defaulted).
- The interface module EZAFTPKE must be accessible to the application in the linklist or in a STEPLIB or JOBLIB DD statement.
- You can either statically link the FTP Client API stub program (EZAFTPKE) into the user application program or load it dynamically for execution. The stub program resides in SYSn.CSSLIB and is designed to maintain upward compatibility.
- For a PL/I program, include this statement before your first call instruction:
DCL EZAFTPKE ENTRY OPTIONS(RETCODE,ASM,INTER) EXT;

COBOL, C, REXX, Assembler, and PL/I call formats

The FTP Client API is invoked by calling the EZAFTPKE program. The following list shows formats for the COBOL, C, Assembler, and PL/I languages.

- COBOL language call format
The EZAFTPKE call format for COBOL programs is the following:

```
>>---CALL EZAFTPKE USING FCAI-Map, request_type, parm1, parm2, ... -----><
```
- C language call format
See “FTP Client API C functions” on page 430 for in-line functions that can be used with C/C++ programs. These in-line functions provide the calls to EZAFTPKE.
- REXX language call format
See “FTP Client API REXX function” on page 433 for an external REXX function that can be used with REXX programs.
- Assembler language call format
The EZAFTPKE call format for assembler language programs is the following:

```
>>---CALL EZAFTPKE,(FCAI_Map, request_type, parm1, parm2, ... ),VL -----><
```
- PL/I language call format
The EZAFTPKE call format for PL/I programs is the following:

```
>>---CALL EZAFTPKE (FCAI_Map, request_type, parm1, parm2, ... ); -----><
```

The following parameter definitions apply for each of the call formats:

FCAI-Map or FCAI_Map

The name of the FTP Client Application Interface block storage that describes an instance of use of the interface, or a pointer to the storage. The storage for this space is acquired by the calling program. COBOL and Assembler callers can append storage within the calling program to the area defined in the COBOL copy member or Assembler macro. PL/I or C callers must alter the INCLUDE member to add user storage to the area.

request_type

The type of processing requested by the invocation of the interface.

parm_n

A variable number of parameters, depending on the request type.

Some parameters are optional depending on the request. When an optional parameter is omitted but more parameters follow, use a placeholder appropriate for the language:

- COBOL uses the special name OMITTED in place of the missing parameter.
- C and PL/I use the special name NULL in place of the missing parameter.

- Assembler language uses a comma to indicate the position of the missing parameter.

Converting parameter descriptions

The coding examples in this topic use IBM Enterprise COBOL for z/OS language syntax and conventions. The application program should use the syntax and conventions that are appropriate for the language in which it is written.

Example storage definition statements for COBOL, C, PL/I, and assembler language programs are:

- IBM Enterprise COBOL for z/OS

```
PIC S9(4) COMP-5      HALFWORD BINARY VALUE
PIC S9(8) COMP-5      FULLWORD BINARY VALUE
PIC X(n)              CHARACTER FIELD OF n BYTES
```

- C

```
short int      /* HALFWORD BINARY VALUE      */
long int       /* FULLWORD BINARY VALUE       */
char x[n]      /* CHARACTER FIELD OF n BYTES */
```

- PL/I declare statement

```
DCL HALF FIXED BIN(15),  HALFWORD BINARY VALUE
DCL FULL FIXED BIN(31),  FULLWORD BINARY VALUE
DCL CHARACTER CHAR(n)    CHARACTER FIELD OF n BYTES
```

- Assembler declaration

```
DS   H          HALFWORD BINARY VALUE
DS   F          FULLWORD BINARY VALUE
DS   CLn        CHARACTER FIELD OF n BYTES
```

z/OS FTP client behavior when invoked from the FTP Client API

The z/OS FTP client that is used by the FTP Client API is described in the File Transfer Protocol (FTP) information in the *z/OS Communications Server: IP User's Guide and Commands* and in the File Transfer Protocol information in the *z/OS Communications Server: IP Configuration Reference*. The z/OS FTP client, when started with the FTP Client API, operates essentially as it does when invoked in an interactive environment under the z/OS UNIX shell.

The following are the differences in the behavior of the z/OS FTP client when it is invoked by the FTP Client API:

- When the z/OS FTP client starts, options (parameters) are processed that affect the operation of the client. The user program uses the START-PARM parameter on the INIT request to pass its options to the FTP Client API, which passes the options on to the client (see “INIT” on page 413). All of the options that are defined for the z/OS FTP client are accepted when the client is started with the FTP Client API. However, the following conditions apply:
 - The -e and the EXIT options are ignored by the FTP Client API.

These options are intended to affect the operation of the FTP client by causing it to stop when an eligible subcommand encounters an error. In the FTP Client API, those errors are passed back to the user program as a client error code (see Table 9 on page 399). The user program can process the error and decide whether to continue or to end the client process.

- The -i option to disable prompting for the subcommands MGET, MPUT, MDELETE has no effect on the API.

See “Prompts from the client” on page 457 for a discussion on how prompts are handled when the z/OS FTP client is invoked from the FTP Client API.

- When the z/OS FTP client is invoked within the z/OS UNIX shell, a backslash (\) is required before the open parenthesis [(] that signals the start of the MVS-type parameters. Do not use the backslash when invoking the client with the FTP Client API.
- When the z/OS FTP client is invoked from a batch job or from TSO, data sets and files can be allocated to DD names for use by the client. When the z/OS FTP client is spawned from the FTP Client API, DD names that are associated with the application are not available to the client process. Specifically, the use of the following DD names is not supported by the FTP Client API:

SYSFTPD and SYSTCPD

NETRC

INPUT (SYSIN) and OUTPUT

Transfer of data sets by DD name is not possible in the spawned client process. If the application sends a transfer subcommand (PUT, GET, and so on) that includes //DD:ddname, the client returns an error such as FCAI_CEC_FILE_ACCESS.

- Changing local site defaults using FTP.DATA in *z/OS Communications Server: IP User's Guide and Commands* describes how to change local site defaults by using FTP.DATA. The search order for locating the FTP.DATA configuration file for the client under the FTP Client API is as follows:

1. -f parameter
2. \$HOME/ftp.data
3. userid.FTP.DATA
4. /etc/ftp.data
5. SYS1.TCPPARMS(FTPDATA)
6. tcpip_hlq.FTP.DATA

Restriction: The -f parameter cannot be a DD name when the FTP client is invoked from the FTP Client API.

Tip: The \$HOME variable is taken from the user's RACF® user profile OMVS segment. The \$HOME variable can be modified with the environment variable list passed during FTP Client API initialization. Initialization is performed with an ftpapi('init') request in the REXX environment; with a call to the EZAFTPKS stub program with the INIT keyword for Assembler, COBOL, and PL/1; or with a call to the FAPI_INIT function for C/C++.

- The FTP.DATA statements that can be used to change local site defaults for the z/OS FTP client are defined in the FTP.DATA statements information in the *z/OS Communications Server: IP Configuration Reference*. One of the statements is CLIENTERRCODES, which controls return code settings in the client. When the client is started by the FTP Client API, the value on the CLIENTERRCODES statement does not affect the reporting of results. See “Interpreting results from an interface request” on page 459 for a complete description of how results from the interface and the client are reported.
- When the z/OS FTP client or server prompts for a password or accounting information, the prompt must be satisfied before any other subcommand or command is accepted. Under the FTP Client API, the user program has the option to issue GETL or TERM even when a password or accounting information is expected. If the request is TERM, the interface generates a QUIT

subcommand, which is accepted and stops the client process. See “GETL” on page 422, “TERM” on page 428, and “Prompts from the client” on page 457 for more information about how the FTP Client API handles prompts.

- The FTP Client API requires a secondary subcommand parameter with an SCMD PROXY request. See “SCMD” on page 417 and “Prompts from the client” on page 457 for more information.

FTP Client Application Interface (FCAI) control block

The user program written in Cobol, C, Assembler, and PL/I and the FTP Client Application Programming Interface use the FCAI control block to describe an instance of use of the interface. The space for this control block is acquired by the user program.

Tip: REXX programs do not use an FCAI control block. For REXX programs, see “FTP Client Application Interface (FCAI) stem variables” on page 407.

Requirement: The FCAI control block must be aligned on at least a fullword boundary and reside in primary storage.

Guideline: FCAI_Map can be altered to embed in a structure that generates multiple copies of the FCAI. If this is done, ensure that additional storage in FCAI_UserArea is acquired in fullword increments to preserve the alignment of each copy of the FCAI control block.

Table 9 is a layout of the control block. The **Type** column indicates the type of value that the field contains: text (all text fields must be in EBCDIC), binary, or undefined. This column also contains the following information:

- (I) to indicate input from the user program. If the field is defined by values that appear in a table following Table 9, there is a reference to that table.
- (O) to indicate output from the interface program. If the field is defined by values that appear in a table following Table 9, there is a reference to that table.
- (R) to indicate a reserved field.
- (U) to indicate user area.

The field names in this table are the names used for Assembler and PL/I. All sections of this topic use this name syntax with the following exceptions:

- “Sending requests to the FTP Client API” on page 413 uses the COBOL syntax; the field names contain a dash (-) instead of an underscore (_).
- “FTP Client API C functions” on page 430 uses the C syntax; the field names are identical to Assembler and PL/I, but the constant definitions are all in upper case.

Other than these differences for C and COBOL, the field names in the supplied macros and samples for each language are similar.

Table 9. FCAI control block

Field name	Length	Offset	Description	Type
FCAI_Map	256 and higher	0	FCAI control block.	various
FCAI_DefinedFields	76	0	Fields defined to the interface.	various
FCAI_Eyecatcher	4	0	Eyecatcher= FCAI; this field is required.	(I) text

Table 9. FCAI control block (continued)

Field name	Length	Offset	Description	Type
FCAI_Size	2	4	Size of FCAI; this field is required and has a minimum value of 256.	(I) binary
FCAI_Version	1	6	Version of FCAI; this field is required.	(I - see Table 10 on page 401) binary
FCAI_PollWait	1	7	POLL wait timer in seconds (see "FCAI_PollWait — specifying a wait time before POLL" on page 463).	(I) binary
FCAI_ReqTimer	1	8	Request timer in minutes or 0 for none (see "FCAI_ReqTimer — controlling requests that retrieve results from the spawned z/OS FTP client process" on page 463).	(I) binary
FCAI_TraceIt	1	9	Trace indicator for this request (see "Using the FTP Client API trace" on page 465).	(I - see Table 11 on page 401) binary
FCAI_TraceID	3	10	ID used in a trace record. This value is used only when a request initiates the interface trace function and does not change thereafter.	(I) text
FCAI_TraceCAPI	1	13	TRACECAPI value on FTP.DATA statement.	(O - see Table 12 on page 402) binary
FCAI_TraceStatus	1	14	Status of the trace (see "FCAI_Status_TraceFailed and FCAI_TraceStatus — reporting failures in the interface trace function" on page 462).	(O - see Table 13 on page 402) binary
FCAI_TraceSClass	1	15	SYSOUT class for trace file. This value is used only when a request initiates the interface trace function and does not change thereafter.	(I) text
FCAI_TraceName	8	16	ddname of the trace file.	(O) text
FCAI_Token	4	24	Interface token (do not alter after INIT).	(O) binary
FCAI_RequestID	4	28	Last request (for example, 'SCMD').	(O) text
FCAI_RCV	16	32	Request completion values (see "Interpreting results from an interface request" on page 459).	(O) binary
FCAI_Result	1	32	Request result (the return code register also contains this value).	(O - see Table 14 on page 402) binary
FCAI_Status	1	33	Status code.	(O - see Table 15 on page 403) binary
FCAI_IE	1	34	Interface error.	(O - see Table 16 on page 403) binary

Table 9. FCAI control block (continued)

Field name	Length	Offset	Description	Type
FCAI_CEC	1	35	Client error code (see FTP return codes in the <i>z/OS Communications Server: IP User's Guide and Commands</i>).	(O) binary
FCAI_ReplyCode	2	36	Server reply code or 0 if no reply (see FTPD reply codes in <i>z/OS Communications Server: IP and SNA Codes</i>).	(O) binary
FCAI_SCMD	1	38	Subcommand code (see FTP subcommand codes in <i>z/OS Communications Server: IP User's Guide and Commands</i>).	(O) binary
Reserved	1	39	Reserved.	(R) undefined
FCAI_ReturnCode	4	40	Return code (see Table 13 on page 402 and Table 16 on page 403 for errors that have associated return code data).	(O) binary
FCAI_ReasonCode	4	44	Reason code (see Table 13 on page 402 and Table 16 on page 403 for errors that have associated reason code data).	(O) binary
Summary fields for output lines that are held in the interface buffer				
FCAI_NumberLines	4	48	Number of output lines returned by the request.	(O) binary
FCAI_LongestLine	4	52	Size of the longest line.	(O) binary
FCAI_SizeAll	4	56	Size of all output lines.	(O) binary
FCAI_SizeMessages	4	60	Size of all message lines.	(O) binary
FCAI_SizeReplies	4	64	Size of all reply lines.	(O) binary
FCAI_SizeList	4	68	Size of all list lines.	(O) binary
FCAI_SizeTrace	4	72	Size of all trace lines.	(O) binary
FCAI_PID	4	76	Process ID of FTP client.	(O) binary
Reserved and user areas				
FCAI_ReservedForInterface	176	80	Reserved.	(R) undefined
FCAI_UserArea	0 to unlimited	256	Start of user area. It is not necessary to add the size of the user area to the value in FCAI_Size.	(U) undefined

Table 10. FCAI_Version field value

Name	Value	Description
FCAI_Version_Number	1	Version number

For more information about the values found in Table 11, see “Using the FTP Client API trace” on page 465.

Table 11. FCAI_TraceIt field value

Name	Value	Description
FCAI_TraceIt_No	0	Do not trace this request.
FCAI_TraceIt_Yes	1	Trace this request.

Table 12. FCAI_TraceCAPI field value

Name	Value	Description
FCAI_TraceCAPI_C	0	Trace according to FCAI_TraceIt.
FCAI_TraceCAPI_A	1	Trace all events.
FCAI_TraceCAPI_N	2	Trace no events.

For more information about the values found in Table 13, see “FCAI_Status_TraceFailed and FCAI_TraceStatus — reporting failures in the interface trace function” on page 462.

Table 13. FCAI_TraceStatus field value

Name	Value	Description	Additional information returned with FCAI_Status_TraceFailed
FCAI_TraceStatus_OK	0	Tracing OK or not started	None
FCAI_TraceStatus_StorageErr	1	Failed to acquire or access storage	FCAI_ReturnCode = GETMAIN return code
FCAI_TraceStatus_AllocErr	2	Allocation error	FCAI_ReturnCode = S99ERROR value or 8; FCAI_ReasonCode = S99ERSN for SMS error, S99INFO otherwise
FCAI_TraceStatus_OpenErr	3	Open error	FCAI_ReturnCode contains the OPEN return code or FCAI_ReasonCode contains the ABEND code
FCAI_TraceStatus_WriteErr	4	Write error	FCAI_ReasonCode=ABEND code
FCAI_TraceStatus_CloseErr	5	Close error	FCAI_ReturnCode contains the CLOSE return code or FCAI_ReasonCode contains the ABEND code
FCAI_TraceStatus_SysoutClassErr	6	FCAI_TraceSClass contains a Sysout output class that is not valid	None

For more information about the values found in Table 14, see “Interpreting results from an interface request” on page 459.

Table 14. FCAI_Result field value

Name	Value	Description
FCAI_Result_OK	0	OK with no additional status.
FCAI_Result_Status	1	Status code returned in FCAI_Status.
FCAI_Result_IE	2	Interface error returned in FCAI_IE.
FCAI_Result_CEC	3	Client Error Code returned in FCAI_CEC.
FCAI_Result_NoMatch	4	GETL request has no matches.
FCAI_Result_UnusableFCAI	17	FCAI is not usable.
FCAI_Result_TaskMismatch	18	Task is not the same as INIT task.
FCAI_Result_CliProcessKill	32	TERM issued BPX1KIL to end the client process. This is informational.

For more information about the values found in Table 15, see “Prompts from the client” on page 457 and “Interpreting results from an interface request” on page 459.

Table 15. FCAI_Status field values

Name	Value	Description	Additional Information
FCAI_Status_InProgress	1	Subcommand is in-progress.	This status is returned for an SCMD issued in no-wait mode or if FCAI_ReqTimer expires on an SCMD issued in wait mode, and on any subsequent POLL requests until the SCMD completes.
FCAI_Status_PromptPass	2	Request prompted for a PASS subcommand.	The interface accepts only SCMD PASS or a GETL request until the prompt is satisfied or this instance of the interface is terminated.
FCAI_Status_Acct	3	Request prompted for an ACCT subcommand.	The interface accepts only SCMD ACCT or a GETL request until the prompt is satisfied or this instance of the interface is terminated.
FCAI_Status_TraceFailed	200	The interface trace failed on this request and has been disabled.	This status is <i>added</i> to any other status returned. See “FCAI_Status_TraceFailed and FCAI_TraceStatus — reporting failures in the interface trace function” on page 462 for more information.

Table 16. FCAI_IE field values

Name	Value	Description	Additional Information
General interface errors			
FCAI_IE_RequestMissing	1	Request ID is missing.	Request ID parameter not passed to EZAFTPXS.
FCAI_IE_RequestUnknown	2	Unknown request.	Request ID not INIT, TERM, POLL, GETL, or SCMD.
FCAI_IE_ParmMissing	3	Parameter missing.	Required parameter not passed to EZAFTPXS.
FCAI_IE_ParmStorageErr	4	Storage error for a parameter.	Parameter list points to inaccessible storage.
FCAI_IE_TooManyParameters	5	More parameters were passed than are defined for this request type.	Failure to include VL on an Assembler language call to EZAFTPXS is one cause.
FCAI_IE_ControlErr	6	Error altering an open file descriptor.	FCAI_ReturnCode and FCAI_ReasonCode contain values set by BPX1FCT in <i>z/OS UNIX System Services Programming: Assembler Callable Services Reference</i> .

Table 16. FCAI_IE field values (continued)

Name	Value	Description	Additional Information
FCAI_IE_InternalErr	7	Internal error in the interface.	For example, allocated buffer not found in chain; see “FCAI_IE_InternalErr — unanticipated exceptional conditions in the interface” on page 464.
FCAI_IE_LengthInvalid	8	Negative or zero length.	For example, zero buffer length with GETL; see “FCAI_IE_LengthInvalid — improper lengths passed to the interface” on page 462.
INIT errors			
FCAI_IE_APIAlreadyInit	16	Interface already initialized.	This FCAI was used on a prior INIT request.
FCAI_IE_InitParmTooBig	17	INIT parameter is too big.	FTP start parms string exceeds 2393 bytes.
FCAI_IE_APILoadFailed	18	The load of the interface failed.	FCAI_ReturnCode and FCAI_ReasonCode contain values set by the BLDL service.
FCAI_IE_NoTokenAddr	19	Token address is 0.	This FCAI has not been initialized and the current request is not INIT.
FCAI_IE_BadTokenAddr	20	Bad token field.	FCAI_Token is not valid.
FCAI_IE_GetWorkareaFailed	21	Error acquiring workarea.	FCAI_ReturnCode contains the value returned by the GETMAIN service.
FCAI_IE_ReqTimerExpired	22	INIT timed out waiting for output from the client.	See “FCAI_ReqTimer — controlling requests that retrieve results from the spawned z/OS FTP client process” on page 463 for more information.
FCAI_IE_TooManyInitParms	23	More than 30 separate tokens were passed in the start parameters.	Tokens are defined as characters or punctuation surrounded by whitespace.
FCAI_IE_TooManyEnvVars	24	More than nine environment variables were passed on the INIT.	Use _CEE_ENVFILE= <i>lfs_filename</i> to pass more than nine environment variables.
FCAI_IE_CreatePipeErr	26	Error creating pipe to the client.	FCAI_ReturnCode and FCAI_ReasonCode contain values set by BPX1PIP in the <i>z/OS UNIX System Services Programming: Assembler Callable Services Reference</i> .
FCAI_IE_SpawnErr	27	Error spawning the client.	FCAI_ReturnCode and FCAI_ReasonCode contain values set by BPX1SPN in the <i>z/OS UNIX System Services Programming: Assembler Callable Services Reference</i> .
SCMD errors			
FCAI_IE_ScmdParmTooBig	32	SCMD subcommand string too long.	SCMD subcommand parameter string must not exceed 2064 bytes.

Table 16. FCAI_IE field values (continued)

Name	Value	Description	Additional Information
FCAI_IE_UNKMode	33	Mode parameter value incorrect.	Mode parameter value must be W or N.
FCAI_IE_PassPromptErr	34	The current SCMD request is in error because PASS is required.	A prior request set FCAI_Status_PromptPass and the current SCMD is not PASS.
FCAI_IE_AcctPromptErr	35	The current SCMD request is in error because ACCT is required.	A prior request set FCAI_Status_PromptAcct and the current SCMD is not ACCT.
FCAI_IE_AlreadyInProgress	37	The current SCMD request is in error because an SCMD is in-progress.	A prior SCMD returned FCAI_Status_InProgress. Issue a POLL request to complete the prior SCMD.
FCAI_IE_CliProcessStopped	38	The current request is in error because the client process was stopped normally with a QUIT subcommand.	Only GETL can be issued prior to TERM when the client has processed a QUIT subcommand; the current request is not GETL or TERM.
FCAI_IE_WriteErr	41	Error writing to the client.	FCAI_ReturnCode and FCAI_ReasonCode contain values set by BPX1WRT in the <i>z/OS UNIX System Services Programming: Assembler Callable Services Reference</i> .
INIT, SCMD, and POLL errors			
FCAI_IE_ReadErr	42	Error reading from the client.	FCAI_ReturnCode and FCAI_ReasonCode contain values set by BPX1RED in the <i>z/OS UNIX System Services Programming: Assembler Callable Services Reference</i> .
FCAI_IE_CliProcessBroken	47	Client process broken; send a TERM request.	A previous error was encountered when communicating with the client or the client has terminated unexpectedly. Only GETL or TERM are accepted when this occurs.
POLL errors			
FCAI_IE_NotInProgress	48	A POLL request was issued when no subcommand was in-progress.	Processing can continue normally with the next request.
GETL errors			
FCAI_IE_UnknownOperation	64	GETL OPERATION parameter is not recognized.	OPERATION must be FIND or COPY.
FCAI_IE_UnknownType	65	GETL TYPE parameter is not recognized.	TYPE must be one of the following: M - client message R - server reply T - client trace L - LIST/NLST output A - any

Table 16. FCAI_IE field values (continued)

Name	Value	Description	Additional Information
FCAI_IE_UnknownSequence	66	GETL FIND SEQUENCE parameter is not recognized.	Sequence must be one of the following: F - first L - last N - next
FCAI_IE_VectorStorageErr	67	The buffer described by the vector cannot be accessed.	See the description of the VECTOR parameter in "Parameter values that are set by the application" on page 424.
FCAI_IE_BufferTooSmall	68	The buffer described by the vector is too small to hold the first line of returned output.	See "GETL" on page 422 for more information.
FCAI_IE_TraceIDTooBig	69	Length of traceID must be 0-3 characters.	Set only by the REXX FTP Client API.
FCAI_IE_TraceSClassTooBig	70	Length of traceSClass value must be 0-3 characters.	Set only by the REXX FTP Client API.
FCAI_IE_UnknownTraceIt	71	The traceIt value is not recognized.	Set only by the REXX FTP Client API.
FCAI_IE_ReqTimerInvalid	72	The request timer value is not in the range 0-255.	Set only by the REXX FTP Client API.
FCAI_IE_LinesParmTooBig	73	The GETL lines stem name is more than 200 characters in length.	Set only by the REXX FTP Client API.
FCAI_IE_PollWaitInvalid	74	The pollWait value is not in the range 0-255.	Set only by the REXX FTP Client API.
FCAI_FCAI_IE_NumTraceInvalid	75	The numTrace value is not in the range 1-1000000.	Set only by the REXX FTP Client API.
FCAI_IE_FcaiMapParmTooBig	76	The FCAI stem name is more than 250 characters in length.	Set only by the REXX FTP Client API.
FCAI_IE_EnvVarStorageErr	77	Unable to allocate storage for an environment variable.	Set only by the REXX FTP Client API.
FCAI_IE_SysoutClassErr	78	FCAI_TraceSClass contains a Sysout output class that is not valid.	Set only by the REXX FTP Client API.

Define the space for the FCAI by including the appropriate macro or source copy book in your program as follows:

- EZAFTPKA — Assembler macro found in SEZACMAC
- EZAFTP KC — COBOL copy book found in SEZANMAC
- ftpcapi.h — C header file found in /usr/include/
- EZAFTP KP — PL/I include deck found in SEZANMAC

FTP Client Application Interface (FCAI) stem variables

The user-written REXX program uses an FCAI stem to represent an instance of use of the interface.

Guideline: When passing the REXX stem to the REXX FTP Client API, include the terminating period (.).

Table 17 describes the stem variables that are created from the REXX stem. The Type column indicates the type of value that the field contains, which can be decimal or binary.

All stem variables, with the exception of *stem.FCAI_Map*, are output fields. The *stem.FCAI_Map* variable is used internally by the REXX FTP Client API function package.

Table 17. FCAI stem variables

Name	Type	Description	Additional Information
<i>stem.FCAI_Result</i>	Decimal	This stem variable corresponds to the FCAI_Result field in the FCAI_Map control block.	See Table 15 on page 403.
<i>stem.FCAI_IE</i>	Decimal	This stem variable corresponds to the FCAI_IE field in the FCAI_Map control block.	See Table 16 on page 403.
<i>stem.FCAI_CEC</i>	Decimal	This stem variable corresponds to the FCAI_CEC field in the FCAI_Map control block.	See FTP return codes in the <i>z/OS Communications Server: IP User's Guide and Commands</i> .
<i>stem.FCAI_ReplyCode</i>	Decimal	This stem variable corresponds to the FCAI_ReplyCode in the FCAI_Map control block.	See FTPD reply codes in <i>z/OS Communications Server: IP and SNA Codes</i> .
<i>stem.FCAI_TraceStatus</i>	Decimal	This stem variable corresponds to the FCAI_TraceStatus field in the FCAI_Map control block. It can be used to determine whether the last FTP Client API trace succeeded or failed.	See Table 13 on page 402.
<i>stem.FCAI_ReturnCode</i>	Decimal	This stem variable corresponds to the FCAI_ReturnCode field in the FCAI_Map control block.	See Table 13 on page 402 and Table 16 on page 403.
<i>stem.FCAI_ReasonCode</i>	Decimal	This stem variable corresponds to the FCAI_ReasonCode field in the FCAI_Map control block.	See Table 13 on page 402 and Table 16 on page 403.

Table 17. FCAI stem variables (continued)

Name	Type	Description	Additional Information
<i>stem.FCAI_SCMD</i>	Decimal	This stem variable corresponds to the FCAI_SCMD field in the FCAI_Map control block.	See FTP subcommand codes in the <i>z/OS Communications Server: IP User's Guide and Commands</i> .
<i>stem.FCAI_Map</i>	Binary	This stem variable contains a binary representation of the FCAI_Map control block, plus additional fields used by the REXX FTP Client API.	This stem variable must not be modified by the REXX program.

Predefined REXX variables

Predefined REXX variables make symbolic references easier and more consistent. Instead of using a numeric or non-numeric value, you can use the predefined variable, which defines that value for you. Table 18 shows the data type and value for each predefined variable. The predefined variables are created on the first CREATE request issued by a REXX program.

Requirement: The REXX program must treat the predefined variables as read only, and must not assign any values to them.

The predefined variables, listed alphabetically, are shown in Table 18.

Table 18. Predefined REXX variables

Name	Type	Description	Additional Information
FCAI_CEC_ALREADY_CONNECTED	Decimal	6	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_AUTHENTICATION	Decimal	17	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_CLIENT_ERR	Decimal	24	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_CONNECT_FAILED	Decimal	8	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_CONVERSION	Decimal	21	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_FILE_ACCESS	Decimal	18	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_FILE_READ	Decimal	19	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_FILE_WRITE	Decimal	20	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_INPUT_ERR	Decimal	12	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_INTERNAL_ERROR	Decimal	1	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_INVALID_ENVIRONMENT	Decimal	15	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_INVALID_PARAM	Decimal	4	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_LOGIN_FAILED	Decimal	11	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_NOT_ENABLED	Decimal	16	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_NOTFOUND	Decimal	14	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_OPEN_IOSTREAM_FAILED	Decimal	5	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_PROXY_ERR	Decimal	22	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_SERVER_ERROR	Decimal	2	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_SESSION_ERROR	Decimal	10	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_SQL_ERR	Decimal	23	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_TIMEOUT	Decimal	9	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_CEC_USAGE	Decimal	7	Can be stored in <i>stem.FCAI_CEC</i>
FCAI_GETL_ANY_LINE	Char	A	Value for type parameter on GETL_FIND or GETL_COPY requests

Table 18. Predefined REXX variables (continued)

Name	Type	Description	Additional Information
FCAI_GETL_FIND_FIRST	Char	F	Value for sequence parameter on GETL_FIND requests
FCAI_GETL_FIND_LAST	Char	L	Value for sequence parameter on GETL_FIND requests
FCAI_GETL_FIND_NEXT	Char	N	Value for sequence parameter on GETL_FIND requests
FCAI_GETL_LIST_LINE	Char	L	Value for sequence parameter on GETL_FIND requests
FCAI_GETL_MESSAGE_LINE	Char	M	Value for type parameter on GETL_FIND or GETL_COPY requests
FCAI_GETL_REPLY_LINE	Char	R	Value for type parameter on GETL_FIND or GETL_COPY requests
FCAI_GETL_TRACE_LINE	Char	T	Value for type parameter on GETL_FIND or GETL_COPY requests
FCAI_IE_ACCTPROMPTERR	Decimal	35	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_ALREADYINPROGRESS	Decimal	37	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_APIALREADYINIT	Decimal	16	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_APILOADFAILED	Decimal	18	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_BADTOKENADDR	Decimal	20	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_BUFFERTOOSMALL	Decimal	68	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_CLIPROCESSBROKEN	Decimal	47	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_CLIPROCESSSTOPPED	Decimal	38	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_CONTROLERR	Decimal	6	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_CREATEPIPEERR	Decimal	26	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_ENVVARSTORAGEERR	Decimal	77	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_GETWORKAREAFAILED	Decimal	21	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_INITPARMTOOBIG	Decimal	17	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_INTERNALERR	Decimal	7	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_LENGTHINVALID	Decimal	8	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_LINESPARMTOOBIG	Decimal	73	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_NOTINPROGRESS	Decimal	48	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_NOTOKENADDR	Decimal	19	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_PARMMISSING	Decimal	3	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_PARMSTORAGEERR	Decimal	4	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_PASSPROMPTERR	Decimal	34	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_POLLWAITINVALID	Decimal	74	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_READERR	Decimal	42	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_REQTIMEREXPIRED	Decimal	22	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_REQTIMERINVALID	Decimal	72	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_REQUESTMISSING	Decimal	1	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_REQUESTUNKNOWN	Decimal	2	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_SCMDPARMTOOBIG	Decimal	32	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_SPAWNERR	Decimal	27	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_SYSOUTCLASSERR	Decimal	78	Can be stored in <i>fcaiMap.FCAI_IE</i>
FCAI_IE_TOOMANYENVVARS	Decimal	24	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_TOOMANYINITPARMS	Decimal	23	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_TOOMANYPARAMETERS	Decimal	5	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_TRACEIDTOOBIG	Decimal	69	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_TRACESCLASSTOOBIG	Decimal	70	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_UNKMODE	Decimal	33	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_UNKNOWNOPERATION	Decimal	64	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_UNKNOWNSEQUENCE	Decimal	66	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_UNKNOWNTRACEIT	Decimal	71	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_UNKNOWNWNTYPE	Decimal	65	Can be stored in <i>stem.FCAI_IE</i>
FCAI_IE_VECTORSTORAGEERR	Decimal	67	Can be stored in <i>stem.FCAI_IE</i>

Table 18. Predefined REXX variables (continued)

Name	Type	Description	Additional Information
FCAI_IE_WRITEERR	Decimal	41	Can be stored in <i>stem.FCAI_IE</i>
FCAI_MODE_NOWAIT	Char	N	Can be stored in <i>stem.FCAI_IE</i>
FCAI_MODE_WAIT	Char	W	Can be stored in <i>stem.FCAI_IE</i>
FCAI_RESULT_CEC	Decimal	-3	Can be returned by an FTPAPI call
FCAI_RESULT_IE	Decimal	-2	Can be returned by an FTPAPI call
FCAI_RESULT_INPROGRESS	Decimal	1	Can be stored in <i>stem.FCAI_IE</i>
FCAI_RESULT_OK	Decimal	0	Can be stored in <i>stem.FCAI_Result</i> or returned by an FTPAPI call
FCAI_RESULT_PROMPTACCT	Decimal	3	Can be stored in <i>stem.FCAI_Result</i> or returned by an FTPAPI call
FCAI_RESULT_PROMPTPASS	Decimal	2	Can be stored in <i>stem.FCAI_Result</i> or returned by an FTPAPI call
FCAI_RESULT_REXXERROR	Decimal	-19	Can be returned by an FTPAPI call
FCAI_RESULT_UNUSABLEFCAI	Decimal	-17	Can be returned by an FTPAPI call
FCAI_SCMD_ACCT	Decimal	3	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_AMBIGUOUS	Decimal	1	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_APPE	Decimal	4	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_ASCII	Decimal	5	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_BIG5	Decimal	57	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_BINARY	Decimal	6	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_BLOCK	Decimal	58	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_CC	Decimal	77	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_CD	Decimal	7	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_CDUP	Decimal	51	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_CLEAR	Decimal	72	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_CLOSE	Decimal	8	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_COMPRESS	Decimal	59	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_CPROTECT	Decimal	73	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_DEBUG	Decimal	11	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_DELETE	Decimal	13	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_DELIMIT	Decimal	12	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_DIR	Decimal	14	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_DUMP	Decimal	70	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_EBCDIC	Decimal	15	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_EUCKANJI	Decimal	46	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request
FCAI_SCMD_FEAT	Decimal	78	Can be stored in <i>fcaiMap.FCAI_SCMD</i> by a GET_FCAI_MAP request

Table 18. Predefined REXX variables (continued)

Name	Type	Description	Additional Information
FCAI_SCMD_FILE	Decimal	60	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_GENHELP	Decimal	2	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_GET	Decimal	16	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_GLOB	Decimal	65	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_HANGEUL	Decimal	53	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_HELP	Decimal	17	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_IBMKANJI	Decimal	47	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_JIS78KJ	Decimal	48	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_JIS83KJ	Decimal	49	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_KSC5601	Decimal	54	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_LANG	Decimal	79	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_LCD	Decimal	41	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_LMKDIR	Decimal	45	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_LOCSITE	Decimal	42	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_LOCSTAT	Decimal	18	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_LPWD	Decimal	43	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_LS	Decimal	20	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_MDELETE	Decimal	21	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_MGET	Decimal	22	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_MKD	Decimal	44	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_MODE	Decimal	23	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_MPUT	Decimal	24	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_NOOP	Decimal	25	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_OEEXCL	Decimal	68	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_OPEN	Decimal	10	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_PASS	Decimal	26	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_PRIVATE	Decimal	74	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_PROMPT	Decimal	66	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_PROTECT	Decimal	75	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_PROXY	Decimal	61	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request

Table 18. Predefined REXX variables (continued)

Name	Type	Description	Additional Information
FCAI_SCMD_PUT	Decimal	27	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_PWD	Decimal	28	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_QUIT	Decimal	29	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_QUOTE	Decimal	30	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_RECORD	Decimal	62	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_RENAME	Decimal	31	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_REST	Decimal	56	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_RMD	Decimal	52	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_SAFE	Decimal	76	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_SCHINESE	Decimal	63	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_SENDPORT	Decimal	32	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_SENDSITE	Decimal	33	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_SITE	Decimal	34	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_SJISKJ	Decimal	50	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_SREST	Decimal	80	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_STAT	Decimal	35	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_STREAM	Decimal	64	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_STRU	Decimal	36	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_SUNIQUE	Decimal	37	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_SYST	Decimal	38	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_TCHINESE	Decimal	55	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_TSO	Decimal	9	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_TYPE	Decimal	40	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_UCS2	Decimal	67	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_UNKNOWN	Decimal	99	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_USER	Decimal	19	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_VERBOSE	Decimal	71	Can be stored in fcAiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_TASK_CLIPROCESSKILL	Decimal	-32	Can be stored in <i>stem</i> .FCAI_Result or returned by an FTPAPI function call
FCAI_TASK_TASKMISMATCH	Decimal	-18	Can be stored in <i>stem</i> .FCAI_Result or returned by an FTPAPI function call

Table 18. Predefined REXX variables (continued)

Name	Type	Description	Additional Information
FCAI_TRACE_DATASET_NAME	String		Data set name to which the REXX FTP Client trace is written if the REXX FTP Client API is active, or an empty string (") if the trace is not active.
FCAI_TRACE_DATASET_RETCODE	Integer		Return code from REXX FTP Client API tracing from the last invocation of the REXX FTP Client API
FCAI_TRACE_WORKAREA	Binary		Binary data used by the REXX FTP Client API when writing trace records. Rule: This variable must not be modified by the REXX program.
FCAI_TRACECAPA_A	Decimal	1	Can be stored in fcaiMap.FCAI_TraceCAPI by a GET_FCAI_MAP request
FCAI_TRACECAPA_C	Decimal	0	Can be stored in fcaiMap.FCAI_TraceCAPI by a GET_FCAI_MAP request
FCAI_TRACECAPA_N	Decimal	2	Can be stored in fcaiMap.FCAI_TraceCAPI by a GET_FCAI_MAP request
FCAI_TRACESTATUS_ALLOCERR	Decimal	2	Can be stored in fcaiMap.FCAI_TraceStatus by a GET_FCAI_MAP request
FCAI_TRACESTATUS_CLOSEERR	Decimal	5	Can be stored in fcaiMap.FCAI_TraceStatus by a GET_FCAI_MAP request
FCAI_TRACESTATUS_OK	Decimal	0	Can be stored in fcaiMap.FCAI_TraceStatus by a GET_FCAI_MAP request
FCAI_TRACESTATUS_OPENERR	Decimal	3	Can be stored in fcaiMap.FCAI_TraceStatus by a GET_FCAI_MAP request
FCAI_TRACESTATUS_STORAGEERR	Decimal	1	Can be stored in fcaiMap.FCAI_TraceStatus by a GET_FCAI_MAP request
FCAI_TRACESTATUS_WRITEERR	Decimal	4	Can be stored in fcaiMap.FCAI_TraceStatus by a GET_FCAI_MAP request

Sending requests to the FTP Client API

This section contains the description, syntax, parameters, and other related information for each of the following requests submitted to this API:

- INIT
- SCMD
- POLL
- GETL
- TERM

Note: The text in this topic sometimes uses shorthand when referring to results. When a request is said to return FCAI_IE_LengthInvalid, for example, it means that FCAI_Result contains FCAI_Result_IE and FCAI_IE contains FCAI_IE_LengthInvalid. See “Interpreting results from an interface request” on page 459.

Tip: This section provides information on the Cobol, C, Assembler and PL/I programming languages. For REXX programs, see “FTP Client API REXX function” on page 433.

INIT

The user program issues the INIT request to initialize the FTP Client API. This call is the first call to the interface and is made only one time for each FCAI control block that defines an instance of use of the interface.

Rules:

- Align the FCAI on at least a fullword boundary.
- The FCAI must reside in primary space (not a dataspace).
- Initialize the FCAI by performing the steps listed in step 3 in “Application tasks for the INIT request” on page 416.
- The interface enforces a minimum FCAI-ReqTimer of five minutes on the INIT request. The caller can specify a longer interval to use (or specify that no timer is to be used). See “FCAI_ReqTimer — controlling requests that retrieve results from the spawned z/OS FTP client process” on page 463.
- Before exiting, your application should issue a TERM request for each INIT request.

Example of the INIT call instruction

WORKING-STORAGE SECTION.

```
COPY EZAFTPKC.
01 REQUEST-INIT PIC X(4) VALUE IS 'INIT'.
01 START-PARM.
    05 PARM-LEN PIC 9(2) COMP-5 VALUE IS 6.
    05 PARM-VAL PIC X(6) VALUE IS '-n t1s'.

01 ENV-VAR-LIST.
    05 ENV-VAR-COUNT PIC 9(8) COMP-5 VALUE IS 3.
    05 ENV-VAR1-LEN PIC 9(8) COMP-5.
    05 ENV-VAR2-LEN PIC 9(8) COMP-5.
    05 ENV-VAR3-LEN PIC 9(8) COMP-5.
    05 ENV-VAR1-P USAGE IS POINTER.
    05 ENV-VAR2-P USAGE IS POINTER.
    05 ENV-VAR3-P USAGE IS POINTER.

01 ENV-VAR-VALUES.
    05 ENV-VAR1.
        10 ENV-VAR PIC X(17) VALUE IS '_CEE_DMPTARG=/etc'.
        10 FILLER PIC X(1) VALUE LOW-VALUES.
    05 ENV-VAR2.
        10 ENV-VAR PIC X(18) VALUE IS '_BPX_JOBNAME=MYJOB'.
        10 FILLER PIC X(1) VALUE LOW-VALUES.
    05 ENV-VAR3.
        10 ENV-VAR PIC X(20) VALUE IS 'NLSPATH=/u/user79/%N'.
        10 FILLER PIC X(1) VALUE LOW-VALUES.
```

PROCEDURE DIVISION.

```
MOVE LOW-VALUES TO FCAI-Map.
MOVE FCAI-C-Eyecatcher TO FCAI-Eyecatcher.
MOVE FCAI-C-Version-1 TO FCAI-Version.
MOVE LENGTH OF FCAI-Map TO FCAI-Size.

MOVE LENGTH OF ENV-VAR1 TO ENV-VAR1-LEN.
SET ENV-VAR1-P TO ADDRESS OF ENV-VAR1.
MOVE LENGTH OF ENV-VAR2 TO ENV-VAR2-LEN.
SET ENV-VAR2-P TO ADDRESS OF ENV-VAR2.
MOVE LENGTH OF ENV-VAR3 TO ENV-VAR3-LEN.
SET ENV-VAR3-P TO ADDRESS OF ENV-VAR3.
```

```
CALL 'EZAFTPKS' USING FCAI-Map REQUEST-INIT START-PARM ENV-VAR-LIST.
```

Since both START-PARM and ENV-VAR-LIST are optional,
use OMITTED for START-PARM if it is not to be passed on
a CALL that passes ENV-VAR-LIST:

```
CALL 'EZAFTPKS' USING FCAI-Map REQUEST-INIT OMITTED ENV-VAR-LIST.
```

For equivalent PL/I and assembler language declarations, see “Converting parameter descriptions” on page 397.

Parameter values that are set by the application

FCAI-Map

Storage area (defined in EZAFTPCK for COBOL) used to save information about the requests using the FTP Client API.

REQUEST-TYPE

A 4-byte field that contains INIT.

START-PARM

An optional parameter that comprises a 2-byte length followed by a string that contains parameters that are valid to enter on a z/OS FTP command.

PARM-LEN

A 2-byte binary field that contains the length of PARM-VAL.

PARM-VAL

Storage containing the parameters for the z/OS FTP command. See the FTP subcommands information in the *z/OS Communications Server: IP User's Guide and Commands* for the parameters that are valid when using the FTP command.

Requirement: Align START-PARM on at least a halfword boundary.

Tip: Start parameters that include a host name or IP address cause the client to perform an implicit OPEN to connect to that host, which suspends the application until the connection is complete. If this delay is undesirable during INIT, use a subsequent SCMD OPEN instead of specifying a host in the start parameters.

ENV-VAR-LIST

An optional parameter that comprises a series of contiguous fullwords (4 bytes each) that are used to describe environment variables that are passed on the spawn of the FTP Client.

See the following for information about FTP environment variables:

- Defining environment variables for the FTP server (optional) in the *z/OS Communications Server: IP Configuration Guide*
- FTP server environment variables in *z/OS Communications Server: IP Configuration Reference*
- Environment variables in *z/OS Communications Server: IP User's Guide and Commands*

Also see the z/OS UNIX System Services and z/OS Language Environment libraries of publications for information concerning environment variables.

ENV-VAR-COUNT

Count of environment variables (n) to be passed. There can be 1–9 environment variables.

ENV-VAR1_LEN through ENV-VARn_LEN (maximum of nine)

The fullword length of each environment variable found in ENV-VAR1 through ENV-VARn.

ENV-VAR1_P through ENV-VARn-P (maximum of nine)

The address of each environment variable found in ENV-VAR1 through ENV-VARn.

Rules:

- Each environment variable passed in the ENV-VAR-LIST must be a NULL terminated string; that is, X'00' follows the last text character. Set the corresponding ENV-VARn-LEN field to the length of the environment variable text, plus 1 for the NULL terminator.
- Ensure that no duplicate environment variables are specified.
- Do not pass environment variable _CEE_RUNOPTS on INIT. The environment variables are established too late in the spawn() process for run-time options to be honored. See the *z/OS Language Environment Programming Guide* for information about using CEEDOPT or CEEBXITA to specify run-time options for the FTP client process.
- Do not pass environment variables in the _CEE_ENVFILE file that are required to be used in the spawn() process. The _CEE_ENVFILE variable is processed after the spawn() processing is complete. Variables like _BPX_JOBNAME should be specified as one of the nine environment variables in the ENV-VAR-LIST
- Run-time options and environment variables that are specified on the EXEC statement or in CEEUOPT for the application program are not available to the spawned FTP client process.
- Align ENV-VAR-LIST on at least a fullword boundary.

Parameter values that are returned to the application

The results of the request are returned in the FCAI-Result field. See “Interpreting results from an interface request” on page 459. See “Other output that is returned to the application” on page 456 for a discussion of the output and statistics returned by the request.

Guidelines for INIT results:

- If the INIT request returns FCAI-IE-CliProcessBroken, check FCAI-CEC for a client error code that might have been returned to explain the failure.
- FCAI-Token is zeros when the interface fails to initialize.

Application tasks for the INIT request

Before you begin: Create an FCAI control block for use by the interface.

Perform the following steps to issue the INIT request:

1. Specify start parameters for the FTP client (optional).

2. Specify a list of environment variables to pass to the FTP client (optional).

3. Initialize the FCAI.
 - a. Clear the entire block to 0.
 - b. Set the eyecatcher (or FCAI-Eyecatcher) to FCAI.
 - c. Set the *size* field (or FCAI-Size) to 256 or greater.
 - d. Set the version number (or FCAI-Version) to 1.

4. Set FCAI-ReqTimer to the desired value.

5. Set FCAI-TraceIt as desired for tracing. A request that initiates the interface trace also uses FCAI-TraceID and FCAI-TraceSClass.

-
6. Issue the INIT request.

 7. Check the result of the request.

 8. Check the results from an implicit OPEN, if one was performed.

 9. Retrieve the FTP client output with GETL (optional).

After the INIT request completes, you can issue a subsequent SCMD or TERM request.

SCMD

The user program issues the SCMD request to send a subcommand to the FTP client.

Example of the SCMD call instruction

WORKING-STORAGE SECTION.

```
COPY EZAFTPKC.
01 REQUEST-SCMD PIC X(4) VALUE IS 'SCMD'.
01 SCMD-MODE-W PIC X(1) VALUE IS 'W'.
01 SCMD-MODE-N PIC X(1) VALUE IS 'N'.
01 SUBCOMMAND.
   05 SUBCOMMAND-LEN PIC 9(2) COMP-5 VALUE IS 13.
   05 SUBCOMMAND-VAL PIC X(13) VALUE IS 'open hostname'.
```

PROCEDURE DIVISION.

```
CALL 'EZAFTPKS' USING FCAI-Map REQUEST-SCMD SUBCOMMAND
SCMD-MODE-W.
```

For equivalent PL/I and assembler language declarations, see “Converting parameter descriptions” on page 397.

Parameter values that are set by the application

FCAI-Map

Storage area (defined in EZAFTPKC for COBOL) used to save information about the requests using the FTP Client API.

REQUEST-TYPE

A 4-byte field that contains SCMD.

SUBCOMMAND

A required parameter that comprises a 2-byte field, followed by a string with a z/OS FTP subcommand. See the introductory material at the beginning of this topic for a discussion of FTP subcommands and commands.

SUBCOMMAND-LEN

A 2-byte binary field that contains the length of SUBCOMMAND-VAL.

SUBCOMMAND-VAL

Storage that contains a z/OS FTP subcommand. Leading spaces are not allowed; begin the subcommand in the first column of this storage area. A null terminator is unnecessary but is accepted if included. See FTP subcommands in the *z/OS Communications Server: IP User's Guide and Commands* for the supported subcommands and their parameters.

Rule: Align SUBCOMMAND on at least a halfword boundary.

MODE

An optional parameter that indicates whether the interface should wait for the subcommand to complete before returning to the caller. If the parameter is not present, the default is to wait. The parameter is a 1-byte field that contains W for wait mode or N for no-wait mode.

An SCMD issued in no-wait mode returns to the caller as soon as the interface sends the subcommand. Issue a POLL request to retrieve the results from the subcommand. If you choose no-wait mode, you cannot start a new SCMD request with this FCAI until you execute a POLL against this FCAI that indicates that the outstanding subcommand is complete.

An SCMD issued in wait mode essentially polls the client for you until the subcommand completes. The automatic polling operates similarly to the progressive wait timer described in “FCAI_PollWait — specifying a wait time before POLL” on page 463. It polls initially after 1 second and then doubles the wait time before each read to a maximum interval of 16 seconds until data is returned or the subcommand completes.

If you choose wait mode, you can limit the length of time the interface waits for completion by using FCAI-ReqTimer. See “FCAI_ReqTimer — controlling requests that retrieve results from the spawned z/OS FTP client process” on page 463. If FCAI-ReqTimer causes the subcommand to be interrupted, a POLL is required to complete and retrieve the results of the subcommand before another subcommand can be issued.

Processing rules for SCMD QUIT:

- The MODE parameter is ignored for SCMD QUIT. The QUIT subcommand is always issued in wait mode.
- The interface accepts a TERM and generates a successful SCMD QUIT even when the interface does not accept an explicit SCMD QUIT request. See “TERM” on page 428 and “Prompts from the client” on page 457 for more information.
- After SCMD QUIT has successfully completed, only GETL and TERM are accepted for this FCAI.
- Failure to issue SCMD QUIT can strand the client process. See “TERM” on page 428 for tips on preventing stranded clients.
- SCMD and POLL requests between SCMD QUIT and TERM return the interface error FCAI_IE_CliProcessStopped.

Parameter values that are returned to the application

The results of the request are returned in the FCAI-Result field; see “Interpreting results from an interface request” on page 459. See “Other output that is returned to the application” on page 456 for a description of the output and statistics that are returned by the request.

Guidelines for SCMD results:

- If FCAI-Result contains FCAI-Result-Status, additional processing is required by the user program:
 - If the FCAI-Status field is FCAI-Status-InProgress, the request was issued in no-wait mode or the FCAI-ReqTimer value was exceeded. Issue a POLL request to obtain the final results.
 - If the FCAI-Status field is FCAI-Status-PromptPass, the request was accepted but the next SCMD request must be a PASS subcommand. This status is applicable for the USER subcommand.
 - If the FCAI-Status field is FCAI-Status-PromptAcct, the request was accepted, but the next SCMD request must be an ACCT subcommand. This status is applicable for the USER, PASS, and CD subcommands.
 - If FCAI-Status contains FCAI-Status-TraceFailed or higher, the interface trace function failed on this request. Subtract FCAI-Status-TraceFailed from FCAI-Status to obtain the remaining value for FCAI-Status. See “FCAI_Status_TraceFailed and FCAI_TraceStatus — reporting failures in the interface trace function” on page 462.
- After an INIT that results in an implicit OPEN, FCAI-SCMD contains FCAI-SCMD-OPEN, and FCAI-ReplyCode contains the last reply from the server.

See “Prompts from the client” on page 457 for a discussion of how prompts are handled when the z/OS FTP client is invoked from the FTP Client API.

Application tasks for the SCMD request

Before you begin: Have an initialized interface defined by an FCAI and an active z/OS FTP client process (not stopped or broken).

Perform the following steps to issue the SCMD request:

1. Specify a MODE parameter (optional).

2. Specify a subcommand parameter string.

3. Set FCAI-ReqTimer to the desired value.

4. Set FCAI-TraceIt as desired for tracing. A request that initiates the interface trace also uses FCAI-TraceID and FCAI-TraceSClass.

5. Issue the SCMD request.

6. Check the result of the request.

7. Issue one or more POLL requests to complete the subcommand, if necessary.

8. Check the result of the completed subcommand.

9. Retrieve the FTP client output with GETL (optional).

After the SCMD request completes, you can issue a subsequent SCMD or TERM request.

POLL

The user program issues the POLL request to complete and retrieve the results from a prior SCMD request.

The POLL request is rejected if a prior subcommand is not in-progress. That is, a prior SCMD or POLL request must have returned the status code FCAI-Status-InProgress; otherwise, the current POLL request returns FCAI-IE-NotInProgress.

Issuing an SCMD request in no-wait mode enables you to perform other program functions while the subcommand is running. While you are performing these functions, the FTP client might be writing a large amount of data to the interface.

Guideline: Avoid large intervals of time between POLL requests. Large time intervals can cause a pipe overflow and lead to an error or a wait in the client until the results from the subcommand are retrieved.

Each POLL request copies up to 32 KB of data from the client to the interface buffer even if the subcommand has not yet completed. The actual amount that is returned depends upon the size of the output, the timing of the request, and how much space is available in the current part of the interface buffer.

See “GETL” on page 422 for a discussion of the interface buffer. See “FCAI_PollWait — specifying a wait time before POLL” on page 463 for more information about managing POLL requests.

Example of the POLL call instruction

WORKING-STORAGE SECTION.

```
COPY EZAFTPKC.  
01 REQUEST-POLL PIC X(4) VALUE IS 'POLL'.
```

PROCEDURE DIVISION.

```
CALL 'EZAFTPKS' USING FCAI-Map REQUEST-POLL.
```

For equivalent PL/I and assembler language declarations, see “Converting parameter descriptions” on page 397.

Parameter values that are set by the application

FCAI-Map

Storage area (defined in EZAFTPKC for COBOL) used to save information about the requests using the FTP Client API.

REQUEST-TYPE

A 4-byte field that contains POLL.

Parameter values that are returned to the application

The results of the request are returned in the FCAI-Result field. See “Interpreting results from an interface request” on page 459. See “Other output that is returned to the application” on page 456 for a discussion of the output and statistics returned by the request.

Guidelines for POLL results:

- If FCAI-Result contains FCAI-Result-Status, additional processing is required by the user program:
 - If the FCAI-Status field is FCAI-Status-InProgress, a prior SCMD request issued in no-wait mode or one that exceeded the FCAI-ReqTimer value is still in-progress. Obtain the final results with another POLL request.
 - If the FCAI-Status field is FCAI-Status-PromptPass, the prior request has now completed but the next SCMD request must be a PASS subcommand. This status is applicable for the USER subcommand.
 - If the FCAI-Status field is FCAI-Status-PromptAcct, the prior request has now completed but the next SCMD request must be an ACCT subcommand. This status is applicable for the USER, PASS, and CD subcommands.
 - If FCAI-Status contains FCAI-Status-TraceFailed or higher, the interface trace function failed on this request. Subtract FCAI-Status-TraceFailed from FCAI-Status to obtain the remaining value for FCAI-Status. See “FCAI_Status_TraceFailed and FCAI_TraceStatus — reporting failures in the interface trace function” on page 462 for information about FCAI-Status-TraceFailed.
- Aside from interface errors, POLL itself returns only FCAI-Status-InProgress. All other results are those of the prior SCMD request.
- If the application issues POLL until FCAI-IE-NotInProgress is returned, the results from the prior subcommand are lost. Issue POLL only until FCAI-Status-InProgress is no longer returned.

Application tasks for the POLL request

Before you begin: Have an initialized interface defined by an FCAI and an active z/OS FTP client process (not stopped or broken). Ensure that a prior SCMD or POLL request returned FCAI-Status-InProgress.

Perform the following steps to issue the POLL request:

1. Set FCAI-PollWait to the desired value.

2. Set FCAI-TraceIt as desired for tracing. A request that initiates the interface trace also uses FCAI-TraceID and FCAI-TraceSClass.

3. Issue the POLL request.

4. Check the result of the request.

5. Repeat the POLL request to complete the subcommand, if necessary.

6. Check the result of the completed subcommand.

7. Retrieve the FTP client output with GETL (optional).

After the POLL request completes, you can issue a subsequent SCMD or TERM request.

GETL

The user program issues the GETL request to find and copy lines of output that are returned from the FTP client by an INIT, SCMD, or POLL request. The request copies output from the client into a space that is acquired by the user and described to the interface by a vector. The vector includes the address of the space, the ALET for the space (or 0), and the length of the space. In the description that follows, this space is referred to as the *user's buffer*.

The internal buffers maintained by the FTP Client API are referred to collectively as the *interface buffer*. The interface buffer is populated upon return from an INIT, an SCMD issued in wait mode, or a POLL issued for an SCMD that did not complete. The interface acquires the number of 32 KB interface buffer parts that are needed to contain the output from the request.

- INIT and SCMD in wait mode acquire interface buffer parts automatically until the request completes.
- POLL either fills the remainder of the current part of the interface buffer or acquires a new part before reading from the client. The actual amount returned for each POLL request varies depending on available output and current buffer utilization.
- The interface retains and reuses all interface buffer parts until TERM is issued to minimize acquiring and freeing storage. If the user program does not wish to retain acquired buffer parts, it should issue TERM to free them and then optionally reinitialize the interface.
- An SCMD QUIT that is generated by TERM uses the interface buffer, but only to allow the interface trace function to print the subcommand results. All interface buffer parts are freed by TERM and output is not available to the user program after that time.

See "Other output that is returned to the application" on page 456 for a discussion of the statistics that can be used in preparing to retrieve the output (such as setting the size of your buffer).

GETL has an OPERATION parameter that specifies whether to FIND and retrieve one line or to COPY many lines of output.

GETL has a TYPE parameter that specifies the type of lines to copy to the user's buffer. Lines are classified into the following types: client messages, server replies, DIR or LS subcommand output, and trace output.

Each line of output that is written to the user's buffer has a 3-byte prefix that consists of a 1-byte ID and a 2-byte length field. The IDs are defined as follows:

ID	Definition
M	Message from the client
R	Reply from the server
T	Trace output that is generated as a result of activating FTP trace and extended trace with DEBUG and DUMP subcommands
L	Data from a DIR or LS subcommand
A last line marker is appended to the user's buffer upon completion of a successful COPY request. It has a length of zero and one of the following IDs:	
Y	Last line of output for a request so far (The subcommand is not complete or you ran out of room in the user's buffer. More output will be or is already available to be retrieved.)

ID	Definition
Z	Last line of output for the request

There are two concepts about the output lines in the user's buffer and the interface buffer that are important to understand. The first concept is the *last line marker* of output in the user's buffer after a successful COPY operation. (A successful FIND copies exactly one line without writing a last line marker.) The last line marker has an ID of either Y or Z and a length field of x'0000.

When the last line marker ID is Y, partial output has been copied to the user's buffer and the user program must now handle one or both of the following conditions:

- A prior subcommand has not yet completed, and more results must be retrieved with POLL.
- The COPY operation found more lines of output than can fit in the user's buffer.

Tip: A last line marker with an ID of Y does not guarantee that any more lines of the type specified on the COPY were generated by the request.

When the last line marker ID is Z, the last line of output for the request has been retrieved.

See "Other output that is returned to the application" on page 456 and "POLL" on page 420.

Rule: A GETL request that specifies the same user's buffer overwrites the contents of that buffer; it does not append lines to the user's buffer.

Guideline: If the returned output must be preserved, move it or update the buffer pointer before a subsequent GETL request. This does not have to be done when a COPY immediately follows a FIND using the same criteria, as discussed in the remainder of this section.

The second concept is that of the *current line* in the interface buffer. Initially, the current line is the first line of the interface buffer. The current line changes as you successfully FIND and COPY lines of output, as follows:

- After you successfully FIND a line, that line is copied into the buffer addressed by your vector and becomes the current line in the interface buffer.
 - An immediate COPY of one or more lines includes the line that was found on the preceding FIND when the TYPE parameter is the same on both operations.
 - A subsequent FIND (NEXT, FIRST, or LAST) advances the current line pointer to the appropriate matching line and copies that line.
- When you successfully COPY lines, the current line pointer first advances to the next matching line of the type specified on the COPY. At the end of the COPY operation, the current line pointer advances to the location that immediately follows the last copied line, which can be a line of any TYPE or the end of the interface buffer.
- When the current line pointer advances past a line in the interface buffer, that line can be located only by resetting the current line to a point at or before the desired line with a FIND FIRST with TYPE A (any) or the TYPE that matches the desired line.

- If FIND or COPY does not locate any matching lines, GETL returns FCAI-Result-NoMatch in FCAI-Result. This result or any result other than FCAI-Result-OK means that the current line pointer was unchanged by the request.

See “Using the FIND and COPY operations” on page 426 for more information about FIND and COPY.

Example of the GETL call instruction

WORKING-STORAGE SECTION.

```
COPY EZAFTPKC.
01 REQUEST-GETL PIC X(4) VALUE IS 'GETL'.
01 OPERATION PIC X(4).
01 TYPE PIC X(1).
01 SEQUENCE PIC X(1).
01 VECTOR.
    05 BUFF-ADDR USAGE IS POINTER.
    05 BUFF-ALET USAGE IS POINTER.
    05 BUFF-LEN PIC 9(8) COMP-5.
01 BUFFER PIC X(100).
```

PROCEDURE DIVISION.

```
MOVE 'FIND' TO OPERATION.
MOVE 'M' TO TYPE.
MOVE 'F' TO SEQUENCE.
```

```
SET BUFF-ADDR TO THE ADDRESS OF BUFFER.
MOVE ZEROS TO BUFF-ALET.
MOVE SIZE OF BUFFER TO BUFF-LEN.
```

```
CALL 'EZAFTPKS' USING FCAI-Map REQUEST-GETL OPERATION
TYPE SEQUENCE VECTOR.
```

```
MOVE 'COPY' TO OPERATION.
MOVE 'M' TO TYPE.
```

```
SET BUFF-ADDR TO THE ADDRESS OF BUFFER.
MOVE ZEROS TO BUFF-ALET.
MOVE SIZE OF BUFFER TO BUFF-LEN.
```

```
CALL 'EZAFTPKS' USING FCAI-Map REQUEST-GETL OPERATION
TYPE VECTOR.
```

For equivalent PL/I and assembler language declarations, see “Converting parameter descriptions” on page 397.

Parameter values that are set by the application

FCAI-Map

Storage area (defined in EZAFTPKC for COBOL) used to save information about the requests using the FTP Client API.

REQUEST-TYPE

A 4-byte field that contains GETL.

OPERATION

A 4-byte field that contains the operation to be performed. The OPERATION values are:

FIND Find an output line matching TYPE and copy it into the user’s buffer.

COPY Copy all remaining output lines matching TYPE into the user's buffer.

TYPE A 1-byte field that indicates what type of output is requested by OPERATION. The TYPE values are:

- M** Message from the client.
- R** Reply from the server.
- L** List data from a DIR or LS subcommand.
- T** Trace output from debug or dump.
- A** Any type of output line.

SEQUENCE (FIND operation only)

A 1-byte parameter that indicates which of the output lines of TYPE to FIND. SEQUENCE values are:

- F** Find the first line of the requested TYPE.
- N** Find the next line of the requested TYPE.
- L** Find the last line of the requested TYPE.

Tip: The SEQUENCE parameter is not included on the call for a COPY operation, which always begins with the first line that matches TYPE at or after the current line.

VECTOR

A 3-word vector that describes the user's buffer that receives a copy of an output line. VECTOR values are:

BUFF-ADDR

The address of the buffer used in the operation.

BUFF-ALET

The ALET of the buffer pointed to by BUFF-ADDR. This can be 0. If not 0, the ALET must reside in the PASN-AL for the application or be a public entry in the DU-AL. All programs that the application invokes must have the authority to access the space. See the *z/OS MVS Programming: Extended Addressability Guide* for information about using ALETs to access data spaces.

BUFF-LEN

The length of the buffer used in the operation. For a COPY operation the interface reserves 4 bytes at the end of the buffer to ensure that there is room for the Y or Z line.

Rule: Align VECTOR on at least a fullword boundary.

Parameter values that are returned to the application

The results of the request are returned in the FCAI-Result field. See "Interpreting results from an interface request" on page 459.

Guidelines for GETL results:

- If FCAI-Result contains FCAI-Result-NoMatch:
 - If OPERATION was FIND and SEQUENCE was FIRST or LAST, no lines of the specified type currently exist in the interface buffer.
 - If OPERATION was FIND and SEQUENCE was NEXT, or if OPERATION was COPY, no lines of the specified type exist between the current line pointer and the end of the interface buffer.

- FCAI-IE-BufferTooSmall indicates that the user's buffer will not hold the first (or only) matching line that was found in the interface buffer.

Tip: FCAI-IE-TooManyParameters might indicate that a SEQUENCE parameter was included for a COPY operation.

Application tasks for the GETL request

Before you begin: Have an initialized interface defined by an FCAI.

Perform the following steps to issue the GETL request:

1. Specify the desired operation (FIND or COPY).

2. Specify the desired line type.

3. Specify the desired sequence (for FIND).

4. Specify the buffer vector that describes the user's buffer.

5. Set FCAI-TraceIt as desired for tracing. A request that initiates the interface trace also uses FCAI-TraceID and FCAI-TraceSClass.

6. Issue the GETL request.

7. Check the result of the request.

8. Issue one or more POLL requests to retrieve output for a subcommand that is in-progress.

9. Issue another GETL request, optionally changing the operation, sequence, or type of line.

After the GETL request completes, you can issue a subsequent SCMD or TERM request.

Using the FIND and COPY operations

The FIND operation syntax for COBOL is as follows:

```
CALL 'EZAFTPKS'  
USING FCAI-Map REQUEST-GETL OPERATION TYPE SEQUENCE VECTOR.
```

A FIND operation copies one line from the interface buffer into the user's buffer described by the vector. The FIND locates and copies the line indicated by the sequence parameter [F (first), L (last), or N (next)] that matches the requested TYPE.

The COPY operation syntax for COBOL is as follows:

```
CALL 'EZAFTPKS'  
USING FCAI-Map REQUEST-GETL OPERATION TYPE VECTOR.
```

A COPY operation copies one or more lines from the interface buffer into the user's buffer described by the vector. The COPY searches for a line at or after the current line that matches the TYPE parameter. It copies from that line through the last line of output in the interface buffer that matches TYPE.

The COPY stops when the user's buffer is full or when there are no more lines of the requested type in the interface buffer. The last line marker is written into the user's buffer after the last output line of the requested type (the interface reserves 4 bytes in the user's buffer to ensure that there is room for the marker).

If the FIND or COPY operation does not locate any matching lines, GETL returns FCAI-Result-NoMatch in FCAI-Result and does not change the current line pointer. The user's buffer contents are not predictable when this result is returned.

Tips:

- If you want to set the current line pointer to the top of the interface buffer, use a GETL request with TYPE set to A (any type of output line) and SEQUENCE set to F (first).
- GETL FIND and COPY cannot find output that has not yet been retrieved from the client and copied to the interface buffer. When handling the results from an incomplete subcommand, any POLL might append data to the interface buffer, but the results are only complete after a POLL request does not return FCAI-Status-InProgress.

Assembler language GETL example: Assume that an SCMD was issued with the subcommand **LS a***. Following are the lines of output that are in the buffer at the end of the SCMD request. (The lines have been numbered for this example.)

```

1  T 0026 CA0149 dirlist: entered for ls command
2  T 0017 CA0798 getList: entered
3  M 0018 >>> PORT 9,42,105,93,4,8
4  R 0014 200 Port request OK.
5  M 000B >>> NLST a*
6  R 0013 125 List started OK
7  T CA1275 rcvListData: entered
8  L 0001 a
9  L 0002 ab
10 R 0020 250 List completed successfully.
11 T 0022 CA1521 resetTS: value of rcode = 0
12 M 0018 Command(00-20-NLST-250):
13 Z 0000

```

The following fields are also defined in the user program:

GETL	DC	C'GETL'
FIND	DC	C'FIND'
COPY	DC	C'COPY'
FIRST	DC	C'F'
NEXT	DC	C'N'
LAST	DC	C'L'
*		
MESSAGE	DC	C'M'
REPLY	DC	C'R'
LIST	DC	C'L'
ANY	DC	C'A'
*		
	DS	0F
BUFFVEC	DS	0XL12

```

BUFFADDR DC    A(BUFFER)  THIS BUFFER IS LARGER THAN TOTAL OUTPUT RETURNED
          DC    F'0'
BUFFLENG DC    A(L'BUFFER)
BUFFER    DC    CL'4096'

```

These calls retrieve the following output lines:

```

CALL EZAFTPKS,(FCAI_Map,GETL,FIND,MESSAGE,FIRST,BUFFVEC)
      copies line 3: M 0018 >>> PORT 9,42,105,93,4,8
CALL EZAFTPKS,(FCAI_Map,GETL,FIND,MESSAGE,NEXT,BUFFVEC)
      copies line 5: M 000B >>> NLST a*
CALL EZAFTPKS,(FCAI_Map,GETL,FIND,MESSAGE,NEXT,BUFFVEC)
      copies line 12: M 0018 Command(00-20-NLST-250):
CALL EZAFTPKS,(FCAI_Map,GETL,FIND,MESSAGE,NEXT,BUFFVEC)
      returns FCAI_Result_NoMatch
CALL EZAFTPKS,(FCAI_Map,GETL,FIND,REPLY,LAST,BUFFVEC)
      copies line 10: R 0020 250 List completed successfully.
CALL EZAFTPKS,(FCAI_Map,GETL,FIND,LIST,FIRST,BUFFVEC)
      copies line 8: L 0001 a
CALL EZAFTPKS,(FCAI_Map,GETL,COPY,LIST,BUFFVEC)
      copies line 8: L 0001 a
      copies line 9: L 0002 ab
      copies line 13: Z 0000

```

TERM

The user program issues the TERM request to terminate this instance of interface use. A TERM request is accepted by the interface at any time.

For the most orderly termination of the interface, the client program should ensure that no SCMDs are in-progress and then issue SCMD with the QUIT subcommand before using the TERM request. To assist the caller with exceptional conditions, TERM performs the following steps unless the client process is broken:

1. If the TERM request detects that a prior SCMD is in-progress, it generates up to three POLL requests at 16-second intervals in an attempt to retrieve the results from the client. Any generated POLL requests appear in the interface trace (if active) with (Generated by TERM) appended to the request record along with any results that are retrieved.
2. If the TERM request detects that a QUIT has not yet been issued to the client, it generates an SCMD QUIT request. The generated SCMD QUIT request appears in the interface trace (if active) with (Generated by TERM) appended to the request record along with any results that are retrieved.

Rule: The interface enforces a minimum FCAI-ReqTimer of 2 minutes on the generated QUIT. The caller can specify a longer interval (or specify that no timer is to be used). See “FCAI_ReqTimer — controlling requests that retrieve results from the spawned z/OS FTP client process” on page 463.

If the client process is broken, or the interface fails to complete a subcommand that was in-progress, or a generated QUIT fails to complete successfully, then the interface issues BPX1KIL with no signal to kill the client process. The reason for the failure is reported on a prior request or in the interface trace.

When TERM completes, FCAI-Token is zeros and only an INIT request is accepted by the interface using this FCAI.

Rule: Never update FCAI-Token or attempt to reinstate it after TERM. If FCAI-Token is corrupted, you must terminate your application to free acquired storage and stop the client process.

Guidelines:

- To ensure that complete results are returned, always check for FCAI-Status-InProgress upon return from an SCMD request and POLL to complete the request if it did not complete. Although TERM generates POLLS to the client, this does not guarantee that the prior subcommand will complete or that all results will be returned. The three POLL requests can retrieve a maximum of 96 KB. The actual amount that is retrieved depends upon the timing of the request and current buffer utilization.
- Set FCAI-TraceIt to FCAI-TraceIt-Yes on TERM to see the results from any generated POLLS or QUIT. The results are available only in the trace because TERM frees the interface buffer.
- If an active FCAI is cleared and reused on an INIT request without first issuing an SCMD QUIT or TERM, the FTP client process that was associated with it is stranded. The client child process can persist until the parent process terminates. Failure to issue TERM after SCMD QUIT retains acquired storage until the parent task terminates. It is especially important in long-running application processes to terminate instances of the interface that are no longer in use.

Example of the TERM call instruction

WORKING-STORAGE SECTION.

```
COPY EZAFTPKC.
01 REQUEST-TERM PIC X(4) VALUE IS 'TERM'.
```

PROCEDURE DIVISION.

```
CALL 'EZAFTPKS' USING FCAI-Map REQUEST-TERM.
```

For equivalent PL/I and assembler language declarations, see “Converting parameter descriptions” on page 397.

Parameter values that are set by the application

FCAI-Map

Storage area (defined in EZAFTPKC for COBOL) used to save information about the requests using the FTP Client API.

REQUEST-TYPE

A 4-byte field that contains TERM.

Parameter values that are returned to the application

The results of the request are returned in the FCAI-Result field. See “Interpreting results from an interface request” on page 459.

Guidelines for TERM results:

- See the results from any generated POLL or SCMD QUIT requests in the interface trace output (if active).
- FCAI-Result-OK means the interface terminated normally (whether or not additional requests were automatically generated).
- FCAI-Result-CliProcessKill means that BPX1KIL was issued to end the client process.
- FCAI-Result-Status means that FCAI-Status-TraceFailed was returned.
- FCAI-Status-TraceFailed can also be returned when FCAI-Result contains FCAI-Result-CliProcessKill.

Application tasks for the TERM request

Perform the following steps to issue the TERM request:

1. Set FCAI-ReqTimer to the desired value (for a generated SCMD QUIT).
2. Set FCAI-TraceIt as desired for tracing. A request that initiates the interface trace also uses FCAI-TraceID and FCAI-TraceSClass.
3. Issue the TERM request.
4. Check the result of the request.
5. See the interface trace output for the results of any generated POLL or SCMD QUIT requests.

After the TERM request completes, you can reinitialize the FCAI and issue a subsequent INIT request.

FTP Client API C functions

For the C programming language, in-line static functions are provided in the `ftpcapi` header file. These functions provide typecasting compiler checking and tools to facilitate calling the interface from a C language program.

The following are in-line static functions:

- `FAPI_INIT` initializes the interface.
- `FAPI_SCMD` sends an FTP subcommand.
- `FAPI_POLL` checks the status of an outstanding subcommand.
- `FAPI_GETL_COPY` retrieves output related to a subcommand and copies to a user buffer.
- `FAPI_GETL_FIND` retrieves output related to a subcommand and searches for a line of a specific type of output.
- `FAPI_TERM` ends the interface.

FAPI_INIT

Use the `FAPI_INIT` function to initialize the FTP Client API. See “Sending requests to the FTP Client API” on page 413 for more information about an INIT request.

`FAPI_INIT` accepts the following parameters:

FCAI_MAP

A pointer to an `fcai_map_t` structure used to save information about the request using the FTP Client API.

startparm

A pointer to a NULL terminated string that contains the parameters for the z/OS FTP command. See the FTP command — Entering the FTP environment information in the *z/OS Communications Server: IP User's Guide and Commands* for descriptions of valid FTP command parameters.

envVars

A pointer to an `fcai_envvarlist_t` structure that contains a count followed by an array of up to nine pointers to NULL terminated strings. Each string represents an environment variable definition.

FAPI_INIT example:

```
#define OPENSTRING    "-w 300 127.0.0.1 21 (trace"
fcai_map_t          fcai;
fcai_envvarlist_t    my_envvars;

my_envvars.envVarCount = 3;
my_envvars.envVarEnt[0] = "_CEE_DMPTARG=/etc";
my_envvars.envVarEnt[1] = "_BPX_JOBNAME=MYJOB";
my_envvars.envVarEnt[2] = "NLSPATH=/u/myuser/%N";

memset(&fcai,          0, sizeof(fcai));
fcai.FCAI_Eyecatcher = FCAI_EYECATCHER;
fcai.FCAI_Size       = FCAI_NUMINTERFACEBYTES;
fcai.FCAI_Version    = FCAI_VERSION;
rc = FAPI_INIT(&fcai, OPENSTRING, &my_envvars);
```

FAPI_SCMD

Use the `FAPI_SCMD` function to issue the SCMD request to send a subcommand to the FTP Client API. See “Sending requests to the FTP Client API” on page 413 for more information about an SCMD request.

`FAPI_SCMD` accepts the following parameters:

FCAI_MAP

A pointer to an `fcai_map_t` structure used to save information about the request using the FTP Client API.

subcommand

A pointer to a NULL-terminated string that contains a z/OS FTP client subcommand with its parameters.

mode A required character that indicates whether the interface should wait for the subcommand to complete before returning to the caller. Valid values are `FAPI_MODE_WAIT` or `FAPI_MODE_NOWAIT`.

FAPI_SCMD example:

```
/* Assume FAPI_INIT was successfully called with &fcai */
/* prior to calling FAPI_SCMD */
rc = FAPI_SCMD(&fcai, "USER user1", FAPI_MODE_WAIT);
```

FAPI_POLL

Use the `FAPI_POLL` function to issue the POLL request to complete and retrieve the results from a prior `FAPI_SCMD` request. See “Sending requests to the FTP Client API” on page 413 for more information on a POLL request.

`FAPI_POLL` accepts the following parameter:

FCAI_MAP

A pointer to an `fcai_map_t` structure used to save information about the request using the FTP Client API.

FAPI_POLL example:

```
/* Assume FAPI_SCMD was successfully called with &fcai */
/* prior to calling FAPI_POLL */
rc = FAPI_POLL(&fcai);
```

FAPI_GETL_COPY

Use the FAPI_GETL_COPY function to issue the GETL COPY request to copy lines of output that are returned from the FTP client by an FAPI_INIT, FAPI_SCMD, or FAPI_POLL functions. See “Sending requests to the FTP Client API” on page 413 for more information on a GETL request.

FAPI_GETL_COPY accepts the following parameters:

FCAI_MAP

A pointer to an fcai_map_t structure used to save information about the request using the FTP Client API.

type A character that indicates what type of output is requested by the copy.

Valid type values include:

Field constant defined in ftpcapi.h	Value	Meaning
FAPI_GETL_MESSAGE_LINE	M	Message from the client.
FAPI_GETL_REPLY_LINE	R	Reply from the server.
FAPI_GETL_TRACE_LINE	T	Trace output from debug or dump.
FAPI_GETL_LIST_LINE	L	List data from a DIR or LS subcommand.
FAPI_GETL_ANY_LINE	A	Any type of output line.

buffer_len

The length of the buffer used for the copy.

buffer

The address of the buffer used in the operation.

FAPI_GETL_COPY example:

```
/* Assume FAPI_INIT was successfully called with &fcai */
/* prior to calling FAPI_GETL_COPY */
char buffer[4096];
rc = FAPI_GETL_COPY(&fcai, FAPI_GETL_LIST_LINE,
    sizeof(buffer), buffer);
```

FAPI_GETL_FIND

Use the FAPI_GETL_FIND function to issue the GETL FIND request to find one line of output returned from the FTP client by an FAPI_INIT, FAPI_SCMD, or FAPI_POLL. See “Sending requests to the FTP Client API” on page 413 for more information on a GETL request.

FAPI_GETL_FIND accepts the following parameters:

FCAI_MAP

A pointer to an fcai_map_t structure used to save information about the request using the FTP Client API.

type A character that indicates what type of output is requested by the copy.

Possible type values include:

Field constant defined in ftpcapi.h	Value	Meaning
FAPI_GETL_MESSAGE_LINE	M	Message from the client.
FAPI_GETL_REPLY_LINE	R	Reply from the server.
FAPI_GETL_TRACE_LINE	T	Trace output from debug or dump.

Field constant defined in ftpcapi.h	Value	Meaning
FAP_GETL_LIST_LINE	L	List data from a DIR or LS subcommand.
FAP_GETL_ANY_LINE	A	Any type of output line.

buffer_len

The length of the buffer used for the copy.

buffer

The address of the buffer used in the operation.

sequence

A character that indicates which of the output lines of type should be found. Valid sequence values include:

Field constant defined in ftpcapi.h	Value	Meaning
FAP_GETL_FIND_FIRST	F	Find the first line of the requested type.
FAP_GETL_FIND_NEXT	N	Find the next line of the requested type.
FAP_GETL_FIND_LAST	L	Find the last line of the requested type.

FAP_GETL_FIND example:

```
/* Assume FAP_INIT was successfully called with &fcai */
/* prior to calling FAP_GETL_FIND */
char buffer[4096];
rc = FAP_GETL_FIND(&fcai, FAP_GETL_LIST_LINE,
    sizeof(buffer), buffer, FAP_GETL_FIND_LAST);
```

FAP_TERM

Use the FAP_TERM function to terminate the FTP Client API instance associated with this FCAI_MAP. See “Sending requests to the FTP Client API” on page 413 for more information on a TERM request.

FAP_TERM allows the following parameter:

FCAI_MAP

A pointer to an fcai_map_t structure used to save information about the request using the FTP Client API.

FAP_TERM example:

```
/* Assume FAP_INIT was successfully called with &fcai */
/* prior to calling FAP_TERM */
rc = FAP_TERM(&fcai);
```

FTP Client API REXX function

A REXX language program requires an intermediate routine to translate from the string format used within REXX programs to the binary format used by the EZAFTPKS program. An external function package is provided that serves as this intermediary, facilitating calling the FTP Client API from a REXX language program.

The REXX FTP Client API is shipped as a default system function package, and is included in the default parameter modules IRXTSPRM, IRXISPRM, and IRXPARMS.

The REXX FTP Client API has been verified to run in the following environments:

- TSO exec
- Batch environment
- ISPF
- UNIX shell

The REXX FTP Client API might run in additional environments, but it has not been tested and verified to work in other environments.

Handling of SIGCHLD signals

The FTP Client API spawns a child process for the z/OS FTP client for each successful INIT request. When running in a Posix environment, the SIGCHLD signal is raised when the FTP client terminates. To avoid creating a zombie process when the child process terminates, the SIGCHLD signal must be caught or ignored.

If, when invoking the first CREATE request, the REXX program does not have a SIGCHLD signal handler and is running in a Posix environment, the REXX FTP Client API requests that the SIGCHLD signal be ignored. If the REXX program has a SIGCHLD handler, no change is made.

REXX FTP Client API trace

The REXX FTP Client API trace is used to debug problems in the REXX FTP Client API function package. The interface cannot be used to debug errors in the trace itself or any error that prevents the interface from accessing the trace data set. The trace writes records for requests to the interface and the results of interface requests.

Tip: The REXX FTP Client API trace is separate from the FTP Client API trace. It uses a different output file and has a different enabling mechanism. The REXX FTP Client API trace is used to debug problems in the REXX FTP Client API function package, while the FTP Client API trace is used to debug problems in the underlying EZAFTPKS interface and record activity and data that are returned to the interface that might not otherwise be available to the application.

The REXX FTP Client API trace is activated by specifying the z/OS UNIX FTP_REXX_TRACE_FILE environment variable or allocating the FTPRXTRC DD name. The REXX FTP Client API first looks for the FTP_REXX_TRACE_FILE environment variable (z/OS UNIX environment only) and then for the FTPRXTRC DD allocation.

The REXX FTP Client API trace can be written to any of the following:

- JES SYSOUT
- An MVS sequential data set (a member of a PDS is not supported); the data set must already exist or be allocated as new with DCB characteristics of an LRECL value in the range 80–256 and a RECFM value of Fixed Block
- A z/OS UNIX file. The file can be either an existing file or a file dynamically allocated by the REXX FTP Client API when needed

Restriction: In order for the REXX FTP Client API to be able to write trace records, the output data set or file must meet the following conditions:

- The data set or file must be a fixed block data set.
- For an MVS data set, the record format must be fixed.
- The data set or file record length must be in the range 80 – 256 bytes.

- The data set or file block size must be a multiple of the record length.
- The data set or file must not be block mode.

Data sets and files created by the REXX FTP Client API meet these conditions by default. If a file or data set is created by some other means, then you must ensure these conditions are met or no trace records will be written.

Specifying the REXX FTP Client API trace output location

Your environment determines the method you use to specify the REXX FTP Client API trace output location.

TSO environment: In the TSO environment, the location specified by the FTPRXTRC DD statement is used as the REXX FTP Client API trace output location. Use the TSO ALLOCATE command to associate FTPRXTRC with these outputs. Following is an example:

```
ALLOC FILE(FTPRXTRC) DA(FTP.TRACE.OUT) NEW LRECL(80) RECFM(F B) TRACK SPACE(10 10)
```

See the *z/OS TSO/E Command Reference* for more details about using the ALLOCATE command.

z/OS UNIX shell environment: In the z/OS UNIX shell environment, use one of the following to specify the REXX FTP Client API trace output location.

- For a new z/OS UNIX file or an existing MVS data set, enter the following:

```
export FTP_REXX_TRACE_FILE=/tmp/myjob.ftpapi.trace
export FTP_REXX_TRACE_FILE="//appl.ftprxtrc"
```

Tip: When the specified MVS data set name is not fully qualified, the user ID is added as the first qualifier for the data set. For example, if USER3 enters this command, it is equivalent to the following:

```
export FTP_REXX_TRACE_FILE="//'USER3.APPL.FTPRXTRC'"
```

- For a non-qualified z/OS UNIX file (myjob.ftpapi.trace) with a current working directory /u/user1/, the file used for tracing is /u/user1/myjob.ftpapi.trace. Enter the following:

```
export FTP_REXX_TRACE_FILE=ftpapi.trace
```

- For a z/OS UNIX file or an MVS data set that is already allocated to a ddname, enter the following:

```
export FTP_REXX_TRACE_FILE="//dd:ddname"
```

Restriction: When using the FTP_REXX_TRACE_FILE environment variable, the maximum length for an MVS data set name or a z/OS UNIX file name is 64 characters. If the data set name or file name length exceeds 64 characters, then the name is truncated. If the MVS data set is not qualified, the 64 character limit is applied after the high-level qualifier is added. If the z/OS UNIX file path is a relative path, then the 64 character limit is calculated after the current working directory name is added.

MVS batch job environment: In the MVS batch environment, an FTPRXTRC DD must be specified in the JCL for trace output to be written. You can write trace output as follows using the JES SYSOUT facility:

```
//FTPRXTRC DD SYSOUT=*
```

z/OS UNIX environment batch job: When using the z/OS UNIX environment from a batch job, use one of the following methods to specify the REXX FTP Client API trace output location:

- If the application exists in a file system, is invoked using the BPXBATSL utility, and does not perform any fork calls, use the FTPRXTRC DD statement to specify the output location as you would with an MVS batch job.
- In all other cases, the FTP_REXX_TRACE_FILE environment variable must be set. When using BPXBATSL or BXPBATCH utilities, set this and any other required environment variables using the STDENV DD statement as follows:
//STDENV DD JCL statement

Following is an example:

```
//STDENV DD DISP=SHR,DSN=USER3.APPL.ENVIRON
```

The STDENV data set can a fixed or variable (nospanned) record format type. It can contain multiple environment variables, as shown in the following sample:

```
FTP_REXX_TRACE_FILE=//'USER3.APPL.RESTRACE'  
_BPXK_SETIBMOPT_TRANSPORT=TCPCS
```

Guidelines:

- Environment variables must start in column 1, and the data set must not contain any sequence numbers (sequence numbers would be treated as part of the environment variable).
- For the FTP_REXX_TRACE_FILE environment variable, any blanks from a fixed format STDENV data set are removed. Because this might not be true for all variables, you should use a variable record format data set.
- For applications that fork, you should use an MVS data set. If you use a file system file, a C03 ABEND might occur when the forked process ends.

Restriction: When using the FTP_REXX_TRACE_FILE environment variable, the maximum length for the MVS data set name or the z/OS UNIX file name is 64 characters. If the data set name or file name length exceeds 64 characters, the name is truncated. If the MVS data set is not qualified, the 64 character limit is applied after the high-level qualifier is added. If the z/OS UNIX file path is a relative path, then the 64 character limit is calculated after the current working directory name is added.

See the *z/OS UNIX System Services Command Reference* for additional considerations when using the BXPBATCH or BPXBATSL utilities.

FTP Client API requests

All REXX FTP Client API requests use the same format:

```
result = ftpapi(stem, request_type, parm1, parm2, ...);
```

The first parameter is the name of the REXX stem that refers to a specific instance of the FTP client environment. The second parameter identifies the API request that is being made (for example INIT or SCMD). The remaining parameters are used to pass data which is sometimes optional for the specific API request that is being made.

Tip: When passing string literals, enclose them in single or double quotes.

All REXX FTP Client API return codes use a consistent format, as shown in Table 19 on page 437.

Table 19. REXX FTP Client API return codes

Code value	Definition
<0	A negative return code indicates that the REXX FTP Client API function failed to complete because of an error.
0	The call completed successfully and there is no additional status available.
>0	The call completed successfully and there is additional information available.

The following requests are supported by the REXX FTP Client API:

- CREATE: Creates a new instance of the interface.
- INIT: Initializes the interface.
- SCMD: Sends an FTP subcommand.
- POLL: Checks the status of an outstanding subcommand.
- GETL_FIND: Retrieves output related to a subcommand and searches for a line of a specific type of output.
- GETL_COPY: Retrieves output related to a subcommand and copies it to a user buffer.
- SET_TRACE: Enables or disables the tracing of subsequent FTP Client API calls within the EZAFTPXS program.
- SET_REQUEST_TIMER: Sets the length of time that the FTP Client API waits for a INIT, SCMD, or TERM request to complete.
- GET_FCAI_MAP: Returns the complete contents of the FCAI_Map structure.
- TERM: Ends the interface.

CREATE request

Format:

```
→ftpapi(—stem—,—'create'— [,—' '—] [,—'A'—] [,—1000000—] )→
```

[,—trace_id—] [,—traceSClass—] [,—traceNum—]

Purpose: Creates a new instance of the FTP Client API.

Parameters:

stem

The name of a stem used to return the FTP client environment. This stem is included as the first parameter on all subsequent REXX FTP Client API calls.

'create'

Requests the creation of a new FTP Client API stem. The string literal is not case sensitive.

trace_id

The identifier to be used in trace records. The ID is a 3-character ID string that is written as the first three characters of each trace record. If not specified, then three blank characters (' ') are used. The string literal is case sensitive.

traceSClass

The SYSOUT class for the trace. Valid values are in the range A – Z and 0 – 9. The default value is A. The string literal is case sensitive.

traceNum

The number of trace records written to the REXX trace file before the file is closed and then reopened. This value can be any decimal value in the range 1 – 1 000 000. The default is 1 000 000 records.

Results:

Table 20. FTP client CREATE request return codes

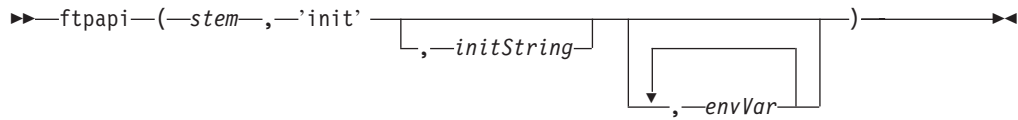
Value	Return codes	Explanation
<0		The call did not complete successfully. The FTP Client API stem variable was not created. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the <i>stem.FCAI_IE</i> stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the <i>stem.FCAI_CEC</i> field stem variable.
	-17 (FCAI_RESULT_UNUSABLESTEM)	An error occurred in creating the stem variable.
	-19 (FCAI_RESULT_REXXERROR)	An error occurred when storing, fetching, or dropping a REXX variable.
≥0		The call completed successfully. The FTP Client API stem variable was successfully created.
	0 (FCAI_RESULT_OK)	No additional information is available.

Examples:

```
rc = ftpapi('fcai.', 'create', 'PAZ')
```

INIT request

Format:



Purpose: Initializes a new instance of the FTP Client API.

Parameters:

stem

The name of a stem used to return the FTP client environment. The stem must have been passed on a prior successful CREATE function call.

'init'

Requests the initialization of an FTP Client API environment. The string literal is not case sensitive.

initString

Start parameters that are valid to enter on a z/OS FTP client command. The string literal is case sensitive.

envVar

Zero to nine environment variables that can be passed to the FTP Client API. The string literals are case sensitive. See the following for information about FTP environment variables:

- Defining environment variables for the FTP server (optional) in the *z/OS Communications Server: IP Configuration Guide*
- FTP server environment variables in the *z/OS Communications Server: IP Configuration Reference*
- Environment variables in *z/OS Communications Server: IP User's Guide and Commands*

Also see the z/OS UNIX System Services library and z/OS Language Environment library of publications for information concerning environment variables.

Rules:

- Do not specify duplicate environment variables.
- Do not pass environment variable `_CEE_RUNOPTS` on an INIT request. The environment variables are established too late in the `spawn()` process for run-time options to be honored. See the *z/OS Language Environment Programming Guide* for information about using the `CEEDOPT` program or the `CEEBXITA` exit to specify run-time options for the FTP client process.
- Do not pass environment variables like `_BPX_JOBNAME` in the `_CEE_ENVFILE` file because the `_CEE_ENVFILE` file processing is too late in the `spawn()` process to set the job name. Specify `_BPX_JOBNAME` as one of the nine environment variables in the `envVar` list.

Results:

Table 21. FTP client INIT request return codes

Value	Return codes	Explanation
<0		The call did not complete successfully. The subcommand was not successfully executed. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the <i>stem.FCAI_IE</i> stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the <i>stem.FCAI_CEC</i> field stem variable.
	-17 (FCAI_RESULT_UNUSABLESTEM)	The stem variable is not usable.
	-19 (FCAI_RESULT_REXXERROR)	An error occurred when storing, fetching, or dropping a REXX variable.
≥0		The call completed successfully. The FTP client environment was initialized.
	0 (FCAI_RESULT_OK)	No additional information is available.

Examples:

```
rc = ftpapi('fcai.', 'init', '-w 300 127.0.0.1 21', '_CEE_DMPTARG=/tmp', '_BPX_JOBNAME=MYJOB')
```

SCMD request

Format:

```
►►—ftpapi—(—stem—,—'scmd' —,—cmd—

|         |
|---------|
| ,—'W'   |
| ,—mode— |

—)——►
```

Purpose: Send a subcommand to the FTP client.

