z/OS Communications Server

IP Programmer's Guide and Reference

Version 1 Release 13

SC31-8787-14

Note:
Before using this information and the product it supports, be sure to read the general information under "Notices" on page 1013.

Fifteenth Edition (April 2012)

This edition applies to Version 1 Release 13 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 email:

comsvrcf@us.ibm.com

World Wide Web:

http://www.ibm.com/systems/z/os/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.

Contents

Figures

Tables

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.

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 the 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.

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 document 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 163 describes the calls provided though 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 197 describes the X Window System API.
- Chapter 8, “Remote procedure calls in the z/OS Communications Server environment,” on page 209 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 339 provides information on use of UNIX System Services RPC and deviations from Sun RPC 4.0.
- Chapter 10, “Network Computing System,” on page 343 describes the NCS tools used for heterogeneous distributed computing.
- Chapter 11, “Running the sample mail filter program,” on page 369 explains how to run the sample mail filter program, lf_smpl.c.
- Chapter 12, “Policy API (PAPI),” on page 385 describes the Policy Agent API (PAPI).
- Chapter 13, “FTP Client Application Programming Interface (API),” on page 399 describes the callable application programming interface to the z/OS FTP client.
- Chapter 14, “Network management interfaces,” on page 479 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 667 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, “Trusted TCP connections,” on page 687 describes how TCP/IP stacks within a sysplex or a subplex can use the SO_CLUSTERCONNTYPE socket option, the SIOCSPARTNERINFO ioctl call, and the SIOCGPARTNERINFO ioctl to exchange security information.
- Chapter 17, “Interfacing with the Digital Certificate Access Server (DCAS),” on page 707 documents the programming interface specifications for the Digital Certificate Access Server (DCAS) that runs on the z/OS operating system.
- Chapter 18, “Miscellaneous programming interfaces,” on page 713 describes programming interfaces including the TCP_KeepAlive function.
- Appendix A, “Well-known port assignments,” on page 721 lists the well-known port assignments for transport protocols TCP and UDP.
- Appendix B, “Programming interfaces for providing classification data to be used in differentiated services policies,” on page 725 provides information on the Differentiated Services (DS) aspect of QoS and the passing of application classification data on SENDMSG.
- Appendix C, “Type 109 SMF records,” on page 731 describes the format of syslogd messages, as written to SMF.
- Appendix D, “Type 118 SMF records,” on page 733 describes the type 118 SMF records for the Telnet and FTP servers, API calls, FTP and Telnet client calls, and syslogd. This appendix also shows the record layouts.

- Appendix E, “Type 119 SMF records,” on page 743 describes the type 119 SMF records that are created for several TCP/IP functions. This appendix also shows the record layouts.
- Appendix G, “X Window System interface V11R4 and Motif version 1.1,” on page 933 describes the X Window System application programming interface (API).
- Appendix H, “Related protocol specifications,” on page 991 lists the related protocol specifications for TCP/IP.
- "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 website: http://www.software.ibm.com/network/commserver/support/

Most problems can be resolved at this website, 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 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 1041.

Conventions and terminology that are 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 the exit routines described in this document are *installation-wide exit routines*. The installation-wide exit routines also called installation-wide exits, exit routines, and exits throughout this document.

The TPF logon manager, although included 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 that are used in this document, you can view the latest IBM terminology at the IBM Terminology website.

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.

► Indicates that the command syntax is continued.

| Marks the beginning and end of a fragment or part of the command syntax.

►◄ Marks the end 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

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

- 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).

Parameters

The following types of parameters are used in syntax diagrams.

Required
Required parameters are displayed on the main path.

Optional
Optional parameters are displayed below the main path.

Default
Default parameters are displayed above the main path.

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).

For the z/OS UNIX commands, the keywords must be entered in the case indicated in the syntax diagram.

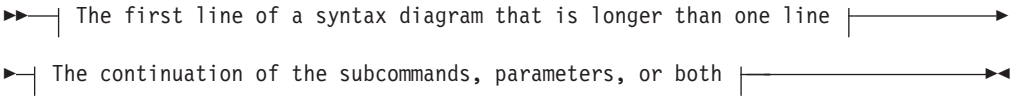
Variables are italicized, appear in lowercase letters, and represent names or values you supply. For example, a data set is a variable.

Syntax examples

In the following example, the PUt subcommand is a keyword. The required variable parameter is *local_file*, and the optional variable parameter is *foreign_file*. 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.

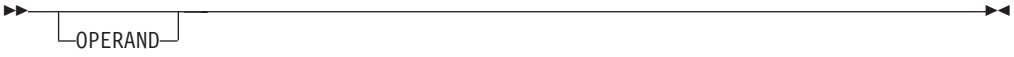


Required operands

Required operands and values appear on the main path line. You must code required operands and values.

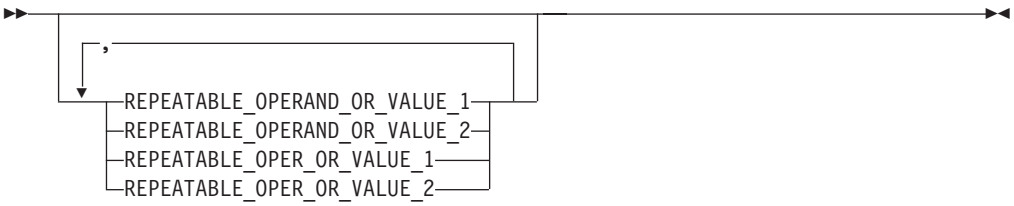
Optional values

Optional operands and values appear below the main path line. You do not have 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).

Default operands

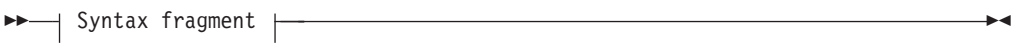
Default operands and values appear above the main path line. TCP/IP uses the default if you omit the operand entirely.

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.

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:

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 "Bibliography" on page 1023, 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.

Softcopy information

Softcopy publications are available in the following collections.

Titles	Order Number	Description
z/OS V1R13 and Software Products DVD Collection	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 format and PDF files. This collection combines SK3T-4269 and SK3T-4270.
IBM System z® Redbooks Collection	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 (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
DNS and BIND, Fifth Edition, O'Reilly Media, 2006	ISBN 13: 978-0596100575
Routing in the Internet, Second Edition, Christian Huitema (Prentice Hall 1999)	ISBN 13: 978-0130226471
sendmail, Fourth Edition, Bryan Costales, Claus Assmann, George Jansen, and Gregory Shapiro, O'Reilly Media, 2007	ISBN 13: 978-0596510299
SNA Formats	GA27-3136
TCP/IP Illustrated, Volume 1: The Protocols, W. Richard Stevens, Addison-Wesley Professional, 1994	ISBN 13: 978-0201633467
TCP/IP Illustrated, Volume 2: The Implementation, Gary R. Wright and W. Richard Stevens, Addison-Wesley Professional, 1995	ISBN 13: 978-0201633542
TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols, W. Richard Stevens, Addison-Wesley Professional, 1996	ISBN 13: 978-0201634952

Title	Number
TCP/IP Tutorial and Technical Overview	GG24-3376
Understanding LDAP	SG24-4986
z/OS Cryptographic Services System SSL Programming	*SC24-5901*
z/OS Integrated Security Services LDAP Server Administration and Use	*SC24-5923*
z/OS JES2 Initialization and Tuning Guide	*SA22-7532*
z/OS Problem Management	*G325-2564*
z/OS MVS Diagnosis: Reference	*GA22-7588*
z/OS MVS Diagnosis: Tools and Service Aids	*GA22-7589*
z/OS MVS Using the Subsystem Interface	*SA22-7642*
z/OS Program Directory	*GI10-0670*
z/OS UNIX System Services Command Reference	*SA22-7802*
z/OS UNIX System Services Planning	*GA22-7800*
z/OS UNIX System Services Programming: Assembler Callable Services Reference	*SA22-7803*
z/OS UNIX System Services User's Guide	*SA22-7801*
z/OS XL C/C++ Run-Time Library Reference	*SA22-7821*
zEnterprise 196, System z10, System z9 and eServer zSeries OSA-Express Customer's Guide and Reference	*SA22-7935*

Redbooks

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

Title	Number
IBM z/OS V1R12 Communications Server TCP/IP Implementation, Volume 1: Base Functions, Connectivity, and Routing	SG24-7896
IBM z/OS V1R12 Communications Server TCP/IP Implementation, Volume 2: Standard Applications	SG24-7897
IBM z/OS V1R12 Communications Server TCP/IP Implementation, Volume 3: High Availability, Scalability, and Performance	SG24-7898
IBM z/OS V1R12 Communications Server TCP/IP Implementation, Volume 4: Security and Policy-Based Networking	SG24-7899
IBM Communication Controller Migration Guide	SG24-6298
IP Network Design Guide	SG24-2580
Managing OS/390® TCP/IP with SNMP	SG24-5866
Migrating Subarea Networks to an IP Infrastructure Using Enterprise Extender	SG24-5957
SecureWay™ Communications Server for OS/390 V2R8 TCP/IP: Guide to Enhancements	SG24–5631
SNA and TCP/IP Integration	SG24-5291
TCP/IP in a Sysplex	SG24-5235
TCP/IP Tutorial and Technical Overview	GG24-3376
Threadsafe Considerations for CICS	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/systems/z/os/zos/

z/OS Internet Library

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

www.ibm.com/systems/z/os/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 Communications Server performance information

This site contains links to the most recent Communications Server performance reports.

http://www.ibm.com/support/docview.wss?uid=swg27005524

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/atsmastr.nsf

RFCs

Search for and view Request for Comments documents in this section of the Internet Engineering Task Force website, 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 website

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 websites are provided for convenience only and do not in any manner serve as an endorsement of these websites.

DNS websites

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

USENET news groups
comp.protocols.dns.bind

BIND mailing lists
https://lists.isc.org/mailman/listinfo

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.

The z/OS Basic Skills Information Center

The z/OS Basic Skills Information Center is a web-based information resource intended to help users learn the basic concepts of z/OS, the operating system that runs most of the IBM mainframe computers in use today. The Information Center is designed to introduce a new generation of Information Technology professionals to basic concepts and help them prepare for a career as a z/OS professional, such as a z/OS system programmer.

Specifically, the z/OS Basic Skills Information Center is intended to achieve the following objectives:

- Provide basic education and information about z/OS without charge
- Shorten the time it takes for people to become productive on the mainframe
- Make it easier for new people to learn z/OS

To access the z/OS Basic Skills Information Center, open your web browser to the following website, which is available to all users (no login required):
http://publib.boulder.ibm.com/infocenter/zoslnctr/v1r7/index.jsp

Summary of changes

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

Changes made in z/OS Communications Server Version 1 Release 13, as updated April 2012

This document contains information previously presented in *z/OS Communications Server: IP Programmer's Guide and Reference*, SC31-8787-13, which supports z/OS Version 1 Release 13.

Changes made in z/OS Communications Server Version 1 Release 13

This document contains information previously presented in *z/OS Communications Server: IP Programmer's Guide and Reference*, SC31-8787-12, which supports z/OS Version 1 Release 12.

New information:

- TCP/IP serviceability enhancements, see the following topics:
  - "snmpSetLogLevel – Set the logging level" on page 145
  - "Debugging the SNMP manager API" on page 159
- Network address translation traversal support for IKE version 2, see the following topics:
  - "NMsec_GET_IPTUNDYNSTACK" on page 518
  - "NMsec_GET_IKETUN" on page 525
  - "IPSec IKE tunnel activation and refresh record (subtype 73)" on page 888
- Simplified authorization requirements for real-time TCP/IP network monitoring NMI, see the following topics:
  - "Packet and data trace formatting NMI" on page 547
  - "Real-time TCP/IP network monitoring NMI" on page 564
  - "SNA network monitoring NMI request/response format" on page 626
- HPR packet trace analyzer for Enterprise Extender, see Table 62 on page 561.
- Performance improvements for Enterprise Extender traffic, see "Format of service-specific portion of output records" on page 578.
- NMI for retrieving system resolver configuration information, see "Resolver NMI (EZBREIFR)" on page 605.
- System resolver autonomic quiescing of unresponsive name servers, see "Resolver NMI (EZBREIFR)" on page 605.
- Expanded intrusion detection services, see Table 155 on page 753.
- OSA-Express4S QDIO IPv6 checksum and segmentation offload, see the following topics:
  - "TCP/IP profile record IPv4 configuration section" on page 773
  - "TCP/IP profile record IPv6 configuration section" on page 777
  - "TCP/IP profile record Global configuration section" on page 782

- HiperSockets™ optimization for intraensemble data networks, see the following topics:
  - Table 173 on page 782
  - Table 196 on page 828
- Improved security granularity for VIPARANGE DVIPAs, see "TCP/IP profile record dynamic VIPA (DVIPA) address section" on page 808.
- CSSMTP extended retry, see the following tables:
  - Table 228 on page 856
  - Table 230 on page 857
  - Table 233 on page 860
  - Table 240 on page 866
  - Table 245 on page 870
  - Table 248 on page 875
  - Table 249 on page 876
  - Table 250 on page 877
- CSSMTP enhancements, see Table 230 on page 857.

Changed information:

- Require the _ALL_SOURCE feature test macro for RPC compiling, see "Using z/OS UNIX System Services RPC" on page 340.
- Replace unsigned long with uint32_t and uint64_t types, see "Real-time NMI: Processing the output records" on page 577.
- DCAS behavior clarification, see "Understanding how clients interface to DCAS" on page 707.
- AT-TLS cipher values for SMF records, see "Standard data format concepts" on page 747.
- Description clarification to SMF119ML_MH_RCPTRPY data type, see Table 242 on page 868.

Changes made in z/OS Communications Server Version 1 Release 12

This document contains information previously presented in *z/OS Communications Server: IP Programmer's Guide and Reference*, SC31-8787-11, which supports z/OS Version 1 Release 11.

New information:

- Management data for CSSMTP, see "Real-time TCP/IP network monitoring NMI" on page 564.
- Trusted TCP connections, see Chapter 16, "Trusted TCP connections," on page 687.

Changed information:

- Enhancements to SNMP Manager API, see the following topics:
  - "SNMP manager API functions" on page 132
  - "SNMP manager API configuration file" on page 156
- SMF event records for sysplex events, see the following topics:
  - Chapter 14, "Network management interfaces," on page 479
  - Appendix E, "Type 119 SMF records," on page 743

- IPSec support for certificate trust chains and certificate revocation lists, see the following topics:
  - "IPSec NMI response messages" on page 500
  - "IPSec IKE tunnel activation and refresh record (subtype 73)" on page 888
- IPSec support for cryptographic currency, see the following topics:
  - "IPSec NMI response messages" on page 500
  - "IPSec dynamic tunnel activation and refresh record (subtype 75)" on page 897
  - Appendix E, "Type 119 SMF records," on page 743
- Packet trace filtering for encapsulated packets, see "Format of service-specific portion of output records" on page 578.
- Data trace records for socket data flow start and end, see "Processing the *cte* records for SYSTCPDA and SYSTCPOT" on page 578.
- Performance improvement to TCP/IP callable NMI (EZBNMIFR) GetConnectionDetail request, see "TCP/IP NMI request format" on page 642.
- Enhancements to TCP/IP callable NMI (EZBNMIFR) - network interface and TCP/IP statistics, see the following topics:
  - "TCP/IP NMI request format" on page 642
  - "TCP/IP NMI response format" on page 653
  - "File storage locations" on page 664
- Enterprise Extender connection health verification, see "SNA network monitoring NMI request and response data structures and records" on page 630.
- Support for z/OS Communications Server for intraensemble networks, see the following topics:
  - "Sysplex-specific connection routing information" on page 687
  - "TCP/IP profile event record (subtype 4)" on page 763
- Control joining the sysplex XCF group, see Table 173 on page 782.
- Performance improvements for Sysplex Distributor connection routing, see Table 175 on page 789.
- Extend Sysplex Distributor support for DataPower® for IPv6, see the following topics:
  - Table 183 on page 808
  - Table 185 on page 813
- Sysplex distributor support for hot-standby server, see Table 185 on page 813.

Moved information:

- Information about SMF records that previously appeared in *z/OS Communications Server: IP Configuration Reference* has been moved to the following appendixes:
  - Appendix C, "Type 109 SMF records," on page 731
  - Appendix D, "Type 118 SMF records," on page 733
  - Appendix E, "Type 119 SMF records," on page 743
- Information about application data that previously appeared in *z/OS Communications Server: IP Configuration Reference* has been moved to Appendix F, "Application data," on page 917.

Changes made in z/OS Communications Server Version 1 Release 11

This document contains information previously presented in *z/OS Communications Server: IP Programmer's Guide and Reference*, SC31-8787-10, which supports z/OS Version 1 Release 10.

New information:

- FTP access to UNIX named pipes, see "Predefined REXX variables" on page 415.
- IPSec enhancements, see "Local IPSec NMI" on page 480.
- NSS private key and certificate services for XML appliances, see "NMsec_GET_CLIENTINFO" on page 541.
- The Network Management Interfaces (NMI) has added the real time collection of trace data from the OSAENTA traces, see "Real-time TCP/IP network monitoring NMI" on page 564.
- Network management interface enhancements - stack configuration data, see the following topics:
  - "Real-time TCP/IP network monitoring NMI" on page 564
  - "TCP/IP callable NMI (EZBNMIFR)" on page 637
- AT-TLS enhancements, see the following topics:
  - "Real-time TCP/IP network monitoring NMI" on page 564
  - "SIOCTTLSCTL (X'C038D90B')" on page 674
- Network management interface enhancements - sysplex networking data, see "TCP/IP callable NMI (EZBNMIFR)" on page 637.
- Network management interface enhancements - detailed CSM usage, see "SNA network monitoring NMI" on page 624.

Deleted information:

- The SYSTCPDA packet records type 1, 2, and 3 are no longer created by TCP/IP and are no longer described in this document.

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 *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 list 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, users who are writing new subagents or modifying old ones should 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 that is included 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_dpx.h, that is provided in SEZACMAC. When an included header file exists as a member of an MVS

partitioned data set, the underscore (_) in the header file name is changed to an at sign (@) when the header file is located during the compiling of a program. Therefore, header file snmp_dpx.h can be found as member SNMP@DPX in the SEZACMAC data set. 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 "Required actions for 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 H, "Related protocol specifications," on page 991 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() call information in *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 *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 example shows 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_dpx.h
```

This header file is installed in the SEZACMAC data set as member SNMP@DPX.

SNMP agent DPI: Compiling and linking

You can use several methods to compile, link-edit, and execute your TCP/IP C source program in MVS. This topic 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

SNMP agent DPI: Sample compilation cataloged procedure additions

Include the following steps in the compilation 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
```

SNMP agent DPI: Sample link-edit cataloged procedure additions

Include the following step 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 *z/OS XL C/C++ User's Guide*.

SNMP DPI library routines

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

mkDPIlist()

```
#include <snmp_dpx.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:

packet	A pointer to a structure dpi_set_packet, or NULL
oid_name	The object identifier of the variable
type	The type of the value
len	The length of the value
value	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_dpx.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_dpx.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 example shows 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_dpx.h>
#include <bsdtypes.h>

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

Parameters:

ret_code Specifies the error code to be returned.

value_list 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 example shows the mkDPIresponse() call.

```
unsigned char *packet;

int error_code;
struct dpi_set_packet *ret_value;

packet = mkDPIresponse(error_code, ret_value);

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

mkDPIset()

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

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

Parameters:

oid_name	Specifies the object identifier of the variable.
type	Specifies the type of the object identifier.
len	Indicates the length of the value.
value	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 values, 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_dpx.h>
#include <bsdtypes.h>

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

Parameters:

generic	Specifies the generic field in the SNMP TRAP packet.
specific	Specifies the specific field in the SNMP TRAP packet.
value_list	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 mkDPIregister(). If an error is encountered while creating the packet, a NULL pointer is returned.