Parameters:

stem

The name of a stem that represents the FTP client environment. The stem must have been passed on a prior successful INIT function call.

'scmd'

Requests that a subcommand be sent to the FTP client. The string literal is not case sensitive.

cmd

The z/OS FTP subcommand. The string literal is case sensitive.

mode

Indicates whether the interface should wait ('W') for the subcommand to complete before returning to the caller, or return immediately ('N') regardless of whether the subcommand has completed. The string literal is not case sensitive.

Results:

Table 22. FTP client SCMD request return codes

Value	Return codes	Explanation
<0		The call did not complete successfully. The subcommand was not successfully executed. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the <i>stem.FCAI_IE</i> stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the <i>stem.FCAI_CEC</i> field stem variable.
	-17 (FCAI_RESULT_UNUSABLESTEM)	The stem variable is not usable.
	-18 (FCAI_TASK_TASKMISMATCH)	The task is not the same as the INIT task.
	-19 (FCAI_RESULT_REXXERROR)	An error occurred when storing, fetching, or dropping a REXX variable.
	-32 (FCAI_TASK_CLIPROCESSKILL)	A TERM request issued BPX1KIL to end the client process.
≥0		The call completed successfully.
	0 (FCAI_RESULT_OK)	The prior request has completed successfully; there are no restrictions on the next subcommand.
	1 (FCAI_RESULT_INPROGRESS)	The prior request is still in-progress.

Table 22. FTP client SCMD request return codes (continued)

Value	Return codes	Explanation
	2 (FCAL_RESULT_PROMPTPASS)	The prior request has completed successfully but the next request must be a PASS subcommand.
	3 (FCAL_RESULT_PROMPTACCT)	The prior request has completed successfully but the next request must be an ACCT subcommand.

Examples:

```
rc = ftpapi('fcai.', 'scmd', 'DIR /tmp/*', 'N')
```

POLL request

Format:

```
➤ ftpapi (—stem—, —'poll'—, —pollWait—) ➤
```

Purpose: Completes and retrieves the results from a prior SCMD request.

Parameters:

stem

The name of a stem that represents the FTP client environment. The stem must have been passed on a prior successful INIT function call and the previous request must have been an SCMD request with the nowait option.

'poll'

Requests that the program wait for a prior SCMD request to complete and retrieve the results. The string literal is not case sensitive.

pollWait

Controls the length of time that the REXX program waits before it checks for output. This parameter is equivalent to the FCAI_PollWait field in the FCAI_MAP structure used by the C and callable FTP Client APIs. Valid values are any decimal value in the range 0 – 255.

Results:

Table 23. FTP client POLL request return codes

Value	Return codes	Explanation
<0		The call did not complete successfully. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the <i>stem.FCAI_IE</i> stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the <i>stem.FCAI_CEC</i> field stem variable.
	-17 (FCAI_RESULT_UNUSABLESTEM)	The stem variable is not usable.
	-18 (FCAI_TASK_TASKMISMATCH)	The task is not the same as the INIT task.
	-19 (FCAI_RESULT_REXXERROR)	An error occurred when storing, fetching, or dropping a REXX variable.
	-32 (FCAI_TASK_CLIPROCESSKILL)	A TERM request issued BPX1KIL to end the client process.
≥0		The call completed successfully; the next request can be any subcommand.
	0 (FCAI_RESULT_OK)	The prior request has completed successfully; there are no restrictions on the next subcommand.
	1 (FCAI_RESULT_INPROGRESS)	The prior request is still in-progress.

Table 23. FTP client POLL request return codes (continued)

Value	Return codes	Explanation
	2 (FCAL_RESULT_PROMPTPASS)	The prior request has completed successfully but the next request must be a PASS subcommand.
	3 (FCAL_RESULT_PROMPTACCT)	The prior request has completed successfully but the next request must be an ACCT subcommand.

Examples:

```
rc = ftpapi('fcai.', 'poll')
```

GETL_FIND request

Format:

►►—ftpapi—(—*stem*—,—'getl_find' —,—*lines*—

,—'A'
,— <i>type</i>

,—'N'
,— <i>sequence</i>

—)——►►

Purpose: Finds and returns a line of output that was returned from the FTP client by an INIT, SCMD, or POLL request.

Parameters:

stem

The name of a stem that represents the FTP client environment. The stem must have been passed on a prior successful INIT function call.

'getl_find'

Requests the return of a single line of output that was returned from the FTP client by an INIT, SCMD, or POLL request. The string literal is not case sensitive.

lines

The stem where the output lines are to be copied. The results are:

lines.0

The number of output lines returned. This is 1 if the function result is 0, and 0 if the function result is 4.

lines.id.1

If the lines.0 field is 1, this contains a 1-character line identifier. Possible values are M, R, T, and L. These are described under the *type* parameter.

lines.1

If the lines.0 field is 1, this contains the contents of the output line.

type

A 1-character parameter that indicates what type of output is requested. The string literal is not case sensitive. Possible values are:

- M** Message from the client
- R** Reply from the server
- L** List data from a DIR or LS subcommand
- T** Trace output from debug or dump routine
- A** Any type of output

sequence

A 1-character parameter that indicates which of the output lines specified by the *type* parameter to find. The string literal is not case sensitive. Possible values are:

- F** Find the first line of the requested type.
- N** Find the next line of the requested type.
- L** Find the last line of the requested type.

The default value is N.

Results:

Table 24. FTP client GETL_FIND request return codes

Value	Return codes	Explanation
<0		The call did not complete successfully. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the <i>stem.FCAI_IE</i> stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the <i>stem.FCAI_CEC</i> field stem variable.
	-17 (FCAI_RESULT_UNUSABLESTEM)	The stem variable is not usable.
	-18 (FCAI_TASK_TASKMISMATCH)	The task is not the same as the INIT task.
	-19 (FCAI_RESULT_REXXERROR)	An error occurred when storing, fetching, or dropping a REXX variable.
	-32 (FCAI_TASK_CLIPROCESSKILL)	A TERM request issued BPX1KIL to end the client process.
≥0		The call completed successfully.
	0 (FCAI_RESULT_OK)	A matching output line was found and returned in the <i>lines</i> stem variable.
	4 (FCAI_RESULT_NOMATCH)	The prior request has completed successfully but no matching output line was found.

Examples:

```
rc = ftpapi('fcai.', 'getl_find', 'lines.', 'L')
```

GETL_COPY request

Format:

```
➤➤—ftpapi—(—stem—,—'getl_copy' —,—lines—

|         |
|---------|
| ,—'A'—  |
| ,—type— |

)—➤➤
```

Purpose: Finds and returns all lines of the output that was returned from the FTP client by an INIT, SCMD, or POLL request.

Parameters:

stem

The name of a stem that represents the FTP client environment. The stem must have been passed on a prior successful INIT function call.

'getl_copy'

Requests the return of a single line of output that was returned from the FTP client by an INIT, SCMD, or POLL request. The string literal is not case sensitive.

lines

The stem where the output lines are to be copied. The results are:

lines.0

The number of output lines returned. This is ≥ 1 if the function result is 0, and 0 if the function result is 4.

lines.id.i

The 1-character identifier for line *i* of output. Possible values are M, R, T, and L. These are described under the *type* parameter.

lines.i

The contents of line *i* of output.

type

A 1-character parameter that indicates what type of output is requested. The string literal is not case sensitive. Possible values are:

M Message from the client

R Reply from the server

L List data from a DIR or LS subcommand

T Trace output from debug or dump routine

A Any type of output

The default value is A.

Results:

Table 25. FTP client GETL_COPY request return codes

Value	Return codes	Explanation
<0		The call did not complete successfully. The possible failure return codes are listed in the Return codes column.

Table 25. FTP client GETL_COPY request return codes (continued)

Value	Return codes	Explanation
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the <i>stem.FCAI_IE</i> stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the <i>stem.FCAI_CEC</i> field stem variable.
	-17 (FCAI_RESULT_UNUSABLESTEM)	The stem variable is not usable.
	-18 (FCAI_TASK_TASKMISMATCH)	The task is not the same as the INIT task.
	-19 (FCAI_RESULT_REXXERROR)	An error occurred in storing, fetching, or dropping a REXX variable.
	-32 (FCAI_TASK_CLIPROCESSKILL)	A TERM request issued BPX1KIL to end the client process.
≥0		The call completed successfully.
	0 (FCAI_RESULT_OK)	The matching output lines were found and returned in the <i>lines</i> stem variable.
	4 (FCAI_RESULT_NOMATCH)	The prior request has completed successfully but no matching output lines were found.

Examples:

```
rc = ftpapi('fcai.', 'getl_copy', 'lines.', 'A')
```

SET_TRACE request

Format:

►►—ftpapi—(—*stem*—,—'set_trace' —,—*traceit*—)————►►

Purpose: Enables or disables the tracing of subsequent FTP Client API calls. After each subsequent call, you can examine the `fcai.FCAI_TraceStatus` variable to determine whether the tracing of that specific call was successful.

Parameters:

stem

The name of a stem that represents the FTP client environment. The stem must have been passed on a prior successful INIT function call.

'set_trace'

Indicates whether the trace should be enabled or disabled on subsequent FTP Client API calls. The string literal is not case sensitive.

traceit

Sets the status of the FTP Client API trace. The string literal is not case sensitive. Possible values are:

ON

Begin tracing FTP Client API calls.

OFF

Stop tracing FTP Client API calls.

Results:

Table 26. FTP client SET_TRACE request return codes

Value	Return codes	Explanation
<0		The call did not complete successfully. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the <i>stem.FCAI_IE</i> stem variable.
	-17 (FCAI_RESULT_UNUSABLESTEM)	The stem variable is not usable.
	-19 (FCAI_RESULT_REXXERROR)	An error occurred when storing, fetching, or dropping a REXX variable.
≥0		The call completed successfully.
	0 (FCAI_RESULT_OK)	No additional information is provided.

Examples:

```
rc = ftpapi('fcai.', 'set_trace', 'ON')
```

SET_REQUEST_TIMER request

Format:

►►—ftpapi—(—*stem*—,—'set_request_timer' —,—*reqtimer*—)————►►

Purpose: Sets the length of time that the FTP Client API waits for an INIT, SCMD, or TERM request to complete.

Parameters:

stem

The name of a stem that represents the FTP client environment. The stem must have been passed on a prior successful INIT function call.

'set_request_timer'

Requests that the request timer be set for subsequent INIT, SCMD, or TERM requests. The string literal is not case sensitive.

reqtimer

The number of minutes to wait for the request to complete. Valid values are in the range 0 – 256. The value corresponds to the FCAI_ReqTimer field used by the C and callable FTP Client APIs.

Results:

Table 27. FTP client SET_REQUEST_TIMER request return codes

Value	Return codes	Explanation
<0		The call did not complete successfully. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the <i>stem</i> .FCAI_IE stem variable.
	-17 (FCAI_RESULT_UNUSABLESTEM)	The stem variable is not usable.
	-19 (FCAI_RESULT_REXXERROR)	An error occurred when storing, fetching, or dropping a REXX variable.
≥0		The call completed successfully.
	0 (FCAI_RESULT_OK)	No additional information is provided.

Examples:

```
rc = ftpapi('fcai.', 'set_request_timer', 1)
```

GET_FCAI_MAP request

Format:

►►ftppapi(—*stem*—,—'get_fcai_map' —,—*fcaimap*—)————►►

Purpose: Returns the contents of the full FCAI_Map structure.

Parameters:

stem

The name of a stem that represents the FTP client environment. The stem must have been passed on a prior successful INIT function call.

'get_fcai_map'

Requests the return of the full FCAI_Map structure. The string literal is not case sensitive.

fcaimap

The name of the stem where the FCAI_Map structure should be placed. The map elements shown in Table 28 can be returned.

Table 28. FCAI_Map structure elements

Element	Description
fcaiMap.FCAI_EyeCatcher	Eyecatcher='FCAI'
fcaiMap.FCAI_Size	Size of the FCAI=256
fcaiMap.FCAI_Version	Version of the FCAI=1
fcaiMap.FCAI_PollWait	POLL wait timer in seconds
fcaiMap.FCAI_ReqTimer	Request timer in minutes or 0 for none
fcaiMap.FCAI_TraceIt	Trace indicator for this request
fcaiMap.FCAI_TraceID	ID used in a trace record
fcaiMap.FCAI_TraceCAPI	TRACECAPI value on FTP.DATA statement
fcaiMap.FCAI_TraceStatus	Status of the trace
fcaiMap.FCAI_TraceSClass	SYSOUT class for trace file
fcaiMap.FCAI_TraceName	ddname of the trace file
fcaiMap.FCAI-Token	Interface token
fcaiMap.FCAI_RequestID	Last request processed by the EZAFTPKS program (for example, 'SCMD')
fcaiMap.FCAI_Result	Request result
fcaiMap.FCAI_IE	Interface error
fcaiMap.FCAI_CEC	Client error code (see FTP return codes in <i>z/OS Communications Server: IP User's Guide and Commands</i>)
fcaiMap.FCAI_ReplyCode	Server reply code or 0 if no reply (see FTPD reply codes in <i>z/OS Communications Server: IP and SNA Codes</i>)
fcaiMap.FCAI_SCMD	Subcommand code (see FTP subcommand codes in <i>z/OS Communications Server: IP User's Guide and Commands</i>)
fcaiMap.FCAI_ReturnCode	Return code

Table 28. FCAI_Map structure elements (continued)

Element	Description
fcaiMap.FCAI_ReasonCode	Reason code
fcaiMap.FCAI_NumberLines	Number of output lines returned by the request
fcaiMap.FCAI_LongestLine	Size of the longest line
fcaiMap.FCAI_SizeAll	Size of all output lines
fcaiMap.FCAI_SizeMessages	Size of all message lines
fcaiMap.FCAI_SizeReplies	Size of all reply lines
fcaiMap.FCAI_SizeList	Size of all list lines
fcaiMap.FCAI_SizeTrace	Size of all trace lines
fcaiMap.FCAI_PID	Process ID of FTP client

Results:

Table 29. FTP client GET_FCAI_MAP request return codes

Value	Return codes	Explanation
<0		The call did not complete successfully. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the <i>stem.FCAI_IE</i> stem variable.
	-17 (FCAI_RESULT_UNUSABLESTEM)	The stem variable is not usable.
	-19 (FCAI_RESULT_REXXERROR)	An error occurred when storing, fetching, or dropping a REXX variable.
≥0		The call completed successfully.
	0 (FCAI_RESULT_OK)	No additional information is provided.

Examples:

```
rc=ftppapi('fcai.', 'get_fcai_map', 'fcaiMap.')
```

TERM request

Format:

►►—ftpapi—(—*stem*—,—'term' —)—►►

Purpose: The user program issues the TERM request to terminate this instance of interface use. A TERM request is accepted by the interface at any time.

Rule: Each successful invocation of INIT should be terminated with a corresponding invocation of TERM.

Parameters:

stem

The name of a stem that represents the FTP client environment. The stem must have been passed on a prior successful INIT function call.

'term'

Requests the termination of this instance of the FTP interface. The string literal is not case sensitive.

Results:

Table 30. FTP client TERM request return codes

Value	Return codes	Explanation
<0		The call did not complete successfully. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the <i>stem</i> .FCAI_IE stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the <i>stem</i> .FCAI_CEC field stem variable.
	-17 (FCAI_RESULT_UNUSABLESTEM)	The stem variable is not usable.
	-18 (FCAI_TASK_TASKMISMATCH)	The task is not the same as the INIT task.
	-19 (FCAI_RESULT_REXXERROR)	An error occurred when storing, fetching, or dropping a REXX variable.
	-32 (FCAI_TASK_CLIPROCESSKILL)	A TERM request issued BPX1KIL to end the client process.
≥0		The call completed successfully.
	0 (FCAI_RESULT_OK)	No additional information is provided.

Examples:

```
rc = ftpapi('fcai.', 'term')
```

REXX FTP Client API trace return codes

Table 31 describes the return codes that can be stored in the FCAI_TRACE_DATASET_RETCODE variable.

Table 31. REXX FTP Client API trace return codes

Code type	Code	Description
Success	0	No errors detected with the REXX FTP Client API trace
Data set allocation error	65537 (X'10001')	Insufficient storage
Data set allocation error	65538 (X'10002')	DD name not valid
Data set allocation error	65539 (X'10003')	Data set in use
Data set allocation error	65540 (X'10004')	Bad DSorg value
Data set allocation error	65542 (X'10006')	No DD name returned by DYNALLOC
Data set allocation error	65543 (X'10007')	PDS is not supported
Data set allocation error	131072–196607 (X'20000'–X'2FFFF')	DYNALLOC failure. The lower 16 bits contain the SVC99ERROR value; see the <i>z/OS MVS Diagnosis: Reference</i>
Open data set error	262145 (X'40001')	Data set characteristics not valid
Open data set error	262146 (X'40002')	OPEN failure
Data set deallocation error	524288 (X'80000')	DYNALLOC failure
Write failure	786433 (X'C0001')	SYNAD error
Write failure	786434 (X'C0002')	End of data set

Output register information for the FTP Client API

For the Cobol, C, assembler, and PL/I programming languages, when control returns to the caller, the general purpose registers (GPRs) contain the following:

Register	Contents
0-1	Used as work registers by the system
2-13	Unchanged
14	Used as a work register by the system
15	Used to contain a return code

Restriction: The REXX programming language does not use output registers to return information to the REXX program. For information about REXX program output, see “FTP Client API REXX function” on page 433.

When control returns to the caller, the access registers (ARs) contain:

Register	Contents
0-1	Used as work registers by the system
2-15	Unchanged

If a caller depends on the contents of a register that is to be used by the system, the caller must save the contents of that register before invoking the interface and restore the contents after the system returns control.

Other output that is returned to the application

After a request drives processing in the client, the client sends output to the interface. The interface retrieves this output automatically for INIT and SCMD requests issued in wait mode (or an SCMD QUIT request generated by a TERM request). An SCMD request that is issued in no-wait mode retrieves no output and an SCMD request that fails to complete before FCAI_ReqTimer expires might or might not retrieve partial output. The caller must retrieve the results for an SCMD request that returns a value of FCAI_Status_InProgress with the POLL request. See “Messages and replies” on page 458 for a discussion of the final message, EZA2121I.

When the output is retrieved, the interface stores it in the *interface buffer*. This output can be retrieved by the user program with a GETL request. The output is composed of four types of lines:

- Messages generated by the FTP client
- Replies sent from the FTP server
- List data from the FTP server for a DIR or LS subcommand
- Trace data for the FTP client

See “Example of the GETL call instruction” on page 424, which shows message, reply, list, and trace output lines. See “Messages and replies” on page 458 for additional information about the difference between messages and replies.

The FCAI_PID value is updated after a successful INIT request and is cleared after a TERM request. The following statistics in the FCAI are updated for each complete INIT or SCMD request, and for each POLL request until the prior SCMD request completes. A GETL request does not change the statistics because it retrieves only the output that has already been copied to the interface buffer from the client. A TERM request terminates the interface and frees the interface buffer; it returns no output.

Field name	Description
FCAI_NumberLines	Total number of lines of output.
FCAI_LongestLine	Size of the longest line of output. Use this value to acquire or define a space into which a line of output can be copied.
FCAI_SizeAll	Total size of all lines of output.
FCAI_SizeMessages	Size of all lines of messages from the client.
FCAI_SizeReplies	Size of all lines of replies from the server.
FCAI_SizeList	Size of all lines of data from a DIR or LS subcommand.
FCAI_SizeTrace	Size of all lines of trace data.

The output data is available for retrieval with a GETL request until the next SCMD or TERM request is processed. The next SCMD request clears all status fields for the new request and reuses the space allocated to the interface buffer. You should use a GETL request with the COPY operation if you want to preserve the output.

Prompts from the client

Some z/OS FTP client subcommands cause the client to prompt the user for more input. For example, processing the USER subcommand causes the client to prompt for a password after sending a USER command to the FTP server. Prompt behavior changes when the z/OS FTP client is invoked from the FTP Client API.

As shown in “Parameter values that are returned to the application” on page 418, some prompts affect status codes returned by the interface. See “Interpreting results from an interface request” on page 459 for more information about status codes. The following sections describe the current prompt situations for the FTP client and how they are handled by the FTP Client API.

Prompts not used by the FTP Client API

- Prompt for IP address if not supplied as a parameter on the INIT request
The FTP client prompts immediately if an IP address or host name was not supplied at client start. The FTP Client API does not pass this prompt to the user program. The user program should use an SCMD request to send an OPEN subcommand as soon as it wants a session with the FTP server.
- Prompt for a user ID after an OPEN subcommand
The FTP client prompts for a user ID for login after the session is set up with the server. The FTP Client API does not pass this prompt to the user program. The user program should use an SCMD request to send a USER subcommand as soon as it wants to log in with the FTP server. The user program can provide the password as well as the user ID as parameters with the USER subcommand.
- Prompt for subcommand after a PROXY subcommand
The FTP client prompts for a subcommand if the PROXY subcommand is entered without a subcommand parameter. The FTP Client API does not support PROXY without a subcommand. If the client is invoked by the FTP Client API and receives PROXY without a subcommand, the request fails with FCAI-Result = FCAI-Result-CEC and FCAI-CEC = FCAI-CEC-PROXY-ERR.
Requirement: Every PROXY subcommand that is issued to the FTP Client API must be in the following format: *PROXY subcommand <optional parameters>*. This requirement includes the PASS subcommand, which must be entered as *PROXY PASS password* (if the password was not included on the prior USER subcommand, as in *PROXY USER userid password*). See the PROXY subcommand information in the *z/OS Communications Server: IP User's Guide and Commands* for an explanation of the PROXY subcommand and its parameters.
- Prompt for confirmation for MGET, MPUT, and MDELETE subcommands
The FTP client prompts for confirmation for these subcommands if the PROMPT subcommand has toggled to prompting. The FTP Client API does not pass this prompt to the user program. The subcommand is executed in no-prompt mode.

Prompts returned in FCAI-Status

- Prompt for password after a USER subcommand
The FTP client prompts for a password to complete a login if the password was not passed with the USER subcommand. The FTP Client API passes this prompt to the user program by using the status FCAI-Status-PromptPass. The user program should use an SCMD request to send a PASS subcommand as the next subcommand. If any subcommand other than PASS is sent, the request fails with FCAI-Result = FCAI-Result-IE and FCAI-IE = FCAI-PassPromptErr.
- Prompt for accounting information after a PASS or a CD (CWD) subcommand

Some FTP servers prompt the FTP client for accounting information after a PASS or CWD command is processed. The FTP Client API passes this prompt to the user program by using the status FCAI-Status-PromptAcct. The user program should use an SCMD request to send an ACCT subcommand as the next subcommand. If any subcommand other than ACCT is sent, the request fails with FCAI-Result = FCAI-Result-IE and FCAI-IE = FCAI-AcctPromptErr.

Tip: When a PASS or ACCT subcommand is expected, the interface refuses any other SCMD request until the prompt is satisfied. The user program can issue a GETL or TERM request without satisfying the prompt. A TERM request generates a QUIT subcommand that is accepted and stops the client process.

Command prompt

Each subcommand that is sent to the FTP client ends with an output message that is a prompt for the next subcommand. This message is the last of the messages that are returned as output; it can be retrieved with the GETL request for the last message. See “Messages and replies” for an example of EZA2121I, which is the command prompt used by the FTP Client API.

Messages and replies

Messages are information statements that are provided by the FTP client. Replies are the responses to commands that are returned from the FTP server to the client. Replies are described in FTPD reply codes in *z/OS Communications Server: IP and SNA Codes*.

Messages are composed of a message ID followed by message text. FTP client messages are described in *z/OS Communications Server: IP Messages Volume 1 (EZA)* and *z/OS Communications Server: IP Messages Volume 4 (EZZ, SNM)*. You can use the message ID to look up the message in these volumes. However, the message IDs are not written to output unless the client is executing in verbose mode.

Following is an example that uses the **verbose** subcommand in an interactive environment. The FTP Client API processes the subcommand on an SCMD request as previously described. Note that the **verbose** subcommand acts as a toggle.

```
verbose
EZA2859I Message IDs are displayed when running in z/OS UNIX
EZA1460I Command:
verbose
Message IDs are not displayed when running in z/OS UNIX
Command:
```

EZA2859I and EZA1460I (the first token that is displayed when executing in verbose mode) are message IDs. The phrases that follow the message ID are message text. Notice that after the **verbose** command is executed the second time to toggle verbose off, the message texts appear with no message ID. See the Verbose subcommand information in the *z/OS Communications Server: IP User's Guide and Commands* for more information about entering and exiting verbose mode.

The FTP Client API does not use the EZA1460I Command: message. Instead, it uses a new message with additional status information about the subcommand that completed. The syntax is:

```
EZA2121I Command (ee-ss-cccc-rrr):
```

where:

- *ee* is the 2-digit decimal client error code for the subcommand (00 if no error)
- *ss* is the 2-digit decimal subcommand code (the field is blank on INIT when an implicit OPEN was not performed)
- *cccc* is the final 4-character FTP command sent to the server by the subcommand (all blanks if no command was sent)
- *rrr* is the last 3-character server reply to the final FTP command (all blanks if no reply was received)

Replies are composed of a 3-digit numeric reply code followed by text. The significance of the reply prefix is described in RFC 959, *File Transfer Protocol* (see Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs); replies used by the z/OS FTP server are described in FTPD reply codes in *z/OS Communications Server: IP and SNA Codes*. Unlike message numbers, reply codes are never suppressed. Your program can usually disregard the text and inspect the reply code to determine whether the server processed the FTP command successfully. The one exception is when an FTP client sends the SITE command to a z/OS FTP server. The z/OS FTP server SITE reply is always 200 (implying success), even when one or more errors occurred when processing the SITE command, as in the following example:

```
site bogus
>>> SITE bogus
200-Unrecognized parameter 'bogus' on SITE command.
200 SITE command was accepted
Command(00-34-SITE-200):
```

The following sample output from the CD subcommand shows messages EZA1701I and EZA2121I from the FTP client and the reply 250 from the FTP server. Note that the CD subcommand causes the client to send the CWD command to the server.

```
CD /u/user33/
EZA1701I >>> CWD /u/user33/
250 HFS directory /u/user33/ is the current working directory
EZA2121I Command(00-07-CWD -250):
verbose
Message IDs are not displayed when running in z/OS UNIX
Command(00-71- - ):
CD /u/user33/
>>> CWD /u/user33/
250 HFS directory /u/user33/ is the current working directory
Command(00-07-CWD -250):
```

Interpreting results from an interface request

The results of a request to the FTP Client Application Interface are reported to the user program in the FCAI control block. See “FTP Client Application Interface (FCAI) control block” on page 399.

FCAI request completion values

The following list describes how to interpret the contents of FCAI_Result and how to use the secondary results fields.

If FCAI_Result contains . . .	Then . . .	Explanation
FCAI_Result_OK	no further action is required.	The request is complete with no additional status or errors to report.

If FCAI_Result contains . . .	Then . . .	Explanation
FCAI_Result_Status	check the FCAI_Status field for additional status information.	The interface uses the status field to report prompts from the client, to report an indication that the last request is still in-progress, and to report an error in the interface trace function.
FCAI_Result_IE	check the FCAI_IE field for error information.	The request returned an interface error, which indicates that the interface was unable to process the request for some reason. That reason might be a parameter on the call that was not valid, a failure in a service routine (such as GETMAIN), or termination of the client process.
FCAI_Result_CEC	check the FCAI_CEC field for the error code that was returned by the client. You can optionally issue GETL to retrieve diagnostic information (see "GETL" on page 422).	The request completed with an FTP Client Error Code. For information about Client Error Codes and diagnosing errors in the client and server, see FTP return codes in the <i>z/OS Communications Server: IP User's Guide and Commands</i> and Diagnosing FTP problems in the <i>z/OS Communications Server: IP Diagnosis Guide</i> .
Other values returned in FCAI_Result are listed along with their descriptions in "FTP Client Application Interface (FCAI) control block" on page 399.		

Tips:

1. The value in FCAI_Result is returned to the caller in the return code register. Additionally, callers in COBOL, C, and PL/I can access the return code value in the FCAI_ReturnCode field and the reason code value in the FCAI_ReasonCode field.
2. The result code FCAI_Result_UnusableFCAI is returned only in the return code register.
3. Always verify that the return code register is set to 0 before inspecting the FCAI.

For all requests that communicate with the FTP server, the field FCAI_ReplyCode reports the last reply that was received from the final server command sent to the FTP server by the client for the request. The FCAI_ReplyCode field is a binary field with a length of 2. For example, a 250 reply from the server is recorded in the field as X'00FA'. A value of all zeros indicates that the client did not communicate with the FTP server. The following conditions apply to the FCAI_ReplyCode field.

- GETL and TERM requests do not populate the FCAI_ReplyCode field.
- An INIT request populates this field only if an implicit OPEN was performed.
- SCMD requests for locally processed subcommands such as LOCSITE and LPWD do not communicate with the FTP server. Those subcommands, as well as subcommands that fail before sending a command to the server, do not populate this field.
- A POLL request might or might not populate this field, depending on the prior subcommand that was in-progress.

Requests that initiate a subcommand in the z/OS FTP client return a value in the FCAI_SCMD field. This field is set by all SCMD requests and by the INIT request when start parameters that were passed on the request generate an implicit OPEN subcommand.

FCAI_ReturnCode and FCAI_ReasonCode values are set when certain services fail. See “FTP Client Application Interface (FCAI) control block” on page 399 as well as “Programming notes for the FTP Client API” for more information.

Considerations when evaluating request completion values

- The request completion values FCAI_CEC and FCAI_ReplyCode are available when the request completes. An FCAI_SCMD value is available when the subcommand completes but might also be set by a POLL request prior to completion of the subcommand. See “POLL” on page 420 and the MODE parameter definition in “SCMD” on page 417 for more information about the completion of subcommands.
- After a valid PROXY subcommand, the FCAI_SCMD value that is returned and the value in message EZA2121I reflect the client subcommand that was passed as a parameter with PROXY. The server command (if any) that was sent to the PROXY server appears in message EZA2121I. For example, PROXY DIR returns the subcommand code for DIR in FCAI_SCMD. Message EZA2121I displays the DIR subcommand code and the LIST server command.
- If a failure occurs in the interface trace function while processing the request, FCAI_Status_TraceFailed is added to any other status value that is returned in FCAI_Status. FCAI_Result contains FCAI_Result_Status unless there was a concurrent error on the request. See “FCAI_Status_TraceFailed and FCAI_TraceStatus — reporting failures in the interface trace function” on page 462 for more information.
- When a failure occurs in the interface with the client, all subsequent requests except GETL and TERM return FCAI_IE_CliProcessBroken.
- When the client processes a QUIT subcommand, all subsequent requests except GETL and TERM return FCAI_IE_CliProcessStopped.
- Prior to a successful INIT, any request other than INIT returns FCAI_IE_NoTokenAddr.
- FCAI_Result_CliProcessKill is an informational code returned only by a TERM request. See “TERM” on page 428.

Programming notes for the FTP Client API

The following sections contain additional information about the following:

- FCAI_Status_TraceFailed
- FCAI_TraceStatus
- FCAI_IE_LengthInvalid
- FCAI_ReqTimer
- FCAI_PollWait
- FCAI_IE_InternalErr

There is also a discussion of exceptional conditions in the z/OS FTP client process.

FCAI_Status_TraceFailed and FCAI_TraceStatus — reporting failures in the interface trace function

- A failure in the interface trace function is not treated as a severe error; if possible, processing continues for the request.
- The interface trace function is disabled when it encounters an error and cannot be restarted for this instance of use of the interface.
- The additional information described in Table 13 on page 402 is available only immediately after the request that returns FCAI_Status_TraceFailed. A concurrent error on the request can overwrite the additional information fields.
- When a request returns FCAI_Status_TraceFailed and sets a reason of FCAI_TraceStatus_AllocErr (error in dynamic allocation), the additional information can be interpreted as follows:
 - The return code, upon completion of the call to the interface, does not contain the return code from dynamic allocation. The return code from dynamic allocation is not reported except as noted in the following item.
 - The FCAI_ReturnCode field contains S99ERROR or the number 8.
If the FCAI_ReturnCode field contains the value 8, the dynamic allocation return code was 8. This return code means an installation validation routine failed the request and S99ERROR is not available.
 - The FCAI_ReasonCode field contains S99ERSN for DFSMS™ failures. For all other cases, it contains S99INFO.
 - S99ERSN is reported in FCAI_ReasonCode for all S99ERROR values that begin with X'97'.
 - S99INFO can be returned when no error was reported on the allocation. In those cases, the FCAI_ReasonCode field contains the S99INFO value even though no other results fields are set.
 - S99INFO can be (and often is) 0 when an error is reported. Only certain conditions set S99INFO.
- FCAI_Result contains FCAI_Result_Status when the trace encounters a failure unless a concurrent error sets FCAI_Result to a higher value.
- The request that disables the interface trace function adds FCAI_Status_TraceFailed to any other value that is returned in FCAI_Status. When FCAI_Status_TraceFailed is subtracted from FCAI_Status, the result is one of the FCAI_Status values or 0. The result that remains in FCAI_Status is described in Table 15 on page 403.
- FCAI_TraceStatus always reflects the current status of the interface trace function and, if applicable, the reason it was disabled. FCAI_TraceStatus field values are described in Table 13 on page 402.

FCAI_IE_LengthInvalid — improper lengths passed to the interface

- On an INIT request, a length value less than 0 was passed for the optional start parameters. The value 0 is accepted and bypasses sending start parameters.
- On an INIT request, the list of environment variables contained a negative count word or length.
- On a GETL request, a length value less than or equal to 0 was passed in the buffer vector.

Tip: Values greater than zero but insufficient to hold the first line of selected output return FCAI_IE_BufferTooSmall.

- On an SCMD request, the subcommand string length value is less than or equal to 0. A subcommand string is required on the request.

FCAI_ReqTimer — controlling requests that retrieve results from the spawned z/OS FTP client process

- FCAI_ReqTimer is used to limit the time the interface attempts to retrieve results from INIT and SCMD requests issued in wait mode, and TERM that automatically generates SCMD QUIT. (POLL reads data from the client but accepts only what has been written and returns immediately; GETL does not read from the client.)
- FCAI_ReqTimer is an unsigned, 1-byte hexadecimal value in the range 1-255 that indicates the number of minutes to wait for the request to complete. A value of 0 means not to use a timer on the request (wait until completion).
 - FCAI_ReqTimer is approximate; the actual wait time can be noticeably longer but should not be shorter.
 - FCAI_ReqTimer is not related to performance. Using a low value does not improve response time on the request. It is intended only to prevent the interface from polling a non-responsive client process indefinitely.
 - FCAI_ReqTimer is ignored by POLL, GETL, SCMD issued in no-wait mode, and TERM that does not generate a QUIT subcommand.
- If the interface detects that the client process is no longer there, it returns FCAI_IE_CliProcessBroken on any request except TERM. When FCAI_ReqTimer expires, the client process is still there but was unable to return all the results in the time limit that was specified. The application must determine how to respond when FCAI_ReqTimer expires.
- INIT enforces a minimum request timer setting of five minutes. The user can specify a value longer than five minutes or 0 (wait until completion) for the INIT request. Other FCAI_ReqTimer settings are ignored by INIT.
Timer expiration during INIT returns FCAI_Result_IE in FCAI_Result and FCAI_IE_ReqTimerExpired in FCAI_IE. INIT is the only request that returns this interface error. The error indicates that the interface failed to initialize.
- TERM enforces a minimum request timer setting of two minutes on a generated QUIT subcommand. The user can specify a value longer than two minutes or 0 (wait until completion) for the TERM request. Other FCAI_ReqTimer settings are ignored by TERM. See “TERM” on page 428.
Timer expiration during a QUIT generated by TERM causes TERM to return FCAI_Result_CliProcessKill.
- SCMD issued in wait mode accepts any FCAI_ReqTimer setting.
Timer expiration on an SCMD issued in wait mode returns FCAI_Result_Status in FCAI_Result and FCAI_Status_InProgress in FCAI_Status. At that point the behavior of the interface is the same as if the SCMD had been issued in no-wait mode. See “SCMD” on page 417 and “POLL” on page 420 for more information about no-wait processing.

FCAI_PollWait — specifying a wait time before POLL

To assist the application in managing POLL requests, the FTP Client API automatically pauses before it reads from the pipe to the client process. This wait suspends the interface and the application and it protects the application from sending POLL requests at a rate that degrades performance.

FCAI_PollWait is used as follows:

- A value of 0 (the default setting) in FCAI_PollWait instructs a POLL request to wait 1 second prior to reading the pipe from the client.
- A value greater than 0 enables a progressive wait timer and sets the maximum number of seconds that will be used. The field accepts a value in the range 0–255 seconds (4.25 minutes), but a value of 32 seconds or less is recommended for most subcommands.
- The current timer value is stored internally in the interface and persists between POLL requests that are issued for the same prior subcommand. If a progressive timer has not been enabled, the current timer value is always 1 second. A progressive wait timer begins at 1 second and doubles after each POLL until the maximum setting is reached or exceeded. If exceeded, the timer value is set to the value that was supplied by the user.
- The application can enable or disable the progressive timer and increase or decrease the maximum value to use on any POLL, regardless of the current timer value. The interface checks prior to each POLL to ensure that the timer does not exceed the specified maximum (a value of 0 sets the maximum to 1 second).
- The current timer value resets to 1 second after a POLL receives data from the pipe. This enables the user to retrieve all available output efficiently and removes much of the application's burden of managing a progressive timer. If a progressive timer has been requested, it begins to progress again as no data is returned (unless all output has been retrieved).
- The application is responsible for sending each POLL request. POLL requests are not generated automatically by the interface. The PollWait interval is in addition to any wait done in the application program.

FCAI_IE_InternalErr — unanticipated exceptional conditions in the interface

Examples of conditions that return FCAI_IE_InternalErr are:

- A linked interface buffer part was expected but none was found.
- A length field that was not valid was detected in the interface buffer.
- A computation during buffer navigation unexpectedly resulted in a negative value.

Conditions of this type indicate a logic error, storage overlay, or some other unrecoverable error in the interface. The interface accepts only a TERM request after the error.

To diagnose the error, dump all storage in the application address space as soon as the error code is returned. Contact the IBM support center for assistance if needed.

Guideline: If the application remains active after the storage dump, issue TERM to kill the client process and then terminate the application.

Exceptional conditions in the z/OS FTP client

- The following failures in the FTP client are considered fatal by the FTP Client API.
 - If the FTP client process experiences an abnormal termination, the next request that attempts to read from the client detects the condition and returns FCAI_IE_CliProcessBroken.

- If an error is encountered while spawning the FTP client, establishing pipes to the client, or communicating with the client, the interface error that describes the condition is returned to the application.

If you need assistance in diagnosing these failures, contact the IBM support center.

- If the FTP client stops responding to the interface during INIT or SCMD processing (including a QUIT subcommand generated by TERM) and FCAI_ReqTimer is 0 (wait until completion), the application waits indefinitely for the client. Dump all the storage in the address spaces for the interface and the spawned FTP client process, cancel the application, and kill the FTP client process if necessary. Contact the IBM support center if you need assistance.
- When a request returns FCAI_Status_InProgress because it exceeded the FCAI_ReqTimer value, the application must ascertain whether this is an error condition and how to proceed.
- If a signal is raised by a service invoked by the FTP client process, it is generally handled within the FTP client and reported to the application by a Client Error Code when appropriate.
- If a signal is raised on behalf of the client (that is, when the child process ends), neither the client nor the FTP Client API blocks or handles the signal.
- If the interface must issue BPX1KIL to kill the FTP client process during TERM, FCAI_Result_CliProcessKill is returned to inform the application. No further action is required.

Using the FTP Client API trace

The FTP Client API trace is used to debug problems in the interface or record activity and data that are returned to the interface that might not otherwise be available to the application (see “TERM” on page 428). The interface trace cannot be used to debug errors in the trace itself or any error that prevents the interface from accessing the trace data set. The trace writes records for interface events, which include requests to the interface, the results of interface requests, and output from the client. Client output includes client messages, server replies, list data, and DEBUG and DUMP trace data that is received from the client.

The FTP Client API trace is controlled by a statement that is coded in the FTP.DATA file for the FTP client. The statement is named TRACECAPI and is described in *z/OS Communications Server: IP Configuration Reference*.

Tip: The FTP Client API trace does not include trace records for the REXX FTP Client API function package. See “FTP Client API REXX function” on page 433 for information about trace records for the REXX FTP Client API function package.

The following table lists the settings accepted for TRACECAPI.

If the statement is coded with a value of	Then . . .	Notes
ALL	all interface events are traced.	n/a
NONE	no events are traced.	n/a

If the statement is coded with a value of	Then . . .	Notes
CONDITIONAL	events are traced only when requested by the user program. This is the default.	The user program can request that events be traced by setting the FCAI_TraceIt field. FCAI_TraceIt_Yes indicates that events are traced by the interface; FCAI_TraceIt_No indicates that events are not traced.

The settings in the following two fields in the FCAI are applicable when the interface trace is initialized (the interface trace is initialized by the first request that requests tracing):

- FCAI_TraceSClass

The interface trace is written to a spool file of the SYSOUT class supplied in FCAI_TraceSClass. Valid values are in the range A—Z and 0—9. The default value is A. The first request that is traced allocates and opens the spool file. When the trace file is opened, its ddname is placed in FCAI_TraceName. Once the file is opened, it stays open until a TERM request is processed.

- FCAI_TraceID

The user program can request that an ID be placed at the start of each trace record. This is done by putting a 3-byte character ID string in the field FCAI_TraceID. This value is written as the first three characters of each trace record. The trace ID is placed on each record to uniquely identify records from the same process. Trace records from different processes are written to different spool files. The ID ensures that records retain their identity when aggregated. The following example shows the trace records if FCAI_TraceID=TRC:

```
TRC INIT>-a never
TRC INIT<00000000 00000000 00000000 00000000
TRC SCMD>open 9.42.105.93 6321|W
```

At the top of the trace output and every 64 lines thereafter, the interface writes a header record to provide information about the trace records being written. The header record contains the FCAI_TraceID, the updated date and time, the decimal value of process ID (*pid*) of the spawned z/OS client, and the decimal value of the TCB address for the user program's task. The following is a sample header record:

```
ID-PAW Date-02/26/2004 Time-21:16:31 Process ID-0000000067108873 TCB Address-000000008256584
```

The following example uses a sample trace for a very simple session with the FTP Client API to show the format of the trace. The lines are numbered for the discussion of the trace; that is, the numbers do not appear in an actual trace. This trace shows requests and request results as well as client messages, server replies, and debug traces entries. The FCAI_TraceID value is 0, so each trace record is preceded by three blank characters (not reflected in this example). Also, the trace header records are not shown in the example.

```
1 PAW INIT>-a never |_CEE_DMPTARG=/etc
2 INIT<00000000 00000000 00000000 00000000
3 SCMD>open 9.42.105.93 6321|W
4 Connecting to: 9.42.105.93 port: 6321.
5 220-FTPDJG1 IBM FTP CS V1R6 at MVS164, 14:44:15 on 2007.
6 220 Connection will not timeout.
7 Command(00-10- -220):
8 SCMD<00000000 0A000000 00000000 00000000
```



```

 9 SCMD>user user33
10 >>> USER user33
11 331 Send password please.
12 Command(00-19-USER-331):
13 SCMD<01020000 13000000 00000000 00000000
14 SCMD>pass *****
15 >>> PASS
16 230 USER33 is logged on. Working directory is "/u/user33".
17 Command(00-26-PASS-230):
18 SCMD<00000000 1A000000 00000000 00000000
19 SCMD>debug fsc
20 Active client traces - FSC(1)
21 Command(00-11- - ):
22 SCMD<00000000 0B000000 00000000 00000000
23 SCMD>get a abc111 (repl|W
24 CG0226 get: F=1 p=FSA ARTWT=00001
25 CG3531 rcvFile: entered
26 MR1278 set_filename: entered with pathname abc111
27 CG1359 hfs_rcvFile: entered
28 MF0750 seq_open_file: recfm is NONE
29 MF1068 seq_open_file: OSBN -> wb,recfm=*,NOSEEK
    for /u/user33/abc111
30 MF1216 seq_open_file: stream 166412C4 has maxreclen 0
31 >>> PORT 9,42,105,93,4,25
32 200 Port request OK.
33 >>> RETR a
34 125 Sending data set /u/user33/a
35 CU2009 write_smf_record: entered with type 16.
36 CU1474 write_smf_record_119: entered with type 16.
37 TI1120 write_stream: 0=0 HGPES=10000 BCTE=0000 RLB=0/0/0
38 MF0632 seq_close_file: file closed
39 250 Transfer completed successfully.
40 200 bytes transferred in 0.010 seconds. Transfer rate
    20.00 Kbytes/sec
41 CU2009 write_smf_record: entered with type 16.
42 CU1474 write_smf_record_119: entered with type 16.
43 CU2279 write_smf_record: length of smfrecord: 240
44 Command(00-16-RETR-250):
45 SCMD<00000000 10000000 00000000 00000000
46 SCMD>QUIT|W (Generated by TERM)
47 >>> QUIT
48 221 Quit command received. Goodbye.
49 SCMD<00000000 13000000 00000000 00000000
50 TERM>
51 TERM<00000000 00000000 00000000 00000000

```

- Line 1: This is the INIT request to the interface. The character > shows the direction of the flow. The character | is used to separate the the start options "-a never" from the environment variable "_CEE_DMPTARG=/etc" that was passed on the spawn. Each parameter passed to the FTP client in the start options is separated from what follows in the trace by a null. Nulls should not be inserted into the start options by the application program (an ending null is permissible). The nulls in the trace are a result of the parsing mechanism that the interface uses to build the argument list for the spawn of the client.
- Line 2: This is the result of the request from the interface. The character < shows the direction of flow. The four words of output are the "Request Completion Values" from the FCAI. The values are in hexadecimal and do not display as shown here. The 00 in byte 0 indicates successful initialization of the interface.
- Line 3: This is the first subcommand. The subcommand string is displayed. If a mode parameter is entered, it is displayed following the character |. The character | is used to separate parameters in all of the request records. In this example, the user program entered a W for wait mode.

Line 4: This is a client message.
 Line 5: This is the first line of the 220 server reply.
 Line 6: This is the last line of the 220 server reply.
 Line 7: This is the client message that indicates the end of the subcommand. The result is 00 (no errors) for the open (subcommand code decimal 10). The client subcommand caused a connect to the server but no server command actually flowed -- so the command field has four blanks. However, the last reply from the server was the 220 reply.
 Line 8: These are the completion results for the open subcommand. The results in byte 0 are 00 (OK). The x'0A' in the second word is the open subcommand code in hexadecimal.
 Line 9: Another subcommand. This time the user program did not pass a mode parameter (default mode is wait).
 Line 10: This line shows the client message indicating a command to the server.
 Line 11: This line shows the server reply which requests a password.
 Line 12: This line shows as result of 00 (no errors) for the user subcommand (code 19). The last server command was USER and the last reply was 331.
 Line 13: This line contains the completion results for the user subcommand. The x'01' in byte 0 indicates that there is additional status. The x'02' in byte 1 indicates that the user program is prompted for a pass subcommand.
 Line 14: This is the pass subcommand. ***** is displayed to keep the actual password out of the trace.
 Line 15: This line shows the client message indicating a command to the server.
 Line 16: This line shows the server reply.
 Line 17: This line shows a result of 00 (OK) for the pass subcommand (code 26). The last server command was PASS and the last reply was 230 -- the user program is logged in.
 Line 18: This line shows the completion results for the pass subcommand.
 Line 19: This line shows a subcommand to activate one of the client traces.
 Line 24: This line is a client trace entry.
 Line 45: This line shows a result of 00 (no errors) for the get subcommand (code 16). The last server command was RETR and the last reply was 250 -- the file transfer completed successfully.
 Line 46: This line shows a quit subcommand that was generated by a request to terminate the interface. The user program failed to issue a QUIT to stop the client so TERM automatically generated a QUIT on behalf of the application.
 Line 50: This line shows a request to end the interface to the Client API.
 Line 51: The interface has ended.

The following example shows a portion of a trace after login and before the interface ends; this example shows some of the errors that can be reported by the trace.

```

1  OOPS>
2  OOPS<02000200 00000000 00000000 00000000
3  SCMD>
4  SCMD<02000300 00000000 00000000 00000000
5  GETL>D
6  GETL<02004000 00000000 00000000 00000000
7  GETL>FIND|J
8  GETL<02004100 00000000 00000000 00000000
9  POLL>
10 POLL<02003000 00000000 00000000 00000000
11 SCMD>get ! abc111 (rep|
12 >>> PORT 9,42,105,93,4,17
13 200 Port request OK.
14 >>> RETR !
15 501 Invalid data set name "!". Use MVS Dsname conventions.
16 Command(02-16-RETR-501):
17 SCMD<03000002 10000000 00000000 00000000
  
```



```

18 SCMD>get a 'user33.aaaaaaaaaaaaaaaa'|
19 Invalid local file identifier
20 Command(18-16- - ):
21 SCMD<03000012 10000000 00000000 00000000

```

Line 1: This is a request type that is unknown to the interface.
Line 2: The results in byte 0 are 02 (interface error) with an explanation in byte 2 of 02 (unknown request).
Line 3: The SCMD request has no parameters.
Line 4: The results in byte 0 are 02 (interface error) with an explanation in byte 2 of 03 (parameter missing).
Line 5: The GETL request has an unknown operation value.
Line 6: The results in byte 0 are 02 (interface error) with an explanation in byte 2 of 40 (decimal 64) (unknown operation)
Line 7: The GETL request has an unknown type value for the Find operation.
Line 8: The results in byte 0 are 02 (interface error) with an explanation in byte 2 of 41 (decimal 65) (unknown type)
Line 9: This is a POLL request when a prior subcommand is not in progress.
Line 10: The results in byte 0 are 02 (interface error) with an explanation in byte 2 of 30 (decimal 48) (request not in progress)
Line 11: This SCMD request has a get with a bad remote file identifier. Commands are sent to the server (see lines 12 to 15).
Line 16: This line shows a result of 02 (server error) for the get subcommand (code 16). The last server command was RETR and the last reply was 501 -- an error reply.
Line 17: This line contains the completion results for the get subcommand. The result in byte 0 is 03 (FTP error). The client error code in byte 3 indicates that the FTP server detected the error.
Line 18: This SCMD request has a get with a bad local file identifier. No command is sent to the server.
Line 20: This line shows a result of 18 (file access error) for the get subcommand (code 16). Since no command was sent to the server, the last server command and last reply are blanks.
Line 21: This line contains the completion results for the get subcommand. The result in byte 0 is 03 (the FTP client returned an error). The client error code in byte 3 indicates that the FTP client could not access a local file.

FTP Client API sample programs

The following sample programs for the FTP Client API are available in the SEZAINST data set:

Program	Description
EZAFTPAW	Assembler language FTP Client API sample program
EZAFTPAX	COBOL language FTP Client API sample program
EZAFTPAY	PL/I language FTP Client API sample program
EZAFTPIR	REXX language FTP Client API sample program

The FTPCAPIC C language FTP Client API sample program is found in /usr/lpp/tcpip/samples/ftpcapic.c.

Chapter 14. Network management interfaces (NMIs)

z/OS Communications Server provides several interfaces that enable network monitor and management applications to obtain information about their network operations, for both TCP/IP and VTAM. These interfaces for z/OS Communications Server TCP/IP provide the following capabilities:

- Programmatically obtain copies of TCP/IP packet and data trace buffers, in real-time, as the traces are collected.
- Format the TCP/IP packet trace records that are collected.
- Obtain the following:
 - Activation and deactivation events for TCP connections in SMF format that are buffered
 - Event information for the FTP and TN3270 clients and servers in SMF format that are buffered
 - Detailed information and statistics for IP filtering and IPSec security associations on local TCP/IP stacks
 - Detailed information and statistics for IP filtering and IPSec security associations on remote network security services (NSS) clients when using the NSS server
- Control the following:
 - IP filters and IPSec security associations on local TCP/IP stacks
 - IP filters and IPSec security associations on remote NSS clients when using the NSS server. See the *z/OS Communications Server: IP Configuration Guide* for more information about network security services.
- Monitor the following using a callable API:
 - TCP connection and UDP endpoint activity
 - TCP/IP storage usage
 - TN3270E Telnet server connection performance
- Drop one or multiple TCP connections or UDP endpoints

The interfaces for z/OS Communications Server VTAM provide the following:

- The ability to collect Enterprise Extender (EE) summary and connection data
- The ability to collect HPR endpoint data
- Communication Storage Manager (CSM) storage statistics

Some of the information provided by these interfaces can be obtained from other types of documented interfaces provided by z/OS Communications Server such as SNMP, SMF, command display output, and VTAM exits. TCP/IP packet trace collection and formatting interfaces provide access to packet trace data that was not previously available through an authorized, real-time z/OS Communications Server interface. Some of the event information in SMF format is currently available through traditional SMF services, and can be collected using an SMF user exit to monitor SMF records.

The interfaces that are described in this topic provide an alternative for collecting some of the TCP/IP SMF records and are expected to perform efficiently. Most of the data that is provided by the network management interface for monitoring TCP/UDP endpoints and TCP/IP storage described in “TCP/IP callable NMI

(EZBNMIFR)” on page 583 can be collected from supported SNMP MIBs. Storage usage information is available through displays and the VTAM Performance Monitor Interface (PMI). When used correctly, the interfaces documented in this document provide well-defined and efficient APIs to be used for obtaining management information related to the IP and SNA (VTAM) components of z/OS Communications Server. They also allow for easy application migration to subsequent z/OS Communications Server releases. They are targeted for use by responsible network management applications.

The following describe the programming interfaces for these functions in detail, and provide the information required to develop network management applications that use them. These interfaces have the following characteristics:

- Use a client/server model or a called interface
- Require all network management clients to be run locally on the same z/OS image as the Communications Server
- Are provided for C/C++ and Assembler, except as otherwise indicated

In this topic, the term TCP/IP is used to represent the IP component of z/OS Communications Server and the term VTAM refers to the SNA component of z/OS Communications Server.

Local IPSec NMI

The z/OS Communications Server IKE daemon provides the IPSec network management interface (NMI). The IPSec NMI is an AF_UNIX socket interface through which network management applications can manage IP filtering and IPSec on local TCP/IP stacks. Use this interface for network management applications that expect to maintain agents on each individual z/OS system or use it in any environments where z/OS network security services (NSS) is not enabled. If your applications use a centralized management and monitoring approach, you should consider using the NSS management interface that is described in “Network security services NMI” on page 520.

This interface enables applications to obtain the following types of data regarding the local TCP/IP stacks and the IKE daemon:

- Information about which TCP/IP stacks are configured for integrated IPSec/VPN
- Summary statistics for IKE, IPSec, and IP filtering activity for a particular TCP/IP stack
- Detailed information about IP filters for a particular TCP/IP stack
- Detailed information about IPSec and IKE security associations (SAs) for a particular TCP/IP stack
- Port translation information for NAT traversal
- Information about which IP interfaces are active for a given TCP/IP stack
- Information about NSS clients that are active in the local IKE daemon

In addition, network management applications can perform the following functions to control IP filtering and IPSec over the same AF_UNIX socket:

- Activate and deactivate manual and dynamic tunnels
- Refresh dynamic tunnels
- Switch between default IP filters and policy-based IP filters

With the IPSec network management interface, a client network management application makes requests and performs management actions by sending

messages over an AF_UNIX stream socket connection to the IKE daemon. The requested data is returned to the application directly over the AF_UNIX connection.

Configuring the interface

The z/OS system administrator can restrict access to the IKE network management interface as follows:

- Access to the stack monitoring functions (those that request information only about specific stacks) within this interface is controlled by defining the RACF (or equivalent external security manager product) resource EZB.NETMGMT.sysname.tcpipname.IPSEC.DISPLAY in the SERVAUTH class (where the sysname value represents the MVS system name where the interface is being invoked, and the tcpipname value is the name of the TCP/IP stack).
- Access to the stack control functions (those that take some action on a specific stack) is controlled through the EZB.NETMGMT.sysname.tcpipname.IPSEC.CONTROL resource.
- Access to IKE daemon-level monitoring functions (those that request information at the daemon level) is controlled through the EZB.NETMGMT.sysname.sysname.IKED.DISPLAY resource.

For applications that use the interface, the MVS user ID should be permitted to the defined resource. If the resource is not defined, then only superusers or users permitted to the BPX.SUPERUSER resource in the FACILITY class are permitted to access the interface.

Additionally, permitted client applications must have permission to enter the /var/sock directory and to write to the /var/sock/ipsecmgmt socket.

Guideline: If you are developing a feature for a product to be used by other parties, include instructions in your documentation indicating that administrators must define and give appropriate permission to the given security resource to use that feature; if the resource is not defined, indicate that administrators must run your program as superuser.

Requirements:

- The IKED OMVS user ID must have write access to the /var/sock directory (or else have permission to create this directory).
- z/OS Communications Server IKE daemon and Policy Agent must be active on the system where data is being collected.

Connecting to the server

For an application to use this interface, it must connect to the AF_UNIX stream socket provided by the IKE daemon for this interface. The socket path name is /var/sock/ipsecmgmt. You can use the Language Environment C/C++ API or the UNIX System Services BPX Callable Assembler services to create AF_UNIX sockets and connect to this service.

When an application connects to the socket, the IKE daemon sends an initialization message to the client application. When the IKE daemon closes a client connection (reasons for doing so include severe errors in the format of data requests sent by the application to the IKE daemon, or IKE daemon termination), the IKE daemon attempts to send a termination message to the client before closing the connection. Both the initialization and termination messages conform to the general response

message structure used by the IKE daemon to send data to the application (see “IPSec NMI request/response format”).

The initialization message contains only a message header (see “Initialization and termination messages” on page 517). The version number reported in the message header indicates the maximum version of the interface supported by the IKE daemon. After the initialization message has been received by the client, the client can send requests for IPSec management data to the server.

The termination message also contains only a message header (see “Initialization and termination messages” on page 517). The message header contains a return code and a reason code that indicates the reason for terminating the connection.

IPSec NMI request/response format

This interface exchanges messages over an AF_UNIX socket using a request-response model. The client application builds and sends an NMI request over the socket. The request specifies the action or the type of information requested and might contain optional input parameters. The IKE daemon then provides a response message over the socket that contains the results of the request, including the requested data if this is a monitoring request. The client application must then read this response data from the socket. A severe formatting error in the client application’s NMI request might result in the IKE daemon sending a termination record and closing the connection.

The IPSec network management interface provides the formatted response data directly to the client application over the AF_UNIX connection.

Rule: All EBCDIC data is right-padded with blank characters and uses codepage IBM-1047 encoding.

Request and response data structures

The network management interface for monitoring IP filtering and IPSec provides data structures for C, C++, and Assembler programs to access the interface. The C and C++ header file is contained in the ezbnmsec.h file, located in the /usr/include directory. The assembler macro is located in the EZBNMSEA member of the SEZANMAC data set.

Message format

NMI request and response messages share a common format. An IPSec NMI message consists of a message header followed by zero or more records. The message header is defined by the NMsecMessageHeader structure.

Table 32. NMsecMessageHeader structure

Field	Offset	Length	Format	Description
NMsMIdent	0	4 bytes	EBCDIC	Message header identifier; set to NMsec_MSGIDENT (EBCDIC ‘NMsM’).
NMsMHdrLength	4	4 bytes	Binary	Length of the message header. See the NMsMMsgLength field for the length of entire message.
NMsMVersion	8	2 bytes	Binary	NMI version. Only version 2 is currently supported by this interface (NMsec_VERSION2).

Table 32. *NMsecMessageHeader* structure (continued)

Field	Offset	Length	Format	Description
NMsMType	10	2 bytes	Binary	Message type. For a request, this indicates the type of request being made. For a response, this indicates the type of response data, and is identical to the request type. See “Request messages” on page 479 for a description of the request types.
NMsMCorrelator	12	16 bytes	Binary	User-defined field for correlating NMI requests with responses. The interface echoes the correlator for a given request on the corresponding response.
NMsMRsvd1	28	4 bytes	Binary	Reserved; set to 0.
NMsMRc	32	4 bytes	Binary	Return code. The client must set this field to 0 in a request message. For a reply, this field is 0 for a successful reply, or a nonzero value for an error (see “Return and reason codes” on page 525).
NMsMRsn	36	4 bytes	Binary	Reason code. The client must set this field to 0 in a request message. For a reply, if the NMsMRc field indicates an error, this field might provide additional information about the error (see “Return and reason codes” on page 525).
NMsMMsgLength	40	4 bytes	Binary	Length of entire message, including the message header.
NMsMTime	44	4 bytes	Binary	Timestamp. The server ignores this field in a request message. For a reply, this value indicates the UNIX timestamp for the server. This might be correlated with timestamps in result fields.
NMsMRsvd2	48	20 bytes	Binary	Reserved; set to 0.
NMsMInRec	68	8 bytes	Binary	Input record descriptor. This field is set by the client application for request messages and describes which records, if any, are present on the request. This descriptor is described by the <i>NMsecInRecDesc</i> structure. See Table 33 on page 476 for details.
NMsMRsvd3	76	8 bytes	Binary	Reserved; set to 0.
NMsMOutRec	84	16 bytes	Binary	Output record descriptor. The client application should set this field to 0 on input. The server completes the field with information describing the records that contain the result data. This descriptor is described by the <i>NMsecOutRecDesc</i> structure. See Table 34 on page 476 for details.

Table 32. NMsecMessageHeader structure (continued)

Field	Offset	Length	Format	Description
NMsMOutRec2	100	16 bytes	Binary	Secondary output record descriptor. This descriptor identifies zero or more secondary result records for a given request. For example, a request might provide a single record containing global configuration information. The client application should set this field to 0 on input. The server completes this field with information describing the secondary result records. Secondary result records are provided only for certain requests; such requests describe the layout of the corresponding secondary result records. This field is described by the NMsecOutRecDesc structure. See Table 34 for details.
NMsMTarget	116	24 bytes	EBCDIC	The target for routing the request. Most request types apply to a single TCP/IP stack. The target field must contain the job name for that TCP/IP stack, right-padded with blanks. Rule: If the request applies to all stacks (this is valid only for the NMsec_GET_STACKINFO request), then this field must be filled with blanks. Result: The server echoes the client's target string on a reply message.

Input record descriptor

Table 33. Input record descriptor

Field	Offset	Length	Format	Description
NMsIROffset	0	4 bytes	Binary	Offset to the first input record, measured in bytes from the start of the message.
NMsIRRsvd1	4	2 bytes	Binary	Reserved; set to 0.
NMsIRNumber	6	2 bytes	Binary	Number of input records present in message.

Result: These fields are set to 0 on a reply message sent by the server.

Output record descriptor

Table 34. Output record descriptor

Field	Offset	Length	Format	Description
NMsOROffset	0	4 bytes	Binary	Offset to first output record, measured in bytes from the start of the message.

Table 34. Output record descriptor (continued)

Field	Offset	Length	Format	Description
NMsORTotal	4	4 bytes	Binary	Number of output records that would have been generated in the absence of input filters. If the request did not have input filters, or if input filters were not applicable for the request, the value of this field is the same as the NMsORNumber field value.
NMsORNumber	8	4 bytes	Binary	Number of output records present in message.
NMsORRsvd1	12	4 bytes	Binary	Reserved; set to 0.

The message header is followed by zero or more records. Records can vary in length. Each record consists of a record header, followed by one or more section descriptors that describe the sections within the record, followed by one or more sections that contain the actual record data. Conceptually, the layout of a message and its records is as shown in Figure 9.

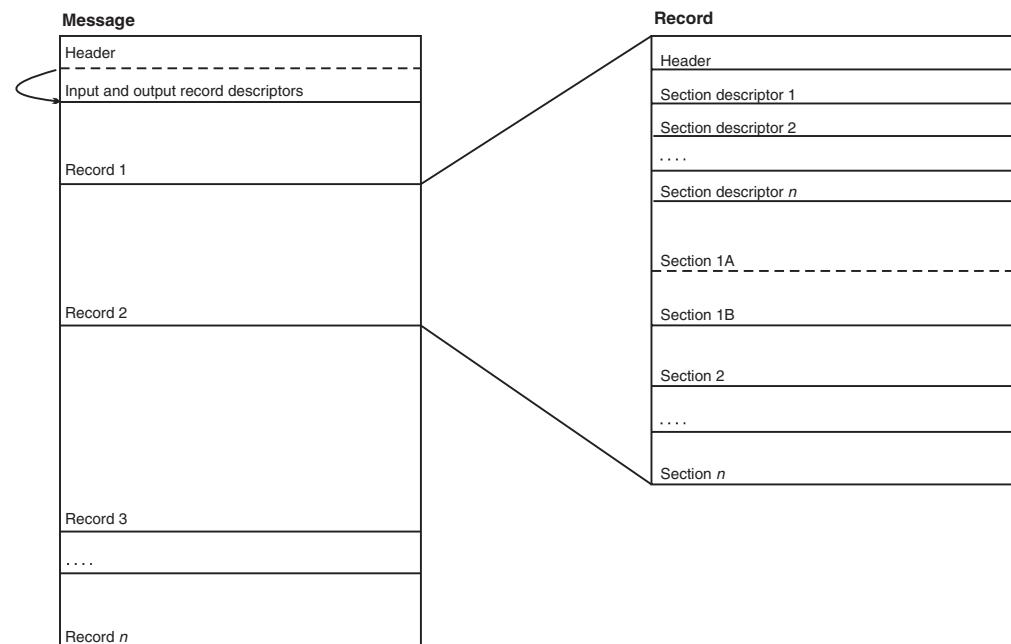


Figure 9. Message header and records

The record header is described by the NMsecRecordHdr structure.

Table 35. NMsecRecordHdr structure

Field	Offset	Length	Format	Description
NMsRIdent	0	4 bytes	EBCDIC	Record header identifier; set to NMsec_RECIDENT (EBCDIC 'NMsR').
NMsRLength	4	4 bytes	Binary	Total record length, including record header, section descriptors and data.

Table 35. *NMsecRecordHdr* structure (continued)

Field	Offset	Length	Format	Description
NMsRNumCascadeSecDesc	8	2 bytes	Binary	Number of cascading section descriptors present in this record.
NMsRNumSecDesc	10	2 bytes	Binary	Number of section descriptors present in this record.

A record's section descriptors immediately follow the record header. Standard section descriptors are described by the *NMsecSecDesc* structure.

Table 36. *NMsecSecDesc* structure

Field	Offset	Length	Format	Description
NMsSOffset	0	4 bytes	Binary	Offset from the start of record to the first section referenced by this section descriptor.
NMsSLength	4	4 bytes	Binary	Length of each section referenced by this section descriptor.
NMsSNumber	8	4 bytes	Binary	Number of sections referenced by this section descriptor (can be 0).

Each standard section descriptor describes a set of sections present in the record. Each descriptor indicates the offset to the first such section from the start of the record (not from the start of the message), the length of each section in the set, and the number of sections in the set. The number of sections can be zero. If there is more than one section, then the sections identified by a given descriptor are uniform in length. In Figure 9 on page 477, sections 1A and 1B are both described by section descriptor 1 and have the same length, and section 2 does not need to be present if the length or count of section 2 is 0.

A special kind of section descriptor called a cascading section descriptor indicates the offset and length of a section that contains a set of records of a different type. A section that contains these kinds of records is called a cascading record container section. The *NMsCSRecords* field of a cascading section descriptor indicates how many records are contained within the cascading record container section. These cascading constructs enable records of one type to be nested within another record. Cascading section descriptors appear after all standard descriptors. The record type determines the number of each type of descriptor. Cascading descriptors are described by the *NMsecCascadingSecDesc* structure.

Table 37. *NMsecCascadingSecDesc* structure

Field	Offset	Length	Format	Description
NMsCSOffset	0	4 bytes	Binary	Offset from the start of record to the cascading record container section referenced by this cascading section descriptor
NMsCSLength	4	4 bytes	Binary	Length of the cascading record container section referenced by this cascading section descriptor (can be 0)
NMsCSRecords	8	4 bytes	Binary	Number of records within the referenced cascading record container section (can be 0)

Following the section descriptors are the record's sections, whose location and number are described by the set of section descriptors. The format of the sections is determined by the message type for the message. For example, the first section

descriptor might identify a single section containing statistical data, while the second section descriptor might identify a section containing a variable-length IKE identity. In Figure 9 on page 477, sections 1A and 1B might always have the same length, but the length of section 2 in one record can differ from the length of section 2 in another record. Section 2 is not always present in every record, but section descriptor 2 is always present.

The records for a message can differ in length because some data is present or absent, or because there is variable-length data. However, all records in a message have the same type and format for the data that is present in those records. In other words, for a given message, all records have the same number of section descriptors, and the sections referenced by each descriptor have the same semantic content. However, data for any given section in each record within a message might or might not be present (data not present would be indicated by a section count value 0 in the associated section descriptor).

The size of any given structure that is contained in a section can increase from one release to the next, but the format of the data from the earlier release does not change. If new data is added to a section for a given release, it is added at the end of the section so that existing data mappings continue to resolve correctly without recompiling applications. To ensure that applications are compatible with future releases, if applications check the validity of a section's length, they should always test for a length that is greater than or equal to the expected length.

Result: If a message contains records (described by the NMsmOutRec field) and secondary records (described by the NMsmOutRec2 field), then the records and secondary records are not necessarily of the same type and format. See "Request messages" for details about the format the records and secondary records for each request.

Request messages

Client applications send request messages to the server. Request records contain the input parameters for the request. Input records for monitoring requests are called filter records or input filters. Control requests have a variety of input record formats. The following message types are supported by the server.

- Monitoring requests.

Access to each of these functions is controlled using the EZB.NETMGMT.sysname.tcpipname.IPSEC.DISPLAY resource definition in the SERVAUTH class, unless otherwise noted.

Each number in parentheses represents the value of the given request type constant, which is to be stored in the request message's NMsmType field.

- NMsec_GET_STACKINFO (2)—Obtain IP security configuration information for a given TCP/IP stack, or optionally obtain IP security configuration information for all active TCP/IP stacks.

Rule: To obtain configuration information for a specific TCP/IP stack, set the NMsmTarget field in the request message header to be the same as the value for the stack's job name. To obtain configuration information for all TCP/IP stacks, set the NMsmTarget field in the request message header to blanks.

- NMsec_GET_SUMMARY (3)—Retrieve summary IKE, IPSec, and IP filtering data from and for a particular stack.
- NMsec_GET_IPFLTCURR (4)—Retrieve detailed information about the currently active IP filters from a particular stack. These can be either the default IP filters (filters originating from the TCP/IP profile) or the policy IP filters (filters originating from Policy Agent).

- NMsec_GET_IPFLTDEFAULT (5)—Retrieve detailed information about the default IP filters (filters originating from the TCP/IP profile) from a particular stack.

Result: The default IP filters are returned regardless of whether they comprise the currently active filter set in use by the stack.

- NMsec_GET_IPFLTPOLICY (6)—Retrieve detailed information about the policy IP filters (filters originating from Policy Agent) from a particular stack.

Results:

- The policy IP filters are returned regardless of whether they are the currently active filter set in use by the stack.
- If Policy Agent has not installed IP filters in the stack, then a message containing zero filters is returned.
- NMsec_GET_PORTTRAN (7)—Retrieve IPv4 NAT traversal port translation information from a particular stack.
- NMsec_GET_IPTUNMANUAL (8)—Retrieve detailed information about manual tunnels from a particular stack.
- NMsec_GET_IPTUNDYNSTACK (9)—Retrieve detailed information about dynamic tunnels (phase 2 tunnels) from a particular stack.
- NMsec_GET_IPTUNDYNIKE (10)—Retrieve detailed IKE-related information about dynamic tunnels (phase 2 tunnels) for a particular stack.
- NMsec_GET_IKETUN (11)—Retrieve detailed information about IKE tunnels (phase 1 tunnels) for a particular stack.
- NMsec_GET_IKETUNCASCADE (12)—Retrieve detailed information about IKE tunnels for a particular stack, along with information about the associated dynamic tunnels (phase 2 tunnels) for each IKE tunnel.
- NMsec_GET_IPINTERFACES (13)—Retrieve the list of IP interfaces that belong to a particular stack.
- NMsec_GET_IKENSINFO (14)—Retrieve network security services information for the IKE daemon.

Rules:

- Access to this function is controlled using the EZB.NETMGMT.sysname.sysname.IKED.DISPLAY resource definition in the SERVAUTH class.
- Set the NMsmTarget field in the request message header to blanks for this request.

- Control requests.

Access to each of these functions is controlled using the EZB.NETMGMT.sysname.tcpipname.IPSEC.CONTROL resource definition in the SERVAUTH class

- NMsec_ACTIVATE_IPTUNMANUAL (1001)—Activate a manual tunnel.
- NMsec_ACTIVATE_IPTUNDYN (1002)—Activate a dynamic IPsec tunnel.
- NMsec_DEACTIVATE_IPTUNMANUAL (1003)—Deactivate a manual tunnel.
- NMsec_DEACTIVATE_IPTUNDYN (1004)—Deactivate a dynamic IPsec tunnel.
- NMsec_DEACTIVATE_IKETUN (1005)—Deactivate an IKE tunnel.
- NMsec_REFRESH_IPTUNDYN (1006)—Refresh a dynamic IPsec tunnel.
- NMsec_REFRESH_IKETUN (1007)—Refresh an IKE tunnel.
- NMsec_LOAD_POLICY (1008)—Switch between default IP filters and policy-based IP filters.

Monitoring request format

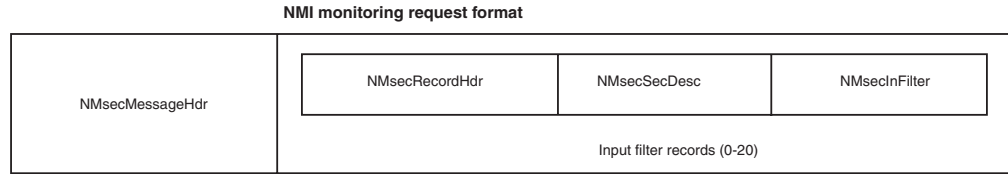


Figure 10. NMI monitoring request format

Monitoring requests that allow request records (not all of them do) call their request records filter records or input filters. If no input filters are provided, then all applicable data is returned over the interface. However, if input filters are provided, then the returned data is limited based on the input filters. Some requests do not support input filters (see Table 38 on page 482). Each input filter is specified by the client application in a separate record in the request message. Up to twenty input filter elements can be specified. Each input filter specifies one or more attribute to be restricted in the results returned over the interface. The attributes filtered by a single filter are combined with a logical AND; that is, all of the attributes must match a response record in order for that record to be returned over the interface. Multiple filters are combined with a logical OR; a response record needs to match only one input filter for that record to be returned over the interface.

Table 38 on page 482 shows which input filter specifications are valid for each request type. The filter specifications are described in detail in subsequent sections.

Table 38. Valid input filter specifications for request types

Filter specification	STACK INFO	SUMMARY	IPFLTCURR, IPFLTDEFAULT, IPFLTPOLICY	PORT TRAN	IPTUN MANUAL	IPTUN DYNSTACK	IPTUN DYNIKE	IKETUN, IKETUN CASCADE	IP INTER FACES	IKENS INFO
NMsfiltSrcAddr4			x			x	x			
NMsfiltSrcAddr6										
NMsfiltDstAddr4			x	x		x	x			
NMsfiltDstAddr6										
NMsfiltProtocol			x	x		x	x			
NMsfiltSrcPort			x			x	x			
NMsfiltDstPort			x	x		x	x			
NMsfiltLclEndpt4					x	x	x	x		
NMsfiltLclEndpt6										
NMsfiltRmtEndpt4					x	x	x	x		
NMsfiltRmtEndpt6										
NMsfiltTunnelID			x		x	x	x	x		
NMsfiltObjName			x (filter name)		x (VPN action)	x (VPN action)	x (VPN action)	x (Key exchange rule)		
NMsfiltObjGroupName			x (filter group name)							
NMsfiltAssocName			x (VPN action)			x (Dyn VPN rule)	x (Dyn VPN rule)			
NMsfiltFlagIPfitType			x							
NMsfiltSAState					x	x	x	x		
NMsfiltSWSShadow			x			x				

The client should provide each input filter element in a record in the request message. Each record should contain a record header, a single section descriptor identifying the input filter, and the input filter structure. The input filter is described by the NMsecInFilter structure. The input filter consists of a bit mask that indicates which filtering attributes are specified, followed by fields that specify the actual attribute values that are to be filtered. The layout of the structure is shown in Table 39, where bit 0 represents the high-order bit of a byte.

Table 39. NMsecInFilter structure

Field	Offset	Length	Format	Description
NMsFltFlagIPv6	0, bit 0	1 bit	Binary	IPv6 indicator. If set, addresses in input filter are IPv6, otherwise, addresses are IPv4.
NMsFltFlagSrcAddr	0, bit 1	1 bit	Binary	Source address indicator. If set, the input filter specifies a source address for filtering.
NMsFltFlagDstAddr	0, bit 2	1 bit	Binary	Destination address indicator. If set, the input filter specifies a destination address for filtering.
NMsFltFlagProto	0, bit 3	1 bit	Binary	Protocol indicator. If set, the input filter specifies an IP protocol number for filtering.
NMsFltFlagSrcPort	0, bit 4	1 bit	Binary	Source port indicator. If set, the input filter specifies a source port number for filtering.
NMsFltFlagDstPort	0, bit 5	1 bit	Binary	Destination port indicator. If set, the input filter specifies a destination port number for filtering.
NMsFltFlagLclEndpt	0, bit 6	1 bit	Binary	Local security endpoint indicator. If set, the input filter specifies a local security endpoint address for filtering.
NMsFltFlagRmtEndpt	0, bit 7	1 bit	Binary	Remote security endpoint indicator. If set, the input filter specifies a remote security endpoint address for filtering.
NMsFltFlagTunnelID	1, bit 0	1 bit	Binary	Tunnel ID indicator. If set, the input filter specifies a tunnel ID for filtering.
NMsFltFlagObjName	1, bit 1	1 bit	Binary	Object name indicator. If set, the input filter specifies an object name for filtering.
NMsFltFlagObjGrpName	1, bit 2	1 bit	Binary	Object group name indicator. If set, the input filter specifies an object group name for filtering.
NMsFltFlagAssocName	1, bit 3	1 bit	Binary	Associated object name indicator. If set, the input filter specifies an associated object name for filtering.
NMsFltFlagSASState	1, bit 4	1 bit	Binary	Security association state indicator. If set, the input filter specifies a security association state for filtering.
NMsFltFlagShadow	1, bit 5	1 bit	Binary	Shadow indicator. If set, the input filter specifies a SWSA shadow disposition for filtering.
NMsFltFlagIPFltType	1, bit 6	1 bit	Binary	IP filter type mask. If set for an IP filter request, the NMsIPFltTypexxx bits indicate the types of IP filters that match the input filter. If not set, the NMsIPFltTypexxx bits are ignored and IP filters of any type match. Details about IP filter mask types are listed in this table.

Table 39. NMsecInFilter structure (continued)

Field	Offset	Length	Format	Description
NMsFltRsvd1	1, bit 7	17 bits	Binary	Reserved bits. Must be set to 0.
NMsFltSrcAddr4	4	4 bytes	Binary	IPv4 or IPv6 source address selector. For an IP filter request the following apply: • If the IP filter represents outbound traffic, this input filter matches the IP filter if the IP filter contains this address within its source IP address specification. • If the IP filter represents inbound traffic, this input filter matches the IP filter if the IP filter contains this address within its destination IP address specification. For a dynamic IP tunnel request, this input filter matches the dynamic IP tunnel if the dynamic tunnel contains this address within its source IP address specification for tunnel data.
NMsFltSrcAddr6	4	16 bytes	Binary	
NMsFltDstAddr4	20	4 bytes	Binary	IPv4 or IPv6 destination address selector. For an IP filter request the following apply: • If the IP filter represents outbound traffic, this input filter matches the IP filter if the IP filter contains this address within its destination IP address specification. • If the IP filter represents inbound traffic, this input filter matches the IP filter if the IP filter contains this address within its source IP address specification. For a dynamic IP tunnel request, this input filter matches the dynamic IP tunnel if the dynamic tunnel contains this address within its destination IP address specification for tunnel data.
NMsFltDstAddr6	20	16 bytes	Binary	
NMsFltProtocol	36	1 byte	Binary	Protocol selector. For IP filter requests, port translation requests, and dynamic IP tunnel requests, this field limits the results based on the IP protocol number (corresponding to the IP protocol number in the IPv4 or IPv6 header). This input filter matches the result data if the result data contains this protocol within its IP protocol specification.
NMsFltRsvd2	37	1 byte	Binary	Reserved field. Must be set to 0.

Table 39. NMsecInFilter structure (continued)

Field	Offset	Length	Format	Description
NMsFltSrcPort	38	2 bytes	Binary	Source port selector. For an IP filter request the following apply: • If the IP filter represents outbound traffic, this input filter matches the IP filter if the IP filter contains this port within its source port specification. • If the IP filter represents inbound traffic, this input filter matches the IP filter if the IP filter contains this port within its destination port specification. For a dynamic IP tunnel request, this input filter matches the dynamic IP tunnel if the dynamic tunnel contains this port within its source port specification for tunnel data.
NMsFltDstPort	40	2 bytes	Binary	Destination port selector. For an IP filter request the following apply: • If the IP filter represents outbound traffic, this input filter matches the IP filter if the IP filter contains this port within its destination port specification. • If the IP filter represents inbound traffic, this input filter matches the IP filter if the IP filter contains this port within its source port specification. For a dynamic IP tunnel request, this input filter matches the dynamic IP tunnel if the dynamic tunnel contains this port within its destination port specification for tunnel data. For a port translation request, this input filter matches the port translation entry if the translated source port matches this port.
NMsFltRsvd3	42	2 bytes	Binary	Reserved field. Must be set to 0.
NMsFltLclEndpt4	44	4 bytes	Binary	Local security endpoint selector. For all IP and IKE tunnels, manual or dynamic, this input filter matches the tunnel if the tunnel's local security endpoint IP address is the same as this address.
NMsFltLclEndpt6	44	16 bytes	Binary	
NMsFltRmtEndpt4	60	4 bytes	Binary	Remote security endpoint selector. For all IP and IKE tunnels, manual or dynamic, this input filter matches the tunnel if the tunnel's remote security endpoint IP address is the same as this address.
NMsFltRmtEndpt6	60	16 bytes	Binary	
NMsFltTunnelID	76	48 bytes	EBCDIC	Tunnel ID selector. For all IKE and IP tunnels, manual or dynamic, this input filter matches the tunnel if the tunnel's tunnel ID matches this EBCDIC string. For IP filter requests, this input filter matches any IP filter associated with a manual or dynamic IP tunnel that has this tunnel ID.

Table 39. NMsecInFilter structure (continued)

Field	Offset	Length	Format	Description
NMsFltObjName	124	48 bytes	EBCDIC	Object name selector. For an IP filter request, this field limits the results based on the filter name. For IKE tunnels, this field limits the results based on the KeyExchangeRule name. For IP tunnels, this field limits the results based on the IPDynVpnAction or IPManVpnAction name.
NMsFltObjGroupName	172	48 bytes	EBCDIC	Group name selector. For an IP filter request, this field limits the results based on the filter group name.
NMsFltAssocName	220	48 bytes	EBCDIC	Associated object name selector. For an IP filter request, this field limits the results based on the IPDynVpnAction or IPManVpnAction name. For IP tunnels, this field limits the results based on the LocalDynVpnRule name.
NMsFltIPFltTypeGeneric	268, bit 0	1 bit	Binary	Generic IP filter mask. If set for an IP filter request, this input filter matches generic PERMIT and DENY IP filters. If not set, generic IP filters are not matched.
NMsFltIPFltTypeManual	268, bit 1	1 bit	Binary	Manual IP filter mask. If set for an IP filter request, this input filter matches IP filters referencing manual IP tunnels. If not set, manual IP filters are not matched.
NMsFltIPFltTypeDynAnchor	268, bit 2	1 bit	Binary	Dynamic anchor IP filter mask. If set for an IP filter request, this input filter matches IP filters that serve as anchors for dynamic IP tunnels. If not set, dynamic anchor IP filters are not matched.
NMsFltIPFltTypeDynamic	268, bit 3	1 bit	Binary	Dynamic IP filter mask. If set for an IP filter request, this input filter matches dynamic IP filters for dynamic IP tunnels. If not set, dynamic IP filters are not matched.
NMsFltIPFltTypeNATTAnchor	268, bit 4	1 bit	Binary	NATT anchor IP filter mask. If set for an IP filter request, this input filter matches IP filters that serve as anchors for NAT traversal IP tunnels. If not set, NATT anchor IP filters are not matched.
NMsFltIPFltTypeNATTDyn	268, bit 5	1 bit	Binary	NATT dynamic IP filter mask. If set for an IP filter request, this input filter matches dynamic IP filters for NAT traversal IP tunnels. If not set, NATT dynamic IP filters are not matched.
NMsFltIPFltTypeNRF	268, bit 6	1 bit	Binary	NATT resolution filter mask. If set for an IP filter request, this input filter matches NATT resolution filters for NAT traversal IP tunnels. If not set, NATT resolution filters are not matched.
NMsFltRsvd4	268, bit 7	9 bits	Binary	Reserved bits. Must be set to 0.

Table 39. NMsecInFilter structure (continued)

Field	Offset	Length	Format	Description
NMsFltSAState	270	1 byte	Binary	SA state selector. For an IP or IKE tunnel request, this field limits the results based on the security association (SA) state. Valid state values are as follows: NMsec_SASTATE_INACTIVE (1) Tunnel is inactive NMsec_SASTATE_PENDING (2) Tunnel is awaiting negotiation NMsec_SASTATE_INCOMPLETE (3) Tunnel is in negotiation NMsec_SASTATE_ACTIVE (4) Tunnel is active NMsec_SASTATE_EXPIRED (5) Tunnel is expired
NMsFltSWSAShadow	271	1 byte	Binary	SWSA shadow indicator. This is applicable for IP filters and for dynamic IP tunnels. Valid values are as follows: NMsec_SHADOW (1) Match objects that are SWSA shadow objects originating from a remote distributor only. NMsec_NONSHADOW (0) Match objects that are not SWSA shadow objects only.

Control request formats

Control request record formats vary with each request type.

- The following section, NMsecTunnel, is used across several record types.
- All EBCDIC fields are blank-padded and they are not NUL-terminated.

Table 40. NMsecTunnel field descriptions

Field	Offset	Length	Format	Description
NMsTunName	0	48 bytes	EBCDIC	The name that is associated with the tunnel. This name comes from a Policy Agent configuration file. • For manual tunnels, this is an IpManVpnActionName name. • For dynamic IPSec tunnels, this is a LocalDynVpnRuleName name. • For IKE tunnels, this is a KeyExchangeRuleName name. This field must be set to blanks when a tunnel name is not specified.
NMsTunTunnelID	48	48 bytes	EBCDIC	The tunnel ID that is associated with this tunnel. This field is used for any refresh and deactivation requests. This field must be set to blanks when a tunnel ID is not specified.

Table 40. NMsecTunnel field descriptions (continued)

Field	Offset	Length	Format	Description
NMsTunStatus	96	1 byte	Binary	Tunnel Status. This field is set to 0 on a request message. On a response this field is set to the status of the tunnel's state change. Valid state values are as follows: NMsec_TUNSTATUS_NOTFOUND (1) The requested tunnel was not found. NMsec_TUNSTATUS_STATEUPDATED (2) The tunnel's state was updated. NMsec_TUNSTATUS_STATEALREADYSET (3) The tunnel's state was already set to the state requested. NMsec_TUNSTATUS_NOKER (5) Applicable only to dynamic tunnel activation, this status indicates that there is no KeyExchangeRule rule corresponding to the requested LocalDynVpnRule rule. NMsec_TUNSTATUS_NOFILTER (6) Applicable only to dynamic tunnel activation, this status indicates that there is no dynamic IPsec IpFilterRule rule corresponding to the requested LocalDynVpnRule rule.
NMsTunRsvd1	97	3 bytes	Binary	Reserved. Must be set to 0.

NMsec_ACTIVATE_IPTUNMANUAL

NMsec_ACTIVATE_IPTUNMANUAL request format

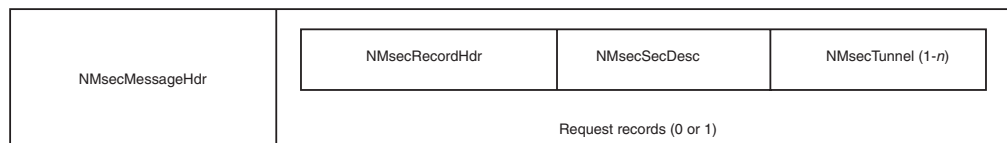


Figure 11. NMsec_ACTIVATE_IPTUNMANUAL request form

Activates one or more manual tunnels. The request format contains zero or one record with one fixed-length section that contains one or more NMsecTunnel instances (described in Table 40 on page 487). Each NMsecTunnel instance identifies a manual tunnel to activate. If the Request Record is not present then *all* manual tunnels are activated.

Restriction: Manual tunnel activation requests for multiple tunnels must contain uniform tunnel specifications, either tunnel IDs or tunnel names.

NMsec_ACTIVATE_IPTUNDYN

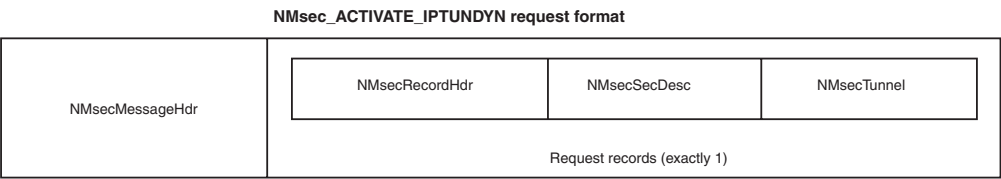


Figure 12. NMsec_ACTIVATE_IPTUNDYN request format

Activates a dynamic IPSec tunnel. Each record has one section, NMsecTunnel (described in Table 40 on page 487). The NMsecTunnel section identifies the dynamic tunnel to be activated.

NMsec_DEACTIVATE_IPTUNMANUAL

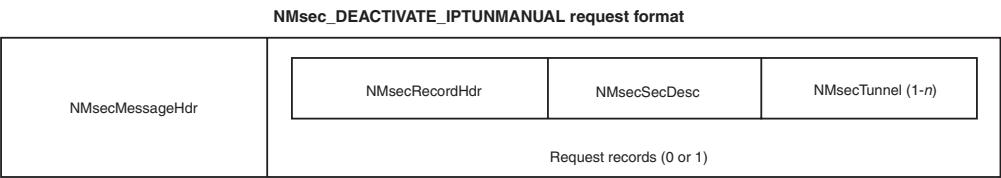


Figure 13. NMsec_DEACTIVATE_IPTUNMANUAL request format

Deactivates one or more manual tunnels. The request format contains zero or one record with one fixed-length section that contains one or more NMsecTunnel instances (described in Table 40 on page 487). Each NMsecTunnel instance identifies a manual tunnel to deactivate. If the request record is not present then all manual tunnels are deactivated.

Restriction: Manual tunnel deactivation requests for multiple tunnels must contain uniform tunnel specifications, either tunnel IDs or tunnel names.

NMsec_DEACTIVATE_IPTUNDYN

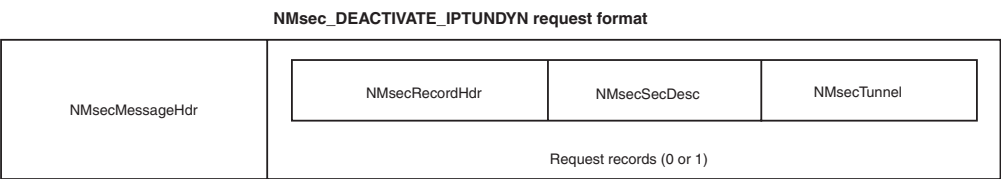


Figure 14. NMsec_DEACTIVATE_IPTUNDYN request format

Deactivates one or all dynamic tunnels. The request format contains zero or one record with one section, NMsecTunnel (described in Table 40 on page 487). The NMsecTunnel section identifies the dynamic tunnel to be deactivated. If the Request Record is not present then *all* dynamic tunnels are deactivated.

NMsec_DEACTIVATE_IKETUN

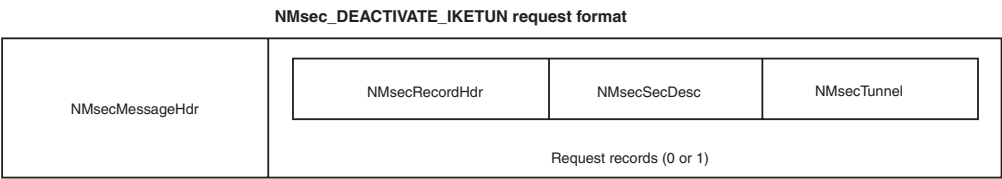


Figure 15. NMsec_DEACTIVATE_IKETUN request format

Deactivates one or all IKE tunnels. The request format contains zero or one record with one section, NMsecTunnel, (described in Table 40 on page 487). The NMsecTunnel section identifies the IKE tunnel to be deactivated. If the Request Record is not present then all IKE tunnels are deactivated.

NMsec_REFRESH_IPTUNDYN

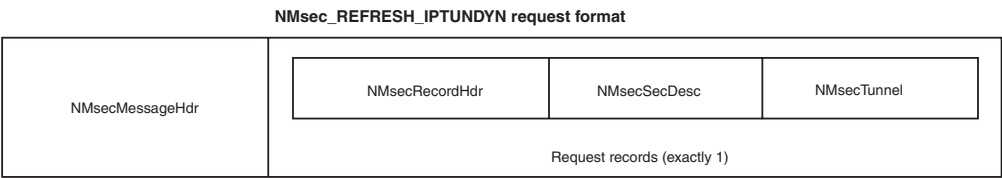


Figure 16. NMsec_REFRESH_IPTUNDYN request format

Refreshes a dynamic IPsec tunnel. Contains a single record that has one section, NMsecTunnel (described in Table 40 on page 487). The NMsecTunnel section identifies the dynamic tunnel to be refreshed.