Example: The following example shows 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_dpx.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:

generic The generic field for the SNMP TRAP packet.

specific The specific field for the SNMP TRAP packet.

value_list A pointer to a structure dpi_set_packet, which contains one or more variables to be sent with the SNMP TRAP packet. Or NULL if no variables are to be sent.

enterprise_oid 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 mkDPItrape() 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_dpx.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 example shows 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_dpx.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 topic 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 H, “Related protocol specifications,” on page 991 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 values 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 list shows 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 H, "Related protocol specifications," on page 991 for information about accessing RFCs.

dpiSample table MIB descriptions

The following sample 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 sample 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           */
/*                                                                   */
/*                                                                   */
/*  Licensed Materials - Property of IBM                             */
/*  5694-A01                                                         */
/*  Copyright IBM Corp. 1991,  2010                                  */
/*                                                                   */
/*  Status = CSV1R12                                                 */
/*                                                                   */
/* dpisampl.c   - a sample SNMP-DPI subagent                         */
/*              - can be used to test agent DPI implementations.     */
/*                                                                   */
/* Change Activity:                                                  */
/*                                                                   */
/* Flag Reason   Release  Date   Origin    Description               */
/* ---- -------- -------- ------ --------  ----------------------    */
/* $P1= MV11816  TCPV3R2  960524 jab     : zero siucv fields for     */
/*                                         connect                   */
/* $Q1= MV27495  D316     030218 SHAGGAR:  Include current header    */
/*                                         file                      */
/* $E1= D144106  RABASE   080327 ADAMSON : Change snmp@dpi.h to      */
/*                                         snmp_dpx.h                */
/* $H1= D149128  RCBASE   091011 ADAMSON : Correct missing ending    */
/*                                         comment chars in prolog   */
/*                                                                   */
/*********************************************************************/
/*  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"
```

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

```
#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>
#include <string.h>                                          /*@Q1C*/
#include <inet.h>
#define  asciitoebcdic asciitoe
#define  ebcdictoascii ebcdicto
extern char   ebcdicto[], asciitoe[];
#pragma linkage(cmxlate,OS)
#define DO_ETOA(a) cmxlate((a),ebcdictoascii,strlen((a)))
#define DO_ATOE(a) cmxlate((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
```

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

```
// #include <sys/select.h>
#endif /* AIX221 */

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

#include <stdio.h>
#include "snmp_dpx.h"                                          /*@E1C*/

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

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

#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_trape();
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_trape(void);
static void issue_std_traps(void);
static void issue_ent_traps(void);
static void issue_ent_trapse(void);
```

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

```
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)

/* 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, /* SNMP_TYPE_EMPTY */  /* 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 */
```

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

```
        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 */
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 */
```

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

```
      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 */
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_trape();            /* 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;
   }                               /* end switch (do_trap) */
   do_trap = 0;                    /* 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
```

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

```
   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 */
      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");
```

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

```
         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 */
{
   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() */
```

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

```
       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;
      }
   } 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);
```

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

```
     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);
     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);
}
```

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

```
#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");
      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) */
```

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

```
     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);
     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 */
```

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

```
   }
}

#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) {
     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 octect 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)
```

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

```
#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++)
          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 octect 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("'");
```

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

```
        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 */
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);
```

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

```
      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);
      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("'H\n");
     } else printf("%s\n", trap_data);
  }
  if (packet) send_packet(packet);
  else printf("Could not make trap packet\n");
```

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

```
}

#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));
   }
#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 */
       }
   }
}
```

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

```
/*  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) {
    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);
}
```

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

```
#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;
   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                            /* for classic K&R C */
static void check_arguments(argc, argv)     /* check arguments */
```

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

```
int   argc;
char *argv[];
#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]);
            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;
#if defined(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) {
```

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

```
         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 */
   if (cname == NULL) cname = PUBLIC_COMMUNITY_NAME; /* use default */
#if defined(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,"");
#if defined(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);
```

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

```
   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;

   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 */
#if defined(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*/
```

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

```
       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(SNMPIUCVNAME, siu.siucv_name, min(8,strlen(SNMPIUCVNAME)));
       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;
       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);
#if defined(VM) || defined (MVS)
    }
#endif
    if (dpi_fd < 0) {                           /* exit on error */
       DO_ERROR("init_connection:  socket");
       exit(1);
    }
    rc = connect(dpi_fd, sa, sasize);           /* connect to agent */
    if (rc != 0) {                              /* exit on error */
       DO_ERROR("init_connection:  connect");
       close(dpi_fd);
       exit(1);
    }
}

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

   if (len == 0) printf("     empty buffer\n"); /* buffer is empty */
   for (i=0;i<len;i++) {                    /* loop through buffer */
       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 H, "Related protocol specifications," on page 991 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 "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**, included 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 *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 H, "Related protocol specifications," on page 991 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.

See the following related information.
- "DPI subagent GETNEXT processing" on page 53
- "DPI subagent UNREGISTER request" on page 55
- "DPI subagent TRAP request" on page 54
- "DPI subagent CLOSE request" on page 56
- "Overview of subagent processing" on page 107
- "SNMP DPI: Connecting to the agent" on page 109
- "SNMP DPI: Registering a subtree with the agent" on page 111
- "SNMP DPI: Processing requests from the agent" on page 113
- "SNMP DPI: Processing a GET request" on page 116
- "SNMP DPI: 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_lDPI.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_lDPI.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.
6. 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 *z/OS Communications Server: IP Configuration Reference* or the Managing TCP/IP network resources with SNMP information in *z/OS Communications Server: IP System Administrator's Commands*.

Compiling and linking DPI Version 2.0

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 topics is intended as general guidance.

Compiling and linking DPI Version 2.0: 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 example shoes 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.

Compiling and linking DPI Version 2.0: 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_lDPI.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.

Migrating your SNMP DPI subagent to Version 2.0

The information presented in this topic *are guidelines and are not exact procedures*. Your specific implementation will vary from the guidelines presented.

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 for migrating your SNMP DPI subagent to Version 2.0

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 mkDPIregister() calls and pass the parameters according to the new function prototype. You must also expect a RESPONSE to the REGISTER request.

  See "The mkDPIregister() function" on page 67 for more information.
- Change mkDPIset() and mkDPIlist() calls to the new mkDPIset() call. Basically all mkDPIset() calls are now of the DPI Version 1.1 mkDPIlist() form.

  See "The mkDPIset() function" on page 71 for more information.
- Change mkDPItrap() and mkDPItrape() calls to the new mkDPItrap() call. Basically all mkDPItrap() calls are now of the DPI Version 1.1 mkDPItrape() form.

  See "The mkDPItrap() 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 mkDPIunregister() 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 "DPI subagent SET processing" on page 52.

  See the sample handling of SET/COMMIT/UNDO in "SNMP DPI: Processing a SET/COMMIT/UNDO request" on page 122.

Recommended actions for migrating your SNMP DPI subagent to Version 2.0

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 *z/OS Communications Server: IP Configuration Reference*.

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

snmp_dpi_xxxx_packet structures 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.

SNMP DPI subagent programming concepts

When implementing a subagent, use the DPI Version 2 approach and keep the following information 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.

See the following related information.

"DPI subagent example" on page 107

Specifying the SNMP DPI API

The following topics 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

DPI subagent 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 path name (for UNIX) that allows the socket connect to occur.

See the following related information.

"SNMP DPI: Connecting to the agent" on page 109

DPI subagent 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 that is being handled by this subagent.

  The agent can have an absolute maximum timeout value which is used if the subagent asks for too large a timeout value. The value 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. If a specific subtree needs more 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. The value 0 means that 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 value 0 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 EBCDIC character set on MVS.

  If your platform is EBCDIC-based, using the native EBCDIC character set 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 from EBCDIC to ASCII NVT 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 "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 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.

See the following related information.

"SNMP DPI: Connecting to the agent" on page 109

DPI subagent REGISTER request

Before a subagent receives any requests for MIB objects, it must first register with the SNMP agent the variables or subtree that it supports. 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 that specifies 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 that 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 the object number 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 that is used if the subagents ask for too large a timeout value. The value 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 error might be caused by the result of one of the following situations:

- 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 receive the higherPriorityRegistered error code.

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

See the following related information.

"SNMP DPI: Registering a subtree with the agent" on page 111

DPI subagent 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 nonexisting object or a nonexisting 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 values should be 0.

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

See the following related information.

"SNMP DPI: Processing a GET request" on page 116

"The mkDPIresponse() function" on page 69

DPI subagent SET processing

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

"SNMP DPI: Processing a SET/COMMIT/UNDO request" on page 122

DPI subagent 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
```

DPI subagent 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.

See the following related information.

"DPI subagent GETNEXT processing" on page 53

DPI subagent 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.

See the following related information.

"SNMP DPI: Generating a TRAP" on page 126

DPI subagent 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.

DPI subagent 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 that is to be unregistered. Object level unregistration requires a trailing period, which is following the object number, indicating an unregister request to all supported instances of an object (for example, ifDescr). Object level unregistration requires that 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 be done only 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 adding the instance number after the object number when building the unregistration packet using the mkDPIunregister call. For example, passing the object number 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 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 contains a reason code for the UNREGISTER request. 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.

See the following related information.
"SNMP DPI: Processing an UNREGISTER request" on page 125

DPI subagent 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, the subagent 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 contains 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.

See the following related information.
"SNMP DPI: 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:

DPI Transport-Related Functions:

Data Structures:

Constants and Values:

Related Information:

Basic DPI API functions

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

The Basic DPI Functions are:

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(snmp_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 mkDPIresponse() 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 = pDPIpacket(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 = mkDPIresponse(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 fDPIparse() function" on page 61
"The snmp_dpi_set_packet structure" on page 97
"The mkDPIresponse() 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 mkDPIopen() function

Format

```
#include <snmp_dpi.h>

unsigned char *mkDPIopen(       /* 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 mkDPIopen 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 *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 "DPI OPEN 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

"DPI OPEN 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_xxx*/
  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(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 */
```

Context

"The pDPIpacket() function" on page 76
"The snmp_dpi_hdr structure" on page 93
"The snmp_dpi_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_xxx*/
  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 that follows 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 of DPI SNMP value types" 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(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 = 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 topic 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

Format

```
#include <snmp_dpi.h>

int DPIawait_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 DPIconnect_to_agent_TCP() function

Format

```
#include <snmp_dpi.h>

int DPIconnect_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 DPIconnect_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 DPIconnect_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 DPIconnect_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® path name 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 path name. 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 sends an SNMP GET request to the SNMP agent to retrieve the path name 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.

The DPIconnect_to_agent_UNIXstream() function uses a path name in the z/OS UNIX file system as the name of the socket for the connect. This path name 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 uses the default name

/var/dpi_socket if you do not supply another name in the agent startup parameter (-s) or in the OSNMPD.DATA file. Whichever name is used, the SNMP agent creates the path name as a character special file during initialization.

You must either define the subagents with superuser authority or set the read and write file access permission bits for the path name for the class associated with the user ID of the subagent before subagents can successfully connect to the agent using the path name. You can use the agent -C startup parameter to specify which permission bits should be set.

To run a user-written subagent from a non-privileged user ID, set the permission bits for the character special file to ***write*** access. Otherwise, a subagent using this function must 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 DPIsend_packet_to_agent() function

Format

```
#include <snmp_dpi.h>

int DPIsend_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 DPIsend_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 DPIconnect_to_agent_TCP() function" on page 80
"The DPIconnect_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 topic describes each data structure that is used in the SNMP DPI API:

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 that follows 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 that follows 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 respons 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 that follows 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 of DPI SNMP value types" 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 of DPI SNMP value types" 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 of DPI SNMP value types" on page 104

DPI OPEN character set selection

The version of DPI Version 2.0 included 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 *mkDPIxxxx(...)* 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*.

See the following related information.
"The mkDPIopen() function" on page 65

SNMP DPI constants, values, return codes, and include file

This topic 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

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.

See the following 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.

See the following 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 *mkDPIresponse()* function and in the *error_code* field in the *snmp_dpi_resp_packet* structure.

See the following related information.

"The snmp_dpi_resp_packet structure" on page 96
"The mkDPIresponse() 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 *mkDPIunregister()* function and in the *reason_code* field in the *snmp_dpi_ureg_packet* structure.

See the following related information.

"The snmp_dpi_ureg_packet structure" on page 99
"The mkDPIunregister() function" on page 75

DPI SNMP value types

These are the currently defined value types as 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.

See the following related information.

"The snmp_dpi_set_packet structure" on page 97
"The mkDPIset() function" on page 71
"Value representation of DPI SNMP value types" on page 104
"Value ranges and limits for DPI SNMP value types" on page 105

Value representation of DPI SNMP value types

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 "1.3.6.1.2.1.1.1.0" which is sysDescr.0.

  An EBCDIC example:

  ```
  'f14bf34bf64bf14bf24bf14bf14bf14bf000'h
  ```

  represents "1.3.6.1.2.1.1.1.0" 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.

See the following related information.

"Value ranges and limits for DPI SNMP value types" on page 105

Value ranges and limits for DPI SNMP value types

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 H, "Related protocol specifications," on page 991 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 receivd 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.

See the following 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>
```

snmp_dpi.h include parameters

None

snmp_dpi.h include 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.

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
- "DPI OPEN character set selection" on page 101

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 might be differences in the code presented and the code that is included 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
- "SNMP DPI: Connecting to the agent" on page 109
- "SNMP DPI: Registering a subtree with the agent" on page 111
- "SNMP DPI: Processing requests from the agent" on page 113
- "SNMP DPI: Processing a GET request" on page 116
- "SNMP DPI: Processing a GETNEXT request" on page 119
- "SNMP DPI: Processing a SET/COMMIT/UNDO request" on page 122
- "SNMP DPI: Processing an UNREGISTER request" on page 125
- "SNMP DPI: Processing a CLOSE request" on page 126
- "SNMP DPI: Generating a TRAP" on page 126

See the following related information.
"SNMP DPI 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 might be a bit different.

In this overview, the agent is requested to send at most one varBind per DPI packet, so there is no need to loop through a list of varBinds. You might gain performance improvements if you allow for multiple varBinds per DPI packet on GET, GETNEXT, SET requests. To allow multiple varBinds, your code must loop through the varBind list, which makes the situation more complicated. The DPI subagent programmer can handle that when you understand the basics of the DPI API.

The following sample shows 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;
#ifndef 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  */
```

SNMP DPI: 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 can 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 */

#ifndef 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 /* ndef 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() */
```

SNMP DPI: 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() */
```

SNMP DPI: Processing requests from the agent

After registering your sample MIB subtree with the agent, expect that SNMP requests for that subtree are passed back to you for processing. Since the requests 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;
#ifndef 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];
#ifndef 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];
#ifdef INCLUDE_UNIX_DOMAIN_FOR_DPI
          } else if (strcmp(argv[i],"-unix") == 0) {
             unix_sock = 1;
#endif /* 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 */

#ifndef DPI_VERY_MINIMAL_SUBAGENT
       if (debug) {
          printf("\n%s - %s\n",__FILE__, VERSION);
          DPIdebug(debug);                /* turn on DPI dubugging   */
          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");          /* @L1A*/
    exit(1);                                          /* @L1A*/
  }
  else {                            /* gethostname worked  @L1A*/
    hostname_p = hostname;                            /* @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() */
```

SNMP DPI: 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 mkDPIresponse() 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(snmp_dpi_hdr *hdr_p, snmp_dpi_get_packet *pack_p)
{
       unsigned char       *packet_p;
       int                  rc;
       snmp_dpi_set_packet *varBind_p;
       char                *i_p;

       varBind_p =                        /* init the varBind chain  */
          snmp_dpi_set_packet_NULL_p;     /* 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(            /* 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(               /* 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(               /* 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         */

#ifndef EXCLUDE_SNMP_SMIv2_SUPPORT
        } else if (i_p && (strcmp(i_p,"4.0") == 0)) {
          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_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(               /* 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(               /* 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(          /* 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_get() */
```

SNMP DPI: 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 mkDPIresponse() 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 =                        /* init the varBind chain  */
          snmp_dpi_set_packet_NULL_p;     /* 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(               /* 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(             /* 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(               /* 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;
#ifndef 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;                            /*                *Apr23*/
#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() */
```

SNMP DPI: 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 =                         /* init the varBind chain  */
          snmp_dpi_set_packet_NULL_p;      /* 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 =                       /* allocate memory for     */
          malloc(pack_p->value_len);    /* 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() */
```

SNMP DPI: 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() */
```

SNMP DPI: 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() */
```

SNMP DPI: 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 =                        /* init the varBind chain  */
          snmp_dpi_set_packet_NULL_p;     /* to a NULL pointer       */

       varBind_p = mkDPIset(              /* Make DPI set packet     */
                    varBind_p,            /* ptr to varBind chain    */
                    DPI_SIMPLE_MIB,       /* 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            */

       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    */
        }

    #ifndef EXCLUDE_SNMP_SMIv2_SUPPORT
                                            /*                *Apr23*/
        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 topic 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 H, "Related protocol specifications," on page 991 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 SNMP DPI client 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 that can be used 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, which has resulted 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 159 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 about the specific cause of the error. See "Debugging the SNMP manager API" on page 159 for information about 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 that were used to create the outgoing packet.

Configuration entry considerations

Configuration entries represent either SNMP agents to which requests are to be sent, or notification receivers. The snmpInitialize() function, which is the first function your SNMP manager must invoke, can create configuration entries if you specify a configuration file to be processed by this function. The function returns the configuration entries as a linked-list of SnmpConfigEntry structures, which are

built from the information in the configuration file. If you do not specify a configuration file, your SNMP manager application must manually create the configuration entries by using the SnmpConfigEntry structure. This structure is required by other SNMP manager API functions. It is defined in the SNMP manager API header file, <snmpmgr.h>, which is available in the /usr/include directory. This structure supports both community-based and SNMPv3 user-based security.

Requirement: If your SNMP manager application manually creates the SnmpConfigEntry structures and the application sets the functionsRequested input parameter to the snmpInitialize() function to request the functional support of the current release, then the application must perform the following tasks:

- Ensure that reserved fields in the structure are set to binary zeros.
- Set the cfgVersion field in the structure to the SNMP_CONFIGENTRY_VER constant.
- Set the cfgLen field in the structure to the SNMP_CONFIGENTRY_LEN constant.

A sample manager implementation, snmpSMgr.c, is found in the usr/lpp/tcpip/samples directory.

Note: The functions snmpValueCreateBits, snmpValueCreateUnsignedInteger32, and snmpValueCreateNSApAddr have been removed. If these functions are used, SNMP_MGR_RC_DEPRECATED is returned.

snmpAddVarBind – Adds a VarBind to the SnmpVarBinds structure

```
#include <snmpmgr.h>
int snmpAddVarBind(SnmpVarBinds **varbinds, const char *oid, const smiValue *value)
```

snmpAddVarBind 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 space is not available in the SnmpVarBinds structure, storage space is reallocated and the SnmpVarbinds structure is updated.

snmpAddVarBind 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.

snmpAddVarBind 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)
```

snmpBuildPDU 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.

snmpBuildPDU 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.

snmpBuildPDU 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 Snmp PDU 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)
```

snmpBuildSession 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.

snmpBuildSession 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.

snmpBuildSession result

- SNMP_MGR_RC_OK if successful.
- SNMP_MGR_RC_NULL_PTR if the *configEntry* or *snmpSession* parameter of this function 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.
- SNMP_MGR_RC_INVALID PARAMETERS if the cfgLen or cfgVersion field of the *configEntry* parameter does not match the required value, as defined in the SNMP Manager API header file, <snmpmgr.h>.
- 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 and the internal return code value. Your IBM service representative uses this internal return code to help solve your error.

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)
```

snmpCreateVarBinds 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.

snmpCreateVarBinds 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.

snmpCreateVarBinds 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)
```

snmpFreeDecodedPDU description

This function frees the storage that was allocated for the decoded PDU by the snmpSendRequest() function.

snmpFreeDecodedPDU parameters

decodedPDU

This input parameter is a pointer to the decoded PDU that is returned by the snmpSendRequest() function. This parameter is required.

snmpFreeDecodedPDU 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)
```

snmpFreeOID description

This function frees the storage that was allocated for the OID string by the snmpGetOID() function.

snmpFreeOID parameters

oidString
This input parameter is a pointer to the OID string that is returned by the snmpGetOID() function. This parameter is required.

snmpFreeOID 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)
```

snmpFreePDU description

Frees the storage for the SNMP PDU, which was allocated by the snmpBuildPDU() function.

snmpFreePDU parameters

pdu This input parameter is the address of the PDU that is to be freed. This parameter is required.

snmpFreePDU 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)
```

snmpFreeVarBinds 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.

snmpFreeVarBinds parameters

varbinds
This input parameter specifies the address of the SnmpVarBinds structure that is to be freed. This parameter is required.

snmpFreeVarBinds 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)
```

snmpGetErrorInfo description

This function is used to retrieve the *error_index* and *error_status* values from a decoded response PDU.

snmpGetErrorInfo 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.

snmpGetErrorInfo 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 snnmpSendRequest() 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)
```

snmpGetNumberOfVarBinds 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.

snmpGetNumberOfVarBinds 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.

snmpGetNumberOfVarBinds 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)
```

snmpGetOID description

This function retrieves the OID value from a VarBind structure and returns the OID value as a string.

snmpGetOID parameters

arbind 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.

snmpGetOID 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)
```

snmpGetRequestId description

This function is used to retrieve the *requestId* field from a decoded response PDU.

snmpGetRequestId 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.

snmpGetRequestId 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)
```

snmpGetSockFd description

This function retrieves the socket's file descriptor, which is stored in the session.

snmpGetSockFd 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.

snmpGetSockFd 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 snnmpBuildSession() 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)
```

snmpGetValue description

This function retrieves the value from a VarBind structure and returns the value in an smiValue structure.

snmpGetValue 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.

snmpGetValue 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)
```

snmpGetVarbind description

Given an index into a PDU's array of VarBind structures, this function completes the contents of the input VarBind structure.

snmpGetVarbind 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.

snmpGetVarbind 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)
```

snmpInitialize 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 SnmpConfigEntry structure as input to the snmpBuildSession() function.

snmpInitialize parameters

functionsRequested
: This input parameter specifies the functional support requested from the SNMP manager API. The following values can be specified:
  - SNMP_INIT_FUNCREQ_BASE - Basic functional support as of z/OS V1R11
  - SNMP_INIT_FUNCREQ_V1R12 - All functional support from current release

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 SnmpConfigEntry 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 SnmpConfigEntry structures, as defined in the snmpmgr.h file. Each SnmpConfigEntry 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.

snmpInitialize 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_INVALID PARAMETERS if the functionsRequested parameter is not SNMP_INIT_FUNCREQ_BASE (0) or SNMP_INIT_FUNCREQ_V1R12 (1).
- 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 an SnmpConfigEntry 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 SnmpConfigEntry structures, you must free their storage by calling the snmpTerminate() function.
- If your functionsRequested parameter is 1, each of the SNMPv3 entries in your configuration file must contain a value for the authEngineID field.

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)
```

snmpSendRequest 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.

snmpSendRequest 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 try to retry receive the response PDU again 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 try to receive a response again. If you specify a value of 1, a PDU is not sent to the target agent.

snmpSendRequest 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)
```

snmpSetLogFunction 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.

snmpSetLogFunction parameters

funcName

This input parameter specifies the SNMP manager's preferred logging function.

snmpSetLogFunction 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)
```

snmpSetLogLevel description

This function sets the log level for messages. See “Debugging the SNMP manager API” on page 159 for more information about logging.

snmpSetLogLevel 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)
- SNMP_LOG_INTERNAL (8)

snmpSetLogLevel 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)
```

snmpSetRequestId 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.

snmpSetRequestId 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.

snmpSetRequestId 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(SnmpConfigEntry *headEntry)
```

snmpTerminate description

This function releases the resources that were allocated by the snmpInitialize() function.

snmpTerminate parameters

headEntry

This input parameter points to the first entry in the linked list of SnmpConfigEntry objects returned by the snmpInitialize() function.

snmpTerminate 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(SnmpSession *snmpSession)
```

snmpTerminateSession description

This function terminates an SNMP session and releases all of the resources held by that session.

snmpTerminateSession parameters

snmpSession

This input parameter is the address of the session that was created by the snmpBuildSession() call.

snmpTerminateSession result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *snmpSession* parameter is NULL
- SNMP_MGR_RC_SOCK_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.

snmpValueCreateCounter32 – Create an smiValue of type Counter32