NMsec_REFRESH_IKETUN

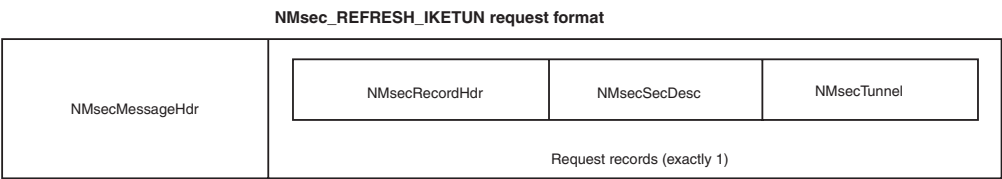


Figure 17. NMsec_REFRESH_IKETUN request format

Refreshes an IKE tunnel. Contains a single record that has one section, NMsecTunnel (described in Table 40 on page 487). The NMsecTunnel section identifies the IKE tunnel to be refreshed.

NMsec_LOAD_POLICY

Switches between default IP filters and policy-based IP filters. The call indicates whether the default policy or configured policy should be loaded. After this call completes, the client will have initiated the policy load operation.

Selecting the NMsec_FLT_DEFAULT option causes the stack to use the default IP filter rules. Default IP filter rules consist of the IP filter rules that are specified by the TCPIP profile, if any, and an implicit DENY-ALL rule. While the profile IP filters are in effect, manual, dynamic, and IKE tunnels still exist, but they are not used. These tunnels might expire or be deactivated. Tunnel refreshes might not occur and new dynamic tunnels might not be activated.

Switching between default and configured policy is useful when there is a need to quickly restrict system access to a very small subset of allowable traffic. This might occur when a system is under some sort of security attack or just before going into a maintenance state.

Selecting the NMsec_FLT_POLICY option causes the stack to use the policy IP filter rules as supplied from a policy configuration file or server. If no policy IP filters were previously defined to the stack, the stack continues to use the default IP filter rules until the policy configuration file is installed by the Policy Agent. If policy IP filter rules were previously defined to the stack, those policy IP filters become effective again. Tunnel activity can resume, including refreshes and new activations. The IKE daemon attempts to perform all configured autoactivations.

The active policy definitions (default or configured) are remembered across activations of the stack and system IPLs.

Each record has one section, NMsecPolicySource, which contains the following data.

Table 41. NMsecPolicySource data

Field	Offset	Length	Format	Description
NMsPolSrcSource	0	1 byte	Binary	Indicates which policy should be loaded or reloaded. One of the following: NMsec_FLT_POLICY (1) NMsec_FLT_DEFAULT (0)
NMsPolSrcRsvd1	1	3 bytes	Binary	Reserved. Set to zeroes.

Response messages

Response messages contain zero or more response records. The layout of each record depends on the message type. The fields in the response records for each request type are described in the following sections. Some section layouts are shared between several record types.

All EBCDIC fields are blank-padded and are not NUL-terminated.

NMsec_GET_STACKINFO

NMsec_GET_STACKINFO response format

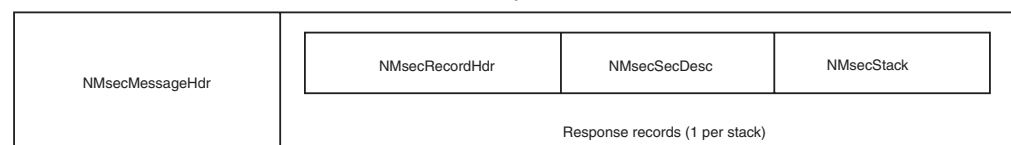


Figure 18. NMsec_GET_STACKINFO response format

Each record returned identifies a single stack that is active on the system. Each record has one section, NMsecStack, with the following data, where bit 0 is the high-order bit of each byte.

Table 42. NMsecStack structure

Field	Offset	Length	Format	Description
NMsStackIPSecurity	0, bit 0	1 bit	Binary	If set, IP security is enabled for this stack.

Table 42. NMsecStack structure (continued)

Field	Offset	Length	Format	Description
NMsStackIPv6Security	0, bit 1	1 bit	Binary	If set, IPv6 IP security is enabled for this stack.
NMsStackDVIPSec	0, bit 2	1 bit	Binary	If set, sysplex-wide security associations (DVIPSEC) is enabled for this stack.
NMsStackLogging	0, bit 3	1 bit	Binary	If set, filter logging is enabled for this stack.
NMsStackPreDecap	0, bit 4	1 bit	Binary	If set, pre-decapsulation filtering is enabled for this stack.
NMsStackFilterSet	0, bit 5	1 bit	Binary	Current filter set indicator. Possible values are: NMsec_FLT_DEFAULT (0) Default filters are currently in effect. The default filters originate from the TCP/IP profile. NMsec_FLT_POLICY (1) Policy filters are currently in effect. The policy filters originate in the Policy Agent configuration.
NMsStackRsvd1	0, bit 6	26 bits	Binary	Reserved bits.
NMsStackName	4	24 bytes	EBCDIC	The job name of the TCP/IP stack.
NMsStackNATKeepAlive	28	4 bytes	Binary	NAT keep-alive interval, in seconds, used to regulate sending NAT keep-alive messages for a NAT traversal tunnel when a NAT device is detected in front of the local host.
NMsStackFilterCount	32	4 bytes	Binary	Number of configured filters in the current filter set. This number does not include any dynamic filters.

NMsec_GET_SUMMARY

NMsec_GET_SUMMARY response format

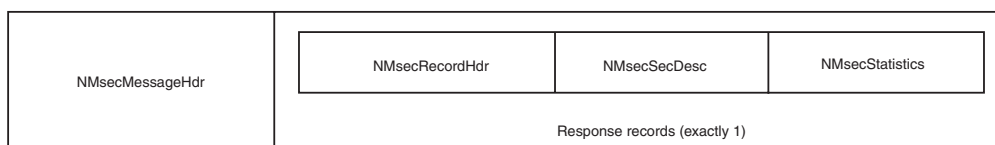


Figure 19. NMsec_GET_SUMMARY response format

For the requested stack, one record is returned, which indicates statistical data. This record has a single section, NMsecStatistics, that contains the following data.

Table 43. NMsec_GET_SUMMARY structure

Field	Offset	Length	Format	Description
NMsStatP1Active	0	4 bytes	Binary	Current number of active IKE tunnels.
NMsStatP1InProgress	4	4 bytes	Binary	Current number of IKE tunnels in-progress, either pending or in negotiation.
NMsStatP1Expired	8	4 bytes	Binary	Current number of expired IKE tunnels. This is a current count (not cumulative). Expired IKE tunnels are retained until all associated dynamic tunnels have expired.
NMsStatP1LclActSuccess	12	8 bytes	Binary	Cumulative number of successful IKE tunnel activations that were initiated locally for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.
NMsStatP1RmtActSuccess	20	8 bytes	Binary	Cumulative number of successful IKE tunnel activations that were initiated remotely for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.
NMsStatP1LclActFailure	28	8 bytes	Binary	Cumulative number of failed IKE tunnel activations that were initiated locally for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.
NMsStatP1RmtActFailure	36	8 bytes	Binary	Cumulative number of failed IKE tunnel activations that were initiated remotely for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.
NMsStatP1Retransmit	44	8 bytes	Binary	Cumulative number of retransmitted key exchange (phase 1) messages sent for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.
NMsStatP1Replay	52	8 bytes	Binary	Cumulative number of replayed key exchange (phase 1) messages received for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.

Table 43. NMsec_GET_SUMMARY structure (continued)

Field	Offset	Length	Format	Description
NMsStatP1Invalid	60	8 bytes	Binary	Cumulative number of key exchange (phase 1) messages that are not valid that have been received for this stack over the life of the IKE daemon. This number does not include message authentication failures. This data is cumulative even across stack restarts.
NMsStatP1AuthFail	68	8 bytes	Binary	Cumulative number of key exchange (phase 1) message authentication failures for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.
NMsStatP2Active	76	4 bytes	Binary	Current number of active dynamic tunnels known to the TCP/IP stack. This number does not include SWSA shadow tunnels or manual tunnels.
NMsStatP2ActiveShadow	80	4 bytes	Binary	Current number of active dynamic SWSA shadow tunnels known to the TCP/IP stack.
NMsStatP2InProgress	84	4 bytes	Binary	Current number of dynamic tunnels in progress, either pending or in negotiation.
NMsStatP2Expired	88	4 bytes	Binary	Current number of expired dynamic tunnels known to the TCP/IP stack. This includes both non-shadow and shadow tunnels.
NMsStatP2ActSuccess	92	8 bytes	Binary	Cumulative number of successful dynamic tunnel activations for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.
NMsStatP2ActFailure	100	8 bytes	Binary	Cumulative number of failed dynamic tunnel activations for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.
NMsStatP2Retransmit	108	8 bytes	Binary	Cumulative number of retransmitted QUICKMODE (phase 2) messages sent for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.

Table 43. *NMsec_GET_SUMMARY* structure (continued)

Field	Offset	Length	Format	Description
NMsStatP2Replay	116	8 bytes	Binary	Cumulative number of replayed QUICKMODE (phase 2) messages received for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.
NMsStatP2Invalid	124	8 bytes	Binary	Cumulative number of QUICKMODE (phase 2) messages that were not valid that were received for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.
NMsStatP1BytesOut	132	8 bytes	Binary	Cumulative number of outbound bytes of IKE traffic protected by IKE tunnels for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.
NMsStatP1BytesIn	140	8 bytes	Binary	Cumulative number of inbound bytes of IKE traffic protected by IKE tunnels for this stack over the life of the IKE daemon. This data is cumulative even across stack restarts.
NMsStatP2BytesOut	148	8 bytes	Binary	Cumulative number of outbound bytes of IP traffic protected by dynamic tunnels for this stack over the life of the TCP/IP stack.
NMsStatP2BytesIn	156	8 bytes	Binary	Cumulative number of inbound bytes of IP traffic protected by dynamic tunnels for this stack over the life of the TCP/IP stack.
NMsStatFilterDeny	164	8 bytes	Binary	Cumulative number of packets denied as the result of IP filter action DENY for this stack over the life of the TCP/IP stack.
NMsStatFilterMismatch	172	8 bytes	Binary	Cumulative number of packets denied as the result of mismatch with filter action for this stack over the life of the TCP/IP stack.
NMsStatFilterMatch	180	8 bytes	Binary	Cumulative number of packets matching an IP filter over the life of the TCP/IP stack. This includes generic (permit and deny) filters and IPSec filters.

NMsec_GET_IPFLTCURR, NMsec_GET_IPFLTDEFAULT, and NMsec_GET_IPFLTPOLICY

NMsec_GET_IPFLTCURR, NMsec_GET_IPFLTDEFAULT, NMsec_GET_IPFLTPOLICY response format

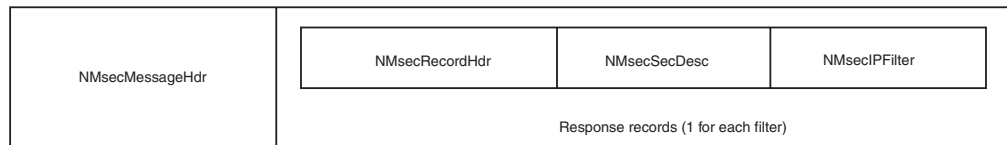


Figure 20. NMsec_GET_IPFLTCURR, NMsec_GET_IPFLTDEFAULT, and NMsec_GET_IPFLTPOLICY response format

For the requested stack, zero or more records are returned representing IP filters. Each record that is returned identifies a single IP filter and contains a single section, NMsecIPFilter, that contains the data shown in Table 44.

Filters are presented in an ordered sequence. Generic IP filters (permit or deny), manual tunnel filters, and dynamic anchor filters are presented in the order in which they are configured. Dynamic anchor filters are presented immediately before the dynamic and NATT anchor filters that are associated with them. NATT anchor filters are presented immediately before the NATT dynamic filters that are associated with them. NAT traversal resolution filters (NRFs) for a NATT anchor filter are presented immediately after the NATT dynamic filters for that NATT anchor filter. Some IP filters might be absent from the result list because of input filters provided on the request message.

Table 44. NMsecIPFilter structure

Field	Offset	Length	Format	Description
NMsIPFltName	0	48 bytes	EBCDIC	Filter rule name. Bytes 41-48 of the filter rule name consist of the filter rule name extension, a numeric extension used to distinguish between distinct filter rules that result from the same configured filter rule.
NMsIPFltGroupName	48	48 bytes	EBCDIC	Filter rule group name or blank if there is no filter group.
NMsIPFltLSAName	96	48 bytes	EBCDIC	Local start action name or blank if there is no local start action.
NMsIPFltVPNActionName	144	48 bytes	EBCDIC	VPN action name or blank if there is no VPN action.
NMsIPFltTunnelID	192	48 bytes	EBCDIC	Associated tunnel ID or blank if there is no associated tunnel.
NMsIPFltFlagIPv6	240, bit 0	1 bit	Binary	IPv6 indicator. If set, IP addresses for traffic and security endpoints are IPv6; otherwise they are IPv4.
NMsIPFltFlagOnDemand	240, bit 1	1 bit	Binary	On-demand indicator. If set for a dynamic anchor filter, a dynamic filter, a NAT traversal anchor filter, or a NAT traversal dynamic filter, this value indicates that on-demand activations are permitted for this traffic specification.
NMsIPFltFlagShadow	240, bit 2	1 bit	Binary	SWSA shadow indicator. If set for a dynamic filter, this value indicates that the filter originated from a distributing stack.
NMsIPFltFlagSrcIsSingle	240, bit 3	1 bit	Binary	Single source address indicator. If set, the source address is indicated by the NMsIPFltSrcAddr4 or NMsIPFltSrcAddr6 field.
NMsIPFltFlagSrcIsPrefix	240, bit 4	1 bit	Binary	Prefixed source address indicator. If set, the source address is indicated by the NMsIPFltSrcAddr4 or NMsIPFltSrcAddr6 field, and the source address prefix is indicated by the NMsIPFltSrcAddrPrefix field.

Table 44. NMsecIPFilter structure (continued)

Field	Offset	Length	Format	Description
NMsiPFItFlagSrcIsRange	240, bit 5	1 bit	Binary	Ranged source address indicator. If set, the source address range is indicated by the NMsiPFItSrcAddr4 and NMsiPFItSrcAddrRange4 fields, or the NMsiPFItSrcAddr6 and NMsiPFItSrcAddrRange6 fields.
NMsiPFItFlagDstIsSingle	240, bit 6	1 bit	Binary	Single destination address indicator. If set, the destination address is indicated by the NMsiPFItDstAddr4 or NMsiPFItDstAddr6 field.
NMsiPFItFlagDstIsPrefix	240, bit 7	1 bit	Binary	Prefixed destination address indicator. If set, the destination address is indicated by the NMsiPFItDstAddr4 or NMsiPFItDstAddr6 field, and the destination address prefix is indicated by the NMsiPFItDstAddrPrefix field.
NMsiPFItFlagDstIsRange	241, bit 0	1 bit	Binary	Ranged destination address indicator. If set, the destination address range is indicated by the NMsiPFItDstAddr4 and NMsiPFItDstAddrRange4 fields, or the NMsiPFItDstAddr6 and NMsiPFItDstAddrRange6 fields.
NMsiPFItFlagProtoDef	241, bit 1	1 bit	Binary	Protocol indicator. If set, the filter protocol is indicated by the NMsiPFItProtocol field, otherwise, the filter covers all protocols.
NMsiPFItFlagSrcPortDef	241, bit 2	1 bit	Binary	Source port indicator. If set, the source port range is indicated by the NMsiPFItSrcPort and NMsiPFItSrcPortRange fields; otherwise, the filter covers all source ports. This indicator is not valid and is 0 if the filter protocol is not TCP or UDP.
NMsiPFItFlagDstPortDef	241, bit 3	1 bit	Binary	Destination port indicator. If set, the destination port range is indicated by the NMsiPFItDstPort and NMsiPFItDstPortRange fields; otherwise, the filter covers all destination ports. This indicator is not valid and is 0 if the filter protocol is not TCP or UDP.
NMsiPFItFlagICMPTypeDef	241, bit 4	1 bit	Binary	ICMP type indicator. If set, the ICMP type is indicated by the NMsiPFItICMPType field; otherwise, the filter covers all ICMP types. This indicator is not valid and is 0 if the filter protocol is not ICMP or ICMPv6.
NMsiPFItFlagICMPCodeDef	241, bit 5	1 bit	Binary	ICMP code indicator. If set, the ICMP code is indicated by the NMsiPFItICMPCode field; otherwise, the filter covers all ICMP codes. This indicator is not valid and is 0 if the filter protocol is not ICMP or ICMPv6.
NMsiPFItFlagOSPFTypeDef	241, bit 6	1 bit	Binary	OSPF type indicator. If set, the OSPF type is indicated by the NMsiPFItOSPFType field; otherwise, the filter covers all OSPF types. This indicator is not valid and is 0 if the filter protocol is not OSPF.
NMsiPFItFlagSrcAddrPktGran	241, bit 7	1 bit	Binary	Source address granularity indicator. If set for a dynamic anchor filter, on-demand activations use packet source address; otherwise, they use the filter source address specification.
NMsiPFItFlagDstAddrPktGran	242, bit 0	1 bit	Binary	Destination address granularity indicator. If set for a dynamic anchor filter, on-demand activations use packet destination address; otherwise, they use the filter destination address specification.
NMsiPFItFlagProtoPktGran	242, bit 1	1 bit	Binary	Protocol granularity indicator. If set for a dynamic anchor filter, on-demand activations use packet protocol; otherwise, they use the filter protocol.
NMsiPFItFlagSrcPortPktGran	242, bit 2	1 bit	Binary	Source port granularity indicator. If set for a dynamic anchor filter, on-demand activations use a packet source port; otherwise they use the filter source port specification, where possible.

Table 44. NMsecIPFilter structure (continued)

Field	Offset	Length	Format	Description
NMsecIPFilterFlagDstPortPktGran	242, bit 3	1 bit	Binary	Destination port granularity indicator. If set for a dynamic anchor filter, on-demand activations use packet destination port; otherwise, they use the filter destination port specification, where possible.
NMsecIPFilterFlagNATDetect	242, bit 4	1 bit	Binary	NAT indicator. If set for a dynamic filter, a NAT has been detected in front of the IPSec peer.
NMsecIPFilterFlagNAPTDetect	242, bit 5	1 bit	Binary	NAPT indicator. If set for a dynamic filter, a NAPT has been detected in front of the IPSec peer. It is possible that a NAPT might exist but that it is detected only as a NAT.
NMsecIPFilterFlagGWDetect	242, bit 6	1 bit	Binary	NAT traversal gateway indicator. If set for a dynamic filter, the tunnel uses UDP encapsulation and the peer is acting as an IPSec gateway.
NMsecIPFilterFlagLogPermit	242, bit 7	1 bit	Binary	LogPermit indicator. If set, permitted packets that match this filter are logged.
NMsecIPFilterFlagLogDeny	243, bit 0	1 bit	Binary	LogDeny indicator. If set, denied packets that match this filter are logged.
NMsecIPFilterRsvd1	243, bit 1	7 bits	Binary	Reserved bits.
NMsecIPFilterType	244	1 byte	Binary	IP filter type. One of the following: NMsec_IPFLT_GENERIC (1) NMsec_IPFLT_MANUAL (2) NMsec_IPFLT_DYNANCHOR (3) NMsec_IPFLT_DYNAMIC (4) NMsec_IPFLT_NATTANCHOR (5) NMsec_IPFLT_NATTDYN (6) NMsec_IPFLT_NRF (7)
NMsecIPFilterState	245	1 byte	Binary	IP filter state. One of the following: NMsec_IPFLT_INACTIVE (0) Filter is inactive as a result of time condition NMsec_IPFLT_ACTIVE (1) Filter is active
NMsecIPFilterAction	246	1 byte	Binary	IP filter action. One of the following: NMsec_IPFLT_PERMIT (1) NMsec_IPFLT_DENY (2) NMsec_IPFLT_IPSEC (3)
NMsecIPFilterScope	247	1 byte	Binary	IP filter scope. One of the following: NMsec_IPFLT_LOCAL (1) NMsec_IPFLT_ROUTED (2) NMsec_IPFLT_SCOPEALL (3)
NMsecIPFilterDirection	248	1 byte	Binary	IP filter direction. One of the following: NMsec_IPFLT_INBOUND (1) NMsec_IPFLT_OUTBOUND (2)
NMsecIPFilterSecurityClass	249	1 byte	Binary	IP filter security class. Valid values are in the range 0 - 255. The value 0 matches all security classes.
NMsecIPFilterTCPConnect	250	1 byte	Binary	TCP connect qualifier. One of the following: NMsec_IPFLT_CONNECT_NONE (0) NMsec_IPFLT_CONNECT_INBOUND (1) NMsec_IPFLT_CONNECT_OUTBOUND (2)
NMsecIPFilterRsvd2	251	1 byte	Binary	Reserved.
NMsecIPFilterProtocol	252	1 byte	Binary	IP filter protocol number, if the NMsecIPFilterFlagProtoDef field is set. This value corresponds to the IP protocol number in the IPv4 or IPv6 header.

Table 44. NMsecIPFilter structure (continued)

Field	Offset	Length	Format	Description
NMsiPFItICMPType	253	1 byte	Binary	ICMP type, if the NMsiPFItFlagICMPTypeDef field is set.
NMsiPFItICMPCode	254	1 byte	Binary	ICMP code, if the NMsiPFItFlagICMPCodeDef field is set.
NMsiPFItOSPFType	255	1 byte	Binary	OSPF type, if the NMsiPFItFlagOSPFTypeDef field is set.
NMsiPFItSrcPort	256	2 bytes	Binary	Low end of IP filter source port range, if the NMsiPFItFlagSrcPortDef field is set.
NMsiPFItSrcPortRange	258	2 bytes	Binary	High end of IP filter source port range, if the NMsiPFItFlagSrcPortDef field is set.
NMsiPFItDstPort	260	2 bytes	Binary	Low end of IP filter destination port range, if NMsiPFItFlagDstPortDef field is set.
NMsiPFItDstPortRange	262	2 bytes	Binary	High end of IP filter destination port range, if the NMsiPFItFlagDstPortDef field is set.
NMsiPFItSrcAddr4	264	4 bytes	Binary	• If the NMsiPFItFlagSrcIsSingle field is set, the filter's IPv4 or IPv6 source address • If the NMsiPFItFlagSrcIsPrefix field is set, the filter's IPv4 or IPv6 source address base • If the NMsiPFItFlagSrcIsRange field is set, the low end of the filter's IPv4 or IPv6 source address range
NMsiPFItSrcAddr6	264	16 bytes	Binary	
NMsiPFItSrcAddrRange4	280	4 bytes	Binary	If the NMsiPFItFlagSrcIsRange field is set, the high end of the filter's IPv4 or IPv6 source address range.
NMsiPFItSrcAddrRange6	280	16 bytes	Binary	
NMsiPFItDstAddr4	296	4 bytes	Binary	• If the NMsiPFItFlagDstIsSingle field is set, the filter's IPv4 or IPv6 destination address • If the NMsiPFItFlagDstIsPrefix field is set, the filter's IPv4 or IPv6 destination address base • If the NMsiPFItFlagDstIsRange field is set, the low end of the filter's IPv4 or IPv6 destination address range
NMsiPFItDstAddr6	296	16 bytes	Binary	
NMsiPFItDstAddrRange4	312	4 bytes	Binary	If the NMsiPFItFlagDstIsRange field is set, the high end of the filter's IPv4 or IPv6 destination address range.
NMsiPFItDstAddrRange6	312	16 bytes	Binary	
NMsiPFItSrcAddrPrefix	328	1 byte	Binary	If the NMsiPFItFlagSrcIsPrefix field is set, the length of the filter's source address prefix in bits.
NMsiPFItDstAddrPrefix	329	1 byte	Binary	If the NMsiPFItFlagDstIsPrefix field is set, the length of the filter's destination address prefix in bits.
NMsiPFItRsvd3	330	1 byte	Binary	Reserved.

Table 44. NMsecIPFilter structure (continued)

Field	Offset	Length	Format	Description
NMsecIPFilterNATTClientIDType	331	1 byte	Binary	The NATT client ID is present only if the peer is behind a NAT and a gateway, and the peer supplied a client ID. One of the following: NMsec_IPFLT_IDNONE (0) No client ID. Either this is not a dynamic filter; the peer for this filter's tunnel is not behind a NAT and a gateway; or no client ID was provided. NMsec_IPFLT_IDIP (1) Client ID is an IPv4 address. NMsec_IPFLT_IDRANGE (2) Client ID is an IPv4 address range. NMsec_IPFLT_IDPREFIX (3) Client ID is an IPv4 address prefix. NMsec_IPFLT_IDOTHER (4) Client ID is another type, represented as an MD5 hash of the ID data.
NMsecIPFilterNATTClientIDIP	332	4 bytes	Binary	If NATT client ID type is NMsec_IPFLT_IDIP, NMsec_IPFLT_IDRANGE, or NMsec_IPFLT_IDPREFIX, this field is the base IPv4 address for the client ID.
NMsecIPFilterNATTClientIDHash	332	16 bytes	Binary	If NATT client ID type is NMsec_IPFLT_IDOTHER, this field is the MD5 hash of the client's ID.
NMsecIPFilterNATTClientIDIP2	348	4 bytes	Binary	If NATT client ID type is NMsec_IPFLT_IDRANGE, this field is the high end of the IPv4 client ID address range.
NMsecIPFilterNATTClientIDPrefix	348	4 bytes	Binary	If NATT client ID type is NMsec_IPFLT_IDPREFIX, this field is the prefix length of the IPv4 client ID, in bits.
NMsecIPFilterRsvd4	348	16 bytes	Binary	Reserved.
NMsecIPFilterNATTPeerPort	364	2 bytes	Binary	If this is a dynamic filter for UDP-encapsulated NAT traversal traffic, this field is the UDP port for the IKE peer, otherwise the value is 0.
NMsecIPFilterNATTNRFOrigPort	366	2 bytes	Binary	If this is a NAT traversal Resolution Filter, this field is the original remote port for the TCP or UDP traffic, otherwise the value is 0.
NMsecIPFilterMismatch	368	8 bytes	Binary	The cumulative number of packets denied as a result of a mismatch with this filter's action over the life of the TCP/IP stack.
NMsecIPFilterMatch	376	8 bytes	Binary	The cumulative number of packets that matched this filter's condition and action over the life of the TCP/IP stack.
NMsecIPFilterCreateTime	384	4 bytes	Binary	For a statically defined filter that originates from the Policy Agent configuration, this field contains the UNIX time stamp that indicates when the filter was first defined to the current instance of the TCP/IP stack. For a filter that originates from the TCP/IP profile, this field contains the UNIX time stamp that indicates when the profile filter configuration was last replaced. For all dynamically defined filters, the value in this field is 0.

Table 44. NMsecIPFilter structure (continued)

Field	Offset	Length	Format	Description
NMsiPFiltUpdateTime	388	4 bytes	Binary	For a statically defined filter that originates from the Policy Agent configuration, this field contains the UNIX time stamp that indicates when the filter's attributes were last updated in the current instance of the TCP/IP stack. For a filter that originates from the TCP/IP profile, this field contains the UNIX time stamp that indicates when the profile filter configuration was last replaced. For all dynamically defined filters, the value in this field is 0.

Each of the IP filter requests also returns a single secondary output record (described by the NMsmOutRec2 output record descriptor). This record describes global IP filtering configuration information that is currently in effect for the TCP/IP stack. This global result record contains a single section. This section consists of an NMsecStack structure, which is described “NMsec_GET_STACKINFO” on page 491 for the NMsec_GET_STACKINFO request.

NMsec_GET_PORTTRAN

NMsec_GET_PORTTRAN response format

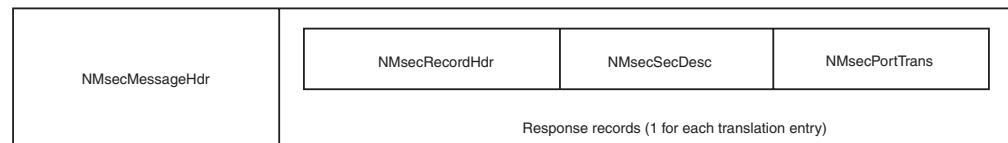


Figure 21. NMsec_GET_PORTTRAN response format

For the requested stack, zero or more records are returned representing NAT traversal port translation entries. Each record that is returned contains a single section, NMsecPortTrans, which contains the following data.

Table 45. NMsec_GET_PORTTRAN structure

Field	Offset	Length	Format	Description
NMsPortTransRemoteAddr	0	4 bytes	Binary	IPv4 public remote address for peer
NMsPortTransRemoteInner	4	4 bytes	Binary	IPv4 private remote address for peer
NMsPortTransProtocol	8	1 byte	Binary	Protocol for port translation entry, either IPPROTO_TCP or IPPROTO_UDP
NMsPortTransRsvd1	9	24 bits	Binary	Reserved bits
NMsPortTransOrigPort	12	2 bytes	Binary	Original remote port for connection
NMsPortTransNewPort	14	2 bytes	Binary	Translated remote port; the port by which the connection is now known to this TCP/IP stack

NMsec_GET_IPTUNMANUAL

NMsec_GET_IPTUNMANUAL response format

NMsecMessageHdr	Response records (1 for each tunnel)			
	NMsecRecordHdr	NMsecSecDesc (2)	NMsecIPTunnel	NMsecIPManualTunnel

Figure 22. NMsec_GET_IPTUNMANUAL response format

For the requested stack, zero or more records are returned representing manual IP tunnels. Tunnels are presented in an unordered sequence. Each record returned contains two sections.

One section, NMsecIPTunnel, describes the basic properties of an IP tunnel. This section contains the following data.

Note: This structure is reused for dynamic tunnels, so some possible field values are applicable only to dynamic tunnels.

Table 46. NMsecIPTunnel structure

Field	Offset	Length	Format	Description
NMsIPTunID	0	48 bytes	EBCDIC	Tunnel ID
NMsIPTunVPNAction	48	48 bytes	EBCDIC	Tunnel VPN action name
NMsIPTunFlagIPv6	96, bit 0	1 bit	Binary	IPv6 indicator. If set, security endpoint and data endpoint addresses are IPv6; otherwise they are IPv4
NMsIPTunRsvd1	96, bit 1	31 bits	Binary	Reserved bits.
NMsIPTunType	100	1 byte	Binary	Tunnel type. One of the following: NMsec_IPTUN_MANUAL (1) Manual IP tunnel NMsec_IPTUN_STACK (2) Dynamic IP tunnel, as known to the TCP/IP stack NMsec_IPTUN_IKE (3) Dynamic IP tunnel, as known to IKE
NMsIPTunState	101	1 byte	Binary	Tunnel state. One of the following: NMsec_SASTATE_INACTIVE (1) Manual tunnel inactive NMsec_SASTATE_PENDING (2) Dynamic tunnel is awaiting negotiation NMsec_SASTATE_INCOMPLETE (3) Dynamic tunnel is in negotiation NMsec_SASTATE_ACTIVE (4) Manual or dynamic tunnel is active NMsec_SASTATE_EXPIRED (5) Dynamic tunnel is expired
NMsIPTunRsvd2	102	2 bytes	Binary	Reserved
NMsIPTunLclEndpt4	104	4 bytes	Binary	If this is an IPv4 tunnel, this field is the local security endpoint address

Table 46. NMsecIPTunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIPTunLclEndpt6	104	16 bytes	Binary	If this is an IPv6 tunnel, this field is the local security endpoint address
NMsIPTunRmtEndpt4	120	4 bytes	Binary	If this is an IPv4 tunnel, this field is the remote security endpoint address
NMsIPTunRmtEndpt6	120	16 bytes	Binary	If this is an IPv6 tunnel, this field is the remote security endpoint address
NMsIPTunEncapMode	136	1 byte	Binary	Tunnel encapsulation mode. One of the following: NMsec_IPTUN_TUNNELMODE (1) NMsec_IPTUN_TRANSPORTMODE (2) This field is not defined if the tunnel state is NMsec_SASTATE_PENDING or NMsec_SASTATE_INCOMPLETE.
NMsIPTunAuthProto	137	1 byte	Binary	Tunnel authentication protocol. One of the following: IPPROTO_AH (51) IPPROTO_ESP (50) This field is not defined if the tunnel state is NMsec_SASTATE_PENDING or NMsec_SASTATE_INCOMPLETE.
NMsIPTunAuthAlg	138	1 byte	Binary	Tunnel authentication algorithm. One of the following: NMsec_AUTH_HMAC_MD5 (38) NMsec_AUTH_HMAC_SHA1 (39) This field is not defined if the tunnel state is NMsec_SASTATE_PENDING or NMsec_SASTATE_INCOMPLETE.
NMsIPTunEncryptAlg	139	1 byte	Binary	Tunnel encryption algorithm. One of the following: NMsec_ENCR_NONE (0) NMsec_ENCR_NULL (11) NMsec_ENCR_DES (18) NMsec_ENCR_3DES (3) NMsec_ENCR_AES (12) This field is not defined if the tunnel state is NMsec_SASTATE_PENDING or NMsec_SASTATE_INCOMPLETE
NMsIPTunInbAuthSPI	140	4 bytes	Binary	Tunnel inbound authentication SPI. This field is not defined if the tunnel state is NMsec_SASTATE_PENDING or NMsec_SASTATE_INCOMPLETE.
NMsIPTunOutbAuthSPI	144	4 bytes	Binary	Tunnel outbound authentication SPI. This field is not defined if the tunnel state is NMsec_SASTATE_PENDING or NMsec_SASTATE_INCOMPLETE.
NMsIPTunInbEncryptSPI	148	4 bytes	Binary	Tunnel inbound encryption SPI. This field is not defined if the tunnel state is NMsec_SASTATE_PENDING or NMsec_SASTATE_INCOMPLETE.

Table 46. NMsecIPTunnel structure (continued)

Field	Offset	Length	Format	Description
NMsecIPTunOutEncryptSPI	152	4 bytes	Binary	Tunnel outbound encryption SPI. This field is not defined if the tunnel state is NMsec_SASTATE_PENDING or NMsec_SASTATE_INCOMPLETE.

One section, NMsecIPManualTunnel, describes the attributes specific to a manual IP tunnel. This section contains the following data.

Table 47. NMsecIPManualTunnel structure

Field	Offset	Length	Format	Description
NMsecIPManTunOutPkt	0	8 bytes	Binary	Outbound packet count for this tunnel
NMsecIPManTunInPkt	8	8 bytes	Binary	Inbound packet count for this tunnel
NMsecIPManTunOutBytes	16	8 bytes	Binary	Outbound byte count for this tunnel, representing the number of outbound data bytes protected by the tunnel
NMsecIPManTunInBytes	24	8 bytes	Binary	Inbound byte count for this tunnel, representing the number of inbound data bytes protected by the tunnel

NMsec_GET_IPTUNDYNSTACK

NMsec_GET_IPTUNDYNSTACK response format

NMsecMessageHdr	Response records (1 for each tunnel)				
	NMsecRecordHdr	NMsecSecDesc (3)	NMsecIPTunnel	NMsecIPDynTunnel	NMsecIPDynamicStack

Figure 23. NMsec_GET_IPTUNDYNSTACK response format

For the requested stack, zero or more records are returned representing dynamic IP tunnels known to the TCP/IP stack. Depending on the input filters provided on the request, the tunnels can include SWSA shadow tunnels. SWSA shadow tunnels originate from a distributing stack and not from the local stack. Tunnels are presented in an unordered sequence, except that instances of a particular tunnel family (all sharing the same tunnel ID) are ordered from most recently activated to least recently activated.

Each record contains the following sections:

One section, NMsecIPTunnel, describes the basic properties of an IP tunnel. The layout of this section is described in Table 46 on page 502.

One section, NMsecIPDynTunnel, describes the basic properties of a dynamic IP tunnel. This section contains the following data.

Table 48. NMsecIPDynTunnel structure

Field	Offset	Length	Format	Description
NMsecIPDynUDPEncap	0, bit 0	1 bit	Binary	UDP encapsulation indicator. If set, the tunnel uses UDP encapsulation mode.
NMsecIPDynLclNAT	0, bit 1	1 bit	Binary	Local NAT indicator. If set, a NAT has been detected in front of the local security endpoint.

Table 48. NMseclPDynTunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIPDynRmtNAT	0, bit 2	1 bit	Binary	Remote NAT indicator. If set, a NAT has been detected in front of the remote security endpoint.
NMsIPDynRmtNAPT	0, bit 3	1 bit	Binary	Remote NAPT indicator. If set, a NAPT has been detected in front of the remote security endpoint. It is possible that a NAPT might exist but might be detected only as a NAT.
NMsIPDynRmtGW	0, bit 4	1 bit	Binary	Remote NAT traversal gateway indicator. If set, the tunnel uses UDP encapsulation and the remote security endpoint is acting as an IPSec gateway.
NMsIPDynRmtZOS	0, bit 5	1 bit	Binary	Remote z/OS indicator. If set, the remote peer has been detected to be z/OS. It is possible that the remote peer might be running z/OS but not detected as such, if NAT traversal is not enabled.
NMsIPDynCanInitP2	0, bit 6	1 bit	Binary	Dynamic tunnel (phase 2) initiation indicator. If set, the local security endpoint can initiate dynamic tunnel negotiations with the remote security endpoint. Otherwise, the remote security endpoint must initiate dynamic tunnel negotiations. Either side can initiate refreshes.
NMsIPDynSrcIsSingle	0, bit 7	1 bit	Binary	Single source address indicator. If set, traffic source address is indicated by the NMsIPDynSrcAddr4 or NMsIPDynSrcAddr6 fields.
NMsIPDynSrcIsPrefix	1, bit 0	1 bit	Binary	Prefixed source address indicator. If set, traffic source address is indicated by the NMsIPDynSrcAddr4 or NMsIPDynSrcAddr6, fields and the source address prefix is indicated by the NMsIPDynSrcAddrPrefix field.
NMsIPDynSrcIsRange	1, bit 1	1 bit	Binary	Ranged source address indicator. If set, traffic source address range is indicated by the NMsIPDynSrcAddr4 and NMsIPDynSrcAddrRange4 fields, or by the NMsIPDynSrcAddr6 and NMsIPDynSrcAddrRange6 fields.
NMsIPDynDstIsSingle	1, bit 2	1 bit	Binary	Single destination address indicator. If set, traffic destination address is indicated by the NMsIPDynDstAddr4 or NMsIPDynDstAddr6 fields.
NMsIPDynDstIsPrefix	1, bit 3	1 bit	Binary	Prefixed destination address indicator. If set, traffic destination address is indicated by the NMsIPDynDstAddr4 or NMsIPDynDstAddr6 fields, and destination address prefix is indicated by the NMsIPDynDstAddrPrefix field.
NMsIPDynDstIsRange	1, bit 4	1 bit	Binary	Ranged destination address indicator. If set, traffic destination address range is indicated by the NMsIPDynDstAddr4 and NMsIPDynDstAddrRange4 fields, or by the NMsIPDynDstAddr6 and NMsIPDynDstAddrRange6 fields.
NMsIPDynRsvd1	1, bit 5	19 bits	Binary	Reserved bits.

Table 48. NMsecIPDynTunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIPDynVPNRule	4	48 bytes	EBCDIC	Dynamic VPN rule name for this tunnel; otherwise, blank if there is no local dynamic VPN rule.
NMsIPDynP1TunnelID	52	48 bytes	EBCDIC	Tunnel ID for this tunnel's parent IKE (phase 1) tunnel. As a result of refreshes, this tunnel ID might represent multiple related IKE tunnels.
NMsIPDynLifesize	100	8 bytes	Binary	Tunnel lifesize. If not 0, indicates the negotiated lifesize value limit for the tunnel, in bytes.
NMsIPDynLifesizeRefresh	108	8 bytes	Binary	Tunnel lifesize refresh. If not 0, indicates the lifesize value at which the tunnel is refreshed, in bytes.
NMsIPDynLifetimeExpire	116	4 bytes	Binary	Tunnel lifetime. Indicates the negotiated time at which the tunnel expires, in UNIX format.
NMsIPDynLifetimeRefresh	120	4 bytes	Binary	Tunnel lifetime refresh. Indicates the time at which the tunnel is refreshed, in UNIX format.
NMsIPDynVPNLifeExpire	124	4 bytes	Binary	Tunnel VPN lifetime expire. If not 0, indicates the time at which the tunnel family ceases to be refreshed, in UNIX format.
NMsIPDynActMethod	128	1 byte	Binary	Tunnel activation method. One of the following: NMsec_DYNTUN_USER (1) User activation (from the command line). NMsec_DYNTUN_REMOTE (2) Remote activation from IPsec peer. NMsec_DYNTUN_ONDEMAND (3) On-demand activation caused by IP traffic. NMsec_DYNTUN_TAKEOVER (5) SWSA activation as a result of a DVIPA takeover. NMsec_DYNTUN_AUTOACT (6) Auto-activation.
NMsIPDynRsvd2	129	24 bits	Binary	Reserved bits.
NMsIPDynRmtUDPPort	132	2 bytes	Binary	If the tunnel uses UDP-encapsulation mode, the IKE UDP port of the remote security endpoint; otherwise, 0.
NMsIPDynRsvd3	134	2 bytes	Binary	Reserved bits.

Table 48. NMsecIPDynTunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIPDynSrcNATOA	136	4 bytes	Binary	Source NAT-OA payload. NAT-OA payloads are exchanged only for certain UDP-encapsulated tunnels. During NAT traversal negotiations, the IKE peer sends the source IP address that it is aware of. If NAT traversal negotiation did not occur or if a peer did not send a source NAT-OA payload, the value of this field is 0. Restriction: An IKE peer at a NAT traversal support level that is prior to RFC3947 is not required to send a source NAT-OA payload.
NMsIPDynDstNATOA	140	4 bytes	Binary	Destination NAT-OA payload. NAT-OA payloads are exchanged only for certain UDP-encapsulated tunnels. During NAT traversal negotiations, the IKE peer sends the destination IP address that it is aware of. If NAT traversal negotiation did not occur or if a peer did not send a destination NAT-OA payload, the value of this field is 0. Restriction: An IKE peer at a NAT traversal support level that is prior to RFC3947 will not send a destination NAT-OA payload.
NMsIPDynProtocol	144	1 byte	Binary	Protocol for tunnel data. If the value is 0, the tunnel covers all protocols.
NMsIPDynRsvd4	145	24 bits	Binary	Reserved bits.
NMsIPDynSrcPort	148	2 bytes	Binary	Source port for tunnel data. If the value is 0, the tunnel covers all source ports.
NMsIPDynDstPort	150	2 bytes	Binary	Destination port for tunnel data. If the value is 0, the tunnel covers all destination ports.
NMsIPDynSrcAddr4	152	4 bytes	Binary	• If the NMsIPDynSrcIsSingle field is set, this field is the IPv4 or IPv6 source address for tunnel data • If the NMsIPDynSrcIsPrefix field is set, this field is the IPv4 or IPv6 source address base for tunnel data • If the NMsIPDynSrcIsRange field is set, this field is the low end of the IPv4 or IPv6 source address range for tunnel data
NMsIPDynSrcAddr6	156	16 bytes	Binary	
NMsIPDynSrcAddrRange4	168	4 bytes	Binary	If the NMsIPDynSrcIsRange field is set, this field is the high end of the IPv4 or IPv6 source address range for tunnel data.
NMsIPDynSrcAddrRange6	168	16 bytes	Binary	

Table 48. NMsecIPDynTunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIPDynDstAddr4	184	4 bytes	Binary	- If the NMsIPDynDstIsSingle field is set, this field is the IPv4 or IPv6 destination address for tunnel data - If the NMsIPDynDstIsPrefix field is set, this field is the IPv4 or IPv6 destination address base for tunnel data - If the NMsIPDynDstIsRange field is set, this field is the low end of the IPv4 or IPv6 destination address range for tunnel data
NMsIPDynDstAddr6	184	16 bytes	Binary	
NMsIPDynDstAddrRange4	200	4 bytes	Binary	If the NMsIPDynDstIsRange field is set, this field is the high end of the IPv4 or IPv6 destination address range for tunnel data.
NMsIPDynDstAddrRange6	200	16 bytes	Binary	
NMsIPDynSrcAddrPrefix	216	1 byte	Binary	If the NMsIPDynSrcIsPrefix field is set, this field is the length of the tunnel data source address prefix in bits.
NMsIPDynDstAddrPrefix	217	1 byte	Binary	If the NMsIPDynDstIsPrefix field is set, this field is the length of the tunnel data destination address prefix in bits.
NMsIPDynRsvd5	218	2 bytes	Binary	Reserved bits.

One section, NMsecIPDynamicStack, describes the properties of a dynamic IP tunnel that are specific to the TCP/IP stack. This section contains the following data.

Table 49. NMsecIPDynamicStack structure

Field	Offset	Length	Format	Description
NMsIPDynStackShadow	0, bit 0	1 bit	Binary	SWSA shadow indicator. If set, the tunnel is an SWSA shadow tunnel originating from a distributing stack.
NmsIPDynStackRsvd1	0, bit 1	31 bits	Binary	Reserved bits.
NMsIPDynStackLifesizeCur	4	8 bytes	Binary	Current lifesize value. If the tunnel lifesize value has been negotiated, this represents the current value of the lifesize counter.
NMsIPDynStackOutPkt	12	8 bytes	Binary	Outbound packet count for this tunnel. For SWSA tunnels, this represents this tunnel's outbound packet count only for this particular TCP/IP stack.
NMsIPDynStackInPkt	20	8 bytes	Binary	Inbound packet count for this tunnel. For SWSA tunnels, this represents this tunnel's inbound packet count only for this particular TCP/IP stack.
NMsIPDynStackOutBytes	28	8 bytes	Binary	Outbound byte count for this tunnel, representing the number of outbound data bytes protected by the tunnel. For SWSA tunnels, this represents this tunnel's outbound byte count only for this particular TCP/IP stack.
NMsIPDynStackInBytes	36	8 bytes	Binary	Inbound byte count for this tunnel, representing the number of inbound data bytes protected by the tunnel. For SWSA tunnels, this represents this tunnel's inbound byte count only for this particular TCP/IP stack.

NMsec_GET_IPTUNDYNIKE

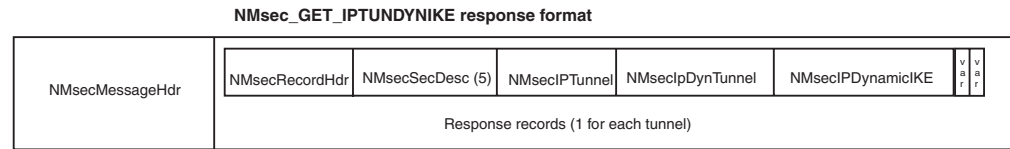


Figure 24. NMsec_GET_IPTUNDYNIKE response format

For the requested stack, zero or more records are returned representing dynamic IP tunnels known to the IKE daemon. Tunnels are presented in an unordered sequence, except that instances of a particular tunnel family (all sharing the same tunnel ID) are ordered from most recently activated to least recently activated. Each record contains the following sections:

One section, NMsecIPTunnel, describes the basic properties of an IP tunnel. The layout of this section is described in Table 46 on page 502.

One section, NMsecIPDynTunnel, describes the basic properties of a dynamic IP tunnel. The layout of this section is described in Table 48 on page 504.

One section, NMsecIPDynamicIKE, describes the properties of a dynamic IP tunnel that are specific to the IKE daemon. This section contains the following data.

Table 50. NMsecIPDynamicIKE structure

Field	Offset	Length	Format	Description
NMsecIPDynamicIKEIsPendingNew	0, bit 0	1 bit	Binary	Pending new activation indicator. If set, this dynamic IP tunnel is in pending state and it represents a new activation rather than a refresh. If not set, the tunnel is either not in pending state or is not a new activation.
NMsecIPDynamicIKERsvd1	0, bit 1	31 bits	Binary	Reserved bits.
NMsecIPDynamicIKEFilter	4	48 bytes	EBCDIC	Filter name for the IP filter related to this dynamic tunnel.
NMsecIPDynamicIKEDHGroup	52	4 bytes	Binary	Diffie-Hellman group used for perfect forward secrecy (PFS) for this dynamic tunnel, or 0 if phase 2 PFS is not configured.
NMsecIPDynamicIKELclIDType	56	1 byte	Binary	ISAKMP identity type for the local client ID, as defined in RFC 2407. Client identities can be exchanged during negotiation to limit or define the scope of data protected by the tunnel. If client identities are not exchanged, then the scope of data protection is defined to cover the peers' tunnel endpoint addresses. If client identities were not exchanged during negotiation, this field is 0.
NMsecIPDynamicIKERmtIDType	57	1 byte	Binary	ISAKMP identity type for the remote client ID, as defined in RFC 2407. Client identities might be exchanged during negotiation to limit or define the scope of data protected by the tunnel. If client identities are not exchanged, then the scope of data protection is defined to cover the peers' tunnel endpoint addresses. If client identities were not exchanged during negotiation, this field is 0.

Table 50. NMsecIPDynamicIKE structure (continued)

Field	Offset	Length	Format	Description
NMslPDynIKEExtState	58	2 bytes	Binary	Extended tunnel state information. One of the following: NMsec_P2STATE_INIT (0) No key exchange messages have been initiated. NMsec_P2STATE_IN_KEP (1) Key exchange messages are being processed, but the full exchange has not completed. NMsec_P2STATE_DONE (2) All key exchange messages have been completed and the tunnel is usable for traffic. NMsec_P2STATE_PENDING_NOTIFY (3) Key exchange messages have been completed, but until a connection notification is received from the tunnel endpoint, the tunnel is not done. NMsec_P2STATE_PENDING_START (4) Tunnel is awaiting the activation of an IKE tunnel to allow it to begin. See the description of the NMslPTunState field in Table 46 on page 502 for more succinct state information.

One variable-length section containing the local client ID for this tunnel's phase 2 negotiation. Regardless of the identity's type, the ID is expressed as an EBCDIC string (an IP address is returned in printable form). The length of this section is 0 if no client IDs were exchanged.

One variable-length section containing the remote client ID for this tunnel's phase 2 negotiation. Regardless of the identity's type, it is expressed as an EBCDIC string (an IP address is returned in printable form). The length of this section is 0 if no client IDs were exchanged.

NMsec_GET_IKETUN

NMsec_GET_IKETUN response format

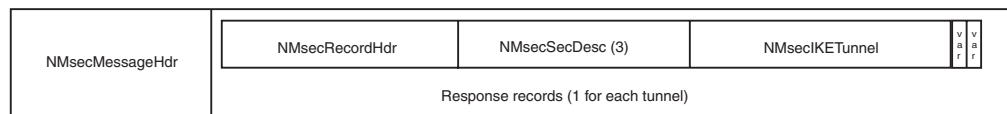


Figure 25. NMsec_GET_IKETUN response format

For the requested stack, zero or more records are returned representing IKE security associations (IKE tunnels) used by IKE to negotiate IPSec security associations (dynamic tunnels) for the given TCP/IP stack. Tunnels are presented in an unordered sequence, except that instances of a particular tunnel family (all sharing the same tunnel ID) are ordered from the most recently activated to the least recently activated. Each record contains the following sections:

One section, NMsecIKETunnel, describes attributes of the IKE security association. This section contains the following data.

Table 51. NMsecIKETunnel structure

Field	Offset	Length	Format	Description
NMsecIKETunIPv6	0, bit 0	1 bit	Binary	IPv6 indicator. If set, the IKE tunnel security endpoints are IPv6 addresses, otherwise they are IPv4
NMsecIKETunNATAllowed	0, bit 1	1 bit	Binary	NAT traversal indicator. If set, the NAT traversal function is enabled for this IKE tunnel.
NMsecIKETunLclNAT	0, bit 2	1 bit	Binary	Local NAT indicator. If set, a NAT has been detected in front of the local security endpoint.
NMsecIKETunRmtNAT	0, bit 3	1 bit	Binary	Remote NAT indicator. If set, a NAT has been detected in front of the remote security endpoint.
NMsecIKETunRmtNAPT	0, bit 4	1 bit	Binary	Remote NAPT indicator. If set, an NAPT has been detected in front of the remote security endpoint. It is possible that an NAPT might exist but that it is detected only as a NAT.
NMsecIKETunCanInitP1	0, bit 5	1 bit	Binary	IKE tunnel (P1) initiation indicator. If set, the local security endpoint can initiate IKE tunnel negotiations with the remote security endpoint. Otherwise, the remote security endpoint must initiate IKE tunnel negotiations. Either side can initiate refreshes.
NMsecIKETunRsvd1	0, bit 6	26 bits	Binary	Reserved bits.
NMsecIKETunID	4	48 bytes	EBCDIC	Tunnel ID for this IKE tunnel.
NMsecIKETunKeyExchRule	52	48 bytes	EBCDIC	Key exchange rule name for this IKE tunnel.
NMsecIKETunKeyExchAction	100	48 bytes	EBCDIC	Key exchange action name for this IKE tunnel.
NMsecIKETunLclEndpt4	148	4 bytes	Binary	IPv4 or IPv6 local security endpoint for this IKE tunnel.
NMsecIKETunLclEndpt6	148	16 bytes	Binary	
NMsecIKETunRmtEndpt4	164	4 bytes	Binary	IPv4 or IPv6 remote security endpoint for this IKE tunnel.
NMsecIKETunRmtEndpt6	164	16 bytes	Binary	
NMsecIKETunICookie	180	8 bytes	Binary	The icookie for this IKE tunnel.
NMsecIKETunRCookie	188	8 bytes	Binary	The rcookie for this IKE tunnel.
NMsecIKETunExchangeMode	196	1 byte	Binary	Tunnel exchange mode. One of the following: NMsec_IKETUN_EXCHMAIN (2) NMsec_IKETUN_EXCHAGGRESSIVE (4)
NMsecIKETunState	197	1 byte	Binary	Tunnel state. One of the following: NMsec_SASTATE_PENDING (2) Tunnel is awaiting negotiation. NMsec_SASTATE_INCOMPLETE (3) Tunnel is in negotiation. NMsec_SASTATE_ACTIVE (4) Tunnel is active. NMsec_SASTATE_EXPIRED (5) Tunnel is expired.
NMsecIKETunAuthAlg	198	1 byte	Binary	Tunnel authentication algorithm. One of the following: NMsec_AUTH_HMAC_MD5 (38) NMsec_AUTH_HMAC_SHA1 (39)
NMsecIKETunEncryptAlg	199	1 byte	Binary	Tunnel encryption algorithm. One of the following: NMsec_ENCR_DES (18) NMsec_ENCR_3DES (3) NMsec_ENCR_AES (12)
NMsecIKETunDHGroup	200	4 bytes	Binary	Diffie-Hellman group used to generate keying material for this IKE tunnel.
NMsecIKETunPeerAuthMethod	204	1 byte	Binary	Tunnel peer authentication method. One of the following: NMsec_IKETUN_PRESHAREDKEY (3) NMsec_IKETUN_RSASIGNATURE (2)

Table 51. NMsecIKETunnel structure (continued)

Field	Offset	Length	Format	Description
NMsiKETunRole	205	1 byte	Binary	Tunnel role. One of the following: NMsec_IKETUN_INITIATOR (1) NMsec_IKETUN_RESPONDER (2)
NMsiKETunNATTLevel	206	1 byte	Binary	NAT traversal support level. One of the following: NMsec_IKETUN_NATTNONE (0) No NAT traversal support; either not configured or not negotiated. NMsec_IKETUN_NATTRFCD2 (1) RFC 3947 draft 2 support. NMsec_IKETUN_NATTRFCD3 (3) RFC 3947 draft 3 support. NMsec_IKETUN_NATTRFC (4) RFC 3947 support with non-z/OS peer. NMsec_IKETUN_NATTZOS (5) RFC 3947 support with z/OS peer.
NMsiKETunExtState	207	1 byte	Binary	Extended tunnel state information. One of the following: NMsec_P1STATE_INIT (0) No key exchange messages have been initiated. NMsec_P1STATE_WAIT_SA (1) The first key exchange message has been sent and the endpoint is waiting for a response. NMsec_P1STATE_IN_KE (2) A key exchange response has been sent. NMsec_P1STATE_WAIT_KE (3) A key exchange message has been sent and the endpoint is waiting for a response. NMsec_P1STATE_DONE (4) All key exchange messages have been completed and the tunnel is available for data traffic. NMsec_P1STATE_EXPIRED (5) Tunnel has exceeded its lifetime or lifesize and is not available for data traffic. See NMsiKETunState for more succinct state information.
NMsiKETunLifesize	208	8 bytes	Binary	Tunnel lifesize. If not 0, indicates the negotiated lifesize limit for the tunnel, in bytes.
NMsiKETunLifetime	216	4 bytes	Binary	Negotiated tunnel lifetime. Indicates the total number of seconds the tunnel remains active.
NMsiKETunLifetimeRefresh	220	4 bytes	Binary	Tunnel lifetime refresh. Indicates the time at which the tunnel is refreshed, in UNIX format.
NMsiKETunLifetimeExpire	224	4 bytes	Binary	Tunnel lifesize expiration. Indicates the time at which the tunnel expires, in UNIX format.
NMsiKETunRmtUDPPort	228	2 bytes	Binary	Remote UDP port used for IKE negotiations.
NMsiKETunLIDType	230	1 byte	Binary	ISAKMP identity type for the local security endpoint identity, as defined in RFC 2407. ISAKMP peers exchange and verify each others' identities as part of the IKE tunnel (phase 1) negotiation.

Table 51. NMsecIKETunnel structure (continued)

Field	Offset	Length	Format	Description
NMsecIKETunRIDType	231	1 byte	Binary	ISAKMP identity type for the remote security endpoint identity, as defined in RFC 2407. ISAKMP peers exchange and verify each others' identities as part of the IKE tunnel (phase 1) negotiation.

One section, NMsecIKETunStats, indicates various counters and statistics for the IKE tunnel. This section contains the following data.

Table 52. IKE tunnel statistics

Field	Offset	Length	Format	Description
NMsecIKETunP2Current	0	4 bytes	Binary	Current count of active dynamic tunnels that are associated with this IKE tunnel.
NMsecIKETunP2InProgress	4	4 bytes	Binary	Current count of pending or in-progress dynamic tunnels that are associated with this IKE tunnel.
NMsecIKETunP2LclActSuccess	8	4 bytes	Binary	Cumulative count of successful dynamic tunnel activations that were initiated locally for this IKE tunnel.
NMsecIKETunP2RmtActSuccess	12	4 bytes	Binary	Cumulative count of successful dynamic tunnel activations that were initiated remotely for this IKE tunnel.
NMsecIKETunP2LclActFailure	16	4 bytes	Binary	Cumulative count of failed dynamic tunnel activations that were initiated locally for this IKE tunnel.
NMsecIKETunP2RmtActFailure	20	4 bytes	Binary	Cumulative count of failed dynamic tunnel activations that were initiated remotely for this IKE tunnel.
NMsecIKETunBytes	24	8 bytes	Binary	Cumulative number of bytes that were protected by this IKE tunnel.

One variable-length section contains the contents of the local identity used to negotiate the IKE tunnel. Regardless of the identity's type, it is expressed as an EBCDIC string (an IP address is returned in printable form).

One variable-length section contains the contents of the remote identity used to negotiate the IKE tunnel. Regardless of the identity's type, it is expressed as an EBCDIC string (an IP address is returned in printable form).

NMsec_GET_IKETUNCASCADE

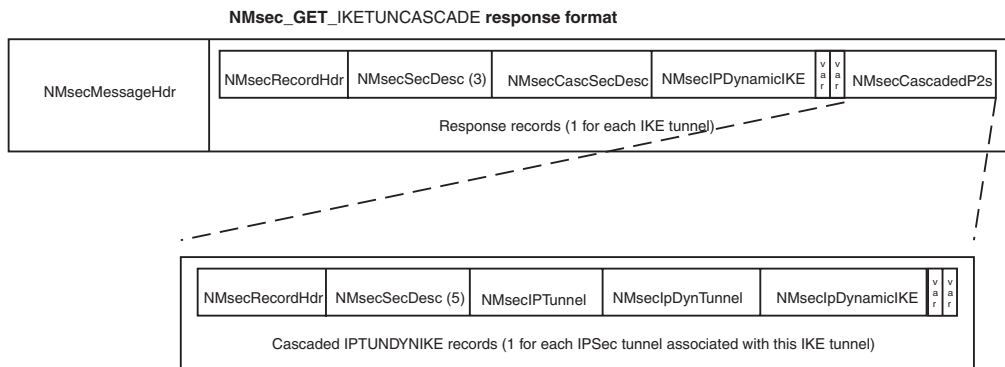


Figure 26. NMsec_GET_IKETUNCASCADE response format

For the requested stack, zero or more records are returned representing IKE security associations (phase 1 tunnels) used by IKE to negotiate IPsec security associations (phase 2 tunnels) for the given TCP/IP stack. The results are similar to the NMsec_GET_IKETUN request, except that cascaded phase 2 tunnel information is also included in the response. Each phase 2 IP tunnel associated with the given phase 1 IKE tunnel is reported in the result record. Each record contains the following sections:

- One section, NMsecIKETunnel, describes attributes of the IKE SA. The layout of this section is described in “NMsec_GET_IKETUN” on page 510.
- One section, NMsecIKETunStats, describes various counters and statistics for the IKE tunnel. The layout of this section is described in Table 52 on page 513.
- One variable-length section contains the contents of the local identity used to negotiate the IKE tunnel.
- One variable-length section contains the contents of the remote identity used to negotiate the IKE tunnel.
- One or zero cascaded record containers with a set of dynamic IPsec tunnel records, identified by a single cascading record descriptor in the record header. The records in this section describe the basic tunnel properties of each IPsec security association associated with this IKE tunnel. The format of these cascaded records is described in “NMsec_GET_IPTUNDYNIKE” on page 509.

NMsec_GET_IPINTERFACES

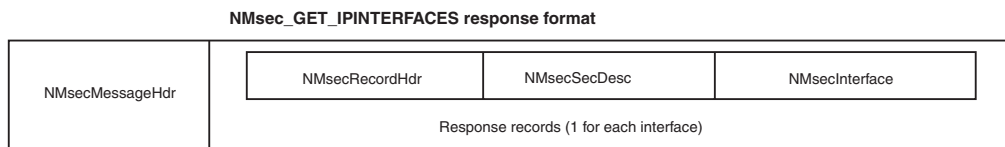


Figure 27. NMsec_GET_IPINTERFACES response format

Each record returned identifies an IP interface that the TCP/IP stack serves. Each record has a single section, NMsecInterface, that describes attributes of the IP interface. This section contains the following data.

Table 53. NMsecInterface structure

Field	Offset	Length	Format	Description
NMsecInterfaceName	0	16 bytes	EBCDIC	Interface name

Table 53. NMsecInterface structure (continued)

Field	Offset	Length	Format	Description
NMsInterfaceAddr	16	16 bytes	Binary	IP address. If this is an IPv4 address, the last 4 bytes contains the address and the first 12 bytes contain zeroes.
NMsInterfaceSecClass	32	1 byte	Binary	Security class
NMsIPv6	33, bit 1	1 bit	Binary	IP addressing mode. If set to 1, the interface is using an IPv6 address; otherwise, it is using IPv4.
NMsInterfaceActive	33, bit 2	1 bit	Binary	State indicator. If set to 1, the interface is active.
NMsInterfaceDVIPA	33, bit 3	1 bit	Binary	DVIPA indicator. If set to 1, the interface is a DVIPA.
NMsInterfaceRsvd1	33, bit 4	21 bits	Binary	Reserved. Must be set to 0.

NMsec_GET_IKENSINFO

NMsec_GET_IKENSINFO response format

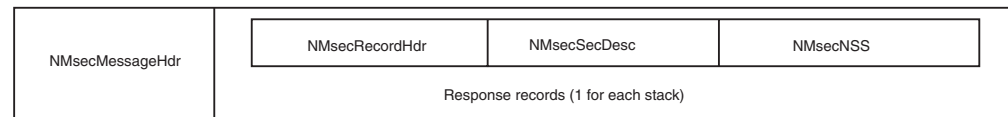


Figure 28. NMsec_GET_IKENSINFO response format

Each record returned describes the network security services (NSS) attributes for a given stack. Each record has a single section, NMsecNSS, that describes the NSS attributes of the stack. This section contains the following data described in Table 54.

Table 54. NMsec_GET_IKENSINFO structure

Field	Offset	Length	Format	Description
NMsIKENSClientName	0	24 bytes	EBCDIC	The stack's NSS client name. The default format is <i>sysname_stackname</i> , where <i>sysname</i> is the client's MVS system name, and <i>stackname</i> is the TCP/IP job name of the stack that the client represents. However, clients can override this default form with any valid 24-character string. If the client is not configured to use an NSS server, this field contains blanks.
NMsIKENSStackName	24	8 bytes	EBCDIC	TCP/IP job name of the stack that the client represents.
NMsIKENSUserid	32	8 bytes	EBCDIC	User ID under which the client is registered with server. If the client is not configured to use an NSS server, this field contains blanks.
NMsIKENSServerSysName	40	8 bytes	EBCDIC	MVS system name of the system on which the NSS server is running. If the client is not configured to use an NSS server, this field contains blanks.
NMsIKENSSConfigured	48, bit 0	1 bit	Binary	Indicates whether this stack is configured to use an NSS server. If the value is set to 1, the stack is configured to use a NS server; otherwise, it is not.
NMsIKENSSvcSelCert	48, bit 1	1 bit	Binary	Certificate services selected. If set to 1, the stack is configured for NSS certificate services.
NMsIKENSSvcEnblCert	48, bit 2	1 bit	Binary	Certificate services enabled. If set to 1, the stack is actively using NSS certificate services.
NMsIKENSSvcSelMgmt	48, bit 3	1 bit	Binary	Remote management services selected. If set to 1, the stack is configured for NSS remote management services.
NMsIKENSSvcEnblMgmt	48, bit 4	1 bit	Binary	Remote management services enabled. If set to 1, the stack is actively using NSS remote management services.
NMsIKENSIPv6	48, bit 5	1 bit	Binary	IP addressing mode. If set to 1, the client is using an IPv6 address. Otherwise, it is using IPv4. If the client is not configured to use an NSS server, this field is set to 0.

Table 54. NMsec_GET_IKENSINFO structure (continued)

Field	Offset	Length	Format	Description
NMsiKENSServerConnectState	50	2 bytes	Binary	NS server connection state. One of the following: NMsec_SERVERSTATUS_DISCONNECTED (0) The stack is disconnected from the NSS server. NMsec_SERVERSTATUS_CONNECTPENDING (1) Connection is pending. The stack has requested a connection to the NSS server but the request has not completed processing. NMsec_SERVERSTATUS_CONNECTED (2) The stack is connected to the NS server. NMsec_SERVERSTATUS_DISCONNECTPENDING (3) Disconnect is pending. The stack has requested that the connection be disconnected but the request has not completed processing. NMsec_SERVERSTATUS_UPDATEPENDING (4) Update is pending. The stack has dynamically reconfigured its authentication information or its requested NSS. The stack has requested a connection update but has not received a successful response from the NSS server.
NMsiIKENSClientAddr	52	16 bytes	Binary	The IPv4 or IPv6 source address of the client connection to the server. If this is an IPv4 address, the destination address is the last 4 bytes of this field, with the first 12 bytes containing zeroes.
NMsiIKENSServerAddr	68	16 bytes	Binary	The IPv4 or IPv6 destination address of the client connection to the server. If this is an IPv4 address, the destination address is the last 4 bytes of this field, with the first 12 bytes containing zeroes.
NMsiIKENSClientPort	84	2 bytes	Binary	The TCP source port of the client connection to the server.
NMsiIKENSServerPort	86	2 bytes	Binary	The TCP destination port of the client connection to the server.
NMsiIKENSConnTime	88	4 bytes	Binary	UNIX-format timestamp indicating when the client connected to the server.
NMsiIKENSLastMsgTime	92	4 bytes	Binary	UNIX-format timestamp indicating when the last message was sent to the server.
NMsiIKENSNumFailedReqs	96	8 bytes	Binary	Number of failed requests to server.

NMsec_LOAD_POLICY

This control request does not contain response records. Rather, the return code and reason code fields in the response message header contain the final status of the request. Because this control request causes IKED to manipulate the file system, a z/OS UNIX System Services I/O error might occur, which causes the return code to contain the error number value EIO. In this case, the reason code contains the error number value of the error condition. For further error diagnosis see *z/OS UNIX System Services Messages and Codes*.

NMsec_ACTIVATE_IPTUNMANUAL, NMsec_ACTIVATE_IPTUNDYN, NMsec_DEACTIVATE_IPTUNMANUAL, NMsec_DEACTIVATE_IPTUNDYN, NMsec_DEACTIVATE_IKETUN, NMsec_REFRESH_IPTUNDYN, NMsec_REFRESH_IKETUN

One record is returned that indicates the response of the tunnel action request. A successful response from a tunnel control request indicates that the requested operation has been successfully initiated. Because IPsec tunnel activation, deactivation, and refresh requires an exchange of messages between IPsec peers, the final status of the operation can be determined later through a subsequent NMI request that returns the filter or tunnel data.

Tunnel Control response format

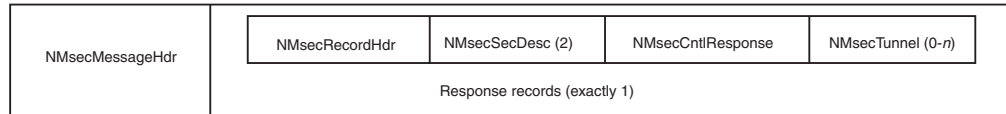


Figure 29. Tunnel control response format

For a request to activate or deactivate all tunnels of a given type, the response record contains the NMsecTunCntlResponse section, which is described in Table 55. For requests that indicate a specific tunnel, this section does not exist.

Table 55. NMsecTunCntlResponse structure

Field	Offset	Length	Format	Description
NMsTunCRCCount	0	4 bytes	Binary	The number of tunnels processed. When the request is for all tunnels, this value is the number of the tunnels found.

When specific tunnels are requested, the response record contains one NMsecTunnel section for each of the tunnels. The sections are returned in the same order as specified in the request. For requests that operate on all tunnels of a given type, this section does not exist.

Initialization and termination messages

When a client successfully connects to the interface, the server sends an initialization message (a message with type NMsec_INIT) to the client. This message contains no records, but the return code and reason code are 0 to indicate successful connection completion. When the server closes the connection (this might be the result of error, IKED termination, or the client's closing of the socket), the server attempts to send a termination message (a message with type NMsec_TERM) to the client. This message contains no records, but the return code and reason code indicate the cause of the connection's termination.

Return and reason codes

When sending a request, the client application should set the message header fields NMsMRc (return code) and NMsMRsn (reason code) to 0. Upon return, the server sets these fields as follows to indicate the status of the request. This service uses the errno values defined by z/OS UNIX System Services.

Table 56. Return and reason codes

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
0	0	No error
ENXIO	0	Requested TCP/IP stack does not exist or is not active. System Action: Request is failed but connection remains open. Response: Send requests only for active TCP/IP stacks.

Table 56. Return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
EOPNOTSUPP	0	Requested TCP/IP stack is not configured for IP security. System Action: Request is failed but connection remains open. Response: Send requests only for TCP/IP stacks configured for IP security.
EINVAL	NMsRsnBadIdent (1)	Invalid message or record identifier supplied in message. System Action: Connection is closed. Response: Re-issue the connection and send a correctly formatted message.
EINVAL	NMsRsnBadVersion (2)	Invalid version supplied in message header. System Action: Request is failed but connection remains open. Response: Send a correctly formatted message.
EINVAL	NMsRsnBadType (3)	Unsupported or unknown message type supplied in message header. System Action: Request is failed but connection remains open. Response: Send a supported message type.
EINVAL	NMsRsnExcessiveSize (4)	Excessive message size. System Action: Connection is closed. Response: Re-issue the connection and send a correctly formatted message.
EINVAL	NMsRsnHdrSize (5)	Message header size is not valid. System Action: Request is failed but connection remains open. Response: Send a message with the header size field set to the correct value.
EINVAL	NMsRsnMsgSize (6)	Message size is not valid. For example, the message might be too short, or the message size might be greater than the sum of its parts. System Action: Connection is closed. Response: Send a correctly formatted message.
EINVAL	NMsRsnReservedNonzero (7)	Reserved data in message header, record header, or record data is a non-zero value. Reserved fields must be set to 0 for compatibility with future versions of the interface. System Action: Request is failed but connection remains open. Response: Send a message with reserved fields set to 0.

Table 56. Return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
EINVAL	NMsRsnRecordLength (8)	Unrecognized record length supplied in message. Length does not correspond to known record data. System Action: Request is failed but connection remains open. Response: Send a message with input filters of the correct length.
EINVAL	NMsRsnRecordCount (9)	Unsupported record count supplied in message. NMI requests currently support a maximum of twenty input filters. System Action: Request is failed but connection remains open. Response: Send a message with the correct number of input filters.
EINVAL	NMsRsnSectionLength (10)	Unrecognized section length supplied in record. Length does not correspond to known section data. System Action: Request is failed but connection remains open. Response: Send a message with correct input filters.
EINVAL	NMsRsnSectionCount (11)	Unrecognized section count supplied in record. NMI requests currently allow one section in an input filter record. System Action: Request is failed but connection remains open. Response: Send a message with correct input filters.
EINVAL	NMsRsnFilterSpec (12)	The input filter specification indicates a combination of filter values that is unsupported for the message's request type. System Action: Request is failed but connection remains open. Response: Send a message with a valid input filter specification for the message type.
EINVAL	NMsRsnFilterValue (13)	The input filter specification contains a value that is out of range. System Action: Request is failed but connection remains open. Response: Send a message with correct input filter values.
EINVAL	NMsRsnManTypeConflict (14)	Manual tunnel activation and deactivation requests for multiple tunnels must contain uniform tunnel specifications: either tunnel IDs or tunnel names. The request contained a mixture of tunnel names and tunnel IDs. System Action: Request is failed but connection remains open. Response: Separate manual tunnel names and tunnel IDs into different requests.

Table 56. Return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
EINVAL	NMsRsnPolicySource (15)	The policy source value in the policy load request is not valid. System Action: No action is required. Response: Send a message with a valid NMsecPolSrcSource value.
EACCES	0	Access denied to the requested resource. System Action: Request is failed but connection remains open. Administrator Response: Permit user to security resource.
ENOMEM	0	Insufficient storage available in the server to process the request. System Action: Request is failed but connection remains open. Response: Increase the REGION size for the IKE daemon, or send a message with a narrower set of input filters to limit the response.
ENOMEM	NMsRsnTooManyConns (1)	The NMI thread is already using its maximum number of 50 connections and cannot accept any more. System Action: Connection is not opened and the request is failed. Response: Try the request again later.
EIO	(z/OS UNIX System Services Errno)	A file system I/O error occurred. The reason code contains the value of the errno that describes the error. System Action: Request is failed but the connection remains open. Response: Diagnose the z/OS UNIX System Services Errno using <i>z/OS UNIX System Services Messages and Codes</i>.

Network security services NMI

z/OS Communications Server network security services (NSS) server provides an AF_UNIX socket interface through which network management applications can manage IP filtering and IPsec on remote NSS clients. This interface is available only through the NSS server and should be used by network management applications that monitor and control multiple systems through a single point of control. Applications can perform the following functions using this interface:

- Issue monitoring or control requests through the NSS server to specified NSS clients. The NSS server routes all monitoring and control requests (described in “Local IPsec NMI” on page 472) to NSS clients, with the exception of the NMsec_GET_STACKINFO and NMsec_GET_IKENSINFO requests. Routing occurs only if the NSS client is connected to the NSS server at the time the request is made.
- Request information about one or all of the NSS clients that are currently connected to the NSS server.

Overview

A client network management application requests information and initiates control operations by sending specific requests over an AF_UNIX stream socket connection to the NSS server. If necessary, the request is then redirected to the specified NSS client, which later responds with the requested data or the results of the requested operation. The response information is then returned to the application directly over the AF_UNIX connection. For most control requests, a successful response indicates that the operation was successfully initiated, but that it is still in progress. You can determine the final status of the control operation later by issuing a subsequent monitoring request for the effected object.

Configuring the interface

Access to the network security services (NSS) server's network management interface is controlled through RACF (or an equivalent external security manager product) resource definitions in the SERVAUTH class. Most of these resource names contain the NSS client's name. The client name is defined by the client. The default value of a client name takes the form *sysname_stackname*, where the *sysname* value is the MVS system name of the client, and the *stackname* value is the TCP/IP stack name that it represents. You can override the *clientname* value in the client's IKE daemon configuration file on the NssStackConfig statement or in the IBM Configuration Assistant NSS Perspective on the Client IPsec Settings tab.

Tip: When you override the *clientname* value, ensure that the name you define does not match the name of an existing NSS client on the NSS server system. If the names match, users with authority to manage IP security on that system also gain authority to remotely manage the NSS client, because the SERVAUTH resource names are identical.

The z/OS system administrator can restrict access to NSS network management interfaces as follows:

- Access to remote client monitoring functions (those that request information only from an NSS client through the NSS server) within this interface can be restricted by defining a RACF (or equivalent external security manager product) resource EZB.NETMGMT.*sysname.clientname*.IPSEC.DISPLAY in the SERVAUTH class (where *sysname* represents the MVS system name where the interface is being invoked, and *clientname* is the name of the NSS client).
- Access to the remote client control functions (those that take some action) is controlled through the EZB.NETMGMT.*sysname.clientname*.IPSEC.CONTROL resource (where *sysname* represents the MVS system name where the interface is being invoked, and *clientname* is the name of the NSS client).
- Access to NSS server monitoring functions (those that request information only about the server itself) is controlled through the resource EZB.NETMGMT.*sysname.sysname*.NSS.DISPLAY in the SERVAUTH class (where the *sysname* value represents the MVS system name where the interface is being invoked).

Requirement: For applications that use the interface, the MVS user ID must be permitted to the defined resource. Additionally, permitted client applications must have permission to enter the /var/sock directory and to write to the /var/sock/nss socket. Ensure that the NSSD OMVS user ID has write access to the /var/sock directory (or ensure that it has permission to create this directory).

Guideline: If you are developing a feature for a product to be used by other parties, include instructions in your documentation indicating that administrators must define and give appropriate permission to the given security resource to use that feature.

Connecting to the server

For an application to use this interface, it must connect to the AF_UNIX stream socket that is provided by the NSS server for this interface. The socket path name is /var/sock/nss. Either the Language Environment C/C++ API or the UNIX System Services BPX services can be used to create AF_UNIX sockets and connect to this service.

When an application connects to the socket, the server sends an initialization message to the client application. When the NSS server closes a client connection (reasons for doing so include severe errors in the format of data requests sent by the application to the server, or NSS server termination), the NSS server attempts to send a termination message to the client before closing the connection. Both the initialization and termination messages match those used by the IKE daemon (see “IPSec NMI request/response format” on page 474).

Network security services NMI request/response format

The NSS server supports a message format that is almost identical to that used by the IKE daemon for local IPSec monitoring and control (see “Local IPSec NMI” on page 472). Like the local monitoring/control interface, these messages are exchanged over an AF_UNIX socket using a request-response model.

The only difference between the NSS and IPSec NMI message format is that when an NMI message is sent to the NSS server, the NMsmTarget string in the message header identifies the remote NSS client to which the request is directed. Use the *clientname* field of the target NSS client in the NMsmTarget string, padded on the right with blanks. You can obtain the *clientname* values of each client connected to the NSS server by issuing the NMsec_GET_CLIENTINFO request. The NMsmTarget field can be set to blanks for an NMsec_GET_CLIENTINFO request. If this field is set to blanks for any other request, the request is rejected with an appropriate error code in the reply header.

Restriction: An NMI request is redirected to an NSS client only if that client has selected the remote management service and is enabled for that service.

Request format

The NSS server supports all of the request messages described for the IKE daemon except for the NMsec_GET_STACKINFO and NMsec_GET_IKENSINFO requests (see “Local IPSec NMI” on page 472). In addition, the NSS server also supports the NMsec_GET_CLIENTINFO request message.

The NMsec_GET_CLIENTINFO request obtains a list of NSS clients that are currently connected to the NSS server as well as summary information about each client. This message does not allow a filtering record. If the NMsmTarget field in the message header is blank, then information for all of the currently connected clients is returned. If a client name is specified in the NMsmTarget field, then information for only that client is returned as long as the client is connected. If the specified client is not connected, the request fails with an ENXIO return code.

Access to this function is controlled through the EZB.NETMGMT.sysname.sysname.NSS.DISPLAY resource definition in the SERVAUTH class.

Response format

The NSS server supports all of the response messages described for the IKE daemon in “Local IPsec NMI” on page 472 except for the NMsec_GET_STACKINFO and NMsec_GET_IKENSINFO responses. In addition, the NSS server also supports the NMsec_GET_CLIENTINFO response message:

NMsec_GET_CLIENTINFO

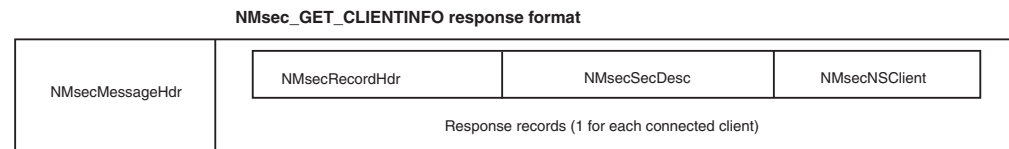


Figure 30. NMsec_GET_CLIENTINFO response format

Each record returned identifies and describes a single NSS client. Each record has a single section, NMsecNSClient, which contains the following data.

Table 57. NMsecNSClient structure

Field	Offset	Length	Format	Description
NMsNSClientName	0	24 bytes	EBCDIC	The name of the NSS client. The default format is <i>sysname_stackname</i> , where <i>sysname</i> is the MVS system name, and <i>stackname</i> is the TCP/IP job name. However, clients can override this default form with any valid 24-character string
NMsNSClientSysName	24	8 bytes	EBCDIC	The MVS system name of the system on which the client is running
NMsNSClientStackName	32	8 bytes	EBCDIC	The job name of the TCP/IP stack represented by the client
NMsNSClientUserid	40	8 bytes	EBCDIC	The user ID under which the client is registered
NMsNSClientClientAddress	48	16 bytes	Binary	The IPv4 or IPv6 address from which the client is connected. If this is an IPv4 address, it resides in the last 4 bytes of this field, with the first 12 containing zeroes
NMsNSClientServerAddress	64	16 bytes	Binary	The IPv4 or IPv6 address on which the NSS server is communicating with the client. If this is an IPv4 address, it resides in the last 4 bytes of this field, with the first 12 containing zeroes
NMsNSClientClientPort	80	2 bytes	Binary	The client's local TCP port over which the client is communicating with the server.
NMsNSClientServerPort	82	2 bytes	Binary	The TCP port over which the NSS server is communicating with the client
NMsNSClientSvcSelCert	84, bit 0	1 bit	Binary	Certificate services selected
NMsNSClientSvcEnblCert	84, bit 1	1 bit	Binary	Certificate services enabled
NMsNSClientSvcSelMgmt	84, bit 2	1 bit	Binary	Remote management services selected
NMsNSClientSvcEnblMgmt	84, bit 3	1 bit	Binary	Remote management services enabled
NMsNSClientIPv6	84, bit 4	1 bit	Binary	IP addressing mode. If set to 1, the client and server are using IPv6 addresses, otherwise, they are using IPv4
NMsNSClientRsvd1	84, bit 5	11 bits	Binary	Reserved

Table 57. NMsecNSClient structure (continued)

Field	Offset	Length	Format	Description
NMsNSClientConnectState	86	2 bytes	Binary	Client connection state. One of the following: NMsec_CLIENTSTATUS_CONNECTPENDING (1) Connection is pending. The initial TCP connection has been completed, but the ConnectClientReqToSrv request has not yet been successfully processed. The client's name and access rights cannot be determined until after the ConnectClientReqToSrv request has been processed. NMsec_CLIENTSTATUS_CONNECTED (2) The client is connected to the server. NMsec_CLIENTSTATUS_DISCONNECTPENDING (3) Disconnect is pending. The client is still in the NSS server's tables, but no more requests from that client are processed. The server is in the process of cleaning up after the client and removing the data from the server tables. The application enters this state under one of the following conditions: • The client's user ID authorization fails during the processing of a ConnectClientReqToSrv request • A DisconnectClientReqToSrv request is received from the client • The TCP/IP connection to the client was terminated or was lost. NMsec_CLIENTSTATUS_UPDATEPENDING (4) Update is pending. The client authorization information or selected services have been reconfigured at the client but the UpdateClientInfoReqToSrv request has not completed processing.
NMsNSClientConnTime	88	4 bytes	Binary	UNIX-format timestamp indicating when the client connected to the server
NMsNSClientLastMsgTime	92	4 bytes	Binary	UNIX-format timestamp indicating when the last message was received at the server from the client
NMsNSClientNumreqSigCreate	96	8 bytes	Binary	Number of create signature requests received from client
NMsNSClientNumreqSigVerify	104	8 bytes	Binary	Number of verify signature requests received from client
NMsNSClientNumreqCACacheRefresh	112	8 bytes	Binary	Number of CA cache refreshes requested by client
NMsNSClientNumNMIForward	120	8 bytes	Binary	Number of NMI requests forwarded to the client by the server

Initialization and termination messages

When a network management application successfully connects to the interface, the server sends an initialization message (a message with type NMsec_INIT) to the client. This message contains no records, but the return code and reason code are 0 to indicate successful connection completion.

When the server closes the connection (the connection can close as a result of an error, NSSD terminating, or the application's closing of the socket), the server attempts to send a termination message (a message with type NMsec_TERM) to the client. This message contains no records, but the return code and reason code indicate the cause of the connection's termination.

Return and reason codes

When sending a request, the client application should set the message header fields NMsMRc (return code) and NMsMRsn (reason code) to 0. Upon return, the server sets these fields as follows to indicate the status of the request.

Table 58. Request return and reason codes

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
0	0	No error.
ENXIO	0	The requested NSS client is not connected. System Action: Request is failed but the connection remains open. Response: Send requests for active NSS clients only.
EOPNOTSUPP	0	The requested NSS client is not enabled for remote monitoring. System Action: Request is failed but connection remains open. Response: Send requests only for NSS clients that are enabled for remote monitoring. Otherwise, configure and permit the given NSS client for remote monitoring.
EINVAL	NMsRsnBadIdent (1)	Invalid message or record identifier supplied in message. System Action: Connection is closed. Response: Re-issue the connection and send a correctly formatted message.
EINVAL	NMsRsnBadVersion (2)	Invalid version supplied in message header. System Action: Request is failed but connection remains open. Response: Send a correctly formatted message.
EINVAL	NMsRsnBadType (3)	Unsupported or unknown message type supplied in message header. System Action: Request is failed but connection remains open. Response: Send a supported message type.
EINVAL	NMsRsnExcessiveSize (4)	Excessive message size. System Action: Connection is closed. Response: Reissue the connection and send a correctly formatted message.
EINVAL	NMsRsnHdrSize (5)	Message header size is invalid. System Action: Request is failed but connection remains open. Response: Send a message with the header size field set to the correct value.

Table 58. Request return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
EINVAL	NMsRsnMsgSize (6)	Message size is invalid. For example, the message might be too short, or the message size might be greater than the sum of its parts. System Action: Request is failed but connection remains open. Response: Send a correctly formatted message.
EINVAL	NMsRsnReservedNonzero (7)	Reserved data in message header, record header, or record data specifies a non-zero value. Reserved fields must be set to 0 for compatibility with future versions of the interface. System Action: Request is failed but connection remains open. Response: Send a message with reserved fields set to 0.
EINVAL	NMsRsnRecordLength (8)	Unrecognized record length supplied in message. Length does not correspond to known record data. System Action: Request is failed but connection remains open. Response: Send a message with input filters of the correct length.
EINVAL	NMsRsnRecordCount (9)	Unsupported record count supplied in message. NMI requests currently support a maximum of 20 input filters. System Action: Request is failed but connection remains open. Response: Send a message with the correct number of input filters.
EINVAL	NMsRsnSectionLength (10)	Unrecognized section length supplied in record. Length does not correspond to known section data. System Action: Request is failed but connection remains open. Response: Send a message with correct input filters.
EINVAL	NMsRsnSectionCount (11)	Unrecognized section count supplied in record. NMI requests currently allow one section in an input filter record. System Action: Request is failed but connection remains open. Response: Send a message with correct input filters.

Table 58. Request return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
EINVAL	NMsRsnFilterSpec (12)	The input filter specification indicates a combination of filter values that is unsupported for the message's request type. System Action: Request is failed but connection remains open. Response: Send a message with a valid input filter specification for the message type.
EINVAL	NMsRsnFilterValue (13)	The input filter specification contains a value that is out of range. System Action: Request is failed but connection remains open. Response: Send a message with correct input filter values.
EINVAL	NMsRsnManTypeConflict(14)	Manual tunnel activation and deactivation requests for multiple tunnels must contain uniform tunnel specifications: either tunnel IDs or tunnel names. The request contained a mixture of tunnel names and tunnel IDs. System Action: Request is failed but connection remains open. Response: Separate manual tunnel names and tunnel IDs into different requests.
EINVAL	NMsRsnPolicySource (15)	The policy source value in the policy load request is invalid. System Action: NO action is required. Response: Send a message with a valid NMsecPolSrcSource value.
EACCES	0	Access denied to the requested resource. System Action: Request is failed but connection remains open. Administrator Response: Permit the user to the security resource.
ENOMEM	0	Insufficient storage available in the server to process the request. System Action: Request is failed but connection remains open. Response: Increase the REGION size for the IKE daemon, or send a message with a narrower set of input filters to limit the response.
ENOMEM	NMsRsnTooManyConns (1)	The NMI thread is already using its maximum number of 50 connections and cannot accept any more. System Action: Connection is not opened and the request is failed. Response: Try the request again later.

Table 58. Request return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
ENOMEM	NMsRsnNSClient (2)	Insufficient storage available in the NSS client to process the request. System Action: Request is failed but connection remains open. Response: Increase the REGION size for the NSS client, or send a message with a narrower set of input filters to limit the response.
ETIMEDOUT	NMsRsnNSClient (2)	Response message was not received from the NSS client in sufficient time. System Action: Request is failed but connection remains open. Response: Resend the request message to the server.
EIO	(z/OS UNIX System Services Errno)	A file system I/O error occurred. The reason code contains the errno value that describes the error. System Action: Request is failed but connection remains open. Response: Diagnose the z/OS UNIX System Services Errno using <i>z/OS UNIX System Services Messages and Codes</i>.
EMVSERR	0	A call to an MVS service failed or an internal NSSD error has occurred. System Action: Request fails but connection remains open. A message appears in the MVS system log with additional diagnostic information. Response: Contact IBM service.

Real-time TCP/IP network monitoring NMI

Table 59 shows the z/OS Communications Server network management interfaces that are described in this section.

Table 59. z/OS Communications Server network management interfaces

Name	Description
SYSTCPDA	Real-time TCP/IP packet trace data NMI
SYSTPCPN	Real-time TCP connection SMF data NMI
SYSTCPSM	Real-time SMF data NMI

These interfaces allow network management applications to programmatically obtain data in real time. Details for invoking these interfaces and the data provided from them are documented in the following sections. Programmers understand how to parse the data retrieved from these interfaces, and the data structures required to perform this function. Instructions for compiling and linking applications are also provided.

Overview

Each of the interfaces described in this section provides a unique type of data to be processed by the end user, but the general interface by which the data is obtained is essentially the same. The records are retrieved using a common data layout, although the records themselves might differ in format depending on the interface.

Tip: New SMF 119 records might be added with new releases. If you write an application that processes the SMF 119 records from these NMIs, design the application to receive SMF 119 records that it might not recognize.

The information provided by each interface is as follows:

Table 60. Interface descriptions

Interface	Description
Real-time TCP/IP packet trace data NMI (SYSTCPDA)	Using this interface applications can obtain a copy of network packets (for example, packet trace records) or data trace records that are buffered by the TCP/IP stack's packet or data trace functions. The packet trace function, data trace function, or both must be enabled with the VARY TCPIP,,PKTTRACE command or VARY TCPIP,,DATTRACE command. See the <i>z/OS Communications Server: IP System Administrator's Commands</i> for more information about using the Vary command.
Real-time TCP connection SMF data NMI (SYSTPCPN)	Using this interface applications can be notified when TCP connections are established or terminated in a near real-time fashion. SYSTPCPN provides applications with a copy of records indicating a TCP connection initiation or termination. These records are presented in the same format as SMF type 119 TCP connection initiation and termination records (for example, subtype 1 and 2 records). The interface can also be used to provide records describing existing TCP connections. This interface does not require TCP/IP SMF recording to be active.

Table 60. Interface descriptions (continued)

Interface	Description
Real-time SMF data NMI (SYSTCPSM)	The records provided through the interface are type 119 SMF records. The specific subtypes that are provided are: • TN3270E Telnet server session initiation and termination records (subtypes 20 and 21) • TSO Telnet client connection initiation and termination records (subtypes 22 and 23) • FTP server transfer completion records (subtype 70) • FTP server logon failure records (subtype 72) • FTP client transfer completion records (subtype 3) Except for the MVS SMF header, these records are identical in format to SMF records created by TCP/IP. Some fields in the MVS SMF header are not set. These records offer several key advantages over SMF records: • They do not require that TCP/IP SMF record capturing is activated. • They are presented to the application in a buffered format (for example, when several SMF records are created within a short time interval, they are collected and passed to the application as a group of records instead of individual records). In addition to these records, more records are available across this interface that are not currently available from TCP/IP SMF records processing: • FTP server transfer initiation records (subtype 100) • FTP client transfer initiation records (subtype 101) • FTP client login failure records (subtype 102) • FTP client session records (subtype 103) • FTP server session records (subtype 104) See “Real-time SMF data NMI (SYSTCPSM) record formats” on page 545 for the structures and mappings of records 100 through 104.

All of these interfaces provide the same two-step process for accessing the data:

1. The Communications Server TCP/IP stack provides an AF_UNIX stream socket for each of the interfaces (see Table 60 on page 529) that allows one or more applications to receive notifications for the data that is being collected. The TCP/IP stack is acting as the server for these AF_UNIX stream sockets, performing the listen() and waiting for incoming connection requests. Applications wishing to exploit this interface connect to the server's listening AF_UNIX stream socket. Each of the interfaces has its own, distinct AF_UNIX path name that uniquely identifies the socket to be used by the interface. In the case of SYSTCPDA and SYSTCPSM, once connected, the application immediately starts receiving applicable data. In the case of SYSTPCPN, after connecting, the application must send a record to the server to indicate the type of data it desires, only after which does it start receiving applicable data.
2. Each notification record received by the application over the socket represents a buffer being stored by the TCP/IP stack. It is important to understand that the actual SYSTCPDA, SYSTPCPN and SYSTCPSM data is not part of this notification record. After receiving the entire notification record from the

AF_UNIX socket, the application must then pass this record along with a user-allocated storage buffer to the EZBTMIC1 API routine provided. EZBTMIC1 populates the provided storage buffer with the output records related to the interface that the input notification record defines. Once the notification is received over the AF_UNIX socket, the application must invoke EZBTMIC1 (or TMI_CopyBuffer) immediately because the buffers are stored in a circular queue by the TCP/IP stack, and might eventually be overwritten and invalidated. The network management application also needs to execute at a relatively high priority to ensure that it gets dispatched by the system reasonably quickly so that it can obtain the data before those buffers are overwritten.

The buffer copied using the EZBTMIC1 API call contains the actual data of interest to the application. The format of these buffers, and the records contained therein, are described in “Understanding the common buffer output of the TMI copy buffer service” on page 539.

In summary, the application connects to open an AF_UNIX socket path name that is defined for the network management interface for which it would like to collect information (for example, SYSTPCPN, SYSTCPDA, SYSTCPSN) and receives notification records. It passes these records to EZBTMIC1 to copy the actual data of interest into the application’s storage. The application then parses the records in the returned buffer to obtain the actual packet trace or SMF-type records. It is possible for the network management application to connect to one or more of these interfaces from the same application. The application passes these records to an API call to copy the actual data of interest into the application’s storage.

Configuration and enablement

In order for the TCP/IP stack to collect the data for these interfaces and accept connections over the AF_UNIX socket from clients that want to connect, you must first enable them within the TCP/IP configuration using the NETMONITOR statement in the TCP/IP profile. See the NETMONITOR statement information in the *z/OS Communications Server: IP Configuration Reference* for details. If you are developing a feature for a product to be used by other parties, you should include in your documentation instructions indicating that administrators must make these configuration changes in order to use that feature.

The z/OS system administrator might restrict access to each of these interfaces by defining the SERVAUTH class EZB.NETMGMT.*sysname.tcpprocname.interface* profile with UACC of NONE in RACF (or the equivalent security product), and permitting only certain management applications or users to access that interface.

Guidelines:

1. The user ID referenced for this authorization check is the user ID associated with the task and MVS address space that issues the connect() call for the AF_UNIX stream socket.
2. *sysname* represents the MVS system name where the interface is being invoked.
3. *tcpprocname* represents the job name associated with a TCP/IP started task procedure.
4. *interface* represents SYSTCPDA, SYSTPCPN, or SYSTCPDM.

Information about the SERVAUTH classes
EZB.NETMGMT.*sysname.tcpprocname*.SYSTCPDA,
EZB.NETMGMT.*sysname.tcpprocname*.SYSTPCPN, and

EZB.NETMGMT.sysname.tcprocname.SYSTCPSM can be found in *z/OS Communications Server: IP Configuration Guide*.

If the RACF profile is not defined for the interface, then only superusers (users with an OMVS UID of 0 or users permitted to access the BPX.SUPERUSER resource in the FACILITY class) are permitted to use the interface. If you are developing a feature for a product to be used by other parties, include in your documentation instructions indicating that administrators must either define and give appropriate permission to the given security resource for use of that feature, or must run your program as superuser.

Connecting to the server

The application wishing to make use of one of the interfaces must connect to the appropriate AF_UNIX stream socket provided by TCP/IP, which acts as the server. The socket path names for each of these interfaces are as follows. For each of the following, *tcprocname* is the procedure name used to start TCP/IP.

- Network monitor interface for capturing data packets (SYSTCPDA)
/var/sock/SYSTCPDA.tcprocname
- Network monitor interface for obtaining TCP connection information (SYSTCPN)
/var/sock/SYSTCPN.tcprocname
- Network monitor interface for obtaining real-time SMF data (SYSTCPSM)
/var/sock/SYSTCPSM.tcprocname

Use either the Language Environment C/C++ API or the UNIX System Services BPX callable services to open AF_UNIX sockets and connect to the given service.

Interacting with the servers

In the case of the TCP connection information service, after connecting to the SYSTCPN server over AF_UNIX socket, */var/sock/SYSTCPN.tcprocname*, the application must then send a connection request record to the server over the connected socket (see the *tmi_conn_request* record in “Requests sent by the client to the server” on page 533). For the other two services, the application does not need to take action.

After the client connects to the desired server (or, in the case of the SYSTCPN service, after sending a connection request record), the server sends an initial record to the client, identifies the server (see the *tmi_init* record in “Initialization record” on page 534). After that record is received, the client is sent *tmi_token* records that represent data buffers. A record is sent for each data buffer that is filled in by TCP/IP. Records for partial buffers are sent if there has been no activity for a brief period. In case there is no activity, the client should be prepared to wait indefinitely for incoming tokens.

When the server needs to terminate the connection, it attempts to send a special termination record (see the *tmi_term* record in “Termination record” on page 535) over the socket to the connected application, after which it closes the socket. This termination record describes the reason for closure. In some cases, the server might be unable to send such a record, and will close the socket. The application should be prepared to handle either case.

Particularly for the SYSTCPDA and SYSTCPN interfaces, large amounts of data can be generated. Care should be taken in the case of SYSTCPDA not to activate a packet trace filter option that is too broad, in order to avoid recording unnecessary

data; see the VARY TCPIP,,DATTRACE or VARY TCPIP,,PKTTRACE information in the *z/OS Communications Server: IP System Administrator's Commands* and the Packet trace (SYSTCPDA) for TCP/IP stacks information in the *z/OS Communications Server: IP Diagnosis Guide* for details. In the case of SYSTPCPN, the NETMONITOR MINLIFETIME TCP/IP profile configuration option can be used to restrict the collection of short-lived connections; see the NETMONITOR statement information in the *z/OS Communications Server: IP Configuration Reference* for details.

Restriction: Except in the case of sending a connection request record for the SYSTPCPN service, the client application must never send data to the server. If data is unexpectedly received by the server, the server sends a termination record with `tmit_termcode = EPIPE` to the client, and closes the connection.

Common record header

All data sent over the AF_UNIX socket by the client and the server is prefixed with a common header indicating the length of the entire record (this length includes the header) and the type of data contained within the record. The format for the header is as follows, as defined in `ezbytmih.h` (an assembler mapping for this structure is in `EZBYTMIA`):

```
struct tmi_header
{
    int TmiHr_len;           /* Length of this record */
    int TmiHr_Id;            /* Identifier for this record */
    int TmiHr_Ver;           /* Version identifier for this */
    int TmiHr_resv;          /* reserved */
};

#define TmiHr_CnRqst 0xC3D5D9D8 /* Constant("CNRQ") */
/* TCP connection request record */

#define TmiHr_Init 0xC9D5C9E3   /* Constant("INIT") */
/* Connection initialization */

#define TmiHr_Term 0xE3C5D9D4   /* Constant("TERM") */
/* Normal connection termination */

#define TmiHr_SmfTok 0xE2D4E3D2 /* Constant("SMTK") */
/* Token for SMF buffer */

#define TmiHr_PktTok 0xE2D7D3E2 /* Constant("TPKT") */
/* Token for packettrc data */

#define TmiHr_Version1 1        /* Version number */
};
```

Requests sent by the client to the server

For the SYSTPCPN service only, the client must send a request record to the server after connecting to the server's AF_UNIX socket. This request record is in the following format, defined in `ezbytmih.h` (an assembler mapping for this structure is in `EZBYTMIA`):

```
struct tmi_conn_request          /* Conn info server request */
{
    struct tmi_header tmicnrq_hdr; /* Header; id=TMI_ID_CNRQST */
    unsigned int      tmicnrq_list :1; /* Requests connection list */
    unsigned int      tmicnrq_smf :1; /* Requests init/term SMFrcd */
    unsigned int      tmicnrq_rsvd1 :30; /* Reserved, set to 0 */
    char              tmicnrq_rsvd2[12]; /* Reserved, set to 0 */
};
```

The client should initialize the fields of this request structure as follows:

- Initialize `tmicnrq_hdr` using the length of `tmi_conn_request`, the appropriate record ID (`TmiHr_CnRqst`), and the correct version (`TmiHr_Version1`).
- Initialize the `tmicnrq_list` and `tmicnrq_smf` fields as described in the following list.
- Initialize all remaining fields to 0.

The two fields *tmicnrq_list* and *tmicnrq_smf* control the data that the SYSTPCPN server sends to the client. These fields should be set as follows:

- *tmicnrq_list*
If set, the server sends the client zero or more tokens that represent data buffers that contain a list of all established TCP connections at the time the client connected. These connections are represented as type 119 TCP connection initiation SMF records. If this field is set to 0, no such list is sent to the client.
- *tmicnrq_smf*
If set, the server sends tokens to the client. These tokens represent data buffers that contain type 119 TCP connection initiation and termination SMF records, representing TCP connections that are established and closed on the TCP/IP stack. If this field is set to 0, the server does not send any tokens, representing ongoing connection establishment and closure.

The SYSTPCPN server waits until it has received this entire record from the client before it starts processing connection information on the client's behalf. If the client does not send a complete record, then the server never reports data to the client, since the client has not completed initialization. If the server receives a record with an unrecognized version, a bad length, or a bad eyecatcher, then it sends a termination record (see "Termination record" on page 535) with *tmit_termcode* = EINVAL to the client, and closes the connection.

Records sent by the server to the client

For each of the three interfaces, the server sends three types of records to the client:

- Initialization records
- Termination records
- Token records

Each record is described in the sections that follow.

Initialization record

After the client connects to the server, the server sends an initialization record to the client. The initialization record might be recognized as having a *TmiHr_Id* equal to *TmiHr_Init*. This record contains miscellaneous information about the server and the stack that the client can choose to use or ignore. This record has the following format, defined in *ezbytmi.h* (an assembler mapping for this structure is in *EZBYTMIA*):

```
struct tmi_init /* Connection startup record */
{
    struct tmi_header tmii_hdr; /* Record header */
    char tmii_syn[8]; /* System name (EBCDIC) */
    char tmii_comp[8]; /* Component name (EBCDIC) */
    char tmii_sub[8]; /* TCPIP job name (EBCDIC) */
    char tmii_time[8]; /* Time TCPIP started (STCK) */
    int tmii_bufsz; /* Maximum size of buffer */
    char tmii_rsvd[12]; /* Reserved */
};
```

- The component name, *tmii_comp*, represents the server the client is connected to. This will be one of SYSTCPDA, SYSTPCPN, or SYSTCPSM, depending on the server being accessed.
- The *tmii_bufsz* value is the minimum size of the buffer required to be provided on the EZBTMIC1 call. If the value is 0, a maximum of a 64 KB buffer is copied.

Termination record

The termination record is sent when the server closes the connection. The termination record might be recognized as having a *TmiHr_Id* equal to *TmiHr_Term*. The connection might be closed as part of normal operation (for example the service is being disabled or the stack is terminating), or it might be closed as the result of some error. A termination code in the record indicates the termination reason.

This record is the last data sent by the server before close; after sending the termination record, the server closes the connection. The stack attempts to send the termination record before it closes the socket. However, under certain abnormal stack termination conditions, it might be unsuccessful; furthermore, if the client's receive buffer is full, it might also be unsuccessful. In such cases the connection is closed.

The format of this record is as follows, as defined in *ezbytmih.h* (an assembler mapping for this structure is in *EZBYTMIA*):

```
struct tmi_term                                /* Termination notification rcd */
{
  struct tmi_header    tmit_hdr;                /* Record header */
  unsigned int          tmit_termcode;           /* Termination code */
  char                  tmit_tstamp[8];         /* Termination timestamp */
  char                  tmit_rsvd[12];          /* Reserved */
};
```

The possible values for *tmit_termcode* and their explanations are as follows, as defined in *errno.h*:

Value	Description
0	No error; planned termination. Either this function is being disabled or the TCP/IP stack is ending.
EACCES (111)	The client is not permitted to connect to the server.
EINVAL (121)	The client has sent data that is not valid to the server.
ENOMEM (132)	The server was unable to allocate necessary storage.
EPIPE (140)	The client has erroneously sent data to the server when the server was not expecting data.
EWOULDBLOCK (1102)	The server could not write to the client socket because the client's receive buffer is full (in which case it is possible that the server might not have been able to write this record and closed the connection).

See *z/OS UNIX System Services Messages and Codes* for more detail.

The *tmit_tstamp* field contains an 8-byte MVS TOD clock value for the time of termination of the connection.

The client should expect to receive no more data on the connection following this record; the connection will be closed by the server.

Token record

The server sends the *tmi_token* record when a buffer has been filled with records for the given service. The token record might be recognized as having a *TmiHr_Id* equal to *TmiHr_PktTok* (in the case of SYSTCPDA) or *TmiHr_SmfTok* (in the case of SYSTPCPN and SYSTCPSM). In addition, each of the servers will, after a brief period of inactivity, flush a partially filled buffer, sending a token for that partial buffer and advancing to the next internal buffer.

The format of this record is as follows, as defined in *ezbytmih.h* (an assembler mapping for this structure is in EZBYTMIA):

```
struct tmi_token
{
    struct tmi_header    tmik_hdr;        /* Record header */
    char                 tmik_token[32];  /* Token representing buffer */
};
```

The *tmik_token* record contains a token describing the data buffer. The client's actions upon receiving this record are discussed in "Copying the trace buffer."

Copying the trace buffer

Use the EZBTMIC1 service to copy the data buffer to the client application's storage. EZBTMIC1 can be invoked through a C function, *TMI_CopyBuffer*, which calls the callable service. EZBTMIC1 uses the *tmi_token* record just read from the AF_UNIX socket as input to locate and copy the data buffer to the user-provided buffer.

EZBTMIC1 - Copy TCP/IP Management Interface Data Buffer

The EZBTMIC1 callable service uses a token provided over a TCP/IP management interface to copy a data buffer into application storage. This service is also referred to as the TMI copy buffer service.

Requirements

Authorization:	Supervisor state or problem state, any PSW key; Caller must be APF authorized.
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
AMODE:	31-bit
ASC mode:	Primary mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and in the primary address space.

Format

```
CALL EZBTMIC1,(Token,
                Bufptr,
                Return_value,
                Return_code,
                Reason_code)
```

Parameters

Token The name of a record containing a token describing a TCP/IP management interface data buffer.

Type: Structure
Length: Size of buffer token record

Bufptr The address of a buffer into which the TCP/IP management data buffer will be copied.

Type: Structure
Length: 12

The *bufptr* parameter is a 12-byte structure describing the address of the buffer:

```
Bufptr      DS 0F          /* Buffer pointer */  
Buf_alet    DC F'0'        /* Buffer ALET, or 0 */  
Buf_addr_hi DC F'0'        /* Highword of 64bit bufptr */  
Buf_addr     DC A(0)       /* Lowword of 64bit bufptr */
```

If the buffer is in a data space, then *Buf_alet* is the ALET of the data space; otherwise it is zero. If the buffer is in 64-bit storage, then *Buf_addr_hi* and *Buf_addr* contain the 64-bit address of the buffer. If the buffer is in 24- or 31-bit storage, then *Buf_addr_hi* contains zeros and the buffer address in *Buf_addr*. To improve performance, place the buffer on a page boundary.

This buffer can represent the following:

- When the token is a *TmiHr_PktTok*, the data buffer will contain the unformatted packet trace data records (SYSTCPDA).
- When the token is a *TmiHr_SmfTok*, the data buffer will contain SMF records (SYSTCPCN or SYSTCPSM).

Return_value

Returned parameter. The name of a fullword in which the TMI buffer copy service returns the results of the request:

Type: Integer
Length: Fullword

- >0: The data buffer has been successfully copied into the application buffer. The return value is the number of bytes of data that has been copied into the buffer. This length does not include the trailing halfword of zeros in the buffer.
- -1: The system could not complete the request, for reasons such as the data buffer being no longer valid. See *Return_code* and *Reason_code* for more details.

Return_code

Returned parameter. The name of a fullword in which the TMI buffer copy service stores the return code. The TMI buffer copy service returns *Return_code* only if *Return_value* is -1. The TMI buffer copy service can return one of the following values in the *Return_code* parameter:

Return_value	Return_code	Meaning
>0	0	The request was successful.

Return_value	Return_code	Meaning
-1	EACCES	The application is not APF authorized.
-1	EBADF	The token provided to locate a buffer is not a valid token.
-1	EFAULT	The address is incorrect.
-1	EINVAL	The token provided to locate a buffer does not specify a valid data buffer.
-1	EILSEQ	The data buffer described by token has been overwritten and is no longer available.

Reason_code

The name of a fullword in which the TMI buffer copy service stores the reason code.

Type: Integer
Length: Fullword

The TMI buffer copy service returns *Reason_code* only if *Return_value* is -1. The reason code contains diagnostic information and is described in *z/OS UNIX System Services Messages and Codes*.

Usage notes

- Compiling and linking
 - Assembler mappings for the various records that flow over the AF_UNIX socket can be found in macro EZBYTMIA.
 - This routine is in SYS1.CSSLIB as the callable stub EZBTMIC1.

TMI_CopyBuffer - Copy TCP/IP Management Interface Data Buffer

The TMI_CopyBuffer() function copies the TMI data buffer described in *token* to the application-provided buffer pointed to by *bufptr*. Ezbytmih.h contains this definition.

Format

```
void Tmi_CopyBuffer (struct tmi_header *token,
                    struct bufptr_t *bufptr,
                    int *retval,
                    int *retcode,
                    int *rsncode);
```

Parameters

token The pointer to the token record read from the TCP/IP management interface service. The record contains a token used to locate a data buffer to be copied.

bufptr A pointer to a *tmi_bufptr* structure describing a 64 KB buffer provided by the user. The indicated buffer will be overwritten with the contents of the TMI data buffer if the call is successful.

The *tmi_bufptr* structure is a 12-byte structure describing the address of the buffer.

```

struct tmi_bufptr      /* Buffer pointer */
{
    int buf_alet;       /* Buffer ALET, or 0 */
    int buf_addr_hi;    /* Highword of 64bit bufptr */
    void *buf_addr;     /* Lowword of 64bit bufptr */
};

```

retval The returned value. If successful, TMI_CopyBuffer() returns the number of bytes copied in *retval*. If unsuccessful, TMI_CopyBuffer() returns -1 in *retval* and returns *retcode* as described in the following definition.

retcode

A pointer to a fullword in which the TMI buffer copy service stores the return code. The TMI buffer copy service returns *retcode* only if *retval* is -1. The TMI buffer copy service can return one of the following values in the *retcode* parameter.

Return_code	Meaning
EACCES	The application is not APF authorized.
EBADF	The token provided to locate a buffer is not a valid token.
EFAULT	Using the buffer parameter as specified would result in an attempt to access storage outside the caller's address space.
EINVAL	The token provided to locate a buffer does not specify a valid data buffer.
EILSEQ	The data buffer described by token has been overwritten and no longer available.

rsncode

The address of a fullword in which the TMI buffer copy service stores the reason code. The TMI buffer copy service returns *rsncode* only if *retval* is -1. The reason code contains diagnostic information and is described in *z/OS UNIX System Services Messages and Codes*.

Usage notes

- Character data

Some of the data contained in the TMI data buffer might be system data, such as job names. Such data is encoded in EBCDIC and the application should be prepared to process it appropriately.

- Compiling and linking

The callable service routine that provides this service is provided as a callable stub located in SYS1.CSSLIB.

Understanding the common buffer output of the TMI copy buffer service

Upon successful completion of the EZBTMIC1 call of the TMI_CopyBuffer(), the user-supplied buffer is filled with *cte* records, which contain the data provided by the service being used.

The data records for the server are stored sequentially within individual data buffers. The *cte* describes the length of the data record. The data record is immediately followed by a *cteeplg* (*cte* epilogue) structure. The first *cte* structure begins at the beginning of the buffer. The last *cteeplg* is followed by a *cte* whose *ctelenp* field is 0; this signifies the end of the data in the buffer. The layout of the

buffer is as follows:

<i>cte</i>	data	<i>cte_epilogue</i>	<i>cte</i>	data	<i>cte_epilogue</i>	...	<i>cte</i>	data	<i>cte_epilogue</i>	binary 0

The *cte* is a 16-byte descriptor whose format is as follows (as defined in *ezbytmi.h*, and in ITTCTE in SYS1.MACLIB):

```
struct cte
{
    unsigned    short    ctelenp;    /* Length of CTE
                                   and cte_epilogue. */
    unsigned    short    cteoff;     /* Offset from start of CTE */
    unsigned    long     ctefmtid;   /* Format ID of record */
    unsigned    long long ctetime;   /* STCK timestamp of record
creation */
};
```

ctelenp holds the total length of the record, including the *cte*, the data record, and the *cte_epilogue*. *cteoff* is the offset to the data record from the start of the *cte*. The *ctefmtid* is a format ID specific to each service; it is described in “Format of service-specific data.” The *ctetime* is an 8-byte STCK timestamp of the time the record was written.

The format of the 2-byte *cteeplg* is as follows (as defined in *ezbytmi.h*, and in ITTCTE in SYS1.MACLIB):

```
struct cteeplg
{
    unsigned    short    ctelene;    /* Length of CTE, data, and
                                   cte_epilogue. */
};
```

The field *ctelene* holds the same value as the *ctelenp* field in the *cte*.

Format of service-specific data

The following sections describe how to process CTE records for SYSTCPDA, SYSTPCPN, and SYSTCPSM.

Processing the CTE records for SYSTCPDA

The following *ctefmtid* values are supported for the SYSTCPDA interface:

ctefmtid	Data area	Description	Command to start
1 (TRCIDPKT) See note	Described by the GtCntl structure	IPv4 packet trace record	VARY TCPIP,,PKTTRACE
2 (TRCIDX25) See note	Described by the GtCntl structure	IPv4 packet trace record	N/A
3 (TRCIDDAT) See note	Described by the GtCntl structure	IPv4 data trace record	VARY TCPIP,,DATTRACE
4 (PTHIdPkt)	Described by the PTHDR_T structure	IPv4 or IPv6 packet trace record	VARY TCPIP,,PKTTRACE
5 (PTHIdDat)	Described by the PTHDR_T structure	IPv4 or IPv6 data trace record	VARY TCPIP,,DATTRACE

ctefmtid	Data area	Description	Command to start
6 (PTHIdEE)	Described by the PTHDR_T structure	IPv4 or IPv6 EE trace record	VARY TCPIP,,PKTTRACE, SRCP =12000
Note: As of the V1R6 release, type 1, 2, and 3 records are no longer written.			

If tracing for the TCP/IP data trace and the TCP/IP packet trace is active, the trace buffer will contain both types of records. The client must handle this condition.

The GtCntl is defined in EZBCTHDR and contains the following information:

<i>gtseqnum</i>	One byte sequence number	
<i>gtsflg</i>	Flag byte	
	GTS PKT	0x80 Packet trace request
	GTSX25	0x40 X.25 Data trace request
	GTS DAT	0x20 Data trace request
	GTSVERS	0x10 Version number always 1
	GTSIUTL	0x08 Data from multiple PDUs
	GTSADJ	0x04 Record size adjust by +1
	GTSABBR	0x02 IP pkt was abbreviated
	GTSPOUT	0x01 IP pkt was sent = 1 rcvd = 0
<i>gtslrct</i>	Lost record count	
<i>gtsrect</i>	Record type (device type)	
	GTS LCSE	1 IFPETH - Ethernet
	GTS LCS8	2 IFP8023 - 802.3 Ethernet
	GTS LCSE8	3 IFPETHOR - Ether 802.3
	GTS LCSTR	4 IFPTR - Token Ring
	GTS LCSFD	5 IFPFDDI - FDDI
	GTS LU62	6 IFPSNA62 - SNA LU6.2
	GTS HCH	10 IFPHCH - HyperChannel
	GTS CLWRS	21 IFPCLIP - CLAW
	GTS CTC	29 IFPCTC - CTC
	GTS CDLC	30 IFPCDIP - CDLC IP
	GTS ATM	32 IFPATM - ATM
	GTS VIPA	33 IFPVIPA - VIPA
	GTS LOOPB	34 IFPLOPB - LoopBack
	GTS MPC	35 IFPMPC - MPC
	GTS X25C	36 IFPX25 - X.25
	GTS SNALN	37 IFPSNALINK - SNA LINK
	GTS MPCIE	39 IFPIPAQENET - MPC IP AQENET link
	GTS MPCOD	40 IFPOSAFDDI - MPC OSAFDDI link
	GTS MPCON	41 IFPOSAENET - MPC OSAFNET link
	GTS MPCIH	42 IFPIPAQTR - MPC IPAQTR link
	GTS QIDIO	43 IFPIPAQIDO - iQdio
<i>gtstlen</i>	Total length of IP packet	
<i>gtslknm</i>	Link name, or data trace job name	
<i>gtssipad</i>	Source IPv4 address	
<i>gtsdpad</i>	Destination IPv4 address	
<i>gtstod</i>	Time of Day timestamp	
<i>gtssport</i>	Source port number (data trace)	
<i>Gtsdport</i>	Destination port number (data trace)	
<i>Gtstcb</i>	MVS TCB address (data trace)	
<i>Gtsasid</i>	ASID (data trace)	

The PTHDR_T is defined in EZBYPTHA and contains the following information:

<i>pth_len</i>	Length of the PTHDR_T structure	
<i>pth_seqnum</i>	Sequence number of this packet trace record	
<i>pth_flag</i>	Flag indicators	
	PTH_Seg_Offload	0x10 TCP Segmentation Offload
	PTH_Pdu	0x08 Data from multiple PDUs
	PTH_Adj	0x04 Record size was adjusted by +1 (reflected in the ctelene and ctelenp). The data length was odd and a single pad byte was added.
	PTH_Abbr	0x02 ABBREV parameter was used on the trace command
	PTH_Out	0x01 IP packet was sent = 1 rcvd = 0
<i>pth_devty</i>	The type of device represented by the interface being traced.	
	PTH LCSE	1 - Ethernet
	PTH LCS8	2 - 802.3 Ethernet
	PTH LCSE8	3 - Ether 802.3
	PTH LCSTR	4 - Token Ring

PTHLCSTFD	5	- FDDI
PTHLU62	6	- SNA LU6.2
PTHCH	10	- HyperChannel
PTHCLWRS	21	- CLAW
PTHCTC	29	- CTC
PTHCDLC	30	- CDLC IP
PTHATM	32	- ATM
PTHVIPA	33	- VIRTUAL
PTHLOOPB	34	- LoopBack
PTHMpc	35	- MPC
PTHX25C	36	- X.25
PTHSNALN	37	- SNA LINK
PTHMPCIG	38	- MPC giga
PTHMPCIE	39	- MPC IPAQENET
PTHMPCOD	40	- MPC OSAFDDI
PTHMPCON	41	- MPC OSAFNET
PTHMPCIH	42	- MPC IPAQTR
PTHQIDIO	43	- iQdio
PTH6loopb	51	- IPv6 loopback
PTH6vipa	52	ifp6vipa
PTH6ipaenet	53	ifp6ipaenet
PTH6ipaqtr	54	ifp6ipaqtr
PTH6mpc	55	ifp6mpc
PTH6ipaqidio	56	ifp6ipaqidio
<i>pth_tlen</i>	Portion of the payload that is actually traced. If ABBREV was not specified on the trace command then this will be the name as <i>pth_plen</i> . If ABBREV was specified, then it will be this value.	
<i>pth_infname</i>	Name of the interface the packet was traced on in EBCDIC character format	
<i>pth_time</i>	Stored time of day clock when packet was processed by the trace	
<i>pth_src</i>	Hexadecimal source IP address of this packet (IPv6 or IPv4)	
<i>pth_dst</i>	Hexadecimal destination IP address of this packet (IPv6 or IPv4)	
<i>pth_sport</i>	Hexadecimal source IP port number	
<i>pth_dport</i>	Hexadecimal destination IP port	
<i>pth_trcnt</i>	Total count of records traced	
<i>pth_tcb</i>	Task control block address of the sender of the outbound. On inbound, this will usually be task associated with the TCP/IP stack	
<i>pth_asid</i>	Ascbasid of the sender of the outbound packet. On inbound, this will usually be the asid of the TCP/IP stack	
<i>pth_lost</i>	Total lost record count	
<i>pth_plen</i>	Payload length (If segmentation is offloaded, then this is the total data length of all segments being offloaded plus the length of one set of headers.)	
PTH_OffSegLen	Length of each of the first N-1 segments being offloaded not including headers - i.e. the MSS (only meaningful when the PTH_Seg_Offload flag is on)	

The fields in the GtCntl and PTHDR_T with the same suffix serve the same purpose in both headers. IPv4 addresses in *pth_src* and *pth_dst* are prefixed with x'000000000000', x'00000000FFFE' or x'00000000FFFF'.

Processing trace records in a buffer: The EZBTMIC1 call or the TML_CopyBuffer() service is used to receive a buffer of trace records defined by a starting CTE structure and ending with a 2-byte *ctelene* field, which has the same value as the *ctelenp*. The PTHDR_T structure follows the CTE and has many fields for use when processing the trace records. The *pth_tlen* field is the IP packet payload length, although this field could reflect the ABBREV parameter on the PKTTRACE command. When TCP segmentation is being offloaded (indicated by flag *pth_seg_offload*), then the *pth_plen* field represents the data length of all the segments being offloaded. When the *pth_seg_offload* flag is set, the *pth_numsegs* field indicates the number of offloaded segments and the *pth_offseglen* field indicates the length of each segment (the MSS). In some cases, to obtain the entire IP packet, multiple trace records must be processed. These trace records could span multiple

buffers and will probably not be contiguous. In this case, several fields must be examined. See the example of packet records in “Example of split buffers for IP packet X.” The *ctelenp* will be less than the *pth_tlen*. The *pth_seqnum* fields must be used to determine the ordered chain of records that make up the IP packet. The first record in the sequence will have *pth_seqnum*=0 and will contain the IP protocol headers. The *pth_tlen* and *pth_time* will be the same for each record in the sequence.

Example of split buffers for IP packet X: First TMI_CopyBuffer() issued; a complete buffer received:

Trace Records			
Record1 for IP packet X CTE structure <i>ctelenp</i> =1 KB	PTHDR_T structure <i>pth_seqnum</i> =0 <i>pth_tlen</i> =64 KB <i>pth_time</i> =Time X	trace data (IP packet X) contains IP headers	<i>ctelente</i> =1 KB

Second TMI_CopyBuffer issued; a complete buffer received:

Trace Records			
Record1 for IP packet Y CTE structure <i>ctelenp</i> =1 KB	PTHDR_T structure <i>pth_seqnum</i> =0 <i>pth_tlen</i> =ip payload length (less than 1 KB)	trace data (IP packet Y) contains IP headers	<i>ctelente</i> =1 KB
Record2 for IP packet X CTE structure <i>ctelenp</i> = <i>nnn</i>	PTHDR_T structure <i>pth_seqnum</i> =1 <i>pth_tlen</i> =64 KB <i>pth_time</i> =Time X	trace data (IP packet X continued) no headers	<i>ctelente</i> = <i>nnn</i>

Last TMI_CopyBuffer() issued; a partial buffer received:

Trace Records			
Record <i>n</i> for IP packet X CTE structure <i>ctelenp</i> = <i>nnn</i>	PTHDR_T structure <i>pth_seqnum</i> = <i>n</i> -1 <i>pth_tlen</i> =64 KB <i>pth_time</i> =Time X	trace data (IP packet X continued) No headers	<i>ctelente</i> = <i>nnn</i>

Processing the CTE records for SYSTCPCN

The TCP connection information server (SYSTCPCN) presents information about the establishment and closing of TCP connections as they occur. Type 119 SMF TCP connection initiation and termination records (subtypes 1 and 2) are stored in the data buffer to reflect this activity. Each record in the data buffer will be a complete type 119 SMF record, of subtype 1 or 2.

Additionally, if requested, the server will fill one or more buffers with the list of currently active connections. This list is provided as type 119 TCP connection initiation records (subtype 1), so that entries in the list will be indistinguishable

from newly established connections (except that the connection establishment timestamp will be in the past). This set of records is sent only once per new connection, after the initialization.

For the TCP connection information server, the *ctefmtid* for the CTE will always be equal to the subtype of the SMF record (either 1 or 2) following the CTE in the data buffer.

Applications can use this interface to dynamically maintain a list of active TCP connections. As a result of timing issues, it is possible that an application will receive two initiation records for a given connection (if the connection is established around the time the client connects, its initiation record will be sent, as will a record identifying it as a pre-existing established connection). It is also possible that an application will receive a termination record for a connection for which it has not received an initiation record. Client applications should be prepared to handle both of these possibilities.

SMF recording for TCP connection initiation and termination records does not need to be active for this service to function. Moreover, activating this service does not cause TCP connection initiation and termination SMF records to be recorded into the SMF data sets if they are not already enabled.

C structures for mapping the SMF type 119 records can be found in *ezasmf.h*. Assembler mappings for the structures can be found in *EZASMF77* in *SYS1.MACLIB*.

Processing the CTE records for SYSTCPSM

The real-time SMF data server (SYSTCPSM) reports type 119 SMF event records for TCP/IP applications. Each record in the data buffer is a complete type 119 SMF record. The records reported, and their subtypes, are as follows:

- FTP client transfer initialization (subtype 101)
- FTP client transfer completion (subtype 3)
- FTP client login failure records (subtype 102)
- FTP client session records (subtype 103)
- FTP server transfer initialization (subtype 100)
- FTP server session records (subtype 104)
- FTP server transfer completion (subtype 70)
- FTP server logon failure (subtype 72)
- TN3270E Telnet server session initialization (subtype 20)
- TN3270E Telnet server session termination (subtype 21)
- TSO Telnet client connection initialization (subtype 22)
- TSO Telnet client connection termination (subtype 23)

For the real-time SMF data server, the *ctefmtid* for the CTE will always be equal to the subtype of the SMF record (one of the values listed above) following the CTE in the data buffer. The structures and macros for mapping the SMF 119 record subtypes delivered by these interfaces are as follows:

Subtype	C/C++	Assembler macro
3, 20, 21, 22, 23, 70, 72	SEZANMAC(EZASMF)	SYS1.MACLIB(EZASMF77)

Subtype	C/C++	Assembler macro
100, 101, 102, 103, 104	SEZANMAC(EZANMFTC)	SYS1.MACLIB(EZANMFTA)

The FTP type 119 records of subtypes 100 through 104 are available only across the SYSTCPSM real-time interface.

See SMF type 119 records information in the *z/OS Communications Server: IP Configuration Reference* for the formats of SMF type 119 records.

Real-time SMF data NMI (SYSTCPSM) record formats

This section describes record formats.

TCP/IP identification section

Every record starts with a common section followed by the TCP/IP identification section. The self-defining sections vary, as do the sections that follow the TCP/IP identification section. The TCP/IP identification section is identical for all records except one field, SMF119TI_Comp, at offset 32 (X'20') within the record. This field indicates whether the record was written by the FTP client or the FTP server.

Table 61 shows the format of the TCP/IP identification section.

Table 61. TCP/IP identification section format

Offset	Name	Length	Description
0 (X'0')	SMF119TI_SYSName	8	System name from SYSNAME in IEASYSxx
8 (X'8')	SMF119TI_SysplexName	8	Sysplex name from SYSPLEX in COUPLExx
16 (X'10')	SMF119TI_Stack	8	TCP/IP stack name for z/OS Communications Server 390
24 (X'18')	SMF119TI_ReleaseID	8	TCP/IP release identifier
32 (X'20')	SMF119TI_Comp	8	TCP/IP subcomponent (right padded with blanks): - FTPS: FTP server; the FTP server wrote this record - FTPC: FTP client; the FTP client wrote this record
40 (X'28')	SMF119TI_ASName	8	Started-task qualifier or address-space name of address space that writes this SMF record
48 (X'30')	SMF119TI_UserID	8	User ID of security context under which this SMF record is written
56 (X'38')	SMF119TI_ASID	4	ASID of address space that writes this SMF record
60 (X'3C')	SMF119TI_Reason	4	Reason for writing this SMF record: X'08': Event SMF record

FTP server transfer initialization record

Table 62 on page 546 shows the FTP server transfer initialization self-defining section of SMF record.

Table 62. FTP server transfer initialization self-defining section

Offset	Name	Length	Format	Description
0 (X'0')	Standard SMF header	24	N/A	Standard SMF header; subtype is 100 (X'64')
Self-defining section				
24 (X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (V1R4: 5, V1R5: 6)
26 (X'1A')		2		Reserved
28 (X'1C')	SMF119IDOff	4	Binary	Offset to TCP/IP identification section. *
32 (X'20')	SMF119IDLen	2	Binary	Length of TCP/IP identification section. *
34 (X'22')	SMF119IDNum	2	Binary	Number of TCP/IP identification sections. *
36 (X'24')	SMF119S1Off	4	Binary	Offset to FTP server transfer initialization section
40 (X'28')	SMF119S1Len	2	Binary	Length of FTP server transfer initialization section
42 (X'2A')	SMF119S1Num	2	Binary	Number of FTP server transfer initialization sections
44 (X'2C')	SMF119S2Off	4	Binary	Offset to FTP server hostname section
48 (X'30')	SMF119S2Len	2	Binary	Length of FTP server hostname section
50 (X'32')	SMF119S2Num	2	Binary	Number of FTP server hostname sections
52 (X'34')	SMF119S3Off	4	Binary	Offset to FTP server first associated data set name section
56 (X'38')	SMF119S3Len	2	Binary	Length of FTP server first associated data set name section
58 (X'3A')	SMF119S3Num	2	Binary	Number of FTP server first associated data set name sections
60 (X'3C')	SMF119S4Off	4	Binary	Offset to FTP server second associated data set name section
64 (X'40')	SMF119S4Len	2	Binary	Length of FTP server second associated data set name section
66 (X'42')	SMF119S4Num	2	Binary	Number of FTP server second associated data set name sections
68 (X'44')	SMF119S5Off	4	Binary	Offset to FTP server security section (V1R5)
72 (X'48')	SMF119S5Len	2	Binary	Length of FTP server security section (V1R5)
74 (X'4A')	SMF119S5Num	2	Binary	Number of FTP server security sections (V1R5)
See Table 61 on page 545 for the contents of the TCP/IP stack identification section.				

Table 63 on page 547 shows the FTP server transfer initialization record section (located physically after the TCP/IP identification section in the record). This section is slightly different from the one in the transfer completion record and the

field names are therefore different from the completion record. The mapping of this record section is in EZANMFTA (assembler macro) for assembler code and in EZANMFTC (a C header) for C code.

Table 63. FTP server transfer initialization record section

Offset	Name	Length	Description
0	SMF119FT_FSIOPer	1	FTP Operation according to SMF77 subtype classification (this is really redundant information, the same information can be found in SMF119FT_FSICmd). • X'01': Append • X'02': Delete • X'03': Rename • X'04': Retrieve • X'05': Store • X'06': Store Unique
1	SMF119FT_FSIActPas	1	Passive or active mode data connection: • X'00' active using default ip and port • X'01' active using PORT • X'02' active using EPRT • X'03' passive using PASV • X'04' passive using EPSV
2		2	Reserved
4	SMF119FT_FSICmd	4	FTP command (according to RFC 959+; see Appendix E, "Related protocol specifications," on page 719 for information about accessing RFCs)
8	SMF119FT_FSIFType	4	File type (SEQ, JES, or SQL)
12	SMF119FT_FSIDRIP	16	Remote IP address (data connection)
28	SMF119FT_FSIDLIP	16	Local IP address (data connection)
44	SMF119FT_FSIDRPort	2	Remote port number (data connection)
46	SMF119FT_FSIDLPort	2	Local port number (data connection - server)
48	SMF119FT_FSICRIP	16	Remote IP address (control connection)
64	SMF119FT_FSICLIP	16	Local IP address (control connection)
80	SMF119FT_FSICRPort	2	Remote port number (control connection - client)
82	SMF119FT_FSICLPort	2	Local port number (control connection - server)
84	SMF119FT_FSISUser	8	Client user ID on server
92	SMF119FT_FSIFType	1	Data type • A: ASCII • E: EBCDIC • I: Image • B: Double-byte • U: UCS-2

Table 63. FTP server transfer initialization record section (continued)

Offset	Name	Length	Description
93	SMF119FT_FSIMode	1	Transmission mode • B: Block • C: Compressed • S: Stream
94	SMF119FT_FSIStruct	1	Data structure • F: File • R: Record
95	SMF119FT_FSIDsType	1	Data set type • S: SEQ • P: PDS • H: z/OS UNIX file
96	SMF119FT_FSISTime	4	Data connection start time, formatted in 1/100 seconds since midnight [using Coordinated Universal Time (UTC)]
100	SMF119FT_FSISDate	4	Data connection start date (format: 0ccyydddF). If the start date is not available, the value specified will be X'0000000F'.
104	SMF119FT_FSICSTime	4	Control connection start time in 1/100 seconds since midnight [using Coordinated Universal Time (UTC)] (FTP session start time)
108	SMF119FT_FSICSDate	4	Control connection start date (format: 0ccyydddF). If the end date is not available, the value specified will be X'0000000F' (FTP sessions start date)
112	SMF119FT_FSIM1	8	PDS Member name
120	SMF119FT_FSIM2	8	Second PDS member name (if rename operation)
128	SMF119FT_FSICConnID	4	TCP connection ID of FTP control connection (V1R8 and later)
132	SMF119FT_FSIDConnID	4	TCP connection ID of FTP data connection, or zero (V1R8 and later)
136	SMF119FT_FSISessionID	15	FTP activity logging session ID (V1R8 and later)
151	Reserved	1	Reserved (V1R8 and later)

Table 64 shows the FTP server hostname section, physically located after the FTP server transfer initialization section. This section is optional and is identical to the one present in the transfer completion record, and is present only if a gethostbyaddr operation was performed for the Local IP address.

Table 64. FTP server hostname section

Offset	Name	Length	Description
0	SMF119FT_Hostname	<i>n</i>	Hostname

Table 65 on page 549 shows the FTP server MVS data set name section, physically located after the FTP server hostname section (if present) or the FTP server transfer

initialization section. This section represents the MVS data set names associated with the file transfer and is identical to the one present in the completion record. A second instance of the section will be included for Rename File Transfer operations.

Table 65. FTP server MVS data set name section

Offset	Name	Length	Description
0	SMF119FT_MVSDataset	44	MVS Data set name

Table 66 shows the FTP server z/OS UNIX file name section, physically located after the FTP server MVS data set name section. It is identical to the one present in the completion record. One or two names might be included in this section.

Table 66. FTP server z/OS UNIX file name section

Offset	Name	Length	Description
0	SMF119FT_HFSLen1	2	Length of first z/OS UNIX file name
2	SMF119FT_HFSName1	<i>n</i>	z/OS UNIX file name
2+ <i>n</i>	SMF119FT_HFSLen2	2	Length of second z/OS UNIX file name (0 if only one z/OS UNIX file name is being reported)
4+ <i>n</i>	SMF119FT_HFSName2	<i>m</i>	z/OS UNIX file name

Table 67 displays the FTP server security section.

Table 67. FTP server security section

Offset	Name	Length	Format	Description
0 (X'0')	SMF119FT_FSMechanism	1	EBCDIC	Protection Mechanism • N: None • T: TLS • G: GSSAPI • A: AT-TLS
1 (X'1')	SMF119FT_FSCProtect	1	EBCDIC	Control Connection Protection Level • N: None • C: Clear • S: Safe • P: Private
2 (X'2')	SMF119FT_FSDProtect	1	EBCDIC	Data Connection Protection Level • N: None • C: Clear • S: Safe • P: Private
3 (X'3')	SMF119FT_FSLoginMech	1	EBCDIC	Login Method • P: Password • C: Certificate • T: Kerberos ticket

Table 67. FTP server security section (continued)

Offset	Name	Length	Format	Description
4 (X'4')	SMF119FT_FSPProtoLevel	8	EBCDIC	Protocol level (present only if Protocol Mechanism is TLS or AT-TLS). Possible values are: SSLV2, SSLV3, and TLSV1
12 (X'C')	SMF119FT_FSCipherSpec	20	EBCDIC	Cipher Specification (present only if Protocol Mechanism is TLS or AT-TLS). Possible values when Protocol Level is SSLV2 • RC4 US • RC4 Export • RC2 US • RC2 Export • DES 56-bit • Triple-DES US Possible values when Protocol Level is SSLV3 or TLSV1: • SSL_NULL_MD5 • SSL_NULL_SHA • SSL_RC4_MD5_EX • SSL_RC4_MD5 • SSL_RC4_SHA • SSL_RC2_MD5_EX • SSL_DES_SHA • SSL_3DES_SHA • SSL_AES_128_SHA • SSL_AES_256_SHA
32 (X'20')	SMF119FT_FSPProtoBufSize	4	Binary	Negotiated protection buffer size
36 (X'24')	SMF119FT_FSCipher	2	EBCDIC	Hexadecimal value of Cipher Specification (present only if Protocol Mechanism is TLS or AT-TLS).

FTP server session record

Table 68 describes the FTP server session record.

Table 68. FTP server session record

Offset	Name	Length	Format	Description
0 (X'0')	Standard SMF header	24	N/A	Standard SMF header; subtype is 104 (X'68')
Self-defining section				
24 (X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record
26 (X'1A')		2	Binary	Reserved

Table 68. FTP server session record (continued)

Offset	Name	Length	Format	Description
28 (X'1C')	SMF119IDOff	4	Binary	Offset to TCP/IP identification section. *
32 (X'20')	SMF119IDLen	2	Binary	Length of TCP/IP identification section. *
34 (X'22')	SMF119IDNum	2	Binary	Number of TCP/IP identification sections. *
36 (X'24')	SMF119S1Off	4	Binary	Offset to FTP server session section
40 (X'28')	SMF119S1Len	2	Binary	Length of FTP server session section
42 (X'2A')	SMF119S1Num	2	Binary	Number of FTP server session sections
44 (X'2C')	SMF119S2Off	4	Binary	Offset to FTP server security section
48 (X'30')	SMF119S2Len	2	Binary	Length of FTP server security section
50 (X'32')	SMF119S2Num	2	Binary	Number of FTP server security sections.
* See Table 61 on page 545 for the contents of the TCP/IP identification section.				

Table 69 describes the server session section.

Table 69. Server session section

Offset	Name	Length	Format	Description
0 (X'0')	SMF119FT_FSRRIP	16	Binary	Remote IP address
16 (X'10')	SMF119FT_FSRLIP	16	Binary	Local IP address
32 (X'20')	SMF119FT_FSRRPort	2	Binary	Remote port number (client)
34 (X'22')	SMF119FT_FSRLPort	2	Binary	Local port number (server)
36 (X'24')	SMF119FT_FSRUserID	8	Binary	Client user ID
44 (X'2C')	SMF119FT_FSRReason	4	Binary	Session end reason. - X'00': Normal session end; QUIT or REIN command received - X'01': Security authentication or negotiation failed; incorrect specification of security keywords; possible security handshake deadlock - X'02': Control connection socket error; network error - X'03': Control connection closed prematurely - X'04': Sequence received on control connection was not valid This field is valid only when the value of the SMF119FT_FSREvent field is T.

Table 69. Server session section (continued)

Offset	Name	Length	Format	Description
48 (X'30')	SMF119FT_FSREvent	1	Binary	Session event • I: Session start; client is logged into server • T: Session has ended
49 (X'31')	Reserved	3	Binary	Reserved
52(X'34')	SMF119FT_FSRSTime	4	Binary	Session start time in hundredths of a second since midnight [using Coordinated Universal Time (UTC)].
56 (X'38')	SMF119FT_FSRSDate	4	Binary	Session start date (format: <i>0cyydddF</i>). If the date is not available, the value specified will be X'0000000F'. (FTP sessions start date).
60 (X'3C')	SMF119FT_FSRETime	4	Binary	Session end time in hundredths of a second since midnight [using Coordinated Universal Time (UTC)]. This field is defined only when the value of the SMF119FT_FSREvent field is T.
64 (X'40')	SMF119FT_FSREDate	4	Binary	Session end date (format: <i>0cyydddF</i>). If the date is not available, the value specified will be X'0000000F'. This field is defined only when the value of the SMF119FT_FSREvent field is T.
68 (X'44')	SMF119FT_FSRConnID	4	Binary	TCP connection ID of FTP control connection
72 (X'48')	SMF119FT_FSRSessionID	15	EBCDIC	FTP activity logging session ID
87 (X'57')		1	N/A	Reserved

Table 70 describes the FTP server security section.

Table 70. FTP server security section

Offset	Name	Length	Format	Description
0 (X'0')	SMF119FT_FSNMechanism	1	EBCDIC	Possible values are: • N: None • T: TLS • G: GSSAPI • A: AT-TLS
1 (X'1')	SMF119FT_FSNCPProtect	1	EBCDIC	Control Connection Protection Level. Possible values are: • N: None • C: Clear • S: Safe • P: Private

Table 70. FTP server security section (continued)

Offset	Name	Length	Format	Description
2 (X'2')	SMF119FT_FSNDProtect	1	EBCDIC	Data Connection Protection Level. Possible values are: • N: None • C: Clear • S: Safe • P: Private
3 (X'3')	SMF119FT_FSNLoginMech	1	EBCDIC	Login method: • P: Password • C: Certificate • T: Kerberos ticket
4 (X'4')	SMF119FT_FSNProtoLevel	8	EBCDIC	Protocol level (present only if Protocol Mechanism is TLS or AT-TLS). Possible values are: SSLV2, SSLV3, and TLSV1.
12 (X'C')	SMF119FT_FSNCipherSpec	20	EBCDIC	Cipher Specification (present only if the value of Protocol Mechanism is TLS or AT-TLS). Possible values when Protocol Level is SSLV2: • RC4 US • RC4 Export • RC2 US • RC2 Export • DES 56-Bit • Triple DES US Possible values when Protocol Level is SSLV3 or TLSV1: • SSL_NULL_MD5 • SSL_NULL_SHA • SSL_RC4_MD5_EX • SSL_RC4_MD5 • SSL_RC4_SHA • SSL_RC2_MD5_EX • SSL_DES_SHA • SSL_3DES_SHA • SSL_AES_128_SHA • SSL_AES_256_SHA
32 (X'20')	SMF119FT_FSNProtoBufSize	4	Binary	Negotiated protection buffer size
36(X'24')	SMF119FT_FSNCipher	2	EBCDIC	Hexadecimal value of Cipher Specification (present only if Protocol Mechanism is TLS or AT-TLS).

FTP client transfer initialization record

The following table shows the FTP client transfer initialization self-defining section of the SMF record.

Offset	Name	Length	Format	Description
0 (x'0')	Standard SMF header	24	N/A	Standard SMF header; subtype will be 101 (X'65')
Self-defining section				
24 (x'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (V1R4: 4, V1R5: 5, V1R8: 6)
26 (x'1A')		2		Reserved
28 (x'1C')	SMF119IDOff	4	Binary	Offset to TCP/IP identification section *
32 (x'20')	SMF119IDLen	2	Binary	Length of TCP/IP identification section *
34 (x'22')	SMF119IDNum	2	Binary	Number of TCP/IP identification sections *
36 (x'24')	SMF119S1Off	4	Binary	Offset to FTP client transfer initialization section
40 (x'28')	SMF119S1Len	2	Binary	Length of FTP client transfer initialization section
42 (x'2A')	SMF119S1Num	2	Binary	Number of FTP client transfer initialization sections
44 (x'2C')	SMF119S2Off	4	Binary	Offset to FTP client associated data set name section
48 (x'30')	SMF119S2Len	2	Binary	Length of FTP client associated data set name section
50 (x'32')	SMF119S2Num	2	Binary	Number of FTP client associated data set name sections
52 (x'34')	SMF119S3Off	4	Binary	Offset to FTP client SOCKS section
56 (x'38')	SMF119S3Len	2	Binary	Length of FTP client SOCKS section
58 (x'3A')	SMF119S3Num	2	Binary	Number of FTP client SOCKS sections
60 (x'3C')	SMF119S4Off	4	Binary	Offset to FTP client security section (V1R5 and later)
64 (x'40')	SMF119S4Len	2	Binary	Length of FTP client security section (V1R5 and later)
66 (x'42')	SMF119S4Num	2	Binary	Number of FTP client security sections (V1R5 and later)
68 (x'44')	SMF119S5Off	4	Binary	Offset to FTP client user name section (V1R8 and later)
72 (x'48')	SMF119S5Len	2	Binary	Length of FTP client user name section (V1R8 and later)
74 (x'4A')	SMF119S5Num	2	Binary	Number of FTP client user name sections (V1R8 and later)
* See Table 61 on page 545 for the contents of the TCP/IP stack identification section.				

Table 71 on page 555 describes the FTP client transfer initialization record section, which is physically located after the TCP/IP identification section. This section is slightly different from the one in the transfer completion record and the field

names are therefore different from the completion record. The mapping of this record section is in EZANMFTA (assembler macro) for assembler code and in EZANMFTC (a C header) for C code.

Table 71. FTP client transfer initialization record section

Offset	Name	Length	Description
0	SMF119FT_FCICmd	4	FTP subcommand (according to RFC 959; see Appendix E, “Related protocol specifications,” on page 719 for information about accessing RFCs)
4	SMF119FT_FCIFType	4	Local file type (SEQ or SQL)
8	SMF119FT_FCIDRIP	16	Remote IP address (data connection)
24	SMF119FT_FCIDLIP	16	Local IP address (data connection)
40	SMF119FT_FCIDRPort	2	Remote port number (data connection)
42	SMF119FT_FCIDLPort	2	Local port number (data connection)
44	SMF119FT_FCICRIP	16	Remote IP address (control connection)
60	SMF119FT_FCICLIP	16	Local IP address (control connection)
76	SMF119FT_FCICRPort	2	Remote port number (control connection)
78	SMF119FT_FCICLPort	2	Local port number (control connection)
80	SMF119FT_FCIRUser	8	User ID (login name) on server
88	SMF119FT_FCILUser	8	Local User ID
96	SMF119FT_FCIType	1	Data format • A: ASCII • E: EBCDIC • I: Image • B: Double-byte • U: UCS-2
97	SMF119FT_FCIMode	1	Transfer mode • B: Block • C: Compressed • S: Stream
98	SMF119FT_FCIStruct	1	Structure • F: File • R: Record
99	SMF119FT_FCIDSType	1	Data set type • S: SEQ • P: PDS • H: z/OS UNIX file system
100	SMF119FT_FCISTime	4	Start time of data connection, in a hundredth of a second, since midnight [using Coordinated Universal Time (UTC)]
104	SMF119FT_FCISDate	4	Start date of data connection (format: <i>OcyyddF</i>). If the start date is not available, the value specified will be X'000000F'.
108	SMF119FT_FCICSTime	4	Start time of control connection, in a hundredth of a second, since midnight [using Coordinated Universal Time (UTC)]. FTP session start time.

Table 71. FTP client transfer initialization record section (continued)

Offset	Name	Length	Description
112	SMF119FT_FCICSSDate	4	Start date of the control connection (format <i>0ccyydddF</i>). If the start date is not available, the value specified will be X'0000000F'. FTP session start date.
116	SMF119FT_FCIM1	8	PDS member name
124	SMF119FT_FCIActPas	1	Passive or active mode data connection. Possible values are: • X'00': Active using default IP and port • X'01': Active using PORT • X'03': Passive using PASV • X'04': Passive using EPSV
125	Reserved	3	Reserved
128	SMF119FT_FCICConnID	4	TCP connection ID of FTP control connection (V1R8 and later)
132	SMF119FT_FCIDConnID	4	TCP connection ID of FTP data connection (V1R8 and later)

Table 72 describes the FTP client associated data set name section, which is physically located after the FTP client transfer initialization section. This section is identical to the one present in the transfer completion record.

Table 72. FTP client associated data set name section

Offset	Name	Length	Description
0	SMF119FT_FCFileName	<i>n</i>	MVS data set or z/OS UNIX file name associated with a file transfer operation. Use the data set type field information in the FTP client transfer initialization section to determine the type of file name that is represented by this value.

Table 73 describes the FTP client SOCKS section, which is present only if the connection passes through a SOCKS server.

Table 73. FTP client SOCKS section

Offset	Name	Length	Description
0	SMF119FT_FCCIP	16	SOCKS server IP address
16	SMF119FT_FCCPort	2	SOCKS Server port number
18	SMF119FT_FCCProt	1	SOCKS protocol version. Possible values are: • X'01': SOCKS Version 4 • X'02': SOCKS Version 5

Table 74 describes the FTP client security section.

Table 74. FTP client security section

Offset	Name	Length	Format	Description
0 (X'0')	SMF119FT_FCMechanism	1	EBCDIC	Protection Mechanism. Possible values are: • N: None • T: TLS • G: GSSAPI • A: AT-TLS
1 (X'1')	SMF119FT_FCCProtect	1	EBCDIC	Control Connection Protection Level. Possible values are: • N: None • C: Clear • S: Safe • P: Private
2 (X'2')	SMF119FT_FCDProtect	1	EBCDIC	Data Connection Protection Level. Possible values are: • N: None • C: Clear • S: Safe • P: Private
3 (X'3')	SMF119FT_FCLoginMech	1	EBCDIC	Login Method. Possible values are: • U: Undefined; the login method is not defined for the client. • P: Password • C: Certificate
4 (X'4')	SMF119FT_FCProtoLevel	8	EBCDIC	Protocol level (present only if Protocol Mechanism is TLSor AT-TLS). Possible values are: • SSLV2 • SSLV3 • TLSV1

Table 74. FTP client security section (continued)

Offset	Name	Length	Format	Description
12 (X'C')	SMF119FT_FCCipherSpec	20	EBCDIC	Cipher Specification (present only if Protocol Mechanism is TLS or AT-TLS). Possible values when Protocol Level is SSLV2 are: • RC4 US • RC4 Export • RC2 US • RC2 Export • DES 56-bit • Triple-DES US Possible values when Protocol Level is SSLV3 or TLSV1 are: • SSL_NULL_MD5 • SSL_NULL_SHA • SSL_RC4_MD5_EX • SSL_RC4_MD5 • SSL_RC4_SHA • SSL_RC2_MD5_EX • SSL_DES_SHA • SSL_3DES_SHA • SSL_AES_128_SHA • SSL_AES_256_SHA
32 (X'20')	SMF119FT_FCProtBuffSize	4	Binary	Negotiated protection buffer size
36(X'24')	SMF119FT_FCCipher	2	EBCDIC	Hexadecimal value of the Cipher Specification (present only if the Protocol Mechanism value is TLS or AT-TLS).

Table 75 describes the FTP client user name section.

Table 75. FTP client user name section

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FCUserID	<i>n</i>	EBCDIC	User name or user ID used to log into the FTP server.

FTP client login failure record

Table 76 describes the FTP client login failure self-defining section of the SMF record.

Table 76. FTP client login failure self-defining section

Offset	Name	Length	Format	Description
0 (x'0')	Standard SMF header	24	N/A	Standard SMF header; subtype is 102 (X'66')
Self-defining section				
24 (x'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record

Table 76. FTP client login failure self-defining section (continued)

Offset	Name	Length	Format	Description
26 (x'1A')	Reserved	2	N/A	Reserved
28 (x'1C')	SMF119IDOff	4	Binary	Offset to TCP/IP identification section *
32 (x'20')	SMF119IDLen	2	Binary	Length of TCP/IP identification section *
34 (x'22')	SMF119IDNum	2	Binary	Number of TCP/IP identification sections *
36 (x'24')	SMF119S1Off	4	Binary	Offset to FTP client login failure section
40 (x'28')	SMF119S1Len	2	Binary	Length of FTP client login failure section
42 (x'2A')	SMF119S1Num	2	Binary	Number of FTP client login failure sections
44 (x'2C')	SMF119S2Off	4	Binary	Offset to FTP client SOCKS section
48 (x'30')	SMF119S2Len	2	Binary	Length of FTP client SOCKS section
50 (x'32')	SMF119S2Num	2	Binary	Number of FTP client SOCKS sections
52 (x'34')	SMF119S3Off	4	Binary	Offset to FTP client Security section
56 (x'38')	SMF119S3Len	2	Binary	Length of FTP client Security section
58 (x'3A')	SMF119S3Num	2	Binary	Number of FTP client Security sections
60 (x'3C')	SMF119S4Off	4	Binary	Offset to FTP client user name section (V1R8 and later)
64 (x'40')	SMF119S4Len	2	Binary	Length of FTP client user name section (V1R8 and later)
66 (x'42')	SMF119S4Num	2	Binary	Number of FTP client user name sections (V1R8 and later)
* See Table 61 on page 545 for the contents of the TCP/IP identification section.				

Table 77 shows the client login failure session section, which follows the TCP/IP identification section.

Table 77. Client login failure session section

Offset	Name	Length	Format	Description
0(x'0')	SMF119FT_FCLRIP	16	Binary	Remote IP address (server)
16(x'10')	SMF119FT_FCLLIP	16	Binary	Local IP address (client)
32(x'20')	SMF119FT_FCLRPort	2	Binary	Remote port number (server)
34(x'22')	SMF119FT_FCLLPort	2	Binary	Local port number (client)
36(x'24')	SMF119FT_FCLUserID	8	EBCDIC	Local user ID

Table 77. Client login failure session section (continued)

Offset	Name	Length	Format	Description
44(x'2C')	SMF119FT_FCLReason	4	Binary	Login failure reason. The reason is a Client Error Code as documented in FTP Client Error Codes in <i>z/OS Communications Server: IP User's Guide and Commands</i>. Following are the client error codes most likely for login failures. X'0A'FTP_SESSION_ERROR Socket, send, or receive error. X'0B' FTP_LOGIN_FAILED User ID, password, or account information is not valid. X'11' FTP_AUTHENTICATION Security authentication or negotiation failed; incorrect specification of security keywords.
48 (x'30')	SMF119FT_FCLConnID	4	Binary	TCP connection ID of FTP control connection

Table 78 describes the FTP client SOCKS section, which is present only if the connection passes through a SOCKS server.

Table 78. FTP client SOCKS section

Offset	Name	Length	Description
0	SMF119FT_FCLIP	16	SOCKS server IP address
16	SMF119FT_FCLPort	2	SOCKS server port number
18	SMF119FT_FCLProt	1	SOCKS protocol version. Possible values are: • X'01': SOCKS Version 4 • X'02': SOCKS Version 5

Table 79 defines the FTP client login failure security section.

Table 79. FTP client login failure security section

Offset	Name	Length	Format	Description
0 (X'0')	SMF119FT_FCLMechanism	1	EBCDIC	Protection Mechanism. Possible values are: • N: None • T: TLS • G: GSSAPI • A: AT-TLS

Table 79. FTP client login failure security section (continued)

Offset	Name	Length	Format	Description
1 (X'1')	SMF119FT_FCLCProtect	1	EBCDIC	Control Connection Protection Level. Possible values are: • N: None • C: Clear • S: Safe • P: Private
2 (X'2')	SMF119FT_FCLDProtect	1	EBCDIC	Data Connection Protection Level. Possible values are: • N: None • C: Clear • S: Safe • P: Private
3 (X'3')	SMF119FT_FCLLoginMech	1	EBCDIC	Login Method. Possible values are: • U: Undefined; the login method is not defined for the client. • P: Password • C: Certificate
4 (X'4')	SMF119FT_FCLProtoLevel	8	EBCDIC	Protocol level (present only if Protocol Mechanism is TLS or AT-TLS). Possible values are: • SSLV2 • SSLV3 • TLSV1

Table 79. FTP client login failure security section (continued)

Offset	Name	Length	Format	Description
12 (X'C')	SMF119FT_FCLCipherSpec	20	EBCDIC	Cipher specification (present only if Protocol Mechanism is TLS or AT-TLS). Possible values when Protocol Level is SSLV2 are: • RC4 US • RC4 Export • RC2 US • RC2 Export • DES 56-bit • Triple-DES US Possible values when Protocol Level is SSLV3 or TLSV1 are: • SSL_NULL_MD5 • SSL_NULL_SHA • SSL_RC4_MD5_EX • SSL_RC4_MD5 • SSL_RC4_SHA • SSL_RC2_MD5_EX • SSL_DES_SHA • SSL_3DES_SHA • SSL_AES_128_SHA • SSL_AES_256_SHA
32 (X'20')	SMF119FT_FCLProtBuffSize	4	Binary	Negotiated protection buffer size
36(xX'24')	SMF119FT_FCLCipher	2	EBCDIC	Hexadecimal value of Cipher Specification (present only if Protocol Mechanism is TLS or AT-TLS).

Table 80 shows the FTP client user name section.

Table 80. FTP client user name section

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FCLUserID	<i>n</i>	EBCDIC	User name or user ID used to log on to the FTP server

FTP client session record

Table 81 shows the FTP client session record self-defining section.

Table 81. FTP client session record self-defining section

Offset	Name	Length	Format	Description
0 (x'0')	Standard SMF header	24	N/A	Standard SMF header; subtype is 103 (x'67')
Self-defining section				
24 (x'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record
26 (x'1A')	Reserved	2	N/A	Reserved

Table 81. FTP client session record self-defining section (continued)

Offset	Name	Length	Format	Description
28 (x'1C')	SMF119IDOff	4	Binary	Offset to TCP/IP identification section *
32 (x'20')	SMF119IDLen	2	Binary	Length of TCP/IP identification section *
34 (x'22')	SMF119IDNum	2	Binary	Number of TCP/IP identification sections *
36 (x'24')	SMF119S1Off	4	Binary	Offset to FTP client session section
40 (x'28')	SMF119S1Len	2	Binary	Length of FTP client session section
42 (x'2A')	SMF119S1Num	2	Binary	Number of FTP client session sections
44 (x'2C')	SMF119S2Off	4	Binary	Offset to FTP client SOCKS section
48 (X'32)	SMF119S2Len	2	Binary	Length of FTP client SOCKS section
50 (X'34')	SMF119S2Num	2	Binary	Number of FTP client SOCKS sections
52 (x'34')	SMF119S3Off	4	Binary	Offset to FTP client security section
56 (x'38')	SMF119S3Len	2	Binary	Length of FTP client security section
58 (x'3A')	SMF119S3Num	2	Binary	Number of FTP client security sections
60 (x'3C')	SMF119S4Off	4	Binary	Offset to FTP client session user name section
64 (x'40')	SMF119S4Len	2	Binary	Length of FTP client session user name section
66 (x'42')	SMF119S4Num	2	Binary	Number of FTP client session user name section
*See Table 61 on page 545 for the contents of the TCP/IP identification section.				

Table 82 shows the FTP client session section.

Table 82. FTP client session section

Offset	Name	Length	Format	Description
0 (X'0')	SMF119FT_FCNRIP	16	Binary	Remote IP address (server)
16 (X'10')	SMF119FT_FCNLIP	16	Binary	Local IP address (client)
32 (X'20')	SMF119FT_FCNRPort	2	Binary	Remote port number (server)
34 (X'22')	SMF119FT_FCNLPort	2	Binary	Local port number (Client)
36 (X'24')	SMF119FT_FCNUserID	8	EBCDIC	Local User ID
44 (X'2C')	SMF119FT_FCNReason	4	Binary	Session end reason. The reason is a client error code as documented in FTP Client Error Codes in z/OS Communications Server: IP User's Guide and Commands. If no error occurred, the value of this field is 0. This field is defined only when the value of the SMF119FT_FCNEvent field is T.

Table 82. FTP client session section (continued)

Offset	Name	Length	Format	Description
48 (X'30')	SMF119FT_FCNEvent	1	EBCDIC	- I: Session has started; client is logged into the server - T: Session has ended
49 (X'31')	Reserved	3	Binary	Reserved
52 (X'34')	SMF119FT_FCNSTime	4	Binary	Session start time, in one hundredths of a second, since midnight [using Coordinated Universal Time (UTC)].
56 (X'38')	SMF119FT_FCNSDate	4	Binary	Session start date (format: <i>0cyydddF</i>). If the date is not available, the value specified is X'0000000F'.
60 (X'3C')	SMF119FT_FCNETime	4	Binary	Session end time, in one hundredths of a second, since midnight [using Coordinated Universal Time (UTC)]. This field is defined only when the value of SMF119FT_FCNEvent is T.
64 (X'40')	SMF119FT_FCNEDate	4	Binary	Session end date (format: <i>0cyydddF</i>). If the date is not available, the value specified is X'0000000F'. This field is defined only when the value of SMF119FT_FCNEvent is T.
68 (X'44')	SMF119FT_FCNCConnID	4	Binary	TCP connection ID of FTP control connection

Table 83 shows the FTP client SOCKS section, which is present only if the connection passes through a SOCKS server.

Table 83. FTP client SOCKS section

Offset	Name	Length	Description
0	SMF119FT_FCNIP	16	SOCKS server IP address
16	SMF119FT_FCNPort	2	SOCKS server port number
18	SMF119FT_FCNProt	1	SOCKS protocol version. Possible values are: - X'01' SOCKS Version 4 - X'02' SOCKS Version 5

Table 84 shows the FTP client security section:

Table 84. FTP client security section

Offset	Name	Length	Format	Description
0 (X'0')	SMF119FT_FCNMechanism	1	EBCDIC	Protection Mechanism. Possible values are: • N: None • T: TLS • G: GSSAPI • A: AT-TLS
1 (X'1')	SMF119FT_FCNCProtect	1	EBCDIC	Control Connection Protection Level. Possible values are: • N: None • C: Clear • S: Safe • P: Private
2 (X'2')	SMF119FT_FCNDProtect	1	EBCDIC	Data Connection Protection Level. Possible values are: • N: None • C: Clear • S: Safe • P: Private
3 (X'3')	SMF119FT_FCNLoginMech	1	EBCDIC	Login Method. Possible values are: • P: Password • C: Certificate
4 (X'4')	SMF119FT_FCNProtoLevel	8	EBCDIC	Protocol level (present only if Protocol Mechanism value is TLS or AT-TLS). Possible values are: • SSLV2 • SSLV3 • TLSV1

Table 84. FTP client security section (continued)

Offset	Name	Length	Format	Description
12 (X'C')	SMF119FT_FCNCipherSpec	20	EBCDIC	Cipher specification (present only if Protocol Mechanism value is TLS or AT-TLS). Possible values when Protocol Level is SSLV2 are: • RC4 US • RC4 Export • RC2 US • RC2 Export • DES 56-bit • Triple-DES US Possible values when Protocol Level value is SSLV3 or TLSV1 are: • SSL_NULL_MD5 • SSL_NULL_SHA • SSL_RC4_MD5_EX • SSL_RC4_MD5 • SSL_RC4_SHA • SSL_RC2_MD5_EX • SSL_DES_SHA • SSL_3DES_SHA • SSL_AES_128_SHA • SSL_AES_256_SHA
32 (X'20')	SMF119FT_FCNSProtoBufSize	4	Binary	Negotiated protection buffer size
36 (X'24')	SMF119FT_FCNCipher	2	EBCDIC	Hexadecimal value of Cipher Specification (present only if Protocol Mechanism is TLS or AT-TLS).

Table 85 shows the FTP client session user name section.

Table 85. FTP client session user name section

Offset	Name	Length	Format	Description
0(x'0')	SMF119FT_FCNUserID	<i>n</i>	EBCDIC	User name or user ID used to log into the FTP server.

Packet and data trace formatting NMI

Records collected from the SYSTCPDA interface described in “Real-time TCP/IP network monitoring NMI” on page 528 can be formatted programmatically with the EZBCTAPI macro. This section describes how the EZBCTAPI interface can be used.

Overview

The interface to the formatter described in this topic provides a means for network applications to format packet and data trace records. An application program can capture a copy of the packet and data trace buffers using the Network Management interface for TCP/IP real-time packet and data tracing (SYSTCPDA), which is described in “Real-time TCP/IP network monitoring NMI” on page 528.

Trace records are laid out in the trace buffer as a series of Component Trace Entries (CTEs). Each CTE contains one trace record. The format identification field (CteFmtId) describes the layout of data in the trace record. Types 1, 2 and 3 contain a header (GTCNTL) that is described by the EZBCTHDR macro (or the EZBYCTHH header). Types 4, 5, and 6 contain a header (PTHDR_T) described by the EZBYPTHA macro (or the EZBYPTHH header). The following table depicts the layout of the various records.

CteFmtId	Description	Header	IP Header	Protocol	Data	V1R4	V1R5	V1R6+
1	Packet Trace	GTCNTL	IPv4	variable	variable	Y	N	N
2	X25 Trace	GTCNTL	IPv4	variable	variable	N	N	N
3	Data Trace	GTCNTL	N/A	N/A	variable	Y (EE only)*	Y (EE only)*	N
4	Packet Trace	PTHDR_T	IPv4	variable	variable	N/A	Y	Y
4	Packet Trace	PTHDR_T	IPv6	variable	variable	Y	Y	Y
5	Data Trace	PTHDR_T	N/A	N/A	variable	Y	Y	Y
6	EE Trace*	PTHDR_T	N/A	N/A	variable	N	N	Y
* EE stands for Enterprise Extender. Read about Enterprise Extender in the Enterprise Extender connections information in the <i>z/OS Communications Server: SNA Network Implementation Guide</i> .								

The ABBREV value of the PKTTRACE or DATTRACE command determines the amount of data available. The layout of CTEs in the 64 KB buffer is shown in the following figure.

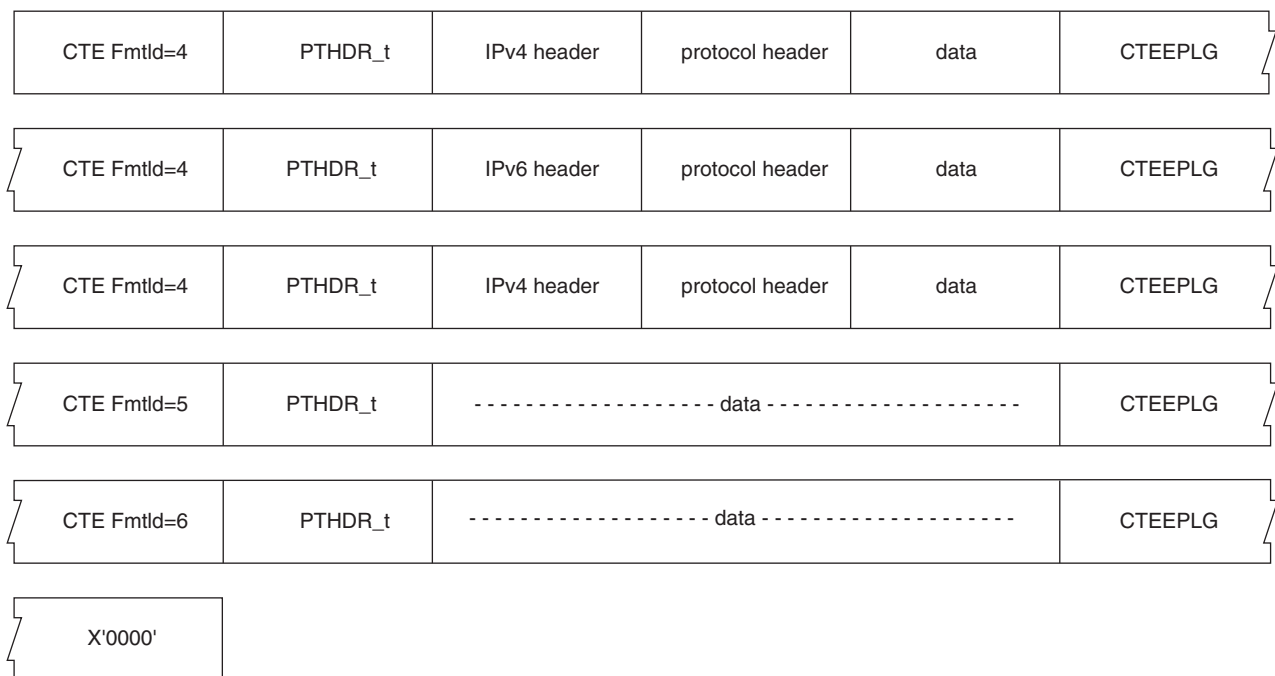


Figure 31. CTE layout

Configuration and enablement

There is no formal configuration required to enable this interface.

EZBCTAPI Network Management interface for formatting packet trace records

The EZBCTAPI macro accepts parameters to format component trace records from the TCP/IP packet trace and data trace. The data is formatted in the same fashion as is done using the IBM-provided packet trace and data trace formatters that are available with the IPCS CTRACE command. Note however that this interface does not require an IPCS environment to be active.

Requirement: High Level Assembler Language, Version 1 Release 4 or later is required to use this macro.

The EZBCTAPI macro enables users to pass component trace records to the format routine for processing and capture the formatted output text. There are several functions performed by the macro:

- **SETUP** - Define the formatting environment with the various parameters.
- **FORMAT** - Pass a record to the formatting interface.
- **TERM** - Delete the formatting environment allowing final output to be shown.
- **QUIT** - Delete the formatting environment without any final output. Summary and statistical reports created at the end of SYSTCPDA processing will not be formatted. This request should be used for quick termination of the interface when no further output is desired.

The header files and macros are described in the following table.

Header files for C/C++ programs	Macros for Assembler programs	Contents
N/A	EZBCTAPI	Used to format the records created by the SYSTCPDA interfaces.
EZBYCTHH	EZBCTHDR	Packet trace header describing the TCP/IP packet for types 1, 2, and 3 trace records.
EZBYPTHH	EZBYPTHA	Packet trace header describing the TCP/IP packets for types 4, 5, and 6 trace records.
N/A	EZBYPTO	Describes packet trace options for the formatter.

These header files and macros are shipped in the SEZANMAC data set. This data set must be available in the concatenation when compiling or assembling a part that makes use of these definitions.

Requirements

Minimum authorization:	Problem state and any PSW key
Dispatchable unit mode:	Task
Cross memory mode:	PASN=HASN=SASN
AMODE:	31-bit
ASC mode:	Primary
Interrupt status:	Enabled for I/O and external interrupts
Locks:	No locks held
Control parameters:	Must be addressable in the primary address space and have a storage key that matches the PSW key

Format

name	name: Symbol. Begin name in column 1.
	One or more blanks must precede EZBCTAPI.
EZBCTAPI	One or more blanks must follow EZBCTAPI.
SETUP	
FORMAT	
TERM	
QUIT	
,WORKAREA=workarea	workarea: RX-type address or register (2) - (12).
,API=epaaddr	epaaddr: RX-type address or register (2) - (12).
,COMP=name	name: RX-type address or register (2) - (12).
,CTE=record	record: RX-type address or register (2) - (12).
,LDT0=stcktime	stcktime: RX-type address or register (2) - (12).
,LSO=stcktime	stcktime: RX-type address or register (2) - (12).
,MAXLINE=number	number: RX-type address or register (2) - (12).

,NMCTF=epaaddr	epaaddr: RX-type address or register (2) - (12).
,OBTAIN=epaaddr	epaaddr: RX-type address or register (2) - (12).
,OPTIONS=options	options: RX-type address or register (2) - (12).
,PRTSRV=epaaddr	epaaddr: RX-type address or register (2) - (12).
,RELEASE=epaaddr	epaaddr: RX-type address or register (2) - (12).
,RETCODE=epaaddr	epaaddr: RX-type address or register (2) - (12) or (15).
,REPORT=FULL	Default: REPORT=FULL
,REPORT=SHORT	
,REPORT=SUMMARY	
,REPORT=TALLY	
,RSNCODE=rsncode	rsncode: RX-type address or register (2) - (12) or (0).
,TABLE=name	name: RX-type address or register (2) - (12).
,TIME=GMT	Default: TIME=LOCAL
,TIME=LOCAL	
,USERTOKEN=token	token: RX-type address or register (2) - (12).
,MF=(L,list_addr)	list_addr: RX-type address or register (2) - (12).
,MF=(L,list_addr,attr)	Default: MF=(L,list_addr, 0D)
,MF=G	
,MF=(M,list_addr)	
,MF=(M(list_addr,COMPLETE)	
,MF=(E,list_addr)	
,MF=(E,list_addr,COMPLETE)	

Parameters

The parameters are explained in this section. First, select one of the four required parameters that define the function that the interface is to perform (SETUP, FORMAT, TERM, QUIT). Next, select the optional parameters that you need.

The required parameters are as follows:

SETUP

Initialize the interface by allocating and initializing control blocks and loading the component trace format table. Most of the other keywords can be specified to define the processing options.

FORMAT

Locate the specific entry in the format table and call the format routine. The CTE keyword identifies the record to be formatted.

TERM End the interface by calling the filter routine one last time to issue any final reports and release all the allocated resources.

QUIT End the interface by calling the filter routine one last time to release all the allocated resources acquired by the formatter.

The optional parameters are as follows:

,API=epaaddr

Specifies the location of a word that contains the location of the EZBCTAPI routine. Use this keyword in the SETUP call to pass the entry pointer

address to the interface. This might be useful to avoid the overhead of loading and deleting this reentrant interface module. If the API keyword is not used, then the EZBCTAPI routine will be loaded by the SETUP function and deleted by the TERM or QUIT function.

,COMP=name

Specifies the location of an 8-byte character field containing the name of the CTRACE component. If not specified, the component name of SYSTCPDA is used.

,CTE=record

Specifies the location of a component trace record. Used with the FORMAT function.

,LDTO=stcktime

Specifies the location of an 8-byte store clock field. This field is in units of STCK timer units. It contains the local date time offset. This field is used to convert STCK time stamps in the component trace records to local time. If not specified, the field CVTLDTO is used as the default.

,LSO=stcktime

Specifies the location of an 8-byte store clock field. This field is in units of STCK timer units. It contains the leap seconds time offset. This field is used to convert STCK time stamps in the component trace records to GMT time and local time. If not specified, the field CVTLISO is used as the default.

,MAXLINE=number

Specifies the location of a word that contains the maximum line width for formatted output. The minimum value is 60 and the maximum value is 250. The default value is 80.

,NMCTF=epaaddr

Specifies the location of a word that contains the location of the EZBNMCTF stub routine. This might be useful to avoid the overhead of loading and deleting this reentrant interface module. This keyword should be used on each invocation that will invoke the interface (MF=(E)). If the NMCTF keyword is not specified, then the EZBNMCTF routine will be called by the macro as an external reference and EZBNMCTF must be link-edited with the application program.

,OBTAIN=epaaddr

Specifies the location of a word that contains an entry point location of a routine used by the interface to obtain storage. The default is a routine that uses the STORAGE (OBTAIN) macro to obtain the storage from the operating system. If the OBTAIN keyword is specified then the RELEASE keyword must be specified. It is passed these pointers in a parameter list addressed by register 1:

- The work area
- The 4-word user token (see the USERTOKEN definition later in this section)
- The word where the location of the obtained storage is returned
- The word with the length of the storage to be obtained

The following return codes are supported:

- 00: The storage was obtained. The location of the storage is returned.
- 04: The storage could not be obtained. The address is null.

Standard calling conventions are used to call the routine in the same environment when the EZBCTAPI interface was called.

,OPTIONS=options

Specifies the address of options to be passed to the packet trace formatter. These options are described by EZBYPTO data area. See "Passing options to the packet trace formatter" on page 581 for more information.

,PRTSRV=epaaddr

Specifies the location of a word that contains entry point location of a routine used by the interface and formatter to print lines of text and messages. It is passed these parameters in a parameter list addressed by register 1:

- The BLSUPPR2 parameter list.
- The 4-word user token (see the USERTOKEN definition later in this section).

The following return codes are supported from the print routine:

- 00: The line of text was printed.
- 04: The line was not printed and future output is to be suppressed.

Standard calling conventions are used to call the routine in the same environment when the EZBCTAPI interface was called.

To generate the BLSUPPR2 parameter list use the BLSUPPR2 macro:

```
PPR2 BLSUPPR2 DSECT=YES
```

The BLSUPPR2 macro is described in the *z/OS MVS Programming: Assembler Services Reference ABE-HSP*.

The following fields are defined as:

PPR2BUF	Location of buffer containing the data to be printed
PPR2BUFL	Length of data in the buffer to be printed
PPR2MSG	The buffer contains a message
PPR2OVIN	Overflow indentation level (0 for the first line, 2 for subsequent lines)

The print buffer is in the EBCDIC code page. The buffer has been translated to change unprintable characters to periods. The new line character (X'15') is located in each data line and the print function is called for each new line. Should the data buffer be larger than the MAXLINE value minus 1, then the print function will be called as many times as needed with the rest of the print line with PPR2OVIN set to 2.

,RELEASE=epaaddr

Specifies the location of a word that contains the entry point location of a routine used by the interface to release storage. The default is a routine that uses the STORAGE (RELEASE) macro to release the storage back to the operating system. If the RELEASE keyword is specified, then the OBTAIN keyword must be specified. It is passed these pointers in a parameter list addressed by register 1:

- The work area
- The 4-word user token (see the USERTOKEN definition later in this section)

- The word with the location of the storage to be released
- The word with the length of the storage to be obtained

The following return codes are supported:

- 00: The storage was released.
- 04: The storage could not be released.

Standard calling conventions are used to call the routine in the same environment when the EZBCTAPI interface was called.

,RETCODE=retcode

Specifies the location where the interface return code is stored. The return code is also in general purpose register (GPR) 15.

,REPORT=FULL

,REPORT=SHORT

,REPORT=SUMMARY

,REPORT=TALLY

Formats the report.

FULL Formats the IP protocol headers and packet data. This includes the component mnemonic, entry identifier, date and time, and a description of the trace record. FULL is the default report option.

SHORT

Formats the IP protocol headers. This includes the component mnemonic, entry identifier, date and time, and a description of the trace record.

SUMMARY

Requests one line per trace record. Key fields from each qualifying trace record will be printed following the date, time, and entry description.

TALLY

Requests a list of trace entry definitions for the component and counts how many times each trace entry occurred.

,RSNCODE=rsncode

Specifies the location where the interface reason code is stored. The reason code is also in GPR 0. EZBCTAPI provides a reason code if the return code is other than 0.

,TABLE=name

Specifies the location of the 8-character field that contains the name for the format table (EZBPTFM4) or two words. The first word contains zeros and the second word contains the entry point address of EZBPTFM4. If not specified or the name is not used, then the EZBPTFM4 table is loaded. This might be useful to avoid the overhead of loading and deleting this format table.

,TIME=GMT

,TIME=LOCAL

Specifies the conversion of the time field in the component trace records. The default is TIME=LOCAL.

GMT The time is shown as Greenwich Mean Time.

LOCAL

The time is shown as local time.

,USERTOKEN=token

Specifies the location of a 4-word field that is copied and passed to the print service routine and the storage functions. The default is four words of zeros.

,WORKAREA=workarea

The location of a 16 KB work area used by the interface for its control blocks, work area, and save areas. The work area will be cleared by the SETUP function. This work area must remain intact until the TERM or QUIT function is called. The work area cannot be shared across tasks. Specification is optional; if not specified, a 16 KB work area is obtained.

,MF=(L,list_addr)

,MF=(L,list_addr,attr)

Requests that a EZBCTAPI parameter list be defined. **list_addr** is the name assigned to the list. **attr** is an optional attribute used to define the parameter list. The default is 0D. No other keywords can be used with this macro format.

,MF=G

Requests that the EZBCTAPI_t parameter list description be generated. No other keywords can be used with this macro format.

,MF=(M,list_addr)

,MF=(M,list_addr,COMPLETE)

Request that the EZBCTAPI parameter list be modified. **COMPLETE** requests that the parameter list be set to binary zeros before any modifications.

,MF=(E,list_addr)

,MF=(E,list_addr,COMPLETE)

Requests that the EZBCTAPI parameter list be modified. **COMPLETE** requests that the parameter list be set to binary zeros before any modifications. In addition, for the SETUP function the EZBCTAPI interface program is loaded, and for the TERM and QUIT functions the interface program is deleted (see the **API** keyword to modify this behavior). The interface program is then called.

Restriction: **COMPLETE** does not apply to TERM and QUIT functions.

The following table shows supported functions and keyword combinations.

Keyword	Input/ Output	MF(E) SETUP	MF(E) FORMAT	MF(E) TERM	MF(E) QUIT	MF(M)	MF(L)	MF(G)
WORKAREA	I	X				X		
API	I	X		X	X	X		
COMP	I	X	X			X		
CTE	I		R			X		
LDTO	I	X				X		
LSO	I	X				X		
NMCTF	I	X	X	X	X			
MAXLINE	I	X				X		
OBTAIN	I	X				X		

Keyword	Input/ Output	MF(E) SETUP	MF(E) FORMAT	MF(E) TERM	MF(E) QUIT	MF(M)	MF(L)	MF(G)
OPTIONS	I	X				X		
PRTSRV	I	R				X		
RELEASE	I	X				X		
REPORT	I	X				X		
RETCODE	O	X	X	X	X			
RSNCODE	O	X	X	X	X			
TABLE	I	X				X		
TIME	I	X				X		
USERTOKEN	I	X				X		
Legend: • I: Input parameter • O: Output parameter • R: Required parameter • X: Optional parameter								

Input register information

Before issuing the EZBCTAPI macro, the caller must ensure that the following general purpose register (GPR) contains the specified information:

Register contents

13 The location of a 72-byte standard save area in the primary address space.

Before issuing the EZBCTAPI macro, the caller does not have to place any information into any access register (AR).

Output register information

When control returns to the caller, the general purpose registers (GPRs) contain:

Register contents

0 Reason code, if GPR 15 contains a nonzero return code; otherwise, used as a work register by the system

1 Used as a work register by the system

2-13 Unchanged

14 Used as a work register by the system

15 Return code

When control returns to the caller, the access registers (ARs) contain:

Register contents

0-1 Used as work registers by the system

2-13 Unchanged

14-15 Used as a work register by the system

Some callers depend on register contents remaining the same before and after issuing a service. If the system changes the contents of registers on which the caller depends, the caller must save them before issuing the service, and restore them after the system returns control.

ABEND codes

There are no ABEND codes.

Return and reason codes

When control returns from EZBCTAPI, GPR 15 (and retcode, if you coded RETCODE) contains one of the return codes shown in Table 86. GPR 0 (and rsncode, if you coded RSNCODE) might contain one of the reason codes shown in Table 86.

Table 86. EZBCTAPI return and reason codes

Hexadecimal return code (CtApi_IRtnCd)	Hexadecimal reason code (CtApi_IRsnCd)	Meaning
00	00	Function was successful.
04	See note	The FORMAT function was not successful.
04	10	The SETUP function was not done or did not complete.
04	11	The trace record is not the correct format.
04	18	The trace record could not be identified.
04	1B	The filter/analysis routine failed.
08	See note	The SETUP function was not successful.
08	01	The SETUP function has already initialized the interface.
08	02	Print callback function was not provided.
08	03	Unable to load format table.
08	04	Unable to allocate storage for tables.
08	05	Unable to load analysis/format exit.
0C	xx	Unknown function code xx.
10	See note	Unable to load the function interface.
10	04	The EZBCTAPI interface routine could not be found.
10	08	An error occurred loading the EZBCTAPI interface routine.
14	See note	Unable to obtain storage for a work area.
14	04	The program was not able to obtain storage for the work area.
18	xxxxxxx	The interface routine or the analysis routine abended; xxxxxx is the abend code.
Note: The first line of a new return code is a generic line about the return code.		

Table 87. EZBCTAPI formatter return and reason codes

Hexadecimal return code (CtApi_FRtnCd)	Hexadecimal reason code (CtApi_FRsnCd)	Meaning
00	N/A	Normal processing of the entry
04	N/A	Reread the records from the first
08	N/A	The current entry is bypassed
0C	N/A	No further calls to the format/analysis routine
10	N/A	Ending of the subcommand

These return codes are described in *z/OS MVS IPCS Customization* for a CTRACE formatter filter/analysis exit. The packet trace formatter uses only a return code of 0 or 8. The interface return code (CtApi_IRtnCd) is always 0 for formatter return codes of 0, 4, 8, and 12; otherwise, an interface return code of 4 is returned (see interface reason code X'1B').

Capturing trace records: This section describes the process of capturing trace records.

Before you begin: You need to have done the following:

1. Update the TCP/IP profile to allow trace data to be copied: NETMONITOR PKTTRCService.
2. Grant authority to an application program to capture trace data:
 - Make the program APF authorized, and
 - Define the user with BPX.SUPERUSER, or
 - Permit the user to access EZB.NETMGMT.sysname.tcpipname.SYSTCPDA

Perform the following steps to capture trace records.

1. Start the application program.

The application program does the following:

- a. Defines the format options in the EZBYPTO control block, passed to EZBCTAPI.
- b. Uses the EZBCTAPI macro to set up the packet trace formatter interface.
- c. Connects an AF_UNIX socket to the SYSTCPDA service (described in "Real-time TCP/IP network monitoring NMI" on page 528).
- d. Allocates a 64 KB buffer.
- e. In a loop, reads a record from the AF_UNIX socket. The first word of each record contains the length of the record. The record contains tokens that describe a TCP/IP trace buffer that contains data to be copied.
- f. Calls EZBTMIC1 to copy the TCP/IP trace buffer to the application 64 KB buffer.
- g. For a return value of zero or negative, reads the next record from the AF_UNIX socket.
The return value contains the amount of data moved into the buffer. The buffer contains a series of component trace entries (CTE). A CTE is described by the ITTCTE data area.
- h. Processes each CTE in the buffer by calling the format function of EZBCTAPI, passing the address of the CTE.

The length of each CTE is the unsigned halfword at the start of each CTE. A CTE with a length of zero indicates the end of the buffer. This last halfword of zeros is not included in the return value of the amount of data moved.

- i. Loops to read the next record from the socket.

2. Issue VARY TCPIP,,PKTTRACE or VARY TCPIP,,DATTRACE commands to collect the data of interest.

At termination, the application program frees the 64 KB buffer, closes the socket, and calls the TERM function of EZBCTAPI.

Performance implications

There are no performance implications.