```
#include <snmpmgr.h>
int snmpValueCreateCounter32(smiValue *value, smiUINT32 inInt)
```

snmpValueCreateCounter32 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.

snmpValueCreateCounter32 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.

snmpValueCreateCounter32 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)
```

snmpValueCreateCounter64 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.

snmpValueCreateCounter64 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.

snmpValueCreateCounter64 result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter is NULL

snmpValueCreateGauge32 – Create an smiValue of type Gauge32

```
#include <snmpmgr.h>
int snmpValueCreateGauge32(smiValue *value, smiUINT32 inInt)
```

snmpValueCreateGauge32 description

This function completes an smiValue structure of type SNMP_SYNTAX_Gauge32, based on the input integer. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

snmpValueCreateGauge32 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.

snmpValueCreateGauge32 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)
```

snmpValueCreateInteger 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.

snmpValueCreateInteger 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.

snmpValueCreateInteger 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)
```

snmpValueCreateInteger32 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.

snmpValueCreateInteger32 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.

snmpValueCreateInteger32 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)
```

snmpValueCreateIPAddr 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.

snmpValueCreateIPAddr 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.

snmpValueCreateIPAddr 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)
```

snmpValueCreateNull 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.

snmpValueCreateNull parameters

value This output parameter is the address of the smiValue structure to be completed. This parameter is required.

snmpValueCreateNull 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)
```

snmpValueCreateOctet 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.

snmpValueCreateOctet 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.

snmpValueCreateOctet 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)
```

snmpValueCreateOID 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.

snmpValueCreateOID 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.

snmpValueCreateOID 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)
```

snmpValueCreateOpaque 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.

snmpValueCreateOpaque 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.

snmpValueCreateOpaque 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)
```

snmpValueCreateTimerTicks 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.

snmpValueCreateTimerTicks 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.

snmpValueCreateTimerTicks result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if this function's *value* parameter is NULL

snmpValueCreateUnsigned32 – Create an smiValue of type Unsigned32

```
#include <snmpmgr.h>
int snmpValueCreateUnsigned32(smiValue *value, smiUINT32 inInt)
```

snmpValueCreateUnsigned32 description

This function fills in an smiValue structure of type SNMP_SYNTAX_UNSIGNED32, based on the input integer. This smiValue structure can then be used as input to the snmpCreateVarBinds() or snmpAddVarBind() function.

snmpValueCreateUnsigned32 parameters

value This output parameter is the address of the smiValue structure to be filled in. This parameter is required.

inInt This input parameter is the unsigned integer to be stored in the value. This parameter is required.

snmpValueCreateUnsigned32 result

- SNMP_MGR_RC_OK if successful
- SNMP_MGR_RC_NULL_PTR if the *value* parameter of this function 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 is 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);)
```

snmpBuildV1TrapPDU 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.

snmpBuildV1TrapPDU 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.

snmpBuildV1TrapPDU 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 an 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)
```

snmpBuildV2TrapOrInformPDU 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.

snmpBuildV2TrapOrInformPDU 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.

snmpBuildV2TrapOrInformPDU 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

You can create a configuration information file for use with the snmpInitialize() function of the SNMP manager API. The configuration statements can be defined and stored in a z/OS UNIX file or an MVS data set. See "snmpInitialize – Initialize the manager environment" on page 142 for more information about using this file. Following is a sample 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  authEngineID
#
#-------------------------------------------------------------------------------------------------------------------------
#  User-based security (SNMPv3)
#-------------------------------------------------------------------------------------------------------------------------
9.8.0.1    162     snmpv3    userid   -    AuthPriv   HMAC-SHA  f40b19aa7c2d3b685655ba74d7771522faa3571c  DES
f40b19aa7c2d3b685655ba74d7771522faa3571   8000000205092a67b63698ec
```

SNMP manager API statement syntax

This section describes the configuration entry parameters. The term target is a target SNMP agent or to an application that receives a trap or inform sent by an SNMP manager application.

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
SNMPV3 user-based security (USM)

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 USM 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)

authEngineID
This parameter is only valid for SNMPv2 traps with USM security and is required only when the functionsRequested parameter on the snmpInitialize() call is not 0. If you specify this parameter, it represents the authoritative engine ID to be used to send a trap. A valid authEngineID is a string of 10-64 (must be an even number) hexadecimal digits. By default, the engine identifier is created by using a vendor-specific formula and incorporates the IP address of the manager. However, a customer can choose to use any engine identifier that is consistent with the snmpEngineID definition in RFC 3411 and that is also unique within the administrative domain. A dash (-) indicates the generated default value. See Appendix H, “Related protocol specifications,” on page 991 for information about accessing RFCs.

SNMP manager API 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.

   To enable the use of MVS-specific data structures, you must define the MVS constant using either a compilation option (-DMVS) or a compiler directive (#define MVS) in your 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, <snmpntfy.h>, which is also available in the /usr/include directory.
2. Compile your application using the DLL compiler option. See *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 SNMP manager API 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 LNKLST 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 except SNMP_LOG_INTERNAL

Table 3. SNMP manager API debug levels (continued)

Debug level	Description
SNMP_LOG_INTERNAL (8)	Log all traces for packet processing

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 SNMP manager API 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 H, "Related protocol specifications," on page 991 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 can accept or deny the request, depending upon the availability of resources. In the case of failure, the RSVP agent returns 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 can 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 can 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 H, "Related protocol specifications," on page 991 for information about accessing RFCs) contains a definition and explanation of the different reservation styles. Also see *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 can then be used to register as a data sender, and the rapi_reserve() call can be used to make a QoS reservation as a data receiver.

The rapi_sender() or rapi_reserve() calls can 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 can be established for the same RSVP session. For example, if an application sends multiple UDP data flows that are 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 is 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 *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 *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 */
);
```

rapi_event_rtn_t description

This is the template for the function address that is supplied on the rapi_session call. The event upcall function is invoked from the asynchronous event mechanism when an event occurs.

rapi_event_rtn_t 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 in “rapi_event_rtn_t result” on page 166.

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 “RAPI error handling” on page 179 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 that contains a sequential vector of RAPI *filter spec* or *sender template* objects. The number of objects in this vector is specified in *FilterSpecNo*. If the *FilterSpecNo* value is 0, the *FilterSpec_list* parameter value is NULL.

Flowspec_list, FlowspecNo
The *Flowspec_list* parameter is a pointer to an area that contains a sequential vector of RAPI *flowspec* or *Tspec* objects. The number of objects in this vector is specified in *FlowspecNo*. If the *FlowspecNo* value is 0, the *Flowspec_list* parameter value is NULL.

Adspec_list, AdspecNo
The *Adspec_list* parameter is a pointer to an area the contains a sequential vector of RAPI *adspec* objects. The number of objects in this vector is specified in *AdspecNo*. If the *AdspecNo* value is 0, the *Adspec_list* parameter value is NULL.

Event_arg
This is the value that is supplied in the rapi_session() call.

rapi_event_rtn_t 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 can 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 is triggered whenever the path state changes.

FilterSpec_list, *Flowspec_list*, and *Adspec_list* are of equal length, and corresponding entries contain *sender templates*, *sender Tspecs*, and *Adspecs*, respectively, for all senders known at this node. A missing object is generally indicated by an empty RAPI object.

RAPI_PATH_EVENT events are enabled by the initial rapi_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 are 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* contain zero or more corresponding *filter spec* objects. *Adspec_list* is empty.

RAPI_RESV_EVENT upcalls are enabled by a rapi_sender() call; the *sender template* from the latter call matches 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 specify the error. *FilterSpec_list* and *Flowspec_list* each contain one object, the *sender template* and corresponding *sender Tspec* (if any) in error, while *Adspec_list* is empty. If there is no *sender Tspec*, the object in *Flowspec_list* is an empty RAPI object. The *Adspec_list* is empty.

RAPI_PATH_ERROR events are enabled by a rapi_sender() call, and the *sender Tspec* in that call matches 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 specify the error. *Flowspec_list* contains one *flowspec*, while *FilterSpec_list* can contain zero or more corresponding filter specs. *Adspec_list* is 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	rapi_session	n	n	n
RAPI_PATH_ERROR	rapi_sender	1	1	0
RAPI_RESV_EVENT	rapi_sender	n	1 or 0	0
RAPI_RESV_ERROR	rapi_reserve	n	1	0
RAPI_RESV_CONFIRM	rapi_reserve	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 "RAPI header files" on page 181 for more information on header files.

rapi_release - Remove a session

```
#include <rapi.h>


int rapi_release (rapi_sid_t Sid)
```

rapi_release 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.

rapi_release parameters

Sid This parameter is the session ID for the session initiated by a successful rapi_session() call.

rapi_release 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      */
)
```

rapi_reserve description

The rapi_reserve() function is called to make, modify, or delete a resource reservation for a session. The call can be repeated with different parameters, allowing the application to modify or remove the reservation; the latest call will take precedence.

rapi_reserve 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 can 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 can point to a list of any length.

rapi_reserve result

Depending upon the parameters, each call might or might not result in new *admission control* calls, which could fail asynchronously.

If there is a synchronous error in this call, rapi_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 might 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.

rapi_sender - Specify sender parameters

```
#include <rapi.h>


int rapi_sender(
    rapi_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 */
)
```

rapi_sender description

An application must issue a rapi_sender() call if it intends to send a flow of data for which receivers can make reservations. This call defines, redefines, or deletes the parameters of that flow. A rapi_sender() call can 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 can receive upcalls of type RAPI_RESV_EVENT or RAPI_PATH_ERROR.

rapi_sender 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 can point to a rapi_addr_t structure specifying the IP source address and, if applicable, the source port from which data is sent, or it can be NULL.

If the IP source address is INADDR_ANY, the API uses the default IP address of the local host. This is sufficient unless the host is multihomed. The port number can 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 can be a pointer to a RAPI filter specification structure specifying the format of data packets to be sent, or it can 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 can 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.

rapi_sender 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*/
)
```

rapi_session description

The rapi_session() call creates an API session.

After a successful rapi_session() call has been made, the application can receive upcalls of type RAPI_PATH_EVENT for the API session.

rapi_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.

rapi_session 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.

rapi_session 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)
```

rapi_version description

This call obtains the version of the interface. It can be used by an application to adapt to different versions.

rapi_version 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 * major + 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_tspec()

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    */
)
```

rapi_fmt_adspec 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.

rapi_fmt_adspec 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.

rapi_fmt_adspec 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      */
)
```

rapi_fmt_filtspec 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.

rapi_fmt_filtspec 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.

rapi_fmt_filtspec 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      */
)
```

rapi_fmt_flowspec 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.

rapi_fmt_flowspec 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.

rapi_fmt_flowspec 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    */
)
```

rapi_fmt_tspec 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.

rapi_fmt_tspec 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.

rapi_fmt_tspec 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 H, “Related protocol specifications,” on page 991 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.

RAPI objects - Flowspecs

There are two formats for RAPI *flowspecs*. For further details, see "The <rapi.h> header" on page 182.

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.

RAPI_FLOWSTYPE_Intserv upcalls

In an upcall, a *flowspec* 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.

RAPI objects - Sender tspecs

There are two formats for RAPI *Sender Tspecs*. For further details, see "The <rapi.h> header" on page 182.

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.

RAPI_TSPECTYPE_Intserv 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 rapi_session() call, then the *IntServ* format is used in upcalls.

RAPI objects - Adspecs

There are two formats for RAPI *adspecs*. For further details, see "The <rapi.h> header" on page 182.

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.

RAPI_ADSTYPE_Intserv 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.

RAPI objects - 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.

RAPI 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)
```

rapi_dispatch 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 can be called at any time, but it will generally have no effect unless there is a pending event.

rapi_dispatch parameters

There are no parameters to this call.

rapi_dispatch result

Calling this routine can 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 180 for a list of error codes.

rapi_getfd - Get file descriptor

```
#include <rapi.h>


int rapi_getfd (rapi_sid_t  Sid)
```

rapi_getfd description

After a rapi_session() call has completed successfully and before rapi_release() has been called, the application can 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().

rapi_getfd parameters

Sid This parameter is the session ID for the session initiated by a successful rapi_session() call.

rapi_getfd result

If *Sid* is illegal or undefined, this call returns -1; otherwise, it returns the file descriptor.

RAPI 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 H, "Related protocol specifications," on page 991 for information about accessing RFCs.

RAPI error codes

[RAPI_ERR_OK]
No error

[RAPI_ERR_INVAL]
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 can be indicated by *errno*.

[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_BADVDPORT]
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

RAPI header files

The following topics apply to RAPI header files.

RAPI header files: Integer and floating point types

Types *u_int8_t*, *u_int16_t* and *u_int32_t*, which appear in the <rapi.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 <rapi.h> header file.

The <rapi.h> header

This header file contains the definitions of the RSVP API (RAPI) library calls.

Inclusion of this header can make available other symbols in addition to those specified in this topic.

<rapi.h> header general definitions

The following general definitions apply to the <rapi.h> header:

- Macro RAPI_VERSION is defined with value 100 * *major* + *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
QOS_CNTR_LOAD	Controlled-load service
QOS_GUARANTEED	Guaranteed service
QOS_TSPEC	Generic Tspec

- Enumeration *rapi_format_t* is defined by typedef and has at least the following members:

Member	Meaning
RAPI_ADSTYPE_Intserv	Int-Serv format adspec
RAPI_ADSTYPE_Simplified	Simplified format adspec
RAPI_EMPTY_OTYPE	Empty object
RAPI_FILTERFORM_BASE	Simple V4: Only *sockaddr*
RAPI_FLOWSTYPE_Intserv	Int-Serv format flowspec
RAPI_FLOWSTYPE_Simplified	Simplified format flowspec
RAPI_TSPECTYPE_Intserv	Int-Serv format (sndr)Tspec
RAPI_TSPECTYPE_Simplified	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
form	int	Format
len	unsigned int	Actual length in bytes

<rapi.h> header 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
spec_Tspec_r	float32_t	Token bucket average rate in bytes per second
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_tspbody_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

<rapi.h> header 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
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

<rapi.h> header 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

Member	Type	Usage
xClaspec_override	u_int8_t	See note (2)
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

Macro	Value
adspecbody_qosx	adsbody_u.adsx

<rapi.h> header 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

<rapi.h> header 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

<rapi.h> header 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*.

<rapi.h> header 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_INVAL
RAPI_ERR_MAXSESS
RAPI_ERR_MEMFULL
RAPI_ERR_N_FFS

Error code
RAPI_ERR_NORSVP
RAPI_ERR_NOTSPEC
RAPI_ERR_OBJLEN
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 181:

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

The following defines the integrated services (see RFC 2210) data formats. (See Appendix H, "Related protocol specifications," on page 991 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 can be circumstances in which this is not adequate. For example, more general forms of flowspec, containing more than one service, might 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 might change the definitions in this topic. Application writers are advised not to use these definitions except when absolutely necessary.

Notes:

1. The values in the data structures defined in this topic are in host byte order.
2. Inclusion of this header can make available other symbols in addition to those specified in this topic.

Integrated services data structures and macros 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 H, "Related protocol specifications," on page 991 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 H, "Related protocol specifications," on page 991 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
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)
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)->issh_len32b)
Non_Is_Hop	((IS_serv_hdr_t *)p)->issh_flags & ISSH_BREAK_BIT
Set_Break_Bit(p)	((IS_serv_hdr_t *)p)->issh_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)->issh_service = (s); \ (p)->issh_flags = 0; \ (p)->issh_len32b = wordsof(len); }

Integrated services data structures and macros 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 H, “Related protocol specifications,” on page 991 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
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

Integrated services data structures and macros 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))
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

Integrated services data structures and macros 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
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 can 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)

Integrated services data structures and macros 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)

Member	Type	Usage
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	
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_tspbody_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-bit 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 topic 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 201 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-bit 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 *z/OS XL C/C++ Programming Guide*.

X Window System programming considerations

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.

Porting Motif applications to z/OS UNIX MVS

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 the KeySym hexadecimal value 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 -W1,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 -W1,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 *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.

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.

   ```
   export DISPLAY=royal.csc.ibm.com:0.0
   ```

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 topics describe how this is accomplished.

EBCDIC/ASCII translation in the X Window System: 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.

EBCDIC/ASCII translation in the X Window System: 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

Client	Description
bmtoa	Bit map conversion utility
editres	Resource editor
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

X Window System and Motif files locations

The following topics 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 included 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-bit 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 included 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 information:

- 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 H, "Related protocol specifications," on page 991 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 *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*.
- For more information about z/OS UNIX System Services sockets, see *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 210 and Figure 3 on page 211 provide an overview of

the high-level RPC client and server processes from initialization through cleanup.

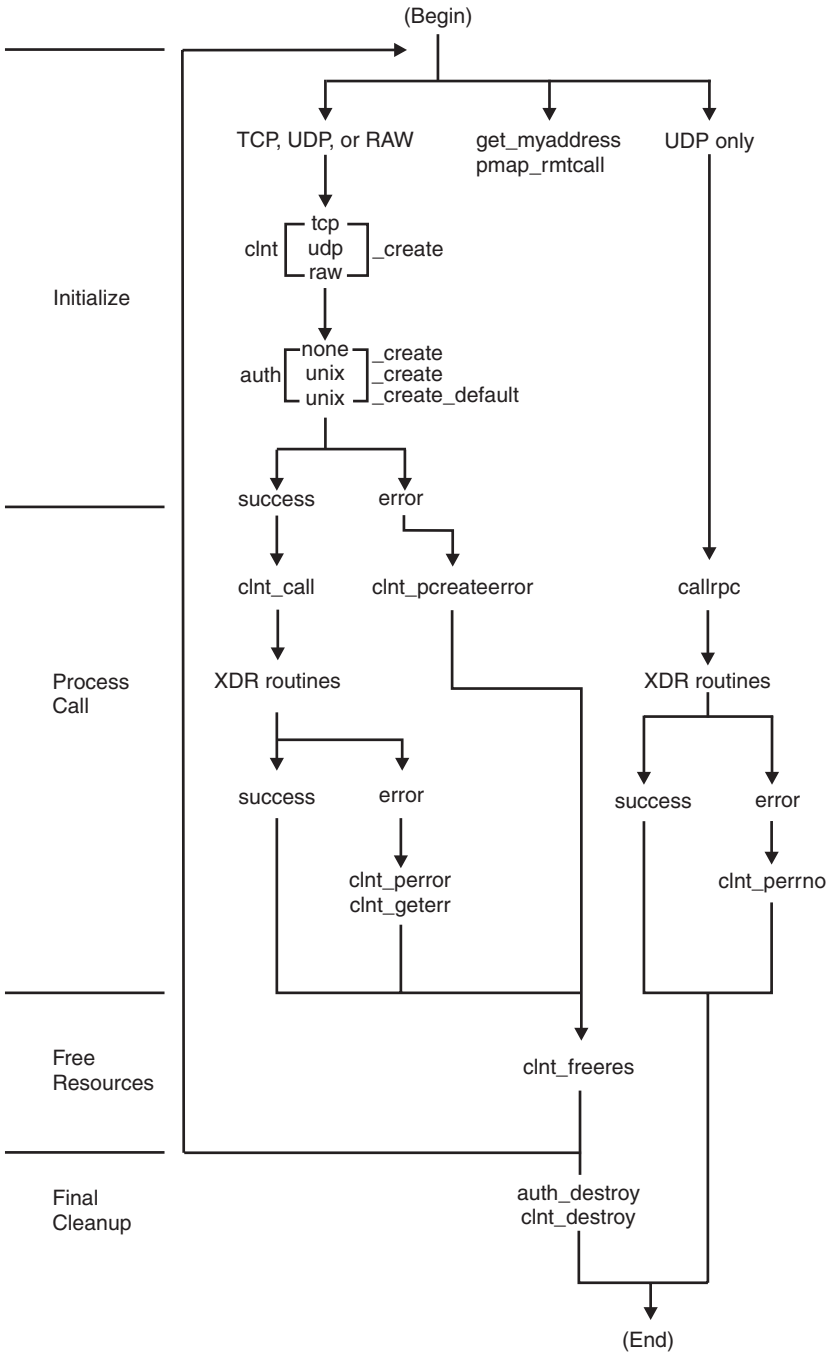


Figure 2. Remote procedure call (client)

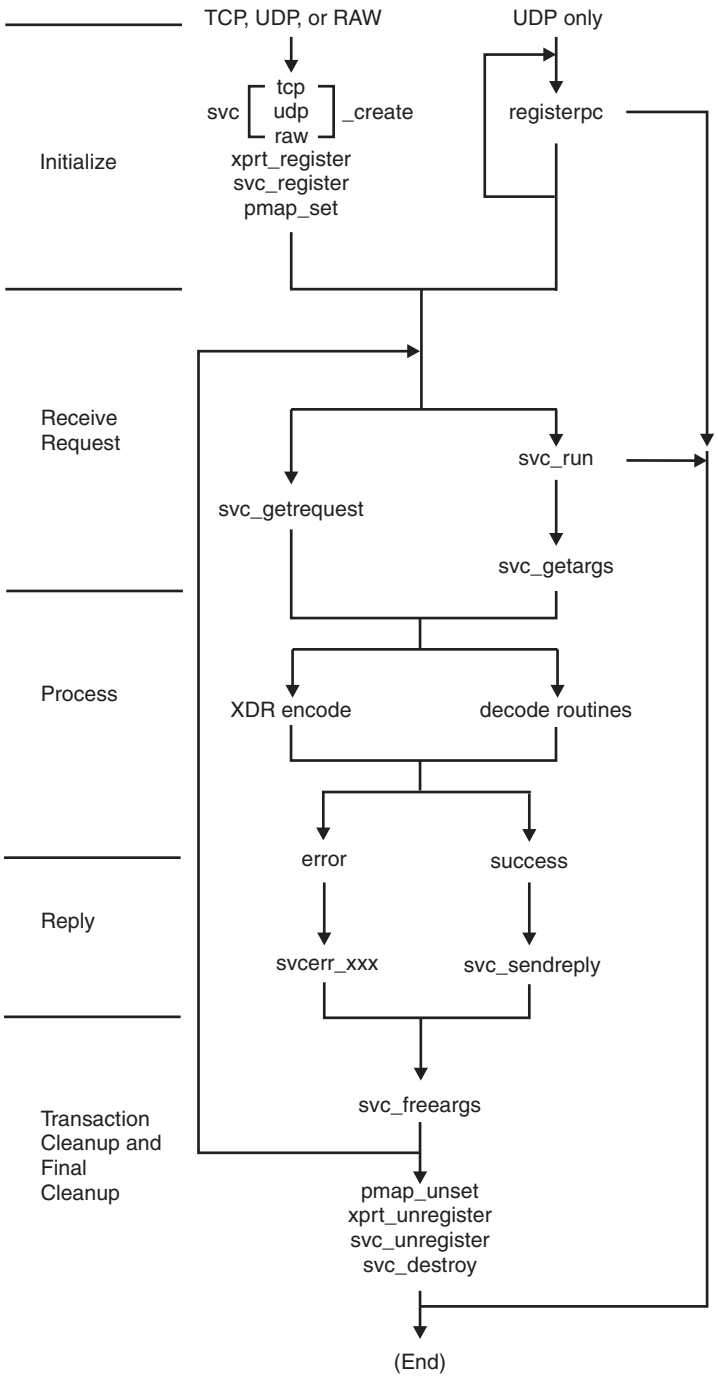


Figure 3. Remote procedure call (server)

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:9.1.1.1.1.4.0
- 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 A, "Well-known port assignments," on page 721, 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 *z/OS Communications Server: IP System Administrator's Commands*. For more information about rpcinfo and portmapper, see the Rpcinfo information in *z/OS Communications Server: IP System Administrator's Commands*.