Example of initializing the EZBCTAPI exit environment

```
*          COPY  EZBCTAPI
EZBCTSMP  CSECT
          SAVE  (14,12),,*
          LR     12,15                      SET A BASE REGISTER
          USING  EZBCTSMP,12
          LA     15,MAINSA                  CHAIN THE SAVE AREA
          ST     15,8(,13)
          ST     13,4(,15)
          LR     13,15
*/*****/
*/**          INITIALIZE THE OPTIONS          */
*/*****/
PTO        USING  EZBYPTO,APTO              MAP THE OPTIONS AREA
          XC      APTO,APTO                  ZERO THE OPTIONS FLAGS AND PTRS
          LA      0,EZBYPTO_SZ              SET LENGTH OF OPTIONS AREA
          STH     0,PTO.PTO_LENGTH
          LA      0,EZBYPTO_SZ-4
          STH     0,PTO.PTO_OFFSET
*          SET  FORMAT(DETAIL) SEGMENT REASSEM STATS(DETAIL)
          OI      PTO.PTO_FORMAT,L'PTO_FORMAT
          OI      PTO.PTO_FMTDTL,L'PTO_FMTDTL
          OI      PTO.PTO_STATS,L'PTO_STATS
          NI      PTO.PTO_STCSUM,255-L'PTO_STCSUM SET STAT(DETAIL)
          OI      PTO.PTO_REASM,L'PTO_REASM
          OI      PTO.PTO_SEGMENT,L'PTO_SEGMENT
*
*
          OPEN    (PRINTDCB,OUTPUT)          OPEN THE PRINT FILE
*
          STORAGE OBTAIN,LENGTH=CTAPI_WKSIZE,ADDR=(8)
*                                          GET STORAGE FOR ABDPL WORK AREA
*
* INITIALIZE THE EZBCTAPI PARAMETER LIST
          EZBCTAPI WORKAREA=(8),
                                COMP==CL8'SYSTCPDA',
                                PRTSRV==A(PRINTSRV),
                                OPTIONS=APTO,
                                REPORT=FULL,
                                TIME=LOCAL,
                                USERTOKEN=PRINTTKN,
                                MAXLINE==A(L'PRINTBUF-1),
*                                MF=(M,CTAPIL,COMPLETE)
* GET A BUFFER FOR READING BUFFERS
*
          STORAGE OBTAIN,LENGTH=64*1024
```

```

        ST      1,ABUFFER31
*
* SET UP THE FORMATTER INTERFACE
*
        EZBCTAPI SETUP,MF=(E,CTAPIL), SET UP THE INTERFACE      C
        RETCODE=RETCDE,RSNCODE=RETRSN
        LTR     15,15      DID THIS WORK
        BNZ     ERROR

*
* READ IN A TOKEN
*
LOOP1    DS      0H
        CALL    BPX1RED,(SOCKET,
                        ABUFFER,PRIMARYALET,LBUFTKN,
                        RETVAL,RETCDE,RETRSN),VL      C
        L       15,RETVL
        LTR     15,15
        BNP     EOF      CLOSE SOCKET AND EXIT

*
* READ IN DATA BUFFERS
*
        ST      15,LBUFTKN
        CALL    EZBTMIC1,(BUFTOKEN,LBUFTKN,RETVL,RETCDE,RETRSN)
        L       15,RETVL
        LTR     15,15      WAS DATA MOVED?
        BNZ     LOOP1      NO, GET NEXT ONE

*
        L       3,ABUFFER31      GET ADDRESS THE BUFFER
        USING   CTE,3      MAP THE BUFFERS

*
LOOP2    DS      0H
*
        LH      2,CTELENP      GET LENGTH OF THIS RECORD
        N       2,=X'0000FFFF' ALLOW UP TO 64K RECORDS
        LTR     2,2      IS THIS THE END
        BNP     LOOP1      YES, DO THE NEXT BUFFER
        EZBCTAPI FORMAT,CTE=CTE,
        MF=(E,CTAPIL)      C
        ALR     3,2      POINT TO THE NEXT CTE
        B       LOOP2      DO THE NEXT RECORD

*
EOF      DS      0H
        EZBCTAPI TERM,MF=(E,CTAPIL)
        STORAGE RELEASE,LENGTH=CTAPI_WKSIZE,ADDR=(8)
*
        CLOSE (PRINTDCB)      RELEASE STORAGE ABDPL WORK AREA
        L       13,4(13)
        RETURN (14,12),RC=0

*
*
ERROR    DS      0H
*
*
* DATA
*
        LTORG
MAINSA   DC      18A(0)
        EZBCTAPI MF=(L,CTAPIL)
        EZBCTAPI MF=G
SOCKET   DC      F'0'      FILE SYSTEM SOCKET NUMBER
ABUFFER  DC      A(BUFTOKEN)
PRIMARYALET DC      F'0'
LBUFTKN  DS      F      LENGTH OF BUFFER TOKEN
BUFTOKEN DS      CL64      A BUFFER TOKEN
RETVL    DS      F
RETCDE   DS      F
RETRSN   DS      F

```

BUFPTR	DC	0F	
	DC	A(0,0)	ALET, HI64BITS
ABUFFER31	DC	A(0)	ADDRESS OF THE BUFFER
*			
APTO	DS	CL(EZBYPTO_SZ)	SPACE FOR THE OPTIONS
*			
PRINTTKN	DC	0F	TOKEN FOR PRINT SERVICE
	DC	A(PRINTDCB)	
	DC	A(PRINTSA)	
	DC	A(PRINTBUF)	
	DC	A(0)	
*			
PRINTDCB	DCB	DDNAME=SYSPRINT,DSORG=PS,MACRF=PM, RECFM=FBA,LRECL=133	C
*			
PRINTBUF	DS	0CL133	A PRINT BUFFER
PRINTCC	DC	C' '	
PRINTDAT	DC	CL132' '	
*			
PRINTSA	DC	18A(0)	A SAVE AREA FOR PRINT SERVICE
*			
*			
*			
PRINTSRV	EJECT		
	CSECT		
	SAVE	(14,12),,*	SAVE REGISTERS
	LR	12,15	SET BASE REGISTER
	USING	PRINTSRV,12	MAP IT
	LR	2,1	COPY PARM LIST POINTER
	USING	PLIST,2	
	LM	2,3,PLIST	GET PLIST POINTERS
	USING	PPR2,2	
	USING	PTKN,3	
	LM	4,6,PTKN	GET POINTERS TO STUFF
*		4 ==> DCB	
*		5 ==> SAVE AREA	
*		6 ==> PRINT BUFFER	
	USING	PBUF,6	.
	ST	5,8(,13)	CHAIN THE SAVE AREAS
	ST	13,4(,5)	.
	LR	13,5	.
*			
	L	7,PPR2BUF	GET ADDRESS OF THE BUFFER
	L	8,PPR2BUFL	GET ITS LENGTH
*			
	MVI	PBUFLNE-1,C' '	BLANK IT ALL OUT
	MVC	PBUFLNE,PBUFLNE-1	.
	LTR	8,8	IS THERE A LINE
	BNP	PSRV0001	NO, JUST DO A BLANK LINE
	BCTR	8,0	TO EXECUTE LENGTH
	EX	8,COPYLINE	COPY LINE OF TEXT
*			
PSRV0001	DS	0H	
*	L	4,PTKNDCB	GET ADDRESS OF PRINT DCB
	PUT	(4),PBUF	PRINT THE LINE OF TEXT
	L	13,4(,13)	UNCHAIN THE SAVE AREAS
	RETURN	(14,12),RC=0	RETURN TO CALLER
*			
			INDICATE PRINT WAS OK
COPYLINE	MVC	PBUFLNE(0),0(7)	COPY THE PRINT LINE
*			
*			
PPR2	BLSUPPR2	DSECT=YES	PPR2 PARAMETER LIST
PLIST	DSECT	,	
PLPR2	DS	A	POINTER TO PPR2 PARM LIST
PLTKN	DS	A	POINTER TO OUR TOKEN
*			
PTKN	DSECT	,	OUR TOKEN

PTKNDCB	DS	A	POINTER TO THE DCB
PTKNSA	DS	A	POINTER TO SAVE AREA
PTKNBUF	DS	A	POINTER TO BUFFER AREA
*			
PBUF	DSECT	,	OUTPUT BUFFER
PBUFCC	DS	C	CARRIAGE CONTROL
PBUFLNE	DS	CL132	OUTPUT LINE
*			
	ITTCTE	,	
	EZBYPTO		COPY FORMAT OPTIONS
	END		

Passing options to the packet trace formatter

The EZBYPTO macro describes a data area that can be passed using the EZBCTAPI OPTIONS keyword. This data area contains flags, values, and pointers that describe packet trace formatter options. Table 88 shows the option and field settings required to select the option.

These same options are available through the SYSTCPDA CTRACE formatter. You can find a detailed explanation in the packet trace (SYSTCPDA) for TCP/IP stacks information in the *z/OS Communications Server: IP Diagnosis Guide*.

Table 88. Available EZBYPTO options

Option	Field setting
ASCII	Pto_Dump=1;Pto_DmpCd=PtoAscii;
BASIC(DETAIL)	Pto_Basic=1;Pto_BasDtl=1;
BASIC(SUMMARY)	Pto_Basic=1;Pto_BasDtl=0;
BOTH	Pto_Dump=1;Pto_DmpCd=PtoBoth;
CLEANUP(nnnnn)	Pto_Cleanup=1;Pto_GcIntvl=nnnnn;
DUMP	Pto_Dump=1;
DUMP(nnnnn)	Pto_Dump=1;Pto_MaxDmp=nnnnn;
EBCDIC	Pto_Dump=1;Pto_DmpCd=PtoEbcDic;
FORMAT(DETAIL)	Pto_Format=1;Pto_FmtDtl=1;
FORMAT(SUMMARY)	Pto_Format=1;Pto_FmtDtl=0;
FULL	Pto_Dump=1,Pto_Format=1,Pro_FmtDtl=1;
HEX	Pto_Dump=1;Pto_DmpCd=PtoHex;
INTERFACE	Pto_Links@=Addr(list),Pto_Links#=nn
IPADDR(list)	Pto_Addr@=Addr(list);Pto_Addr#=nn
PORT(list)	Pto_Port@=Addr(list);Pto_Port#=nn
REASSEMBLY(nnnnn)	Pto_ReAsm=1;Pto_MaxRsm=nnnnn
REASSEMBLY(DETAIL)	Pto_ReAsm=1;Pto_RsmSum=0
REASSEMBLY(SUMMARY)	Pto_ReAsm=1;Pto_RsmSum=1
NOREASSEMBLY	Pto_ReAsm=0;
SEGMENT	Pto_Segment=1;
NOSEGMENT	Pto_Segment=0;
SESSION(DETAIL)	Pto_SesRpt=Pto_SesDetail; Pto_Session=1;
SESSION(SUMMARY)	Pto_SesRpt=Pto_SesSummary; Pto_Session=1;
SESSION(STATE)	Pto_SesRpt=Pto_SesState; Pto_Session=1;
STATISTICS(DETAIL)	Pto_Stats=1;Pto_StcSum=0;

Table 88. Available EZBYPTO options (continued)

Option	Field setting
STATISTICS(SUMMARY)	Pto_Stats=1;Pto_StcSum=1;
STREAMS(nnn)	Pto_Streams=1;Pto_StrmBuf=nnn
STREAMS(DETAIL)	Pto_Streams=1;Pto_StmSum=0;
STREAMS(SUMMARY)	Pto_Streams=1;Pto_StmSum=1;
SUMMARY	Pto_Summary=1;
TALLY	Pto_Stats=1;Pto_StcSum=0;

Notes:

1. A packet might span multiple trace records. When segmented records are encountered, the SEGMENT option re-creates the packet as a single trace record. The packet is not used until the last trace segment record is passed to the formatter. Until that time, the packet is saved in a temporary buffer. Use the NOSEGMENT option to prevent this. The CLEANUP value can be used to free the temporary buffers for segments that will not be completed. The QUIT or TERM function frees all unprocessed segments.
2. When the NOSEGMENT option is used, only the first segment has the IP header and protocol headers.
3. A packet might be fragmented. When you specify the REASSEMBLY option, the formatter saves the fragments in a temporary buffer until all the fragments have been processed to recreate the original complete packet. The packet is not used until the last trace record is passed to the formatter. The CLEANUP value frees temporary buffers that have not completed, for reassembly. The QUIT or TERM function frees all unprocessed fragments.
4. Use the NOREASSEMBLY option to prevent this saving of records.
5. If the CLEANUP value is zero, then the temporary buffers are not released until the QUIT or TERM function.
6. You can use the EZBYPTO options control block to request multiple reports.
7. Use of the EZBCTAPI TERM function creates the SESSION, STATISTICS, and STREAMS reports.
8. The EZBYPTO data area is not processed by the SETUP function call. The values in the data area and values pointed from the data area must remain intact until after the first FORMAT, TERM, or QUIT function call.

Using the formatter

There are two ways of passing the formatter truncated records so that trace records contain only headers.

- Use the ABBREV keyword of the PKTTRACE command to truncate traced records. No matter the value of ABBREV, the record will always contain the IP header and protocol header.
- Shorten the data passed to the formatter. Use these steps:
 1. Determine whether the trace record is the first segment of packet. The sequence number field of the header (PTH_SeqNum) will be zero. The record contains the IP header and protocol header (if any). Otherwise the record just contains data.
 2. Set the CTELENP field (the first halfword of a trace record) to the smaller of CTELENP or the sum of the size of the CTEFDATA field, the size of the PTH_HDR field, the size of the IP header and the size of the protocol header.

3. Set the PTO_SEGMENT flag to zero. The length also includes the 2-byte length field CTELENE.

Records passed to the formatter must always contain at least the ITTCTE, PTHDR_t, the IP header and the protocol header.

TCP/IP callable NMI (EZBNMIFR)

z/OS Communications Server provides a high-speed low-overhead callable programming interface for network management applications to access data related to the TCP/IP stack. Use the EZBNMIFR network management interface to perform the following functions:

- Monitor TCP or UDP endpoints
- Monitor TCP/IP storage
- Drop one or more TCP connections
- Drop one or more UDP endpoint
- Monitor TN3270E Telnet server performance

This section describes the details for invoking the EZBNMIFR interface with the defined input parameters and for processing the output it provides. The following topics are addressed:

- “Overview”
- “Configuration and enablement” on page 584
- “EZBNMIFR — TCP/IP network management request” on page 584
- “TCP/IP network management interface request format” on page 587
- “TCP/IP network management interface response format” on page 593
- “Request and response data structures” on page 596
- “Examples” on page 596

You can invoke the EZBNMIFR interface to perform two types of requests: poll-type requests and action-type requests.

Poll-type requests

The following poll-type requests enable you to obtain the following type of information from the TCP/IP stack:

- Active TCP connections
- Active UDP endpoints
- Active TCP listeners
- TCP/IP storage utilization
- TN3270E Telnet server monitor groups
- TN3270E Telnet server connection performance data

Action-type requests

The following action-type request tells the TCP/IP stack to perform an action:

- Drop TCP connections or UDP endpoints

Overview

EZBNMIFR is a callable polling-type interface that returns the status of connections, listeners, and endpoints at a given point in time, as opposed to an

asynchronous interface that indicates all state changes for these connections, listeners, and endpoints. The caller can specify filters that limit the returned data to a specific set of information.

Tip: Dropping a TCP connection or UDP endpoint does not close the associated socket. The application that owns the associated socket must close the socket.

Requests to drop TCP connections or UDP endpoints are requests for an action. The caller must specify the connection identifier, local IP address, local port, remote IP address and remote port for the TCP connections, or local IP address and local port for UDP endpoints to drop. The remote IP address, remote port, and connection identifier are ignored for UDP endpoints.

The callable interface that drops a TCP connection or UDP endpoint is similar to the Netstat DDrop/-D command this is available from TSO, z/OS UNIX, and MVS operator commands. The major difference is that the callable interface can drop multiple connections at a time. The caller must specify the connection identifier, local IP address, local port, remote IP address, and remote port for TCP and local IP address and local port for UDP endpoints

Configuration and enablement

There is no configuration required to enable this interface when it is used as a poll-type interface.

Authorization to drop a TCP connection or UDP endpoint is identical to the TSO, z/OS UNIX, and MVS Operator Netstat Drop commands. An application can use this interface to drop a TCP connection or UDP endpoint only if the MVS.VARY.TCPIP.DROP security profile in the OPERCMDS class is defined and the user ID associated with the application is permitted to this resource. Therefore, if a user ID is already permitted to issue Netstat DDrop/-D, the user ID can use the EZBNMIFR callable interface to drop a TCP connection or UDP endpoint.

EZBNMIFR — TCP/IP network management request

EZBNMIFR requests network management data from TCP/IP or the TN3270E Telnet server.

Requirements

Minimum authorization:	Supervisor state, executing in system key, APF-authorized, or superuser
Dispatchable unit mode:	Task or SRB
Cross memory mode:	PASN=SASN=HASN
AMODE:	31-bit or 64-bit
ASC mode:	Primary
Interrupt status:	Enabled for I/O and external interrupts
Locks:	Not applicable
Control parameters:	Must reside in an addressable area in the primary address space and must be accessible using caller's execution key

Format

Invoke EZBNMIFR, as follows.

For C/C++ callers:

```

NWMServicesJobName,
    RequestResponseBuffer,
    &RequestResponseBufferAlet,
    &RequestResponseBufferLength,
    &ReturnValue,
    &ReturnCode,
    &ReasonCode);

```

For Assembler callers:

```

CALL EZBNMIFR,JobName,
    RequestResponseBuffer,
    RequestResponseBufferAlet,
    RequestResponseBufferLength,
    ReturnValue,
    ReturnCode,
    ReasonCode)

```

Parameters

TcpipJobName or TnJobName

Supplied and returned parameter.

Type:	Character
Length:	Doubleword

The name of an 8-character field that contains the EBCDIC job name of the target TCP/IP stack or TN3270E Telnet server. If the first character of the supplied job name is an asterisk (*), the call is made to the first active TCP/IP stack and its job name is returned.

Tip: You can use the GetTCPLISTENERS request to find all active TN3270E Telnet servers. The returned listener list indicates which application names are TN3270E Telnet server-type applications. Use the results from all TCP/IP stacks to determine which TN3270E Telnet servers have affinity to a particular stack and which servers have no affinity.

RequestResponseBuffer

Supplied parameter

Type:	Character
Length:	Variable

The name of the storage area that contains an input request. The input request must be in the format of a request header (NWMHeader) as defined in the EZBNMRHC header file. On successful completion of the request, the storage will contain an output response in the same format.

RequestResponseBufferAlet

Supplied parameter.

Type:	Integer
Length:	Fullword

The name of a fullword that contains the ALET of *RequestResponseBuffer*. If a nonzero ALET is specified, the ALET must represent a valid entry in the caller's dispatchable unit access list (DU-AL).

RequestResponseBufferLength

Supplied parameter.

Type: Integer
Length: Fullword

The name of a fullword that contains the length of the request/response buffer.

ReturnValue

Returned parameter.

Type: Integer
Length: Fullword

The name of a fullword in which the EZBNMIFR service returns one of the following:

- 0 or positive, if the request is successful. A value greater than 0 indicates the number of output data bytes copied to the response buffer. See “TCP/IP network management interface response format” on page 593 for additional details on processing request completions.
- -1, if the request is not successful.

ReturnCode

Returned parameter.

Type: Integer
Length: Fullword

The name of a fullword in which the EZBNMIFR service stores the return code (errno). The EZBNMIFR service returns *ReturnCode* only if *ReturnValue* is -1.

ReasonCode

Returned parameter.

Type: Integer
Length: Fullword

The name of a fullword in which the EZBNMIFR service stores the reason code (errnojr). The EZBNMIFR service returns *ReasonCode* only if *ReturnValue* is -1. *ReasonCode* further qualifies the *ReturnCode* value.

The EZBNMIFR service sets the following return codes and reason codes:

ReturnValue	ReturnCode	ReasonCode	Meaning
0	0	0	The request was successful.
-1	ENOBUFFS	JRBufTooSmall	The request was not successful. The request/response buffer is too small to contain all of the requested information. Some of the requested information might be returned.

Return Value	Return Code	Reason Code	Meaning
-1	EACCES	JRSAFNotAuthorized	The request was not successful. The caller is not authorized. For the DropConnection request, this might be returned if the user is not permitted to the MVS.VARY.TCPIP.DROP security profile in the OPERCMDS class.
-1	EAGAIN	JRTCPNOTUP	The request was not successful. The target TCP/IP stack or TN3270E Telnet server was not active.
-1	EFAULT	JRReadUserStorageFailed	The request was not successful. A program check occurred while copying input parameters, or while copying input data from the request/response buffer.
-1	EFAULT	JRWriteUserStorageFailed	The request was not successful. A program check occurred while copying output parameters, or while copying output data to the request/response buffer.
-1	EINVAL	JRInvalidValue	The request was not successful. A value that is not valid was specified in the request/response header.
-1	ETCPERR	JRTcpError	The request was not successful. An unexpected error occurred.

Network Management applications can use any of the following methods to invoke the EZBNMIFR service:

- Issue a LOAD macro to obtain the EZBNMIFR service entry point address, and then CALL that address. The EZBNMIFR load module must reside in a linklist data set (for example, the SEZALOAD load library of TCP/IP), or in LPA.
- Issue a LINK macro to invoke the EZBNMIFR service. The EZBNMIFR load module must reside in a linklist data set (for example, the SEZALOAD load library of TCP/IP), or in LPA.
- Link-edit EZBNMIFR directly into the application load module, and then CALL the EZBNMIFR service. Include SYS1.CSSLIB(EZBNMIFR) in the application load module link-edit.
- For 64-bit C/C++ applications, link-edit the EZBNMIF4 program directly into the application load module, and then CALL the EZBNMIF4 service. Include SYS1.CSSLIB(EZBNMIF4) in the application load module link-edit.

TCP/IP network management interface request format

The following section describes the format and details of the poll-type requests provided with EZBNMIFR. A second section describing action-type requests follows.

Format and details for poll-type requests

- The following poll-type requests are provided by EZBNMIFR:
 - GetTCPListeners — obtain information about active TCP listeners
 - GetUDPTTable — obtain information about active UDP sockets

- GetConnectionDetail — obtain information about active TCP connections
- GetStorageStatistics — obtain information about TCP/IP storage utilization
- GetTnMonitorGroups — obtain information about TN3270E Telnet server monitor groups
- GetTnConnectionData — obtain information about TN3270E Telnet server connection performance data
- The general format of the request consists of the request header and the request section descriptors (triplets), which define the input data. A triplet describes the input filters and contains the offset, in bytes, of the request section relative to the beginning of the request buffer, the number of elements in the request section, and the length of an element in the request section.
- The request sections. The following types of request sections can be specified:
 - Filters (optional) — You can select a subset of TCP or UDP endpoints based on any combination of the items in the following table.

Filter item	Filter item value
ASID	A 16-bit address space number of a socket application address space.
Resource name	An EBCDIC job name, right-padded with blanks if less than 8 characters long, of a socket application address space (Client Name in Netstat displays). A question mark can be used to wildcard a single character, and an asterisk can be used to wildcard zero or more characters. For example, a value of A?C* matches all names with a first character A and a third character C, but does not match two-character names or names beginning with B through Z.
Resource ID	A 32-bit unsigned binary TCP/IP resource identifier (Client ID in Netstat displays).
Server resource ID	A 32-bit unsigned binary TCP/IP resource identifier of the related server listening connection.
Local IP address	A 32-bit IPv4 address or a 128-bit IPv6 address. The local IP address filter value is specified as the IP address field within a sockaddr structure. The sockaddr address family field must be set to indicate whether the local IP address filter value is an IPv4 address or an IPv6 address. For IPv4 connections, the local IP address filter value can be specified as either an IPv4 address (for example, 9.1.2.3) or as an IPv4-mapped IPv6 address (for example, ::FFFF:9.1.2.3). For all connections, a null address can be specified as either an IPv4 address (0.0.0.0), as an IPv4-mapped IPv6 address (::FFFF:0.0.0.0), or as an IPv6 address (::).
Local IP address prefix	A 16-bit signed binary value specifying the number of local IP address bits to use. For example, a value of 12 means that the first 12 bits of a connection's local IP address will be compared to the first 12 bits of the local IP address filter value. A value of 0 means that all address bits will be compared. A value greater than 32 for an IPv4 address, or greater than 128 for an IPv6 address, means that all address bits will be compared.
Local port	A 16-bit unsigned binary port number.

Filter item	Filter item value
Remote IP address	A 32-bit IPv4 address or a 128-bit IPv6 address. The remote IP address filter value is specified as the IP address field within a sockaddr structure. The sockaddr address family field must be set to indicate whether the remote IP address filter value is an IPv4 address or an IPv6 address. For IPv4 connections, the remote IP address filter value can be specified as either an IPv4 address (for example, 9.1.2.3) or as an IPv4-mapped IPv6 address (for example, ::FFFF:9.1.2.3). For all connections, a null address can be specified as either an IPv4 address (0.0.0.0), as an IPv4-mapped IPv6 address (::FFFF:0.0.0.0), or as an IPv6 address (::).
Remote IP address prefix	A 16-bit signed binary value specifying the number of remote IP address bits to use. For example, a value of 12 means that the first 12 bits of a connection's remote IP address will be compared to the first 12 bits of the remote IP address filter value. A value of 0 means that all address bits will be compared. A value greater than 32 for an IPv4 address, or greater than 128 for an IPv6 address, means that all address bits will be compared.
Remote port	A 16-bit unsigned binary port number.
Lu Name	An EBCDIC LU name (right-padded with blanks if less than 8 characters in length) of the TN3270E LU representing the client. Use a question mark (?) as a wildcard for a single character and an asterisk (*) as a wildcard for zero or more characters. For example, a value of A?C* matches all names with a first character A and a third character C, but does not match 2-character names, names beginning with B through Z, or names with anything other than C in the third position.
Monitor Group Identifier	A 32-bit unsigned binary value assigned by the TN3270E Telnet server to identify up to 255 unique monitor groups. Any parameter change within an existing monitor group or a new monitor group causes the TN3270E Telnet server to assign a new identifier. The identifier is reported in the monitor group table and connection data allowing a comparison between monitoring criteria and actual connection performance. The monitor group identifier can be obtained by issuing the GetTnMonitorGroups request.
Application Name	An EBCDIC application name (right-padded with blanks if fewer than 8 characters in length) of the SNA application name in session with the TN3270E secondary LU representing the client. The application name can have wildcard characters. Use a question mark (?) as a wildcard for a single character, and an asterisk (*) as a wildcard for 0 or more characters. For example, a value of A?C* matches all application names with a first character A and a third character C, but does not match 2-character names, names beginning with B through Z, or names with anything other than C in the third position.

Filter item	Filter item value
Application Data	An EBCDIC character string (right-padded with blanks if fewer than 40 characters in length) associated with a TCP socket by the owning application using the SIOCSAPPLDATA IOCTL. The application data filter can have wildcard characters. Use a question mark (?) as a wildcard for a single character and an asterisk (*) as a wildcard for zero or more characters. For z/OS Communications Server applications, see Application data in the <i>z/OS Communications Server: IP Configuration Reference</i> for applications that use the SIOCSAPPLDATA ioctl as a source for information about the content, format, and meaning of the application data that the applications associate with the sockets that they own. For other applications, see the documentation that is supplied by the application. See Chapter 17, “Miscellaneous programming interfaces,” on page 643 for more information about associating application data with a socket.

You can specify up to a maximum of four filter elements. Each filter element can contain any combination of the items listed in the previous table. A filter element with no applicable items matches any connection. A connection must match all items specified in a filter element to pass that filter check; a connection must pass at least one filter check to be selected. The following section provides an example.

Format and details for action-type requests

The following section describes the format and details of the action-type requests provided with EZBNMIFR:

DropConnection

Drop one or more TCP connections or UDP endpoints.

The general format of the input for this request consists of the request header and the request section descriptors (triplets), which define the input data. In this case, a triplet describes the input and output buffer. It consists of the offset, in bytes, of the request section relative to the beginning of the request buffer, the number of elements in the request section, and the length of an element in the request section.

To drop a connection, the NWMDropConnEntry structure describes the input and output to the DropConnection request. Each element must input a resource ID, local address, local port, remote address, remote port, and protocol. It is possible that for a particular connection or endpoint specification, the drop attempt will fail. For this reason, the NWMDropConnEntry structure contains a return code and reason code to describe the reason for the failure. The following table describes the NWMDropConnEntry structure.

Table 89. NWMDropConnEntry description

Descriptor	Type	Description
Resource ID	Input	A 32-bit unsigned binary TCP/IP resource identifier (Client ID in Netstat displays). This descriptor is required for TCP connections and is ignored for UDP endpoints.

Table 89. NWMDropConnEntry description (continued)

Descriptor	Type	Description
Local IP address	Input	A 32-bit IPv4 address or a 128-bit IPv6 address. The local IP address value is specified as the IP address field within a sockaddr structure. The sockaddr address family field must be set to indicate whether the local IP address value is an IPv4 address or an IPv6 address. For IPv4 connections, the local IP address value can be specified as either an IPv4 address (for example, 9.1.2.3) or as an IPv4-mapped IPv6 address (for example, ::FFFF:9.1.2.3). For all connections, a null address can be specified as either an IPv4 address (0.0.0.0), as an IPv4-mapped IPv6 address (::FFFF:0.0.0.0), or as an IPv6 address (::). This descriptor is required.
Local port	Input	A 16-bit unsigned binary port number. The local port value is specified as the port field within the sockaddr structure. This descriptor is required.

Table 89. NWMDropConnEntry description (continued)

Descriptor	Type	Description
Remote IP address	Input	A 32-bit IPv4 address or a 128-bit IPv6 address. The remote IP address filter value is specified as the IP address field within a sockaddr structure. The sockaddr address family field must be set to indicate whether the remote IP address value is an IPv4 address or an IPv6 address. For IPv4 connections, the remote IP address value can be specified as either an IPv4 address (for example, 9.1.2.3) or as an IPv4-mapped IPv6 address (for example, ::FFFF:9.1.2.3). For all connections, a null address can be specified as either an IPv4 address (0.0.0.0), an IPv4-mapped IPv6 address (::FFFF:0.0.0.0), or an IPv6 address (::). This descriptor is required for TCP connections and is ignored for UDP endpoints.
Remote port	Input	A 16-bit unsigned binary port number. The remote port value is specified as the port field within the sockaddr structure. This descriptor is required for TCP connections and is ignored for UDP endpoints
Protocol	Input	An 8-bit character representing either IPPROTO_TCP or IPPROTO_UDP.
Return Code	Output	A 4-byte value, NWMDropConnRc. If this value is nonzero, it indicates that the drop attempted for this connection failed. This return code describes the reason for failure.
Reason Code	Output	A 4-byte value, NWMDropConnRs. When the Return Code is set, this value might provide more detailed information about why the drop request failed for this connection.

See “TCP/IP network management interface response format” for information about processing the result of a DropConnection request.

Filter definition example

Two filters are defined:

- Local IP Address = 9.0.0.1, Local Port = 5000
- Resource Name = FTP*

The following TCP connections exist:

- Resource Name = FTP1, Local IP Address = 9.0.0.2, Local Port = 5001
- Resource Name = FTP2, Local IP Address = 9.0.0.1, Local Port = 5000
- Resource Name = USR1, Local IP Address = 9.0.0.1, Local Port = 5002

When a GetConnectionDetail request is made, connection 1 is selected because it matches filter 2, connection 2 is selected because it matches filter 1, and connection 3 is not selected because it does not match either filter.

Specifying no filters (triplet offset field is zero, or triplet element count field is zero, or triplet element length field is zero) means that the caller is requesting information for all endpoints.

The following table shows which filter items are applicable for each request type. If you specify inapplicable filters for a particular request type, they are ignored.

Table 90. Filter items applicable to each request type

Filter items	GetTCPLListeners	GetUDPTTable	GetConnectionDetail	GetStorageStatistics	GetTnMonitorGroups	GetTnConnectionData
ASID	Yes	Yes	Yes	No	No	No
Resource name	Yes	Yes	Yes	No	No	No
Resource ID	Yes	Yes	Yes	No	No	Yes
Server resource ID	No	No	Yes	No	No	Yes
Local IP address	Yes	Yes	Yes	No	No	Yes
Local IP address prefix	Yes	Yes	Yes	No	No	Yes
Local port	Yes	Yes	Yes	No	No	Yes
Remote IP address	No	No	Yes	No	No	Yes
Remote IP address prefix	No	No	Yes	No	No	Yes
Remote port	No	No	Yes	No	No	Yes
Lu Name	No	No	No	No	No	Yes
Monitor Group Identifier	No	No	No	No	No	Yes
Application Name	No	No	No	No	No	Yes
Application Data	Yes	No	Yes	No	No	No

TCP/IP network management interface response format

The general format of the response is:

- The response header, the request section descriptors (triplets), and the response section descriptors (quadruplets). Processing is slightly different for the request types (poll-type and action-type) as described in the following sections.

Processing poll-type requests

Processing for the following poll-type requests is described in this section:

- GetTCPListeners
- GetUDPTTable
- GetConnectionDetail
- GetStorageStatistics
- GetTnMonitorGroups
- GetTnConnectionData

A quadruplet consists of the offset, in bytes, of the response section relative to the beginning of the response buffer, the number of elements in the response section, the length of a response section element, and the total number of connections that passed the request filter checks.

The response header has the number of bytes required to contain all the requested data. When the return code is ENOBUFS, use this value to allocate a larger request/response buffer and reissue the request.

Processing action-type requests

Processing for the following action-type request is described in this section:

- DropConnection

For this type of request, the quadruplet contains the offset and number of elements, which is the same as the offset and number of elements in the triplet (output is the same as the input). If the call to EZBNMIFR returns a non-negative return value, and the value for NWMQMatch returned in the quadruplet section is equal to the number of entries input, NWMQNumber, then all of the connections or endpoints were dropped successfully. If the call to EZBNMIFR returns a non-negative return value, and if NWMQMatch is less than NWMQNumber, then not all of the connections or endpoints were successfully dropped. In this case, the program should examine the return code that is set in each NWMDropConnEntry field. If the value of the return code is nonzero, then this connection was not dropped; if the value of the return code is 0, then the connection was dropped.

The following describes the codes:

Table 91. Return code values

NWMDropConnRC	NWMDropConnRSN	Description
ENOENT	JRGetConnErr	The connection was not in the correct state for retrieving or the connection was not found.
EMVSERR	JRPATDELErr	Deletion of a restricted port entry failed.
EACCES	JRPORTACCESSAUTH	User does not have authority to access this port.
EMVSERR	JRPATFNDErr	Search for a restricted port failed or the connection was not found.
ENOENT	JRPATFNDErr	Search for a restricted port failed or the connection was not found.

Table 91. Return code values (continued)

NWMDropConnRC	NWMDropConnRSN	Description
ENOENT	JRGETCONNERR	The connection was not in the correct state for retrieving.
EAGAIN	JRUDPNOTUP	TCP/IP was not initialized
EAGAIN	JRTCPNOTUP	The request was not successful. The target TCP/IP stack was not active.
EINVAL	JRINVALIDVALUE	The request was not successful. A value that is not valid was specified in the request/response header.

Guideline: Input to the DropConnection request will most likely be from the output result of a GetUDPTTable or GetConnectionDetail request where the filtered connection information might return connections that are not intended for termination. Applications that support the DropConnection request should be coded to ensure that the connections input for termination have been examined carefully by programming logic that selects connections that meet a specific criteria, such as state.

Example: One NWMDropConnEntry is submitted:

Resource ID =003A, Local IP Address=9.0.0.1, Local Port=5003,
Remote IP Address=9.0.0.5, Remote Port=3000, Protocol=TCP

The following TCP connections exist:

- Resource Name = FTP1, Resource ID = 001A, Local IP Address = 9.0.0.2, Local Port = 5000,Remote IP Address = 9.0.0.5, Remote Port = 3001
- Resource Name = FTP2, Resource ID = 002A, Local IP Address = 9.0.0.1, Local Port = 5001,Remote IP Address = 9.0.0.5, Remote Port = 3002
- Resource Name = USR1, Resource ID = 004F, Local IP Address = 9.0.0.1, Local Port = 5002,Remote IP Address = 9.0.0.5, Remote Port = 3003
- Resource Name = USR7, Resource ID = 003A, Local IP Address = 9.0.0.1, Local Port = 5003, Remote IP Address = 9.0.0.5, Remote Port = 3000

When a DropConnection request is made, connection 4 is dropped because it matches the five required items.

- The request sections.
- The response sections. One of the following types of response sections is returned:
 - TCP Connection Information
 - TCP Listener Information
 - UDP Connection Information
 - TCP/IP Storage Statistics
 - DropConnection Entries
 - TN3270E Telnet server Monitor Groups
 - TN3270E Telnet server Connection Performance Information

Tip: Connection elements for TN3270E Telnet server connection performance data are returned only if the connection is being monitored by a MonitorGroup mapped to the connection. See Connection monitoring mapping statement in the *z/OS Communications Server: IP Configuration Guide* for details.

Request and response data structures

The NMI request and response data structures for C/C++ and Assembler programs are located as follows:

Header file for C++ programs	Macros for Assembler programs	Contents
EZBNMRHC	EZBNMRHA	The NMI request and response data structure definitions.

These header files and macros are shipped in the SEZANMAC data set. This data set must be available in the concatenation when compiling or assembling a part that makes use of these definitions.

Examples

Example 1: The following C/C++ code fragment shows how to format a request to obtain TCP connection information using the filters in the filter definition example (see “Filter definition example” on page 593):

```
/*
 * NMI data definitions
 */
typedef struct {
    NWMHeader NMIheader;
    NWMFilter NMIfilter[2];
} NMIBuftype;
NMIBuftype *NMIBuffer;
unsigned int NMIAlet;
int NMILength;
int RV;
int RC;
unsigned int RSN;
#define NMIBUFSIZE 8192
NMIBuffer=malloc(NMIBUFSIZE);
NMIAlet=0;
NMILength=NMIBUFSIZE;
/*
 * Format the header
 */
NMIBuffer->NMIheader.NWMHeaderIdent=NWMHEADERIDENTIFIER;
NMIBuffer->NMIheader.NWMHeaderLength=sizeof(NWMHeader);
NMIBuffer->NMIheader.NWMVersion=NWMVERSION1;
NMIBuffer->NMIheader.NWMType=NWMTCPCONNTYPE;
NMIBuffer->NMIheader.NWMBytesNeeded=0;
NMIBuffer->NMIheader.NWMInputDataDescriptors.\
    NWMFiltersDesc.NWMTOffset=sizeof(NWMHeader);
NMIBuffer->NMIheader.NWMInputDataDescriptors.\
    NWMFiltersDesc.NWMTLength=sizeof(NWMFilter);
NMIBuffer->NMIheader.NWMInputDataDescriptors.\
    NWMFiltersDesc.NWMTNumber=2;
/*
 * Format filter 1
 */
NMIBuffer->NMIfilter[1].NWMFilterIdent=NWMFILTERIDENTIFIER;
NMIBuffer->NMIfilter[1].NWMFilterFlags=NWMFILTERLCLADDRMASK|\
    NWMFILTERLCLPORTMASK;
NMIBuffer->NMIfilter[1].NWMFilterLocal.\
```



```

    NWMFilterLocalAddr4.sin_family=AF_INET;
    NMIBuffer->NMIfilter[1].NWMFilterLocal.\
    NWMFilterLocalAddr4.sin_port=5000;
    NMIBuffer->NMIfilter[1].NWMFilterLocal.\
    NWMFilterLocalAddr4.sin_addr.s_addr=0x09000001;

/*****
/*
/* Format filter 2
/*
*****/
NMIBuffer->NMIfilter[2].NWMFilterIdent=NWMFILTERIDENTIFIER;
NMIBuffer->NMIfilter[2].NWMFilterFlags=NWMFILTERRESNAMEMASK;
memcpy(NMIBuffer->NMIfilter[2].NWMFilterResourceName,"FTP* ",8);
/*****
/*
/* Invoke NMI service
/*
*****/
NWMServices(TcpipJobName,NMIBuffer,&NMIalet,&NMILength,&RV,&RC,&RSN);

```

Guideline: In z/OS releases prior to V1R7, the current version is 1. In releases V1R7 and later, the current version is 2. Applications coded with NWMVERSION1 (as in example 1) will have the version accepted in release V1R4 and later. However, applications coded with NWMCURRENTVER and compiled using the version 2 headers work only on z/OS V1R7 and later releases. Applications using NWMCURRENTVER in V1R7 and later releases should recognize that the current version might not be accepted on prior releases of the operating system. When these applications receive an error code indicating an error in the version, they should drop back to the prior (or lowest) version number and verify that that version is acceptable with the current operating system.

The version used does not restrict which functions are available. If an application using version 1 and compiled with a version 2 header is executed on a prior release of the operating system, the application will receive the data corresponding to the release of the operating system on which it executes. Therefore, if the application is executing on a system running version 2 and specifies version 1, it still receives all data including the new version 2 data (STOKEN). If the same application is executed on a release that supports only version 1, it receives everything except the new version 2 data.

Example 2: The following C/C++ code fragment shows how to drop a connection using the following values:

```

Resource ID = 003A
Local IP Address = 9.0.0.1
Local Port = 5003
Remote IP Address = 9.0.0.5
Remote Port = 3000

```

```

/*****
/*
/* NMI data definitions
/*
*****/
typedef struct {
    NWMHeader      NMIheader;
    NWMDropConnEntry NMIDropConnEntry;
} NMIBuftype;
NMIBuftype NMIBuffer;
unsigned int NMIalet;
int NMILength = sizeof(NMIBuffer);

```

```

int RV;
int RC;
unsigned int RSN;
/*****
*/
/* Format the header
*/
/*****
memset(&NMIBuffer, 0, sizeof(NMIBuffer));
NMIBuffer.NMIheader.NWMHeaderIdent=NWMHEADERIDENTIFIER;
NMIBuffer.NMIheader.NWMHeaderLength=sizeof(NWMHeader);
NMIBuffer.NMIheader.NWMVersion=NWMCURRENTVER;
NMIBuffer.NMIheader.NWMType=NWMDROPCONNTYPE;
NMIBuffer.NMIheader.NWMBytesNeeded=0;
NMIBuffer.NMIheader.NWMInputDataDescriptors.\
    NWMIODesc.NWMTOffset=sizeof(NWMHeader);
NMIBuffer.NMIheader.NWMInputDataDescriptors.\
    NWMIODesc.NWMTLength=sizeof(NWMDropConnEntry);
NMIBuffer.NMIheader.NWMInputDataDescriptors.\
    NWMIODesc.NWMTNumber=1;
/*****
*/
/* Format the NMIDropConnEntrytor
*/
/*****
NMIBuffer.NMIDropConnEntry.NWMDropConnIdent=NWMDropConnIdentifier;
NMIBuffer.NMIDropConnEntry.NWMDropVersion=NWMDropCurrentVersion;
NMIBuffer.NMIDropConnEntry.NWMDropConnId = 0x003a;
NMIBuffer.NMIDropConnEntry.NWMDropConnLocalAddr4.sin_family=AF_INET;
NMIBuffer.NMIDropConnEntry.NWMDropConnLocalAddr4.sin_port=5003;
NMIBuffer.NMIDropConnEntry.NWMDropConnLocalAddr4.sin_addr.s_addr=0x09000001;

NMIBuffer.NMIDropConnEntry.NWMDropConnRemoteAddr4.sin_family=AF_INET;
NMIBuffer.NMIDropConnEntry.NWMDropConnRemoteAddr4.sin_port=3000;
NMIBuffer.NMIDropConnEntry.NWMDropConnRemoteAddr4.sin_addr.s_addr=0x09000005;
NMIBuffer.NMIDropConnEntry.NWMDropProtocol = IPPROTO_TCP;

/*****
*/
/* Invoke NMI service
*/
/*****
NWMServicesEZBNMIFR(TcpipJobName,NMIBuffer,&NMIAlet,&NMILength,&RV,&RC,&RSN);

/*****
*/
/* Check the return code
*/
/*****
if (rc != sizeof(NMIBuffer))
{
    printf("EZBNMIFR drop rc=%d\n", rc);
}
else
    /* Ensure that the number of entries input for drop match the
    /* number of entries actually dropped
    if (NMIBuffer.NMIHeader.NWMDropConnDesc.NWMQMatch !=
        NMIBuffer.NMIheader.NWMInputDataDescriptors.NWMIODesc.NWMTNumber)
    {
        printf("EZBNMIFR drop for connection 0x%8.8x errno=%d / 0x%8.8X\n",
            NMIBuffer.MWMDropConnEntry.NWMDropConnIdent,
            NMIBuffer.NMIDropConnEntry.NWMDropConnRc,
            NMIBuffer.NMIDropConnEntry.NWMDropConnRs);
    }
    else

```

```

{
    printf("EZBNMIFR drop for connection 0x%8.8x successful\n",
        NMIBuffer.NWMDropConnEntry.NWMDropConnIdent);
}

```

Example 3: The following assembler code fragment shows how to format a request to obtain TCP connection information using the filters in the filter definition example (see “Filter definition example” on page 593):

```

R0      EQU    0
R1      EQU    1
R2      EQU    2
R3      EQU    3
R4      EQU    4
R5      EQU    5
R6      EQU    6
R7      EQU    7
R8      EQU    8
R9      EQU    9
R10     EQU    10
R11     EQU    11
R12     EQU    12
R13     EQU    13
R14     EQU    14
R15     EQU    15
        STORAGE OBTAIN,LENGTH=@DYN SIZE,ADDR=(R13),LOC=ANY
        USING NMIDATA,R13
        ST      R13,NMIBuffer
*****
*
* Format the header
*
*****
        LA      R2,NMIheader
        USING NWMHeader,R2
        XC      NWMHeader,NWMHeader
        MVC      NWMHeaderIdent,=A(NWMHeaderIdentifier)
        LHI     R0,NWMHeaderSize
        ST      R0,NWMHeaderLength
        LHI     R0,NWMVersion1
        STH     R0,NWMVersion
        LHI     R0,NWMTcpConnType
        STH     R0,NWMType
        XC      NWMBytesNeeded,NWMBytesNeeded
        LA      R3,NWMInputDataDescriptors
        USING NWMTriplet,R3
        LHI     R0,NWMHeaderSize
        ST      R0,NWMTOffset
        LHI     R0,NWMFilterSize
        ST      R0,NWMTLength
        LHI     R0,NMIFilter#
        ST      R0,NWMTNumber
        DROP    R3
        DROP    R2
*****
*
* Format filter 1
*
*****
        LA      R2,NMIFilter1
        USING NWMFilter,R2
        XC      NWMFilter,NWMFilter
        MVC      NWMFilterIdent,=A(NWMFilterIdentifier)
        MVC      NWMFilterFlags,=A(NWMFilterLclAddrMask)
        OC      NWMFilterFlags,=A(NWMFilterLclPortMask)
        LA      R3,NWMFilterLocalAddr4
        USING SOCKADDR,R3

```

```

        LHI    R0,AF_INET
        STC    R0,SOCK_FAMILY
        LHI    R0,5000
        STH    R0,SOCK_SIN_PORT
        MVC    SOCK_SIN_ADDR,=XL4'09000001'
        DROP   R2,R3

*****
*
* Format filter 2
*
*****
        LA     R2,NMIfilter2
        USING  NWMFilter,R2
        XC     NWMFilter,NWMFilter
        MVC    NWMFilterIdent,=A(NWMFilterIdentifier)
        MVC    NWMFilterFlags,=A(NWMFilterResNameMask)
        MVC    NWMFilterResourceName,=CL8'FTP*'
        DROP   R2

*****
*
* Invoke NMI service
*
*****
        CALL   EZBNMIFR,
              (TcpipJobName,NMIBuffer,NMIAlet,NMILength,RV,RC,RSN)
        STORAGE RELEASE,LENGTH=@DYN SIZE,ADDR=(R13)
        DROP   R13
        EJECT
RV       DC    F'0'
RC       DC    F'0'
RSN      DC    A(0)
NMIAlet  DC    A(0)
NMIBuffer DC    A(0)
NMILength DC    A(NMIBUFSIZE)
TcpipJobName DC CL8'TCPIP'
        LTORG ,
NMIdata  DSECT
NMIBuftype DS CL8192
        ORG    NMIdata
NMIheader DS CL(NWMHeaderSize)
NMIfilter1 DS CL(NWMFilterSize)
NMIfilter2 DS CL(NWMFilterSize)
NMIfilter# EQU (*-NMIfilter1)/NWMFilterSize
        ORG    ,
NMIBUFSIZE EQU *-NMIdata
BPXYSOCK DSECT=YES,LIST=NO
EZBNMRHA DSECT=YES,LIST=NO,TITLE=NO,NWMLIST=YES

```

I SNA network monitoring NMI

z/OS Communications Server VTAM provides a single AF_UNIX socket interface for allowing network management applications to obtain the following types of data:

- Enterprise Extender (EE) connection data
This data contains information about all EE connections or a desired set of EE connections as specified by the application using the local IP address or host name, the remote IP address or host name, or both.
- Enterprise Extender summary data
This data contains information comprising a summary of EE activity for this host.
- High Performance Routing (HPR) connection data

This data contains information about specific HPR connections Rapid Transport Protocol physical units (RTP PUs) as specified by the application using either 1) the RTP PU name, or 2) the RTP partner CP name with an optional APPN COS specification. These RTP PUs are not limited to those using EE connections.

- Common Storage Manager (CSM) statistics

This data contains CSM storage pool statistics and CSM summary information.

Overview

A client network management application polls for information through specific requests using an AF_UNIX stream socket connection that uses VTAM as the server for that socket. The requested data is provided to the application directly using the AF_UNIX stream socket connection.

Configuration

The z/OS system administrator might restrict access to this interface by defining the RACF (or equivalent external security manager product) resource `IST.NETMGMT.sysname.SNAMGMT` in the `SERVAUTH` class (*sysname* represents the MVS system name where the interface is being invoked).

For applications that use the interface, the MVS user ID is permitted to the defined resource. If the resource is not defined, then only superusers (users permitted to `BPX.SUPERUSER` resource in the `FACILITY` class) are permitted to it. If you are developing a feature for a product to be used by other parties, include instructions in your documentation indicating that either administrators must define and give appropriate permission to the given security resource to use that feature, or you must run your program as superuser.

Requirements:

- The administrator must define an OMVS segment for VTAM if one is not already defined.
- The VTAM OMVS user ID must have write access to the `/var` directory.

Enabling and disabling the interface

You can enable the SNA Network Monitoring data interface by setting the VTAM start option `SNAMGMT` to `YES`, and you can disable the interface by setting the VTAM start option `SNAMGMT` to `N0`. The default for this start option is `N0`, and the start option is modifiable after VTAM is started. This start option can be specified in any of the following ways:

- Using the `START` command for VTAM
 1. IBM default value is `N0`
 2. Within the default VTAM start option list `ATCSTR00` (`ATCSTR00` is always used regardless of whether `LIST=xx` was entered to specify a supplemental VTAM start option list)
 3. Within the supplemental VTAM start list (`ATCSTRxx`, if `LIST=xx` entered) as `SNAMGMT=YES` or `SNAMGMT=N0`
 4. From the backup start option list (specified by the `LISTBKUP` start option)
 5. `START` command options entered by operator as `SNAMGMT=YES` or `SNAMGMT=N0`
 6. `START` command options reentered by the operator

See Sources of start options in the *z/OS Communications Server: SNA Network Implementation Guide* for more information.

- Using the MODIFY VTAMOPTS command
`MODIFY vtamprocname,VTAMOPTS,SNAMGMT=YES`
`MODIFY vtamprocname,VTAMOPTS,SNAMGMT=NO`

The current value of the SNAMGMT start option is displayable using any of the following VTAM DISPLAY commands:

```
DISPLAY NET,VTAMOPTS
DISPLAY NET,VTAMOPTS,OPTION=SNAMGMT
DISPLAY NET,VTAMOPTS,FUNCTION=VTAMINIT
```

Communicating with the server

Applications that need to communicate with the VTAM AF_UNIX server can do so by creating an AF_UNIX stream socket using either the Language Environment C/C++ API or UNIX System Services BPX callable services. The VTAM server provides a well-known AF_UNIX stream socket with a socket path name of /var/sock/SNAMGMT that applications can use in connecting to the server.

Perform the following steps to communicate with the VTAM server:

1. Open an AF_UNIX socket.
2. Connect to the VTAM server using the socket path name /var/sock/SNAMGMT.
3. Read any data on the socket.
4. Build the NMI request packet.
5. Send the packet to the VTAM server.
6. Read the reply.
 - If the reply is a termination record, perform any application cleanup.
 - If the reply is a response to the request, process the response.
7. Repeat the process beginning with step 4 or close the connection.

Tips:

- When an application establishes a successful connection to the VTAM server, the server responds by sending an initialization record to the application. The application must read this record before it can start processing request responses.
- When VTAM needs to close the connection with the application, it attempts to send a termination record to the application before closing the connection. VTAM will close the connection when VTAM terminates, when the interface is disabled by an operator, or when there are severe formatting errors in the data requests sent by the application to the VTAM server.
- Both the initialization and termination records conform in structure to the solicited response records sent by VTAM to the application; see “SNA network management interface (NMI) request/response format” for details.

SNA network management interface (NMI) request/response format

This interface uses a request/response method over the socket. The application builds and sends an NMI request over the socket. The request specifies the type of information to be received and might contain data filters. The application must issue a receive to get the NMI response over the socket. The NMI response provides either 1) data that satisfies the request (matching any input filters

specified on the request), or 2) an error response. A severe formatting error in the application's NMI request results in VTAM sending a termination record and closing the connection.

The SNA Network Management Interface provides the formatted response data directly to the application over the AF_UNIX socket. This is in contrast to the NMIs described in "Real-time TCP/IP network monitoring NMI" on page 528, which return a token to a response buffer that the application must use as input to the EZBTMIC1 callable service in order to obtain the formatted response data.

The NMI request and response mappings are provided for programming to this interface.

Request format

All SNA NMI requests flow on the socket from the client application to the VTAM server. The general format of an SNA NMI request is:

- The request header includes the request type and the request section descriptors (triplets). The following request types can be made:
 - EE Connection Request — obtain information about some or all Enterprise Extender connections.
 - EE Summary Request — obtain summary information about all Enterprise Extender connections.
 - HPR Connection Request — obtain information about one or more HPR connections.
 - CSM Statistics Request — obtain information about global CSM statistics.

A triplet consists of the offset (in bytes) of the request section relative to the beginning of the request header, the number of elements in the request section, and the length of a request section element.

- The request sections. The only type of request section that can be specified is a filter element.
 - In an EE Connection Request, either zero or one filter elements can be included. The set of all EE connections can be selected either by not including a filter element in the request or by supplying a filter element with no filter parameters specified. A subset of EE connections can be selected by supplying a filter element that includes any combination of the filter parameters in the following table. z/OS Communications Server does not perform name resolution (to an IP address) on any supplied host name, but simply looks for connections that were established using the given host name.

Restriction: Local Hostname and IPv6 filter parameters are applicable only to z/OS V1R5 Communications Server and later releases. If the initialization record that was received by the client when the connection was opened specifies that Local Hostname and IPv6 addresses are not supported by this VTAM level (any VTAM level prior to z/OS V1R5), then the server rejects any request that contains a Local Hostname value or IPv6-format addresses.

Local Hostname	An EBCDIC name, right-padded with nulls or blanks if less than 64 characters long (applicable to CS for z/OS V1R5 and later releases only). The Local Hostname parameter is ignored if Local IP Address is specified.
Local IP Address	A 32-bit IPv4 address or a 128-bit IPv6 address. (IPv6 address is applicable to CS for z/OS V1R5 and later releases only.)

Remote Hostname	An EBCDIC name, right-padded with nulls or blanks if less than 64 characters long. The Remote Hostname parameter is ignored if a Remote IP Address value is specified.
Remote IP Address	A 32-bit IPv4 address or a 128-bit IPv6 address. (IPv6 address is applicable to CS for z/OS V1R5 and later releases only).

- In an HPR Connection Request, you select a subset of HPR connections based on any combination of the following items that includes, at a minimum, either the RTP PU Name or the Partner CP Name (between one and four filter elements can be specified per request):

RTP PU Name	An EBCDIC name, right-padded with nulls or blanks if less than 8 characters long.
Partner CP Name	A fully qualified EBCDIC name, right-padded with nulls or blanks if less than 17 characters long. Partner CP Name is ignored if RTP PU Name is specified. If a network identifier is not supplied, the Partner CP Name is qualified with the host's network ID. Use a question mark (?) as a wildcard for a single character or an asterisk (*) as a wildcard for zero or more characters. For example, a value of A?C* matches all names with a first character equal to A and a third character equal to C, but does not match 2-character names or names beginning with characters B through Z. To request all known connections, use the string *.*. To request all known connections in the same network as this host, use as asterisk (*).
COS Name	An EBCDIC name, right-padded with nulls or blanks if less than 8 characters long. COS is ignored if RTP PU Name is specified.

- An EE Summary Request cannot contain any Filter Elements. No filters are applicable to an EE summary request.
- A CSM Statistics Request cannot contain any Filter Elements. No filters are applicable to a CSM statistics request

The following table shows which filter parameters are required, optional, or not applicable (N/A) for each request type. If inapplicable filters are specified for a particular request type, an EE Connection Request or HPR Connection Request, they are ignored. EE Summary Requests or CSM Statistics Requests that contain Filter Elements are rejected by VTAM.

Request Type	Local IP Address or Hostname	Remote IP Address or Hostname	RTP PU name or Partner CP name	COS name
EE Connection Request	Optional; Local Hostname ignored if Local IP Address is specified	Optional; Remote Hostname is ignored if Remote IP Address is specified	N/A	N/A
EE Summary Request	N/A	N/A	N/A	N/A

Request Type	Local IP Address or Hostname	Remote IP Address or Hostname	RTP PU name or Partner CP name	COS name
HPR Connection Request	N/A	N/A	One is required; Partner CP Name ignored if RTP PU Name is specified	Optional; ignored if RTP PU Name is specified
CSM Statistics Request	N/A	N/A	N/A	N/A

Conceptually, every valid request record sent to VTAM by the client has the following general request format structure:

Common Request/Response Header
Input Triplet information: a single triplet is defined • Offset from start of request header to first input section • Length of each input section of this type • Number of input sections of this type
Start of input information (offset from start of request header to this data given in Input Triplet)

Response format

All SNA NMI responses flow on the socket from the VTAM server to the client application. The general format of an NMI response is as follows:

- The response header, which includes the response type, the return code and reason code, the request section descriptors (triplets), and the response section descriptors (quadruplets). A quadruplet consists of the offset (in bytes) of the response section relative to the beginning of the response header, a reserved field, the number of elements in the response section, and the total number of elements that passed the request filter checks.

Tip: This last field in the quadruplet is applicable only to responses that have a corresponding request. Initialization and termination records do not have corresponding requests. Therefore, this field is reserved and is set to a value of 0 on responses containing initialization and termination records.

- The request sections.
- The response sections.
 - Response sections of the following solicited response types are returned if data is found that matches the corresponding filtered or unfiltered request (if no matches were found, no response data sections are returned):
 - EE connection information
 - EE summary information
 - HPR connection information
 - CSM statistics information
 - An initialization record always contains a single response section.
 - A termination record does not contain a response section (all information is contained within the response header).

The NMI response section consists of one or more records containing information that passed the request filter checks.

The general format of an NMI response section record is:

- The record header, which contains the overall length of the record and one or more subrecord descriptors (triplets). The record triplet consists of the offset in bytes, relative to the start of the response section record, for the first instance of a given subrecord; the length in bytes of this particular subrecord; and the total number of instances of this subrecord.
- The subrecord sections associated with this response section record.

An application wishing to navigate an NMI response must use the overall length value in the response section record to move to the next variable length record. The application should use the response section record triplet data to navigate within the record itself.

The following response section records are returned for the solicited response types:

- EE Summary Response
 1. One EE Summary Global Data Section Record
 2. One or more EE Summary IP Address Data Section Records
- EE Connection Response
 1. One or more EE Connection Data Section Records
- HPR Connection Response
 1. One HPR Connection Global Data Section Record
 2. One or more HPR Connection Specific Data Section Records
- CSM Global Statistics Response
 1. One CSM Global Pools Section Record
 2. One CSM Summary Section Record

Conceptually, every response record sent by VTAM to the client will look like the format that follows.

Common Request/Response Header
Input Triplet information (copied from corresponding request, if any): a single triplet is defined. • Offset from start of response data to first input record • Length of each input section of this type • Number of input sections of this type
Output Quadruplet information: a single quadruplet is defined. • Offset from start of response data to first output record • 0 • Number of output records included in this response. If this value is less than the number of records matching the filters supplied on the corresponding request (if any), then some data was not reported as the result of storage constraints. • Number of output records matching the filters supplied on corresponding request, if any
Start of input information (copied from corresponding request, if any — offset from start of response data saved in Input Triplet)
Start of output information (offset from start of response data saved in Output Quadruplet)

Request and response data structures and records

The SNA network monitor request and response data structures for C/C++ and Assembler programs are located as follows:

Header files for C/C++ programs	Macros for Assembler programs	Contents
ISTEEHNC	ISTEEHNA	The NMI request and response header, initialization record, and termination record structure definitions
ISTEESUC	ISTEESUA	The EE summary response data structure definitions
ISTEECOC	ISTEECOA	The EE connection response data structure definitions
ISTHPRCC	ISTHPRCA	The HPR connection response data structure definitions
ISTCSMGC	ISTCSMGA	The CSM statistics response data structure definitions

These header files and macros are shipped in SYS1.MACLIB. This data set must be available in the concatenation when compiling or assembling a part that makes use of these definitions.

Initialization record: The structure of the initialization record follows.

Enterprise Extender initialization record format

Common Request/Response Header
Input Triplet information (no corresponding input request): a single triplet is defined. • Offset from start of response data to first input section • Length of each input section of this type: 0 • Number of input sections of this type: 0
Output Quadruplet information: a single quadruplet is defined. • Offset from start of response data to first output record • 0 • Number of output records included in this response: 1 • 0
Start of output information (offset from start of response data saved in Output Quadruplet), specifically one: • Enterprise Extender initialization record

Record Identifier (4 characters): NMII
VTAM Level, from ATCVT (8 bytes)
TOD VTAM Started, from ATCVT (8 bytes)
SNA Network Management Component Name: SNAMGMT

Functions Supported (8 bits) • IPv6 addresses supported (1 bit) – 0=IPv6 addresses not supported – 1=IPv6 addresses supported • Local Hostname filter parameter supported (1 bit) – 0= Local Hostname filter parameter not supported – 1= Local Hostname filter parameter supported • Reserved (6 bits): '000000'B
Reserved (15 bytes): 0

Termination record: The following table describes the structure of the termination record. The termination record contains no output data other than the return code and reason code in the response header.

Enterprise Extender termination record format

Common Request/Response Header
Input Triplet information (no corresponding input request): a single triplet is defined. • Offset from start of response data to first input section • Length of each input section of this type: 0 • Number of input sections of this type: 0
Output Quadruplet information: a single quadruplet is defined. • Offset from start of response data to first output record • 0 • Number of output records included in this response: 0 • 0

EE summary response record: The structure of the EE Summary response follows.

Enterprise Extender Summary Response format

Common Request/Response Header
Input Triplet information (copied from request): a single triplet is defined. • Offset from start of response data to first input section • Length of each input section of this type • Number of input sections of this type
Output Quadruplet information: a single quadruplet is defined. • Offset from start of response data to first output record • 0 (since the records that follow are variable length records) • Number of output records included in this response (if this value is less than number of records matching the filters supplied on the corresponding request, then some data was not reported due to storage constraints) • Number of output records matching the filters supplied on the corresponding request
Start of input information (copied from request, offset from start of response data saved in Input Triplet)
Start of output information (offset from start of response data saved in Output Quadruplet), specifically a collection of: • Enterprise Extender Summary Global Output Record (one instance) • One or more Enterprise Extender Summary IP Address Output Records (one instance per IP address being reported)

Enterprise Extender Summary Global Output Record

Record Identifier (4 characters) — EESG
Length of overall record (4 bytes)
Reserved field (2 characters)
Number of triplets for this output record (2 bytes): 1
Output Record Triplet information • Offset from start of the record to first section of this type within the output record (4 bytes) • Length of every section of this type within the output record (2 bytes) • Number of output sections of this type within the output record (2 bytes)
Start of Enterprise Extender Summary static information section (one instance)

Enterprise Extender Summary IP Address Output Record

Record Identifier (4 characters): EESI
Length of overall record (4 bytes)
Reserved field (2 characters)
Number of triplets for this output record (2 bytes): 2
Output Record Triplet information • Offset from start of the record to first section of this type within the output record (4 bytes) • Length of every section of this type within the output record (2 bytes) • Number of output sections of this type within the output record (2 bytes)
Start of Enterprise Extender Summary IP address information section (one instance)
Start of Enterprise Extender Summary Hostname information section (one per host name used to obtain this IP address, zero if no host name resolution was performed)

EE connection response record: The structure of the response record is as follows:

Common Request/Response Header
Input Triplet information (copied from request): a single triplet is defined. • Offset from start of response data to first input section • Length of each input section of this type • Number of input sections of this type
Output Quadruplet information: a single quadruplet is defined. • Offset from start of response data to first output record • 0 • Total number of output records • Number of output records included in this response (if this value is not equal to total, then some data was not reported)
Start of input information (copied from request, offset from start of response data saved in Input Triplet)
Start of output information (offset from start of response data saved in Output Quadruplet), specifically a collection of: • One or more Enterprise Extender Connection Specific Output Records (one instance per EE connection reported)

Enterprise Extender Connection Specific Output Record

Record Identifier (4 characters): EECO
Length of overall record (4 bytes)
Reserved field (2 characters)
Number of triplets for this output record (2 bytes): 4
Output Record Triplet information • Offset from start of the record to first section of this type within the output record (4 bytes) • Length of every section of this type within the output record (2 bytes) • Number of output sections of this type within the output record (2 bytes)
Start of Enterprise Extender Connection static information section (one instance)
Start of one or more Enterprise Extender Connection Hostname sections (0–2 possible instances, one for local and one for remote host name if applicable)
Start of Enterprise Extender Connection Associated VRN name section (one instance, included only if the EE connection is across a virtual routing node)
Start of one or more Enterprise Extender Connection Associated RTP PU name sections (one instance per RTP PU that is using this EE connection)

HPR connection response record:

HPR Connection Response format

Common Request/Response Header
Input Triplet information (copied from request): a single triplet is defined. • Offset from start of response data to first input section • Length of each input section of this type • Number of input sections of this type
Output Quadruplet information: a single quadruplet is defined. • Offset from start of response data to first output record • 0 • Number of output records included in this response (if this value is less than number of records matching the filters supplied on the corresponding request, then some data was not reported as the result of storage constraints) • Number of output records matching the filters supplied on the corresponding request
Start of input information (copied from request, offset from start of response data saved in Input Triplet)
Start of output information (offset from start of response data saved in Output Quadruplet), specifically a collection of: • HPR Connection Global Output Record (one instance) • One or more HPR Connection Specific Output Records (one instance per HPR connection reported)

HPR Connection Global Output Record

Record Identifier (4 characters): HPRG
Length of overall record (4 bytes)
Reserved field (2 characters)
Number of triplets for this output record (2 bytes): 1

HPR Connection Global Output Record

Output Record Triplet information
• Offset from start of the response data to first section of this type within the output record (4 bytes) • Length of every section of this type within the output record (2 bytes) • Number of output sections of this type within the output record (2 bytes)
Start of HPR Connection Global data

HPR Connection Specific Output Record

Record Identifier (4 characters): HPRC
Length of overall record (4 bytes)
Reserved field (2 characters)
Number of triplets for this output record (2 bytes): 3
Output Record Triplet information
• Offset from start of the record to first section of this type within the output record (4 bytes) • Length of every section of this type within the output record (2 bytes) • Number of output sections of this type within the output record (2 bytes)
Start of HPR Connection static information section (one instance)
Start of HPR Connection Route Selection Control Vector (SNA Control Vector X'2B') section (one instance, potentially none if connection is in the process of performing a pathswitch)
Start of HPR Connection Pathswitch information section (present only if pathswitch had ever occurred on this connection, one instance if present)

CSM statistics response record: The structure of the CSM Statistics response is as follows:

CSM Statistics Response format

Common Request/Response Header
Input Triplet information (copied from request): a single triplet is defined.
• Offset from start of response data to first input section • Length of each input section of this type • Number of input sections of this type
Output Quadruplet information: a single quadruplet is defined.
• Offset from start of response data to first output record • 0 • Number of output records included in this response (if this value is less than number of records matching the filters supplied on the corresponding request, then some data was not reported as the result of storage constraints) • Number of output records matching the filters supplied on the corresponding request
Start of input information (copied from request, offset from start of response data saved in Input Triplet)
Start of output information (offset from start of response data saved in Output Quadruplet), specifically a collection of:
• CSM Global Pool Output Record containing multiple CSM Buffer Pool data records (CSMPoolGData), one per pool • CSM Global Summary Output Record containing a single CSM Summary Data record (CSMSummGData) representing CSM system wide summary information

CSM Global Output Pool Record

Record Identifier (4 characters): CSMP
Length of overall record (4 bytes)
Reserved field (2 characters)
Number of triplets for this output record (2 bytes): 1
Output Record Triplet information • Offset from start of the response data to first section of this type within the output record (4 bytes) • Length of every section of this type within the output record (2 bytes) • Number of output sections of this type within the output record (2 bytes)
Start of CSM Global Pool data (CSMPoolGDdata) records, one per CSM pool

CSM Global Output Summary Record

Record Identifier (4 characters): CSMS
Length of overall record (4 bytes)
Reserved field (2 characters)
Number of triplets for this output record (2 bytes): 1
Output Record Triplet information • Offset from start of the response data to first section of this type within the output record (4 bytes) • Length of every section of this type within the output record (2 bytes) • Number of output sections of this type within the output record (2 bytes)
Start of CSM Global Summary data (CSMSummGData) record one single system wide record

NMI request errors

The following table describes the errors in an NMI Request for which VTAM will send a termination record with the given Return Code and Reason Code, and then close the connection.

Return Code	Reason Code	Meaning
EINVAL	X'00007110'	Request header too short.
EINVAL	X'00007111'	Unsupported version number in request header.
EINVAL	X'00007112'	Triplet format is not valid: first request section is not contiguous to request header.
EINVAL	X'00007112'	Triplet format is not valid: length of filter element is not correct for given version.
EINVAL	X'00007113'	Length of request header plus length of request sections does not equal total length of request.
EINVAL	X'00007114'	Eyecatcher in request header is not valid.

The following table describes the error in an NMI request for which VTAM will return a negative response of the same type as the request. VTAM will leave the connection active after returning the negative response for these errors.

Return Code	Reason Code	Meaning
EINVAL	'00007115'X	Unrecognized request type.
EINVAL	'00007116'X	Too many filter elements (request sections) included for request type.
EINVAL	'00007117'X	Too few filter elements (request sections) included for request type.
EINVAL	'00007118'X	Undefined filter parameter indicator set in filter element.
EINVAL	'00007119'X	Required filter parameter missing from filter element.
EINVAL	'0000711A'X	Unsupported filter parameter indicator set in filter element.
EINVAL	'0000711B'X	Request not valid for HPR or EE information in a pure subarea VTAM node.

Diagnosis

The interfaces that are described in this topic are designed to return error information as either a `return_value`, `return_code`, or `reason_code`, where applicable. The information in this section should be used to further diagnose the problem that is being reported.

When the `return_value` is -1, the `return_code` and `reason_code` indicate the problem that was incurred by the interface. See the section that describes the interface that is being used for `return_value`, `return_code`, and `reason_code` descriptions.

If you are not able to diagnose the problem using the returned error information, gather the following information that documents the error and contact IBM Customer Support.

Interface	Documentation
Local IPsec NMI	Collect a dump of the IKED address space.
Network security services (NSS) NMI	Collect a dump of the NSSD address space.
Real-time TCP/IP network monitoring NMI	- Set the SYSTCPIP MISC trace as active. - Collect a dump of the TCP/IP address space and data space.
Packet and data trace formatting NMI	Collect a dump of the TCP/IP address space and data space.
TCP/IP network management NMI (EZBNMIFR)	Collect a dump of the TCP/IP address space and data space.
SNA network monitoring NMI	Collect a dump of the VTAM address space.

File storage locations

The following table shows parts that are needed in order to compile Network management interface (NMI) applications and their locations. Your compiler should be configured to have access to these libraries.

Table 92. File storage locations

Function	File name	Type	Library
Allow applications to capture data packets	EZBYTMIA (1)	MACRO	SEZANMAC
	EZBYTMIH (1)	H	
Packet and data trace formatting NMI	EZBCTAPI	MACRO	SEZANMAC
	EZBYPTO	MACRO	
	EZBYPTHA	MACRO	
	EZBCTHDR	MACRO	
	EZBYCTHH	H	
	EZBYPTHH	H	
Allow applications to obtain TCP connection information	EZBYTMIA (1)	MACRO	SEZANMAC
	EZBYTMIH (1)	H	SEZANMAC
	EZASMF77	MACRO	SYS1.MACLIB
	EZASMF	H	SEZANMAC
TCP/IP callable NMI to retrieve local TCP and UDP endpoint data	EZBNMRHA	MACRO	SEZANMAC
	EZBNMRHC	H	
TCP/IP callable NMI to retrieve new TCP/IP storage statistics details	EZBNMRHA	MACRO	SEZANMAC
	EZBNMRHC	H	
Enterprise Extender Network Management	ISTEEHNC	H	SYS1.MACLIB
	ISTEESUC	H	
	ISTEECOC	H	
	ISTHPRCC	H	
	ISTCSMGC	H	
	ISTEEHNA	MACRO	
	ISTEESUA	MACRO	
	ISTEECOA	MACRO	
	ISTHPRCA	MACRO	
	ISTCSMGA	MACRO	
Real-time TCP/IP network monitoring NMI	EZBYTMIA (1)	MACRO	SEZANMAC
	EZBYTMIH (1)	H	
	EZANMFTA	MACRO	
	EZANMFTC	H	
Local IPSec NMI	EZBNMSEA	MACRO	SEZANMAC
Local IPSec NMI	EZBNMSEC	H	/usr/include
Network security services (NSS) NMI	EZBNMSEA	MACRO	SEZANMAC

Table 92. File storage locations (continued)

Function	File name	Type	Library
Network security services (NSS) NMI	EZBNMSEC	H	/usr/include
(1) Part used for multiple functions.			

Chapter 15. Application Transparent Transport Layer Security (AT-TLS)

Application Transparent Transport Layer Security (AT-TLS) creates a secure session on behalf of an application. Instead of implementing TLS in every application requiring a secure connection, AT-TLS provides encryption and decryption of data based on policy statements coded in the Policy Agent. The application sends and receives cleartext (unencrypted data) as usual while AT-TLS encrypts and decrypts data at the TCP transport layer. For more information about AT-TLS and AT-TLS policy setup, see the Application Transparent Transport Layer Security (AT-TLS) information in the *z/OS Communications Server: IP Configuration Guide* and the Policy Agent information in the *z/OS Communications Server: IP Configuration Reference*.

Most applications do not need any awareness of the security negotiations and encryption done by TCP/IP on its behalf. However, you might want some applications to be aware of AT-TLS or have control over the security functions being performed by TCP/IP. For example, if the application is a server requesting client authentication, you might want the application to get the partner certificate or the user ID associated with the partner certificate. Or the application might negotiate in cleartext with its partner to decide whether a secure session is necessary. If both agree to a secure session, then the application needs to tell AT-TLS to set up a secure session. The `SIOCTTLSCTL` ioctl provides the interface for the application to query or control AT-TLS.

Applications that are taking advantage of AT-TLS can be separated into three different types (basic, aware and controlling) as described in Table 93. An application's type is based on whether an awareness of the service is needed and, if so, the amount of control that the application is given over the security functions. Basic applications are unchanged. Aware applications are changed to invoke the `SIOCTTLSCTL` ioctl to query a socket about AT-TLS status using a `TTLSi_Req_Type` value of `TTLS_QUERY_ONLY` or `TTLS_RETURN_CERTIFICATE`. Controlling applications are changed to invoke the `SIOCTTLSCTL` ioctl to control the secure session on a socket using a `TTLSi_Req_Type` value of `TTLS_INIT_CONNECTION`, `TTLS_RESET_SESSION` or `TTLS_RESET_CIPHER`.

Table 93. Application types

Application type	SIOCTTLSCTL ioctl calls issued	ApplicationControlled setting in AT-TLS policy
Basic	application does not issue any AT-TLS ioctl calls	Off
Aware	query requests	Off
Controlling	query and control requests	On

- A basic application is unaware that AT-TLS is performing encryption or decryption of data. Most applications can match this model.
- An aware application is aware of AT-TLS and can query information such as AT-TLS status, partner certificate, and derived RACF user ID without any advanced setting in AT-TLS policy. A server that requires a RACF user ID derived from a partner certificate matches this model.

- A controlling application is aware of AT-TLS and needs to control the secure session. It must have the ApplicationControlled parameter in AT-TLS policy set to **ON**. Any application that must control when the initial handshake is done or when sessions or ciphers must be reset matches this model.

The SIOCTTLCTL ioctl blocks during the initial handshake if the socket is in blocking mode. If the socket is non-blocking, SIOCTTLCTL returns EWouldBlock during the initial handshake.

Applications that use non-blocking sockets can use the select function to wait for the socket to become writable. When the socket becomes writable, the initial handshake is complete.

The following APIs are supported by AT-TLS:

- Macro API (EZASMI)
- CALL instruction API (EZASOKET) supporting COBOL, PL/I, and System/370™ Assembler languages
- REXX socket API
- Language Environment C socket call [ioctl()]
- UNIX System Services Assembler Callable Service (BPX1IOC or BPX4IOC)
- CICS® C socket calls
- CICS CALL instruction API (EZASOKET - by including EZACICAL or EZACICSO)
- IMS™ CALL instruction API (EZASOKET)

Restrictions: The following APIs are not supported by AT-TLS:

- TCP C Socket API
- X/Open Transport Interface (XTI)
- Pascal API

CICS transaction considerations

CICS transaction security environments are not visible to AT-TLS support. The CICS job and all of its transactions appear to the stack as a single server application with a single z/OS UNIX callable services process ID running in the security environment of the CICS job. Connections established, whether active or passive, can perform TLS handshake processing as either CLIENT or SERVER. All of the connections that are established by a single CICS job are able to share the Session ID cache in the SSL environment. The CICS job should use a private keyring with a server certificate. The keyring used must contain the chain of root certificates needed to validate the server certificate it presents to the client. If the server requires the CLIENT AUTHENTICATION call, it must also have any other root certificates necessary to validate presented client certificates on its keyring.

TCP/IP CICS Socket Support provides a Listener transaction that has a configuration option to get the client's certificate-associated user ID. When this option is configured, the Listener waits for the TLS handshake to complete on the accepted connection (select for write) and then uses the SIOCTTLCTL ioctl to see whether an associated user ID is present. A user ID is present when the HandshakeRole parameter is defined in AT-TLS policy as ServerWithClientAuth, the client passed in a certificate, and the certificate was registered with RACF with an associated user ID. This user ID is passed into the Listener security exit, if one

is configured. The security exit can remove or change the user ID. The Listener then starts the transaction to process the connection under this user ID.

A CICS transaction that participates in a TLS handshake as CLIENT when the server requests CLIENT AUTHENTICATION presents a certificate identifying the CICS job, not the transaction user.

See the Application Transparent Transport Layer Security (AT-TLS) information in the *z/OS Communications Server: IP CICS Sockets Guide* for more information on configuring TCP/IP CICS socket support.

Using the SIOCTTLCTL ioctl

An application uses the SIOCTTLCTL ioctl to query AT-TLS information for a connection and to control the use of AT-TLS on a connection.

Starting AT-TLS on a connection

Use the SIOCTTLCTL ioctl with option TTLS_INIT_CONNECTION to start AT-TLS on a connection. This starts the SSL handshake. If using non-blocking sockets, the server can wait for the handshake to complete by waiting for the socket to become writable. If using blocking sockets, the ioctl blocks until the handshake is complete. If the handshake times out or fails for any reason, the connection is reset.

Some server applications need to support some clients using cleartext security negotiation and other clients using implicit security. This means that the SSL handshake starts as soon as the connection is established with the server. For server applications that support both types of clients, the TTLS_ALLOW_HSTIMEOUT option is helpful. This option enables the server to request an SSL handshake and keep the TCP connection active if the SSL handshake times out. This option is most effective if the server normally sends data to the client first. The server application must request both the TTLS_INIT_CONNECTION and the TTLS_ALLOW_HSTIMEOUT option on the SIOCTTLCTL start handshake request to keep the connection active after an SSL handshake timeout.

The server application waits for the SSL handshake to complete, either by blocking the socket or by waiting for the socket to become writable. After the handshake completes, the server application can check the SIOCTTLCTL status to determine the state of the connection, the protocol and cipher used, and other information. If a non-blocking socket is used, the final status is queried by issuing another SIOCTTLCTL ioctl with option TTLS_QUERY_ONLY. If a blocking socket is used, the final status is contained in the returned SIOCTTLCTL. Ensure that your server application checks the SIOCTTLCTL status and takes appropriate action based on the returned status.

Restriction: The TTLS_ALLOW_HSTIMEOUT option is supported only when the HandshakeRole value is Server or ServerWithClientAuth and the HandshakeTimer value is non-zero.

Stopping AT-TLS on a connection

Use the SIOCTTLCTL ioctl with option TTLS_STOP_CONNECTION to stop secure traffic on the TCP connection. The SSL session ends on the connection and the TCP connection returns to cleartext communication. The connection retains the

policy mapping, but the connection is in the same state as before a SIOCTTLCTL ioctl with the TTLS_INIT_CONNECTION option was issued.

Applications that negotiate security can use this option to stop the secure connection. For example, an application negotiates, using cleartext, that a secure session needs to be established. Later, the application performs a separate negotiation to stop the secure connection. After both sides agree to stop security, the application issues the SIOCTTLCTL ioctl with option TTLS_STOP_CONNECTION. The application must clear all application data from the connection before issuing the TTLS_STOP_CONNECTION request. The connection is reset if any unread application data exists when the application issues the TTLS_STOP_CONNECTION request. If nonblocking sockets are used, the application can wait for the request to complete by waiting for the socket to become writable. If blocking sockets are used, the ioctl blocks until the request is complete. After the request completes, the connection state is NONSECURE.

Restriction: The TTLS_STOP_CONNECTION option cannot be used on SSLv2 connections.

Tip: Do not use the TTLS_STOP_CONNECTION option if the application is not going to send or receive any clear text data after the request completes. AT-TLS closes the SSL session when the application closes the TCP socket.

Requesting AT-TLS queries and additional functions

Use the TTLSHeader structure pointed to by the TTLSi_BufferPtr pointer to query additional information for the secure connection. The TTLSHeader structure can be used to obtain the TTLSRule, TTLSGroupAction, TTLSEnvironmentAction, TTLSConnectionAction names, and the partner certificate. The application can also provide a host name that is validated against the host name in the partner's certificate.

Steps for implementing an aware server application

To implement an aware server application, create or update the server application as follows:

1. If the server is using non-blocking sockets, the server should issue select on the new socket to wait for the socket to become writable, which indicates that the initial handshake is complete. If using blocking sockets, the select is not needed.
2. When the new socket is writeable the server can issue the SIOCTTLCTL ioctl with TTLSi_Req_Type set to TTLS_RETURN_CERTIFICATE to retrieve the certificate presented by the client (if provided). The ioctl should return with a policy status of TTLS_POL_ENABLED and a connection status of TTLS_CONN_SECURE. The server program can examine the negotiated session attributes and the certificate that is supplied by the client (if provided). If this certificate is registered with the security product and associated with a user ID, then the user ID fields are also returned in the ioctl data.

Steps for implementing a controlling server application

To implement a simple aware and controlling application as a server, create or update the server application as follows:

1. When a new connection is accepted, the server should issue an `SIOCTTLCTL` ioctl with `TTLSi_Req_Type` value set to `TTLS_QUERY_ONLY` to verify that policy is correctly set up for this connection. The ioctl should return a policy status of `TTLS_POL_APPLCNTRL` and a connection status of `TTLS_CONN_NOTSECURE`. This means that the security of the connection is application-controlled and that the connection is not yet secure. If any other status is returned, the application cannot initiate a secure session for the connection. See “Coding the `SIOCTTLCTL` ioctl” on page 623 for an explanation of all status values. If you are sure the connection will be set for application control, this step can be omitted.

2. The server and client send and receive cleartext data to negotiate the use of TLS. The negotiation protocol is the responsibility of the applications and is not performed by the stack. Ensure that your negotiation protocol causes all cleartext data to be read on both ends before continuing.

3. If both sides agree to use a secure connection, the server should issue an `SIOCTTLCTL` ioctl with `TTLSi_Req_Type` value set to `TTLS_INIT_CONNECTION` to start the handshake. The client must also initiate a secure connection at this time.

4. If using non-blocking sockets, the server can wait for the handshake to complete by waiting for the socket to become writable. If using blocking sockets, the ioctl will block until the handshake is complete.

5. If the server wants to verify that the session is now secure, it can issue an `SIOCTTLCTL` ioctl with `TTLSi_Req_Type` value set to `TTLS_QUERY_ONLY` to retrieve the negotiated session attributes. The ioctl should return a connection status of `TTLS_CONN_SECURE` along with additional information. To retrieve the certificate that is presented by the client (if one is provided), use the `TTLS` buffer with a `TTLISK_Certificate Get` request. If this certificate is registered with the security product and associated with a user ID, then the user ID fields are also returned in the ioctl data.

Steps for starting an aware or controlling server application

To start an aware or controlling server application, perform the following steps:

1. Code or modify existing AT-TLS policy to cover the TCP server port. Ensure that the `HandshakeRole` parameter in the rule is set to `Server` or `ServerWithClientAuth`. The action should specify one of the following:
 - `ApplicationControlled Off` for an aware server application (this is also the default).
 - `ApplicationControlled On` for a controlling server application.

2. Install the policy to the appropriate TCP stack or stacks using Policy Agent. If any policy errors are reported, you must correct them and reinstall the policy.

3. Issue the **pasearch** command to verify the policy rule and actions for the server.

4. Start the aware or controlling server application.

The client and server exchange data, which to them is cleartext, but which is automatically encrypted by AT-TLS.

If at some point a controlling server application should need to reset the secure session or the current cipher, it can issue an SIOCTTLSCTL ioctl with the TTLSi_Req_Type value set to TTLS_RESET_SESSION or TTLS_RESET_CIPHER.

In addition to connection status and policy status, the SIOCTTLSCTL ioctl also provides the following secure session attributes after the handshake completes: security type, SSL protocol in use, negotiated cipher in use, certificate associated user ID, and, if requested, partner certificate information. The certificate associated user ID is available when the the HandshakeRole parameter is specified in AT-TLS policy as ServerWithClientAuth and the partner-supplied certificate has an associated user ID.

Additional SIOCTTLSCTL request type options are available to reset an SSL session ID so that it is not reused by this or another connection and to reset the cipher used by the SSL session. AT-TLS policy also provides ways for these resets to occur automatically based on elapsed time. Resets requested by issuing the SIOCTTLSCTL ioctl reset the beginning of the elapsed time intervals specified in the policy.

Figure 32, Figure 33 on page 623, and Figure 34 on page 623 show when the SIOCTTLSCTL ioctl can be issued and which AT-TLS functions are performed based on the ioctl call. The SIOCTTLSCTL ioctl initiates AT-TLS obtaining the user ID and job name of the requesting application and performing an AT-TLS rule search if these steps have not already been performed. An AT-TLS policy lookup assigns a rule and actions to the connection if a match is found. An SIOCTTLSCTL ioctl with a TTLS_Init_Connection request type initiates a System SSL environment search or creates a new environment and initiates the SSL connection (which consists of the SSL handshake). Policy and connection status fields are returned on all SIOCTTLSCTL ioctls.

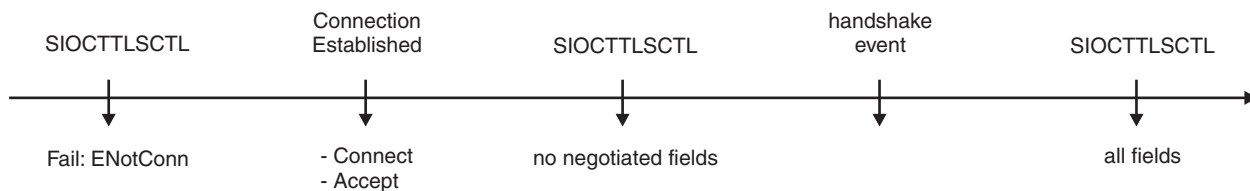


Figure 32. SIOCTTLSCTL with TTLS_Query_Only

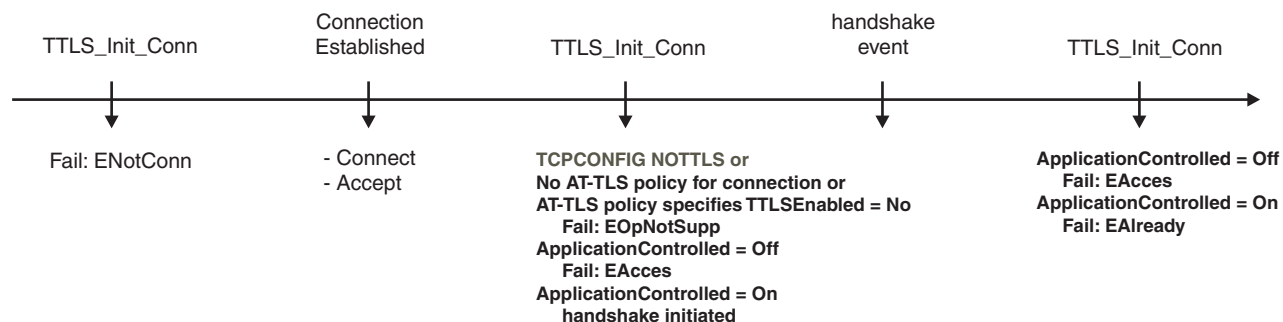


Figure 33. SIOCTTLCTL with TTLS_Init_Conn

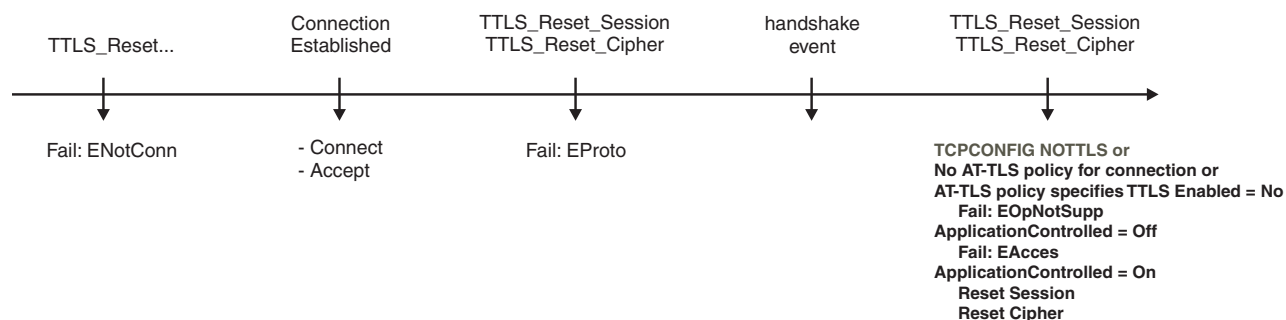


Figure 34. SIOCTTLCTL with TTLS_Reset_Session or TTLS_Reset_Cipher

Coding the SIOCTTLCTL ioctl

General coding guidelines

General coding guidelines for the sockets ioctl calls can be found in the following publications:

- *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*
 - Macro API (EZASMI) for assembler programs
 - CALL instruction API (EZASOKET) supporting COBOL, PL/I, and System/370 Assembler languages
 - REXX socket API
- *z/OS Communications Server: IP CICS Sockets Guide*
 - CICS C socket calls (EZACIC07 or EZACIC17, which calls EZASOKET with entry in EZACICAL)
 - CICS CALL instruction API (EZASOKET with entry in EZACICAL) supporting COBOL, PL/I, and System/370 Assembler languages
- *z/OS Communications Server: IP IMS Sockets Guide*
 - IMS CALL instruction API (EZASOKET) supporting COBOL, PL/I, and System/370 Assembler languages
- *z/OS XL C/C++ Run-Time Library Reference*
 - z/OS IBM C/C++ sockets API within the z/OS Language Environment
- *z/OS UNIX System Services Programming: Assembler Callable Services Reference*
 - Assembler Callable Services (BPX1IOC or BPX4IOC)

Each programming language has its own control block structure mapping. All mappings and header files are stored in SEZANMAC and the C language headers are also installed in file system directory /usr/include. The following programming languages are supported:

Assembler

Include EZBZTLSP mapping. See the *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference* for coding an ioctl call in assembler or coding an ioctl call for a callable API.

See the *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for coding BPX1IOC or BPX4IOC.

PL/I Include EZBZTLS1 mapping. See the *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*, *z/OS Communications Server: IP CICS Sockets Guide*, or *z/OS Communications Server: IP IMS Sockets Guide* for coding an ioctl in PL/I.

COBOL

Include EZBZTLSC mapping. See the *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*, *z/OS Communications Server: IP CICS Sockets Guide*, or *z/OS Communications Server: IP IMS Sockets Guide* for coding an ioctl call in COBOL.

REXX No mapping or header file used. See the *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference* for coding an ioctl call in REXX.

C Include EZBZTLSC header file, which is installed in SEZANMAC and in the file system directory /usr/include. See the *z/OS XL C/C++ Run-Time Library Reference* or *z/OS Communications Server: IP CICS Sockets Guide* for coding an ioctl call in C.

See the control block structures in SEZANMAC and in the /usr/include directory for variable names and locations and their enumerated values.

- All ioctl calls must set the Version field. See the control block structures in SEZANMAC for constants declared for these variables.

Guideline: The TTLS_CURRENT_VERSION constant in EZBZTLSC.h is being deprecated and will remain defined as 1. Use a specific TTLS version level constant, such TTLS_VERSION2, to set the TTLSi_Ver level when coding SIOCTTLCTL Ioctl requests.

- Any field not used must be set to 0.
- If the additional buffer is required for the SIOCTTLCTL ioctl, the buffer pointer is specified in *TTLSi_BufferPtr* and must point to the beginning of the buffer. The length of the buffer area is specified in *TTLSi_BufferLen*. Obtain enough buffer storage to hold the returned data. The buffer area can be part of the storage obtained for the base ioctl or it can be stand-alone storage. If the buffer is not large enough, the variable errno indicates the value *ENOBUFFS* with the required buffer size specified in *TTLSi_Cert_Len*.

SIOCTTLCTL (X'C038D90B')

The SIOCTTLCTL ioctl provides an interface for the application to query and control AT-TLS for the connection. The following data items are returned by the SIOCTTLCTL ioctl:

TTLSi_Stat_Policy

Indicates the level of AT-TLS enablement for the connection. Possible values include the following:

TTLS_POL_OFF (1)

AT-TLS was not enabled on the stack when AT-TLS policy mapping was performed for the connection

TTLS_POL_NO_POLICY (2)

No matching policy rule was found when AT-TLS policy mapping was performed for the connection. The application should issue a message, if appropriate, directing the system administrator to create a policy rule that matches this connection.