Portmapper and rpcbind 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.

Requirements: The following apply when the rpcbind server runs on a multilevel secure host.

- The rpcinfo utility issues a target assistance request on behalf of the user who invoked it when rpcinfo is invoked with the -b parameter. When the SAF profile BPX.POE is defined on your host and rpcbind is started, the rpcbind user ID must be granted at least READ access to the profile to enable rpcbind to respond to **rpcinfo -b** requests.
- When the SAF profile BPX.POE is defined in class FACILITY and the rpcbind server is in use, the rpcbind user ID must be granted at least READ access to the profile to enable the server to support target assistance requests.

Rules:

- The target assistance RPCs: PMAPPROC_CALLIT, RPCBPROC_CALLIT, RPCBPROC_BCAST, and RPCBPROC_INDIRECT, are defined in RFC 1833: *Binding Protocols for ONC RPC*.
- The following RPC library routines issue target assistance requests on behalf of the calling application: pmap_rmtcall() and clnt_broadcast().

Registering with rpcbind

RPC applications register with rpcbind by sending an RPCBPROC_SET or PMAPPROC_SET RPC to rpcbind, or by invoking an RPC library routine that sends one of these RPCs to rpcbind on its behalf.

Requirements:

- Your registration request must originate from an IP address on the host where rpcbind is running.
- When the SAF profile EZB.RPCBIND.*sysname.rpcbindname*.REGISTRY is defined in the SERVAUTH class, the user ID that is associated with the RPC server that registers with rpcbind must be granted at least READ access to the profile. If your application sends a PMAPPROC_SET or RPCBPROC_SET request to rpcbind, you must grant the user ID that is associated with your application at least READ access to the profile when the profile is defined.
- 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 rcpb field values:

  – r_addr = 1.2.3.4.4.0 instead of ::FFFF:1.2.3.4.4.0
  – r_netid = tcp instead of tcp6
- 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
```

- When processing an RPCBPROC_SET request, rpcbind ignores the r_owner field of the input rpcb.

Deregistering with rpcbind

RPC applications deregister with rpcbind by sending an RPCBPROC_UNSET or PMAPPROC_UNSET RPC request to rpcbind, or by invoking an RPC library routine that sends one of these RPCs to rpcbind on its behalf.

Requirements:

- Your deregistration request must originate from an IP address on the local host.
- When the SAF profile EZB.RPCBIND.*sysname.rpcbindname*.REGISTRY is defined in the SERVAUTH class, the user ID that is associated with the RPC server that deregisters with rpcbind must be granted at least READ access to the profile. If your application sends a PMAPPROC_UNSET or RPCBPROC_UNSET request to rpcbind, you must grant the user ID that is associated with your application at least READ access to the profile when the profile is defined.

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 H, “Related protocol specifications,” on page 991 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 H, “Related protocol specifications,” on page 991 for information on accessing RFCs.

RPC servers in a CINET environment

Tip: This topic 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

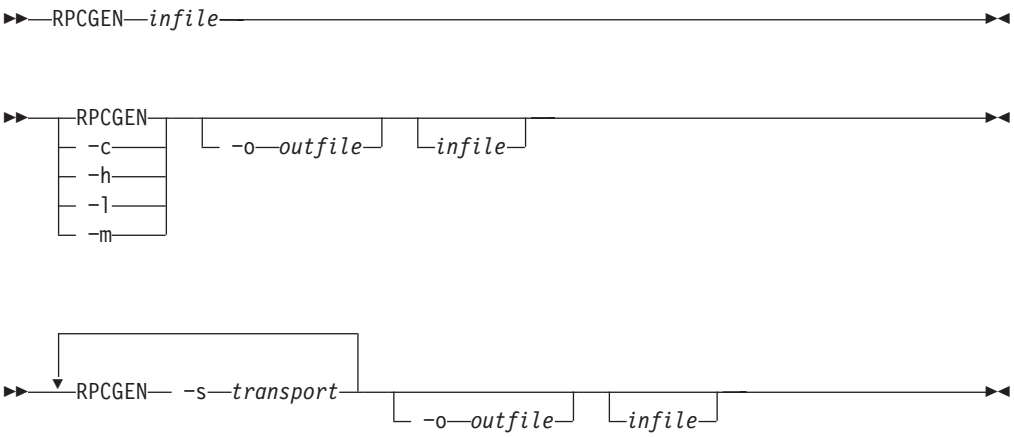
For more information about detecting a stack that is starting, see the setibmsockopt() -- Set IBM Specific Options Associated with a Socket information in *z/OS XL C/C++ Run-Time Library Reference*.

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 PROTOC.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(   )         /* VERSION OF COMPILER           */
SET COMPL    = &STR(SCCNCMP)     /* C COMPILER MODULES            */
SET EDCMSGS  = &STR(SCBCDMSG)    /* C COMPILER MESSAGES           */
SET LANG     = &STR(CBCLMSGS)    /* MESSAGE LANGUAGE              */
SET SCEEHDRS = &STR(SCEEH)       /* C SYSTEM HEADER FILES         */
SET CMOD     = &STR(CCNDRVR)     /* C COMPILER EXECUTABLE MODULE  */
SET WORKDA   = &STR(SYSDA)       /* UNIT TYPE FOR WORK FILES      */
SET WRKSPC   = &STR(1,1)         /* CYLS ALLOCATED FOR WORK FILES */
```

The CCRPGEN 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_VERSMISMATCH=6,       /* RPC versions not compatible */
   RPC_AUTHERROR=7,          /* authentication error */
   RPC_PROGUNAVAIL=8,        /* program not available */
   RPC_PROGVERSMISMATCH=9,   /* program version mismatched */
   RPC_PROCUNAVAIL=10,       /* procedure unavailable */
   RPC_CANTDECODEARGS=11,    /* decode arguments error */
   RPC_SYSTEMERROR=12,       /* generic "other problem" */
   /*
    * callrpc errors
    */
   RPC_UNKNOWNHOST=13,       /* unknown host name */
   /*
    * create errors
    */
   RPC_PMAPFAILURE=14,        /* the pmapper failed in its call */
   RPC_PROGNOTREGISTERED=15,  /* remote program is not registered */
   /*
    * unspecified error
    */
   RPC_FAILED=16
              };
```

Porting RPC applications

This topic 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 292.

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.

```
auth.h
auth@uni.h
bsdtime.h
bsdtocms.h
clnt.h
in.h
inet.h
manifest.h
netdb.h
pmap@cln.h
pmap@pro.h
rpc.h
rpc@msg.h
svc.h
svc@auth.h
socket.h
tcperrno.h
types.h
xdr.h
```

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 topic 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 when compiling and linking RPC applications

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 compilation cataloged procedure additions

Include the following in the compilation 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
  ```

Compiling and linking RPC applications: Nonreentrant modules

To compile and link nonreentrant RPC applications, the procedure is similar to the procedure for nonreentrant C applications as described in the topic 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
```

Compiling and linking RPC applications: Reentrant modules

To compile and link reentrant RPC applications, the procedure is similar to the procedure for reentrant C applications as described in the topic 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 topics 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 topics 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 292 for more information.
3. This procedure uses UDP as its transport layer. See "clntudp_create()" on page 249 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_sperrno()
- clnt_sperrno()
- 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_spcreateerror()
- 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_spcreateerror()
- 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_spcreateerror()
- 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 “svcraw_create() ” on page 279 for more information.

Return codes

NULL indicates failure.

Context

- clnt_call()
- clnt_create()
- clnt_destroy()
- clnt_freeres()
- clnt_geterr()
- clnt_pcreateerror()
- clnt_perror()
- clnt_spcreateerror()
- clnt_sperror()
- clntudp_create()
- clnttcp_create()
- svcraw_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*.

Requirements: If you use this handle to send the PMAPPROC_SET, PMAPPROC_UNSET, RPCBPROC_SET, or RPCBPROC_UNSET RPC to rpcbind, the following requirements apply:

- Your registration request must originate from an IP address on the local host.
- If the SAF profile EZB.RPCBIND.*sysname.rpcbindname*.REGISTRY is defined in the SERVAUTH class, your application user ID must be granted at least READ access to the profile.

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 that is 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*.

Requirements: If you use this handle to send the PMAPPROC_SET, PMAPPROC_UNSET, RPCBPROC_SET, or RPCBPROC_UNSET RPC to rpcbind, the following requirements apply:

- Your registration request must originate from an IP address on the local host.
- If the SAF profile EZB.RPCBIND.*sysname.rpcbindname*.REGISTRY is defined in the SERVAUTH class, your application user ID must be granted at least READ access to the profile.

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_spcreateerror()
- 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.

Requirements: When your application registers with rpcbind rather than with portmapper, the following requirements apply:

- Your registration request must originate from an IP address on the local host.
- If you have defined the SAF profile EZB.RPCBIND.*sysname*.*rpcbindname*.REGISTRY in the SERVAUTH class, your application user ID must be granted at least READ access to permit this library call.

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.

Requirements: When your application registers with rpcbind rather than with portmapper, the following requirements apply:

- Your registration request must originate from an IP address on the local host.
- If you have defined the SAF profile EZB.RPCBIND.*sysname.rpcbindname*.REGISTRY in the SERVAUTH class, your application user ID must be granted at least READ access to permit this library call.

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 292 for more information.

Requirements: When your application registers with rpcbind rather than with portmapper, the following requirements apply:

- Your registration request must originate from an IP address on the local host.
- If you have defined the SAF profile EZB.RPCBIND.*sysname.rpcbindname*.REGISTRY in the SERVAUTH class, your application user ID must be granted at least READ access to permit this library call.

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

- svcraw_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 the value 0 is used, the service is not registered with portmapper.

Rule: When using a toy RPC service transport created with svcraw_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.

Requirements: When your application registers with rpcbind rather than with portmapper, the following requirements apply:

- Your registration request must originate from an IP address on the local host.
- If you have defined the SAF profile EZB.RPCBIND.*sysname.rpcbindname*.REGISTRY in the SERVAUTH class, your application user ID must be granted at least READ access to permit this library call.

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.

Requirements: When your application registers with rpcbind rather than with portmapper, the following requirements apply:

- Your registration request must originate from an IP address on the local host.
- If you have defined the SAF profile EZB.RPCBIND.*sysname*.*rpcbindname*.REGISTRY in the SERVAUTH class, your application user ID must be granted at least READ access to permit this library call.

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_noprog()
- 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_noprog()
- 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_noprog()
- svcerr_progvers()
- svcerr_systemerr()
- svcerr_weakauth()

svcerr_noprog()

Format

```
#include <rpc.h>
void
svcerr_noprog(xprt)
SVCXPRT *xprt;
```

Parameters

xprt
Indicates the pointer to the service transport handle.

Usage

The svcerr_noprog() call is used when the desired program is not registered.

Context

- svcerr_decode()
- svcerr_noproc()
- 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_TOOWEAK).`

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()

svcraw_create()

Format

```
#include <rpc.h>
SVCXPRT *
svcraw_create()
```

Parameters

None.

Usage

The svcraw_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 “clntraw_create()” on page 246 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, recvsize, handle, readit, writeit)
XDR *xdrs;
u_int sendsize;
u_int recvsize;
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.

recvsize
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 topic 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.";

#ifndef 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(fltrcvprog,fltvers,fltrcvproc,
               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.
 *
 */
#ifndef 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 clntraw_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 topic 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 compilation 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 topic 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

z/OS UNIX System Services RPC deviates from Sun RPC 4.0 in the following ways:

- The source was modified to fit into 72 columns.
- **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

Requirement: The _ALL_SOURCE feature test macro is required to compile applications that use z/OS UNIX System Services RPC function. The source code can use #define _ALL_SOURCE or the _ALL_SOURCE macro can be passed as a compiler option.

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&colon;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
xdr_hyper()	Translates between C long longs and their external representatives.
xdr_u_hyper()	Translates between C unsigned long longs and their external representatives.
xdr_longlong_t()	Translates between C long longs and their external representatives.
xdr_u_longlong_t()	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 temeval wait;
    int *sockp;
    u_int sendsz;
    u_int recvsz;
```

RPC 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 *NCS for IBM AIX/ESA® Planning and Administration Guide* and *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 (0x7fffffff)

/* 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 *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:

```
base.h
conv.h
glb.h
bsdtocms.h
idl@base.h
lb.h
llb.h
nbase.h
ncsdefs.h
ncssock.h
pfm.h
rpc.h
rrpc.h
socket.h
uuid.h
```

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.

NCS 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 compilation 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 topic 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 *IDL_interface_name* 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 *xxx* could be replaced with the first 3 characters of the *IDL_interface_name*.

See "NIDL compiler options" on page 350 for a description of NIDL run-time options.

NCS: Preprocessing, compiling, and linking

The following topics 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 350.

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 *NCS for IBM AIX/ESA Planning and Administration Guide*.

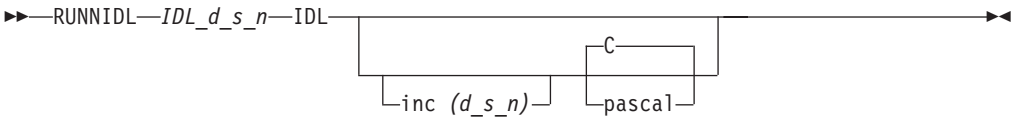
A command list (CLIST) called RUNNIDL is provided to assist you in invoking the NIDL compiler. RUNNIDL is a member of SEZAINST. The NIDL options specified in RUNNIDL CLIST are set to the most frequently used NIDL run-time options. If you do not want to run with these NIDL options, you can invoke the NIDL compiler directly.

The *NIDL* compiler does not support IDL include files that are members of a partitioned data set.

Any NCS system IDL files that are imported by your IDL data set must be copied from SEZAINST to your user ID. The following are the members of SEZAINST that you might need to copy.

Member	Data set name
basei	*user_id*.base.idl
convi	*user_id*.conv.idl
glbi	*user_id*.glb.idl
lbi	*user_id*.lb.idl
llbi	*user_id*.llb.idl
nbasei	*user_id*.nbase.idl
rpci	*user_id*.rpc.idl
rrpci	*user_id*.rrpc.idl
socketi	*user_id*.socket.idl
uuidi	*user_id*.uuid.idl

Use the RUNNIDL CLIST command in the following format:



Parameter	Description
IDL_d_s_n	Specifies the data set name of the NIDL data set.
IDL	Specifies the data set type of the NIDL data set. The data set type must be IDL.
inc (*d_s_n***)**	Specifies the data set name of a header data set that contains redefines specific to your programs and stubs. The NIDL compiler generates code to include the user-specified-include data set name in the stub data set and switch code that it generates. The data set name defaults to the USERDEFS.H data set.
pascal	Specifies that the NIDL compiler generates a Pascal language include data set as output. The server stub data set, client stub data set, client switch data set, and header data sets are generated in C language.

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.

```
RUNNIDL BANK IDL inc (bankdefs)
```

NIDL compiler limitations: You should be aware of the following limitations concerning the NIDL compiler options on MVS.

- –no_cpp

  You cannot invoke the NCS CPP routine from within the NIDL compiler. If you invoke NIDL directly, you must specify the –no_cpp option.

- –ext

  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
IDL_data_set_name@C	C@CSTUB	Client stub
IDL_data_set_name@W	C@CSWTCH	Client switch
IDL_data_set_name@S	C@SSTUB	Server stub
IDL_data_set_name	H	C header data set
IDL_data_set_name	COPY	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:

Parameter	Description
data_set_name	Specifies the name of the data set used as input to NCS CPP.
data_set_type	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
ncssock1	*user_id*.socket.h
ncsrpc	*user_id*.rpc.h
base	*user_id*.base.h
conv	*user_id*.conv.h
glb	*user_id*.glb.h
bsdtocms	*user_id*.bsdtocms.h

idl@base	*user_id*.idl@base.h
lb	*user_id*.lb.h
llb	*user_id*.llb.h
nbase	*user_id*.nbase.h
ncsdefs	*user_id*.ncfdefs.h
ncssock	*user_id*.ncssock.h
pfm	*user_id*.pfm.h
rrpc	*user_id*.rrpc.h
uuid	*user_id*.uuid.h

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 topic 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.

NCS: Sample compilation cataloged procedure additions

Include the following in the compilation 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
```

NCS: 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(RPC@S)
INCLUDE SYSLIB(RPC@SEQ)
INCLUDE SYSLIB(RPC@UTIL)
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 *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 *NCS for IBM AIX/ESA Planning and Administration Guide*.

Use the following format to invoke the 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 352 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 355, "Compiling, linking, and running the NCSSMP program" on page 359, and "Compiling, linking, and running the sample BANK program" on page 364.

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)
- NCSCLNT1 (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 topics describe steps required to run the sample BINOP program successfully.

- "Setting up the sample BINOP program" on page 356
- "Compiling the sample BINOP program" on page 357
- "Linking the sample BINOP program" on page 358
- "Running the sample BINOP program" on page 359

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.

Setting up the sample BINOP program

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)	*user_id*.binop.c
SEZAINST(BINOPCC)	*user_id*.binopc.c
SEZAINST(BINOPSC)	*user_id*.binops.c
SEZAINST(BINDEFS)	*user_id*.bindefs.h
SEZAINST(BINOPI)	*user_id*.binop.idl

---

2. Copy the imported data sets from SEZAINST to your user ID.

From location	To location
SEZAINST(BASEI)	*user_id*.base.idl
SEZAINST(NBASEI)	*user_id*.nbase.idl
SEZAINST(RPCI)	*user_id*.rpc.idl

---

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)	*user_id*.base.h
SEZACMAC(NBASE)	*user_id*.nbase.h
SEZACMAC(NCSDEFS)	*user_id*.ncsdefs.h
SEZACMAC(TYPES)	*user_id*.types.h
SEZACMAC(BSDTIME)	*user_id*.bsdtime.h
SEZACMAC(BSDTOCMS)	*user_id*.bsdtocms.h
SEZACMAC(BSDTYPES)	*user_id*.bsdtypes.h
SEZACMAC(IDL@BASE)	*user_id*.idl@base.h
SEZACMAC(PFM)	*user_id*.pfm.h
SEZACMAC(NCSRPC)	*user_id*.rpc.h
'C' library	*user_id*.setjmp.h
'C' library	*user_id*.stdio.h
'C' library	*user_id*.time.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 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.

Compiling the sample BINOP program

Before you begin: You need to have completed the steps in "Setting up the sample BINOP program" on page 356.

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(IBMCPP,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 compilation 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=BINOP@S
/s EDCC,INFILE=BINOP@C
/s EDCC,INFILE=BINOP@W
/s EDCC,INFILE=BINOP
```

You know you are done when no errors are received.

Linking the sample BINOP program

Before you begin: You need to have completed the steps in "Setting up the sample BINOP program" on page 356 and "Compiling the sample BINOP program" on page 357.

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(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 *user_id*.LOADLIST, by adding the following lines to the data set BINOPC.

```
INCLUDE SYSLIB(RPC@S)
INCLUDE SYSLIB(RPC@SEQ)
INCLUDE SYSLIB(RPC@UTIL)
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.

Running the sample BINOP program

Before you begin: You need to have completed the steps in "Setting up the sample BINOP program" on page 356, "Compiling the sample BINOP program" on page 357, and "Linking the sample BINOP program" on page 358.

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.

NCSCLNT1 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 355.

The following topics describe steps required to run the NCSSMP program successfully.

- "Setting up the NCSSMP program"
- "Compiling the NCSSMP program" on page 361
- "Linking the NCSSMP program" on page 362
- "Running the NCSSMP program" on page 363

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.

Setting up the NCSSMP program

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)	*user_id*.ncsserv1.c
SEZAINST(NCSCLNT1)	*user_id*.ncsclnt1.c
SEZAINST(NCSSMPI)	*user_id*.ncssmp.idl
SEZAINST(NSMPDEFS)	*user_id*.nsmpdefs.h

2. Copy the imported data sets from SEZAINST to your user ID.

From location	To location
SEZAINST(RPCI)	*user_id*.rpc.idl
SEZAINST(BASEI)	*user_id*.base.idl
SEZAINST(NBASEI)	*user_id*.nbase.idl

3. To generate stubs, run NIDL using the following command:

```
RUNNIDL NCSSMP IDL INC(nsmpdefs)
```

4. Copy the data sets included by CPP to your user ID.

From location	To location
SEZACMAC(NCSDEFS)	*user_id*.ncsdefs.h
SEZACMAC(BSDTOCMS)	*user_id*.bsdtocms.h

From location	To location
SEZACMAC(BASE)	*user_id*.base.h
SEZACMAC(IDL@BASE)	*user_id*.idl@base.h
SEZACMAC(NBASE)	*user_id*.nbase.h
SEZACMAC(LB)	*user_id*.lb.h
SEZACMAC(GLB)	*user_id*.glb.h
SEZACMAC(TYPES)	*user_id*.types.h
SEZACMAC(BSDTYPES)	*user_id*.bsdtypes.h
SEZACMAC(BSDTIME)	*user_id*.bsdtime.h
SEZACMAC(PFM)	*user_id*.pfm.h
C library	*user_id*.stdio.h
C library	*user_id*.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 NCSCLNT1 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.

Compiling the NCSSMP program

Before you begin: You need to have completed the steps in "Setting up the NCSSMP program" on page 360.

You can use several methods to compile, link-edit, and execute your program in MVS. This topic 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 compilation job at the Spool Display and Search Facility (SDSF) command panel, by entering the following:

```
/s EDCC,MEM=NCSSERV1
/s EDCC,MEM=NCSCLNT1
/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.

Linking the NCSSMP program

Before you begin: You need to have completed the steps in "Setting up the NCSSMP program" on page 360 and "Compiling the NCSSMP program" on page 361.

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 NCSCLNT1.

```
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=NCSCLNT1
/s EDCL,MEM=NCSSERV1
```

You know you are done when no errors are received.

Running the NCSSMP program

Before you begin: You need to have completed the steps in "Setting up the NCSSMP program" on page 360, "Compiling the NCSSMP program" on page 361, and "Linking the NCSSMP program" on page 362.

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(NCSCLNT1)' '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.
UUIDBIND	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 topics describe steps required to run the sample BANK program successfully.

- "Setting up the sample BANK program"
- "Compiling the sample BANK program" on page 366
- "Linking the sample BANK program" on page 367
- "Running the sample BANK program" on page 368

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.

Setting up the sample BANK program

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)	*user_id*.bankd.c
SEZAINST(BANKC)	*user_id*.bank.c
SEZAINST(UTILC)	*user_id*.util.c
SEZAINST(UUIDBIND)	*user_id*.uuidbind.c
SEZAINST(UTILH)	*user_id*.util.h
SEZAINST(BANKIDL)	*user_id*.bank.idl
SEZAINST(SHAWMUT)	*user_id*.shawmut.bank
SEZAINST(BAYBANK)	*user_id*.baybank.bank
SEZAINST(BANKDEFS)	*user_id*.bankdefs.h

2. Copy the data sets imported by IDL from SEZAINST to your user ID.

From location	To location
SEZAINST(BASEI)	*user_id*.base.idl
SEZAINST(NBASEI)	*user_id*.nbase.idl
SEZAINST(RPCI)	*user_id*.rpc.idl

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)	*user_id*.ncsdefs.h
SEZACMAC(BSDTOCMS)	*user_id*.bsdtocms.h
SEZACMAC(BASE)	*user_id*.base.h
SEZACMAC(IDL@BASE)	*user_id*.idl@base.h
SEZACMAC(NBASE)	*user_id*.nbase.h
SEZACMAC(LB)	*user_id*.lb.h
SEZACMAC(GLB)	*user_id*.glb.h
SEZACMAC(TYPES)	*user_id*.types.h
SEZACMAC(BSDTYPES)	*user_id*.bsdtypes.h
SEZACMAC(BSDTIME)	*user_id*.bsdtime.h
SEZACMAC(PFM)	*user_id*.pfm.h
SEZACMAC(UUID)	*user_id*.uuid.h
'C' library	*user_id*.stdio.h
'C' library	*user_id*.setjmp.h
'C' library(ERRNO)	*user_id*.errno.h
'C' library(TIME)	*user_id*.time.h

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 UUIDBIND 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.

Compiling the sample BANK program

Before you begin: You need to have completed the steps in "Setting up the sample BANK program" on page 364.

You can use several methods to compile, link-edit, and execute your program in MVS. This topic 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(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 compilation 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.

Linking the sample BANK program

Before you begin: You need to have completed the steps in "Setting up the sample BANK program" on page 364 and "Compiling the sample BANK program" on page 366.

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 *user_id*.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 *user_id*.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.

Running the sample BANK program

Before you begin: You need to have completed the steps in "Setting up the sample BANK program" on page 364, "Compiling the sample BANK program" on page 366, and "Linking the sample BANK program" on page 367.

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 'user_id.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 'user_id.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 *z/OS Communications Server: IP Configuration Guide*.

Running the sample mail filter 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
O InputMailFilters=filter
```

Library control functions

The following are sample mail filter program functions.

smfi_register

```
#include <libmilter/mfapi.h>
int smfi_register(
      smfiDesc_str descr
);
```

smfi_register 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.

smfi_register 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.

smfi_register 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;
);
```

smfi_setconn 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.

smfi_setconn 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

smfi_setconn 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
);
```

smfi_settimeout 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.

smfi_settimeout 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.

smfi_settimeout result

smfi_settimeout always returns MI_SUCCESS.

smfi_main

```
#include <libmilter/mfapi.h>
int smfi_main(
);
```

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.

smfi_main parameters

smfi_main has no parameters.

smfi_main 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

The following are mail filter data access functions.

smfi_getsymval

```
#include <libmilter/mfapi.h>
char* smfi_getsymval(
      SMFICTX *ctx,
      char *symname
);
```

smfi_getsymval 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.

smfi_getsymval parameters

ctx The opaque context structure.

symname
The name of a sendmail macro, optionally enclosed in braces ({ and }). See below for default macros.

smfi_getsymval 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 *Sendmail Installation and Operation Guide* provided with the sendmail distribution.

smfi_getpriv

```
#include <libmilter/mfapi.h>
void* smfi_getpriv(
      SMFICTX *ctx
);
```

smfi_getpriv description

Get the connection-specific data pointer for this connection. smfi_getpriv can be called in any of the xxfi_* callbacks.

smfi_getpriv parameters

ctx Opaque context structure.

smfi_getpriv 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
);
```

smfi_setpriv description

Set the private data pointer for this connection.

smfi_setpriv parameters

ctx Opaque context structure.

privatedata
Pointer to private data. This value is returned by subsequent calls to smfi_getpriv using ctx.

smfi_setpriv 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
);
```

smfi_setreply 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.

smfi_setreply 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.

smfi_setreply 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

The following are mail filter message modification functions.

smfi_addheader

```
#include <libmilter/mfapi.h>
int smfi_addheader(
      SMFICTX *ctx,
      char *headerf,
      char *headerv
);
```

smfi_addheader description

The smfi_addheader API adds a header to the current message. It is called only from xxfi_eom.

smfi_addheader 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.

smfi_addheader 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
);
```

smfi_chgheader description

The smfi_chgheader API changes a header's value for the current message. It is called only from xxfi_eom.

smfi_chgheader 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.

smfi_chgheader 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
);
```

smfi_addrcpt description

The smfi_addrcpt API adds a recipient to the message envelope. It is called only from xxfi_eom.

smfi_addrcpt parameters

ctx Opaque context structure.

rcpt The new recipient's address.

smfi_addrcpt 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_ADDRCPT 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_ADDRCPT flag in the smfiDesc_str passed to smfi_register.

smfi_delrcpt

```
#include <libmilter/mfapi.h>
int smfi_delrcpt(
      SMFICTX *ctx;
      char *rcpt;
);
```

smfi_delrcpt description

The smfi_delrcpt API removes the named recipient from the current message's envelope. It is called only from xxfi_eom.

smfi_delrcpt parameters

ctx Opaque context structure.

rcpt The recipient address to be removed, a non-NULL, null-terminated string.

smfi_delrcpt 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_DELRCPT 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
);
```

smfi_replacebody 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.

smfi_replacebody 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.

smfi_replacebody 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.

Mail filter callbacks

Each of these callbacks should return one of the values that is 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);
```

xxfi_connect description

Called once, at the start of each SMTP connection. Default behavior is to do nothing and return SMFIS_CONTINUE.

xxfi_connect 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.

xxfi_connect 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
);
```

xxfi_helo 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.

xxfi_helo 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
);
```

xxfi_envfrom 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.

xxfi_envfrom 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.

xxfi_envfrom 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
);
```

xxfi_envrcpt 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.

xxfi_envrcpt 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.

xxfi_envrcpt 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
);
```

xxfi_header 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.

xxfi_header 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
);
```

xxfi_eoh 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.

xxfi_eoh parameters

ctx Opaque context structure.

xxfi_body - body block

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_body)(
      SMFICTX * ctx,
      unsigned char * bodyp,
      size_t len
);
```

xxfi_body 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.

xxfi_body 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
);
```

xxfi_eom 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.

xxfi_eom 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
);
```

xxfi_abort 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).

xxfi_abort 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
);
```

xxfi_close 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.

xxfi_close 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 *z/OS XL C/C++ User's Guide* for more information about how to specify compiler options.
3. Include the PAPI definition side deck (papi.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 *z/OS MVS Program Management: User's Guide and Reference* for information about 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 following 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.

PAPI 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.

PAPI: Connecting and retrieving data

Use the following PAPI functions for connecting and retrieving data.

papi_connect - Connect to Policy Agent

```
#include <papiuser.h>

extern int papi_connect(void **papiHandle, void *regReq);
```

papi_connect 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().

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.

papi_connect 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.

papi_connect 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);
```

papi_debug 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.

papi_debug 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  */
```

papi_debug 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);
```

papi_disconnect 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.

papi_disconnect 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.

papi_disconnect 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.

papi_disconnect 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);
```

papi_free_perf_data 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.

papi_free_perf_data parameters

***perfDataHandle**
: This is an input parameter of type void that points to the memory obtained from the papi_get_perf_data() API.

papi_free_perf_data 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.

papi_free_perf_data 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);
```

papi_get_perf_data 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 *z/OS Communications Server: IP Configuration Reference* and the policy-based networking information in *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.

papi_get_perf_data 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 *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_TCPIMAGE_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.

papi_get_perf_data 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.

papi_get_perf_data 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 ... */
}
```

PAPI helper functions

The following are PAPI 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);
```

papi_get_action_perf_by_id 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).

papi_get_action_perf_by_id 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.

papi_get_action_perf_by_id 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);
```

papi_get_action_perf_info 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;
```

papi_get_action_perf_info 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.

papi_get_action_perf_info 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 );
```

papi_get_actions_count description

This function is used to obtain the number of actions in the policy performance data returned by the papi_get_perf_data() function.

papi_get_actions_count 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.

papi_get_actions_count 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 );
```

papi_get_policy_instance 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).

papi_get_policy_instance 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.

papi_get_policy_instance 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);
```

papi_get_rule_perf_by_id 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 396 (the recordId will match the ruleId field).

papi_get_rule_perf_by_id 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.

papi_get_rule_perf_by_id 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);
```

papi_get_rule_perf_info 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;
```

papi_get_rule_perf_info 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.

papi_get_rule_perf_info 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 );
```

papi_get_rules_count description

This function is used to obtain the number of rules in the policy performance data returned by the papi_get_perf_data() function.

papi_get_rules_count 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.

papi_get_rules_count 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 );
```

papi_strerror description

This function is used to obtain a string describing a PAPI return code value. It is similar to the C strerror() function.

papi_strerror parameters

papiReturnCode

This is an input parameter of type int. This parameter contains a PAPI return code value.

papi_strerror 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 421).

  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 multitask 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 406). 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:

FTP client API compatibility considerations

Unless noted in the *z/OS Communications Server: New Function Summary*, an application program that is 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.

FTP client API guidelines and requirements

This topic 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.

Table 8. Programming requirements for the FTP client API (continued)

Function	Restriction
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.
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 406).
- 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 469 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 406). 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 477).

- 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 EZAFTPKI 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 (EZAFTPKS) into the user application program or load it dynamically for execution. The stub program resides in SYS*n*.CSSLIB and is designed to maintain upward compatibility.
- For a PL/I program, include this statement before your first call instruction:

```
DCL EZAFTPKS ENTRY OPTIONS(RETCODE,ASM,INTER) EXT;
```

Restriction: Do not set TRACE RESOLVER in the TCPIP DATA file that an FTP client API application is using. If you need a resolver trace, pass the environment variable RESOLVER_TRACE to the FTP client on the INIT request.

Java call formats

The FTP client API for Java provides an interface to the z/OS FTP client that enables a user program written in Java to send subcommands for the client to process. The user program can also use this interface to retrieve output that includes the messages from the client, replies from the FTP server, and other data that is generated as the result of the request.

Each instance of the interface is represented by an FTPClient object. A user program can create multiple instances of the FTPClient object. A single user program can use these objects to establish multiple simultaneous connections to the same FTP server or to different FTP servers. The user program can multitask to different instances of the interface by requesting that the API not wait for the completion of an FTP subcommand before it returns control.

The z/OS FTP client that is used by the FTP client API is described in File Transfer Protocol (FTP) information in *z/OS Communications Server: IP User's Guide and Commands* and in the File Transfer Protocol information in *z/OS Communications Server: IP Configuration Reference*. The z/OS FTP client, when started with the FTP client API for Java, operates essentially the same as it does when invoked in an interactive environment under the z/OS UNIX shell. See "z/OS FTP client behavior when invoked from the FTP client API" on page 404 for a description of the differences.

FTP client API for Java package uses the Java Native Interface (JNI) to interface with the z/OS FTP client using the C Java FTP client API. See "FTP client API for C functions" on page 438 for more information about the FTP client API for C.

The FTP client API for Java uses the Java logging API (java.util.logging.Logger) to generate debug information. See documentation about the java.util.logging package for details about using the Java logging API.

Guidelines:

- The user program can have more than one FTPClient object initialized and active in a single address space.

- 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 and does not block or explicitly raise any signals.

Requirements:

- All requests that use the same FTPClient object must be made from the same thread.
- The Java JVM in which the application runs must operate in 31-bit addressing mode. No other addressing modes are supported by the interface.
- The application must have an OMVS segment defined (or set by default).
- The interface module EZAFTPKI must be accessible to the application in the link list or in a STEPLIB or JOBLIB DD statement.
- To use this package, you must include the EZAFTP.jar file in your classpath. In addition, the libEZAFTP.so file must be located in $LIBPATH so that the JNI methods can be found. The EZAFTP.jar file is installed into the directory /usr/include/java_classes and the libEZAFTP.so file is installed into the directory /usr/lib.

For more information about the FTP client API for Java, see the JavaDoc that is included in the EZAFTPDoc.jar file, which is installed into the directory /usr/include/java_classes. Download the jar file to a workstation, unpack it, and read it in a web browser.

COBOL, C, REXX, assembler, and PL/I call formats

The FTP client API is invoked by calling the EZAFTPKS program. The following list shows formats for the COBOL, C, assembler, and PL/I languages.

- COBOL language call format

  The EZAFTPKS call format for COBOL programs is the following:

  ```
  >>---CALL EZAFTPKS USING FCAI-Map, request_type, parm1, parm2, ... ----->< 
  ```

- C language call format

  See “FTP client API for C functions” on page 438 for in-line functions that can be used with C/C++ programs. These in-line functions provide the calls to EZAFTPKS.

- REXX language call format

  See “FTP client API for REXX function” on page 441 for an external REXX function that can be used with REXX programs.

- Assembler language call format

  The EZAFTPKS call format for assembler language programs is the following:

  ```
  >>---CALL EZAFTPKS,(FCAI_Map, request_type, parm1, parm2, ... ),VL ----->< 
  ```