TTLS_POL_NOT_ENABLED (3)

The policy rule that matches this connection indicates that AT-TLS should not be used. The application should issue a message, if appropriate, directing the system administrator to change the policy rule for this connection.

TTLS_POL_ENABLED (4)

AT-TLS is enabled for this connection, but application control has not been granted. The application should issue a message, if appropriate, directing the system administrator to change the policy rule to enable application control for this connection.

TTLS_POL_APPLCNTRL (5)

AT-TLS is enabled and is application controllable.

TTLSi_Stat_Conn

Indicates the current level of secure session on the connection.

TTLS_CONN_NOTSECURE (1)

The connection does not have a secure session established.

TTLS_CONN_HS_INPROGRESS (2)

Connection initial handshake in progress.

TTLS_CONN_SECURE (3)

The connection has a secure session.

In addition to connection status and policy status, the SIOCTTLSCTL ioctl also provides the following connection information, when available:

TTLSi_Sec_Type

Indicates the security type for the connection if AT-TLS policy is defined for the connection. Valid values are:

TTLS_SEC_UNKNOWN (0)

The connection does not have a secure session established.

TTLS_SEC_CLIENT (1)

The security type is Client.

TTLS_SEC_SERVER (2)

The security type is Server.

TTLS_SEC_SRV_CA_PASS (3)

The security type is Server with Client Authentication. Client Authentication Type is PassThru.

TTLS_SEC_SRV_CA_FULL (4)

The security type is Server with Client Authentication. Client Authentication Type is Full.

TTLS_SEC_SRV_CA_REQD (5)

The security type is Server with Client Authentication. Client Authentication Type is Required.

TTLS_SEC_SRV_CA_SAFCHK (6)

The security type is Server with Client Authentication. Client Authentication Type is SAFCheck.

TTLSi_SSL_Prot

Indicates the SSL protocol in use for the connection if the connection is secure. Valid values are:

TTLS_PROT_UNKNOWN (0x0000)

The connection does not have a secure session established.

TTLS_PROT_SSLV2 (0x0200)

SSL version 2 is in use.

TTLS_PROT_SSLV3 (0x0300)

SSL version 3 is in use.

TTLS_PROT_TLSV1 (0x0301)

TLS version 1 is in use.

TTLSi_Neg_Cipher

Indicates the cipher in use for the connection if the connection is secure. For the list of cipher suites that are supported, see the `gsk_environment_open()` API information in *z/OS Cryptographic Service System Secure Sockets Layer Programming*.

TTLSi_UserID/TTLSi_UserID_Len

TTLSi_UserID is a null terminated character string.

TTLSi_UserID_Len indicates the number of characters returned prior to the first null.

These fields are returned when the `HandshakeRole` parameter is specified as `ServerWithClientAuth`, the client provides a valid certificate, and the certificate is associated with a user ID in the DIGTCERT General Resource Class. See the RACF and Digital Certificates topic in the *z/OS Security Server RACF Security Administrator's Guide* for more information on associating user IDs with certificates.

TTLSi_Cert_Len

Indicates the size of the partner's certificate if the connection is secure and a certificate was supplied during negotiation.

If the `TTLS_RETURN_CERTIFICATE` request type is specified on the `SIOCTTLCTL` ioctl and the partner certificate is known, the certificate is returned in the additional buffer provided (using fields *TTLSi_BufferPtr* and *TTLSi_BufferLen*) with the ioctl call. The length of the returned certificate is returned in *TTLSi_Cert_Len*. If the buffer provided is not large enough to hold the certificate, then the variable `errno` indicates the value `ENOBUFFS` and the required buffer size is returned in *TTLSi_Cert_Len*.

The following optional behaviors can be requested on the `SIOCTTLCTL` ioctl using the `TTLSi_Req_Type` field:

TTLS_QUERY_ONLY (0x0000)

Query the connection status. If more advanced query information is required, use the optional buffer that includes the `TTLShdr` control block.

Restriction: The TTLS_QUERY_ONLY option must be specified alone without any other request option.

TTLS_RETURN_CERTIFICATE (0x0001)

Return the partner certificate used for authentication if it is available.

Restriction: This request is not valid with TTLS Version 2. For TTLS version 2, use the TTLSHeader structure to request the partner certificate.

TTLS_INIT_CONNECTION (0x0002)

Initialize the secure SSL connection using the role defined by the HandshakeRole parameter.

Restriction: The connection must be application controlled to use this request.

TTLS_RESET_SESSION (0x0004)

Reset a session ID to avoid its reuse by another connection.

Restriction: The connection must be application controlled to use this request.

TTLS_RESET_CIPHER (0x0008)

Reset and renegotiate the cipher used for the secure session. If the session ID has timed out or has been reset, a full handshake is performed. Otherwise, a short handshake is performed.

Restriction: The connection must be application controlled to use this request.

TTLS_STOP_CONNECTION (0x0010)

Close the SSL connection. Data will no longer be encrypted or decrypted on the connection. The state of the TCP connection is unchanged.

Restrictions:

- The application must read all secure application data before the TTLS_STOP_CONNECTION request is issued. The connection will be reset if any secure application data exists when the TTLS_STOP_CONNECTION request is issued.
- The connection must be application controlled to use this request.

TTLS_ALLOW_HSTIMEOUT (0x0020)

Allow the TCP connection to remain active if the SSL handshake fails because no data was received from the client. The timeout value is determined by the HandshakeTimeout value from the policy. The HandshakeTimeout must be a non-zero value. This option is valid only if the HandshakeRole value is Server or ServerWithClientAuth. Any non-SSL data received ends the handshake request and leave the connection state non-secure.

Restriction: The TTLS_INIT_CONNECTION option must be specified when the TTLS_ALLOW_HSTIMEOUT option is requested.

Tip: Use this option for servers that send the first application data to the client and must support SSL and non-SSL clients on the same port.

Using the TTLSHeader control block

The TTLSHeader control block extends the SIOCTTLSCtl ioctl. You can use Get requests to obtain additional information about the AT-TLS connection. You can use Set requests to change the AT-TLS behavior for the connection. TTLS Version 2 is required to use the TTLSHeader control block. The TTLSHeader control block includes a fixed section and a variable length self-defining section that contains Set

and Get requests followed by the set and get data area. The set data area is not overlaid with the Get request data when the SIOCTLCTL returns.

Rule: Set request quadruplets must immediately follow the TLSHeader control block in the buffer, followed by Get request quadruplets. The data for the Set requests must follow the set and get quadruplets. The data area for the Get requests must follow the set data area.

Table 94. TLSHeader control block structure

TLSHeader fixed section
TLSHeader optional Set request quadruplets
TLSHeader optional Get request quadruplets
Beginning of set data area
Beginning of get data area

TLSHeader Fixed section: The TLSHeader fixed section contains control information that describes the Set and Get requests that follow. The fixed section is defined by the TLSHeader structure as shown in Table 95.

Table 95. TLSHeader fixed section

Field	Offset	Length	Format	Description
TLSHeaderIdent	0	8 bytes	EBCDIC	TLSHeader identifier; set to TLSHdr_Ident(EBCDIC 'TSLSHDR ')
TLSHdrRsvd1	8	8 bytes	Binary	Reserved; set to 0
TLSHdr_BytesNeeded	16	4 bytes	Binary	Length of the buffer needed to contain TLSHeader, TLSQuadruplet structures, data for Set requests, and data for Get requests. This value is set on return if all data does not fit in the buffer provided
TLSHdr_SetCount	20	4 bytes	Binary	Number of Set requests in the buffer. Each Set request is represented by a TLSQuadruplet
TLSHdr_GetCount	24	4 bytes	Binary	Number of Get requests in the buffer. Each Get request is represented by a TLSQuadruplet
TLSHdrRsvd2	28	16 bytes	Binary	Reserved; set to 0

TLSHeader variable length structure: The variable length section is used to request options, such as the following:

- Validating a host name and get options
- Returning the partner certificate

Each Set or Get request is defined by a TLSQuadruplet structure as shown in Table 96.

Table 96. TLSQuadruplet structure

Field	Offset	Length	Format	Description
TLSQ_Key	0	4 bytes	Binary	Constant identifying the request
TLSQ_Offset	4	4 bytes	Binary	Offset to the first value for the request, measured in bytes from the start of the TLSHeader structure. For Set requests, this must be a nonzero value. For Get requests, this value must be 0. On return, this value is nonzero if data was returned for this request.

Table 96. *TTLSTriplet structure (continued)*

Field	Offset	Length	Format	Description
TTLSTriplet_Length	8	4 bytes	Binary	On input, this is the length of the value for a Set request or has the value 0 for a Get request. On return for a Get request, this value is the length of the data that is returned.
TTLSTriplet_Rcode	12	4 bytes	Binary	Return code of the request operation. See each request operation for the possible return codes.

Get request: Use the Get request to obtain additional information about the AT-TLS connection.

Table 97. *Get request structure*

Identifier	Constant	Length	Format	Description
TTLSTriplet_Host_Status	4000	1 byte	Binary	Validates the host name that is provided with the partner certificate. The host name must end with a null character. The TTLSTriplet_Offset field must be set to the value of the start of the host name that is in the buffer. The following values can be returned in the TTLSTriplet_Rcode field: 0 The host name has successfully validated against the partner certificate. 1 Partner certificate is not available. 2 Host name did not match the name in the partner certificate. 3 Host name validation failed with an unexpected gsk_validate_hostname value. 4 Host name validation failed with an unexpected gsk_decode_certificate value.
TTLSTriplet_Certificate	4001	Unknown	Binary	Returns the partner certificate. The certificate length is not known until the secure connection is established. The value of the TTLSi_Cert_Len field in the SIOCTTLSCTL structure can be used to determine the certificate length when the secure connection is complete. The following values can be returned in TTLSTriplet_Rcode: 0 The request completed successfully.

Table 97. Get request structure (continued)

Identifier	Constant	Length	Format	Description
TTL SK_TTLRule_Name	4002	48 bytes, ending with a null character	EBCDIC	Returns the name of the TTLRule field that is mapped to the connection. The following value can be returned in TTL SQ_Rcode: 0 The request completed successfully.
TTL SK_GroupAction_Name	4003	48 bytes, ending with a null character	EBCDIC	Returns the name of the TTLGroupAction field that is mapped to the connection. The following value can be returned in TTL SQ_Rcode: 0 The request completed successfully.
TTL SK_EnvironmentAction_Name	4004	48 bytes, ending with a null character	EBCDIC	Returns the name of the TTLEnvironmentAction field that is mapped to the connection. The following value can be returned in TTL SQ_Rcode: 0 The request completed successfully.
TTL SK_ConnectionAction_Name	4005	48 bytes, ending with a null character	EBCDIC	Returns the name of the TTLConnectionAction field that is mapped to the connection. The following value can be returned in TTL SQ_Rcode: 0 The request completed successfully.

For example, assume that an application made a secure connection to a server. However, the application needs to verify that the certificate is from the server the application is connected to. The application has two known server host names, mvs.telnet.raleigh.ibm.com and mvs.prod.rtp.ibm.com. The application would use the following TTL SHdr structure, pointed to by the TTL SI_BufPtr pointer on the SIOCTLCTL ioctl request to validate the server's certificate against these host names:

Table 98. Example TTL SHdr structure

TTL SHdr			
Field	Offset	Format	Value
TTL SHdrIdent	0	EBCDIC	TTL SHDR
TTL SHdrRsvd1	8	Binary	00000000 00000000
TTL SHdr_BytesNeeded	16	Binary	00000000
TTL SHdr_SetCount	20	Binary	00000000
TTL SHdr_GetCount	24	Binary	00000002
TTL SHdrRsvd2	28	Binary	00000000 00000000 00000000 00000000
TTL SQQuadruplet			Get Request buffer
TTL SQ_Key	48	Binary	00000FA0
TTL SQ_Offset	52	Binary	00000050
TTL SQ_Length	56	Binary	00000000
TTL SQ_Rcode	60	Binary	00000000

Table 98. Example *TTLShdr* structure (continued)

TTLShdr			
Field	Offset	Format	Value
TTLShdr			Get Request buffer
TTLShdr_Key	64	Binary	00000FA0
TTLShdr_Offset	68	Binary	0000006B
TTLShdr_Length	72	Binary	00000000
TTLShdr_Rcode	76	Binary	00000000
			Buffer Data
Hostname	80	EBCDIC	mvs.telnet.raleigh.ibm.com
Null character	106	Binary	00
Hostname	107	EBCDIC	mvs.prod.rtp.ibm.com
Null Character	127	Binary	00

Assuming that the certificate listed mvs.prod.rtp.ibm.com as the hostname value, the following *TTLShdr* structure would be returned to the application:

Table 99. Example returned *TTLShdr* structure

TTLShdr			
Field	Offset	Format	Value
TTLShdrIdent	0	EBCDIC	TTLSHDR
TTLShdrRsvd1	8	Binary	00000000 00000000
TTLShdr_BytesNeeded	16	Binary	00000080
TTLShdr_SetCount	20	Binary	00000000
TTLShdr_GetCount	24	Binary	00000002
TTLShdrRsvd2	28	Binary	00000000 00000000 00000000 00000000
TTLShdr			Get Request buffer
TTLShdr_Key	48	Binary	00000FA0
TTLShdr_Offset	52	Binary	00000050
TTLShdr_Length	56	Binary	00000000
TTLShdr_Rcode	60	Binary	00000001
TTLShdr			Get Request buffer
TTLShdr_Key	64	Binary	00000FA0
TTLShdr_Offset	68	Binary	0000006B
TTLShdr_Length	72	Binary	00000001
TTLShdr_Rcode	76	Binary	00000000
			Buffer Data
Hostname	80	EBCDIC	mvs.telnet.raleigh.ibm.com
Null character	106	Binary	00
Hostname	107	EBCDIC	mvs.prod.rtp.ibm.com
Null character	127	Binary	00

Return values

The following are possible return values:

0 Successful completion.

-1 An error occurred. Check the return code and reason code. The following are possible values:

EProtoType

Socket is not TCP.

EInval The error depends on the reason code. The following are possible reason codes:

JrInvalidVersion

Version not valid in TTLS_IOCTL data structure.

JrSocketCallParmError

Denotes one of the following conditions:

- TTLS_RETURN_CERTIFICATE request type is specified along with a zero value in either *TTLSi_BufferPtr* or *TTLSi_BufferLen*
- TTLS_RETURN_CERTIFICATE request type is specified and TTLS_Version is not set to 1
- TTLS_RETURN_CERTIFICATE request type is not specified along with a nonzero value in either *TTLSi_BufferPtr* or *TTLSi_BufferLen* and TTLS_Version is set to 1
- Request type is not valid.
- Length of input data is not length of ioctl structure.

EPerm Denotes one of the following error conditions:

- The TTLS_INIT_CONNECTION option was requested, along with one of the following:
 - TTLS_RESET_SESSION
 - TTLS_RESET_CIPHER
 - TTLS_STOP_CONNECTION
- The TTLS_STOP_CONNECTION option was requested along with the TTLS_RESET_SESSION or TTLS_RESET_CIPHER option
- The TTLS_ALLOW_HSTIMEOUT option was requested without the TTLS_INIT_CONNECTION option

ENotConn

The connection has not reached the established state or has been closed.

EPipe TTLS_INIT_CONNECTION, TTLS_STOP_CONNECTION, or TTLS_RESET_CIPHER option was requested and the connection is no longer in established state.

EMVSERR

Internal failure while mapping AT-TLS policy.

EOpNotSupp

The TTLS_INIT_CONNECTION, TTLS_STOP_CONNECTION, TTLS_RESET_SESSION, or TTLS_RESET_CIPHER option was requested and one of the following is true:

- TCPCONFIG NOTTLS is configured or is the default.
- The connection has no policy.

- The AT-TLS policy for the connection specifies TTLS-enabled=No.

EAccess

The TTLS_INIT_CONNECTION, TTLS_STOP_CONNECTION, TTLS_RESET_SESSION, or TTLS_RESET_CIPHER option was requested and the AT-TLS policy for the connection specifies ApplicationControlled=No.

EAlready

TTLS_INIT_CONNECTION was requested and the connection is already secure or TTLS_STOP_CONNECTION was requested and the connection is not secure.

EProto

Denotes one of the following reason codes:

JrGetConnErr

The TTLS_RESET_SESSION or TTLS_RESET_CIPHER option was requested and the connection is not secure.

JrInvalidVersion

The TTLS_RESET_CIPHER or TTLS_STOP_CONNECTION option was requested; the connection is secure but is SSLv2.

JrConnDeniedPolicy

The TTLS_ALLOW_HSTIMEOUT option was requested but the HandshakeRole value is client or the HandshakeTimeout value is 0.

EInProgress

The TTLS_INIT_CONNECTION or TTLS_STOP_CONNECTION option was requested and handshake is in progress.

EWouldBlock

The socket is a non-blocking socket and an SSL handshake is in progress.

ENoBufs

Denotes one of the following reason codes:

JrBuffTooSmall

- For TTLS_Version1, the TTLS_RETURN_CERTIFICATE option was requested and the buffer provided using *TTLSi_BufferPtr* field is too small. See the *TTLSi_Cert_Len* value for the number of bytes required to hold the certificate.
- For TTLS_Version 2, the buffer supplied was too small. See the *TTLSHdr_BytesNeeded* field value for the number of bytes required .

Coding examples

The following examples show sample code for building and issuing the SIOCTTLCTL ioctl.

Assembler example

The following sample assembler code builds an SIOCTTLCTL ioctl and issues the ioctl using the Macro API (EZASMI). The ioctl requests initialization of the secure connection and the return of the partner's certificate in the provided buffer.

```

...
*****
*
*      Issue SIOCTLSSLCTL IOCTL to start data encryption.
*      This requires AT-TLS policy with ApplicationControlled On
*      specified.
*
*****
* clear ioctl buffer and then set version
  XC  TTLS_IOCTL(TTLS_IOCTL_V1Len),TTLS_IOCTL
  MVI TTLSi_Ver,TTLS_VERSION1
  L   R8,=A(TTLS_INIT_CONNECTION)
  O   R8,=A(TTLS_RETURN_CERTIFICATE)
  STH R8,TTLSi_Req_Type
  LA  R8,BUFFERA
  ST  R8,TTLSi_BufferPtr
  MVC TTLSi_BufferLen,=A(L'BUFFERA)
*
  EZASMI TYPE=IOCTL,      ISSUE IOCTL MACRO
        S=SOCDESCA,
        COMMAND='SIOCTLSSLCTL',
        REQARG=TTLS_IOCTL,
        RETARG=TTLS_IOCTL,
        REQAREA=REQAREA, IN CASE WE ARE DOING EXITS OR ECBS
        ERRNO=ERRNO,     RETURN ERRNO HERE
        RETCODE=RETCODE,  RETURN RETCODE HERE
        ERROR=ERROR      ABEND IF MACRO ERROR
*
...

BUFFERA DS    CL1024
..
        EZBZTLSP DSECT=NO      TTLS ioctl structure
...

```

PL/I example

The following sample PL/I code builds and issues an SIOCTLSSLCTL ioctl that requests secure connection initialization and the return of the partner's certificate in the provided buffer.

```

...
/* get the SIOCTLSSLCTL ioctl mapping and constants */
% include EZBZTLS1;
...
/* area to return the certificate data if available */
DCL CERTIF CHAR(1000) INIT('B');
....

/* allocate storage for the SIOCTLSSLCTL ioctl */
Allocate TTLS_IOCTL;
TTLSI_VER = TTLS_VERSION1;
TTLSI_REQ_TYPE = TTLS_INIT_CONNECTION | TTLS_RETURN_CERTIFICATE;

/* if you DO NOT want to get the certificate then you must */
/* clear the following two fields */
/* TTLSi_BufferPtr = SYSNULL;
/* TTLSi_BufferLen = 0;

/* if you DO want to get the certificate then you must */
/* set the following two fields */
TTLSi_BufferPtr = ADDR(CERTIF);
TTLSi_BufferLen = LENGTH(CERTIF);

call ezasocket(IOCTL,
               SOCKET,
               SIOCTLSSLCTL, /* TTLS ioctl
               TTLS_IOCTL,   /* input buffer

```

```

        TTLS_IOCTL,    /* output buffer      */
        ERRNO,
        RETCODE);
if RETCODE < 0 then do;
    /* do failure logic. If the socket is in non-    */
    /* blocking mode then you may also receive RETCODE */
    /* of -1 with an ERRNO of EINPROGRESS. This does */
    /* not indicate an error. Wait for the completion */
    /* of the handshake with SELECT for WRITEABLE.    */
    ...
end;

```

COBOL example

The following sample COBOL code builds and issues an SIOCTTLSSLCTL ioctl that requests the initialization of the secure connection.

```

...
    *****
    Data Division.
    *****
    *-----*
    * Variables used by the SIOCTTLSSLCTL IOCTL call *
    *-----*
    01  ttls-ioctl-data.
        COPY EZBZTLSCB.
...
    *****
    Procedure Division.
    *****
...
    TTLS-Init.
        move low-values to ttls-ioctl-data.
        set TTLSI-BUFFERPTR to NULL.
        move TTLS-VERSION1 to TTLSI-VER.
        move TTLS-INIT-CONNECTION to TTLSI-REQ-TYPE.
        Call 'EZASOCKET' using socket-ioctl socket-descriptor-new
            SIOCTTLSSLCTL
            TTLS_IOCTL TTLS_IOCTL
            errno retcode.
    * if error other than EINPROGRESS then *
    * IF ((RETCODE < 0) AND (ERRNO NOT = EINPROGRESS)) THEN *
    * handle error here *
    * ELSE *
    * normal case *
    *
    TTLS-INIT-Exit.
        Exit.

```

C example

The following sample C code builds and issues an SIOCTTLSSLCTL ioctl that requests initialization of the secure connection and the return of the partner's certificate in the provided buffer.

```

...
#include "ezbztls.h"                /* SIOCTTLSSLCTL ioctl */
...
struct TTLS_IOCTL ioc;              /* ioctl data structure */
char buff[1000];                   /* buffer for certificate */
...
/* issue the SIOCTTLSSLCTL ioctl */
memset(&ioc,0,sizeof(ioc));         /* set all unused fields to zero */
ioc.TTLSi_Ver = TTLS_VERSION1;
ioc.TTLSi_Req_Type = TTLS_INIT_CONNECTION | TTLS_RETURN_CERTIFICATE;
ioc.TTLSi_BufferPtr = &buff;
ioc.TTLSi_BufferLen = sizeof(buff);

rc = ioctl(s,SIOCTTLSSLCTL,(char *)&ioc);

```

```

if (rc < 0)
{
    /* do failure logic. If the socket is in non-    */
    /* blocking mode then you may also receive rc    */
    /* of -1 with an errno of EINPROGRESS. This does */
    /* not indicate an error. Wait for the completion */
    /* of the handshake with select() for WRITEABLE. */
}

```

Chapter 16. Interfacing with the Digital Certificate Access Server (DCAS)

This topic documents the programming interface specifications for the Digital Certificate Access Server (DCAS) that runs on z/OS.

Overview

The DCAS can be used by providers of logon and single sign-on services where access to z/OS-based applications is needed. The DCAS is a TCP/IP server that enables clients to connect over the network and obtain a passticket and z/OS user ID from a SAF-compliant product, such as RACF. This topic refers to RACF as the SAF product.

A passticket is like a temporary password, because it is valid for only a short period of time. Applications on z/OS can be configured to support logon access with passtickets. For information about passtickets, see the *z/OS Security Server RACF Security Administrator's Guide*.

IBM does not provide header files or samples for programming DCAS clients, but the specifications for developing a client are defined in the following sections.

Understanding how clients interface to DCAS

Clients connect to DCAS using the TCP protocol.

- By default, the DCAS listens on port 8990, but it can listen on any configured port.
- Clients that connect to DCAS must use the SSL protocol (DCAS supports SSL Version 3). Client authentication is performed.
- DCAS provides a request and response interface that enables clients to obtain two types of information. After the TCP connection and SSL handshake processing completes, the DCAS client sends a request and in turn receives a response. The request and its response determine which of the following types of information DCAS provides:
 - Clients can request a user ID and passticket for an application. The client sends a Format 1 type request that includes an application ID1 and an x.509 certificate. DCAS returns a user ID and passticket in the Format 1 response. In this case, DCAS converts the x.509 certificate to a valid user ID, which is returned.
Requirement: The x.509 certificate must have been mapped to a valid user ID in RACF.
 - Clients can request a passticket for an application. The client sends a Format 2 type request that includes an application ID1 and a user ID. DCAS returns a passticket in the Format 1 response.

See the Configuring RACF services for Express Logon information in the *z/OS Communications Server: IP Configuration Guide* for more information about passtickets.

The request and response formats are described in the following section.

Where text is required in the formatted request, DCAS requires that they are encoded in EBCDIC (IBM-1047 codepage). Responses that contain text are also encoded in EBCDIC (IBM-1047 codepage).

If the request from the client was not processed successfully, DCAS returns error code information in the response. The client must be designed to examine this information.

Defining the format for request and response specifications

Table 100 contains format 1 request information.

Table 100. Format 1 request

Field byte offset	Field name	Field description
0	opcode	01 = request
1	Format	01 = request user ID and passticket
2-5	Correlator	User-defined value
6-25	Appl ID	Application for which the passticket is generated. This must have the same name as the passticket data profile that is defined for the application using the RACF PTKTDATA class. ¹ (EBCDIC).
26-27	reserved	not used
28-31	Certificate Length	Input certificate length. Maximum length is 32 767 bytes. This field is a binary integer.
32- <i>n</i>	Certificate	Base-64 encoded certificate

¹ The application ID required in the DCAS Format 1 and Format 2 requests must match the name of a valid passticket data profile defined in RACF using the PTKTDATA class. See the *z/OS Security Server RACF Security Administrator's Guide* for information about defining PTKTDATA for applications.

Table 101 contains format 1 response information.

Table 101. Format 1 response

Field byte offset	Field name	Field description
0	opcode	02 = response
1	Format	01 = request user ID and passticket
2-5	Correlator	User-defined value that matches the value of the request.
6-7	Return Code 1	If nonzero, examine the extended return codes: Return Code 2, Return Code 3, Return Code 4
8-11	Return Code 2	Extended (see Table 103 on page 639)
12-15	Return Code 3	Extended (see Table 103 on page 639)
16-19	Return Code 4	Extended (see Table 103 on page 639)
20-28	User ID	If Return Code 1 is 0, a user ID is returned (EBCDIC)
29	reserved	null
30-37	Passticket	If Return Code 1 is 0, a passticket is returned.

Table 102 contains format 2 request information.

Table 102. Format 2 request

Field byte offset	Field name	Field description
0	opcode	02 = request
1	Format	02 = request passticket
2-5	Correlator	User-defined value
6-25	Appl ID	Application for which the passticket is generated. Must have the same name as the passticket data profile that is defined for the application using the RACF PTKTDATA class. ¹ (EBCDIC)
26-27	reserved	Not used
28-31	User ID Length	Length of the input user ID (binary integer)
32- <i>n</i>	User ID	Input user ID (EBCDIC)
¹ The response to a Format 2 request is a Format 1 Response. The application ID required in the DCAS Format 1 and Format 2 requests must match the name of a valid passticket data profile defined in RACF using the PTKTDATA class. See the <i>z/OS Security Server RACF Security Administrator's Guide</i> for information about defining PTKTDATA for applications.		

Table 103. Understanding return codes in the response

Return Code 1	Return Code 2	Return Code 3	Return Code 4	Comments
0	Not Set	Not Set	Not Set	The response indicates that the request completed successfully.
250	Not Set	Not Set	Not Set	An internal error occurred on the DCAS server. Request that the system operator obtain a DCAS trace. See the <i>z/OS Communications Server: IP Diagnosis Guide</i> for instructions.
251	Not Set	Not Set	Not Set	Passticket generation failed. The most likely cause is that the application ID in the DCAS Format 1 or 2 request does not match a valid passticket data profile name defined in the RACF PTKTDATA class. ¹
252	8	8	36 – Certificate is not valid. 40 – Certificate is not mapped to a valid user ID.	For a Format 1 type request, RACF has determined that the input certificate is in error or has not been mapped to a valid RACF user ID. For return codes other than the ones described, see the <i>z/OS Communications Server: IP Diagnosis Guide</i> .

Table 103. Understanding return codes in the response (continued)

Return Code 1	Return Code 2	Return Code 3	Return Code 4	Comments
253	10 – Format 1 request has a certificate length that is not valid. 11 – The request format is incorrect. 12 – The opcode that is specified in the request is not valid.	Not Set	Not Set	The input format 1 or 2 request is incorrect. Examine Return Code 2 for details. Verify that the input request to DCAS matches the defined format specifications. Verify that DCAS is configured with a SERVERTYPE in the DCAS profile that is consistent with the input request format.
254	8	8	36 – Certificate is not valid. 40 – Certificate is not mapped to a valid user ID.	DCAS failed to authenticate the client. The DCAS server has been configured with AUTHTYPE LOCAL2. This requires that the certificate of the DCAS client (as a result of the SSL handshake) be mapped to a defined and valid user ID in RACF. The user ID must be permitted to the following SERVAUTH class profile: EZA.DCAS.cvtsysname. If the DCAS client receives this error, then the user ID is not permitted to the defined SERVAUTH class profile. For return codes other than the ones described, see the Diagnosing problems with Express Logon information in the <i>z/OS Communications Server: IP Diagnosis Guide</i> for diagnosing DCAS.
255	8	8	36 – Certificate is not valid. 40 – Certificate is not mapped to a valid user ID.	DCAS failed to authenticate the client. The DCAS server has been configured with AUTHTYPE LOCAL2. This requires that the certificate of the DCAS client (as a result of the SSL handshake) be mapped to a defined and valid user ID in RACF. If the DCAS client receives this error, then the certificate does not map to a valid user ID.
¹ The application ID required in the DCAS Format 1 and Format 2 requests must match the name of a valid passticket data profile defined in RACF using the PTKTDATA class. See the <i>z/OS Security Server RACF Security Administrator's Guide</i> for information about defining PTKTDATA for applications.				

Configuring the DCAS server to work with your solution

When interfacing to DCAS as a provider of logon services, work with the system administrator to verify that DCAS is configured to work with your solution. For more information about configuring DCAS, see the EXPRESS LOGON using DCAS (Digital Certificate Access Server) information in the *z/OS Communications Server: IP Configuration Reference*. The DCAS configuration statements described in Table 104 require coordination between the DCAS client and server.

Table 104. DCAS client and server coordination

DCAS client interface	DCAS server configuration statement	Description
Input Request Format	SERVERTYPE options	ALLTYPES – Allows Format 1 and Format 2 input requests to be accepted by the DCAS server. CERTTYPE – Allows only Format 1 requests. USERIDTYPE – Allows only Format 2 requests.
SSL connection parameters	V3CIPHER	The DCAS client must communicate with the DCAS server using SSL. The V3CIPHER allows for specification of the cipher.
TCP port used for connection	PORT	The default DCAS listening port is 8990 but DCAS can be configured to use any port.

Chapter 17. Miscellaneous programming interfaces

SIOCSAPPLDATA IOCTL

The SIOCSAPPLDATA ioctl enables applications to associate 40 bytes of application-specific information with TCP sockets they own. This information can assist problem determination, capacity planning, and accounting applications. This ioctl supports both 31-bit and 64-bit addressing modes.

This application-specific information, which is referred to as ApplData, is available from the following sources:

- In the Netstat ALL/-A, ALLConn/-a, and COnn/-c reports where it can be searched using the APPLD/-G filter. See the *z/OS Communications Server: IP System Administrator's Commands* for additional information on using ApplData information with Netstat.
- In the SMF 119 TCP connection termination record. See the *z/OS Communications Server: IP Configuration Reference* for additional information.
- Through the callable TCP/IP network management interface. See "TCP/IP callable NMI (EZBNMIFR)" on page 583 for more information.

The SIOCSAPPLDATA IOCTL constant and data structures for assembler applications are defined in the EZBYAPPL macro in the SEZANMAC data set, and for C/C++ applications they are defined in the EZBYAPLC header file in the SEZANMAC data set and the /usr/include file system directory.

Consider the following guidelines when using this ioctl:

- The application is responsible for documenting the content, format, and meaning of the ApplData strings that it might associate with sockets it owns.
- The application should uniquely identify itself with printable EBCDIC characters at the beginning of the string. Strings beginning with 3-character IBM product identifiers, such as TCP/IP's EZA or EZB, are reserved for IBM use. IBM product identifiers begin with a letter in the range A – I.
- You should use printable EBCDIC characters for the entire string to enable searching with Netstat filters.

Input

Input is provided using a pointer to a SetApplData structure, which in turn defines the version, size, and location of the SetADcontainer structure that contains the application data to be associated with the stream socket.

Table 105. SetApplData

Field name	Size	Description
SetAD_eye1	8	constant SETADEYE1
SetAD_ver	4	constant SETADVER
SetAD_len	4	sizeof(SetADcontainer)
SetAD_ptr	8	A bimodal pointer to a SetADcontainer structure. In 31-bit addressing mode the first 4 bytes are reserved and should be 0; the second 4 bytes contain the 31-bit address.

Table 106. SetADcontainer

Field name	Size	Description
SetAD_eye2	8	constant SETADEYE2
SetAD_buffer	40	A character buffer that contains the data to associate with this end of the connection. This buffer should be padded on the right with space characters.

Output

The SIOCSAPPLDATA IOCTL sets the following return codes and reason codes:

Table 107. SIOCSAPPLDATA IOCTL return and reason codes

ReturnValue	ReturnCode	ReasonCode	Meaning
0	0	0	The request was successful.
-1	EProtoType	JrSocketTypeNotSupported	The request was not successful. The socket is not a stream (TCP) socket.
-1	EINVAL	JrSocketCallParmError	The input parameter is not a correctly formatted SetApplData structure. Either the SetAD_eye1 or the SetAD_ver field is incorrect or the storage pointed to by the SetAD_ptr field did not contain a correctly formatted SetADcontainer structure. The SetAD_eye2 field is incorrect.
-1	EINVAL	JrBuffLenInvalid	SetAD_len contains an incorrect length for the SetAD_ver value of the SetADcontainer structure.
-1	EFault	JrBadInputBufAddr	An abend occurred while attempting to copy the SetADcontainer structure from the address provided in SetAD_ptr field.
-1	ENOBUFFS	JrSmNoStorage	There was no storage available to store the associated data.

The SIOCSAPPLDATA call can be issued on stream sockets only. No application authorization is required. Each time the ioctl call is issued, the application data is replaced. If the call is issued on a socket prior to issuing a listen() call, the application data is inherited by all connections accepted over that socket. If the call is issued on a socket after issuing a listen() call, the application data is inherited by all connections accepted over that socket that arrive after the ioctl call is processed.

C language example

```
#include <ezbyaplc.h>
char myappldata[SETADBUFLen+1];      /* extra byte for null string terminator */
SetApplData myIoctlParm;
SetADcontainer myBuffer;

sprintf(myappldata, "@HRSERVER%8.8s%8.8s%8.8s%8.8s", a, b, c,d); /* prefix
and 4 char[8] fields */

memcpy(myIoctlParm.SetAD_eye1, SETADEYE1, sizeof(myIoctlParm.SetAD_eye1));
myIoctlParm.SetAD_ver = SETADVER;
myIoctlParm.SetAD_len = sizeof(SetADcontainer);
myIoctlParm.SetAD_ptr = &myBuffer

memcpy(myBuffer.SetAD_eye2, SETADEYE2, sizeof(myIoctlParm.SetAD_eye2));
memcpy(myBuffer.SetADbuffer, myappldata, SETADBUFLen);

rc = ioctl(soc, SIOCSAPPLDATA, (char *)&myIoctlParm);
```


TCP_KeepAlive socket option

Some TCP/IP users require a keep alive function with better timing granularity (in seconds) than that provided by the existing SO_KeepAlive socket option, which uses a stack-wide time value provided by configuration data.

The Posix.1g standard defines an alternative keep alive function, TCP_KeepAlive, which provides a value in seconds that is specific to a particular socket.

The value of TCP_KeepAlive, which is used for the current connection in place of the configuration default keep alive time (when keep alive timing is made active by the SO_KeepAlive socket option), can be in the range 1 – 2 147 460 seconds. If a value greater than 2 147 460 is specified, 2 147 460 is used. If a TCP_KeepAlive value of 0 is specified for a specific socket, keep alive timing for that socket is disabled.

SetSockOpt for TCP_KeepAlive

Specifies a socket-specific timer value that remains in effect until respecified by the SetSockOpt option or until the socket is closed. Timeout values in the range 1 – 2 147 460 seconds (or 0) are valid for TCP_KeepAlive. If a value larger than the allowed range is specified, a value of 2 147 460 seconds is used.

GetSockOpt for TCP_Keepalive

Returns the specific timer value (in seconds) that is in effect for the given socket, or a value of 0 if keep alive timing is not active.

Unlike the current algorithm used for issuing probes during an SO_KeepAlive cycle, the TCP_KeepAlive function time values the number of probes issued before terminating the connection. Probe retry intervals are scaled in proportion to the interval specified, as shown in Table 108.

Table 108. TCP_KeepAlive time values

TCP_KeepAlive time (T) specified in seconds	Seconds to first probe	Number of probes	Probe interval	Maximum interval
T = 0 (KeepAlive Disabled)	n/a	n/a	n/a	n/a
0 < T ≤ 5	T	1	1	T + 1
5 < T ≤ 10	T	1	2	T + 2
10 < T ≤ 30	T	1	5	T + 5
30 < T ≤ 60	T	1	10	T + 10
60 < T ≤ 120	T	1	20	T + 20
120 < T ≤ 300	T	2	20	T + 40
300 < T ≤ 600	T	2	30	T + 160
600 < T ≤ 1800	T	5	30	T + 150
1800 < T ≤ 3600	T	5	60	T + 300
3600 < T ≤ 7200	T	9	60	T + 540
7200 < T ≤ 2 147 460 (35 791 × 60 = 2 147 460)	T	9	75	T + 675
T > 2 147 460	2 147 460	9	75	2 147 460 + 675

The TCP_KeepAlive option value can range from 1 – 2 147 460 seconds. For values greater than 2 hours (7200 seconds), the probe interval and number of probes are adjusted as the specified interval increases until they coincide with the default algorithm. If no response is received from the remote partner after the listed number of probes, the connection is terminated.

Tips:

1. The SO_KeepAlive function must be activated before any keep alive processing is done. The KEEPALIVEOPTIONS configuration value is used for timing unless a specific value has been provided through the TCP_KeepAlive option.
2. The TCP_KeepAlive option can be set before or after the SO_KeepAlive function is activated, but timing does not take effect until the SO_KeepAlive status is set to active.

Appendix A. TCP/IP in the sysplex

This topic introduces the `SO_CLUSTERCONNTYPE` socket option. This socket option provides sysplex-specific connection routing information to a sockets application, which might enable the application to offer better function, performance, and scalability. Specific information about how to use `SO_CLUSTERCONNTYPE` can be found in the *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*.

Restriction: This enhancement does not apply to UDP or raw socket connections.

Sockets applications are typically designed to communicate with an appropriate partner on any platform, but they might be able to perform better if both partners know they are on zSeries, within the same sysplex, or on the same MVS image.

TCP sockets applications can benefit from information about the partner. Table 109 lists examples of these benefits.

Table 109. SO_CLUSTERCONNTYPE socket option benefits

Scenario	Potential benefit
Same Cluster (sysplex)	Avoid parameter conversions, because both sides of the connection use the same machine architectures and data representation (z/OS).
Same MVS image	Share memory information that is costly to generate (for example, security contexts).
Internal	Communications are not exposed outside the cluster. This means that application security might cost less. For example, the application might not encrypt application data.

The internal indication is returned only when the partner is part of the same cluster. This means that the data flows to the partner over a link or interface that is never exposed outside the cluster (sysplex), and the link or interface type is one of the following:

- CTC
- HiperSockets™ (iQDIO)
- MPCPTP (including XCF and IUTSAMEH)
- Loopback
- Both connection partners are owned by the same multi-homed stack

These devices/interfaces are assumed to participate in the same physical security as the cluster itself, so that the links/interfaces carrying IP traffic have the same physical security as links/interfaces to the attached DASD. When the internal indication is returned, the application can choose not to encrypt data exchanged with a partner application in the same cluster. This saves CPU cycles and improves throughput. The application itself determines whether or not to exploit the internal indication.

Tip: If the destination IP address for a connection (for example, the partner's IP address) is a dynamic VIPA or distributed dynamic VIPA residing in the cluster, the internal indication will not be on since traffic for these connections can be forwarded to the target TCP/IP stacks over links or interfaces that are external to the cluster.

For example, exploiting the internal indication might be used by an application to avoid the cost of encrypting data. If an application has just established a connection for which SSL would be the appropriate protection if the partner were not in the sysplex, and the application has assumed or has been configured to know that data within a sysplex is protected by physical security (controlled physical access), then the application might choose to implement the following:

- Immediately after connection setup, but before initiating SSL handshaking, issue the GETSOCKOPT call to obtain SO_CLUSTERCONNTYPE information. If the internal indication is not returned, proceed to initiate the SSL handshaking with appropriate levels of encryption specified (negotiated) between the two connection endpoints.
- If the internal indication is returned, initiate SSL handshaking as usual to gain the benefits of authenticating the partner, but specify only null encryption as an encryption choice. Because support for null encryption is a required feature of SSL, the SSL handshake is not destined to fail for architectural (IETF RFC) reasons. It is then up to the partner to determine whether a negotiated null encryption is acceptable to the partner or the connection should be closed.

While the expensive SSL handshaking cannot be avoided in any case, encryption of the data exchanged between the partners can be turned off as appropriate. If the applications were doing bulk data transfer, and normal encryption would be triple-DES, the savings in CPU cycles might be considerable.

Additional benefits include:

- Avoiding costly application operations (such as parameter marshalling) at the discretion of the application
- Sharing of information that provides the following:
 - Reduced CPU utilization
 - Reduced application workload
 - Better application performance

In general, sockets applications are designed so that any partners (client and server) using the same protocol can be used to connect with each other to do useful work. Typically, applications had to determine (for each partner) its platform and then exchange (through application protocol) this information with its partner. In some cases, this application level exchange cannot be performed:

- If both sides of the connection are not owned by the same company
- If the application protocol is governed by industry standards that do not include platform-related information

The SO_CLUSTERCONNTYPE socket option reports the same image (same MVS image or Virtual Server), same cluster (same sysplex), or cluster internal to a sockets application when a connection is established.¹ The information is determined and reported only when specifically requested, so that the application

1. When all of the TCP/IP stacks in the sysplex have been initialized and are in a steady state, they will have exchanged information within the sysplex, such that each stack recognizes all of the IP addresses supported by the other stacks in the sysplex, and which particular stacks support which IP addresses. The name of the MVS image for each stack is also made known

that does not need to use the function does not incur the expense. This option performs similarly whether the sockets application was the listening (server) application or the initiating (client) application.

If the TCP/IP stacks within a sysplex have been partitioned using a sysplex subplex, then results from the `SO_CLUSTERCONNTYPE` socket option request are impacted in the following ways depending on what is reported:

- Same cluster (sysplex) indicator
If two stacks on the same sysplex belong to different TCP/IP subplexes, this indicator is no longer on. Communication between these two stacks must go across a network interface, as opposed to using a dynamic XCF connection.
- Same MVS image indicator
If two stacks on the same MVS system belong to different TCP/IP subplexes, this indicator is no longer on.
- Internal indicator
If two TCP/IP stacks are in separate subplexes, connections between them no longer have this indicator on if CTC, MPCPTP (including XCF and IUTSAMEH), or HiperSockets is used. The reason for this is that each stack is not aware of the IP addresses for the partner stack because they belong to different subplexes.
Connections using Loopback or one of the local interfaces on a stack will continue to have this indicator on.

The reason the indicators can no longer be set is because when stacks are in different subplexes, they do not appear to each other to be in the same image or cluster.

to all other stacks. Thus, for any TCP connections, a stack can determine from the partner IP address whether or not the stack supporting the partner application is part of the same sysplex, and whether the stack resides in the same MVS image as the local stack.

Appendix B. Well-known port assignments

This topic lists the well-known port assignments for transport protocols TCP and UDP, and includes port number, keyword, and a description of the reserved port assignment. You can also find a list of these well-known port numbers in the *hlq.ETC.SERVICES* data set. The official assignment of port numbers is managed by the Internet Assigned Numbers Authority. The current list can be viewed at <http://www.iana.org/assignments/port-numbers>.

Table 110 lists the well-known port assignments for TCP.

Table 110. TCP well-known port assignments

Port number	Keyword	Assigned to	Services description
0		reserved	
5	rje	remote job entry	remote job entry
7	echo	echo	echo
9	discard	discard	sink null
11	systat	active users	active users
13	daytime	daytime	daytime
15	netstat	netstat	who is up or netstat
19	chargen	ttytst source	character generator
21	ftp	FTP	File Transfer Protocol
23	telnet	telnet	telnet
25	smtp	mail	Simple Mail Transfer Protocol
37	time	timeserver	timeserver
39	rlp	resource	Resource Location Protocol
42	nameserver	name	host name server
43	nicname	who is	who is
53	domain	name server	domain name server
57	mtp	private terminal access	private terminal access
69	tftp	TFTP	Trivial File Transfer protocol
77	rje	netrjs	any private RJE service
79	finger	finger	finger
80	http	http	Web Server
87	link	ttylink	any private terminal link
95	supdup	supdup	SUPDUP protocol
101	hostname	hostname	nic hostname server, usually from SRI-NIC
109	pop	postoffice	Post Office Protocol
111	sunrpc	sunrpc	Sun remote procedure call
113	auth	authentication	authentication service
115	sftp	sftp	Simple File Transfer Protocol
117	uucp-path	UUCP path service	UUCP path service
119	untp	readnews untp	USENET News Transfer Protocol

Table 110. TCP well-known port assignments (continued)

Port number	Keyword	Assigned to	Services description
123	ntp	NTP	Network Time Protocol
160–223		reserved	
712	vexec	vice-exec	Andrew File System authenticated service
713	vlogin	vice-login	Andrew File System authenticated service
714	vshell	vice-shell	Andrew File System authenticated service
2001	datasetsrv		Andrew File System service
2106	venus.itc		Andrew File System service, for the Venus process

Well-known UDP port assignments

Table 111 lists the well-known port assignments for UDP.

Table 111. Well-known UDP port assignments

Port number	Keyword	Assigned to	Services description
0		reserved	
5	rje	remote job entry	remote job entry
7	echo	echo	echo
9	discard	discard	sink null
11	users	active users	active users
13	daytime	daytime	daytime
15	netstat	Netstat	Netstat
19	chargen	ttytst source	character generator
37	time	timeserver	timeserver
39	rlp	resource	Resource Location Protocol
42	nameserver	name	host name server
43	nicname	who is	who is
53	domain	nameserver	domain name server
69	tftp	TFTP	Trivial File Transfer Protocol
75			any private dial out service
77	rje	netrjs	any private RJE service
79	finger	finger	finger
111	sunrpc	sunrpc	Sun remote procedure call
123	ntp	NTP	Network Time Protocol
135	llbd	NCS LLBD	NCS local location broker daemon
161	snmp	SNMP	SNMP server
162	snmptrap	SNMPTRAP	SNMP trap
531	rxd-control		rxd control port
2001	rauth2		Andrew File System service, for the Venus process
2002	rfilebulk		Andrew File System service, for the Venus process

Table 111. Well-known UDP port assignments (continued)

Port number	Keyword	Assigned to	Services description
2003	rfilesrv		Andrew File System service, for the Venus process
2018	console		Andrew File System service
2115	ropcons		Andrew File System service, for the Venus process
2131	rupdsrv		assigned in pairs; bulk must be srv +1
2132	rupdbulk		assigned in pairs; bulk must be srv +1
2133	rupdsrv1		assigned in pairs; bulk must be srv +1
2134	rupdbulk1		assigned in pairs; bulk must be srv +1
12000	entextxid		IBM Enterprise Extender SNA XID Exchange
12001	entextnetwk		IBM Enterprise Extender SNA COS Network Priority
12002	entexthigh		IBM Enterprise Extender SNA COS High Priority
12003	entextmed		IBM Enterprise Extender SNA COS Medium Priority
12004	entextlow		IBM Enterprise Extender SNA COS Low Priority

Appendix C. Programming interfaces for providing classification data to be used in differentiated services policies

Applications and users of TCP/IP networks may have different requirements for the service they receive from those networks. A network that treats all traffic as best effort may not meet the needs of such users. Service differentiation is a mechanism to provide different service levels to different traffic types based on their requirements and importance in an enterprise network. For example, it might be critical to provide Enterprise Resource Planning (ERP) traffic better service during peak hours than that of FTP or web traffic. The overall service provided to applications or users, in terms of elements such as throughput and delay, is termed Quality of Service (QoS).

One aspect of QoS is Differentiated Services (DS), which provides QoS to broad classes of traffic or users, for example all outbound web traffic accessed by a particular subnet. z/OS provides support for DS by allowing network administrators to define policies that describe how different z/OS TCP/IP workload traffic should be treated. Administrators can define service policy rules that identify desired workloads and map them to service policy actions that dictate the DS attributes assigned to these workloads. For more information on QoS and DS, see the *z/OS Communications Server: IP Configuration Guide*.

Service policy rules can specify generic attributes to identify a given workload, such as the server's well-known port or jobname. However, there are cases where a more granular level of classification for a server's outgoing TCP/IP traffic is desired. For example, a server application may provide services for several different types of requests using a single well-known port. A network administrator may want to be able to specify unique DS attributes for each service type the application supports. One way of accomplishing this is by allowing applications to provide additional information that can be used by an administrator to define more granular service policy rules and actions. The programming interfaces described in this section provide this capability.

Application defined policy classification data can be specified using extensions to the `sendmsg()` socket API. The `sendmsg()` API is similar to other socket APIs, such as `send()` and `write()` that allow an application to send data, but also provides the capability of specifying ancillary data. Ancillary data allows applications to pass additional option data to the TCP/IP protocol stack along with the normal data that is sent to the TCP/IP network. This ancillary data can be used by the application to define the attributes of the outgoing traffic for a particular TCP connection or for the specific data being sent in that `sendmsg()` invocation. These extensions to the `sendmsg()` API are only available to applications using the TCP protocol and the following socket API libraries:

- z/OS IBM C/C++ sockets with the z/OS Language Environment. For more information on these APIs see the *z/OS XL C/C++ Run-Time Library Reference*.
- z/OS UNIX System Services Assembler Callable services socket APIs. For more information on these APIs see the *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

The policy classification data is defined by the application and contains one (or both) of the following two formats:

- **Application defined token:** This token is a free format character string that can represent any application defined resource (for example, as transaction identifier, user ID, URL, and so on). When an application passes this token in `sendmsg()`, TCP/IP will invoke the policy classification function passing it the application-defined token in addition to any of the existing classification attributes (local/remote IP address and port, jobname, and so on). The application defined token maps to the `ApplicationData` attribute of a DS policy rule.
- **Application priority levels:** An application specified priority that maps to one of five predefined QoS service levels: Expedited, High, Medium, Low and Best Effort. Applications using this format of application classification data need to map their outgoing data types to one of these priority levels. For example, the application may already have a concept of transaction priority that it can use to map to one of these priority levels. It is important to note that the priority specified by the application does not automatically translate to a QoS service level. The actual service level assigned is derived by the contents of the service policy. Application priority rules are mapped to the `ApplicationPriority` attribute of a DS policy rule.

Applications may decide to pass classification data of either format or for both formats. The latter option allows applications to specify the same application defined token yet associate it with different priorities depending on the type of request being processed. For example, an application can pass an application token of `ORDER` and a `HIGH` priority for one user and a token of `ORDER` with a `LOW` priority for another user. The policy administrator would then be able to distinguish the service level assigned to these two different classes of users. When passing classification data on the `sendmsg()` API, applications also need to determine the scope of the classification:

- **Connection-Level:** The DS policy action assigned will be used for all traffic on this TCP connection until another `sendmsg()` with different classification data is specified.
- **Message-Level:** The DS policy action assigned will be used only for the outgoing data passed on this `sendmsg()` invocation. Any future data sent on this connection without the specification of any classification data will use the original DS policy action that was assigned to this TCP connection.

Passing application classification data on SENDMSG

A key difference in the `sendmsg()` API versus the more common `send()` API is that most parameters are passed in a message header input parameter. The mapping for the message header is defined in *socket.h* for C/C++ and in the *BPXYMSGH* macro for users of the UNIX System Services Assembler Callable services. For simplicity, only the C/C++ version of the data structures are shown in this section:

```
struct msghdr {
    void          *msg_name;           /* optional address      */
    size_t        msg_namelen;         /* size of address       */
    struct iovec  *msg_iov;             /* scatter/gather array  */
    int           msg_iovlen;          /* # elements in msg_iov */
    void          *msg_control;         /* ancillary data        */
    size_t        msg_controllen;      /* ancillary data length  */
    int           msg_flags;           /* flags on received msg */
};
```

The following are some key points regarding the usage of `sendmsg()` for the purpose of passing application defined classification data:

- Since application policy classification data is only supported for TCP sockets, the *msg_name* and *msg_namelen* parameters are not applicable.
- Data to be sent using *sendmsg()* needs to be described in the *msg_iov* structure.
- The address of the ancillary data is passed in the *msg_control* field.
- *msg_controllen* contains the length of the ancillary data passed.

Note: If multiple ancillary data sections are passed, this length should reflect the total length of ancillary data sections.

- *msg_flags* is not applicable for *sendmsg()*

The ancillary data (in this case the application classification data) is pointed to by the *msg_control* parameter. This *msg_control* pointer points to the following structure (C/C++ example shown below) that describes the ancillary data (also defined in *socket.h* and *BPXYMSGH* respectively):

```
struct cmsghdr {
    size_t  cmsg_len;      /* data byte count includes hdr */
    int     cmsg_level;    /* originating protocol */
    int     cmsg_type;     /* protocol-specific type */
    /* followed by u_char  cmsg_data[]; */
};
```

- The *cmsghdr* should be set to the length of the *cmsghdr* plus the length of all application classification data that follows immediately after the *cmsghdr*. This is represented by the commented out *cmsghdr* field.
- The *cmsghdr* must be set to the constant *IPPROTO_IP* for AF_INET sockets and *IPPROTO_IPV6* for AF_INET6 sockets. *IPPROTO_IP* and *IPPROTO_IPV6* are defined in *in.h* and *BPXYSOCK*.
- The *cmsghdr* must be set to the constant *IP_QOS_CLASSIFICATION_DATA* (defined in header file *ezaqosdc.h* for C/C++ users and in macro *EZAQOSDA* for assembler users). The header file and macro are both shipped in the *SEZANMAC* data set. This data set must be available in the concatenation when compiling or assembling a part that makes use of these definitions.

The data that follows the *cmsghdr* structure is described by the following structure:

```
struct ip_qos_classification_data {
    int      ip_qos_version;      /* Version of structure */
    int      ip_qos_classification_scope; /* Classification Scope */
    int      ip_qos_classification_type; /* Type of QoS classification */
    u_char   ip_qos_reserved[12]; /* Reserved for IBM use */
    int      ip_qos_appl_token_len; /* Length of application data */
    /* u_char ip_qos_appl_token[128]; /* Application Classification Token*/
};
```

The *ip_qos_classification_data* structure should be filled in as follows:

- *ip_qos_version*: This field indicates version of the structure. This must be filled in using the constant *IP_QOS_CURRENT_VERSION*.
- *ip_qos_classification_scope*: Specify a connection level scope (use constant *IP_QOS_CONNECTION_LEVEL*) or a message level scope (constant *IP_QOS_MESSAGE_LEVEL*).

Connection level scope indicates that the DS policy action assigned by the way of classification of this message will remain in effect for all subsequent messages sent until a *sendmsg()* with new classification data is issued. Message level scope indicates that the DS policy action assigned will only be used for the message data included in this *sendmsg()* invocation. Future data sent without classification data will inherit the previous connection level DS policy action

assignment (from last Connection Level classification by the way of sendmsg() or from the original TCP connection classification during connection establishment).

- *ip_qos_classification_type*: This specification indicates the type of classification data being passed. An application can choose to pass an application defined token, an application specified priority, or both a token and a priority. If the latter option is selected the two selected classification types should be logically ORed together. The following types can be specified:

- Application defined token classification. A single type should be specified. If more than one type is specified the results are unpredictable.
- *IP_SET_QOSLEVEL_W_APPL_TOKEN_ASCII*: This indicates that the classification data is a character string in ASCII format. When this option is specified the application token needs to be passed in the *ip_qos_appl_token* field.

Note: If the application needs to pass numerical values for the classification data it should first convert them to printable ASCII format. Also note that the string specified can be in mixed case and will be used in the exact format specified for comparison purposes.

- *IP_SET_QOSLEVEL_W_APPL_TOKEN_EBCDIC*: Same as above except that the string is in EBCDIC format.

Note: The *IP_SET_QOSLEVEL_W_APPL_TOKEN_ASCII* does perform slightly better than this option as the application data specified in the policy is saved in ASCII format inside of the TCP/IP stack, thereby eliminating the need to translate the application defined token on every sendmsg() request.

- Application defined priority classification. A single type should be specified. If multiple priority types are specified the results are unpredictable.
- *IP_SET_QOSLEVEL_EXPEDITED*: Indicates that Expedited priority is requested.
- *IP_SET_QOSLEVEL_HIGH*: Indicates that High priority is requested.
- *IP_SET_QOSLEVEL_MEDIUM*: Indicates that Medium priority is requested.
- *IP_SET_QOSLEVEL_LOW*: Indicates that Low priority is requested.
- *IP_SET_QOSLEVEL_BEST_EFFORT*: Indicates that Best Effort priority is requested.
- *ip_qos_appl_token_len*: The length of the *ip_qos_appl_token* specified. This length should not include any null terminating characters.
- *ip_qos_appl_token*: This virtual field immediately follows the *ip_qos_classification_len* field and contains the application classification token string in either ASCII or EBCDIC format depending on which flavor of *IP_SET_QOSLEVEL_W_APPL_TOKEN_xxxx* was specified for the classification type. This field is only referenced when an application defined token type is specified. Note that this string should not exceed 128 bytes. If a larger size is specified, only the first 128 bytes will be used.

Additional considerations

The `sendmsg()` enhancements to allow for QoS classification data will only be available through the Language Environment C/C++ `sendmsg()` API and the UNIX System Services BPX2SMS service. The `sendmsg()` API supported across the TCP/IP provided socket API libraries (C, Macro, Callable, CICS, and so on) do not currently support the passing of ancillary data. Some additional considerations for these `sendmsg()` enhancements follow:

- UNIX System Services Assembler Callable Services Environment
 - Applications should ensure that the BPX2SMS (`sendmsg()`) service is invoked. An older version of `sendmsg()`, named BPX1SMS, also exists but does not support the application classification enhancements described in this section.
 - Include the *EZAQOSDA* macro from the SEZANMAC library for the definitions needed for the application classification ancillary data.
 - Include the *BPXYSOCK* and *BPXYMSGH* macros from SYS1.MACLIB.
- IBM C/C++ applications using the z/OS Language Environment:
 - Applications need to include the following header files:
 - `socket.h`, `in.h`
 - `ezaqosdc.h` (from SEZANMAC)
- AF_INET6 considerations

The `sendmsg()` enhancements for QoS classification data are supported for AF_INET6 sockets. However, they are supported only for AF_INET6 sockets when the connection's traffic flows over an IPv4 network (such as, the remote partner's IP address is an IPv4-mapped IPv6 address). This feature is not supported for AF_INET6 sockets when the connection's traffic flows over an IPv6 network (such as, the remote partner's IP address is an IPv6 address); the `sendmsg()` enhancements will be ignored if used on an IPv6 connection.

In order to exploit these enhancements for an AF_INET6 socket, the application should be coded as indicated in this topic, but should substitute `IPPROTO_IPV6` for `IPPROTO_IP` in the `cmsghdr`'s `cmsg_level` field.

Note: The Language Environment C/C++ library supports 2 versions of the `sendmsg()` API. The key difference is in the definition of the *msghdr* structure. In order to use the correct version of `sendmsg()` the application needs to ensure that the macro symbolic `_OE_SOCKETS` is not specified. `_OE_SOCKETS` causes the older version of *msghdr* and `sendmsg()` to be used. The older version does not support passing of application classification data.

Applications providing classification data should document the content and format of this data so that network administrators can use this information when defining DS policies.

Appendix D. X Window System interface V11R4 and Motif version 1.1

Support for X Window System Version 11 Release 4 and Motif Version 1.1 is available as feature HIP614X and is documented here.

The current support, provided as part of the base IP support in z/OS Communications Server, is documented in Chapter 7, “X Window System interface in the z/OS Communications Server environment,” on page 195.

What is provided

The X Window System support provided with TCP/IP includes the following APIs from the X Window System Version 11 Release 4:

- SEZAX11L (Xlib, Xmu, Xext, and Xau routines)
- SEZAOLDX (X Release 10 compatibility routines)
- SEZAXTLB (Xt Intrinsics)
- SEZAXAWL (Athena widget set)
- Header files needed for compiling X clients
- Standard MIT X clients
- Sample X clients (XSAMP1, XSAMP2, and XSAMP3)
- SEZARNT1 (a combination of the X Window System libraries listed previously and SEZACMTX)

Note: SEZARNT1 contains the reentrant versions of the libraries.

- SEZARNT2 (Athena widget set for reentrant modules)
- SEZARNT3 (Motif widget set for reentrant modules). The SEZARNT1, SEZARNT2, and SEZARNT3 library members are:
 - Fixed block 80, in object deck format.
 - Compiled with the C/370 RENT compile-time option.
 - Used as input for X Window System and socket programmers who make their programs reentrant.
 - Passed to the C/370 prelinker. Use the prelink utility to combine all input text decks into a single text deck.

The X Window System support provided with TCP/IP also includes the following APIs based on Release 1.1 of the Motif-based widget set:

- SEZAXMLB Motif-based widget set)
- Header files needed for compiling clients using the Motif-based widget set.

Three-dimensional graphics are available as an extension of the X Window System. For information about using three-dimensional graphics, see *PEXlib Specification and C Language Binding*, SR28-5166.

In addition, the X Window System support provided with TCP/IP includes support for z/OS UNIX System Services. For information about the z/OS UNIX System Services support provided, see “z/OS UNIX System Services support” on page 713.

Software requirements

Application programs using the X Window System API are written in C and should be compiled, linked, and executed using the z/OS Language Environment z/OS C/C++ compiler and run-time environment.

To run sample X clients (XSAMP1, XSAMP2, and XSAMP3), you require IBM C for System/370, Library Licensed Program (5688-188).

How the X Window System interface works in the MVS environment

The X Window System is a network transparent protocol that supports windowing and graphics. The protocol is communicated between a client or application and an X server over a reliable bidirectional byte stream. This byte stream is provided by the TCP/IP communication protocol. In the MVS environment, X Window System support consists of a set of application calls that create the X protocol, as requested by the application. This application programming interface allows an application to be created, which uses the X Window System protocol to be displayed on an X server.

In an X Window System environment, the X server distributes user input to and accepts requests from various client programs located either on the same system or elsewhere on a network. The X client code uses sockets to communicate with the X server.

Figure 35 on page 663 shows a high-level abstraction of how the X Window System works in a MVS environment. As an application writer, you need to be concerned only with the client API in writing your application.

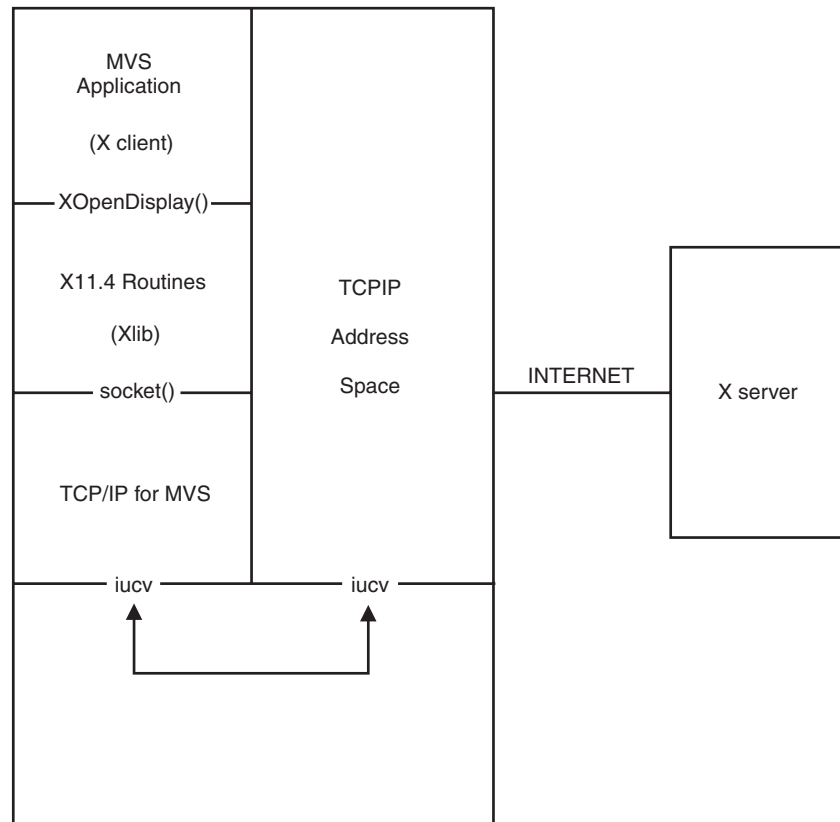


Figure 35. MVS X Window System application to server

The communication path from the MVS X Window System application to the server involves the client code and TCP/IP. The application program that you create is the client part of a client-server relationship. The X server provides access to the resources that are shared among many X applications, such as the screen, keyboard, mouse, fonts, and graphics contexts. A single X server can control more than one physical screen.

Each client can interact with multiple servers, and each server can interact with multiple clients.

If your application is written to the Xlib interface, it calls `XOpenDisplay()` to start communication with an X server on a workstation. The Xlib code opens a communication path called a socket to the X server, and sends the appropriate X protocol to initiate client-server communication.

The X protocol generated by the Window System client code uses an ISO Latin-1 encoding for character strings, while the MVS encoding for character strings is EBCDIC. The X Window System client code in the MVS environment automatically transforms character strings from EBCDIC to ISO Latin-1 or from ISO Latin-1 to EBCDIC, as needed using internal translate tables.

In the MVS environment, external names must be eight characters or less. Many of the X Window System application programming interface names exceed this limit. To support the X API in MVS, all X names longer than eight characters are remapped to unique names using the C compiler preprocessor. This name remapping is found in a file called `X11GLUE.H`, which is automatically included in

your program when you include the standard X header file called `XLIB.H`. When debugging your application, you can see the `X11GLUE.H` file to find the remapped names of the X API routines.

Identifying the target display

The `user_id.XWINDOWS.DISPLAY` data set is used by the X Window System to identify the host name of the target display.

The following is the format of the environment variable in the `user_id.XWINDOWS.DISPLAY` data set:

▶▶—*host_name*:*target_server*—┐
 └. *target_screen*—▶▶

The environment variable in the `user_id.XWINDOWS.DISPLAY` data set contains the following values:

Value Description

host_name

Specifies the host name or IP address of the host machine on which the X Window System server is running.

target_server

Specifies the number of the display server on the host machine.

target_screen

Specifies the screen to be used on the same target server.

Notes:

1. You should be aware that the `userid.XWINDOWS.DISPLAY` data set cannot contain sequence numbers.
2. For information about identifying the target display in z/OS UNIX System Services see, “Identifying the target display in z/OS UNIX System Services” on page 714.

Application resource file

The X Window System allows you to modify certain characteristics of an application at run time by means of application resources. Typically, application resources are set to tailor the appearance and possibly the behavior of an application. The application resources can specify information about an application’s window sizes, placement, coloring, font usage, and other functional details.

On a UNIX system, this information can be found in the user’s home directory in a file called `.Xdefaults`. In the MVS environment, this data set is called `user_id.X.DEFAULTS`. Each line of this data set represents resource information for an application.

Note: For information about the application resource file in z/OS UNIX System Services, see “z/OS UNIX System Services support” on page 713.

Figure 36 on page 665 shows an example of a set of resources specified for a typical X Window System application.

```
XClock*geometry:      500x60+5-5
XClock*font:          -bitstream-*bold-r--33-240-*
XClock*foreground:    orange
XClock*background:    skyblue
XClock*borderWidth:   4
XClock*borderColor:   blue
XClock*analog:        false
```

Figure 36. Resources specified for a typical X Window System application

In this example, the xclock application automatically creates a window in the lower left corner of the screen with a digital display in orange letters on a skyblue background.

These resources can also be set on the RESOURCE_MANAGER property of the X server, which allows a single, central place where resources are found, that control all applications, displayed on an X server. You can use the xrbdb program to control the X server resource database in the resource property.

xrbdb is an X client that you can use either to get or to set the contents of the RESOURCE_MANAGER property on the root window of screen 0. This property is then used by all applications at startup to control the application resource.

Creating an application

To create an application that uses the X Window System protocol, you should study the X Window System application programming interface. In addition, sample programs called XSAMP1, XSAMP2, and XSAMP3 (see “Using sample X Window System programs” on page 671) illustrate simple examples of programs that use the X Window System API. These programs are distributed with TCP/IP.

You should ensure that the first X header file your program includes is the XLIB.H header file. This file defines a number of preprocessor symbols, which enable your program to compile correctly. If your program uses the Xt Intrinsics, you should ensure that the INTRINSIC.H header file is the first X header file included in your program. This file contains a number of preprocessor symbols that allow your program to compile correctly. In addition, these header files include the MVS header files that remap the external names of the X Window System routines to the shorter names used by the X Window System that is supported by TCP/IP.

X Window System header files

This section describes the X Window System, X Intrinsics, Athena widget set, and Motif-based widget set headers used by X Window System applications.

X Window System and Xt Intrinsic header files

The following is a list of X Window System and Xt Intrinsic headers:

ap@keysy.h	IntriniI.h	StringDe.h
Atoms.h	IntriniP.h	SysUtil.h
Callback.h	Intrinsi.h	Translat.h
CharSet.h	keysym.h	VarargsI.h
CloseHoo.h	keysymde.h	Vendor.h
ComposiI.h	ks@names.h	VendorP.h
ComposiP.h	Misc.h	WinUtil.h
Composit.h	MITMisc.h	X.h
Constrai.h	mitmiscs.h	Xatom.h
ConstraP.h	multibst.h	Xatomtyp.h
Converte.h	multibuf.h	Xauth.h
ConvertI.h	Object.h	Xct.h
copyrigh.h	ObjectP.h	Xext.h
Core.h	PassivGr.h	Xkeymap.h
CoreP.h	poly.h	Xlib.h
cursorfo.h	Quarks.h	Xlibint.h
CurUtil.h	RectObj.h	Xlibos.h
CvtCache.h	RectObjP.h	Xllglue.h
DEKeysy.h	region.h	Xmd.h
DisplayQ.h	Resource.h	Xmu.h
Drawing.h	Selectio.h	Xos.h
Error.h	shape.h	Xproto.h
EventI.h	shapestr.h	Xprotost.h
extutil.h	Shell.h	Xresourc.h
fd.h	ShellP.h	Xt@remap.h
InitialI.h	StdCmap.h	Xtos.h
Initer.h	StdSel.h	Xutil.h
		XWDFile.h
		X10.h

Athena widget set header files

The following is a list of the Athena widget set headers:

ACommand.h	BoxP.h	SimpleMe.h
ACommanP.h	Cardinal.h	SimpleP.h
AForm.h	Clock.h	Sme.h
AFormP.h	ClockP.h	SmeBSB.h
ALabel.h	CommandI.h	SmeBSBP.h
ALabelP.h	Dialog.h	SmeLine.h
AList.h	DialogP.h	SmeLineP.h
AListP.h	Grip.h	SmeP.h
AScrollb.h	GripP.h	StripChP.h
AScrollP.h	Logo.h	StripCha.h
AText.h	LogoP.h	Template.h
ATextP.h	Mailbox.h	TemplatP.h
ATextSrP.h	MailboxP.h	TextSink.h
AsciiSin.h	MenuButP.h	TextSinP.h
AscSinkP.h	MenuButt.h	TextSrc.h
AsciiSrc.h	Paned.h	Toggle.h
AscSrcP.h	PanedP.h	ToggleP.h
AsciiTex.h	Scroll.h	VPaned.h
AscTextP.h	Simple.h	Viewport.h
Box.h	SimpleMP.h	ViewporP.h
		XawInit.h

Motif header files

The following is a list of headers for the Motif-based widget set:

ArrowB.h	FormP.h	SashP.h
ArrowBG.h	Frame.h	Scale.h
ArrowBGP.h	FrameP.h	ScaleP.h
ArrowBP.h	Label.h	ScrollBa.h
bitmaps.h	LabelG.h	ScrollBP.h
BulletBP.h	LabelGP.h	Scrolled.h
Bulletin.h	LabelP.h	ScrollWP.h
CascaBGP.h	List.h	SelectBP.h
CascadBG.h	ListP.h	SelectiB.h
CascaBGP.h	MainW.h	SeparatG.h
CascadBP.h	MainWP.h	Separato.h
CascadeB.h	MenuShel.h	SeparatP.h
Command.h	MenuShep.h	StringSr.h
CommandP.h	MessageBP.h	Text.h
CutPaste.h	MessageB.h	TextInP.h
CutPastP.h	PanedW.h	TextOutP.h
DialogS.h	PanedWP.h	TextP.h
DialogSP.h	PushB.h	TextSrcP.h
DrawingA.h	PushBG.h	TogglBGP.h
DrawinAP.h	PushBGP.h	TogglEB.h
DrawnB.h	PushBP.h	TogglEBG.h
DrawnBP.h	RowColum.h	TogglEBP.h
FileSB.h	RowColuP.h	Xm.h
FileSBP.h		XmP.h
Form.h		

Compiling and linking

You can use several methods to compile, link-edit, and execute your program in MVS. This section contains information about the data sets that you must include to run your C source program under MVS batch using cataloged procedures supplied by IBM.

The following list contains partitioned data set names, which are used as examples in the JCL statements below:

Data Set Name	Contents
<i>user_id</i> .MYPROG.C	Contains user C source programs.
<i>user_id</i> .MYPROG.C(PROGRAM1)	Member PROGRAM1 in <i>user_id</i> .MYPROG.C partitioned data set.
<i>user_id</i> .MYPROG.H	Contains user #include files.
<i>user_id</i> .MYPROG.OBJ	Contains object code for the compiled versions of user C programs in <i>user_id</i> .MYPROG.C.
<i>user_id</i> .MYPROG.LOAD	Contains link-edited versions of user programs in <i>user_id</i> .MYPROG.OBJ.

Nonreentrant modules

The following lines describe the additions that you must make to the compile step of your cataloged procedure to compile a nonreentrant module. Catalogued procedures are included in the samples supplied by IBM for your MVS system.

Note: Compile all C source using the `def(IBMCP)` preprocessor symbol.

- Add the following statement as the first `//SYSLIB DD` statement:

```
//SYSLIB DD DSN=SEZACMAC,DISP=SHR
```

- Add the following `//USERLIB DD` statement:

```
//USERLIB DD DSN=user_id.MYPROG.H,DISP=SHR
```

The following lines describe the additions that you must make to the link-edit step of your cataloged procedure to link-edit a nonreentrant module:

- To link-edit programs that use only X11 library functions, add the following statements as the first `//SYSLIB DD` statements:

```
// DD DSN=SEZAX11L,DISP=SHR
// DD DSN=SEZACMTX,DISP=SHR
```

- You must include the following statements when you link-edit your application code, because not all entry points are defined as external references in SEZAX11L:

```
INCLUDE SYSLIB(XMACROS)
INCLUDE SYSLIB(XLIBINT)
INCLUDE SYSLIB(XRM)
```

- To link-edit programs that use the Athena Toolkit functions, including Athena widget sets, add the following after the `//SYSLIB DD` statement:

```
// DD DSN=SEZAXAWL,DISP=SHR
// DD DSN=SEZAXTLB,DISP=SHR
// DD DSN=SEZAX11L,DISP=SHR
// DD DSN=SEZACMTX,DISP=SHR
```

- You must include the following when you link-edit your application code, because not all entry points are defined as external references in SEZAX11L, SEZAXTLB, and SEZAXAWL:

```
INCLUDE SYSLIB(XMACROS)
INCLUDE SYSLIB(XLIBINT)
INCLUDE SYSLIB(XRM)
INCLUDE SYSLIB(CALLBACK)
INCLUDE SYSLIB(CONVERT)
INCLUDE SYSLIB(CONVERTE)
INCLUDE SYSLIB(INTRINSI)
INCLUDE SYSLIB(DISPLAY)
INCLUDE SYSLIB(ERROR)
INCLUDE SYSLIB(EVENT)
INCLUDE SYSLIB(NEXTEVEN)
INCLUDE SYSLIB(TMSTATE)
INCLUDE SYSLIB(ASCTEXT)
INCLUDE SYSLIB(ATOMS)
INCLUDE SYSLIB(ATEXT)
```

- To link-edit programs that use the Motif Toolkit functions, add the following after the `//SYSLIB DD` statement:

```
// DD DSN=SEZAXMLB,DISP=SHR
// DD DSN=SEZAXTLB,DISP=SHR
// DD DSN=SEZAX11L,DISP=SHR
// DD DSN=SEZACMTX,DISP=SHR
```

- You must include the following when you link-edit your application code, because not all entry points are defined as external references in SEZAX11L, SEZAXTLB, and SEZAXMLB.

```
INCLUDE SYSLIB(XMACROS)
INCLUDE SYSLIB(XLIBINT)
INCLUDE SYSLIB(XRM)
INCLUDE SYSLIB(CALLBACK)
INCLUDE SYSLIB(CONVERT)
INCLUDE SYSLIB(CONVERTE)
```



```

INCLUDE SYSLIB(INTRINSI)
INCLUDE SYSLIB(DISPLAY)
INCLUDE SYSLIB(ERROR)
INCLUDE SYSLIB(EVENT)
INCLUDE SYSLIB(NEXTEVEN)
INCLUDE SYSLIB(TMSTATE)
INCLUDE SYSLIB(ATOMS)
INCLUDE SYSLIB(CUTPASTE)
INCLUDE SYSLIB(FILESB)
INCLUDE SYSLIB(GEOUTILS)
INCLUDE SYSLIB(LIST)
INCLUDE SYSLIB(MANAGER)
INCLUDE SYSLIB(PRIMITIV)
INCLUDE SYSLIB(RESIND)
INCLUDE SYSLIB(ROWCOLUM)
INCLUDE SYSLIB(MSELECTI)
INCLUDE SYSLIB(TEXT)
INCLUDE SYSLIB(TEXTF)
INCLUDE SYSLIB(TRAVERSA)
INCLUDE SYSLIB(VISUAL)
INCLUDE SYSLIB(XMSTRING)

```

Note: If you are using X Release 10 compatibility routines, add the following in the //SYSLIB DD statement:

```
//      DD DSN=SEZAOLDX,DISP=SHR
```

The following steps describe how to execute your program:

1. Specify the IP address of the X server on which you want to display the application output by creating or modifying the *user_id*.XWINDOWS.DISPLAY data set. The following is an example of a line in this data set.

```
CHARM.RALEIGH.IBM.COM:0.0 or      9.67.43.79:0.0
```
2. Allow the host application access to the X server.
3. On the workstation where you want to display the application output, you must grant permission for the MVS host to access the X server. To do this, enter the xhost command:

```
xhost ralmvs1
```
4. To execute your program under TSO, enter the following:

```
CALL 'user_id.MYPROG.LOAD(PROGRAM1)'
```

Reentrant modules

The following lines describe the additions that you must make to the compile step of your cataloged procedure to compile a reentrant module. Cataloged procedures are included in the samples supplied by IBM for your MVS system.

Note: Compile all C source using the def(IBM CPP) preprocessor symbol. See “Compiling and linking” on page 667 for information about compiling and linking your program in MVS.

- Add the following statement as the first //SYSLIB DD statement:

```
//SYSLIB DD DSN=SEZACMAC,DISP=SHR
```

- Add the following //USERLIB DD statement:

```
//USERLIB DD DSN=user_id.MYPROG.H,DISP=SHR
```

The following lines describe the additions that you must make to the prelink-edit and link-edit steps of your cataloged procedure to create a reentrant module.

- To create reentrant modules that use only the X11 library functions, do the following:

- Add the following statement as the first //SYSLIB DD statement in the prelink-edit step:


```
//          DD DSN=SEZARNT1,DISP=SHR
```
- Add the following statement as the first //SYSLIB DD statement in the link-edit step:


```
//          DD DSN=SEZACMTX,DISP=SHR
```
- To create reentrant modules that use only the Athena Toolkit functions, including Athena widget sets, do the following:
 - Add the following statements as the first //SYSLIB DD statements in the prelink-edit step:


```
//          DD DSN=SEZARNT2,DISP=SHR
//          DD DSN=SEZARNT1,DISP=SHR
```
 - Add the following statement as the first //SYSLIB DD statement in the link-edit step:


```
//          DD DSN=SEZACMTX,DISP=SHR
```
- To create reentrant modules that use only the Motif Toolkit functions, do the following:
 - Add the following statements as the first //SYSLIB DD statements in the prelink-edit step:


```
//          DD DSN=SEZARNT3,DISP=SHR
//          DD DSN=SEZARNT1,DISP=SHR
```
 - Add the following statement as the first //SYSLIB DD statement in the link-edit step:


```
//          DD DSN=SEZACMTX,DISP=SHR
```

Following is a sample cataloged procedure for an X11 library function.

```
/*-----
/* PRELINK-EDIT STEP:
/*-----
//PRELNK EXEC PGM=EDCPRLK,REGION=4096K,COND=(4,LT),
//          PARM='MAP,NONCAL'
//STEPLIB DD DSN=C370.LL.V2R1M0.SEDCLINK,DISP=SHR
//          DD DSN=C370.LL.V2R1M0.COMMON.SIBMLINK,DISP=SHR
//          DD DSN=C370.LL.V2R1M0.SEDCCOMP,DISP=SHR
//SYSLIB DD DSN=B37.SEZARNT1,DISP=SHR
//OBJLIB DD DSN=&OBJLIB;,DISP=SHR;
//SYSMOD DD UNIT=VIO,SPACE=(TRK,(50,10)),DISP=(MOD,PASS),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120)
//SYSMSGSGS DD DSN=C370.V2R1M0.SEDCMMSGSGS(EDCMMSGGE),DISP=SHR
//SYSPRINT DD SYSOUT=&SYSOUT;
//SYSOUT DD SYSOUT=&SYSOUT;
/*
/*-----
/* LINK-EDIT STEP:
/*-----
//LKED EXEC PGM=IEWL,PARM='&LPARM;',COND=(4,LT)
//SYSLIB DD DSN=&VSCCHD;&CVER;&CBASE;,DISP=SHR;
//          DD DSN=C370.LL.V2R1M0.COMMON.SIBMLINK,DISP=SHR
//          DD DSN=&COMHD;&COMVER;&COMBASE;,DISP=SHR;
//          DD DSN=C370.V2R1M0.SEDCSPC,DISP=SHR
//          DD DSN=B37.SEZACMTX,DISP=SHR
//NEWOBJ DD DSN=*.PRELNK.SYSMOD,DISP=(OLD,DELETE)
//OBJLIB DD DSN=&OBJLIB;,DISP=SHR;
//SYSMOD DD DSN=&XWDLOAD;,DISP=SHR;
//SYSPRINT DD SYSOUT=&SYSOUT;
//SYSUT1 DD DSN=&&SYSUT1;,UNIT=&WORKDA;,DISP=&LKDISP;,SPACE=&WRKSPC;
/*
```

Note: For more information about installing a reentrant module in the LPA area, see the *z/OS XL C/C++ User's Guide*.

The following steps describe how to execute your program:

1. Specify the IP address of the X server on which you want to display the application output by creating or modifying the *user_id*.XWINDOWS.DISPLAY data set. The following is an example of a line in this data set:
CHARM.RALEIGH.IBM.COM:0.0 or 9.67.43.79:0.0
2. Allow the host application access to the X server.
On the workstation where you want to display the application output, you must grant permission for the MVS host to access the X server. To do this, enter the xhost command:
xhost ralmvs1
3. If you have installed your program in the LPA as a reentrant module and you want to run it under TSO, enter the following:

PROGRAM1

Note: For more information about compiling and linking, see the *z/OS XL C/C++ User's Guide*.

Using sample X Window System programs

This section contains information about the sample X programs provided. The C source code can be found in the SEZAINST data set.

The following are sample C source programs:

Module	Description
XSAMP1	Xlib sample program
XSAMP2	Athena widget sample program
XSAMP3	Motif-based widget sample program

Running a sample program

For information about running a sample program, see “Compiling and linking” on page 667 and “Compiling and linking with z/OS UNIX System Services” on page 715.

Environment variables

Table 112 provides a list of environment variables that can be explicitly set by X Window System Interface V11r4.

Table 112. Environment variables for X Window System Interface V11r4

Environment variable	Description
HOME	The system initializes this variable at login to the path name of the user's home directory.
LANG	Determines the locale category for the native language, local customs, and coded character set in the absence of the LC_ALL and other LC_* (LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC, LC_TIME) environment variables. See Note.

Table 112. Environment variables for X Window System Interface V11r4 (continued)

Environment variable	Description
LOGNAME	The system initializes this variable at login to the user's login name.
RESOURCE_NAME	Used by XtOpenDisplay as an alternative specification of an application name. There is no default value.
SESSION_MANAGER	If defined, this environment variable causes a Session Shell widget to connect to a session manager. There is no default value.
USER	The name of the user account; this is determined by the name that was entered at login.
XAPPLRESDIR	Specifies the directory to search for files that contain application defaults.
XAUTHORITY	Specifies the name of the authority file on the local host.
XBLANGPATH	Used to locate desktop icons, if XMICONBMSEARCHPATH or XMICONSEARCHPATH are not set.
XENVIRONMENT	Contains the full path name of the file that contains resource defaults. There is no default value.
XFILESEARCHPATH	Specifies where the X resources file for the current locale is located.
XLOCALEDIR	Specifies the directory to search for locale files. The default value is /usr/lib/X11/locale.
XMODIFIERS	Used by the XSetLocaleModifiers function to specify additional modifiers. There is no default value.
XPROPFORMATS	Specifies the name of a file from which to obtain additional formats.
XUSERFILEPATH	Specifies the search paths for files containing application defaults. There is no default value.
XWTRACE	Controls the generation of traces of the socket level communications between Xlib and the X Window System server. These traces are as follows: • XWTRACE undefined or 0: No trace generated • XWTRACE=1: Error messages • XWTRACE>=2: API function tracing for TRANS functions There is no default value. The output is sent to stderr.
Note: This can be used by applications to determine the language to use for error messages, instructions, collating sequences, date formats, and so on.	

Standard X client applications

The following standard MIT X clients are also provided with TCP/IP as examples of how to use the X Window System API:

Application	Description
appres	Lists application resource database

atobm	Bit map conversion utilities
bitmap	Bit map editor
bmtoa	Bit map conversion utilities
listres	Lists resources in widgets
oclock	Displays time of day
xauth	X authority data set utility
xcalc	Scientific calculator for X
xclock	Analog/digital clock for X
xdpyinfo	Displays information utility for X
xfd	Font displayer for X
xfontsel	Point and click interface for selecting X11 font names
xkill	Stops a client by its X resource
xlogo	X Window System logo
xlsatoms	Lists interned atoms defined on server
xlsclients	Lists client applications running on a display
xlsfonts	Displays server font list displayer for X
xlswins	Displays server window list displayer for X
xmag	Magnify parts of the screen
xprop	Property displayer for X
xrdb	X server resource database utility
xrefresh	Refreshes all or part of an X screen
xset	User preference utility for X
xsetroot	Root window parameter setting utility for X
xwd	Dumps an image of an X window
xwininfo	Window information utility for X
xwud	Displays image displayer for X

These standard X Window client application programs also contain information about X Window System programming techniques.

Consult the following members of the SEZAINST data set for documentation about the MIT X clients:

Member Name	Description
HLPAPPRE	Help for APPRES module
HLPBITMA	Help for BITMAP module
HLPLISTR	Help for LISTRES module
HLPOCLOC	Help for OCLOCK module
HLPXAUTH	Help for XAUTH module
HLPXCALC	Help for XCALC module

HLPXCLOC	Help for XCLOCK module
HLPXDPYI	Help for XDPYINFO module
HLPXFD	Help for XFD module
HLPXFONT	Help for XFONTSEL module
HLPXKILL	Help for XKILL module
HLPXLOGO	Help for XLOGO module
HLPXLSAT	Help for XLSATOMS module
HLPXLSCL	Help for XLSCLIEN module
HLPXLSFO	Help for XLSFONTS module
HLPXLSWI	Help for XLSWINS module
HLPXMAG	Help for XMAG module
HLPXPROP	Help for XPROP module
HLPXRDB	Help for XRDB module
HLPXREFR	Help for XREFRESH module
HLPXSET	Help for XSET module
HLPXSETR	Help for XSETROOT module
HLPXWD	Help for XWD module
HLPXWINI	Help for XWININFO module
HLPXWUD	Help for XWUD module

The SEZAINST data set also contains default application resource data sets for XCALC, XCLOCK, XFD, and XFONTSEL. Copy these data sets from:

- SEZAINST(XXCALC)
- SEZAINST(XXCLOCK)
- SEZAINST(XXFD)
- SEZAINST(XXFONTSE)

to the following data sets for TSO users:

- *user_id*.XAPDF.XCALC
- *user_id*.XAPDF.XCLOCK
- *user_id*.XAPDF.XFD
- *user_id*.XAPDF.XFONTSEL

Notes:

1. The EZAGETIN job includes JCL to copy the sample members from SEZAINST to *user_id*.XAPDF.*classname*, where *classname* is the application specified class name. The high-level qualifier should be tailored to be the user ID using these data sets.
2. For information on default application resource data sets for z/OS UNIX System Services users, see “z/OS UNIX System Services support” on page 713.

Building X client modules

The support for X Window System Version 11 Release 4 provides standard MIT X clients. The C source and header files are found in SEZAINST and SEZACMAC data sets respectively.

You can build the following X client modules based on X11 functions:

Table 113. Building X client modules based on X11 functions.

To build module	Do the following
ATOBM	1. Compile the ATOBM C source program. 2. Link-edit the ATOBM object module.
BITMAP	1. Compile the BITMAP C source program. 2. Compile the BMDIALOG C source program. 3. Link-edit the BITMAP and BMDIALOG object modules.
BMTOA	1. Compile the BMTOA C source program. 2. Link-edit the BMTOA object module.
XAUTH	1. Compile the XAUTH C source program. 2. Compile the GTHOSTXA C source program. 3. Compile the PROCESS source program. 4. Compile the PARSEDPY C source program. 5. Link-edit the XAUTH, GTHOSTXA, PROCESS, and PARSEDPY object modules.
XDYPYINFO C	1. Compile the XDYPYINFO C source program. 2. Link-edit the XDYPYINFO object module.
XKILL	1. Compile the XKILL C source program. 2. Link-edit the XKILL object module.
XLSATOMS	1. Compile the XLSATOMS C source program. 2. Link-edit the XLSATOMS object module.
XLSCLIEN	1. Compile the XLSCLIEN C source program. 2. Link-edit the XLSCLIEN object module.
XLSFONTS	1. Compile the XLSFONTS C source program. 2. Compile the DSIMPLE C source program. 3. Link-edit the XLSFONTS and DSIMPLE object modules.
XLSWINS	1. Compile the XLSWINS C source program. 2. Link-edit the XLSWINS object module.
XMAG	1. Compile the XMAG C source program. 2. Link-edit the XMAG object module.
XPROP	1. Compile the XPROP C source program. 2. Compile the DSIMPLE C source program. 3. Link-edit the XPROP and DSIMPLE object modules.
XRDB	1. Compile the XRDB C source program. 2. Link-edit the XRDB object module.
XREFRESH	1. Compile the XREFRESH C source program. 2. Link-edit the XREFRESH object module.
XSET	1. Compile the XSET C source program. 2. Link-edit the XSET object module.
XSETROOT	1. Compile the XSETROOT C source program. 2. Link-edit the XSETROOT object module.

Table 113. Building X client modules based on X11 functions. (continued)

To build module	Do the following
XWD	1. Compile the XWD C source program. 2. Compile the DSIMPLE C source program. 3. Link-edit the XWD and DSIMPLE object modules.
XWININFO	1. Compile the XWININFO C source program. 2. Compile the DSIMPLE C source program. 3. Link-edit the XWININFO and DSIMPLE object modules.
XWUD	1. Compile the XWUD C source program. 2. Link-edit the XWUD object module.

You can build the following X client modules based on Xt Intrinsics and Athena Toolkit functions:

Table 114. Building X client modules based on Xt Intrinsics and Athena Toolkit functions.

To build module	Do the following
APPRES	1. Compile the APPRES C source program. 2. Link-edit the APPRES object module.
OCLOCK	1. Compile the OCLOCK C source program. 2. Compile the NCLOCK C source program. 3. Compile the TRANSFOR C source program. 4. Link-edit the OCLOCK, NCLOCK, and TRANSFOR object modules.
LISTRES	1. Compile the LISTRES C source program. 2. Compile the UTIL C source program. 3. Compile the WIDGETS C source program. 4. Link-edit the LISTRES, UTIL, and WIDGETS object modules.
XCALC	1. Compile the XCALC C source program. 2. Compile the ACTIONS C source program. 3. Compile the MATH C source program. 4. Link-edit the XCALC, ACTIONS, and MATH object modules.
XCLOCK	1. Compile the XCLOCK C source program. 2. Link-edit the XCLOCK object module.
XFD	1. Compile the XFD C source program. 2. Compile the FONTGRID C source program. 3. Link-edit the XFD and FONTGRID object modules.
XFONTSEL	1. Compile the XFONTSEL C source program. 2. Link-edit the XFONTSEL object module.
XLOGO	1. Compile the XLOGO C source program. 2. Link-edit the XLOGO object module.

X Window System routines

The following tables list the routines supported by TCP/IP. The routines are grouped according to the type of function provided.

Opening and closing a display

Table 115 provides the routines for opening and closing a display.