- PL/I language call format

  The EZAFTPKS call format for PL/I programs is the following:

  ```
  >>---CALL EZAFTPKS (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 421). 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 407). 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 465 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 467 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 429, "TERM" on page 436, and "Prompts from the client" on page 465 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 424 and "Prompts from the client" on page 465 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 414.

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 on page 407 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 421 uses the COBOL syntax; the field names contain a dash (-) instead of an underscore (_).

- "FTP client API for C functions" on page 438 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
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 409) binary
FCAI_PollWait	1	7	POLL wait timer in seconds (see "FCAI_PollWait: Specifying a wait time before POLL" on page 471).	(I) binary
FCAI_ReqTimer	1	8	Request timer in seconds or 0 for none (see "FCAI_ReqTimer: Controlling requests that retrieve results from the spawned z/OS FTP client process" on page 471).	(I) binary
FCAI_TraceIt	1	9	Trace indicator for this request (see "Using the FTP client API trace" on page 473).	(I - see Table 11 on page 409) 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 409) 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 470).	(O - see Table 13 on page 409) 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

Table 9. FCAI control block (continued)

Field name	Length	Offset	Description	Type
FCAI_RCV	16	32	Request completion values (see "Interpreting results from an interface request" on page 467).	(O) binary
FCAI_Result	1	32	Request result (the return code register also contains this value).	(O - see Table 14 on page 410) binary
FCAI_Status	1	33	Status code.	(O - see Table 15 on page 410) binary
FCAI_IE	1	34	Interface error.	(O - see Table 16 on page 410) binary
FCAI_CEC	1	35	Client error code (see FTP return codes in the *z/OS Communications Server: IP User's Guide and Commands*).	(O) binary
FCAI_ReplyCode	2	36	Server reply code or 0 if no reply (see FTPD reply codes in *z/OS Communications Server: IP and SNA Codes*).	(O) binary
FCAI_SCMD	1	38	Subcommand code (see FTP subcommand codesin *z/OS Communications Server: IP User's Guide and Commands*).	(O) binary
Reserved	1	39	Reserved.	(R) undefined
FCAI_ReturnCode	4	40	Return code (see Table 13 on page 409 and Table 16 on page 410 for errors that have associated return code data).	(O) binary
FCAI_ReasonCode	4	44	Reason code (see Table 13 on page 409 and Table 16 on page 410 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 473.

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 470.

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 on page 410, see "Interpreting results from an interface request" on page 467.

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 465 and "Interpreting results from an interface request" on page 467.

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 *added* to any other status returned. See "FCAI_Status_TraceFailed and FCAI_TraceStatus: Reporting failures in the interface trace function" on page 470 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 EZAFTPKS.
FCAI_IE_RequestUnknown	2	Unknown request.	Request ID not INIT, TERM, POLL, GETL, or SCMD.

Table 16. FCAI_IE field values (continued)

Name	Value	Description	Additional Information
FCAI_IE_ParmMissing	3	Parameter missing.	Required parameter not passed to EZAFTPKS.
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 EZAFTPKS is one cause.
FCAI_IE_ControlErr	6	Error altering an open file descriptor.	FCAI_ReturnCode and FCAI_ReasonCode contain values set by BPX1FCT in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
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 472.
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 470.
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 471 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= *hfs_filename* to pass more than nine environment variables.

Table 16. FCAI_IE field values (continued)

Name	Value	Description	Additional Information
FCAI_IE_CreatePipeErr	26	Error creating pipe to the client.	FCAI_ReturnCode and FCAI_ReasonCode contain values set by BPX1PIP in the *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
FCAI_IE_SpawnErr	27	Error spawning the client.	FCAI_ReturnCode and FCAI_ReasonCode contain values set by BPX1SPN in the *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
SCMD errors			
FCAI_IE_ScmdParmTooBig	32	SCMD subcommand string too long.	SCMD subcommand parameter string must not exceed 2064 bytes.
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 *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
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 *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
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			

Table 16. FCAI_IE field values (continued)

Name	Value	Description	Additional Information
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
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 432.
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 429 for more information.
FCAI_IE_TraceIDTooBig	69	Length of traceID must be 0 - 3 characters.	Set only by the FTP client API for REXX.
FCAI_IE_TraceSClassTooBig	70	Length of traceSClass value must be 0 - 3 characters.	Set only by the FTP client API for REXX.
FCAI_IE_UnknownTraceIt	71	The traceIt value is not recognized.	Set only by the FTP client API for REXX.
FCAI_IE_ReqTimerInvalid	72	The request timer value is not in the range 0 - 255.	Set only by the FTP client API for REXX.
FCAI_IE_LinesParmTooBig	73	The GETL lines stem name is more than 200 characters in length.	Set only by the FTP client API for REXX.
FCAI_IE_PollWaitInvalid	74	The pollWait value is not in the range 0 - 255.	Set only by the FTP client API for REXX.
FCAI_FCAI_IE_NumTraceInvalid	75	The numTrace value is not in the range 1 - 1000000.	Set only by the FTP client API for REXX.
FCAI_IE_FcaiMapParmTooBig	76	The FCAI stem name is more than 250 characters in length.	Set only by the FTP client API for REXX.
FCAI_IE_EnvVarStorageErr	77	Unable to allocate storage for an environment variable.	Set only by the FTP client API for REXX.

Table 16. FCAI_IE field values (continued)

Name	Value	Description	Additional Information
FCAI_IE_SysoutClassErr	78	FCAI_TraceSClass contains a Sysout output class that is not valid.	Set only by the FTP client API for REXX.

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
- EZAFTPKC — COBOL copy book found in SEZANMAC
- ftpcapi.h — C header file found in /usr/include/
- EZAFTPKP — 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 FTP client API for REXX, 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 FTP client API for REXX function package.

Table 17. FCAI stem variables

Name	Type	Description	Additional Information
stem.FCAI_Result	Decimal	This stem variable corresponds to the FCAI_Result field in the FCAI_Map control block.	See Table 15 on page 410.
stem.FCAI_IE	Decimal	This stem variable corresponds to the FCAI_IE field in the FCAI_Map control block.	See Table 16 on page 410.
stem.FCAI_CEC	Decimal	This stem variable corresponds to the FCAI_CEC field in the FCAI_Map control block.	See FTP return codes in the *z/OS Communications Server: IP User's Guide and Commands*.
stem.FCAI_ReplyCode	Decimal	This stem variable corresponds to the FCAI_ReplyCode in the FCAI_Map control block.	See FTPD reply codes in *z/OS Communications Server: IP and SNA Codes*.

Table 17. FCAI stem variables (continued)

Name	Type	Description	Additional Information
stem.FCAI_TraceStatus	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 409.
stem.FCAI_ReturnCode	Decimal	This stem variable corresponds to the FCAI_ReturnCode field in the FCAI_Map control block.	See Table 13 on page 409 and Table 16 on page 410.
stem.FCAI_ReasonCode	Decimal	This stem variable corresponds to the FCAI_ReasonCode field in the FCAI_Map control block.	See Table 13 on page 409 and Table 16 on page 410.
stem.FCAI_SCMD	Decimal	This stem variable corresponds to the FCAI_SCMD field in the FCAI_Map control block.	See FTP subcommand codes in the *z/OS Communications Server: IP User's Guide and Commands*.
stem.FCAI_Map	Binary	This stem variable contains a binary representation of the FCAI_Map control block, plus additional fields used by the FTP client API for REXX.	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 *stem*.FCAI_CEC
FCAI_CEC_AUTHENTICATION	Decimal	17	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_CLIENT_ERR	Decimal	24	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_CONNECT_FAILED	Decimal	8	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_CONVERSION	Decimal	21	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_EOD_BEFORE_EOF	Decimal	25	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_FILE_ACCESS	Decimal	18	Can be stored in *stem*.FCAI_CEC

Table 18. Predefined REXX variables (continued)

Name	Type	Description	Additional Information
FCAI_CEC_FILE_READ	Decimal	19	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_FILE_WRITE	Decimal	20	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_INPUT_ERR	Decimal	12	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_INTERNAL_ERROR	Decimal	1	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_INVALID_ENVIRONMENT	Decimal	15	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_INVALID_PARAM	Decimal	4	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_LOGIN_FAILED	Decimal	11	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_NEEDS_CONNECTION	Decimal	26	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_NOT_ENABLED	Decimal	16	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_NOTFOUND	Decimal	14	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_OPEN_IOSTREAM_FAILED	Decimal	5	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_PROXY_ERR	Decimal	22	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_SERVER_ERROR	Decimal	2	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_SESSION_ERROR	Decimal	10	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_SQL_ERR	Decimal	23	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_TIMEOUT	Decimal	9	Can be stored in *stem*.FCAI_CEC
FCAI_CEC_USAGE	Decimal	7	Can be stored in *stem*.FCAI_CEC
FCAI_ERROR_CEC	Decimal	-3	Can be returned by FTPAPI call
FCAI_ERROR_IE	Decimal	-2	Can be returned by FTPAPI call
FCAI_GETL_ANY_LINE	Char	A	Value for type parameter on GETL_FIND or GETL_COPY requests
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 *stem*.FCAI_IE
FCAI_IE_ALREADYINPROGRESS	Decimal	37	Can be stored in *stem*.FCAI_IE
FCAI_IE_APIALREADYINIT	Decimal	16	Can be stored in *stem*.FCAI_IE
FCAI_IE_APILOADFAILED	Decimal	18	Can be stored in *stem*.FCAI_IE
FCAI_IE_BADTOKENADDR	Decimal	20	Can be stored in *stem*.FCAI_IE
FCAI_IE_BUFFERTOOSMALL	Decimal	68	Can be stored in *stem*.FCAI_IE
FCAI_IE_CLIPROCESSBROKEN	Decimal	47	Can be stored in *stem*.FCAI_IE
FCAI_IE_CLIPROCESSSTOPPED	Decimal	38	Can be stored in *stem*.FCAI_IE
FCAI_IE_CONTROLERR	Decimal	6	Can be stored in *stem*.FCAI_IE
FCAI_IE_CREATEPIPEERR	Decimal	26	Can be stored in *stem*.FCAI_IE
FCAI_IE_ENVVARSTORAGEERR	Decimal	77	Can be stored in *stem*.FCAI_IE
FCAI_IE_GETWORKAREAFAILED	Decimal	21	Can be stored in *stem*.FCAI_IE
FCAI_IE_INITPARMTOOBIG	Decimal	17	Can be stored in *stem*.FCAI_IE
FCAI_IE_INTERNALERR	Decimal	7	Can be stored in *stem*.FCAI_IE
FCAI_IE_LENGTHINVALID	Decimal	8	Can be stored in *stem*.FCAI_IE
FCAI_IE_LINESPARMTOOBIG	Decimal	73	Can be stored in *stem*.FCAI_IE
FCAI_IE_NOTINPROGRESS	Decimal	48	Can be stored in *stem*.FCAI_IE
FCAI_IE_NOTOKENADDR	Decimal	19	Can be stored in *stem*.FCAI_IE
FCAI_IE_PARMMISSING	Decimal	3	Can be stored in *stem*.FCAI_IE
FCAI_IE_PARMSTORAGEERR	Decimal	4	Can be stored in *stem*.FCAI_IE
FCAI_IE_PASSPROMPTERR	Decimal	34	Can be stored in *stem*.FCAI_IE

Table 18. Predefined REXX variables (continued)

Name	Type	Description	Additional Information
FCAI_IE_POLLWAITINVALID	Decimal	74	Can be stored in *stem*.FCAI_IE
FCAI_IE_READERR	Decimal	42	Can be stored in *stem*.FCAI_IE
FCAI_IE_REQTIMEREXPIRED	Decimal	22	Can be stored in *stem*.FCAI_IE
FCAI_IE_REQTIMERINVALID	Decimal	72	Can be stored in *stem*.FCAI_IE
FCAI_IE_REQUESTMISSING	Decimal	1	Can be stored in *stem*.FCAI_IE
FCAI_IE_REQUESTUNKNOWN	Decimal	2	Can be stored in *stem*.FCAI_IE
FCAI_IE_SCMDPARMTOOBIG	Decimal	32	Can be stored in *stem*.FCAI_IE
FCAI_IE_SPAWNERR	Decimal	27	Can be stored in *stem*.FCAI_IE
FCAI_IE_SYSOUTCLASSERR	Decimal	78	Can be stored in fcaiMap.FCAI_IE
FCAI_IE_TOOMANYENVVARS	Decimal	24	Can be stored in *stem*.FCAI_IE
FCAI_IE_TOOMANYINITPARMS	Decimal	23	Can be stored in *stem*.FCAI_IE
FCAI_IE_TOOMANYPARAMETERS	Decimal	5	Can be stored in *stem*.FCAI_IE
FCAI_IE_TRACEIDTOOBIG	Decimal	69	Can be stored in *stem*.FCAI_IE
FCAI_IE_TRACESCLASSTOOBIG	Decimal	70	Can be stored in *stem*.FCAI_IE
FCAI_IE_UNKMODE	Decimal	33	Can be stored in *stem*.FCAI_IE
FCAI_IE_UNKNOWNOPERATION	Decimal	64	Can be stored in *stem*.FCAI_IE
FCAI_IE_UNKNOWNSEQUENCE	Decimal	66	Can be stored in *stem*.FCAI_IE
FCAI_IE_UNKNOWNTRACEIT	Decimal	71	Can be stored in *stem*.FCAI_IE
FCAI_IE_UNKNOWNTYPE	Decimal	65	Can be stored in *stem*.FCAI_IE
FCAI_IE_VECTORSTORAGEERR	Decimal	67	Can be stored in *stem*.FCAI_IE
FCAI_IE_WRITEERR	Decimal	41	Can be stored in *stem*.FCAI_IE
FCAI_MODE_NOWAIT	Char	N	Can be stored in *stem*.FCAI_IE
FCAI_MODE_WAIT	Char	W	Can be stored in *stem*.FCAI_IE
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 *stem*.FCAI_IE
FCAI_RESULT_NOMATCH	Decimal	4	Can be stored in *stem*.FCAI_Result or can be returned by an FTPAPI call
FCAI_RESULT_OK	Decimal	0	Can be stored in *stem*.FCAI_Result or returned by an FTPAPI call
FCAI_RESULT_PROMPTACCT	Decimal	3	Can be stored in *stem*.FCAI_Result or returned by an FTPAPI call
FCAI_RESULT_PROMPTPASS	Decimal	2	Can be stored in *stem*.FCAI_Result 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 fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_AMBIGUOUS	Decimal	1	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_APPE	Decimal	4	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_ASCII	Decimal	5	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_BIG5	Decimal	57	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_BINARY	Decimal	6	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_BLOCK	Decimal	58	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_CCC	Decimal	77	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_CD	Decimal	7	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_CDUP	Decimal	51	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_CLEAR	Decimal	72	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_CLOSE	Decimal	8	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_COMPRESS	Decimal	59	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_CPROTECT	Decimal	73	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_DEBUG	Decimal	11	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_DELE	Decimal	13	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_DELIMIT	Decimal	12	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_DIR	Decimal	14	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_DUMP	Decimal	70	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_EBCDIC	Decimal	15	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_EUCKANJI	Decimal	46	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
FCAI_SCMD_FEAT	Decimal	78	Can be stored in fcaiMap.FCAI_SCMD by a GET_FCAI_MAP request
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

Table 18. Predefined REXX variables (continued)

Name	Type	Description	Additional Information
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_MKFIFO	Decimal	82	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
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

Table 18. Predefined REXX variables (continued)

Name	Type	Description	Additional Information
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_STATUS_INPROGRESS	Decimal	1	Can be returned by an FTPAPI call
FCAI_STATUS_PROMPTACCT	Decimal	3	Can be returned by an FTPAPI call
FCAI_STATUS_PROMPTPASS	Decimal	2	Can be returned by an FTPAPI call
FCAI_STATUS_TRACEFAILED	Decimal	200	Can be returned by an FTPAPI call
FCAI_TASK_CLIPROCESSKILL	Decimal	-32	Can be stored in *stem*.FCAI_Result or returned by an FTPAPI function call
FCAI_TASK_TASKMISMATCH	Decimal	-18	Can be stored in *stem*.FCAI_Result or returned by an FTPAPI function call
FCAI_TRACE_DATASET_NAME	String		Data set name to which the REXX FTP Client trace is written if the FTP client API for REXX is active, or an empty string ('') if the trace is not active.
FCAI_TRACE_DATASET_RETCODE	Integer		Return code from FTP client API for REXX tracing from the last invocation of the FTP client API for REXX
FCAI_TRACE_WORKAREA	Binary		Binary data used by the FTP client API for REXX when writing trace records. **Rule:** This variable must not be modified by the REXX program.
FCAI_TRACECAPI_A	Decimal	1	Can be stored in fcaiMap.FCAI_TraceCAPI by a GET_FCAI_MAP request
FCAI_TRACECAPI_C	Decimal	0	Can be stored in fcaiMap.FCAI_TraceCAPI by a GET_FCAI_MAP request
FCAI_TRACECAPI_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 topic 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 467.

Tip: This topic provides information on the Cobol, C, assembler and PL/I programming languages. For REXX programs, see "FTP client API for REXX function" on page 441.

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 424.
- The caller can specify the number of seconds to wait by setting an FCAI_ReqTimer value (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 471.
- Before exiting, your application should issue a TERM request for each INIT request.
- If using the Trace Resolver facility, the trace should be activated by specifying the RESOLVER_TRACE environment variable to collect the trace information in a file or MVS dataset.

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 tls'.

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 404.

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 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 *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 *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 *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 467. See "FTP client API: Other output that is returned to the application" on page 464 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 404.

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 471. 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 471. 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 436 and "Prompts from the client" on page 465 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 436 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 467. See "FTP client API: Other output that is returned to the application" on page 464 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 470.
- 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 465 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 429 for a discussion of the interface buffer. See "FCAI_PollWait: Specifying a wait time before POLL" on page 471 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 404.

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 467. See "FTP client API: Other output that is returned to the application" on page 464 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 470 for information about FCAI-Status-TraceFailed.

- Aside from interface errors, POLL itself returns only FCAI-Status-InProgess. 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 "FTP client API: Other output that is returned to the application" on page 464 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.)
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 "FTP client API: Other output that is returned to the application" on page 464 and "POLL" on page 427.

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 topic.

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 434 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 404.

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 *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 467.

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 caller can specify the number of seconds to wait by setting a FCAI_ReqTimer value (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 471.

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 404.

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 467.

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 for 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 421 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 *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.

Rule: If using the Trace Resolver facility, the trace should be activated by specifying the RESOLVER_TRACE environment variable to collect the trace information in a file or MVS dataset.

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_envars);
```

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 421 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 421 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 421 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 421 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.
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.

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
FAPI_GETL_FIND_FIRST	F	Find the first line of the requested type.

Field constant defined in ftpcapi.h	Value	Meaning
FAPI_GETL_FIND_NEXT	N	Find the next line of the requested type.
FAPI_GETL_FIND_LAST	L	Find the last line of the requested type.

FAPI_GETL_FIND example:

```
/* Assume FAPI_INIT was successfully called with &fcai */
/* prior to calling FAPI_GETL_FIND                     */
char buffer[4096];
rc = FAPI_GETL_FIND(&fcai, FAPI_GETL_LIST_LINE,
 sizeof(buffer), buffer, FAPI_GETL_FIND_LAST);
```

FAPI_TERM

Use the FAPI_TERM function to terminate the FTP client API instance associated with this FCAI_MAP. See "Sending requests to the FTP client API" on page 421 for more information on a TERM request.

FAPI_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.

FAPI_TERM example:

```
/* Assume FAPI_INIT was successfully called with &fcai */
/* prior to calling FAPI_TERM                          */
rc = FAPI_TERM(&fcai);
```

FTP client API for 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 FTP client API for REXX is included as a default system function package, and is included in the default parameter modules IRXTSPRM, IRXISPRM, and IRXPARMS.

The FTP client API for REXX has been verified to run in the following environments:

- TSO exec
- Batch environment
- ISPF
- UNIX shell

The FTP client API for REXX 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 SIGCHILD 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 the REXX program is running in a POSIX environment and does not have a SIGCHLD signal handler when it invokes the first CREATE request, then the FTP client API for REXX requests that the SIGCHLD signal be ignored. The REXX parent program resets the signal handler to the default state only after every FTP client instance that is used in the REXX program has ended. If the REXX program has a SIGCHLD handler, no change is made by the FTP client API.

FTP client API for REXX trace

The FTP client API for REXX trace is used to debug problems in the FTP client API for REXX 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 FTP client API for REXX trace is separate from the FTP client API trace. It uses a different output file and has a different enabling mechanism. The FTP client API for REXX trace is used to debug problems in the FTP client API for REXX 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 FTP client API for REXX trace is activated by specifying the z/OS UNIX FTP_REXX_TRACE_FILE environment variable or allocating the FTPRXTRC DD name. The FTP client API for REXX first looks for the FTP_REXX_TRACE_FILE environment variable (z/OS UNIX environment only) and then for the FTPRXTRC DD allocation.

The FTP client API for REXX 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 FTP client API for REXX when needed

Restriction: In order for the FTP client API for REXX 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 8 – 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 FTP client API for REXX 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 FTP client API for REXX trace output location

Your environment determines the method you use to specify the FTP client API for REXX trace output location.

Specifying the FTP client API for REXX trace output location: TSO environment: In the TSO environment, the location specified by the FTPRXTRC DD statement is used as the FTP client API for REXX 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 *z/OS TSO/E Command Reference* for more details about using the ALLOCATE command.

Specifying the FTP client API for REXX trace output location: z/OS UNIX shell environment: In the z/OS UNIX shell environment, use one of the following to specify the FTP client API for REXX 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.

Specifying the FTP client API for REXX trace output location: 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=*
```

Specifying the FTP client API for REXX trace output location: 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 FTP client API for REXX 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 *z/OS UNIX System Services Command Reference* for additional considerations when using the BPXBATCH or BPXBATSL utilities.

FTP client API requests

All FTP client API for REXX 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 FTP client API for REXX return codes use a consistent format, as shown in Table 19.

Table 19. FTP client API for REXX return codes

Code value	Definition
<0	A negative return code indicates that the FTP client API for REXX 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.

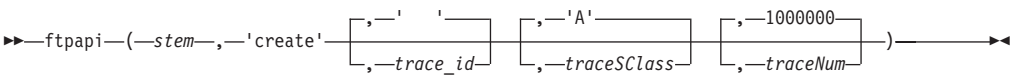
The following requests are supported by the FTP client API for REXX:

- 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 EZAFTPKS 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:



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 FTP client API for REXX 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

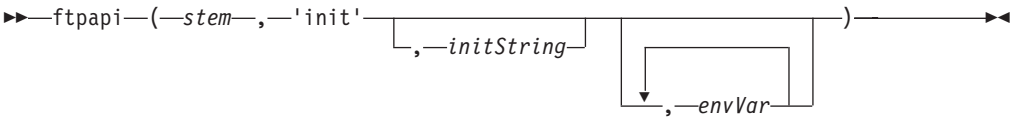
Value	Return codes	Explanation
<0		The call did not complete successfully. The FTP client API stem variable was not created. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the *stem*.FCAI_IE stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the *stem*.FCAI_CEC 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 *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 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 *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.
- If using the Trace Resolver facility, the trace should be activated by specifying the RESOLVER_TRACE environment variable to collect the trace information in a file or MVS dataset.

Results:

Table 21. FTP client INIT request return codes

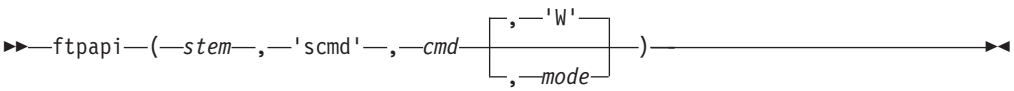
Value	Return codes	Explanation
<0		The call did not complete successfully. The subcommand was not successfully executed. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the *stem*.FCAI_IE stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the *stem*.FCAI_CEC 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:



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 *stem*.FCAI_IE stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the *stem*.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)	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)

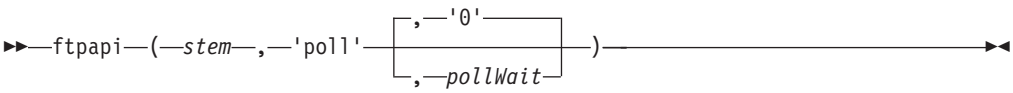
Value	Return codes	Explanation
	2 (FCAI_RESULT_PROMPTPASS)	The prior request has completed successfully but the next request must be a PASS subcommand.
	3 (FCAI_RESULT_PROMPTACCT)	The prior request has completed successfully but the next request must be an ACCT subcommand.

Examples:

```
rc = ftpapi('fcai.', 'scmd', 'DIR /tmp/*', 'N')
```

POLL request

Format:



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 *stem*.FCAI_IE stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the *stem*.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; 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)

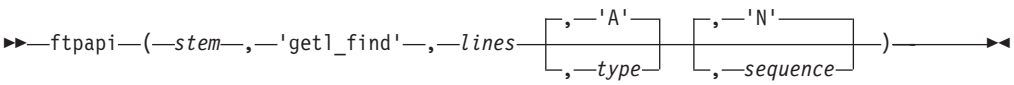
Value	Return codes	Explanation
	2 (FCAI_RESULT_PROMPTPASS)	The prior request has completed successfully but the next request must be a PASS subcommand.
	3 (FCAI_RESULT_PROMPTACCT)	The prior request has completed successfully but the next request must be an ACCT subcommand.

Examples:

```
rc = ftpapi('fcai.', 'poll')
```

GETL_FIND request

Format:



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

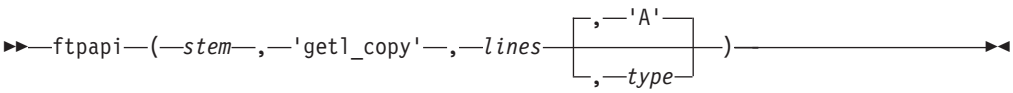
Value	Return codes	Explanation
<0		The call did not complete successfully. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the *stem*.FCAI_IE stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the *stem*.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)	A matching output line was found and returned in the *lines* 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:



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 all remaining lines of output of type *type* that were 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 *stem*.FCAI_IE stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the *stem*.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 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 *lines* 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:

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

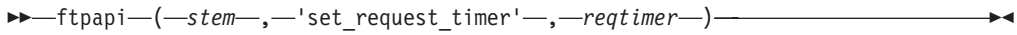
Value	Return codes	Explanation
<0		The call did not complete successfully. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the *stem*.FCAI_IE stem variable.
	-17 (FCAI_RESULT_UNUSABLESTEM)	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_trace', 'ON')
```

SET_REQUEST_TIMER request

Format:



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 seconds 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

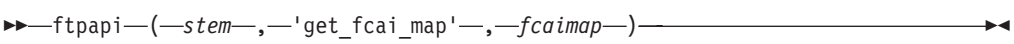
Value	Return codes	Explanation
<0		The call did not complete successfully. The possible failure return codes are listed in the Return codes column.
	-2 (FCAI_RESULT_IE)	An interface error occurred. The interface error code is stored in the *stem*.FCAI_IE stem variable.
	-17 (FCAI_RESULT_UNUSABLESTEM)	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:



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 seconds 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 *z/OS Communications Server: IP User's Guide and Commands*)
fcaiMap.FCAI_ReplyCode	Server reply code or 0 if no reply (see FTPD reply codes in *z/OS Communications Server: IP and SNA Codes*)
fcaiMap.FCAI_SCMD	Subcommand code (see FTP subcommand codes in *z/OS Communications Server: IP User's Guide and Commands*)
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 *stem*.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.', 'get_fcai_map', 'fcaiMap.')
```

TERM request

Format:

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 *stem*.FCAI_IE stem variable.
	-3 (FCAI_RESULT_CEC)	A client error occurred. The client error code is stored in the *stem*.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')
```

FTP client API for REXX trace return codes

Table 31 describes the return codes that can be stored in the FCAI_TRACE_DATASET_RETCODE variable.

Table 31. FTP client API for REXX trace return codes

Code type	Code	Description
Success	0	No errors detected with the FTP client API for REXX 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 *z/OS MVS Diagnosis: Reference*
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 for REXX function” on page 441.

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.

FTP client API: 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 “FTP client API messages and replies” on page 466 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 from the stem variables set using a GET_FCAI_MAP request (see “GET_FCAI_MAP request” on page 460). For example, for the request FtpApi("fcai.", "get_fcai_map", "fcaimap."), the total number of lines of output are stored in the stem variable fcaimap.FCAI_NumberLines. The output is composed of the following 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 431, which shows message, reply, list, and trace output lines. See “FTP client API messages and replies” on page 466 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 426, some prompts affect status codes returned by the interface. See "Interpreting results from an interface request" on page 467 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 logging in after the connection 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 into 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 *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.

FTP client API 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 "FTP client API messages and replies" for an example of EZA2121I, which is the command prompt used by the FTP client API.

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 *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 H, "Related protocol specifications," on page 991 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 406.

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.
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 429).	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 *z/OS Communications Server: IP User's Guide and Commands* and Diagnosing FTP problems in the *z/OS Communications Server: IP Diagnosis Guide*.

Other values returned in FCAI_Result are listed along with their descriptions in "FTP Client Application Interface (FCAI) control block" on page 406.

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 406 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 427 and the MODE parameter definition in "SCMD" on page 424 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 470 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 436.

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 409 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 410.
- 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 409.

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 seconds to wait for the request to complete. The value 0 means not to use a timer on the request (wait until completion).
  - FCAI_ReqTimer is approximate.
  - 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 requests. 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.
- 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.
- Timer expiration during a QUIT generated by TERM causes TERM to return FCAI_Result_CliProcessKill.
- 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 424 and “POLL” on page 427 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:

- The value 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 (the value 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 436). 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 FTP client API for REXX function package. See "FTP client API for REXX function" on page 441 for information about trace records for the FTP client API for REXX function package.

Rule: If using the Trace Resolver facility, the trace should be activated by specifying the RESOLVER_TRACE environment variable to collect the trace information in a file or MVS dataset.

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
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: O=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 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 logging in 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 3indicates 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	FTP client API for REXX sample program

The FTPCAPIC FTP client API for C sample program is found in /usr/lpp/tcpip/samples/ftpcapic.c. The FTPCAPIJ FTP client API for Java sample program is found in /usr/lpp/tcpip/samples/ftpcapij.java.

Chapter 14. Network management interfaces

z/OS Communications Server provides information about network operations by supporting the following functions:

- Systems Management Facilities (SMF) records
- Programming interfaces that are called network management interfaces (NMIs)

For more information about the SMF record support, see "SMF records" on page 621.

Network monitor and management applications can use the network management interfaces to programmatically obtain information about both TCP/IP and VTAM processing.

The z/OS Communications Server TCP/IP NMIs provide the following capabilities:

- Programmatically obtain copies of TCP/IP packet, OSAENTA, and data trace buffers, in real time, as the traces are collected
- Format or filter the TCP/IP packet trace, OSAENTA packet trace, or data trace records that are collected
- Obtain the following information:
  - Activation and deactivation events that are buffered for TCP connections in SMF format
  - Event information that is buffered for the FTP and TN3270 clients and servers in SMF format
  - Event information that is buffered for IP security in SMF format; information is provided from the IKE daemon and from the TCP/IP stack
  - 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
  - TCP/IP profile information and profile change information, which is buffered; this information is provided in SMF event records
  - CSSMTP information that is provided in SMF event records
  - Event information that is buffered for dynamic virtual IP addresses (DVIPAs) and sysplex distributor targets in SMF format
- Control the following filters and associations:
  - 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 *z/OS Communications Server: IP Configuration Guide* for more information about network security services.
- Monitor the following functions by using a callable API:
  - TCP connection and UDP endpoint activity
  - TCP/IP storage usage
  - TN3270E Telnet server connection performance
  - TCP/IP sysplex networking data
  - TCP/IP stack profile statement settings

  - TCP/IP interface attributes, statistics, and global stack statistics
- Drop one or multiple TCP connections or UDP endpoints

The z/OS Communications Server VTAM NMIs provide the following functions:

- 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 that is provided by these interfaces can be obtained from other types of documented interfaces that are provided by z/OS Communications Server such as SNMP, 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 by 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 637 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 represents the IP component of z/OS Communications Server and the term VTAM represents 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 (NSS) network management NMI" on page 538.

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.

Tip: If you are processing IPSec SMF records, there are some structures that were designed to be analogous to IPSec NMI structures. If you have code to process these structures, you might not need to write new parsing code. The section names are indicated in the individual SMF records and are described in detail in Appendix E, "Type 119 SMF records," on page 743.

The terms phase 1 and phase 2 refer to different types of security associations (SAs) that the z/OS IKE daemon can negotiate with its peers. Although the specific terminology for these types of security associations differs between the IKE version 1 and IKE version 2 protocols, the terms phase 1 and phase 2 refers to both versions. IKE terminology includes the following definitions:

Phase 1 security association (SA)
Refers to IKE version 1 phase 1 SAs and IKE version 2 IKE SAs. When a specific version is intended, that version is identified in this document.

Phase 2 security association (SA)
Refers to IKE version 1 phase 2 SAs and IKE version 2 child SAs. When a specific version is intended, that version is identified in this document.

Local IPSec NMI: 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 SERVAUTH resource name 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 that is to be used by other parties, include instructions in your documentation that indicating that administrators must define and give appropriate permission to the given security resource to use that feature; if the resource is not defined, indicating 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.

Local IPSec NMI: 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" on page 483).

The initialization message contains only a message header (see "IPSec NMI initialization and termination messages" on page 535). 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.

Result: The IKE daemon does not send an INIT message to the client application until it has successfully connected to the Policy Agent.

The termination message also contains only a message header (see “IPSec NMI initialization and termination messages” on page 535). 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.

IPSec NMI 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++ structures are contained in the ezbnmsec.h and ezbnmiv2.h files, which are installed in the z/OS UNIX /usr/include directory, and contained as members EZBNMSEC and EZBNMIV2 of the SEZANMAC data set. The assembler macro is installed as member EZBNMSEA of the SEZANMAC data set.

IPSec NMI request and response 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 NMsecMessageHdr structure.

Table 32. NMsecMessageHdr 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).
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 “IPSec NMI request messages” on page 488 for a description of the request types.

Table 32. NMsecMessageHdr structure (continued)

Field	Offset	Length	Format	Description
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 "Network security services NMI return and reason codes" on page 543).
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 "Network security services NMI return and reason codes" on page 543).
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 NMsecInRecDesc structure. See Table 33 on page 485 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 NMsecOutRecDesc structure. See Table 34 on page 485 for details.

Table 32. NMsecMessageHdr 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.

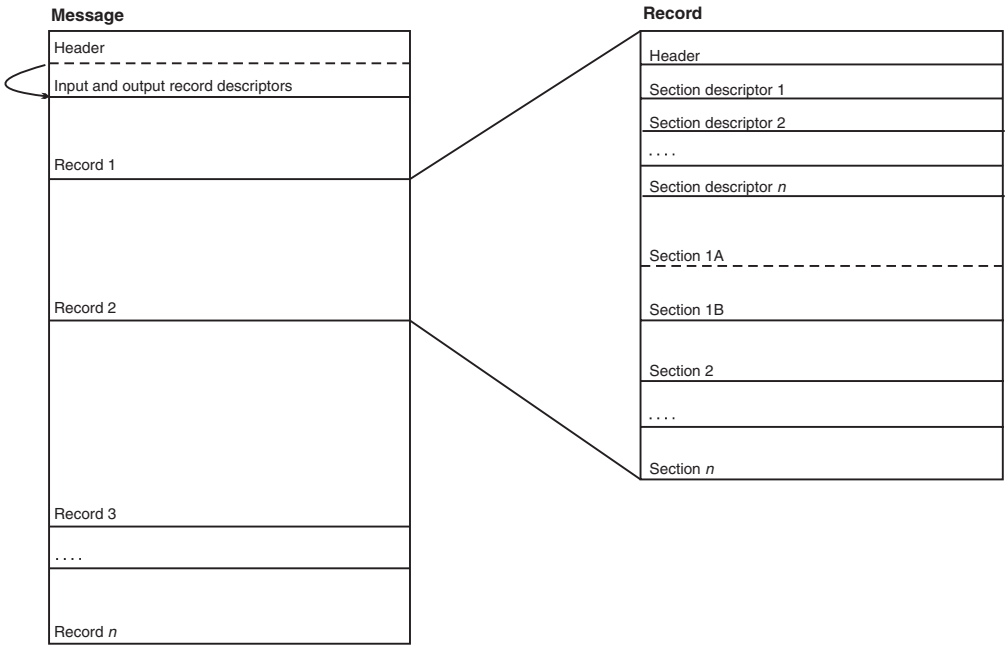


Figure 9. Message header and records

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 486, 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 486, 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 "IPSec NMI request messages" for details about the format the records and secondary records for each request.

IPSec NMI 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 and defensive filtering configuration information for a given TCP/IP stack, or optionally obtain this 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 from a particular stack about the currently active IP filters. These filters can be either the default IP security filters (filters that originate from the TCP/IP profile) or the

policy IP security filters (filters that originate from Policy Agent). Any defensive filters that are installed are also included.

- NMsec_GET_IPFLTDEFAULT (5)—Retrieve detailed information from a particular stack about the default IP security filters (filters that originate from the TCP/IP profile).

  Result: The default IP security filters are returned, regardless of whether they comprise the currently active filter set that is in use by the stack.
- NMsec_GET_IPFLTPOLICY (6)—Retrieve detailed information from a particular stack about the policy IP security filters (filters that originate from Policy Agent).

  Results:

  - The policy IP security 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 security filters in the stack, then a message that contains no 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.

IPSec NMI monitoring request format

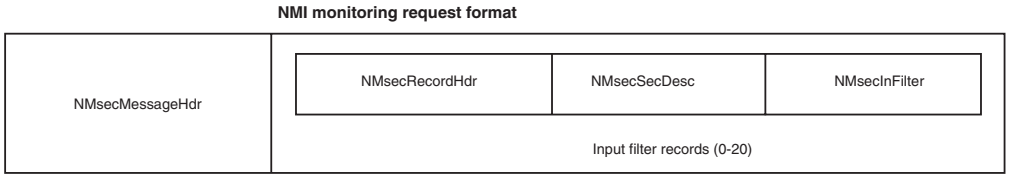


Figure 10. NMI monitoring request format

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 491). 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 491 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	CLIENT INFO
NMsFltSrcAddr4 NMsFltSrcAddr6			x			x	x				
NMsFltDstAddr4 NMsFltDstAddr6			x	x		x	x				
NMsFltProtocol			x	x		x	x				
NMsFltSrcPort			x			x	x				
NMsFltDstPort			x	x		x	x				
NMsFltLclEndpt4 NMsFltLclEndpt6					x	x	x	x			
NMsFltRmtEndpt4 NMsFltRmtEndpt6					x	x	x	x			
NMsFltTunnelID			x		x	x	x	x			
NMsFltObjName			x (filter name)		x (VPN action)	x (VPN action)	x (VPN action)	x (Key exchange rule)			
NMsFltObjGroupName			x (filter group name)								
NMsFltAssocName			x (VPN action)			x (Dyn VPN rule)	x (Dyn VPN rule)				
NMsFltFlagIPFltType			x								
NMsFltSAState					x	x	x	x			
NMsFltSWSAShadow			x			x					
NMsFltFlagDiscipline											x

Restriction: The NMsFltFlagDiscipline filter, which is part of the network security services NMI, can be used only with the NMsec_GET_CLIENTINFO request to the NSS server. See "Network security services NMI response messages" on page 540.

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.
NMsFltFlagSAState	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 NMsIPFltType*xxx* bits indicate the types of IP filters that match the input filter. If not set, the NMsIPFltType*xxx* bits are ignored and IP filters of any type match. Details about IP filter mask types are listed in this table.
NMsFltFlagDiscipline	1, bit 7	1 bit	Binary	Discipline indicator. If set, the input filter specifies a discipline for filtering.
NMsFltRsvd1	1, bit 8	16 bits	Binary	Reserved bits. Must be set to 0.

Table 39. NMsecInFilter structure (continued)

Field	Offset	Length	Format	Description
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.
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.

Table 39. NMsecInFilter structure (continued)

Field	Offset	Length	Format	Description
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.
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 security filters and defensive filters. If not set, generic IP security filters and defensive 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.

Table 39. NMsecInFilter structure (continued)

Field	Offset	Length	Format	Description
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	1 bit	Binary	Reserved bit. Must be set to 0.
NMsFltDisciplineIPSec	269, bit 0	1 bit	Binary	Discipline filter mask. If set for an NSS client info request, this input filter matches NSS clients that are registered for the IPSec discipline.
NMsFltDisciplineXMLApp	269, bit 1	1 bit	Binary	Discipline filter mask. If set for an NSS client info request, this input filter matches NSS clients that are registered for the XMLAppliance discipline.
NMsFltDisciplineRsvd	269, bit 2	6 bits	Binary	Reserved bits. Must be set to 0.
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 **NMsec_SASTATE_HALF_CLOSED (6)** Dynamic tunnel is no longer being used by the local endpoint but the delete process has not been acknowledged by the remote endpoint. Applies to IKEv2 tunnels only.

Table 39. NMsecInFilter structure (continued)

Field	Offset	Length	Format	Description
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.

IPSec NMI 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. **NMsec_TUNSTATUS_NODATAOFFER (7)** Applicable only to dynamic tunnel activation, this status indicates that the IpDataOffers defined on the dynamic IPSec IpFilterRule, corresponding to the requested LocalDynVpnRule, could not be used.
NMsTunRsvd1	97	3 bytes	Binary	Reserved. Must be set to 0.

NMsec_ACTIVATE_IPTUNMANUAL

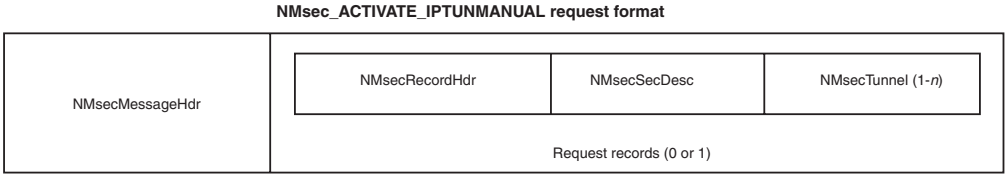


Figure 11. NMsec_ACTIVATE_IPTUNMANUAL request form

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 496). 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

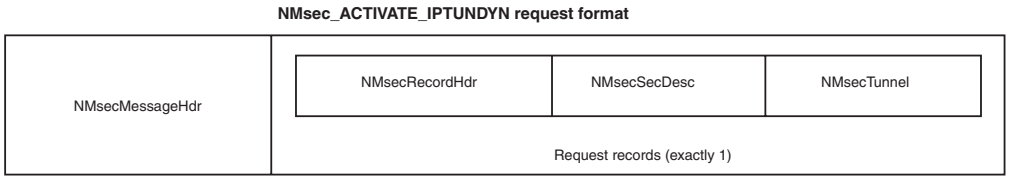


Figure 12. NMsec_ACTIVATE_IPTUNDYN request format

Activates a dynamic IPSec tunnel. Each record has one section, NMsecTunnel (described in Table 40 on page 496). The NMsecTunnel section identifies the dynamic tunnel that is to be activated.

NMsec_DEACTIVATE_IPTUNMANUAL

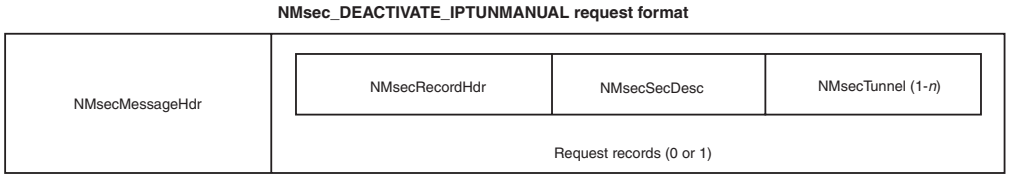


Figure 13. NMsec_DEACTIVATE_IPTUNMANUAL request format

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 496). 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

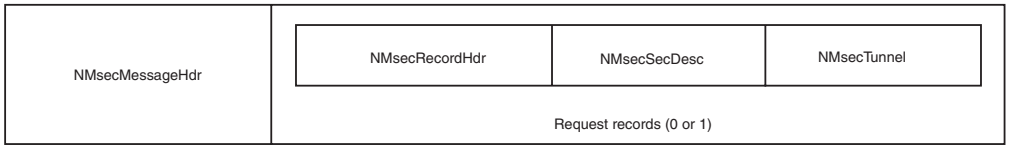


Figure 14. NMsec_DEACTIVATE_IPTUNDYN request format

Deactivates one or all dynamic tunnels. The request format contains zero or one record with one section, NMsecTunnel (described in Table 40 on page 496). 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

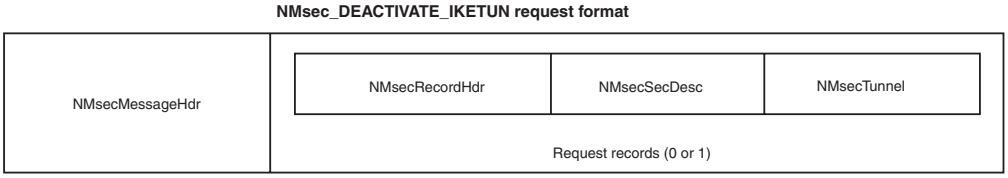


Figure 15. NMsec_DEACTIVATE_IKETUN request format

Deactivates one or all IKE tunnels. The request format contains zero or one record with one section, NMsecTunnel, (described in Table 40 on page 496). 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

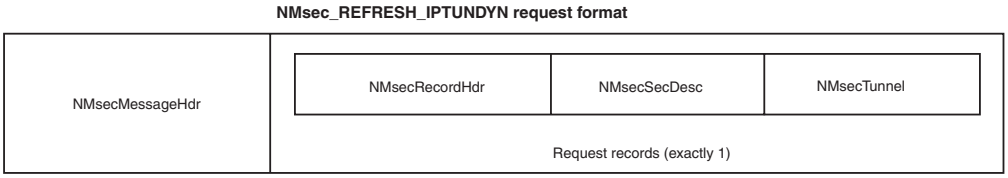


Figure 16. NMsec_REFRESH_IPTUNDYN request format

Refreshes a dynamic IPSec tunnel. Contains a single record that has one section, NMsecTunnel (described in Table 40 on page 496). The NMsecTunnel section identifies the dynamic tunnel to be refreshed.

NMsec_REFRESH_IKETUN

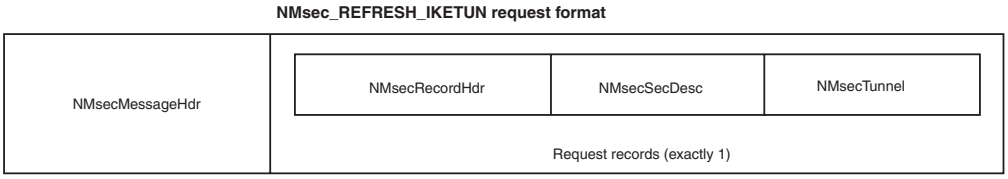


Figure 17. NMsec_REFRESH_IKETUN request format

Refreshes an IKE tunnel. Contains a single record that has one section, NMsecTunnel (described in Table 40 on page 496). 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. The field can have one of the following values: NMsec_FLT_POLICY (1) NMsec_FLT_DEFAULT (0)
NMsPolSrcRsvd1	1	3 bytes	Binary	Reserved. Set to zeroes.

IPSec NMI 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

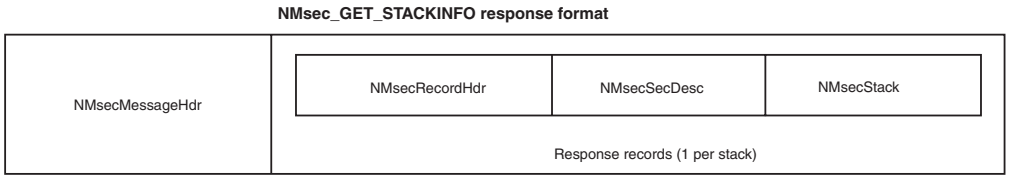


Figure 18. NMsec_GET_STACKINFO response format

Each record returned identifies a single stack that is active on the system. Each record has the following sections:

- One section, NMsecStack, describes attributes of the stack.

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.
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.
NMsStackFIPS140	0, bit 6	1 bit	Binary	FIPS 140 mode indicator. If this field is set, cryptographic operations for this stack are performed by using cryptographic algorithms and modules that are designed to meet the FIPS 140 requirements; otherwise, cryptographic algorithms and modules that do not meet the FIPS 140 requirements might be used.
NMsStackRsvd1	0, bit 7	25 bits	Binary	Reserved bits.
NMsStackName	4	24 bytes	EBCDIC	The job name of the TCP/IP stack.
NMsStackNATKeepAlive	28	4 bytes	Binary	NAT keepalive interval, in seconds, used to regulate sending NAT keepalive 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.

Table 42. NMsecStack structure (continued)

Field	Offset	Length	Format	Description
NMsStackDefFltCount	36	4 bytes	Binary	Number of defensive filters that are currently installed in the TCP/IP stack.
NMsStackDefFltMode	40	1 byte	Binary	Defensive filtering mode. Possible values are: **NMsec_DEFFLT_INACTIVE (0)** Defensive filtering is inactive for the stack. **NMsec_DEFFLT_ACTIVE (1)** Defensive filtering is active for the stack. The filter mode of block or simulate that is specified in the individual defensive filters is honored. **NMsec_DEFFLT_SIMULATE (2)** Defensive filtering is active for the stack. The filter mode, simulate, overrides the mode that is specified in the individual defensive filters.
NMsStackRsvd2	41	3 bytes	Binary	Reserved bytes.

- Zero to ten NMsecStackExclAddr sections that contain the defensive filtering exclusion list.

Table 43. NMsecStackExclAddr structure

Field	Offset	Length	Format	Description
NMsStackExclAddrFlagIsSingle	0, bit 0	1 bit	Binary	Single exclusion address indicator. If set, the exclusion address is indicated by the NMsStackExclAddr4 or NMsStackExclAddr6 field.
NMsStackExclAddrFlagIsPrefix	0, bit 1	1 bit	Binary	Prefixed exclusion address indicator. If set, the exclusion address is indicated by the NMsStackExclAddr4 or NMsStackExclAddr6 field, and the exclusion address prefix length is indicated by the NMsStackExclAddrPrefix field.
NMsStackExclAddrFlagIPv6	0, bit 2	1 bit	Binary	IPv6 indicator. If set, exclusion addresses are IPv6; otherwise they are IPv4.
NMsStackExclAddrRsvd1	0, bit 3	5 bits	Binary	Reserved bits.
NMsStackExclAddrRsvd2	1	2 bytes	Binary	Reserved.
NMsStackExclAddrPrefix	3	1 byte	Binary	If the NMsStackExclAddrIsPrefix field is set, this value is the length of the defensive filter exclusion address prefix, in bits.