Table 115. Opening and closing display

Routine	Description
XCloseDisplay()	Closes a display.
XFree()	Frees in-memory data created by Xlib function.
XNoOp()	Executes a NoOperation protocol request.
XOpenDisplay()	Opens a display.

Creating and destroying windows

Table 116 provides the routines for creating and destroying windows.

Table 116. Creating and destroying windows

Routine	Description
XConfigureWindow()	Configures the specified window.
XCreateSimpleWindow()	Creates unmapped InputOutput subwindow.
XCreateWindow()	Creates unmapped subwindow.
XDestroySubwindows()	Destroys all subwindows of specified window.
XDestroyWindow()	Unmaps and destroys window and all subwindows.

Manipulating windows

Table 117 provides the routines for manipulating windows.

Table 117. Manipulating windows

Routine	Description
XCirculateSubwindows()	Circulates a subwindow up or down.
XCirculateSubwindowsUp()	Raises the lowest mapped child of window.
XCirculateSubwindowsDown()	Lowers the highest mapped child of window.
XIconifyWindow()	Sends a WM_CHANGE_STATE ClientMessage to the root window of the specified screen.
XLowerWindow()	Lowers the specified window.
XMapRaised()	Maps and raises the specified window.
XMapSubwindows()	Maps all subwindows of the specified window.
XMapWindow()	Maps the specified window.
XMoveResizeWindow()	Changes the specified window size and location.
XMoveWindow()	Moves the specified window.
XRaiseWindow()	Raises the specified window.
XReconfigureWMWindow()	Issues a ConfigureWindow request on the specified top-level window.

Table 117. Manipulating windows (continued)

Routine	Description
XResizeWindow()	Changes the specified window's size.
XRestackWindows()	Restacks a set of windows from top to bottom.
XSetWindowBorderWidth()	Changes the border width of the window.
XUnmapSubwindows()	Unmaps all subwindows of the specified window.
XUnmapWindow()	Unmaps the specified window.
XWithdrawWindow()	Unmaps the specified window and sends a synthetic UnmapNotify event to the root window of the specified screen.

Changing window attributes

Table 118 provides the routines for changing window attributes.

Table 118. Changing window attributes

Routine	Description
XChangeWindowAttributes()	Changes one or more window attributes.
XSetWindowBackground()	Sets the window background to a specified pixel.
XSetWindowBackgroundPixmap()	Sets the window background to a specified pixmap.
XSetWindowBorder()	Changes the window border to a specified pixel.
XSetWindowBorderPixmap()	Changes the window border tile.
XTranslateCoordinates()	Transforms coordinates between windows.

Obtaining window information

Table 119 provides the routines for obtaining window information.

Table 119. Obtaining window information

Routine	Description
XGetGeometry()	Gets the current geometry of the specified drawable.
XGetWindowAttributes()	Gets the current attributes for the specified window.
XQueryPointer()	Gets the pointer coordinates and the root window.
XQueryTree()	Obtains the IDs of the children and parent windows.

Obtaining properties and atoms

Table 120 provides the routines for obtaining properties and atoms.

Table 120. Properties and atoms

Routine	Description
XGetAtomName()	Gets a name for the specified atom ID.
XInternAtom()	Gets an atom for the specified name.

Manipulating window properties

Table 121 on page 679 provides the routines for manipulating the properties of windows.

Table 121. Manipulating window properties

Routine	Description
XChangeProperty()	Changes the property for the specified window.
XDeleteProperty()	Deletes a property for the specified window.
XGetWindowProperty()	Gets the atom type and property format for the window.
XListProperties()	Gets the specified window property list.
XRotateWindowProperties()	Rotates the properties in a property array.

Setting window selections

Table 122 provides the routines for setting window selections.

Table 122. Setting window selections

Routine	Description
XConvertSelection()	Converts a selection.
XGetSelectionOwner()	Gets the selection owner.
XSetSelectionOwner()	Sets the selection owner.

Manipulating colormaps

Table 123 provides the routines for manipulating color maps.

Table 123. Manipulating colormaps

Routine	Description
XAllocStandardColormap()	Allocates an XStandardColormap structure.
XCopyColormapAndFree()	Creates a new colormap from a specified colormap.
XCreateColormap()	Creates a colormap.
XFreeColormap()	Frees the specified colormap.
XQueryColor()	Queries the RGB value for a specified pixel.
XQueryColors()	Queries the RGB values for an array of pixels.
XSetWindowColormap()	Sets the colormap of the specified window.

Manipulating color cells

Table 124 provides the routines for manipulating color cells.

Table 124. Manipulating color cells

Routine	Description
XAllocColor()	Allocates a read-only color cell.
XAllocColorCells()	Allocates read/write color cells.
XAllocColorPlanes()	Allocates read/write color resources.
XAllocNamedColor()	Allocates a read-only color cell by name.
XFreeColors()	Frees colormap cells.
XLookupColor()	Looks up a colorname.
XStoreColor()	Stores an RGB value into a single colormap cell.
XStoreColors()	Stores RGB values into colormap cells.

Table 124. Manipulating color cells (continued)

Routine	Description
XStoreNamedColor()	Sets a pixel color to the named color.

Creating and freeing pixmaps

Table 125 provides the routines for creating and freeing pixmaps.

Table 125. Creating and freeing pixmaps

Routine	Description
XCreatePixmap()	Creates a pixmap of a specified size.
XFreePixmap()	Frees all storage associated with specified pixmap.

Manipulating graphics contexts

Table 126 provides the routines for manipulating graphics contexts.

Table 126. Manipulating graphics contexts

Routine	Description
XChangeGC()	Changes the components in the specified Graphics Context (GC).
XCopyGC()	Copies the components from a source GC to a destination GC.
XCreateGC()	Creates a new GC.
XFreeGC()	Frees the specified GC.
XGetGCValues()	Returns the GC values in the specified structure.
XGContextFromGC()	Obtains the GContext resource ID for GC.
XQueryBestTile()	Gets the best fill tile shape.
XQueryBestSize()	Gets the best size tile, stipple, or cursor.
XQueryBestStipple()	Gets the best stipple shape.
XSetArcMode()	Sets the arc mode of the specified GC.
XSetBackground()	Sets the background of the specified GC.
XSetClipmask()	Sets the clip_mask of the specified GC to a specified pixmap.
XSetClipOrigin()	Sets the clip origin of the specified GC.
XSetClipRectangles()	Sets the clip_mask of GC to a list of rectangles.
XSetDashes()	Sets the dashed line style components of a specified GC.
XSetFillRule()	Sets the fill rule of the specified GC.
XSetFillStyle()	Sets the fill style of the specified GC.
XSetFont()	Sets the current font of the specified GC.
XSetForeground()	Sets the foreground of the specified GC.
XSetFunction()	Sets display function in the specified GC.
XSetGraphicsExposures()	Sets the graphics exposure flag of the specified GC.
XSetLineAttributes()	Sets the line drawing components of the GC.
XSetPlaneMask()	Sets the plane mask of the specified GC.

Table 126. Manipulating graphics contexts (continued)

Routine	Description
XSetState()	Sets the foreground, background, plane mask, and function in GC.
XSetStipple()	Sets the stipple of the specified GC.
XSetSubwindowMode()	Sets the subwindow mode of the specified GC.
XSetTile()	Sets the fill tile of the specified GC.
XSetTSTOrigin()	Sets the tile or stipple origin of the specified GC.

Clearing and copying areas

Table 127 provides the routines for clearing and copying areas.

Table 127. Clearing and copying areas

Routine	Description
XClearArea()	Clears a rectangular area of the window.
XClearWindow()	Clears the entire window.
XCopyArea()	Copies the drawable area between drawables of the same root and the same depth.
XCopyPlane()	Copies single bit plane of the drawable.

Drawing lines

Table 128 provides the routines for drawing lines.

Table 128. Drawing lines

Routine	Description
XDraw()	Draws an arbitrary polygon or curve that is defined by the specified list of Vertexes as specified in <i>vlist</i> .
XDrawArc()	Draws a single arc in the drawable.
XDrawArcs()	Draws multiple arcs in a specified drawable.
XDrawFilled()	Draws arbitrary polygons or curves and then fills them.
XDrawLine()	Draws a single line between two points in a drawable.
XDrawLines()	Draws multiple lines in the specified drawable.
XDrawPoint()	Draws a single point in the specified drawable.
XDrawPoints()	Draws multiple points in the specified drawable.
XDrawRectangle()	Draws an outline of a single rectangle in the drawable.
XDrawRectangles()	Draws an outline of multiple rectangles in the drawable.
XDrawSegments()	Draws multiple line segments in the specified drawable.

Filling areas

Table 129 provides the routines for filling areas.

Table 129. Filling areas

Routine	Description
XFillArc()	Fills single arc in drawable.

Table 129. Filling areas (continued)

Routine	Description
XFillArcs()	Fills multiple arcs in drawable.
XFillPolygon()	Fills a polygon area in the drawable.
XFillRectangle()	Fills single rectangular area in the drawable.
XFillRectangles()	Fills multiple rectangular areas in the drawable.

Loading and freeing fonts

Table 130 provides the routines for loading and freeing fonts.

Table 130. Loading and freeing fonts

Routine	Description
XFreeFont()	Unloads the font and frees the storage used by the font.
XFreeFontInfo()	Frees the font information array.
XFreeFontNames()	Frees a font name array.
XFreeFontPath()	Frees data returned by XGetFontPath.
XGetFontPath()	Gets the current font search path.
XGetFontProperty()	Gets the specified font property.
XListFontsWithInfo()	Gets names and information about loaded fonts.
XLoadFont()	Loads a font.
XLoadQueryFont()	Loads and queries font in one operation.
XListFonts()	Gets a list of available font names.
XQueryFont()	Gets information about a loaded font.
XSetFontPath()	Sets the font search path.
XUnloadFont()	Unloads the specified font.

Querying character string sizes

Table 131 provides the routines for querying the character size of a string.

Table 131. Querying character string sizes

Routine	Description
XFreeStringList()	Frees the in-memory data associated with the specified string list.
XQueryTextExtents()	Gets a 1-byte character string bounding box from the server.
XQueryTextExtents16()	Gets a 2-byte character string bounding box from the server.
XStringListToTextProperty()	Converts lists of pointers to character strings and text properties.
XTextExtents()	Gets a bounding box of a 1-byte character string.
XTextExtents16()	Gets a bounding box of a 2-byte character string.
XTextPropertyToStringList()	Returns a list of strings representing the elements of the specified XTextProperty structure.
XTextWidth()	Gets the width of an 8-bit character string.

Table 131. Querying character string sizes (continued)

Routine	Description
XTextWidth16()	Gets the width of a 2-byte character string.

Drawing text

Table 132 provides the routines for drawing text.

Table 132. Drawing text

Routine	Description
XDrawImageString()	Draws 8-bit image text in the specified drawable.
XDrawImageString16()	Draws 2-byte image text in the specified drawable.
XDrawString()	Draws 8-bit text in the specified drawable.
XDrawString16()	Draws 2-byte text in the specified drawable.
XDrawText()	Draws 8-bit complex text in the specified drawable.
XDrawText16()	Draws 2-byte complex text in the specified drawable.

Transferring images

Table 133 provides the routines for transferring images.

Table 133. Transferring images

Routine	Description
XGetImage()	Gets the image from the rectangle in the drawable.
XGetSubImage()	Copies the rectangle on the display to the image.
XPutImage()	Puts the image from memory into the rectangle in the drawable.

Manipulating cursors

Table 134 provides the routines for manipulating cursors.

Table 134. Manipulating cursors

Routine	Description
XCreateFontCursor()	Creates a cursor from a standard font.
XCreateGlyphCursor()	Creates a cursor from font glyphs.
XDefineCursor()	Defines a cursor for a window.
XFreeCursor()	Frees a cursor.
XQueryBestCursor()	Gets useful cursor sizes.
XRecolorCursor()	Changes the color of a cursor.
XUndefineCursor()	Undefines a cursor for a window.

Handling window manager functions

Table 135 on page 684 provides the routines for handling the window manager functions.

Table 135. Handling window manager functions

Routine	Description
XAddToSaveSet()	Adds a window to the client saveset.
XAllowEvents()	Allows events to be processed after a device is frozen.
XChangeActivePointerGrab()	Changes the active pointer grab.
XChangePointerControl()	Changes the interactive feel of the pointer device.
XChangeSaveSet()	Adds or removes a window from the client's saveset.
XGetInputFocus()	Gets the current input focus.
XGetPointerControl()	Gets the current pointer parameters.
XGrabButton()	Grabs a mouse button.
XGrabKey()	Grabs a single key of the keyboard.
XGrabKeyboard()	Grabs the keyboard.
XGrabPointer()	Grabs the pointer.
XGrabServer()	Grabs the server.
XInstallColormap()	Installs a colormap.
XKillClient()	Removes a client.
XListInstalledColormaps()	Gets a list of currently installed colormaps.
XRemoveFromSaveSet()	Removes a window from the client's saveset.
XReparentWindow()	Changes the parent of a window.
XSetCloseDownMode()	Changes the close down mode.
XSetInputFocus()	Sets the input focus.
XUngrabButton()	Ungrabs a mouse button.
XUngrabKey()	Ungrabs a key.
XUngrabKeyboard()	Ungrabs the keyboard.
XUngrabPointer()	Ungrabs the pointer.
XUngrabServer()	Ungrabs the server.
XUninstallColormap()	Uninstalls a colormap.
XWarpPointer()	Moves the pointer to an arbitrary point on the screen.

Manipulating keyboard settings

Table 136 provides the routines for manipulating keyboard settings.

Table 136. Manipulating keyboard settings

Routine	Description
XAutoRepeatOff()	Turns off the keyboard auto-repeat.
XAutoRepeatOn()	Turns on the keyboard auto-repeat.
XBell()	Sets the volume of the bell.
XChangeKeyboardControl()	Changes the keyboard settings.
XChangeKeyboardMapping()	Changes the mapping of symbols to keycodes.
XDeleteModifiermapEntry()	Deletes an entry from the XModifierKeymap structure.
XFreeModifiermap()	Frees XModifierKeymap structure.
XGetKeyboardControl()	Gets the current keyboard settings.

Table 136. Manipulating keyboard settings (continued)

Routine	Description
XGetKeyboardMapping()	Gets the mapping of symbols to keycodes.
XGetModiferMapping()	Gets keycodes to be modifiers.
XGetPointerMapping()	Gets the mapping of buttons on the pointer.
XInsertModifiermapEntry()	Adds an entry to the XModifierKeymap structure.
XNewModifiermap()	Creates the XModifierKeymap structure.
XQueryKeymap()	Gets the state of the keyboard keys.
XSetPointerMapping()	Sets the mapping of buttons on the pointer.
XSetModifierMapping()	Sets keycodes to be modifiers.

Controlling the screen saver

Table 137 provides the routines for controlling the screen saver.

Table 137. Controlling the screen saver

Routine	Description
XActivateScreenSaver()	Activates the screen saver.
XForceScreenSaver()	Turns the screen saver on or off.
XGetScreenSaver()	Gets the current screen saver settings.
XResetScreenSaver()	Resets the screen saver.
XSetScreenSaver()	Sets the screen saver.

Manipulating hosts and access control

Table 138 provides the routines for manipulating hosts and toggling the access control.

Table 138. Manipulating hosts and access control

Routine	Description
XDisableAccessControl()	Disables access control.
XEnableAccessControl()	Enables access control.
XListHosts()	Gets the list of hosts.
XSetAccessControl()	Changes access control.

Handling events

Table 139 provides the routines for handling events.

Table 139. Handling events

Routine	Description
XCheckIfEvent()	Checks event queue for the specified event without blocking.
XCheckMaskEvent()	Removes the next event that matches a specified mask without blocking.
XCheckTypedEvent()	Gets the next event that matches event type.
XCheckTypedWindowEvent()	Gets the next event for the specified window.

Table 139. Handling events (continued)

Routine	Description
XCheckWindowEvent()	Removes the next event that matches the specified window and mask without blocking.
XEventsQueued()	Checks the number of events in the event queue.
XFlush()	Flushes the output buffer.
XGetMotionEvents()	Gets the motion history for the specified window.
XIfEvent()	Checks the event queue for the specified event and removes it.
XMaskEvent()	Removes the next event that matches a specified mask.
XNextEvent()	Gets the next event and removes it from the queue.
XPeekEvent()	Peeks at the event queue.
XPeekIfEvent()	Checks the event queue for the specified event.
XPending()	Returns the number of events that are pending.
XPutBackEvent()	Pushes the event back to the top of the event queue.
XSelectInput()	Selects events to be reported to the client.
XSendEvent()	Sends an event to a specified window.
XSync()	Flushes the output buffer and waits until all requests are completed.
XWindowEvent()	Removes the next event that matches the specified window and mask.

Enabling and disabling synchronization

Table 140 provides the routines for toggling synchronization.

Table 140. Enabling and disabling synchronization

Routine	Description
XSetAfterFunction()	Sets the previous after function.
XSynchronize()	Enables or disables synchronization.

Using default error handling

Table 141 provides the routines for using the default error handling.

Table 141. Using default error handling

Routine	Description
XDisplayName()	Gets the name of the display currently being used.
XGetErrorText()	Gets the error text for the specified error code.
XGetErrorDatabaseText()	Gets the error text from the error database.
XSetErrorHandler()	Sets the error handler.
XSetIOErrorHandler()	Sets the error handler for unrecoverable I/O errors.

Communicating with window managers

Table 142 on page 687 provides the routines for communicating with window managers.

Table 142. Communicating with window managers

Routine	Description
XAllocClassHints()	Allocates storage for an XClassHint structure.
XAllocIconSize()	Allocates storage for an XIconSize structure.
XAllocSizeHints()	Allocates storage for an XSizeHints structure.
XAllocWMHints()	Allocates storage for an XWMHints structure.
XGetClassHint()	Gets the class of a window.
XFetchName()	Gets the name of a window.
XGetCommand()	Gets a window WM_COMMAND property.
XGetIconName()	Gets the name of an icon window.
XGetIconSizes()	Gets the values of icon size atom.
XGetNormalHints()	Gets size hints for window in normal state.
XGetRGBColormaps()	Gets colormap associated with specified atom.
XGetSizeHints()	Gets the values of type WM_SIZE_HINTS properties.
XGetStandardColormap()	Gets colormap associated with specified atom.
XGetTextProperty()	Gets window property of type TEXT.
XGetTransientForHint()	Gets WM_TRANSIENT_FOR property for window.
XGetWM_CLIENT_MACHINE	Gets the value of a window WM_CLIENT_MACHINE property.
XGetWMColormapWindows()	Gets the value of a window WM_COLORMAP_WINDOWS property.
XGetWMHints()	Gets the value of the window manager hints atom.
XGetWMName()	Gets the value of the WM_NAME property.
XGetWMIconName()	Gets the value of the WM_ICON_NAME property.
XGetWMNormalHints()	Gets the value of the window manager hints atom.
XGetWMProtocols()	Gets the value of a window WM_PROTOCOLS property.
XGetWMSizeHints()	Gets the values of type WM_SIZE_HINTS properties.
XGetZoomHints()	Gets values of the zoom hints atom.
XSetCommand()	Sets the value of the command atom.
XSetClassHint()	Sets the class of a window.
XSetIconName()	Assigns a name to an icon window.
XSetIconSizes()	Sets the values of icon size atom.
XSetNormalHints()	Sets size hints for a window in normal state.
XSetRGBColormaps()	Sets the colormap associated with the specified atom.
XSetSizeHints()	Sets the values of the type WM_SIZE_HINTS properties.
XSetStandardColormap()	Sets the colormap associated with the specified atom.
XSetStandardProperties()	Specifies a minimum set of properties.
XSetTextProperty()	Sets window properties of type TEXT.
XSetTransientForHint()	Sets WM_TRANSIENT_FOR property for window.
XSetWMClientMachine()	Sets window WM_CLIENT_MACHINE property.
XSetWMColormapWindows()	Sets a window WM_COLORMAP_WINDOWS property.
XSetWMHints()	Sets the value of the window manager hints atom.

Table 142. Communicating with window managers (continued)

Routine	Description
XSetWMIconName()	Sets the value of the WM_ICON_NAME property.
XSetWMName()	Sets the value of the WM_NAME property.
XSetWMNormalHints()	Sets the value of the window manager hints atom.
XSetWMProperties()	Sets the values of properties for a window manager.
XSetWMProtocols()	Sets the value of the WM_PROTOCOLS property.
XSetWMSizeHints()	Sets the values of type WM_SIZE_HINTS properties.
XSetZoomHints()	Sets the values of the zoom hints atom.
XStoreName()	Assigns a name to a window.

Manipulating keyboard event functions

Table 143 provides the routines for manipulating keyboard event functions.

Table 143. Manipulating keyboard event functions

Routine	Description
XKeycodeToKeysym()	Converts keycode to a keysym value.
XKeysymToKeycode()	Converts keysym value to keycode.
XKeysymToString()	Converts keysym value to keysym name.
XLookupKeysym()	Translates a keyboard event into a keysym value.
XLookupMapping()	Gets the mapping of a keyboard event from a keymap file.
XLookupString()	Translates the keyboard event into a character string.
XRebindCode()	Changes the keyboard mapping in the keymap file.
XRebindKeysym()	Maps the character string to a specified keysym and modifiers.
XRefreshKeyboardMapping()	Refreshes the stored modifier and keymap information.
XStringToKeysym()	Converts the keysym name to the keysym value.
XUseKeymap()	Changes the keymap files.
XGeometry()	Parses window geometry given padding and font values.
XGetDefault()	Gets the default window options.
XParseColor()	Obtains RGB values from color name.
XParseGeometry()	Parses standard window geometry options.
XWMGeometry()	Obtains a window's geometry information.

Manipulating regions

Table 144 provides the routines for manipulating regions.

Table 144. Manipulating regions

Routine	Description
XClipBox()	Generates the smallest enclosing rectangle in the region.
XCreateRegion()	Creates a new empty region.
XEmptyRegion()	Determines whether a specified region is empty.

Table 144. Manipulating regions (continued)

Routine	Description
XEqualRegion()	Determines whether two regions are the same.
XIntersectRegion()	Computes the intersection of two regions.
XDestroyRegion()	Frees storage associated with the specified region.
XOffsetRegion()	Moves the specified region by the specified amount.
XPointInRegion()	Determines if a point lies in the specified region.
XPolygonRegion()	Generates a region from points.
XRectInRegion()	Determines if a rectangle lies in the specified region.
XSetRegion()	Sets the GC to the specified region.
XShrinkRegion()	Reduces the specified region by a specified amount.
XSubtractRegion()	Subtracts two regions.
XUnionRegion()	Computes the union of two regions.
XUnionRectWithRegion()	Creates a union of source region and rectangle.
XXorRegion()	Gets the difference between the union and intersection of regions.

Using cut and paste buffers

Table 145 provides the routines for using cut and paste buffers.

Table 145. Using cut and paste buffers

Routine	Description
XFetchBuffer()	Gets data from a specified cut buffer.
XFetchBytes()	Gets data from the first cut buffer.
XRotateBuffers()	Rotates the cut buffers.
XStoreBuffer()	Stores data in a specified cut buffer.
XStoreBytes()	Stores data in first cut buffer.

Querying visual types

Table 146 provides the routines for querying visual types.

Table 146. Querying visual types

Routine	Description
XGetVisualInfo()	Gets a list of visual information structures.
XListDepths()	Determines the number of depths that are available on a given screen.
XListPixmapFormats()	Gets the pixmap format information for a given display.
XMatchVisualInfo()	Gets visual information matching screen depth and class.
XPixmapFormatValues()	Gets the pixmap format information for a given display.

Manipulating images

Table 147 on page 690 provides the routines for manipulating images.

Table 147. Manipulating images

Routine	Description
XAddPixel()	Increases each pixel in pixmap by a constant value.
XCreateImage()	Allocates memory for the XImage structure.
XDestroyImage()	Frees memory for the XImage structure.
XGetPixel()	Gets a pixel value in an image.
XPutPixel()	Sets a pixel value in an image.
XSubImage()	Creates an image that is a subsection of a specified image.

Manipulating bit maps

Table 148 provides the routines for manipulating bit maps.

Table 148. Manipulating bit maps

Routine	Description
XCreateBitmapFromData()	Includes a bit map in the C program.
XCreatePixmapFromBitmapData()	Creates a pixmap using bit map data.
XDeleteContext()	Deletes data associated with the window and context type.
XFindContext()	Gets data associated with the window and context type.
XReadBitmapFile()	Reads in a bit map from a file.
XSaveContext()	Stores data associated with the window and context type.
XUniqueContext()	Allocates a new context.
XWriteBitmapFile()	Writes out a bit map to a file.

Using the resource manager

Table 149 provides the routines for using the resource manager.

Table 149. Using the resource manager

Routine	Description
Xpermalloc()	Allocates memory that is never freed.
XrmDestroyDatabase()	Destroys a resource database and frees its allocated memory.
XrmGetFileDatabase()	Creates a database from a specified file.
XrmGetResource()	Retrieves a resource from a database.
XrmGetStringDatabase()	Creates a database from a specified string.
XrmInitialize()	Initializes the resource manager.
XrmMergeDatabases()	Merges two databases.
XrmParseCommand()	Stores command options in a database.
XrmPutFileDatabase()	Copies the database into a specified file.
XrmPutLineResource()	Stores a single resource entry in a database.
XrmPutResource()	Stores a resource in a database.
XrmPutStringResource()	Stores string resource in a database.
XrmQGetResource()	Retrieves a quark from a database.

Table 149. Using the resource manager (continued)

Routine	Description
XrmQGetSearchList()	Gets a resource search list of database levels.
XrmQGetSearchResource()	Gets a quark search list of database levels.
XrmQPutResource()	Stores binding and quarks in a database.
XrmQPutStringResource()	Stores string binding and quarks in a database.
XrmQuarkToString()	Converts a quark to a character string.
XrmStringToQuark()	Converts a character string to a quark.
XrmStringToQuarkList()	Converts character strings to a quark list.
XrmStringToBindingQuarkList()	Converts strings to bindings and quarks.
XrmUniqueQuark()	Allocates a new quark.

Manipulating display functions

Table 150 provides the routines for manipulating display functions.

Table 150. Manipulating display functions

Routine	Description
AllPlanes() XAllPlanes()	Returns all bits suitable for use in plane argument.
BitMapBitOrder() XBitMapOrder()	Returns either the most or least significant bit in each bit map unit.
BitMapPad() XBitMapPad()	Returns the multiple of bits padding each scanline.
BitMapUnit() XBitMapUnit()	Returns the size of a bit map unit in bits.
BlackPixel() XBlackPixel()	Returns the black pixel value of the screen specified.
BlackPixelOfScreen() XBlackPixelOfScreen()	Returns the black pixel value of the screen specified.
CellsOfScreen() XCellsOfScreen()	Returns the number of colormap cells.
ConnectionNumber() XConnectionNumber()	Returns the file descriptor of the connection.
CreatePixmapCursor() XCreatePixmapCursor()	Creates a pixmap of a specified size.
CreateWindow() XCreateWindow()	Creates an unmapped subwindow for a specified parent window.
DefaultColormap() XDefaultColormap()	Returns a default colormap ID for allocation on the screen specified.
DefaultColormapOfScreen() XDefaultColormapOfScreen()	Returns the default colormap ID of the screen specified.
DefaultDepth() XDefaultDepth()	Returns the depth of the default root window.
DefaultDepthOfScreen() XDefaultDepthOfScreen()	Returns the default depth of the screen specified.
DefaultGC() XDefaultGC()	Returns the default GC of the default root window.
DefaultGCOfScreen() XDefaultGCOfScreen()	Returns the default GC of the screen specified.
DefaultScreen() XDefaultScreen()	Obtains the default screen referred to in the XOpenDisplay routine.
DefaultScreenofDisplay() XDefaultScreenofDisplay()	Returns the default screen of the display specified.
DefaultRootWindow() XDefaultRootWindow()	Obtains the root window for the default screen specified.
DefaultVisual() XDefaultVisual()	Returns the default visual type of the screen specified.
DefaultVisualOfScreen() XDefaultVisualOfScreen()	Returns the default visual type of the screen specified.
DisplayCells() XDisplayCells()	Displays the number of entries in the default colormap.
DisplayHeight() XDisplayHeight()	Displays the height of the screen in pixels.

Table 150. Manipulating display functions (continued)

Routine	Description
DisplayHeightMM() XDisplayHeightMM()	Displays the height of the screen in millimeters.
DisplayOfScreen() XDisplayOfScreen()	Displays the type of screen specified.
DisplayPlanes() XDisplayPlanes()	Displays the depth (number of planes) of the root window of the screen specified.
DisplayString() XDisplayString()	Displays the string passed to XOpenDisplay when the current display was opened.
DisplayWidth() XDisplayWidth()	Displays the width of the specified screen in pixels.
DisplayWidthMM() XDisplayWidthMM()	Displays the width of the specified screen in millimeters.
DoesBackingStore() XDoesBackingStore()	Indicates whether the specified screen supports backing stores.
DoesSaveUnders() XDoesSaveUnders()	Indicates whether the specified screen supports save unders.
EventMaskOfScreen() XEventMaskOfScreen()	Returns the initial root event mask for a specified screen.
HeightMMOfScreen() XHeightMMOfScreen()	Returns the height of a specified screen in millimeters.
HeightOfScreen() XHeightOfScreen()	Returns the height of a specified screen in pixels.
ImageByteOrder() XImageByteOrder()	Specifies the required byte order for each scanline unit of an image.
IsCursorKey()	Returns TRUE if keysym is on cursor key.
IsFunctionKey()	Returns TRUE if keysym is on function keys.
IsKeypadKey()	Returns TRUE if keysym is on keypad.
IsMiscFunctionKey()	Returns TRUE if keysym is on miscellaneous function keys.
IsModifierKey()	Returns TRUE if keysym is on modifier keys.
IsPFKey()	Returns TRUE if keysym is on PF keys.
LastKnownRequestProcessed() XLastKnownRequestProcessed()	Extracts the full serial number of the last known request processed by the X server.
MaxCmapsOfScreen() XMaxCmapsOfScreen()	Returns the maximum number of colormaps supported by the specified screen.
MinCmapsOfScreen() XMinCmapsOfScreen()	Returns the minimum number of colormaps supported by the specified screen.
NextRequest() XNextRequest()	Extracts the full serial number to be used for the next request to be processed by the X Server.
PlanesOfScreen() XPlanesOfScreen()	Returns the depth (number of planes) in a specified screen.
ProtocolRevision() XProtocolRevision()	Returns the minor protocol revision number (0) of the X server associated with the display.
ProtocolVersion() XProtocolVersion()	Returns the major version number (11) of the protocol associated with the display.
QLength() XQLength()	Returns the length of the event queue for the display.
RootWindow() XRootWindow()	Returns the root window of the current screen.
RootWindowOfScreen() XRootWindowOfScreen()	Returns the root window of the specified screen.
ScreenCount() XScreenCount()	Returns the number of screens available.
XScreenNumberOfScreen()	Returns the screen index number of the specified screen.
ScreenOfDisplay() XScreenOfDisplay()	Returns the pointer to the screen of the display specified.

Table 150. Manipulating display functions (continued)

Routine	Description
ServerVendor() XServerVendor()	Returns the pointer to a null-determined string that identifies the owner of the X server implementation.
VendorRelease() XVendorRelease()	Returns the number related to the vendor's release of the X server.
WhitePixel() XWhitePixel()	Returns the white pixel value for the current screen.
WhitePixelOfScreen() XWhitePixelOfScreen()	Returns the white pixel value of the specified screen.
WidthMMOfScreen() XWidthMMOfScreen()	Returns the width of the specified screen in millimeters.
WidthOfScreen() XWidthOfScreen()	Returns the width of the specified screen in pixels.

Extension routines

X Window System Extension Routines allow you to create extensions to the core Xlib functions with the same performance characteristics. The following are the protocol requests for X Window System extensions:

- XQueryExtension
- XListExtensions
- XFreeExtensionList

Table 151 lists the X Window System Extension Routines and provides a short description of each routine.

Table 151. Extension routines

Routine	Description
XAllocID()	Returns a resource ID that can be used when creating new resources.
XSetCloseDisplay()	Defines a procedure to call when XCloseDisplay is called.
XSetCopyGC()	Defines a procedure to call when a GC is copied.
XSetCreateFont()	Defines a procedure to call when XLoadQueryFont is called.
XSetCreateGC()	Defines a procedure to call when a new GC is created.
XSetError()	Suppresses the call to an external error handling routine and defines an alternative routine for error handling.
XSetErrorString()	Defines a procedure to call when an I/O error is detected.
XSetEventToWire()	Defines a procedure to call when an event must be converted from the host to wire format.
XSetFreeFont()	Defines a procedure to call when XFreeFont is called.
XSetFreeGC()	Defines a procedure to call when a GC is freed.
XSetWireToEvent()	Defines a procedure to call when an event is converted from the wire to the host format.
XFreeExtensionList()	Frees memory allocated by XListExtensions.
XListExtensions()	Returns a list of all extensions supported by the server.
XQueryExtension()	Indicates whether a named extension is present.

MIT extensions to X

The AIX extensions described in the *IBM AIX X-Window Programmer's Reference* are not supported by the X Window System API provided by the TCP/IP library routines.

The following MIT extensions are supported by the TCP/IP X client code:

- SHAPE
- MITMISC
- MULTIBUF

Table 152 lists the routines that allow an application to use these extensions.

Table 152. MIT extensions to X

Routine	Description
XShapeQueryExtension	Queries to see if server supports the SHAPE extension.
XShapeQueryVersion	Checks the version number of the server SHAPE extension.
XShapeCombineRegion	Converts the specified region into a list of rectangles and calls XShapeRectangles.
XShapeCombineRectangles	Performs a CombineRectangles operation.
XShapeCombineMask	Performs a CombineMask operation.
XShapeCombineShape	Performs a CombineShape operation.
XShapeOffsetShape	Performs an OffsetShape operation.
XShapeQueryExtents	Sets the extents of the bounding and clip shapes.
XShapeSelectInput	Selects Input Events.
XShapeInputSelected	Returns the current input mask for extension events on the specified window.
XShapeGetRectangles	Gets a list of rectangles describing the region specified.
XMITMiscQueryExtension	Queries to see if server supports the MITMISC extension.
XMITMiscSetBugMode	Sets the compatibility mode switch.
XMITMiscGetBugMode	Queries the compatibility mode switch.
XmbufQueryExtension	Queries to see if server supports the MULTIBUF extension.
XmbufGetVersion	Gets the version number of the extension.
XmbufCreateBuffers	Requests that multiple buffers be created.
XmbufDestroyBuffers	Requests that the buffers be destroyed.
XmbufDisplayBuffers	Displays the indicated buffers.
XmbufGetWindowAttributes	Gets the multibuffering attributes.
XmbufChangeWindowAttributes	Sets the multibuffering attributes.
XmbufGetBufferAttributes	Gets the attributes for the indicated buffer.
XmbufChangeBufferAttributes	Sets the attributes for the indicated buffer.
XmbufGetScreenInfo	Gets the parameters controlling how mono and stereo windows can be created on the indicated screen.
XmbufCreateStereoWindow	Creates a stereo window.

Associate table functions

When you need to associate arbitrary information with resource IDs, the XAssocTable allows you to associate your own data structures with X resources, such as bit maps, pixmaps, fonts, and windows.

An XAssocTable can be used to *type* X resources. For example, to create three or four types of windows with different properties, each window ID is associated with a pointer to a user-defined window property data structure. (A generic type, called XID, is defined in XLIB.H.)

Follow these guidelines when using an XAssocTable.

- Ensure the correct display is active before initiating an XAssocTable function, because all XIDs are relative to a specified display.
- Restrict the size of the table (number of buckets in the hashing system) to a power of two, and assign no more than eight XIDs for each bucket to maximize the efficiency of the table.

There is no restriction on the number of XIDs for each table or display, or the number of displays for each table.

Table 153 lists the Associate table functions and provides a short description of each function.

Table 153. Associate table functions

Routine	Description
XCreateAssocTable ()	Returns a pointer to the newly created associate table.
XDeleteAssoc()	Deletes an entry from the specified associate table.
XDestroyAssocTable()	Frees memory allocated to the specified associate table.
XLookupAssoc()	Obtains data from the specified associate table.
XMakeAssoc()	Creates an entry in the specified associate table.

Miscellaneous utility routines

The MIT X Miscellaneous Utility routines are included in SEZAX11L. These are a set of common utility functions that have been useful to application writers.

Table 154 lists the Miscellaneous utility routines and provides a short description of each routine.

Table 154. Miscellaneous utility routines

Routine	Description
XctCreate()	Creates an XctData structure for parsing a Compound Text string.
XctFree()	Frees all data associated with the XctData structure.
XctNextItem()	Parses the next <i>item</i> from the Compound Text string.
XctReset()	Resets the XctData structure to reparse the Compound Text string.
XmuAddCloseDisplayHook()	Adds a callback for the given display.

Table 154. Miscellaneous utility routines (continued)

Routine	Description
XmuAddInitializer()	Registers a procedure to be invoked the first time XmuCallInitializers is called on a given application context.
XmuAllStandardColormaps()	Creates all of the appropriate standard colormaps.
XmuCallInitializers()	Calls each of the procedures that have been registered with XmuAddInitializer.
XmuClientWindow()	Finds a window at or below the specified window.
XmuCompareISOLatin1()	Compares two strings, ignoring case differences.
XmuConvertStandardSelection()	Converts many standard selections.
XmuCopyISOLatin1Lowered()	Copies a string, changing all Latin-1 uppercase letters to lowercase.
XmuCopyISOLatin1Uppered()	Copies a string, changing all Latin-1 lowercase letters to uppercase.
XmuCreateColormap()	Creates a colormap.
XmuCreatePixmapFromBitmap()	Creates a pixmap of the specified width, height, and depth.
XmuCreateStippledPixmap()	Creates a two-pixel by one-pixel stippled pixmap of specified depth on the specified screen.
XmuCursorNameToIndex()	Returns the index in the standard cursor font for the name of a standard cursor.
XmuCvtFunctionToCallback()	Converts a callback procedure to a callback list containing that procedure.
XmuCvtStringToBackingStore()	Converts a string to a backing-store integer.
XmuCvtStringToBitmap()	Creates a bit map suitable for window manager icons.
XmuCvtStringToCursor()	Converts a string to a Cursor.
XmuCvtStringToJustify()	Converts a string to an XtJustify enumeration value.
XmuCvtStringToLong()	Converts a string to an integer of type long.
XmuCvtStringToOrientation()	Converts a string to an XtOrientation enumeration value.
XmuCvtStringToShapeStyle()	Converts a string to an integer shape style.
XmuCvtStringToWidget()	Converts a string to an immediate child widget of the parent widget passed as an argument.
XmuDeleteStandardColormap()	Removes the specified property from the specified screen.
XmuDQAddDisplay()	Adds the specified display to the queue.
XmuDQCreate()	Creates and returns an empty XmuDisplayQueue.
XmuDQDestroy()	Releases all memory associated with the specified queue.
XmuDQLookupDisplay()	Returns the queue entry for the specified display.
XmuDQNDisplays()	Returns the number of displays in the specified queue.
XmuDQRemoveDisplay()	Removes the specified display from the specified queue.
XmuDrawLogo()	Draws the <i>official</i> X Window System logo.
XmuDrawRoundedRectangle()	Draws a rounded rectangle.
XmuFillRoundedRectangle()	Draws a filled rounded rectangle.
XmuGetAtomName()	Returns the name of an Atom.

Table 154. Miscellaneous utility routines (continued)

Routine	Description
XmuGetColormapAllocation()	Determines the best allocation of reds, greens, and blues in a standard colormap.
XmuGetHostname()	Returns the host name.
XmuInternAtom()	Caches the Atom value for one or more displays.
XmuInternStrings()	Converts a list of atom names into Atom values.
XmuLocateBitmapFile()	Reads a file in standard bit map file format.
XmuLookupAPL()	This function is similar to XLookupString, except that it maps a key event to an APL string.
XmuLookupArabic()	This function is similar to XLookupString, except that it maps a key event to a Latin and Arabic (ISO 8859-6) string.
XmuLookupCloseDisplayHook()	Determines if a callback is installed.
XmuLookupCyrillic()	This function is similar to XLookupString, except that it maps a key event to a Latin and Cyrillic (ISO 8859-5) string.
XmuLookupGreek()	This function is similar to XLookupString, except that it maps a key event to a Latin and Greek (ISO 8859-7) string.
XmuLookupHebrew()	This function is similar to XLookupString, except that it maps a key event to a Latin and Hebrew (ISO 8859-8) string.
XmuLookupJISX0201()	This function is similar to XLookupString, except that it maps a key event to a string in the JIS X0201-1976 encoding.
XmuLookupKana()	This function is similar to XLookupString, except that it maps a key event to a string in the JIS X0201-1976 encoding.
XmuLookupLatin1()	This function is identical to XLookupString.
XmuLookupLatin2()	This function is similar to XLookupString, except that it maps a key event to a Latin-2 (ISO 8859-2) string.
XmuLookupLatin3()	This function is similar to XLookupString, except that it maps a key event to a Latin-3 (ISO 8859-3) string.
XmuLookupLatin4()	This function is similar to XLookupString, except that it maps a key event to a Latin-4 (ISO 8859-4) string.
XmuLookupStandardColormap()	Creates or replaces a standard colormap if one does not currently exist.
XmuLookupString()	Maps a key event into a specific key symbol set.
XmuMakeAtom()	Creates and initializes an opaque object.
XmuNameOfAtom()	Returns the name of an AtomPtr.
XmuPrintDefaultErrorMessage()	Prints an error message, equivalent to Xlib's default error message.
XmuReadBitmapData()	Reads a standard bit map file description.
XmuReadBitmapDataFromFile()	Reads a standard bit map file description from the specified file.
XmuReleaseStippledPixmap()	Frees a pixmap created with XmuCreateStippledPixmap.

Table 154. Miscellaneous utility routines (continued)

Routine	Description
XmuRemoveCloseDisplayHook()	Deletes a callback that has been added with XmuAddCloseDisplayHook.
XmuReshapeWidget()	Reshapes the specified widget, using the Shape extension.
XmuScreenOfWindow()	Returns the screen on which the specified window was created.
XmuSimpleErrorHandler()	A simple error handler for Xlib error conditions.
XmuStandardColormap()	Creates a standard colormap for the given screen.
XmuUpdateMapHints()	Clears the PPosition and PSize flags and sets the USPosition and USSize flags.
XmuVisualStandardColormaps()	Creates all of the appropriate standard colormaps for a given visual.

X authorization routines

The MIT X Authorization routines are included in SEZAX11L. These routines are used to deal with X authorization data in X clients.

Table 155 lists the X authorization routines and provides a short description of each routine.

Table 155. Authorization routines

Routine	Description
XauFileName()	Generates the default authorization file name.
XauReadAuth()	Reads the next entry from the authfile.
XuWriteAuth()	Writes an authorization entry to the authfile.
XauGetAuthByAddr()	Searches for an authorization entry.
XauLockAuth()	Does the work necessary to synchronously update an authorization file.
XauUnlockAuth()	Undoes the work of XauLockAuth.
XauDisposeAuth()	Frees storage allocated to hold an authorization entry.

X Window System toolkit

An X Window System Toolkit is a set of library functions layered on top of the X Window System Xlib functions that allows you to simplify the design of applications by providing an underlying set of common user interface functions. Included are mechanisms for defining and expanding interclient and intracomponent interaction independently, masking implementation details from both the application and component implementor.

An X Window System Toolkit consists of the following:

- A set of programming mechanisms, called Intrinsics, that are used to build widgets.
- An architectural model to help programmers design new widgets, with enough flexibility to accommodate different application interface layers.

- A consistent interface, in the form of a coordinated set of widgets and composition policies, some of which are application domain-specific, while others are common across several application domains.

The fundamental data type of the X Window System Toolkit is the widget. A widget is allocated dynamically and contains state information. Every widget belongs to one widget class that is allocated statically and initialized. The widget class contains the operations allowed on widgets of that class.

An X Window System Toolkit manages the following functions:

- Toolkit initialization
- Widgets and widget geometry
- Memory
- Window, data set, and timer events
- Input focus
- Selections
- Resources and resource conversion
- Translation of events
- Graphics contexts
- Pixmaps
- Errors and warnings

You must remap many of the X Widget and X Intrinsics routine names. This remapping is done in a header file called `XT@REMAP.H`. This file is automatically included by the `INTRINSIC.H` header file. In debugging your application, see the `XT@REMAP.H` file to find the remapped names of the X Toolkit routines.

Some of the X Window System header data sets have been renamed from their original distribution names, because of the data set naming conventions in the MVS environment. Such name changes are generally restricted to those header files used internally by the actual widget code, rather than the application header files, to minimize the number of changes required for an application to be ported to the MVS environment.

In porting applications to the MVS environment, you might have to make changes to header file names in Table 156.

Table 156. X Intrinsic header file names

MIT distribution name	TCP/IP name
CompositeI.h	ComposiI.h
CompositeP.h	ComposiP.h
ConstrainP.h	ConstraP.h
IntrinsicI.h	IntriniI.h
IntrinsicP.h	IntriniP.h
PassivGral.h	PassivGr.h
ProtocolsP.h	ProtocoP.h
SelectionI.h	SelectiI.h
WindowObjP.h	WindowOP.h

Xt Intrinsic routines

Table 157 provides the Xt Intrinsic routines and a short description of each routine.

Table 157. Xt Intrinsic routines

Routine	Description
CompositeClassPartInitialize	Initializes the CompositeClassPart of a composite widget.
CompositeDeleteChild	Deletes a child widget from a composite widget.
CompositeDestroy	Destroys a composite widget.
CompositeInitialize	Initializes a composite widget structure.
CompositeInsertChild	Inserts a child widget in a composite widget.
RemoveCallback	Removes a callback procedure from a callback list.
XrmCompileResourceList	Compiles an XtResourceList into an XrmResourceList.
XtAddActions	Declares an action table and registers it with the translation manager
XtAddCallback	Adds a callback procedure to the callback list of the specified widget.
XtAddCallbacks	Adds a list of callback procedures to the callback list of specified widget.
XtAddConverter	Adds a new converter.
XtAddEventHandler	Registers an event handler procedure with the dispatch mechanism when an event matching the mask occurs on the specified widget.
XtAddExposureToRegion	Computes the union of the rectangle defined by the specified exposure event and region.
XtAddGrab	Redirects user input to a model widget.
XtAddInput	Registers a new source of events.
XtAddRawEventHandler	Registers an event handler procedure with the dispatch mechanism without causing the server to select for that event.
XtAddTimeout	Creates a timeout value in the default application context and returns an identifier for it.
XtAddWorkProc	Registers a work procedure in the default application context.
XtAppAddActionHook	Adds an actionhook procedure to an application context.
XtAppAddActions	Declares an action table and registers it with the translation manager.
XtAppAddConverter	Registers a new converter.
XtAppAddInput	Registers a new file as an input source for a specified application.
XtAppAddTimeout	Creates a timeout value and returns an identifier for it.
XtAppAddWorkProc	Registers a work procedure for a specified procedure.
XtAppCreateShell	Creates a top-level widget that is the root of a widget tree.
XtAppError	Calls the installed unrecoverable error procedure.
XtAppErrorMsg	Calls the high-level error handler.

Table 157. Xt Intrinsic routines (continued)

Routine	Description
XtAppGetErrorDatabase	Obtains the error database and merges it with an application or database specified by a widget.
XtAppGetErrorDatabaseText	Obtains the error database text for an error or warning for an error message handler.
XtAppGetSelectionTimeout	Gets and returns the current selection timeout (ms) value.
XtAppInitialize	A convenience routine for initializing the toolkit.
XtAppMainLoop	Process input by calling XtAppNextEvent and XtDispatchEvent.
XtAppNextEvent	Returns the value from the top of a specified application input queue.
XtAppPeekEvent	Returns the value from the top of a specified application input queue without removing input from queue.
XtAppPending	Determines if the input queue has any events for a specified application.
XtAppProcessEvent	Processes applications that require direct control of the processing for different types of input.
XtAppReleaseCacheRefs	Decrements the reference count for the conversion entries identified by the refs argument.
XtAppSetErrorHandler	Registers a procedure to call on unrecoverable error conditions. The default error handler prints the message to standard error.
XtAppSetErrorMsgHandler	Registers a procedure to call on unrecoverable error conditions. The default error handler constructs a string from the error resource database.
XtAppSetFallbackResources	Sets the fallback resource list that will be loaded at display initialization time.
XtAppSetSelectionTimeout	Sets the Intrinsic selection timeout value.
XtAppSetTypeConverter	Registers the specified type converter and destructor in all application contexts created by the calling process.
XtAppSetWarningHandler	Registers a procedure to call on nonfatal error conditions. The default warning handler prints the message to standard error.
XtAppSetWarningMsgHandler	Registers a procedure to call on nonfatal error conditions. The default warning handler constructs a string from error resource database.
XtAppWarning	Calls the installed nonfatal error procedure.
XtAppWarningMsg	Calls the installed high-level warning handler.
XtAugmentTranslations	Merges new translations into an existing widget translation table.
XtBuildEventMask	Retrieves the event mask for a specified widget.
XtCallAcceptFocus	Calls the accept_focus procedure for the specified widget.
XtCallActionProc	Searches for the named action routine and, if found, calls it.
XtCallbackExclusive	Calls customized code for callbacks to create pop-up shell.
XtCallbackNone	Calls customized code for callbacks to create pop-up shell.

Table 157. Xt Intrinsic routines (continued)

Routine	Description
XtCallbackNonexclusive	Calls customized code for callbacks to create pop-up shell.
XtCallbackPopdown	Pops down a shell that was mapped by callback functions.
XtCallbackReleaseCacheRef	A callback that can be added to a callback list to release a previously returned XtCacheRef value.
XtCallbackReleaseCacheRefList	A callback that can be added to a callback list to release a list of previously returned XtCacheRef value.
XtCallCallbackList	Calls all callbacks on a callback list.
XtCallCallbacks	Executes the callback procedures in a widget callback list.
XtCallConverter	Looks up the specified type converter in the application context and invokes the conversion routine.
XtCalloc	Allocates and initializes an array.
XtClass	Obtains the class of a widget and returns a pointer to the widget class structure.
XtCloseDisplay	Closes a display and removes it from an application context.
XtConfigureWidget	Moves and resizes the sibling widget of the child making the geometry request.
XtConvert	Invokes resource conversions.
XtConvertAndStore	Looks up the type converter registered to convert from_type to to_type and then calls XtCallConverter.
XtConvertCase	Determines upper and lowercase equivalents for a KeySym.
XtCopyAncestorSensitive	Copies the sensitive value from a widget record.
XtCopyDefaultColormap	Copies the default colormap from a widget record.
XtCopyDefaultDepth	Copies the default depth from a widget record.
XtCopyFromParent	Copies the parent from a widget record.
XtCopyScreen	Copies the screen from a widget record.
XtCreateApplicationContext	Creates an opaque type application context.
XtCreateApplicationShell	Creates an application shell widget by calling XtAppCreateShell.
XtCreateManagedWidget	Creates and manages a child widget in a single procedure.
XtCreatePopupShell	Creates a pop-up shell.
XtCreateWidget	Creates an instance of a widget.
XtCreateWindow	Calls XcreateWindow with the widget structure and parameter.
XtDatabase	Obtains the resource database for a particular display.
XtDestroyApplicationContext	Destroys an application context.
XtDestroyGC	Deallocates graphics context when it is no longer needed.
XtDestroyWidget	Destroys a widget instance.
XtDirectConvert	Invokes resource conversion.

Table 157. Xt Intrinsic routines (continued)

Routine	Description
XtDisownSelection	Informs the Intrinsic selection mechanism that the specified widget is to lose ownership of the selection.
XtDispatchEvent	Receives X events and calls appropriate event handlers.
XtDisplay	Returns the display pointer for the specified widget.
XtDisplayInitialize	Initializes a display and adds it to an application context.
XtDisplayOfObject	Returns the display pointer for the specified widget.
XtDisplayStringConversionWarning	Issues a warning message for conversion routines.
XtDisplayToApplicationContext	Retrieves the application context associated with a Display.
XtError	Calls the installed unrecoverable error procedure.
XtErrorMsg	A low-level error and warning handler procedure type.
XtFindFile	Searches for a file using substitutions in a path list.
XtFree	Frees an allocated block of storage.
XtGetActionKeysym	Retrieves the KeySym and modifiers that matched the final event specification in a translation table entry.
XtGetApplicationNameAndClass	Returns the application name and class as passed to XtDisplayInitialize
XtGetApplicationResources	Retrieves resources that are not specific to a widget, but apply to the overall application.
XtGetConstraintResourceList	Returns the constraint resource list for a particular widget.
XtGetErrorDatabase	Obtains the error database and returns the address of the error database.
XtGetErrorDatabaseText	Obtains the error database text for an error or warning.
XtGetGC	Returns a read-only sharable GC.
XtGetKeysymTable	Returns a pointer to the KeySym to KeyCode mapping table for a particular display.
XtGetMultiClickTime	Returns the multclick time setting.
XtGetResourceList	Obtains the resource list structure for a particular class.
XtGetSelectionRequest	Retrieves the SelectionRequest event that triggered the convert_selection procedure.
XtGetSelectionTimeout	Obtains the current selection timeout.
XtGetSelectionValue	Obtains the selection value in a single, logical unit.
XtGetSelectionValueIncremental	Obtains the selection value using incremental transfers.
XtGetSelectionValues	Takes a list of target types and client data and obtains the current value of the selection converted to each of the targets.
XtGetSelectionValuesIncremental	A function similar to XtGetSelectionValueIncremental except that it takes a list of targets and client_data.
XtGetSubresources	Obtains resources other than widgets.
XtGetSubvalues	Retrieves the current value of a nonwidget resource data associated with a widget instance.
XtGetValues	Retrieves the current value of a resource associated with a widget instance.

Table 157. Xt Intrinsics routines (continued)

Routine	Description
XtGrabButton	Passively grabs a single pointer button.
XtGrabKey	Passively grabs a single key of the keyboard.
XtGrabKeyboard	Actively grabs the keyboard.
XtGrabPointer	Actively grabs the pointer.
XtHasCallbacks	Finds the status of a specified widget callback list.
XtInitialize	Initializes the toolkit, application, and shell.
XtInitializeWidgetClass	Initializes a widget class without creating any widgets.
XtInsertEventHandler	Registers an event handler procedure that receives events before or after all previously registered event handlers.
XtInsertRawEventHandler	Registers an event handler procedure that receives events before or after all previously registered event handlers without selecting for the events.
XtInstallAccelerators	Installs accelerators from a source widget to destination widget.
XtInstallAllAccelerators	Installs all the accelerators from a widget and all the descendants of the widget onto one destination widget.
XtIsApplicationShell	Determines whether a specified widget is a subclass of an application shell widget.
XtIsComposite	Determines whether a specified widget is a subclass of a composite widget.
XtIsConstraint	Determines whether a specified widget is a subclass of a constraint widget.
XtIsManaged	Determines the managed state of a specified child widget.
XtIsObject	Determines whether a specified widget is a subclass of an object widget.
XtIsOverrideShell	Determines whether a specified widget is a subclass of an override shell widget.
XtIsRealized	Determines if a widget has been realized.
XtIsRectObj	Determines whether a specified widget is a subclass of a RectObj widget.
XtIsSensitive	Determines the current sensitivity state of a widget.
XtIsShell	Determines whether a specified widget is a subclass of a shell widget.
XtIsSubclass	Determines whether a specified widget is in a specific subclass.
XtIsTopLevelShell	Determines whether a specified widget is a subclass of a TopLevelShell widget.
XtIsTransientShell	Determines whether a specified widget is a subclass of a TransientShell widget.
XtIsVendorShell	Determines whether a specified widget is a subclass of a VendorShell widget.
XtIsWidget	Determines whether a specified widget is a subclass of a Widget widget.
XtIsWMShell	Determines whether a specified widget is a subclass of a WMShell widget.

Table 157. Xt Intrinsics routines (continued)

Routine	Description
XtKeysymToKeycodeList	Returns the list of KeyCodes that map to a particular KeySym.
XtLastTimestampProcessed	Retrieves the timestamp from the most recent call to XtDispatchEvent.
XtMainLoop	An infinite loop that processes input.
XtMakeGeometryRequest	A request from the child widget to a parent widget for a geometry change.
XtMakeResizeRequest	Makes a resize request from a widget.
XtMalloc	Allocates storage.
XtManageChild	Adds a single child to a parent widget list of managed children.
XtManageChildren	Adds a list of widgets to the geometry-managed, displayable, subset of its composite parent widget.
XtMapWidget	Maps a widget explicitly.
XtMenuPopupAction	Pops up a menu when a pointer button is pressed or when the pointer is moved into the widget.
XtMergeArgLists	Merges two ArgList structures.
XtMoveWidget	Moves a sibling widget of the child making the geometry request.
XtName	Returns a pointer to the instance name of the specified object.
XtNameToWidget	Translates a widget name to a widget instance.
XtNewString	Copies an instance of a string.
XtNextEvent	Returns the value from the header of the input queue.
XtOpenDisplay	Opens, initializes, and adds a display to an application context.
XtOverrideTranslations	Overwrites existing translations with new translations.
XtOwnSelection	Sets the selection owner when using atomic transfer.
XtOwnSelectionIncremental	Sets the selection owner when using incremental transfers.
XtParent	Returns the parent widget for the specified widget.
XtParseAcceleratorTable	Parses an accelerator table into the opaque internal representation.
XtParseTranslationTable	Compiles a translation table into the opaque internal representation of type XtTranslations.
XtPeekEvent	Returns the value from the front of the input queue without removing it from the queue.
XtPending	Determines if the input queue has events pending.
XtPopdown	Unmaps a pop-up from within an application.
XtPopup	Maps a pop-up from within an application.
XtPopupSpringLoaded	Maps a spring-loaded pop-up from within an application.
XtProcessEvent	Processes one input event, timeout, or alternate input source.
XtQueryGeometry	Queries the preferred geometry of a child widget.

Table 157. Xt Intrinsic routines (continued)

Routine	Description
XtRealizeWidget	Realizes a widget instance.
XtRealloc	Changes the size of an allocated block of storage, sometimes moving it.
XtRegisterCaseConverter	Registers a specified case converter.
XtRegisterGrabAction	Registers button and key grabs for a widget window according to the event bindings in the widget translation table.
XtReleaseGC	Deallocates a shared GC when it is no longer needed.
XtRemoveActionHook	Removes an action hook procedure without destroying the application context.
XtRemoveAllCallbacks	Deletes all callback procedures from a specified widget callback list.
XtRemoveCallback	Deletes a callback procedure from a specified widget callback list only if both the procedure and the client data match.
XtRemoveCallbacks	Deletes a list of callback procedures from a specified widget callback list.
XtRemoveEventHandler	Removes a previously registered event handler.
XtRemoveGrab	Removes the redirection of user input to a modal widget.
XtRemoveInput	Discontinues a source of input by causing the Intrinsics read routine to stop watching for input from the input source.
XtRemoveRawEventHandler	Removes previously registered raw event handler.
XtRemoveTimeout	Clears a timeout value by removing the timeout.
XtRemoveWorkProc	Removes the specified background work procedure.
XtResizeWidget	Resizes a sibling widget of the child making the geometry request.
XtResizeWindow	Resizes a child widget that already has the values for its width, height, and border width.
XtResolvePathname	Searches for a file using standard substitutions in a path list.
XtScreen	Returns the screen pointer for the specified widget.
XtScreenOfObject	Returns the screen pointer for the nearest ancestor of object that is of class Widget.
XtSetErrorHandler	Registers a procedure to call under unrecoverable error conditions.
XtSetErrorMsgHandler	Registers a procedure to call under unrecoverable error conditions.
XtSetKeyboardFocus	Redirects keyboard input to a child of a composite widget without calling XSetInputFocus.
XtSetKeyTranslator	Registers a key translator.
XtSetMappedWhenManaged	Changes the widget map_when_managed field.
XtSetMultiClickTime	Sets the multi-click time for an application.
XtSetSelectionTimeout	Sets the Intrinsics selection timeout.
XtSetSensitive	Sets the sensitivity state of a widget.

Table 157. Xt Intrinsic routines (continued)

Routine	Description
XtSetSubvalues	Sets the current value of a nonwidget resource associated with an instance.
XtSetTypeConverter	Registers a type converter for all application contexts in a process.
XtSetValues	Modifies the current value of a resource associated with widget instance.
XtSetWarningHandler	Registers a procedure to be called on non-fatal error conditions.
XtSetWarningMsgHandler	Registers a procedure to be called on nonfatal error conditions.
XtSetWMColormapWindows	Sets the value of the WM_COLORMAP_WINDOWS property on a widget's window.
XtStringConversionWarning	A convenience routine for old format resource converters that convert from strings.
XtSuperclass	Obtains the superclass of a widget by returning a pointer to the superclass structure of the widget.
XtToolkitInitialize	Initializes the X Toolkit internals.
XtTranslateCoords	Translates an [x,y] coordinate pair from widget coordinates to root coordinates.
XtTranslateKey	The default key translator routine.
XtTranslateKeycode	Registers a key translator.
XtUngrabButton	Cancels a passive button grab.
XtUngrabKey	Cancels a passive key grab.
XtUngrabKeyboard	Cancels an active keyboard grab.
XtUngrabPointer	Cancels an active pointer grab.
XtUninstallTranslations	Causes the entire translation table for widget to be removed.
XtUnmanageChild	Removes a single child from the managed set of its parent.
XtUnmanageChildren	Removes a list of children from the managed list of the parent, but does not destroy the children widgets.
XtUnmapWidget	Unmaps a widget explicitly.
XtUnrealizeWidget	Destroys the associated widget and its descendants.
XtVaAppCreateShell	Creates a top-level widget that is the root of a widget tree using varargs lists.
XtVaAppInitialize	Initializes the Xtk internals, creates an application context, opens and initializes a display, and creates the initial application shell instance using varargs lists.
XtVaCreateArgsList	Dynamically allocates a varargs list for use with XtVaNestedList in multiple calls.
XtVaCreateManagedWidget	Creates and manages a child widget in a single procedure using varargs lists.
XtVaCreatePopupShell	Creates a pop-up shell using varargs lists.
XtVaCreateWidget	Creates an instance of a widget using varargs lists.

Table 157. Xt Intrinsic routines (continued)

Routine	Description
XtVaGetApplicationResources	Retrieves resources for the overall application using varargs list.
XtVaGetSubresources	Fetches resources for widget subparts using varargs list.
XtVaGetSubvalues	Retrieves the current values of nonwidget resources associated with a widget instance using varargs lists.
XtVaGetValues	Retrieves the current values of resources associated with a widget instance using varargs lists.
XtVaSetSubvalues	Sets the current values of nonwidget resources associated with a widget instance using varargs lists.
XtVaSetValues	Modifies the current values of resources associated with a widget instance using varargs lists.
XtWarning	Calls the installed non-fatal error procedure.
XtWarningMsg	Calls the installed high-level warning handler.
XtWidgetToApplicationContext	Gets the application context for given widget.
XtWindow	Returns the window of the specified widget.
XtWindowOfObject	Returns the window for the nearest ancestor of object that is of class Widget.
XtWindowToWidget	Translates a window and display pointer into a widget instance.

Application resources

X applications can be modified at run time by a set of resources. Applications that make use of an X Window System toolkit can be modified by additional sets of application resources. These resources are searched until a resource specification is found. The X Intrinsics determine the actual search order used for determining a resource value.

The search order used in the TSO environment, in descending order of preference, is:

1. Command Line
Standard arguments include:
 - a. Command switches (-display, -fg, -foreground, +rv)
 - b. Resource manager directives (-name, -xrm)
 - c. Natural language directive (-xnllanguage)
2. User Environment File
Use the source found from the *user_id.XDEFAULT.host* data set. In this case, *host* is the string returned by the `gethostname()` call.
3. Server and User Preference Resources
Use the first source found from:
 - a. RESOURCE_MANAGER property on the root window [screen0]
 - b. *user_id.X.DEFAULTS* data set
4. Application Class Resources
Use the first source found from:
 - a. The default application resource data set named *user_id.XAPDF.classname*, where *classname* is the application specified class name.

The MVS data set name XAPDF is modified, if a natural language directive is specified as *xnllanguage*XAPDF, where *xnllanguage* is the string specified by the natural language directive.

- b. Fallback resources defined by XtAppSetFallbackResources within the application.

Athena widget support

The X Window System support with TCP/IP includes the widget set developed at Massachusetts Institute of Technology (MIT), which is generally known as the Athena widget set.

The Athena widget set supports the following widgets:

AsciiSink	Paned
AsciiSrc	Scrollbar
AsciiText	Simple
Box	SimpleMenu
Clock	Sme (Simple Menu Entry)
Command	SmeBSB (BSB Menu Entry)
Dialog	SmeLine
Form	StripChart
Grip	Text
Label	TextSink
List	TextSrc
Logo	Toggle
Mailbox	VPaned
MenuButton	Viewport

Table 158 provides the Athena widget routines with a short description of each routine.

Table 158. Athena widget routines

Routine	Description
XawAsciiSave	Saves the changes made in the current text source into a file.
XawAsciiSaveAsFile	Saves the contents of the current text buffer into a named file.
XawAsciiSourceChanged	Determines if the text buffer in an AsciiSrc object has changed.
XawAsciiSourceFreeString	Frees the storage associated with the string from an AsciiSrc widget requested with a call to XtGetValues.
XawDialogAddButton	Adds a new button to a Dialog widget.
XawDialogGetValueString	Returns the character string in the text field of a Dialog Widget.
XawDiskSourceCreate	Creates a disk source.
XawFormDoLayout	Forces or defers a relayout of the Form.
XawInitializeWidgetSet	Forces a reference to vendor shell so that the one in this widget is installed.
XawListChange	Changes the list that is displayed.
XawListHighlight	Highlights an item in the list.
XawListShowCurrent	Retrieves the list element that is currently set.

Table 158. Athena widget routines (continued)

Routine	Description
XawListUnhighlight	Unhighlights an item in the list.
XawPanedAllowResize	Enables or disables a child's request for pane resizing.
XawPanedGetMinMax	Retrieves the minimum and maximum height settings for a pane.
XawPanedGetNumSub	Retrieves the number of panes in a paned widget.
XawPanedSetMinMax	Sets the minimum and maximum height settings for a pane.
XawPanedSetRefigureMode	Enables or disables automatic recalculation of pane sizes and positions.
XawScrollbarSetThumb	Sets the position and length of a Scrollbar thumb.
XawSimpleMenuAddGlobalActions	Registers an XawPositionSimpleMenu global action routine.
XawSimpleMenuClearActiveEntry	Clears the SimpleMenu widget internal information about the currently highlighted menu entry.
XawSimpleMenuGetActiveEntry	Gets the currently highlighted menu entry.
XawStringSourceCreate	Creates a string source.
XawTextDisableRedisplay	Disables redisplay while making several changes to a Text Widget.
XawTextDisplay	Displays batched updates.
XawTextDisplayCaret	Enables and disables the insert point.
XawTextEnableRedisplay	Enables redisplay.
XawTextGetInsertionPoint	Returns the current position of the insert point.
XawTextGetSelectionPos	Retrieves the text that has been selected by this text widget.
XawTextGetSource	Retrieves the current text source for the specified widget.
XawTextInvalidate	Redisplays a range of characters.
XawTextReplace	Modifies the text in an editable Text widget.
XawTextSearch	Searches for a string in a Text widget.
XawTextSetInsertionPoint	Moves the insert point to the specified source position.
XawTextSetLastPos	Sets the last position data in an AsciiSource Object.
XawTextSetSelection	Selects a piece of text.
XawTextSetSelectionArray	Assigns a new selection array to a text widget.
XawTextSetSource	Replaces the text source in the specified widget.
XawTextSinkClearToBackground	Clears a region of the sink to the background color.
XawTextSinkDisplayText	Stub function that in subclasses will display text.
XawTextSinkFindDistance	Finds the Pixel Distance between two text positions.
XawTextSinkFindPosition	Finds a position in the text.
XawTextSinkGetCursorBounds	Finds the bounding box for the insert cursor.
XawTextSinkInsertCursor	Places the InsertCursor.
XawTextSinkMaxHeight	Finds the minimum height that contains a given number of lines.
XawTextSinkMaxLines	Finds the maximum number of lines that fit in a given height.

Table 158. Athena widget routines (continued)

Routine	Description
XawTextSinkResolve	Resolves a location to a position.
XawTextSinkSetTabs	Sets the Tab stops.
XawTextSourceConvertSelection	Dummy selection converter.
XawTextSourceRead	Reads the source into a buffer.
XawTextSourceReplace	Replaces a block of text with new text.
XawTextSourceScan	Scans the text source for the number and type of item specified.
XawTextSourceSearch	Searches the text source for the text block passed.
XawTextSourceSetSelection	Allows special setting of the selection.
XawTextTopPosition	Returns the character position of the left-most character on the first line displayed in the widget.
XawTextUnsetSelection	Unhighlights previously highlighted text in a widget.
XawToggleChangeRadioGroup	Allows a toggle widget to change radio groups.
XawToggleGetCurrent	Returns the RadioData associated with the toggle widget that is currently active in a toggle group.
XawToggleSetCurrent	Sets the Toggle widget associated with the radio_data specified.
XawToggleUnsetCurrent	Unsets all Toggles in the radio_group specified.

Some of the header files have been renamed from their original distribution names because of the data set naming conventions in the MVS environment. In addition, some of the header file names were changed to eliminate duplicate data set names with the Motif-based widget support. If your application uses these header files, you must use the new header file names in Table 159. These data set members can be found in the SEZACMAC partitioned data set. They carry an H extension in this text to distinguish them as header files.

Table 159. Athena header file names

MIT distribution name	TCP/IP name
AsciiSinkP.h	AscSinkP.h
AsciiSrcP.h	AscSrcP.h
AsciiTextP.h	AscTextP.h
Command.h	ACommand.h
CommandP.h	ACommanP.h
Form.h	AForm.h
FormP.h	AFormP.h
Label.h	ALabel.h
LabelP.h	ALabelP.h
List.h	AList.h
ListP.h	AListP.h
MenuButtoP.h	MenuButP.h
Scrollbar.h	AScrollb.h
ScrollbarP.h	AScrollP.h

Table 159. Athena header file names (continued)

MIT distribution name	TCP/IP name
SimpleMenP.h	SimpleMP.h
StripCharP.h	StripChP.h
TemplateP.h	TemplatP.h
Text.h	AText.h
TextP.h	ATextP.h
TextSinkP.h	TextSinP.h
TextSrcP.h	ATextSrP.h
ViewportP.h	ViewporP.h

Motif-based widget support

The X Window System support with TCP/IP includes the Motif-based widget set (Release 1.1).

The Motif-based widget set supports the following gadgets and widgets:

ArrowButton, ArrowGadget, and ArrowButtonGadget	MenuShell
BulletinBoard	MessageBox
CascadeButton	PanedWindow
and CascadeButtonGadget	PushButton and PushButtonGadget
Command	RowColumn
DialogShell	Sash
DrawingArea	Scale
DrawnButton	ScrollBar
Form	ScrolledWindow
Frame	SelectionBox and SelectionDialog
Label and LabelGadget	Separator and SeparatorGadget
List	Text
MainWindow	ToggleButton and ToggleButtonGadget

FileSelectionBox and FileSelectionDialog widgets are not supported in TCP/IP Version 3 Release 2 for MVS.

To run a Motif-based application, you must copy the module SEZAINST(KEYSYMDB) to *hlq*.XKEYSYM.DB or *user_id*.XKEYSYM.DB to make it available to your application at run-time.

Note: The EZAGETIN job copies SEZAINST(KEYSYMDB) to *hlq*.XKEYSYM.DB.

Some of the header files have been renamed from their original distribution names because of the data set naming conventions in the MVS environment. Such name changes are generally restricted to those header files used internally by the actual widget code, rather than the application header data sets, to minimize the number of changes required for an application to be ported to the MVS environment.

When porting applications to the MVS environment, you might have to make changes to the header file names in Table 160 on page 713. These data set members can be found in the SEZACMAC partitioned data set. They carry an H extension in this text to distinguish them as header files.

Table 160. Motif header file names

Motif distribution name	TCP/IP name
BulletinBP.h	BulletBP.h
CascadeBG.h	CascadBG.h
CascadeBGP.h	CascaBGP.h
CascadeBP.h	CascadBP.h
CutPasteP.h	CutPastP.h
DrawingAP.h	DrawinAP.h
ExtObjectP.h	ExtObjeP.h
MenuShellP.h	MenuSheP.h
MessageBP.h	MessageBP.h
ProtocolsP.h	ProtocoP.h
RowColumnP.h	RowColuP.h
ScrollBarP.h	ScrollBP.h
ScrolledWP.h	ScrollWP.h
SelectioB.h	SelectiB.h
SelectioBP.h	SelectBP.h
SeparatoG.h	SeparatG.h
SeparatoGP.h	SeparaGP.h
SeparatorP.h	SeparatP.h
ToggleBGP.h	TogglBGP.h
TraversaI.h	TraversI.h
VirtKeysP.h	VirtKeyP.h

z/OS UNIX System Services support

The following sections provide information about using z/OS UNIX System Services for the X Window System.

For information about using z/OS UNIX System Services sockets, see the *z/OS XL C/C++ Run-Time Library Reference*.

What is provided with z/OS UNIX System Services

The z/OS UNIX System Services X Window System support provided with TCP/IP includes the following APIs:

- SEZAROE1 and SEZACMTX compiled to run under z/OS UNIX System Services. SEZAROE1 is a combination of the reentrant versions of the X Window System libraries (see the *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference* for information about data sets).
- .SEZAROE2 (z/OS UNIX System Services Athena Widget set for reentrant modules).
- SEZAROE3 (z/OS UNIX System Services Motif Widget set for reentrant modules).

The SEZAROE1, SEZAROE2, and SEZAROE3 library members are:

- Fixed block 80, in object deck format.
- Compiled with the C/370 RENT compile-time option.

- Used as input for reentrant z/OS UNIX System Services X Window System and socket programs.
- Passed to the C/370 prelinker. Use the prelink utility to combine all input text decks into a single text deck.

z/OS UNIX System Services software requirements

Application programs using the X Window System API in z/OS UNIX System Services are written in C and should be compiled, linked, and executed using the z/OS C/C++ Compiler and the run-time environment of the Language Environment for MVS that is provided with z/OS.

You must have the AD/Cycle C/370 Library V1R2M0 and the AD/Cycle LE/370 Library V1R3M0 available when you compile and link your program.

You must include MANIFEST.H as the first #include statement in the source of every z/OS UNIX System Services MVS X Window System application program to remap the socket functions to the correct run-time library names.

In z/OS UNIX System Services, the DISPLAY environment variable is used by the X Window System to identify the host name of the target display.

z/OS UNIX System Services application resource file

The X Window System allows you to modify certain characteristics of an application at run time by means of application resources. Typically, application resources are set to tailor the appearance and possibly the behavior of an application. The application resources can specify information about application window sizes, placement, coloring, font usage, and other functional details.

In the z/OS UNIX System Services environment, this information can be found in the file:

```
/u/user_id/.Xdefaults
```

where:

```
/u/user_id
```

is found from the environment variable *home*.

Identifying the target display in z/OS UNIX System Services

The *DISPLAY* environment variable is used by the X Window System to identify the host name of the target display.

The following is the format of the *DISPLAY* environment variable:

```
host_name:target_server.target_screen
```

Value Description

host_name

Specifies the host name or IP address of the host machine on which the X Window System server is running.

target_server

Specifies the number of the display server on the host machine.

target_screen

Specifies the screen to be used on the target server.

For more information about resolving a host name to an IP address, see the *z/OS XL C/C++ Run-Time Library Reference*.

Compiling and linking with z/OS UNIX System Services

The following steps describe how to compile, link-edit, and run your z/OS UNIX System Services X Window System application under MVS batch, using the EDCCLG cataloged procedure supplied by IBM.

You must make the following changes to the EDCCLG cataloged procedure, which is supplied with AD/Cycle C/370 Version 1 Release 2 Compiler Licensed Program (5688-216).

In the compile step, make the following changes:

- Change the CPARM parameters to specify one of the following:
 - CPARM='DEF(IBMCP),RENT,LO'
 - CPARM='DEF(IBMCP,_POSIX1_SOURCE=1),RENT,LO'
 - CPARM='DEF(IBMCP,_OPEN_SYS),RENT,LO'
 - CPARM='DEF(IBMCP,_OPEN_SOCKETS,_POSIX1_SOURCE=1),RENT,LO'
 - CPARM='DEF(IBMCP,_OPEN_SOCKETS,_OPEN_SYS),RENT,LO'

Note: The recommended CPARMS are:

```
CPARM='DEF(IBMCP,_OPEN_SOCKETS,_POSIX1_SOURCE=1),RENT,LO'
```

RENT is the reentrant option and LO is the long-name option. You must specify these options to use z/OS UNIX System Services MVS functions. You must also specify the feature text macro, IBMCP.

If you choose to just access the z/OS UNIX System Services MVS functions defined by the POSIX standards 1003.1, 1003.1a, 1003.2, and 1003.4a, then specify the feature test macro POSIX1_SOURCE=1 to expose the appropriate definitions for the read(), write(), fcntl(), and close() functions.

If you choose to access all of the z/OS UNIX System Services MVS functions supported by C/370, including those defined by the POSIX standards 1003.1, 1003.1a, 1003.2, and 1003.4a, then specify the feature test macro _OPEN_SYS.

If you choose to access the z/OS UNIX System Services MVS socket functions or errno values, then specify the feature test macro _OPEN_SOCKETS to expose the socket-related definitions in all of the include files.

Note: Compile all C source using the def(IBMCP) preprocessor symbol. See “Compiling and linking” on page 667 for information about compiling and linking your program in MVS.

For a complete discussion of all of the AD/Cycle C/370 parameters, see the *AD/Cycle C/370 Programming Guide*.

- Add the following lines after the //SYSLIB DD statement for the IBM C/370 library edc.v1r2m0.SEDCDHDR:

```
// DD DSN=sys1.SFOMHDRS,DISP=SHR
// DD DSN=SEZACMAC,DISP=SHR
```

- Add the following //USERLIB DD statement:

```
//USERLIB DD DSN=USER.MYPROG.H,DISP=SHR
```

In the prelink edit step, make the following changes:

- Add the following prelink parameter:
PPARM='OMVS'

- To link-edit programs that use only X11 library functions, add the following line after the prelink //SYSLIB DD statement for the IBM AD/Cycle C/370 library cee.v1r3m0.SCEE OBJ:

```
// DD DSN=SEZAROE1,DISP=SHR
```

- To link-edit programs that use the Athena Toolkit functions, including Athena widget sets, add the following lines after the prelink //SYSLIB DD statement for the IBM AD/Cycle C/370 library cee.v1r3m0.SCEE OBJ:

```
// DD DSN=SEZAROE2,DISP=SHR
// DD DSN=SEZAROE1,DISP=SHR
```

- To link-edit programs that use the Motif Toolkit functions, add the following lines after the prelink //SYSLIB DD statement for the IBM AD/Cycle C/370 library cee.v1r3m0.SCEE OBJ:

```
// DD DSN=SEZAROE3,DISP=SHR
// DD DSN=SEZAROE1,DISP=SHR
```

For a complete discussion about compiling and link-editing the X Window System in MVS z/OS UNIX System Services, see the *z/OS XL C/C++ Run-Time Library Reference*.

To execute your program in the z/OS UNIX System Services shell, make the following changes:

- Set the DISPLAY environment variable to the name or IP address of the X server on which you want to display the application output. The following is an example:

```
DISPLAY=CHARM.RALEIGH.IBM.COM:0.0
export DISPLAY
```

- Allow the host application access to the X server.

On the workstation where you want to display the application output, you must grant permission for the MVS host to access the X server. To do this, enter the xhost command:

```
xhost ralmvs1
```

Compiling and linking with z/OS UNIX System Services using c89

The following c89 utility options can be specified:

- IBMCPP must always be specified.
- The c89 utility assumes _OPEN_SYS and includes all of the z/OS UNIX System Services MVS functions supported by C/370. However, _OPEN_SOCKETS must be specified if z/OS UNIX System Services MVS sockets are being used by the application program.

```
-D IBMCPP
-D _OPEN_SOCKETS
```

Notes:

1. When you compile and link-edit your application program using the c89 utility with z/OS UNIX System Services sockets and TCP/IP Version 3 Release 1 for X Window System, you must include the z/OS UNIX System Services socket library before the X Window System include files:

```
-I"//'  
sys1.SFOMHDRS"  
-I"//'SEZACMAC"  
  
-I"//'SEZAROE1"
```


2. The flag for the prelinker libraries, -l, is a dash followed by the lowercase letter L.
- If the Athena Toolkit functions are required, then also specify:
-l"// 'SEZAR0E2' "
- If the Motif Toolkit functions are required, then also specify:
-l"// 'SEZAR0E3' "

To execute your program under TSO, enter the following:

```
CALL 'USER.MYPROG.LOAD(PROGRAM1)' 'POSIX(ON)'
```

This loads the run-time library from cee.v1r3m0.SCEERUN. To use the z/OS UNIX System Services MVS C/370 functions, you must either specify the run-time option:

```
POSIX(ON)
```

or include the following statement in your C source program:

```
#pragma runopts(POSIX(ON))
```

Standard X client applications for z/OS UNIX System Services

For information about standard X Client applications for X Window System on z/OS UNIX System Services, see “Standard X client applications” on page 672.

Application resources for z/OS UNIX System Services

X applications can be modified at run time by a set of resources. Applications that make use of an X Window System toolkit can be modified by additional sets of application resources. These resources are searched until a resource specification is found. The X Intrinsics determine the actual search order used for determining a resource value.

The search order used in the z/OS UNIX System Services environment, in descending order of preference, is:

1. Command Line
Standard arguments include:
 - a. Command switches (-display, -fg, -foreground, +rv)
 - b. Resource manager directives (-name, -xrm)
 - c. Natural language directives (-xnllanguage)
2. User Environment File
Use the source found from the file /u/user_id/.Xdefault-host.
/u/user_id/.Xdefault-host is found from the environment variable *home*, and *host* is the string returned by the gethostname() call.
3. Server and User Preference Resources
Use the first source found from:
 - a. RESOURCE_MANAGER property on the root window [screen0]
 - b. /u/user_id/.Xdefaults
/u/user_id is found from the environment variable *home*.
4. Application Class Resources
Use the first source found from:
 - a. The default application resource file
/u/user_id/classname

where *classname* is the application specified class name, and `/u/user_id` is found from the environment variable *home*.

- b. Fallback resources defined in the file `/usr/lib/X11/app-defaults/classname` where *classname* is the application-specified class name.

Appendix E. Related protocol specifications

This appendix lists the related protocol specifications (RFCs) for TCP/IP. The Internet Protocol suite is still evolving through requests for comments (RFC). New protocols are being designed and implemented by researchers and are brought to the attention of the Internet community in the form of RFCs. Some of these protocols are so useful that they become recommended protocols. That is, all future implementations for TCP/IP are recommended to implement these particular functions or protocols. These become the *de facto* standards, on which the TCP/IP protocol suite is built.

You can request RFCs through electronic mail, from the automated Network Information Center (NIC) mail server, by sending a message to `service@nic.ddn.mil` with a subject line of RFC *nnnn* for text versions or a subject line of RFC *nnnn*.PS for PostScript versions. To request a copy of the RFC index, send a message with a subject line of RFC INDEX.

For more information, contact `nic@nic.ddn.mil` or at:

Government Systems, Inc.
Attn: Network Information Center
14200 Park Meadow Drive
Suite 200
Chantilly, VA 22021

Hard copies of all RFCs are available from the NIC, either individually or by subscription. Online copies are available at the following Web address:
<http://www.rfc-editor.org/rfc.html>.

See "Internet drafts" on page 734 for draft RFCs implemented in this and previous Communications Server releases.

Many features of TCP/IP Services are based on the following RFCs:

RFC	Title and Author
RFC 652	<i>Telnet output carriage-return disposition option</i> D. Crocker
RFC 653	<i>Telnet output horizontal tabstops option</i> D. Crocker
RFC 654	<i>Telnet output horizontal tab disposition option</i> D. Crocker
RFC 655	<i>Telnet output formfeed disposition option</i> D. Crocker
RFC 657	<i>Telnet output vertical tab disposition option</i> D. Crocker
RFC 658	<i>Telnet output linefeed disposition</i> D. Crocker
RFC 698	<i>Telnet extended ASCII option</i> T. Mock
RFC 726	<i>Remote Controlled Transmission and Echoing Telnet option</i> J. Postel, D. Crocker
RFC 727	<i>Telnet logout option</i> M.R. Crispin
RFC 732	<i>Telnet Data Entry Terminal option</i> J.D. Day
RFC 733	<i>Standard for the format of ARPA network text messages</i> D. Crocker, J. Vittal, K.T. Pogran, D.A. Henderson

RFC 734	<i>SUPDUP Protocol</i> M.R. Crispin
RFC 735	<i>Revised Telnet byte macro option</i> D. Crocker, R.H. Gumpertz
RFC 736	<i>Telnet SUPDUP option</i> M.R. Crispin
RFC 749	<i>Telnet SUPDUP—Output option</i> B. Greenberg
RFC 765	<i>File Transfer Protocol specification</i> J. Postel
RFC 768	<i>User Datagram Protocol</i> J. Postel
RFC 779	<i>Telnet send-location option</i> E. Killian
RFC 783	<i>TFTP Protocol (revision 2)</i> K.R. Sollins
RFC 791	<i>Internet Protocol</i> J. Postel
RFC 792	<i>Internet Control Message Protocol</i> J. Postel
RFC 793	<i>Transmission Control Protocol</i> J. Postel
RFC 820	<i>Assigned numbers</i> J. Postel
RFC 821	<i>Simple Mail Transfer Protocol</i> J. Postel
RFC 822	<i>Standard for the format of ARPA Internet text messages</i> D. Crocker
RFC 823	<i>DARPA Internet gateway</i> R. Hinden, A. Sheltzer
RFC 826	<i>Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware</i> D. Plummer
RFC 854	<i>Telnet Protocol Specification</i> J. Postel, J. Reynolds
RFC 855	<i>Telnet Option Specification</i> J. Postel, J. Reynolds
RFC 856	<i>Telnet Binary Transmission</i> J. Postel, J. Reynolds
RFC 857	<i>Telnet Echo Option</i> J. Postel, J. Reynolds
RFC 858	<i>Telnet Suppress Go Ahead Option</i> J. Postel, J. Reynolds
RFC 859	<i>Telnet Status Option</i> J. Postel, J. Reynolds
RFC 860	<i>Telnet Timing Mark Option</i> J. Postel, J. Reynolds
RFC 861	<i>Telnet Extended Options: List Option</i> J. Postel, J. Reynolds
RFC 862	<i>Echo Protocol</i> J. Postel
RFC 863	<i>Discard Protocol</i> J. Postel
RFC 864	<i>Character Generator Protocol</i> J. Postel
RFC 865	<i>Quote of the Day Protocol</i> J. Postel
RFC 868	<i>Time Protocol</i> J. Postel, K. Harrenstien
RFC 877	<i>Standard for the transmission of IP datagrams over public data networks</i> J.T. Korb
RFC 883	<i>Domain names: Implementation specification</i> P.V. Mockapetris
RFC 884	<i>Telnet terminal type option</i> M. Solomon, E. Wimmers
RFC 885	<i>Telnet end of record option</i> J. Postel
RFC 894	<i>Standard for the transmission of IP datagrams over Ethernet networks</i> C. Hornig
RFC 896	<i>Congestion control in IP/TCP internetworks</i> J. Nagle

RFC 903	<i>Reverse Address Resolution Protocol</i> R. Finlayson, T. Mann, J. Mogul, M. Theimer
RFC 904	<i>Exterior Gateway Protocol formal specification</i> D. Mills
RFC 919	<i>Broadcasting Internet Datagrams</i> J. Mogul
RFC 922	<i>Broadcasting Internet datagrams in the presence of subnets</i> J. Mogul
RFC 927	<i>TACACS user identification Telnet option</i> B.A. Anderson
RFC 933	<i>Output marking Telnet option</i> S. Silverman
RFC 946	<i>Telnet terminal location number option</i> R. Nedved
RFC 950	<i>Internet Standard Subnetting Procedure</i> J. Mogul, J. Postel
RFC 951	<i>Bootstrap Protocol</i> W.J. Croft, J. Gilmore
RFC 952	<i>DoD Internet host table specification</i> K. Harrenstien, M. Stahl, E. Feinler
RFC 959	<i>File Transfer Protocol</i> J. Postel, J.K. Reynolds
RFC 961	<i>Official ARPA-Internet protocols</i> J.K. Reynolds, J. Postel
RFC 974	<i>Mail routing and the domain system</i> C. Partridge
RFC 1001	<i>Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods</i> NetBios Working Group in the Defense Advanced Research Projects Agency, Internet Activities Board, End-to-End Services Task Force
RFC 1002	<i>Protocol Standard for a NetBIOS service on a TCP/UDP transport: Detailed specifications</i> NetBios Working Group in the Defense Advanced Research Projects Agency, Internet Activities Board, End-to-End Services Task Force
RFC 1006	<i>ISO transport services on top of the TCP: Version 3</i> M.T. Rose, D.E. Cass
RFC 1009	<i>Requirements for Internet gateways</i> R. Braden, J. Postel
RFC 1011	<i>Official Internet protocols</i> J. Reynolds, J. Postel
RFC 1013	<i>X Window System Protocol, version 11: Alpha update April 1987</i> R. Scheifler
RFC 1014	<i>XDR: External Data Representation standard</i> Sun Microsystems
RFC 1027	<i>Using ARP to implement transparent subnet gateways</i> S. Carl-Mitchell, J. Quarterman
RFC 1032	<i>Domain administrators guide</i> M. Stahl
RFC 1033	<i>Domain administrators operations guide</i> M. Lottor
RFC 1034	<i>Domain names—concepts and facilities</i> P.V. Mockapetris
RFC 1035	<i>Domain names—implementation and specification</i> P.V. Mockapetris
RFC 1038	<i>Draft revised IP security option</i> M. St. Johns
RFC 1041	<i>Telnet 3270 regime option</i> Y. Rekhter
RFC 1042	<i>Standard for the transmission of IP datagrams over IEEE 802 networks</i> J. Postel, J. Reynolds
RFC 1043	<i>Telnet Data Entry Terminal option: DODIIS implementation</i> A. Yasuda, T. Thompson

RFC 1044	<i>Internet Protocol on Network System's HYPERchannel: Protocol specification</i> K. Hardwick, J. Lekashman
RFC 1053	<i>Telnet X.3 PAD option</i> S. Levy, T. Jacobson
RFC 1055	<i>Nonstandard for transmission of IP datagrams over serial lines: SLIP</i> J. Romkey
RFC 1057	<i>RPC: Remote Procedure Call Protocol Specification: Version 2</i> Sun Microsystems
RFC 1058	<i>Routing Information Protocol</i> C. Hedrick
RFC 1060	<i>Assigned numbers</i> J. Reynolds, J. Postel
RFC 1067	<i>Simple Network Management Protocol</i> J.D. Case, M. Fedor, M.L. Schoffstall, J. Davin
RFC 1071	<i>Computing the Internet checksum</i> R.T. Braden, D.A. Borman, C. Partridge
RFC 1072	<i>TCP extensions for long-delay paths</i> V. Jacobson, R.T. Braden
RFC 1073	<i>Telnet window size option</i> D. Waitzman
RFC 1079	<i>Telnet terminal speed option</i> C. Hedrick
RFC 1085	<i>ISO presentation services on top of TCP/IP based internets</i> M.T. Rose
RFC 1091	<i>Telnet terminal-type option</i> J. VanBokkelen
RFC 1094	<i>NFS: Network File System Protocol specification</i> Sun Microsystems
RFC 1096	<i>Telnet X display location option</i> G. Marcy
RFC 1101	<i>DNS encoding of network names and other types</i> P. Mockapetris
RFC 1112	<i>Host extensions for IP multicasting</i> S.E. Deering
RFC 1113	<i>Privacy enhancement for Internet electronic mail: Part I — message encipherment and authentication procedures</i> J. Linn
RFC 1118	<i>Hitchhikers Guide to the Internet</i> E. Krol
RFC 1122	<i>Requirements for Internet Hosts—Communication Layers</i> R. Braden, Ed.
RFC 1123	<i>Requirements for Internet Hosts—Application and Support</i> R. Braden, Ed.
RFC 1146	<i>TCP alternate checksum options</i> J. Zweig, C. Partridge
RFC 1155	<i>Structure and identification of management information for TCP/IP-based internets</i> M. Rose, K. McCloghrie
RFC 1156	<i>Management Information Base for network management of TCP/IP-based internets</i> K. McCloghrie, M. Rose
RFC 1157	<i>Simple Network Management Protocol (SNMP)</i> J. Case, M. Fedor, M. Schoffstall, J. Davin
RFC 1158	<i>Management Information Base for network management of TCP/IP-based internets: MIB-II</i> M. Rose
RFC 1166	<i>Internet numbers</i> S. Kirkpatrick, M.K. Stahl, M. Recker
RFC 1179	<i>Line printer daemon protocol</i> L. McLaughlin
RFC 1180	<i>TCP/IP tutorial</i> T. Socolofsky, C. Kale

RFC 1183	<i>New DNS RR Definitions</i> C.F. Everhart, L.A. Mamakos, R. Ullmann, P.V. Mockapetris
RFC 1184	<i>Telnet Linemode Option</i> D. Borman
RFC 1186	<i>MD4 Message Digest Algorithm</i> R.L. Rivest
RFC 1187	<i>Bulk Table Retrieval with the SNMP</i> M. Rose, K. McCloghrie, J. Davin
RFC 1188	<i>Proposed Standard for the Transmission of IP Datagrams over FDDI Networks</i> D. Katz
RFC 1190	<i>Experimental Internet Stream Protocol: Version 2 (ST-II)</i> C. Topolcic
RFC 1191	<i>Path MTU discovery</i> J. Mogul, S. Deering
RFC 1198	<i>FYI on the X window system</i> R. Scheifler
RFC 1207	<i>FYI on Questions and Answers: Answers to commonly asked "experienced Internet user" questions</i> G. Malkin, A. Marine, J. Reynolds
RFC 1208	<i>Glossary of networking terms</i> O. Jacobsen, D. Lynch
RFC 1213	<i>Management Information Base for Network Management of TCP/IP-based internets: MIB-II</i> K. McCloghrie, M.T. Rose
RFC 1215	<i>Convention for defining traps for use with the SNMP</i> M. Rose
RFC 1227	<i>SNMP MUX protocol and MIB</i> M.T. Rose
RFC 1228	<i>SNMP-DPI: Simple Network Management Protocol Distributed Program Interface</i> G. Carpenter, B. Wijnen
RFC 1229	<i>Extensions to the generic-interface MIB</i> K. McCloghrie
RFC 1230	<i>IEEE 802.4 Token Bus MIB</i> K. McCloghrie, R. Fox
RFC 1231	<i>IEEE 802.5 Token Ring MIB</i> K. McCloghrie, R. Fox, E. Decker
RFC 1236	<i>IP to X.121 address mapping for DDN</i> L. Morales, P. Hasse
RFC 1256	<i>ICMP Router Discovery Messages</i> S. Deering, Ed.
RFC 1267	<i>Border Gateway Protocol 3 (BGP-3)</i> K. Lougheed, Y. Rekhter
RFC 1268	<i>Application of the Border Gateway Protocol in the Internet</i> Y. Rekhter, P. Gross
RFC 1269	<i>Definitions of Managed Objects for the Border Gateway Protocol: Version 3</i> S. Willis, J. Burruss
RFC 1270	<i>SNMP Communications Services</i> F. Kastenholtz, ed.
RFC 1285	<i>FDDI Management Information Base</i> J. Case
RFC 1315	<i>Management Information Base for Frame Relay DTEs</i> C. Brown, F. Baker, C. Carvalho
RFC 1321	<i>The MD5 Message-Digest Algorithm</i> R. Rivest
RFC 1323	<i>TCP Extensions for High Performance</i> V. Jacobson, R. Braden, D. Borman
RFC 1325	<i>FYI on Questions and Answers: Answers to Commonly Asked "New Internet User" Questions</i> G. Malkin, A. Marine
RFC 1327	<i>Mapping between X.400 (1988)/ISO 10021 and RFC 822</i> S. Hardcastle-Kille

RFC 1340	<i>Assigned Numbers</i> J. Reynolds, J. Postel
RFC 1344	<i>Implications of MIME for Internet Mail Gateways</i> N. Bornstein
RFC 1349	<i>Type of Service in the Internet Protocol Suite</i> P. Almquist
RFC 1350	<i>The TFTP Protocol (Revision 2)</i> K.R. Sollins
RFC 1351	<i>SNMP Administrative Model</i> J. Davin, J. Galvin, K. McCloghrie
RFC 1352	<i>SNMP Security Protocols</i> J. Galvin, K. McCloghrie, J. Davin
RFC 1353	<i>Definitions of Managed Objects for Administration of SNMP Parties</i> K. McCloghrie, J. Davin, J. Galvin
RFC 1354	<i>IP Forwarding Table MIB</i> F. Baker
RFC 1356	<i>Multiprotocol Interconnect on X.25 and ISDN in the Packet Mode</i> A. Malis, D. Robinson, R. Ullmann
RFC 1358	<i>Charter of the Internet Architecture Board (IAB)</i> L. Chapin
RFC 1363	<i>A Proposed Flow Specification</i> C. Partridge
RFC 1368	<i>Definition of Managed Objects for IEEE 802.3 Repeater Devices</i> D. McMaster, K. McCloghrie
RFC 1372	<i>Telnet Remote Flow Control Option</i> C. L. Hedrick, D. Borman
RFC 1374	<i>IP and ARP on HIPPI</i> J. Renwick, A. Nicholson
RFC 1381	<i>SNMP MIB Extension for X.25 LAPB</i> D. Throop, F. Baker
RFC 1382	<i>SNMP MIB Extension for the X.25 Packet Layer</i> D. Throop
RFC 1387	<i>RIP Version 2 Protocol Analysis</i> G. Malkin
RFC 1388	<i>RIP Version 2 Carrying Additional Information</i> G. Malkin
RFC 1389	<i>RIP Version 2 MIB Extensions</i> G. Malkin, F. Baker
RFC 1390	<i>Transmission of IP and ARP over FDDI Networks</i> D. Katz
RFC 1393	<i>Traceroute Using an IP Option</i> G. Malkin
RFC 1398	<i>Definitions of Managed Objects for the Ethernet-Like Interface Types</i> F. Kastenholz
RFC 1408	<i>Telnet Environment Option</i> D. Borman, Ed.
RFC 1413	<i>Identification Protocol</i> M. St. Johns
RFC 1416	<i>Telnet Authentication Option</i> D. Borman, ed.
RFC 1420	<i>SNMP over IPX</i> S. Bostock
RFC 1428	<i>Transition of Internet Mail from Just-Send-8 to 8bit-SMTP/MIME</i> G. Vaudreuil
RFC 1442	<i>Structure of Management Information for version 2 of the Simple Network Management Protocol (SNMPv2)</i> J. Case, K. McCloghrie, M. Rose, S. Waldbusser
RFC 1443	<i>Textual Conventions for version 2 of the Simple Network Management Protocol (SNMPv2)</i> J. Case, K. McCloghrie, M. Rose, S. Waldbusser
RFC 1445	<i>Administrative Model for version 2 of the Simple Network Management Protocol (SNMPv2)</i> J. Galvin, K. McCloghrie
RFC 1447	<i>Party MIB for version 2 of the Simple Network Management Protocol (SNMPv2)</i> K. McCloghrie, J. Galvin

RFC 1448	<i>Protocol Operations for version 2 of the Simple Network Management Protocol (SNMPv2)</i> J. Case, K. McCloghrie, M. Rose, S. Waldbusser
RFC 1464	<i>Using the Domain Name System to Store Arbitrary String Attributes</i> R. Rosenbaum
RFC 1469	<i>IP Multicast over Token-Ring Local Area Networks</i> T. Pusateri
RFC 1483	<i>Multiprotocol Encapsulation over ATM Adaptation Layer 5</i> Juha Heinanen
RFC 1497	<i>BOOTP Vendor Information Extensions</i> J. Reynolds
RFC 1514	<i>Host Resources MIB</i> P. Grillo, S. Waldbusser
RFC 1516	<i>Definitions of Managed Objects for IEEE 802.3 Repeater Devices</i> D. McMaster, K. McCloghrie
RFC 1521	<i>MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies</i> N. Borenstein, N. Freed
RFC 1533	<i>DHCP Options and BOOTP Vendor Extensions</i> S. Alexander, R. Droms
RFC 1534	<i>Interoperation Between DHCP and BOOTP</i> R. Droms
RFC 1535	<i>A Security Problem and Proposed Correction With Widely Deployed DNS Software</i> E. Gavron
RFC 1536	<i>Common DNS Implementation Errors and Suggested Fixes</i> A. Kumar, J. Postel, C. Neuman, P. Danzig, S. Miller
RFC 1537	<i>Common DNS Data File Configuration Errors</i> P. Beertema
RFC 1540	<i>Internet Official Protocol Standards</i> J. Postel
RFC 1541	<i>Dynamic Host Configuration Protocol</i> R. Droms
RFC 1542	<i>Clarifications and Extensions for the Bootstrap Protocol</i> W. Wimer
RFC 1571	<i>Telnet Environment Option Interoperability Issues</i> D. Borman
RFC 1572	<i>Telnet Environment Option</i> S. Alexander
RFC 1573	<i>Evolution of the Interfaces Group of MIB-II</i> K. McCloghrie, F. Kastenholz
RFC 1577	<i>Classical IP and ARP over ATM</i> M. Laubach
RFC 1583	<i>OSPF Version 2</i> J. Moy
RFC 1591	<i>Domain Name System Structure and Delegation</i> J. Postel
RFC 1592	<i>Simple Network Management Protocol Distributed Protocol Interface Version 2.0</i> B. Wijnen, G. Carpenter, K. Curran, A. Sehgal, G. Waters
RFC 1594	<i>FYI on Questions and Answers—Answers to Commonly Asked "New Internet User" Questions</i> A. Marine, J. Reynolds, G. Malkin
RFC 1644	<i>T/TCP — TCP Extensions for Transactions Functional Specification</i> R. Braden
RFC 1646	<i>TN3270 Extensions for LUname and Printer Selection</i> C. Graves, T. Butts, M. Angel
RFC 1647	<i>TN3270 Enhancements</i> B. Kelly

RFC 1652	<i>SMTP Service Extension for 8bit-MIMEtransport</i> J. Klensin, N. Freed, M. Rose, E. Stefferud, D. Crocker
RFC 1664	<i>Using the Internet DNS to Distribute RFC1327 Mail Address Mapping Tables</i> C. Allochio, A. Bonito, B. Cole, S. Giordano, R. Hagens
RFC 1693	<i>An Extension to TCP: Partial Order Service</i> T. Connolly, P. Amer, P. Conrad
RFC 1695	<i>Definitions of Managed Objects for ATM Management Version 8.0 using SMIv2</i> M. Ahmed, K. Tesink
RFC 1701	<i>Generic Routing Encapsulation (GRE)</i> S. Hanks, T. Li, D. Farinacci, P. Traina
RFC 1702	<i>Generic Routing Encapsulation over IPv4 networks</i> S. Hanks, T. Li, D. Farinacci, P. Traina
RFC 1706	<i>DNS NSAP Resource Records</i> B. Manning, R. Colella
RFC 1712	<i>DNS Encoding of Geographical Location</i> C. Farrell, M. Schulze, S. Pleitner D. Baldoni
RFC 1713	<i>Tools for DNS debugging</i> A. Romao
RFC 1723	<i>RIP Version 2—Carrying Additional Information</i> G. Malkin
RFC 1752	<i>The Recommendation for the IP Next Generation Protocol</i> S. Bradner, A. Mankin
RFC 1766	<i>Tags for the Identification of Languages</i> H. Alvestrand
RFC 1771	<i>A Border Gateway Protocol 4 (BGP-4)</i> Y. Rekhter, T. Li
RFC 1794	<i>DNS Support for Load Balancing</i> T. Brisco
RFC 1819	<i>Internet Stream Protocol Version 2 (ST2) Protocol Specification—Version ST2+</i> L. Delgrossi, L. Berger Eds.
RFC 1826	<i>IP Authentication Header</i> R. Atkinson
RFC 1828	<i>IP Authentication using Keyed MD5</i> P. Metzger, W. Simpson
RFC 1829	<i>The ESP DES-CBC Transform</i> P. Karn, P. Metzger, W. Simpson
RFC 1830	<i>SMTP Service Extensions for Transmission of Large and Binary MIME Messages</i> G. Vaudreuil
RFC 1831	<i>RPC: Remote Procedure Call Protocol Specification Version 2</i> R. Srinivasan
RFC 1832	<i>XDR: External Data Representation Standard</i> R. Srinivasan
RFC 1833	<i>Binding Protocols for ONC RPC Version 2</i> R. Srinivasan
RFC 1850	<i>OSPF Version 2 Management Information Base</i> F. Baker, R. Coltun
RFC 1854	<i>SMTP Service Extension for Command Pipelining</i> N. Freed
RFC 1869	<i>SMTP Service Extensions</i> J. Klensin, N. Freed, M. Rose, E. Stefferud, D. Crocker
RFC 1870	<i>SMTP Service Extension for Message Size Declaration</i> J. Klensin, N. Freed, K. Moore
RFC 1876	<i>A Means for Expressing Location Information in the Domain Name System</i> C. Davis, P. Vixie, T. Goodwin, I. Dickinson
RFC 1883	<i>Internet Protocol, Version 6 (IPv6) Specification</i> S. Deering, R. Hinden

RFC 1884	<i>IP Version 6 Addressing Architecture</i> R. Hinden, S. Deering, Eds.
RFC 1886	<i>DNS Extensions to support IP version 6</i> S. Thomson, C. Huitema
RFC 1888	<i>OSI NSAPs and IPv6</i> J. Bound, B. Carpenter, D. Harrington, J. Houldsworth, A. Lloyd
RFC 1891	<i>SMTP Service Extension for Delivery Status Notifications</i> K. Moore
RFC 1892	<i>The Multipart/Report Content Type for the Reporting of Mail System Administrative Messages</i> G. Vaudreuil
RFC 1894	<i>An Extensible Message Format for Delivery Status Notifications</i> K. Moore, G. Vaudreuil
RFC 1901	<i>Introduction to Community-based SNMPv2</i> J. Case, K. McCloghrie, M. Rose, S. Waldbusser
RFC 1902	<i>Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)</i> J. Case, K. McCloghrie, M. Rose, S. Waldbusser
RFC 1903	<i>Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)</i> J. Case, K. McCloghrie, M. Rose, S. Waldbusser
RFC 1904	<i>Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)</i> J. Case, K. McCloghrie, M. Rose, S. Waldbusser
RFC 1905	<i>Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)</i> J. Case, K. McCloghrie, M. Rose, S. Waldbusser
RFC 1906	<i>Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2)</i> J. Case, K. McCloghrie, M. Rose, S. Waldbusser
RFC 1907	<i>Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)</i> J. Case, K. McCloghrie, M. Rose, S. Waldbusser
RFC 1908	<i>Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework</i> J. Case, K. McCloghrie, M. Rose, S. Waldbusser
RFC 1912	<i>Common DNS Operational and Configuration Errors</i> D. Barr
RFC 1918	<i>Address Allocation for Private Internets</i> Y. Rekhter, B. Moskowitz, D. Karrenberg, G.J. de Groot, E. Lear
RFC 1928	<i>SOCKS Protocol Version 5</i> M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones
RFC 1930	<i>Guidelines for creation, selection, and registration of an Autonomous System (AS)</i> J. Hawkinson, T. Bates
RFC 1939	<i>Post Office Protocol-Version 3</i> J. Myers, M. Rose
RFC 1981	<i>Path MTU Discovery for IP version 6</i> J. McCann, S. Deering, J. Mogul
RFC 1982	<i>Serial Number Arithmetic</i> R. Elz, R. Bush
RFC 1985	<i>SMTP Service Extension for Remote Message Queue Starting</i> J. De Winter
RFC 1995	<i>Incremental Zone Transfer in DNS</i> M. Ohta
RFC 1996	<i>A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY)</i> P. Vixie

RFC 2010	<i>Operational Criteria for Root Name Servers</i> B. Manning, P. Vixie
RFC 2011	<i>SNMPv2 Management Information Base for the Internet Protocol using SMIv2</i> K. McCloaghrie, Ed.
RFC 2012	<i>SNMPv2 Management Information Base for the Transmission Control Protocol using SMIv2</i> K. McCloaghrie, Ed.
RFC 2013	<i>SNMPv2 Management Information Base for the User Datagram Protocol using SMIv2</i> K. McCloaghrie, Ed.
RFC 2018	<i>TCP Selective Acknowledgement Options</i> M. Mathis, J. Mahdavi, S. Floyd, A. Romanow
RFC 2026	<i>The Internet Standards Process — Revision 3</i> S. Bradner
RFC 2030	<i>Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI</i> D. Mills
RFC 2033	<i>Local Mail Transfer Protocol</i> J. Myers
RFC 2034	<i>SMTP Service Extension for Returning Enhanced Error Codes</i> N. Freed
RFC 2040	<i>The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms</i> R. Baldwin, R. Rivest
RFC 2045	<i>Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies</i> N. Freed, N. Borenstein
RFC 2052	<i>A DNS RR for specifying the location of services (DNS SRV)</i> A. Gulbrandsen, P. Vixie
RFC 2065	<i>Domain Name System Security Extensions</i> D. Eastlake 3rd, C. Kaufman
RFC 2066	<i>TELNET CHARSET Option</i> R. Gellens
RFC 2080	<i>RIPng for IPv6</i> G. Malkin, R. Minnear
RFC 2096	<i>IP Forwarding Table MIB</i> F. Baker
RFC 2104	<i>HMAC: Keyed-Hashing for Message Authentication</i> H. Krawczyk, M. Bellare, R. Canetti
RFC 2119	<i>Keywords for use in RFCs to Indicate Requirement Levels</i> S. Bradner
RFC 2132	<i>DHCP Options and BOOTP Vendor Extensions</i> S. Alexander, R. Droms
RFC 2133	<i>Basic Socket Interface Extensions for IPv6</i> R. Gilligan, S. Thomson, J. Bound, W. Stevens
RFC 2136	<i>Dynamic Updates in the Domain Name System (DNS UPDATE)</i> P. Vixie, Ed., S. Thomson, Y. Rekhter, J. Bound
RFC 2137	<i>Secure Domain Name System Dynamic Update</i> D. Eastlake 3rd
RFC 2163	<i>Using the Internet DNS to Distribute MIXER Conformant Global Address Mapping (MCGAM)</i> C. Allocchio
RFC 2168	<i>Resolution of Uniform Resource Identifiers using the Domain Name System</i> R. Daniel, M. Mealling
RFC 2178	<i>OSPF Version 2</i> J. Moy
RFC 2181	<i>Clarifications to the DNS Specification</i> R. Elz, R. Bush

RFC 2205	<i>Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification</i> R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, S. Jamin
RFC 2210	<i>The Use of RSVP with IETF Integrated Services</i> J. Wroclawski
RFC 2211	<i>Specification of the Controlled-Load Network Element Service</i> J. Wroclawski
RFC 2212	<i>Specification of Guaranteed Quality of Service</i> S. Shenker, C. Partridge, R. Guerin
RFC 2215	<i>General Characterization Parameters for Integrated Service Network Elements</i> S. Shenker, J. Wroclawski
RFC 2217	<i>Telnet Com Port Control Option</i> G. Clarke
RFC 2219	<i>Use of DNS Aliases for Network Services</i> M. Hamilton, R. Wright
RFC 2228	<i>FTP Security Extensions</i> M. Horowitz, S. Lunt
RFC 2230	<i>Key Exchange Delegation Record for the DNS</i> R. Atkinson
RFC 2233	<i>The Interfaces Group MIB using SMIv2</i> K. McCloghrie, F. Kastenholz
RFC 2240	<i>A Legal Basis for Domain Name Allocation</i> O. Vaughn
RFC 2246	<i>The TLS Protocol Version 1.0</i> T. Dierks, C. Allen
RFC 2251	<i>Lightweight Directory Access Protocol (v3)</i> M. Wahl, T. Howes, S. Kille
RFC 2253	<i>Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names</i> M. Wahl, S. Kille, T. Howes
RFC 2254	<i>The String Representation of LDAP Search Filters</i> T. Howes
RFC 2261	<i>An Architecture for Describing SNMP Management Frameworks</i> D. Harrington, R. Presuhn, B. Wijnen
RFC 2262	<i>Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)</i> J. Case, D. Harrington, R. Presuhn, B. Wijnen
RFC 2271	<i>An Architecture for Describing SNMP Management Frameworks</i> D. Harrington, R. Presuhn, B. Wijnen
RFC 2273	<i>SNMPv3 Applications</i> D. Levi, P. Meyer, B. Stewart
RFC 2274	<i>User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)</i> U. Blumenthal, B. Wijnen
RFC 2275	<i>View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)</i> B. Wijnen, R. Presuhn, K. McCloghrie
RFC 2279	<i>UTF-8, a transformation format of ISO 10646</i> F. Yergeau
RFC 2292	<i>Advanced Sockets API for IPv6</i> W. Stevens, M. Thomas
RFC 2308	<i>Negative Caching of DNS Queries (DNS NCACHE)</i> M. Andrews
RFC 2317	<i>Classless IN-ADDR.ARPA delegation</i> H. Eidnes, G. de Groot, P. Vixie
RFC 2320	<i>Definitions of Managed Objects for Classical IP and ARP Over ATM Using SMIv2 (IPOA-MIB)</i> M. Greene, J. Luciani, K. White, T. Kuo
RFC 2328	<i>OSPF Version 2</i> J. Moy
RFC 2345	<i>Domain Names and Company Name Retrieval</i> J. Klensin, T. Wolf, G. Oglesby

RFC 2352	<i>A Convention for Using Legal Names as Domain Names</i> O. Vaughn
RFC 2355	<i>TN3270 Enhancements</i> B. Kelly
RFC 2358	<i>Definitions of Managed Objects for the Ethernet-like Interface Types</i> J. Flick, J. Johnson
RFC 2373	<i>IP Version 6 Addressing Architecture</i> R. Hinden, S. Deering
RFC 2374	<i>An IPv6 Aggregatable Global Unicast Address Format</i> R. Hinden, M. O'Dell, S. Deering
RFC 2375	<i>IPv6 Multicast Address Assignments</i> R. Hinden, S. Deering
RFC 2385	<i>Protection of BGP Sessions via the TCP MD5 Signature Option</i> A. Hefferman
RFC 2389	<i>Feature negotiation mechanism for the File Transfer Protocol</i> P. Hethmon, R. Elz
RFC 2401	<i>Security Architecture for Internet Protocol</i> S. Kent, R. Atkinson
RFC 2402	<i>IP Authentication Header</i> S. Kent, R. Atkinson
RFC 2403	<i>The Use of HMAC-MD5-96 within ESP and AH</i> C. Madson, R. Glenn
RFC 2404	<i>The Use of HMAC-SHA-1-96 within ESP and AH</i> C. Madson, R. Glenn
RFC 2405	<i>The ESP DES-CBC Cipher Algorithm With Explicit IV</i> C. Madson, N. Doraswamy
RFC 2406	<i>IP Encapsulating Security Payload (ESP)</i> S. Kent, R. Atkinson
RFC 2407	<i>The Internet IP Security Domain of Interpretation for ISAKMP</i> D. Piper
RFC 2408	<i>Internet Security Association and Key Management Protocol (ISAKMP)</i> D. Maughan, M. Schertler, M. Schneider, J. Turner
RFC 2409	<i>The Internet Key Exchange (IKE)</i> D. Harkins, D. Carrel
RFC 2410	<i>The NULL Encryption Algorithm and Its Use With IPsec</i> R. Glenn, S. Kent,
RFC 2428	<i>FTP Extensions for IPv6 and NATs</i> M. Allman, S. Ostermann, C. Metz
RFC 2445	<i>Internet Calendaring and Scheduling Core Object Specification (iCalendar)</i> F. Dawson, D. Stenerson
RFC 2459	<i>Internet X.509 Public Key Infrastructure Certificate and CRL Profile</i> R. Housley, W. Ford, W. Polk, D. Solo
RFC 2460	<i>Internet Protocol, Version 6 (IPv6) Specification</i> S. Deering, R. Hinden
RFC 2461	<i>Neighbor Discovery for IP Version 6 (IPv6)</i> T. Narten, E. Nordmark, W. Simpson
RFC 2462	<i>IPv6 Stateless Address Autoconfiguration</i> S. Thomson, T. Narten
RFC 2463	<i>Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification</i> A. Conta, S. Deering
RFC 2464	<i>Transmission of IPv6 Packets over Ethernet Networks</i> M. Crawford
RFC 2466	<i>Management Information Base for IP Version 6: ICMPv6 Group D.</i> Haskin, S. Onishi
RFC 2476	<i>Message Submission</i> R. Gellens, J. Klensin

RFC 2487	<i>SMTP Service Extension for Secure SMTP over TLS</i> P. Hoffman
RFC 2505	<i>Anti-Spam Recommendations for SMTP MTAs</i> G. Lindberg
RFC 2523	<i>Photuris: Extended Schemes and Attributes</i> P. Karn, W. Simpson
RFC 2535	<i>Domain Name System Security Extensions</i> D. Eastlake 3rd
RFC 2538	<i>Storing Certificates in the Domain Name System (DNS)</i> D. Eastlake 3rd, O. Gudmundsson
RFC 2539	<i>Storage of Diffie-Hellman Keys in the Domain Name System (DNS)</i> D. Eastlake 3rd
RFC 2540	<i>Detached Domain Name System (DNS) Information</i> D. Eastlake 3rd
RFC 2554	<i>SMTP Service Extension for Authentication</i> J. Myers
RFC 2570	<i>Introduction to Version 3 of the Internet-standard Network Management Framework</i> J. Case, R. Mundy, D. Partain, B. Stewart
RFC 2571	<i>An Architecture for Describing SNMP Management Frameworks</i> B. Wijnen, D. Harrington, R. Presuhn
RFC 2572	<i>Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)</i> J. Case, D. Harrington, R. Presuhn, B. Wijnen
RFC 2573	<i>SNMP Applications</i> D. Levi, P. Meyer, B. Stewart
RFC 2574	<i>User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)</i> U. Blumenthal, B. Wijnen
RFC 2575	<i>View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)</i> B. Wijnen, R. Presuhn, K. McCloghrie
RFC 2576	<i>Co-Existence between Version 1, Version 2, and Version 3 of the Internet-standard Network Management Framework</i> R. Frye, D. Levi, S. Routhier, B. Wijnen
RFC 2578	<i>Structure of Management Information Version 2 (SMIv2)</i> K. McCloghrie, D. Perkins, J. Schoenwaelder
RFC 2579	<i>Textual Conventions for SMIv2</i> K. McCloghrie, D. Perkins, J. Schoenwaelder
RFC 2580	<i>Conformance Statements for SMIv2</i> K. McCloghrie, D. Perkins, J. Schoenwaelder
RFC 2581	<i>TCP Congestion Control</i> M. Allman, V. Paxson, W. Stevens
RFC 2583	<i>Guidelines for Next Hop Client (NHC) Developers</i> R. Carlson, L. Winkler
RFC 2591	<i>Definitions of Managed Objects for Scheduling Management Operations</i> D. Levi, J. Schoenwaelder
RFC 2625	<i>IP and ARP over Fibre Channel</i> M. Rajagopal, R. Bhagwat, W. Rickard
RFC 2635	<i>Don't SPEW A Set of Guidelines for Mass Unsolicited Mailings and Postings (spam*)</i> S. Hambridge, A. Lunde
RFC 2637	<i>Point-to-Point Tunneling Protocol</i> K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, G. Zorn
RFC 2640	<i>Internationalization of the File Transfer Protocol</i> B. Curtin

RFC 2665	<i>Definitions of Managed Objects for the Ethernet-like Interface Types</i> J. Flick, J. Johnson
RFC 2671	<i>Extension Mechanisms for DNS (EDNS0)</i> P. Vixie
RFC 2672	<i>Non-Terminal DNS Name Redirection</i> M. Crawford
RFC 2675	<i>IPv6 Jumbograms</i> D. Borman, S. Deering, R. Hinden
RFC 2710	<i>Multicast Listener Discovery (MLD) for IPv6</i> S. Deering, W. Fenner, B. Haberman
RFC 2711	<i>IPv6 Router Alert Option</i> C. Partridge, A. Jackson
RFC 2740	<i>OSPF for IPv6</i> R. Coltun, D. Ferguson, J. Moy
RFC 2753	<i>A Framework for Policy-based Admission Control</i> R. Yavatkar, D. Pendarakis, R. Guerin
RFC 2782	<i>A DNS RR for specifying the location of services (DNS SRV)</i> A. Gubrandsen, P. Vixie, L. Esibov
RFC 2821	<i>Simple Mail Transfer Protocol</i> J. Klensin, Ed.
RFC 2822	<i>Internet Message Format</i> P. Resnick, Ed.
RFC 2840	<i>TELNET KERMIT OPTION</i> J. Altman, F. da Cruz
RFC 2845	<i>Secret Key Transaction Authentication for DNS (TSIG)</i> P. Vixie, O. Gudmundsson, D. Eastlake 3rd, B. Wellington
RFC 2851	<i>Textual Conventions for Internet Network Addresses</i> M. Daniele, B. Haberman, S. Routhier, J. Schoenwaelder
RFC 2852	<i>Deliver By SMTP Service Extension</i> D. Newman
RFC 2874	<i>DNS Extensions to Support IPv6 Address Aggregation and Renumbering</i> M. Crawford, C. Huitema
RFC 2915	<i>The Naming Authority Pointer (NAPTR) DNS Resource Record</i> M. Mealling, R. Daniel
RFC 2920	<i>SMTP Service Extension for Command Pipelining</i> N. Freed
RFC 2930	<i>Secret Key Establishment for DNS (TKEY RR)</i> D. Eastlake, 3rd
RFC 2941	<i>Telnet Authentication Option</i> T. Ts'o, ed., J. Altman
RFC 2942	<i>Telnet Authentication: Kerberos Version 5</i> T. Ts'o
RFC 2946	<i>Telnet Data Encryption Option</i> T. Ts'o
RFC 2952	<i>Telnet Encryption: DES 64 bit Cipher Feedback</i> T. Ts'o
RFC 2953	<i>Telnet Encryption: DES 64 bit Output Feedback</i> T. Ts'o
RFC 2992	<i>Analysis of an Equal-Cost Multi-Path Algorithm</i> C. Hopps
RFC 3019	<i>IP Version 6 Management Information Base for The Multicast Listener Discovery Protocol</i> B. Haberman, R. Worzella
RFC 3060	<i>Policy Core Information Model—Version 1 Specification</i> B. Moore, E. Ellesson, J. Strassner, A. Westerinen
RFC 3152	<i>Delegation of IP6.ARPA</i> R. Bush
RFC 3164	<i>The BSD Syslog Protocol</i> C. Lonvick
RFC 3291	<i>Textual Conventions for Internet Network Addresses</i> M. Daniele, B. Haberman, S. Routhier, J. Schoenwaelder

	RFC 3363	<i>Representing Internet Protocol version 6 (IPv6) Addresses in the Domain Name System</i> R. Bush, A. Durand, B. Fink, O. Gudmundsson, T. Hain
	RFC 3376	<i>Internet Group Management Protocol, Version 3</i> B. Cain, S. Deering, I. Kouvelas, B. Fenner, A. Thyagarajan
	RFC 3390	<i>Increasing TCP's Initial Window</i> M. Allman, S. Floyd, C. Partridge
	RFC 3410	<i>Introduction and Applicability Statements for Internet-Standard Management Framework</i> J. Case, R. Mundy, D. Partain, B. Stewart
	RFC 3411	<i>An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks</i> D. Harrington, R. Presuhn, B. Wijnen
	RFC 3412	<i>Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)</i> J. Case, D. Harrington, R. Presuhn, B. Wijnen
	RFC 3413	<i>Simple Network Management Protocol (SNMP) Applications</i> D. Levi, P. Meyer, B. Stewart
	RFC 3414	<i>User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)</i> U. Blumenthal, B. Wijnen
	RFC 3415	<i>View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)</i> B. Wijnen, R. Presuhn, K. McCloghrie
	RFC 3419	<i>Textual Conventions for Transport Addresses</i> M. Daniele, J. Schoenwaelder
	RFC 3484	<i>Default Address Selection for Internet Protocol version 6 (IPv6)</i> R. Draves
	RFC 3493	<i>Basic Socket Interface Extensions for IPv6</i> R. Gilligan, S. Thomson, J. Bound, J. McCann, W. Stevens
	RFC 3513	<i>Internet Protocol Version 6 (IPv6) Addressing Architecture</i> R. Hinden, S. Deering
	RFC 3526	<i>More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)</i> T. Kivinen, M. Kojo
	RFC 3542	<i>Advanced Sockets Application Programming Interface (API) for IPv6</i> W. Richard Stevens, M. Thomas, E. Nordmark, T. Jinmei
	RFC 3569	<i>An Overview of Source-Specific Multicast (SSM)</i> S. Bhattacharyya, Ed.
	RFC 3584	<i>Coexistence between Version 1, Version 2, and Version 3 of the Internet-standard Network Management Framework</i> R. Frye, D. Levi, S. Routhier, B. Wijnen
	RFC 3602	<i>The AES-CBC Cipher Algorithm and Its Use with IPsec</i> S. Frankel, R. Glenn, S. Kelly
	RFC 3629	<i>UTF-8, a transformation format of ISO 10646</i> R. Kermode, C. Vicisano
	RFC 3658	<i>Delegation Signer (DS) Resource Record (RR)</i> O. Gudmundsson
	RFC 3678	<i>Socket Interface Extensions for Multicast Source Filters</i> D. Thaler, B. Fenner, B. Quinn
	RFC 3715	<i>IPsec-Network Address Translation (NAT) Compatibility Requirements</i> B. Aboba, W. Dixon

RFC 3810	<i>Multicast Listener Discovery Version 2 (MLDv2) for IPv6</i> R. Vida, Ed., L. Costa, Ed.
RFC 3947	<i>Negotiation of NAT-Traversal in the IKE</i> T. Kivinen, B. Swander, A. Huttunen, V. Volpe
RFC 3948	<i>UDP Encapsulation of IPsec ESP Packets</i> A. Huttunen, B. Swander, V. Volpe, L. DiBurro, M. Stenberg
RFC 4007	<i>IPv6 Scoped Address Architecture</i> S. Deering, B. Haberman, T. Jinmei, E. Nordmark, B. Zill
RFC 4217	<i>Securing FTP with TLS</i> P. Ford-Hutchinson

Internet drafts

Internet drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Other groups may also distribute working documents as Internet drafts. You can see Internet drafts at <http://www.ietf.org/ID.html>.

Several areas of IPv6 implementation include elements of the following Internet drafts and are subject to change during the RFC review process.

Draft Title and Author

draft-bivens-sasp-02

Server/Application State Protocol v1 A. Bivens

draft-ietf-ipngwg-icmp-v3-07

Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification A. Conta, S. Deering

draft-ietf-ipsec-esp-v3-10

IP Encapsulating Security Payload (ESP) S. Kent

draft-ietf-ipsec-rfc2402bis-11

IP Authentication Header S. Kent

draft-ietf-ipsec-rfc2401bis-06

Security Architecture for the Internet Protocol S. Kent, K. Seo

draft-ietf-ospf-ospfv3-auth-07

Authentication/Confidentiality for OSPFv3 M. Gupta, N. Melam

Appendix F. Information APARs and technotes

This appendix lists information APARs for IP and SNA documents.

Note:

1. Information APARs contain updates to previous editions of the documents listed in Table 161 and Table 162 on page 736. Documents updated for V1R9 are complete except for the updates contained in the information APARs that might be issued after V1R9 documents went to press.
2. Information APARs are predefined for z/OS V1R9 Communications Server and might not contain updates.
3. Information APARs for z/OS documents are in the document called *z/OS and z/OS.e DOC APAR and PTF ++HOLD Documentation*, which can be found at http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/ZIDOCMST/CCONTENTS.

Information APARs for IP documents

Table 161 lists information APARs for V1R6 IP documents. For releases V1R7 and later, updates are available as technotes, which can be found at <http://www.ibm.com/support/docview.wss?uid=swg21178966>.

Table 161. IP information APARs for z/OS Communications Server

Title	Information APAR for V1R6
New Function Summary (both IP and SNA)	II13824
Quick Reference (both IP and SNA)	II13831
IP and SNA Codes	II13842
IP Sockets API Guide	II13844
IP Configuration Guide	II13826
IP Configuration Reference	II13827
IP Diagnosis	II13836
IP Messages Volume 1	II13838
IP Messages Volume 2	II13839
IP Messages Volume 3	II13840
IP Messages Volume 4	II13841
IPv6 Network and Application Design Guide	II13825
IP Programmer's Guide and Reference	II13843
IP User's Guide and Commands	II13832
IP System Administrator's Commands	II13833

Information APARs for SNA documents

Table 162 lists information APARs for V1R6 SNA documents. For releases V1R7 and later, updates are available as technotes, which can be found at <http://www.ibm.com/support/docview.wss?uid=swg21178966>.

Table 162. SNA information APARs for z/OS Communications Server

Title	Information APAR for V1R6
New Function Summary (both IP and SNA)	II13824
Quick Reference (both IP and SNA)	II13831
IP and SNA Codes	II13842
SNA Customization	II13857
SNA Diagnosis, Vol. 1: Techniques and Procedures	II13852
SNA Diagnosis, Vol. 2: FFST Dumps and the VIT	II13853
SNA Messages	II13854
SNA Network Implementation Guide	II13849
SNA Operation	II13851
SNA Programming	II13858
SNA Resource Definition Reference	II13850
SNA Data Areas Volume 1	II13855
SNA Data Areas Volume 2	II13856

Other information APARs

Table 163 lists information APARs not related to documents.

Table 163. Non-document information APARs

Content	Number
Index to APARs that list recommended VTAM maintenance	II11220
Index to APARs that list trace and dump requests for VTAM problems	II13202
Index of Communication Server IP information APARs	II12028
Collecting TCPIP CTRACES	II12014
CSM for VTAM	II13442
CSM for TCP/IP	II13951
DLUR/DLUS	II12986, II13456, and II13783
Documentation required for FTP server problems	II12925
Documentation required for OSA/2, OSA Express and OSA QDIO	II13016
DNS — common problems and solutions	II13453
Enterprise Extender	II12223
FTP client and FTP server TLS support	II13516
FTP problems	II12079
FTPing doc to z/OS Ssupport	II12030
Generic resources	II10986
HPR	II10953

Table 163. Non-document information APARs (continued)

Content	Number
iQDIO	II13142
LPR problems	II12022
MNPS	II10370
MPC and CTC	II01501
NCPROUTE problems	II12025
OMPROUTE	II12026
PASCAL API	II11814
Performance	II11710 II11711 II11712
Resolver	II13398 II13399 II13452
Socket API	II11996 II12020
SMTP problems	II12023
SNMP	II13477 II13478
SYSLOGD howto	II12021
TCPIP connection states	II12449
TN3270E Telnet server	II11574 II13135
TN3270E Telnet server SSL common problems	II13369

Appendix G. Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at:

www.ibm.com/servers/eserver/zseries/zos/bkserv/

Notices

IBM may not offer all of the products, services, or features discussed in this document. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

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

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

Site Counsel
IBM Corporation
P.O. Box 12195
3039 Cornwallis Road
Research Triangle Park, North Carolina 27709-2195
U.S.A

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

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application

programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

IBM is required to include the following statements in order to distribute portions of this document and the software described herein to which contributions have been made by The University of California. Portions herein © Copyright 1979, 1980, 1983, 1986, Regents of the University of California. Reproduced by permission. Portions herein were developed at the Electrical Engineering and Computer Sciences Department at the Berkeley campus of the University of California under the auspices of the Regents of the University of California.

Portions of this publication relating to RPC are Copyright © Sun Microsystems, Inc., 1988, 1989.

Some portions of this publication relating to X Window System** are Copyright © 1987, 1988 by Digital Equipment Corporation, Maynard, Massachusetts, and the Massachusetts Institute Of Technology, Cambridge, Massachusetts. All Rights Reserved.

Some portions of this publication relating to X Window System are Copyright © 1986, 1987, 1988 by Hewlett-Packard Corporation.

Permission to use, copy, modify, and distribute the M.I.T., Digital Equipment Corporation, and Hewlett-Packard Corporation portions of this software and its documentation for any purpose without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the names of M.I.T., Digital, and Hewlett-Packard not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T., Digital, and Hewlett-Packard make no representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Copyright © 1983, 1995-1997 Eric P. Allman

Copyright © 1988, 1993 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
This product includes software developed by the University of California, Berkeley and its contributors.
4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software program contains code, and/or derivatives or modifications of code originating from the software program "Popper." Popper is Copyright ©1989-1991 The Regents of the University of California, All Rights Reserved. Popper was created by Austin Shelton, Information Systems and Technology, University of California, Berkeley.

Permission from the Regents of the University of California to use, copy, modify, and distribute the "Popper" software contained herein for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies. HOWEVER, ADDITIONAL PERMISSIONS MAY BE NECESSARY FROM OTHER PERSONS OR ENTITIES, TO USE DERIVATIVES OR MODIFICATIONS OF POPPER.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THE POPPER SOFTWARE, OR ITS DERIVATIVES OR MODIFICATIONS, AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE POPPER SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Copyright © 1983 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by the University of California, Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Copyright © 1991, 1993 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
This product includes software developed by the University of California, Berkeley and its contributors.
4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright © 1990 by the Massachusetts Institute of Technology

Export of this software from the United States of America may require a specific license from the United States Government. It is the responsibility of any person or organization contemplating export to obtain such a license before exporting.

WITHIN THAT CONSTRAINT, permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Furthermore

if you modify this software you must label your software as modified software and not distribute it in such a fashion that it might be confused with the original M.I.T. software. M.I.T. makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Copyright © 1998 by the FundsXpress, INC. All rights reserved.

Export of this software from the United States of America may require a specific license from the United States Government. It is the responsibility of any person or organization contemplating export to obtain such a license before exporting.

WITHIN THAT CONSTRAINT, permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of FundsXpress not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. FundsXpress makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Copyright © 1999, 2000 Internet Software Consortium.

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND INTERNET SOFTWARE CONSORTIUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INTERNET SOFTWARE CONSORTIUM BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Copyright © 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved.

This package is an SSL implementation written by Eric Young (eay@cryptsoft.com). The implementation was written so as to conform with Netscape's SSL.

This library is free for commercial and non-commercial use as long as the following conditions are adhered to. The following conditions apply to all code found in this distribution, be it the RC4, RSA, lhash, DES, etc., code; not just the SSL code. The SSL documentation included with this distribution is covered by the same copyright terms except that the holder is Tim Hudson (tjh@cryptsoft.com).

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed. If this package is used in a product, Eric Young should be

given attribution as the author of the parts of the library used. This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement: "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)". The word 'cryptographic' can be left out if the routines from the library being used are not cryptographic related.
4. If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include acknowledgement:
"This product includes software written by Tim Hudson (tjh@cryptsoft.com)"

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The license and distribution terms for any publicly available version or derivative of this code cannot be changed. i.e. this code cannot simply be copied and put under another distribution license [including the GNU Public License.]

This product includes cryptographic software written by Eric Young.

Copyright © 1999, 2000 Internet Software Consortium.

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND INTERNET SOFTWARE CONSORTIUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INTERNET SOFTWARE CONSORTIUM BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

The IBM implementation of the Resource Reservation Setup Protocol API (RAPI) described in this document is derived from the Resource Reservation Setup Protocol API (RAPI) specification, The Open Group Document Number C809, ISBN 185912264, published by The Open Group, December 1998.

The specification document is copyrighted by The Open Group. See The Open Group Web site at <http://www.opengroup.org/publications/catalog/c809.htm> for the source of the specification. See the "Summary of Changes" in the *z/OS Communications Server V1R7: IP Programmer's Guide and Reference* for details about IBM use of this information.

Copyright © 2004 IBM Corporation and its licensors, including Sendmail, Inc., and the Regents of the University of California. All rights reserved.

Copyright © 1999,2000,2001 Compaq Computer Corporation

Copyright © 1999,2000,2001 Hewlett-Packard Company

Copyright © 1999,2000,2001 IBM Corporation

Copyright © 1999,2000,2001 Hummingbird Communications Ltd.

Copyright © 1999,2000,2001 Silicon Graphics, Inc.

Copyright © 1999,2000,2001 Sun Microsystems, Inc.

Copyright © 1999,2000,2001 The Open Group

All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

X Window System is a trademark of The Open Group.

If you are viewing this information softcopy, photographs and color illustrations may not appear.

You can obtain softcopy from the z/OS Collection (SK3T-4269), which contains BookManager and PDF formats.

Programming interface information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/OS Communications Server.

Trademarks

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

Advanced Peer-to-Peer Networking	NetView
AFS	Network Station
AD/Cycle	Nways
AIX	OfficeVision
AIX/ESA	OS/2
AnyNet	OS/390
APL2	Parallel Sysplex
AS/400	PROFS
BookManager	pSeries
C/370	RACF
CICS	Redbooks
CICS/ESA	RETAIN
C Set ++	REXX
Common User Access	RISC System/6000
CUA	RMF
DB2	RS/6000
DFSMS	S/370
DFSMSdfp	S/390
DFSMShsm	S/390 Parallel Enterprise Server
DPI	SAA
ESCON	SecureWay
eServer	SET
ES/9000	SiteCheck
FFST	SP
FICON	System/360
First Failure Support Technology	System/370
GDDM	System/390
IBM	System z
ibm.com	System z9
IBMLink	Tivoli
IMS	Tivoli Enterprise Console
IMS/ESA	VM/ESA
HiperSockets	VSE/ESA
Language Environment	VTAM
Micro Channel	WebSphere
Multiprise	z9
MVS	z/Architecture
MVS/DFP	z/OS
MVS/ESA	z/VM
MVS/SP	zSeries
	400

The following terms are trademarks of other companies:

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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

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

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

PostScript is a registered trademark of Adobe Systems Incorporated in the United States, other countries, or both.

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

Bibliography

z/OS Communications Server information

This section contains descriptions of the documents in the z/OS Communications Server library.

z/OS Communications Server documentation is available:

- Online at the z/OS Internet Library web page at <http://www.ibm.com/servers/eserver/zseries/zos/bkserv>
- In softcopy on CD-ROM collections. See “Softcopy information” on page xxx.

z/OS Communications Server library

z/OS Communications Server documents are available on the CD-ROM accompanying z/OS (SK3T-4269 or SK3T-4307). Unlicensed documents can be viewed at the z/OS Internet library site.

Updates to documents are available on RETAIN® and in information APARs (info APARs). See Appendix F, “Information APARs and technotes,” on page 735 for a list of the documents and the info APARs associated with them.

Info APARs for z/OS documents are in the document called *z/OS and z/OS.e DOC APAR and PTF ++HOLD Documentation* which can be found at http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/ZIDOCMST/CCONTENTS.

Planning

Title	Number	Description
<i>z/OS Communications Server: New Function Summary</i>	GC31-8771	This document is intended to help you plan for new IP for SNA function, whether you are migrating from a previous version or installing z/OS for the first time. It summarizes what is new in the release and identifies the suggested and required modifications needed to use the enhanced functions.
<i>z/OS Communications Server: IPv6 Network and Application Design Guide</i>	SC31-8885	This document is a high-level introduction to IPv6. It describes concepts of z/OS Communications Server's support of IPv6, coexistence with IPv4, and migration issues.

Resource definition, configuration, and tuning

Title	Number	Description
<i>z/OS Communications Server: IP Configuration Guide</i>	SC31-8775	This document describes the major concepts involved in understanding and configuring an IP network. Familiarity with the z/OS operating system, IP protocols, z/OS UNIX System Services, and IBM Time Sharing Option (TSO) is recommended. Use this document in conjunction with the <i>z/OS Communications Server: IP Configuration Reference</i> .

Title	Number	Description
<i>z/OS Communications Server: IP Configuration Reference</i>	SC31-8776	This document presents information for people who want to administer and maintain IP. Use this document in conjunction with the <i>z/OS Communications Server: IP Configuration Guide</i> . The information in this document includes: • TCP/IP configuration data sets • Configuration statements • Translation tables • SMF records • Protocol number and port assignments
<i>z/OS Communications Server: SNA Network Implementation Guide</i>	SC31-8777	This document presents the major concepts involved in implementing an SNA network. Use this document in conjunction with the <i>z/OS Communications Server: SNA Resource Definition Reference</i> .
<i>z/OS Communications Server: SNA Resource Definition Reference</i>	SC31-8778	This document describes each SNA definition statement, start option, and macroinstruction for user tables. It also describes NCP definition statements that affect SNA. Use this document in conjunction with the <i>z/OS Communications Server: SNA Network Implementation Guide</i> .
<i>z/OS Communications Server: SNA Resource Definition Samples</i>	SC31-8836	This document contains sample definitions to help you implement SNA functions in your networks, and includes sample major node definitions.
<i>z/OS Communications Server: IP Network Print Facility</i>	SC31-8833	This document is for system programmers and network administrators who need to prepare their network to route SNA, JES2, or JES3 printer output to remote printers using TCP/IP Services.

Operation

Title	Number	Description
<i>z/OS Communications Server: IP User's Guide and Commands</i>	SC31-8780	This document describes how to use TCP/IP applications. It contains requests that allow a user to log on to a remote host using Telnet, transfer data sets using FTP, send and receive electronic mail, print on remote printers, and authenticate network users.
<i>z/OS Communications Server: IP System Administrator's Commands</i>	SC31-8781	This document describes the functions and commands helpful in configuring or monitoring your system. It contains system administrator's commands, such as TSO NETSTAT, PING, TRACERTE and their UNIX counterparts. It also includes TSO and MVS commands commonly used during the IP configuration process.
<i>z/OS Communications Server: SNA Operation</i>	SC31-8779	This document serves as a reference for programmers and operators requiring detailed information about specific operator commands.
<i>z/OS Communications Server: Quick Reference</i>	SX75-0124	This document contains essential information about SNA and IP commands.

Customization

Title	Number	Description
<i>z/OS Communications Server: SNA Customization</i>	SC31-6854	This document enables you to customize SNA, and includes the following: • Communication network management (CNM) routing table • Logon-interpret routine requirements • Logon manager installation-wide exit routine for the CLU search exit • TSO/SNA installation-wide exit routines • SNA installation-wide exit routines

Writing application programs

Title	Number	Description
<i>z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference</i>	SC31-8788	This document describes the syntax and semantics of program source code necessary to write your own application programming interface (API) into TCP/IP. You can use this interface as the communication base for writing your own client or server application. You can also use this document to adapt your existing applications to communicate with each other using sockets over TCP/IP.
<i>z/OS Communications Server: IP CICS Sockets Guide</i>	SC31-8807	This document is for programmers who want to set up, write application programs for, and diagnose problems with the socket interface for CICS using z/OS TCP/IP.
<i>z/OS Communications Server: IP IMS Sockets Guide</i>	SC31-8830	This document is for programmers who want application programs that use the IMS TCP/IP application development services provided by IBM's TCP/IP Services.
<i>z/OS Communications Server: IP Programmer's Guide and Reference</i>	SC31-8787	This document describes the syntax and semantics of a set of high-level application functions that you can use to program your own applications in a TCP/IP environment. These functions provide support for application facilities, such as user authentication, distributed databases, distributed processing, network management, and device sharing. Familiarity with the z/OS operating system, TCP/IP protocols, and IBM Time Sharing Option (TSO) is recommended.
<i>z/OS Communications Server: SNA Programming</i>	SC31-8829	This document describes how to use SNA macroinstructions to send data to and receive data from (1) a terminal in either the same or a different domain, or (2) another application program in either the same or a different domain.
<i>z/OS Communications Server: SNA Programmer's LU 6.2 Guide</i>	SC31-8811	This document describes how to use the SNA LU 6.2 application programming interface for host application programs. This document applies to programs that use only LU 6.2 sessions or that use LU 6.2 sessions along with other session types. (Only LU 6.2 sessions are covered in this document.)
<i>z/OS Communications Server: SNA Programmer's LU 6.2 Reference</i>	SC31-8810	This document provides reference material for the SNA LU 6.2 programming interface for host application programs.
<i>z/OS Communications Server: CSM Guide</i>	SC31-8808	This document describes how applications use the communications storage manager.

Title	Number	Description
<i>z/OS Communications Server: CMIP Services and Topology Agent Guide</i>	SC31-8828	This document describes the Common Management Information Protocol (CMIP) programming interface for application programmers to use in coding CMIP application programs. The document provides guide and reference information about CMIP services and the SNA topology agent.

Diagnosis

Title	Number	Description
<i>z/OS Communications Server: IP Diagnosis Guide</i>	GC31-8782	This document explains how to diagnose TCP/IP problems and how to determine whether a specific problem is in the TCP/IP product code. It explains how to gather information for and describe problems to the IBM Software Support Center.
<i>z/OS Communications Server: SNA Diagnosis Vol 1, Techniques and Procedures</i> and <i>z/OS Communications Server: SNA Diagnosis Vol 2, FFST Dumps and the VIT</i>	GC31-6850 GC31-6851	These documents help you identify an SNA problem, classify it, and collect information about it before you call the IBM Support Center. The information collected includes traces, dumps, and other problem documentation.
<i>z/OS Communications Server: SNA Data Areas Volume 1</i> and <i>z/OS Communications Server: SNA Data Areas Volume 2</i>	GC31-6852 GC31-6853	These documents describe SNA data areas and can be used to read an SNA dump. They are intended for IBM programming service representatives and customer personnel who are diagnosing problems with SNA.

Messages and codes

Title	Number	Description
<i>z/OS Communications Server: SNA Messages</i>	SC31-8790	This document describes the ELM, IKT, IST, IUT, IVT, and USS messages. Other information in this document includes: • Command and RU types in SNA messages • Node and ID types in SNA messages • Supplemental message-related information
<i>z/OS Communications Server: IP Messages Volume 1 (EZA)</i>	SC31-8783	This volume contains TCP/IP messages beginning with EZA.
<i>z/OS Communications Server: IP Messages Volume 2 (EZB, EZD)</i>	SC31-8784	This volume contains TCP/IP messages beginning with EZB or EZD.
<i>z/OS Communications Server: IP Messages Volume 3 (EZY)</i>	SC31-8785	This volume contains TCP/IP messages beginning with EZY.
<i>z/OS Communications Server: IP Messages Volume 4 (EZZ, SNM)</i>	SC31-8786	This volume contains TCP/IP messages beginning with EZZ and SNM.
<i>z/OS Communications Server: IP and SNA Codes</i>	SC31-8791	This document describes codes and other information that appear in z/OS Communications Server messages.

Index

Special characters

<rap.h> header 180
adspec definitions 182
filter spec definitions 184
flowspec definitions 181
function interface definitions 184
general definitions 180
policy definitions 184
reservation style definitions 184
tspec definitions 180

A

accessibility 739
addr parameter on RPC call
 on clnttcp_create() 244
 on clntudp_create() 246
 on get_myaddress() 248
 on pmap_getmaps() 250
 on pmap_getport() 251
 on pmap_rmtcall() 252
 on xdrmem_create() 316
adspec definitions 182
adspec pieces 192
adspecs 174
AF_INET6 659
agent distributed protocol interface (DPI) 3, 41
ap parameter on RPC call, on xdr_opaque_auth() 296
application resources, X Windows 708, 717
Application Transparent Transport Layer Security (AT-TLS) 617
applications, functions and protocols
 Network Computing System (NCS) 337
 remote procedure calls (RPC) 207
 SNMP DPI 3, 41
 X Window system interface 195, 661
ar parameter on RPC call, on xdr_accepted_reply() 277
areas, clearing and copying, X Windows 681
areas, filling, X Windows 681
arrp parameter on RPC call, on xdr_array() 278
associate table functions, X Windows 695
asynchronous event handling, RAPI 175
AT-TLS
 CICS transaction considerations 618
 coding the SIOCTTLSTCTL ioctl 623
 defined 617
 sample code for building and issuing the SIOCTTLSTCTL ioctl 633
 starting on a connection 619
 steps for implementing a controlling server application 620
 steps for implementing an aware server application 620
 steps for starting an aware or controlling server application 621
 using the SIOCTTLSTCTL ioctl 619
athena widget set headers 666
Athena Widget Support 709
aup_gids parameter on RPC call, on authunix_create() 224
aupp parameter on RPC call, on xdr_authunix_parms() 279
auth parameter on RPC call, on auth_destroy() 222

auth_destroy(), RPC call 222
authnone_create()(RPC) 223
authorization routines, X Windows 698
authunix_create_default() 225
authunix_create() (RPC) 224

B

BANK sample program data sets, NCS 358
basep parameter on RPC call, on xdr_vector() 313
BINOP sample program
 Data sets, NCS 349
bitmaps, manipulating 690
bp parameter on RPC call, on xdr_bool() 280
buffers, cut and paste, X Windows 689
building X client modules 674

C

c89 utility options 716
callrpc() 226
CC CLIST, processed by RPCGEN 214
changing window attributes 678
Character Set Selection 101
character string sizes, X Windows 682
chdr parameter on RPC call, on xdr_callhdr() 282
choices parameter on RPC call, on xdr_union() 311
Client
 cleanup 207
 free resources 207
 initialize 207
 port numbers 210
 process caLL 207
 remote procedure call 207
clnt parameter on RPC call
 on clnt_call() 230
 on clnt_control() 232
 on clnt_destroy() 234
 on clnt_freeres() 235
 on clnt_geterr() 236
 on clnt_perror() 239
 on clnt_sperror() 242
clnt_broadcast() 228
clnt_call() 230
clnt_control() 232
clnt_create() 233
clnt_destroy() 234
clnt_freeres() 235
clnt_geterr() 236
clnt_pcreateerror() 237
clnt_perrno() 238
clnt_perror() 239
clnt_screateerror() 240
clnt_sperno() 241
clnt_sperror() 242
clnt_stat enumerated type 215
clntraw_create() 243
clnttcp_create() 244
clntudp_create() 246
cmsg parameter on RPC call, on xdr_callmsg() 283

- cnt parameter on RPC call, on xdr_opaque() 295
- color cells, manipulating, X Windows 679
- colormaps, manipulating, X Windows 679
- Communications Server for z/OS, online information xxxii
- compiler nidl 343
- compiling and linking
 - C sockets 6
 - Kerberos 6
 - NCS 342, 346
 - RPC 216
 - SNMP 5, 44
 - UNIX System Services 716
 - X Windows 667, 715
- connecting to an agent through UNIX 82
- controlled-load services formats 190, 191
- COPY 426
- cp parameter on RPC call
 - on xdr_char() 284
 - on xdr_opaque() 295
- creating and destroying windows 677
- cursors, manipulating, X Windows 683

D

- data structures
 - header files for RPCs 216
 - header files for X Window system 340, 665
 - MANIFEST.H, to remap names 215
 - pascal include data set 340
- dfault parameter on RPC call, on xdr_union() 311
- diagnosis, network management 613
- Differentiated Services Policies 655
- Digital Certificate Access Server (DCAS), configuring 641
- Digital Certificate Access Server (DCAS), defining request and response specifications 638
- Digital Certificate Access Server (DCAS), interfacing with 637
- disability 739
- dispatch(), on svc_register() 263
- display functions, X Windows 691
- DNS, online information xxxiii
- dp parameter on RPC call, on xdr_double.parms() 286
- DPI requests, processing 4
- DPI_CLOSE_reason_codes 102
- DPI_PACKET_LEN() 60
- DPI_RC_values 105
- DPI_UNREGISTER_reason_codes 103
- DPI, packet types 102
- DPI, value types 103
- DPIawait_packet_from_agent() 78
- DPIconnect_to_agent_TCP() 80
- DPIconnect_to_agent_UNIXstream() 82
- DPIdebug() 59
- DPIdisconnect_from_agent() 84
- DPIget_fd_for_handle() 85
- DPIsend_packet_to_agent() 86
- drop TCP connections or UDP endpoints
 - interfaces
 - configuration and enablement 584
 - EZBNMIFR 584
 - request and response data structures 596
 - request format 587
 - response format 593
- dscmp parameter on RPC call, on xdr_union() 311

E

- eachresult parameter on RPC call, on clnt_broadcast() 228
- elemsize parameter on RPC call, on xdr_vector() 313
- elproc parameter on RPC call, on xdr_array() 278
- elsize parameter on RPC call, on xdr_array() 278
- ep parameter on RPC call, on xdr_enum.parms() 287, 313
- error code, DPI RESPONSE error codes 102
- error handling, default, X Windows 686
- errp parameter on RPC call, on clnt_geterr() 236
- events handling, X Windows 685
- extension routines, X Windows 693
- EZBCTAPI 568
- EZBNMIFR 584
- EZBNMIFR interface 583
- EZBTMIC1 536

F

- FCAI_IE_InternalErr 464
- FCAI_IE_LengthInvalid 462
- FCAI_PollWait 463
- FCAI_ReqTimer 463
- FCAI_Status_TraceFailed 462
- FCAI_TraceStatus 462
- fDPIparse() 61
- fDPIparse(), SNMP 7
- fDPIset() 62
- file parameter on RPC call, on xdrstdio_create() 321
- Files, Motif, location 203
- filter spec definitions 184
- filter specs 175
- filter, mail 363
- FIND 426
- flowspecs 174
- fonts, loading and freeing, X Windows 682
- formats for controlled-load services 190, 191
- formatting packet and data trace records
 - interfaces 566
 - configuration and enablement 568
 - EZBCTAPI 568
 - passing options 581
 - using the formatter 582
- fp parameter on RPC call, on xdr_float() 289
- FTP Client API 393
 - additional output 456
 - call formats 396
 - compatibility 394
 - control block 399
 - controlling requests 463
 - converting parameter descriptions 397
 - exceptional conditions 464
 - FCAI_IE_InternalErr 464
 - FCAI_IE_LengthInvalid 462
 - FCAI_PollWait 463
 - FCAI_ReqTimer 463
 - FCAI_Status_TraceFailed 462
 - FCAI_TraceStatus 462
 - field values 399
 - guidelines and requirements 394
 - interpreting results 459
 - messages and replies 458
 - output register 455
 - programming notes 461
 - prompts 457
 - reporting failures 462
 - requests 413

FTP Client API (*continued*)

- GETL 422
- INIT 413
- POLL 420
- SCMD 417
- TERM 428
- sample programs 469
- sending requests 413
- specifying a wait time 463
- tracing 465
- unanticipated conditions 464
- z/OS FTP client behavior 397
- FTP client login failure record 558
- FTP client session record 562
- FTP client transfer initialization record 554
- FTP server session record 550
- FTP server transfer initialization record 545
- function
 - DPI_PACKET_LEN() 60
 - DPIawait_packet_from_agent() 78
 - DPIconnect_to_agent_TCP() 80
 - DPIdebug() 59
 - DPIdisconnect_from_agent() 84
 - DPIget_fd_for_handle() 85
 - DPIsend_packet_to_agent() 86
 - fDPIparse() 61
 - fDPIset() 62
 - lookup_host() 88
 - lookup_host6() 89
 - mkDPIAreYouThere() 63
 - mkDPIclose() 64
 - mkDPIopen() 65
 - mkDPIregister() 67
 - mkDPIresponse() 69
 - mkDPIset() 71
 - mkDPItrap() 73
 - mkDPIunregister() 75
 - pDPIpacket() 76

G

- general definitions, RAPI 180
- GET request processing 4
- get_myaddress() 248
- GET-NEXT request processing 4
- GETL 422
 - Assembler example 427
 - COPY operation 422, 426
 - example 424
 - FIND operation 422, 426
- GETL request
 - result guidelines 425
- getreq() (RPC) 261
- getrpcport() 249
- gid parameter on RPC call, on authunix_create() 224
- graphics contexts, manipulating, X Windows 680

H

- handle parameter on RPC call, on xdrrec_create() 317
- header files
 - NCS 340
 - NCS C 340
 - remote procedure calls 216
 - SNMP DPI 5

header files (*continued*)

- X Window system
 - Athena Widget Set 666
 - OSF/Motif 667
 - X Window system and Xt Intrinsics 666
- header files, RAPI 179
- high_vers parameter on RPC call, on svcerr_progvers() 271
- host parameter on RPC call
 - on authunix_create() 224
 - on callrpc() 226
 - on clnt_create() 233
 - on getrpcport() 249
- hosts and access control, X Windows 685

I

- IBM Software Support Center, contacting xxv
- identifying the target display, X Windows 664, 714
- images, manipulating, X Windows 689
- images, transferring 683
- in parameter on RPC call
 - on callrpc() 226
 - on clnt_broadcast() 228
 - on clnt_call() 230
 - on pmap_rmtcall() 252
 - on svc_freeargs() 258
 - on svc_getargs() 259
- include, snmp_dpi.h 106
- info parameter on RPC call, on clnt_control() 232
- information APARs for IP-related documents 735
- information APARs for non- document information 736
- information APARs for SNA-related documents 736
- INIT 413
 - example 414
- INIT request
 - result guidelines 416
- inproc parameter on RPC call
 - on callrpc() 226
 - on clnt_broadcast() 228
 - on clnt_call() 230
 - on pmap_rmtcall() 252
 - on registerrpc() 256
 - on svc_freeargs() 258
 - on svc_getargs() 259
- integrated services adspec 194
- integrated services data structures and macros 186
 - adspec pieces 192
 - formats for controlled-load services 190, 191
 - general definitions 187
 - generic tspec format 189
 - integrated services adspec 194
 - integrated services flowspec 193
 - integrated services tspec 193
- integrated services flowspec 193
- integrated services tspec 193
- interfaces
 - drop TCP connections or UDP endpoints
 - configuration and enablement 584
 - EZBNMIFR 584
 - request and response data structures 596
 - request format 587
 - response format 593
 - formatting packet and data trace records 566
 - configuration and enablement 568
 - EZBCTAPI 568

interfaces (*continued*)

- monitor TCP/UDP endpoints, TCP/IP storage , and TN3270 performance
 - configuration and enablement 584
 - EZBNMIFR 584
 - request and response data structures 596
 - request format 587
 - response format 593
- Real-time TCP/IP network monitoring 528
- RPC 207
- SNA network monitoring data
 - configuration 601
 - data structures and records 606
 - enabling and disabling 601
 - request/response format 602
- SNA network monitoring NMI 600
- TCP/IP callable NMI (EZBNMIFR) 583
- TCP/IP network monitoring
 - communicating with the server 602
 - configuration and enablement 531
 - connecting to the server 532
 - copying the trace buffer 536
 - EZBTMIC1 536
 - records 534
 - requests 533
 - TMI copy buffer service 539
 - TMI_CopyBuffer 538
- X Window System 195, 661
- Internet, finding z/OS information online xxxii
- intrinsics routines, X Windows 700
- ip parameter on RPC call, on xdr_int() 293

K

- keyboard 739
- keyboard events, X Windows 688
- Keyboard settings, manipulating, X Windows 684

L

- len parameter on RPC call
 - on authunix_create() 224
 - on xdr_inline() 292
- lf_smpl.c sample mail filter 363
- libraries
 - SNMP 6
 - X Window system 661
- license, patent, and copyright information 741
- limits 105
- lines, drawing, X Windows 681
- LookAt message retrieval tool xxxiii
- lookup_host() 88
- lookup_host6() 89
- low_vers parameter on RPC call, on svcerr_progrvers() 271
- lp parameter on RPC call, on xdr_long() 294

M

- macro, DPI_PACKET_LEN() 60
- mail filter
 - callbacks
 - xxfi_abort 376
 - xxfi_body 375
 - xxfi_close 377
 - xxfi_connect 372
 - xxfi_envfrom 373

mail filter (*continued*)

- callbacks (*continued*)
 - xxfi_envrcpt 374
 - xxfi_eoh 375
 - xxfi_eom 376
 - xxfi_header 374
 - xxfi_helo 373
- compiling and linking sample source 363
- data access functions
 - smfi_getpriv 367
 - smfi_getsymval 366
 - smfi_setpriv 367
 - smfi_setreply 368
- library control functions
 - smfi_main 366
 - smfi_register 364
 - smfi_setconn 365
 - smfi_settimeout 365
- message modification functions
 - smfi_addheader 368
 - smfi_addrcpt 370
 - smfi_chgheader 369
 - smfi_delrcpt 371
 - smfi_replacebody 371
- running 363
- management information base (MIB) 3, 4, 41
- MANIFEST.H data set, long name remapping 215
- manipulating window properties 678
- manipulating windows 677
- maxsize parameter on RPC call
 - on xdr_array() 278
 - on xdr_bytes() 281
 - on xdr_string() 305
- message retrieval tool, LookAt xxxiii
- messages and replies
 - FTP Client API 458
- MIT extensions to X 694
- mkDPIAreYouThere() 63
- mkDPIclose() 64
- mkDPIopen() 65
- mkDPIregister() 7, 67
- mkDPIresponse() 8, 69
- mkDPIset() 9, 71
- mkDPItrap() 10, 73
- mkDPIunregister() 75
- monitor TCP/UDP endpoints, TCP/IP storage , and TN3270 performance
 - interfaces
 - configuration and enablement 584
 - EZBNMIFR 584
 - request and response data structures 596
 - request format 587
 - response format 593
- Motif-Based Widget Support, X Windows 712

N

NCS

- compiling, linking, and running sample program 353
- IDL data sets 339
- MVS limitations 338
- NCSDEFS.H, defined 340
- portability issues 340
- redefines for sample program 349
- RPC-RUNTIME library 340
- sample programs 348
- USERDEFS.H, user defined 341

- NCS header data sets 340
- NCS portability
 - CLIST, RUNCCP 345
 - converting C identifiers, using CPP define 344
 - NCSDEFS.H, NCS defines 340
 - NCSDEFS.H, required user define 341
 - NIDL compiler 342, 343
 - Running CPP (NCS C Preprocessor) 344
- nelem parameter on RPC call, on xdr_vector() 313
- Network Computing System Reference Manual 342
- Network Driver Interface Specifications 338
- network management
 - diagnosis 613
 - file storage locations 614
 - interfaces
 - formatting packet and data trace records 566
 - Real-time TCP/IP network monitoring 528
 - request errors 612
 - SNA network monitoring NMI 600
 - TCP/IP callable NMI (EZBNMIFR) 583
 - overview 471
- NIDL compiler 342
- NIDL compiler option 344
- NSC, BANK sample program data sets 358
- NSC, BINOP sample program data sets 349
- NSC, NCSSMP sample program data sets 353
- NSC, Running UUID@GEN identifier generator 348

O

- objp parameter on RPC call, on xdr_free() 290
- obtaining properties and atoms, X Windows 678
- obtaining window information, X Windows 678
- op parameter on RPC call
 - on xdrmem_create() 316
 - on xdrstdio_create() 321
- opening and closing a display, X Windows 677
- OSF/Motif header files 667
- out parameter on RPC call
 - on callrpc() 226
 - on clnt_broadcast() 228
 - on clnt_call() 230
 - on clnt_freeres() 235
 - on pmap_rmtcall() 252
 - on svc_sendreply() 265
- outproc parameter on RPC call
 - on callrpc() 226, 228
 - on clnt_broadcast() 228
 - on clnt_call() 230
 - on clnt_freeres() 235
 - on pmap_rmtcall() 252
 - on registerrpc() 256
 - on svc_sendreply() 265

P

- packet DPI, mkDPIpacket() 11
- PAPI
 - client library services 381
 - compiling and linking an application 379
 - connecting and retrieving data
 - papi_connect 382
 - papi_debug 382
 - papi_disconnect 383
 - papi_free_perf_data 384
 - papi_get_perf_data 384

- PAPI (*continued*)
 - helper functions 379
 - papi_get_action_perf_by_id 387
 - papi_get_action_perf_info 387
 - papi_get_actions_count 388
 - papi_get_policy_instance 389
 - papi_get_rule_perf_by_id 389
 - papi_get_rule_perf_info 390
 - papi_get_rules_count 391
 - papi_strerror 391
 - introduction 379
 - return codes 380
 - running an application 380
 - using 379
- papi_connect 382
- papi_debug 382
- papi_disconnect 383
- papi_free_perf_data 384
- papi_get_action_perf_by_id 387
- papi_get_action_perf_info 387
- papi_get_actions_count 388
- papi_get_perf_data 384
- papi_get_policy_instance 389
- papi_get_rule_perf_by_id 389
- papi_get_rule_perf_info 390
- papi_get_rules_count 391
- papi_strerror 391
- pDPIpacket() 76
- pixmaps, creating and freeing, X Windows 680
- pmap_getmaps() 250
- pmap_getport() 251
- pmap_rmtcall() 252
- pmap_set() 254
- pmap_unset() 255
- Policy Agent, programming interface 379
- Policy API (PAPI) 379
- policy performance data retrieval 379
- POLL 420
 - example 420
- POLL request
 - result guidelines 421
- port parameter on RPC call, on pmap_set() 254
- portability issues, NCS 340
- portmapper 209
- portmapper, well-known port 210
- portp parameter on RPC call, on auth_destroy() 252
- pos parameter on RPC call, on xdr_setpos() 303
- pp parameter on RPC call
 - on xdr_pointer() 299
 - on xdr_reference() 300
- proc parameter on RPC call
 - on xdr_free() 290
 - on xdr_pointer() 299
 - on xdr_reference() 300
- procedure calls, remote
 - portmapper, contacting 210
 - target assistance 210
- processing a set request 4
- processing DPI requests 4
- processing GET requests 4
- procname parameter on RPC call, on registerrpc() 256
- procnun parameter on RPC call
 - on callrpc() 226
 - on clnt_broadcast() 228
 - on clnt_call() 230
 - on pmap_rmtcall() 252
 - on registerrpc() 256

prognum parameter on RPC call

- on callrpc() 226
- on clnt_broadcast() 228
- on clnt_create() 233
- on clntraw_create() 243
- on clnttcp_create() 244
- on clntudp_create() 246
- on getrpcport() 249
- on pmap_getport() 251
- on pmap_rmtcall() 252
- on pmap_set() 254
- on pmap_unset() 255
- on registerrpc() 256
- on svc_register() 263
- on svc_unregister() 266

prompts

- FTP Client API 457

protocol parameter on RPC call

- on clnt_create() 233
- on getrpcport() 249
- on pmap_getport() 251
- on pmap_set() 254

Q

query_DPI_port() 12

R

RAPI (Resource Reservation Setup Protocol API) 161

RAPI error codes 178

RAPI error handling 177

RAPI function interface definitions 184

RAPI objects 173

- adspecs 174
- filter specs 175
- flowspecs 174
- sender templates 175
- sender tspecs 174

RAPI policy definitions 184

RAPI reservation style definitions 184

rapi_dispatch() 176

rapi_event_rtn_t 163

rapi_fmt_adspec() 170

rapi_fmt_filtspec() 171

rapi_fmt_flowspec() 172

rapi_fmt_tspecc() 173

rapi_getfd() 177

rapi_release() 166

rapi_reserve() 166

rapi_sender() 167

rapi_session() 169

rapi_version() 170

rc values, DPI_RC_values 105

rdfds parameter on RPC call, on svc_getreq() 261

readit() parameter, on xdrrec_create() 317

Real-time SMF data NMI (SYSTCPSM)

record formats 545

- FTP client login failure record 558
- FTP client session record 562
- FTP client transfer initialization record 554
- FTP server session record 550
- FTP server transfer initialization record 545
- TCP/IP identification section 545

reason code, DPI CLOSE reason codes 102

reason code, DPI UNREGISTER reason codes 103

record formats

Real-time SMF data NMI (SYSTCPSM) 545

FTP client login failure record 558

FTP client session record 562

FTP client transfer initialization record 554

FTP server session record 550

FTP server transfer initialization record 545

TCP/IP identification section 545

recv_buf_size parameter on RPC call

on svctcp_create() 275

on svcudp_create() 276

recvsize parameter on RPC call, on xdrrec_create() 317

recvsz parameter on RPC call, on clnttcp_create() 244

reference sections

well-known port assignments 651

regions, X Windows 688

REGISTER request processing 5

registerrpc() 256

regs parameter on RPC call, on xdr_pmap() 297

remote Procedure and external data representation calls

auth_destroy() 222

authnone_create() 223

authunix_create_default() 225

authunix_create() 224

callrpc() 226

clnt_broadcast() 228

clnt_call() 230

clnt_control() 232

clnt_create() 233

clnt_destroy() 234

clnt_freeres() 235

clnt_geterr() 236

clnt_pcreateerror() 237

clnt_pereno() 238

clnt_perror() 239

clnt_spccreateerror() 240

clnt_sperrno() 241

clnt_sperror() 242

clntraw_create() 243

clnttcp_create() 244

clntudp_create() 246

get_myaddress() 248

getrpcport() 249

pmap_getmaps() 250

pmap_getport() 251

pmap_rmtcall() 252

pmap_set() 254

pmap_unset() 255

registerrpc() 256

rpc_createerr 218

svc_destroy() 257

svc_fds() 219

svc_freeargs() 258

svc_getargs() 259

svc_getcaller() 260

svc_getreq() 261

svc_register() 263

svc_run() 264

svc_sendreply() 265

svc_unregister() 266

svcerr_auth() 267

svcerr_decode() 268

svcerr_noproc() 269

svcerr_noprogram() 270

svcerr_progvers() 271

svcerr_systemerr() 272

svcerr_weakauth() 273

remote Procedure and external data representation calls
(continued)

- svccraw_create() 274
- svctcp_create() 275
- svcudp_create() 276
- xdr_accepted_reply() 277
- xdr_array() 278
- xdr_authunix_parms() 279
- xdr_bool() 280
- xdr_bytes() 281
- xdr_callhdr() 282
- xdr_callmsg() 283
- xdr_char() 284
- xdr_destroy() 285
- xdr_double() 286
- xdr_enum() 287
- xdr_float() 289
- xdr_free() 290
- xdr_getpos() 291
- xdr_inline() 292
- xdr_int() 293
- xdr_long() 294
- xdr_opaque_auth() 296
- xdr_opaque() 295
- xdr_pmap() 297
- xdr_pmaplist() 298
- xdr_pointer() 299
- xdr_reference() 300
- xdr_rejected_reply() 301
- xdr_replymsg() 302
- xdr_setpos() 303
- xdr_short() 304
- xdr_string() 305
- xdr_u_char() 307
- xdr_u_int() 308
- xdr_u_long() 309
- xdr_u_short() 310
- xdr_union() 311
- xdr_vector() 313
- xdr_void() 314
- xdr_wrapstring() 315
- xdrmem_create() 316
- xdrrec_create() 317
- xdrrec_endofrecord() 318
- xdrrec_eof() 319
- xdrrec_skiprecord() 320
- xdrstdio_create() 321
- xprt_register() 322
- xprt_unregister() 323
- remote procedure call (RPC)
 - header files 216
 - portmapper 209
 - portmapper, contacting 210
 - RPCGEN command 213
 - RPCGEN sample programs 331
 - sample programs
 - GENESEND, client 324
 - GENESERV, server 326
 - RAWEX, raw data stream 328
- remote procedure call (RPC) global variables
 - global variables 217
 - rpc_createerr 218
 - svc_fds 219
- remote procedure call (RPC) protocol
 - clnt_stat enumerated type 215
 - compiling and linking 216
 - enumerations 216

- remote procedure call (RPC) protocol (continued)
 - MANIFEST.H, remapping file names with 215
 - porting 215
 - system return messages, accessing 216
 - system return messages, printing 216
- request parameter on RPC call, on clnt_control() 232
- resource manager, X Windows 690
- Resource Reservation Protocol (RSVP) 161
- Resource Reservation Setup Protocol API (RAPI) 161
- return code, DPI CLOSE reason codes 102
- return code, DPI UNREGISTER reason codes 103
- RFC (request for comments)
 - accessing online xxxii
 - list of 719
- rmsg parameter on RPC call, on xdr_replymsg() 302
- rp parameter on RPC call, on xdr_pmaplist() 298
- RPC interface 207
- RPC Porting 215
- rpc_createerr 218
- RPCGEN command parameters 213
- rr parameter on RPC call, on xdr_rejected_reply() 301
- RSVP agent 161
- RSVP error codes 179
- run-time options, nidl 344

S

- s parameter on RPC call
 - on clnt_pcreateerror() 237
 - on clnt_pereno() 239
 - on clnt_spcreateerror() 240
 - on clnt_sperror() 242
- sample FTP Client API programs 469
- sample mail filter 363
- sample NCS programs
 - compiling, linking, and running 353
 - redefines for this sample program 349
- sample RPC programs 324, 348
- SCMD 417
 - example 417
- SCMD request
 - result guidelines 418
- screen saver, controlling, X Windows 685
- Selection, Character Set 101
- send_buf_size parameter on RPC call
 - on svctcp_create() 275
 - on svcudp_create() 276
- sender templates 175
- sender tspecs 174
- sendmail, configuration file 363
- sendmsg() considerations
 - AF_INET6 659
 - IBM C/C++ applications 659
 - UNIX System Services Assembler Callable Services Environment 659
- sendnow parameter on RPC call, on xdrrec_endofrecord() 318
- sendsize parameter on RPC call, on xdrrec_create() 317
- sendsz parameter on RPC call, on clnttcp_create() 244
- server
 - contacting server programs 210
- server, remote procedure calls
 - initialize 207
 - process 207
 - receive request 207
 - reply 207
 - transaction and cleanup 207
- SET, SNMP DPI request 4

- setting window selections 679
- shortcut keys 739
- simple network management protocol (SNMP) 3, 41
- SIOCSAPPLDATA IOCTL 643
- size parameter on RPC call
 - on xdr_pointer() 299
 - on xdr_reference() 300
 - on xdrmem_create() 316
- sizep parameter on RPC call
 - on xdr_array() 278
 - on xdr_bytes() 281
- smfi_addheader 368
- smfi_addrcpt 370
- smfi_chgheader 369
- smfi_delrcpt 371
- smfi_getpriv 367
- smfi_getsymval 366
- smfi_main 366
- smfi_register 364
- smfi_replacebody 371
- smfi_setconn 365
- smfi_setpriv 367
- smfi_setreply 368
- smfi_settimeout 365
- SNA network monitoring data
 - interfaces
 - configuration 601
 - data structures and records 606
 - enabling and disabling 601
 - request/response format 602
- SNA network monitoring NMI interface 600
- SNMP
 - client program 13, 129
 - compiling and linking 5, 44
 - fDPIparse() 7
 - GET-NEXT 4
 - header files 5
 - library routines 6
 - mkDPIpacket() 11
 - mkDPIregister() 7
 - mkDPIresponse() 8
 - mkDPIset() 9
 - mkDPItrap() 10
 - query_DPI_port() 12
 - REGISTER request, processing 5
 - TRAP request 5
- SNMP agents 3, 41
- SNMP manager API 131
- SNMP subagents 3, 41
- SNMP_CLOSE_reason_codes 102
- snmp_dpi_close_packet 91
- snmp_dpi_get_packet 92
- snmp_dpi_hdr 93
- snmp_dpi_next_packet 95
- SNMP_DPI_packet_types 102
- snmp_dpi_resp_packet 96
- snmp_dpi_set_packet 97
- snmp_dpi_u64 100
- snmp_dpi_ureg_packet 99
- snmp_dpi.h 106
- SNMP_ERROR_codes 102
- SNMP_TYPE_value_types 103
- SNMP_UNREGISTER_reason_codes 103
- snmpAddVarBind 133
- snmpBuildPDU 133
- snmpBuildSession 135
- snmpBuildV1TrapPDU 153
- snmpBuildV2TrapOrInformPDU 155
- snmpCreateVarBinds 136
- snmpFreeDecodedPDU 136
- snmpFreeOID 137
- snmpFreePDU 137
- snmpFreeVarBinds 137
- snmpGetErrorInfo 138
- snmpGetNumberOfVarBinds 138
- snmpGetOID 139
- snmpGetRequestId 139
- snmpGetSockFd 140
- snmpGetValue 140
- snmpGetVarbind 141
- snmpInitialize 141
- snmpSendRequest 142
- snmpSetLogFunction 144
- snmpSetLogLevel 144
- snmpSetRequestId 145
- snmpTerminate 146
- snmpTerminateSession 146
- snmpValueCreateBits 147
- snmpValueCreateCounter32 147
- snmpValueCreateCounter64 147
- snmpValueCreateGauge32 148
- snmpValueCreateInteger 148
- snmpValueCreateInteger32 149
- snmpValueCreateIPAddr 149
- snmpValueCreateNSApAddr 150
- snmpValueCreateNull 150
- snmpValueCreateOctet 150
- snmpValueCreateOID 151
- snmpValueCreateOpaque 151
- snmpValueCreateTimerTicks 152
- snmpValueCreateUnsignedInteger32 152
- sock parameter on RPC call, on svcudp_create() 275
- sockets
 - compiler restrictions 713
 - UNIX System Services 713
 - using 713
- sockp parameter on RPC call
 - on clnttcp_create() 244
 - on clntudp_create() 246
 - on svcudp_create() 276
- software requirements
 - UNIX System Services 714
 - X Windows 662
- sp parameter on RPC call
 - on xdr_bytes() 281
 - on xdr_short() 304
 - on xdr_string() 305
 - on xdr_wrapstring() 315
- stat parameter on RPC call
 - on clnt_perrno() 238
 - on clnt_sperrno() 241
- structure
 - snmp_dpi_close_packet 91
 - snmp_dpi_get_packet 92
 - snmp_dpi_hdr 93
 - snmp_dpi_next_packet 95
 - snmp_dpi_resp_packet 96
 - snmp_dpi_set_packet 97
 - snmp_dpi_u64 100
 - snmp_dpi_ureg_packet 99
- subroutines (X Window system) 677
- svc_destroy() 257
- svc_fds() 219
- svc_freeargs() 258

- svc_getargs() 259
- svc_getcaller() 260
- svc_getreq() 261
- svc_register() 263
- svc_run() 264
- svc_sendreply() 265
- svc_unregister() 266
- svcerr_auth() 267
- svcerr_decode() 268
- svcerr_noproc() 269
- svcerr_noprogram() 270
- svcerr_progvers() 271
- svcerr_systemerr() 272
- svcerr_weakauth() 273
- svccraw_create() 274
- svctcp_create() 275
- svcdup_create() 276
- synchronization, enable and disable, X Windows 686
- syntax diagram, how to read xxvi
- SYSTCPCN 543
- SYSTCPCN interface 528
- SYSTCPDA 540
- SYSTCPDA interface 528
- SYSTCPSM 544
 - record formats 545
- SYSTCPSM interface 528
- system toolkit, X Windows 698

T

- tasks
 - (gerund phrase)
 - steps for 426, 430
 - (noun, gerund phrase)
 - steps 158
 - aware or controlling server application, starting
 - steps 621
 - aware server application, implementing
 - steps 620
 - capturing trace records
 - steps for 577
 - Compile
 - steps for the BANK program 360
 - steps for the NCSSMP program 355
 - steps for the sample BINOP program 351
 - controlling server application, implementing
 - steps 620
 - issuing the INIT request
 - steps for 416
 - issuing the POLL request
 - steps for 421
 - issuing the SCMD request
 - steps for 419
 - Link
 - steps for the BANK program 361
 - steps for the NCSSMP program 356
 - steps for the sample BINOP program 352
 - Run
 - steps for the BANK program 362
 - steps for the NCSSMP program 357
 - steps for the sample BINOP program 353
 - Setup
 - steps for the BANK program 358
 - steps for the NCSSMP program 354
 - steps for the sample BINOP program 350
- TCP_KeepAlive socket option 645

- TCP/IP
 - online information xxxii
 - protocol specifications 719
- TCP/IP identification section 545
- TCP/IP network monitoring
 - format of service-specific data 540
 - interfaces 528
 - communicating with the server 602
 - configuration and enablement 531
 - connecting to the server 532
 - copying the trace buffer 536
 - EZBTMIC1 536
 - records 534
 - requests 533
 - TMI_CopyBuffer 538
 - SYSTCPCN 543
 - SYSTCPDA 540
 - SYSTCPSM 544
 - TMI copy buffer service 539
- tcpip.v3r1.data sets
 - SEZAOLDX 661
 - SEZARNT1 661
 - SEZAX11L 661
 - SEZAXAWL 661
 - SEZAXMLB 661
 - SEZAXTLB 661
- TERM 428
 - example 429
- TERM request
 - result guidelines 429
- text, drawing, X Windows 683
- TMI_CopyBuffer 538
- tout parameter on RPC call
 - on clnt_call() 230
 - on pmap_rmtcall() 252
- tracing
 - FTP Client API 465
- trademark information 750
- TRAP request processing 5
- tspec definitions 180
- tspec format 189
- types, DPI packet types 102

U

- ucp parameter on RPC call, on xdr_u_char() 307
- uid parameter on RPC call, on authunix_create() 224
- ulp parameter on RPC call, on xdr_u_long() 309
- UNIX System Services
 - compiling and linking 716
 - sockets 713
 - software requirements 714
 - using 713
 - what is provided 713
- UNIXstream function 82
- unp parameter on RPC call, on xdr_union() 311
- up parameter on RPC call, on xdr_u_int() 308
- using
 - Motif 203
 - X Window System 195
- usp parameter on RPC call, on xdr_u_short() 310
- utility routines, X Windows 695
- UUID@GEN identifier generator 348

V

- value ranges 105
- value types, SNMP_TYPE_value_types 103
- versnum parameter on RPC call
 - on callrpc() 226
 - on clnt_broadcast() 228
 - on clnt_create() 233
 - on clntraw_create() 243
 - on clnttcp_create() 244
 - on clntudp_create() 246
 - on getrpcport() 249
 - on pmap_getport() 251
 - on pmap_rmtcall() 252
 - on pmap_set() 254
 - on pmap_unset() 255
 - on registerrpc() 256
 - on svc_register() 263
 - on svc_unregister() 266
- visual types 689
- VTAM, online information xxxii

W

- wait parameter on RPC call, on clntudp_create() 246
- well-known port assignments 651
- what is provided, UNIX System Services 713
- what is provided, X Windows 661
- why parameter on RPC call, on svcerr_auth() 267
- widgets, defining 699
- window manager functions, X Windows 683
- window manager, communicating with, X Window system 686
- writeln() parameter on RPC, on xdrrec_create() 317

X

- X Window system
 - application resource file 664, 714
 - areas, clearing and copying 681
 - areas, filling 681
 - associate table functions 695
 - bitmaps, manipulating 690
 - buffers, cut and paste 689
 - changing window attributes 678
 - character string sizes 682
 - color cells, manipulating 679
 - colormaps, manipulating 679
 - creating an application 665
 - creating and destroying windows 677
 - cursors, manipulating, X Windows 683
 - defining widgets 699
 - display functions 691
 - error handling, default 686
 - events handling 685
 - extension routines 693
 - fonts, loading and freeing 682
 - graphics contexts 680
 - header files 665, 666
 - hosts and access control 685
 - identifying target display 664, 714
 - images, manipulating 689
 - images, transferring 683
 - keyboard events, manipulating 688
 - keyboard settings, handling 684
 - lines, drawing 681
 - manipulating window properties 678

- X Window system (*continued*)
 - manipulating windows 677
 - obtaining properties and atoms 678
 - obtaining window information 678
 - opening and closing a display 677
 - pixmaps, creating and freeing 680
 - porting applications 699
 - regions, manipulating 688
 - resource manager 690
 - sample programs, X Windows 671
 - screen saver, controlling 685
 - setting window selections 679
 - synchronization, enable and disable 686
 - text, drawing 683
 - visual types 689
 - window manager functions 683
 - window managers, communicating 686
 - X client applications 672
 - X client modules, building 674
 - X defaults 664
 - X Window system Interface 195, 661, 662
 - X Window system Toolkit 698
 - Xt Intrinsics 708, 717
- X Window system, application layer
 - Application Resources 708, 717
 - Athena Widget Support 709
 - Authorization Routines 698
 - Miscellaneous Utility Routines 695
 - MIT Extensions 694
 - Motif-Based Widget Support 712
 - Routines 677
 - Xt Intrinsics Routines 700
- X Window system, what is provided 661
- xdr_accepted_reply() 277
- xdr_array() 278
- xdr_authunix_parms() 279
- xdr_bool() 280
- xdr_bytes() 281
- xdr_callhdr() 282
- xdr_callmsg() 283
- xdr_char() 284
- xdr_destroy() 285
- xdr_double() 286
- xdr_elem parameter on RPC call, on xdr_vector() 313
- xdr_enum() 287
- xdr_float() 289
- xdr_free() 290
- xdr_getpos() 291
- xdr_inline() 292
- xdr_int() 293
- xdr_long() 294
- xdr_opaque_auth() 296
- xdr_opaque() 295
- xdr_pmap() 297
- xdr_pmaplist() 298
- xdr_pointer() 299
- xdr_reference() 300
- xdr_rejected_reply() 301
- xdr_replymsg() 302
- xdr_setpos() 303
- xdr_short() 304
- xdr_string() 305
- xdr_u_char() 307
- xdr_u_int() 308
- xdr_u_long() 309
- xdr_u_short() 310
- xdr_union() 311

- xdr_vector() 313
- xdr_void() 314
- xdr_wrapstring() 315
- xdrmem_create() 316
- xdrrec_create() 317
- xdrrec_endofrecord() 318
- xdrrec_eof() 319
- xdrrec_skiprecord() 320
- xdrs parameter on RPC call
 - on xdr_accepted_reply() 277
 - on xdr_array() 278
 - on xdr_authunix_parms() 279
 - on xdr_bool() 280
 - on xdr_bytes() 281
 - on xdr_callhdr() 282
 - on xdr_callmsg() 283
 - on xdr_char() 284
 - on xdr_destroy() 285
 - on xdr_double() 286
 - on xdr_enum() 287
 - on xdr_float() 289
 - on xdr_getpos() 291
 - on xdr_inline() 292
 - on xdr_int() 293
 - on xdr_long() 294
 - on xdr_opaque_auth() 296
 - on xdr_opaque() 295
 - on xdr_pmap() 297
 - on xdr_pmaplist() 298
 - on xdr_pointer() 299
 - on xdr_reference() 300
 - on xdr_rejected_reply() 301
 - on xdr_replymsg() 302
 - on xdr_setpos() 303
 - on xdr_short() 304
 - on xdr_string() 305
 - on xdr_u_char() 307
 - on xdr_u_int() 308
 - on xdr_u_long() 309
 - on xdr_u_short() 310
 - on xdr_union() 311
 - on xdr_vector() 313
 - on xdr_wrapstring() 315
 - on xdrmem_create() 316
 - on xdrrecc_create() 317
 - on xdrrecc_endofrecord() 318
 - on xdrrecc_eof() 319
 - on xdrrecc_skiprecord() 320
 - on xdrstdio_create() 321
- xdrstdio_create() 321
- xprt parameter on RPC call
 - on svc_destroy() 257
 - on svc_freeargs() 258
 - on svc_getargs() 259
 - on svc_getcaller() 260
 - on svc_register() 263
 - on svc_sendreply() 265
 - on svcerr_auth() 267
 - on svcerr_decode() 268
 - on svcerr_noproc() 269
 - on svcerr_noprogram() 270
 - on svcerr_progvers() 271
 - on svcerr_systemerr() 272
 - on svcerr_weakauth() 273
 - on xprt_register() 322
 - on xprt_unregister() 323
- xprt_register() 322

- xprt_unregister() 323
- xxfi_abort 376
- xxfi_body 375
- xxfi_close 377
- xxfi_connect 372
- xxfi_envfrom 373
- xxfi_envrcpt 374
- xxfi_eoh 375
- xxfi_eom 376
- xxfi_header 374
- xxfi_helo 373

Z

- z/OS, documentation library listing 753
- z/OS, listing of documentation available 735

Communicating Your Comments to IBM

If you especially like or dislike anything about this document, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this document. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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.

Please send your comments to us in either of the following ways:

- If you prefer to send comments by FAX, use this number: 1+919-254-1258
- If you prefer to send comments electronically, use this address:
 - `comsvrcf@us.ibm.com`
- If you prefer to send comments by post, use this address:

International Business Machines Corporation
Attn: z/OS Communications Server Information Development
P.O. Box 12195, 3039 Cornwallis Road
Department AKCA, Building 501
Research Triangle Park, North Carolina 27709-2195

Make sure to include the following in your note:

- Title and publication number of this document
- Page number or topic to which your comment applies.



Program Number: 5694-A01

Printed in USA

SC31-8787-09



Spine information:



z/OS Communications Server

z/OS V1R9.0 Comm Svr: IP Programmer's Guide and Reference

Version 1
Release 9