Table 43. NMsecStackExclAddr structure (continued)

Field	Offset	Length	Format	Description
NMsStackExclAddr4	4	4 bytes	Binary	If the NMsStackExclAddrFlagIsSingle field is set, this value is a defensive filter IPv4 or IPv6 exclusion address. If the NMsStackExclAddrFlagIsPrefix field is set, this value is a defensive filter IPv4 or IPv6 exclusion address base
NMsStackExclAddr6	4	16 bytes	Binary	

NMsec_GET_SUMMARY

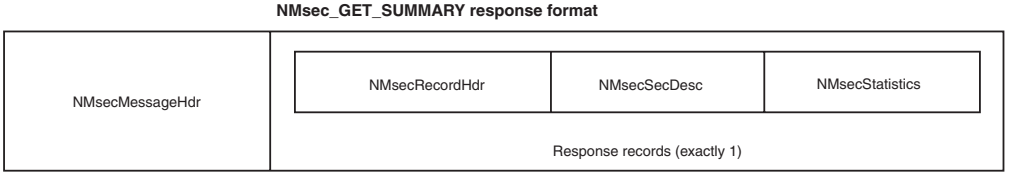


Figure 19. NMsec_GET_SUMMARY response format

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 44. NMsecStatistics 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.

Table 44. NMsecStatistics structure (continued)

Field	Offset	Length	Format	Description
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.
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.

Table 44. NMsecStatistics structure (continued)

Field	Offset	Length	Format	Description
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.
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.

Table 44. NMsecStatistics structure (continued)

Field	Offset	Length	Format	Description
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 and manual 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 and manual 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, IPSec filters, and defensive filters.

NMsec_GET_IPFLTCURR, NMsec_GET_IPFLTDEFAULT, and NMsec_GET_IPFLTPOLICY

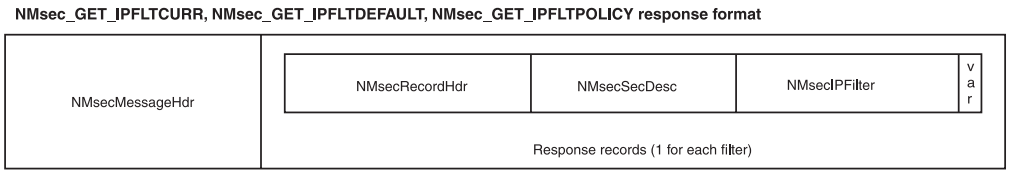


Figure 20. NMsec_GET_IPFLTCURR, NMsec_GET_IPFLTDEFAULT, and NMsec_GET_IPFLTPOLICY response format

For the requested stack, zero or more records, which represent IP filters, are returned. Each record that is returned identifies a single IP filter and contains two sections that describe the data. 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. Defensive filters are presented based

on the order in which they were added to the stack, most recent to least recent. Some IP filters might be absent from the result list because of input filters provided on the request message.

- One section, NMsecIPFilter, describes the basic properties of an IP filter. This section contains the following data.

Table 45. 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, which is 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.
NMsIPFltFlagSrcIsRange	240, bit 5	1 bit	Binary	Ranged source address indicator. If set, the source address range is indicated by the NMsIPFltSrcAddr4 and NMsIPFltSrcAddrRange4 fields, or the NMsIPFltSrcAddr6 and NMsIPFltSrcAddrRange6 fields.
NMsIPFltFlagDstIsSingle	240, bit 6	1 bit	Binary	Single destination address indicator. If set, the destination address is indicated by the NMsIPFltDstAddr4 or NMsIPFltDstAddr6 field.
NMsIPFltFlagDstIsPrefix	240, bit 7	1 bit	Binary	Prefixed destination address indicator. If set, the destination address is indicated by the NMsIPFltDstAddr4 or NMsIPFltDstAddr6 field, and the destination address prefix is indicated by the NMsIPFltDstAddrPrefix field.
NMsIPFltFlagDstIsRange	241, bit 0	1 bit	Binary	Ranged destination address indicator. If set, the destination address range is indicated by the NMsIPFltDstAddr4 and NMsIPFltDstAddrRange4 fields, or the NMsIPFltDstAddr6 and NMsIPFltDstAddrRange6 fields.
NMsIPFltFlagProtoDef	241, bit 1	1 bit	Binary	Protocol indicator. If set, the filter protocol is indicated by the NMsIPFltProtocol field, otherwise, the filter applies to all protocols.
NMsIPFltFlagSrcPortDef	241, bit 2	1 bit	Binary	Source port indicator. If set, the source port range is indicated by the NMsIPFltSrcPort and NMsIPFltSrcPortRange fields; otherwise, the filter applies to all source ports. This indicator is not valid and has the value 0 if the filter protocol is not TCP or UDP.
NMsIPFltFlagDstPortDef	241, bit 3	1 bit	Binary	Destination port indicator. If set, the destination port range is indicated by the NMsIPFltDstPort and NMsIPFltDstPortRange fields; otherwise, the filter applies to all destination ports. This indicator is not valid and has the value 0 if the filter protocol is not TCP or UDP.
NMsIPFltFlagICMPTypeDef	241, bit 4	1 bit	Binary	ICMP type indicator. If set, the ICMP type is indicated by the NMsIPFltICMPType field; otherwise, the filter applies to all ICMP types. This indicator is not valid and has the value 0 if the filter protocol is not ICMP or ICMPv6.

Table 45. NMsecIPFilter structure (continued)

Field	Offset	Length	Format	Description
NMsIPFltFlagICMPCodeDef	241, bit 5	1 bit	Binary	ICMP code indicator. If set, the ICMP code is indicated by the NMsIPFltICMPCode field; otherwise, the filter applies to all ICMP codes. This indicator is not valid and has the value 0 if the filter protocol is not ICMP or ICMPv6.
NMsIPFltFlagOSPFTypeDef	241, bit 6	1 bit	Binary	OSPF type indicator. If set, the OSPF type is indicated by the NMsIPFltOSPFType field; otherwise, the filter applies to all OSPF types. This indicator is not valid and has the value 0 if the filter protocol is not OSPF.
NMsIPFltFlagSrcAddrPktGran	241, bit 7	1 bit	Binary	Source address granularity indicator. If set for a dynamic anchor filter, on-demand activations use the packet source address; otherwise, they use the filter source address specification.
NMsIPFltFlagDstAddrPktGran	242, bit 0	1 bit	Binary	Destination address granularity indicator. If set for a dynamic anchor filter, on-demand activations use the packet destination address; otherwise, they use the filter destination address specification.
NMsIPFltFlagProtoPktGran	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.
NMsIPFltFlagSrcPortPktGran	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, when possible.
NMsIPFltFlagDstPortPktGran	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, when possible.
NMsIPFltFlagNATDetect	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.
NMsIPFltFlagNAPTDetect	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 exists but that it is detected only as a NAT.
NMsIPFltFlagGWDetect	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.
NMsIPFltFlagLogPermit	242, bit 7	1 bit	Binary	LogPermit indicator. If set, permitted packets that match this filter are logged.
NMsIPFltFlagLogDeny	243, bit 0	1 bit	Binary	LogDeny indicator. If set, denied packets that match this filter are logged.
NMsIPFltFlagMIPv6TypeDef	243, bit 1	1 bit	Binary	MIPv6 type indicator. If set, MIPv6 type is indicated by NMsIPFltMIPv6Type; otherwise, the filter applies to all MIPv6 types. This indicator is not valid and has the value 0 if the filter protocol is not MIPv6.
NMsIPFltFlagProtoOpaque	243, bit 2	1 bit	Binary	Opaque protocol indicator. If set, the filter matches packets that have an indeterminate protocol.
NMsIPFltFlagDiscardICMP	243, bit 3	1 bit	Binary	ICMP error indicator. If set and packets are discarded as a result of this filter rule, ICMP or ICMPv6 destination unreachable messages are sent to the packet origin, which indicates that the packet was administratively prohibited.
NMsIPFltFlagFragmentsOnly	243, bit 4	1 bit	Binary	Fragment indicator. If set, the filter matches fragmented packets. If clear, the filter matches both fragmented and non-fragmented packets.
NMsIPFltDefensiveGlobal	243, bit 5	1 bit	Binary	Defensive global indicator. If set for a defensive filter, the filter has a global scope. Not set for non-defensive filters.
NMsIPFltFlagTransOpaque	243, bit 6	1 bit	Binary	Opaque transport selector indicator. If set, the filter matches packets that have indeterminate transport layer selectors (for example, port, type, or code).
NMsIPFltFlagMIPv6TypePktGran	243, bit 7	1 bit	Binary	MIPv6 type granularity indicator. If set for a dynamic anchor filter, on-demand activations use packet MIPv6 type value; otherwise, they use the filter MIPv6 type specification, when possible.

Table 45. NMsecIPFilter structure (continued)

Field	Offset	Length	Format	Description
NMsIPFltType	244	1 byte	Binary	IP filter type. The field can have one of the following values: 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) NMsec_IPFLT_DEFENSIVE (8)
NMsIPFltState	245	1 byte	Binary	IP filter state. The field can have one of the following values: **NMsec_IPFLT_INACTIVE (0)** Filter is inactive as a result of a time condition. **NMsec_IPFLT_ACTIVE (1)** Filter is active.
NMsIPFltAction	246	1 byte	Binary	IP filter action. The field can have one of the following values: NMsec_IPFLT_PERMIT (1) NMsec_IPFLT_DENY (2) NMsec_IPFLT_IPSEC (3) NMsec_IPFLT_DEFENSIVE_SIMULATE (4)
NMsIPFltScope	247	1 byte	Binary	IP filter scope. The field can have one of the following values: NMsec_IPFLT_LOCAL (1) NMsec_IPFLT_ROUTED (2) NMsec_IPFLT_SCOPEALL (3)
NMsIPFltDirection	248	1 byte	Binary	IP filter direction. The field can have one of the following values: NMsec_IPFLT_INBOUND (1) NMsec_IPFLT_OUTBOUND (2)
NMsIPFltSecurityClass	249	1 byte	Binary	IP filter security class. Valid values are in the range 0 - 255. The value 0 matches all security classes.
NMsIPFltTCPConnect	250	1 byte	Binary	TCP connect qualifier. The field can have one of the following values: NMsec_IPFLT_CONNECT_NONE (0) NMsec_IPFLT_CONNECT_INBOUND (1) NMsec_IPFLT_CONNECT_OUTBOUND (2)
NMsIPFltFlagICMPTypePktGran	251, bit 0	1 bit	Binary	ICMP type granularity indicator. If set for a dynamic anchor filter, on-demand activations use packet ICMP type value; otherwise, they use the filter ICMP type specification, when possible.
NMsIPFltFlagICMPCodePktGran	251, bit 1	1 bit	Binary	ICMP code granularity indicator. If set for a dynamic anchor filter, on-demand activations use packet ICMP code value; otherwise, they use the filter ICMP code specification, when possible.
NMsIPFltFlagICMPv6TypePktGran	251, bit 2	1 bit	Binary	ICMPv6 type granularity indicator. If set for a dynamic anchor filter, on-demand activations use packet ICMPv6 type value; otherwise, they use the filter ICMPv6 type specification, when possible.
NMsIPFltFlagICMPv6CodePktGran	251, bit 3	1 bit	Binary	ICMPv6 code granularity indicator. If set for a dynamic anchor filter, on-demand activations use packet ICMPv6 code value; otherwise, they use the filter ICMPv6 code specification, when possible.
NMsIPFltRsvd2	251, bit 4	4 bits	Binary	Reserved
NMsIPFltProtocol	252	1 byte	Binary	IP filter protocol number, if the NMsIPFltFlagProtoDef field is set. This value corresponds to the IP protocol number in the IPv4 or IPv6 header.
NMsIPFltICMPType	253	1 byte	Binary	ICMP type, if the NMsIPFltFlagICMPTypeDef field is set.
NMsIPFltICMPCode	254	1 byte	Binary	ICMP code, if the NMsIPFltFlagICMPCodeDef field is set.
NMsIPFltOSPFType	255	1 byte	Binary	OSPF type, if the NMsIPFltFlagOSPFTypeDef field is set.

Table 45. NMsecIPFilter structure (continued)

Field	Offset	Length	Format	Description
NMsIPFltSrcPort	256	2 bytes	Binary	Low end of IP filter source port range, if the NMsIPFltFlagSrcPortDef field is set.
NMsIPFltSrcPortRange	258	2 bytes	Binary	High end of IP filter source port range, if the NMsIPFltFlagSrcPortDef field is set.
NMsIPFltDstPort	260	2 bytes	Binary	Low end of IP filter destination port range, if NMsIPFltFlagDstPortDef field is set.
NMsIPFltDstPortRange	262	2 bytes	Binary	High end of IP filter destination port range, if the NMsIPFltFlagDstPortDef field is set.
NMsIPFltSrcAddr4	264	4 bytes	Binary	The field can have one of the following values: • If the NMsIPFltFlagSrcIsSingle field is set, the filter's IPv4 or IPv6 source address • If the NMsIPFltFlagSrcIsPrefix field is set, the filter's IPv4 or IPv6 source address base • If the NMsIPFltFlagSrcIsRange field is set, the low end of the filter's IPv4 or IPv6 source address range
NMsIPFltSrcAddr6	264	16 bytes	Binary	
NMsIPFltSrcAddrRange4	280	4 bytes	Binary	If the NMsIPFltFlagSrcIsRange field is set, the high end of the filter's IPv4 or IPv6 source address range.
NMsIPFltSrcAddrRange6	280	16 bytes	Binary	
NMsIPFltDstAddr4	296	4 bytes	Binary	The field can have one of the following values: • If the NMsIPFltFlagDstIsSingle field is set, the filter's IPv4 or IPv6 destination address • If the NMsIPFltFlagDstIsPrefix field is set, the filter's IPv4 or IPv6 destination address base • If the NMsIPFltFlagDstIsRange field is set, the low end of the filter's IPv4 or IPv6 destination address range
NMsIPFltDstAddr6	296	16 bytes	Binary	
NMsIPFltDstAddrRange4	312	4 bytes	Binary	If the NMsIPFltFlagDstIsRange field is set, the high end of the filter's IPv4 or IPv6 destination address range.
NMsIPFltDstAddrRange6	312	16 bytes	Binary	
NMsIPFltSrcAddrPrefix	328	1 byte	Binary	If the NMsIPFltFlagSrcIsPrefix field is set, the length of the filter's source address prefix, in bits.
NMsIPFltDstAddrPrefix	329	1 byte	Binary	If the NMsIPFltFlagDstIsPrefix field is set, the length of the filter's destination address prefix, in bits.
NMsIPFltRsvd3	330	1 byte	Binary	Reserved.
NMsIPFltNATTClientIDType	331	1 byte	Binary	The NATT client ID (client traffic selector) is present only when the peer is behind a NAT and a gateway, and the peer supplied a client ID. The field can have one of the following values: **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.
NMsIPFltNATTClientIDIP	332	4 bytes	Binary	If NATT client ID (client traffic selector) type is NMsec_IPFLT_IDIP, NMsec_IPFLT_IDRANGE, or NMsec_IPFLT_IDPREFIX, this field is the base IPv4 address for the client ID.
NMsIPFltNATTClientIDHash	332	16 bytes	Binary	If NATT client ID (client traffic selector) type is NMsec_IPFLT_IDOTHER, this field is the MD5 hash of the client's ID.
NMsIPFltNATTClientIDIP2	348	4 bytes	Binary	If NATT client ID (client traffic selector) type is NMsec_IPFLT_IDRANGE, this field is the high end of the IPv4 client ID address range.

Table 45. NMsecIPFilter structure (continued)

Field	Offset	Length	Format	Description
NMsIPFltNATTClientIDPrefix	348	4 bytes	Binary	If NATT client ID (client traffic selector) type is NMsec_IPFLT_IDPREFIX, this field is the prefix length of the IPv4 client ID, in bits.
NMsIPFltRsvd4	348	16 bytes	Binary	Reserved.
NMsIPFltNATTPeerPort	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.
NMsIPFltNATTNRFOrigPort	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.
NMsIPFltMismatch	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.
NMsIPFltMatch	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.
NMsIPFltCreateTime	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. For a defensive filter, this field contains the UNIX time stamp that indicates when the defensive filter was created.
NMsIPFltUpdateTime	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. For a defensive filter, this field contains the UNIX time stamp that indicates when the defensive filter's attributes were last updated.
NMsIPFltMIPv6Type	392	1 byte	Binary	MIPv6 type, if NMsIPFltFlagMIPv6TypeDef is set.
NMsIPFltTypeRange	393	1 byte	Binary	High end of ICMP, ICMPv6, or MIPv6 type range, if the corresponding flag is set.
NMsIPFltCodeRange	394	1 byte	Binary	High end of ICMP or ICMPv6 code range, if the corresponding flag is set.
NMsIPFltRemoteIdType	395	1 byte	Binary	ISAKMP identity type for the remote security endpoint identity, as defined in RFC 2407. ISAKMP peers exchange and verify their identities as part of the IKE tunnel (phase 1) negotiation. These identities can be associated with anchor filters, dynamic filters, or NATT dynamic filters, and are used for filtering purposes. This field has the value 0 if the remote IKE identity is not present or if it is not applicable.
NMsIPFltLifetimeExpire	396	8 bytes	Binary	For a defensive filter, this field indicates the time at which the filter expires, in UNIX format; otherwise this field has the value 0 for all non-defensive filters.

- One variable-length section contains the contents of the filter's remote IKE identity. Regardless of the type of the identity, the identity is expressed as an EBCDIC string. An IP address is returned in printable form. A key ID is returned as an EBCDIC string of hex values. For a dynamic anchor filter, this represents the identity or wildcarded identities that are permitted for remote communication on this filter. For a dynamic or NATT dynamic filter, this represents the actual remote IKE identity if remote identity filtering is in use. For all other filters, this section is empty. This section is also empty for SWSA shadow filters.

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 500 for the NMsec_GET_STACKINFO request.

NMsec_GET_PORTTRAN

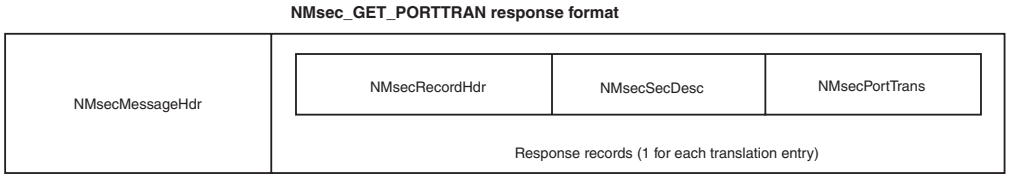


Figure 21. NMsec_GET_PORTTRAN response format

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 46. 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

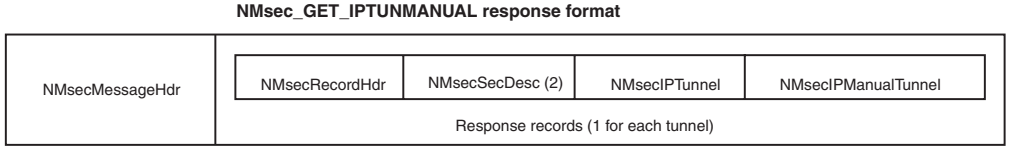


Figure 22. NMsec_GET_IPTUNMANUAL response format

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 47. 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
NMsIPTunFIPS140	96, bit 1	1 bit	Binary	FIPS 140 mode indicator. If this field is set, cryptographic operations for this tunnel are performed using cryptographic algorithms and modules that are designed to meet the FIPS 140 requirements; otherwise, cryptographic algorithms and modules that do not meet the FIPS 140 requirements might be used.
NMsIPTunRsvd1	96, bit 2	30 bits	Binary	Reserved bits.
NMsIPTunType	100	1 byte	Binary	Tunnel type. The field can have one of the following values: **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. The field can have one of the following values: **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 **NMsec_SASTATE_HALF_CLOSED (6)** Dynamic tunnel is no longer being used by the local endpoint but the delete process has not been acknowledged by the remote endpoint. Applies to IKEv2 tunnels only.
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
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

Table 47. NMsecIPTunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIPTunEncapMode	136	1 byte	Binary	Tunnel encapsulation mode. The field can have one of the following values: 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. The field can have one of the following values: IPPROTO_AH (51) IPPROTO_ESP (50) This field is not defined if the tunnel state is NMsec_SASTATE_PENDING or NMsec_SASTATE_INCOMPLETE.

Table 47. NMsecIPTunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIPTunAuthAlg	138	1 byte	Binary	Tunnel authentication algorithm. This field is not defined if the tunnel state is NMsec_SASTATE_PENDING or NMsec_SASTATE_INCOMPLETE. The NMsIPTunAuthAlg field can have one of the following values: **NMsec_AUTH_NULL (0)** The tunnel uses NULL authentication, or obtains authentication using a combined-mode encryption algorithm. Also see the definition of the NMsIPTunEncryptAlg field. **NMsec_AUTH_HMAC_MD5 (38)** The tunnel uses HMAC-MD5 authentication with Integrity Check Value (ICV) truncation to 96 bits. **NMsec_AUTH_HMAC_SHA1 (39)** The tunnel uses HMAC-SHA1 authentication with ICV truncation to 96 bits. **NMsec_AUTH_HMAC_SHA2_256_128 (7)** The tunnel uses HMAC-SHA2-256 authentication with ICV truncation to 128 bits. **NMsec_AUTH_HMAC_SHA2_384_192 (13)** The tunnel uses HMAC-SHA2-384 authentication with ICV truncation to 192 bits. **NMsec_AUTH_HMAC_SHA2_512_256 (14)** The tunnel uses HMAC-SHA2-512 authentication with ICV truncation to 256 bits. **NMsec_AUTH_AES128_XCBC_96 (9)** The tunnel uses AES128-XCBC authentication with ICV truncation to 96 bits. **NMsec_AUTH_AES_GMAC_128 (4)** The tunnel uses AES-GMAC authentication with a key length of 128 bits. **NMsec_AUTH_AES_GMAC_256 (6)** The tunnel uses AES-GMAC authentication with a key length of 256 bits.

Table 47. NMsecIPTunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIPTunEncryptAlg	139	1 byte	Binary	Tunnel encryption algorithm. This field is not defined if the tunnel state is NMsec_SASTATE_PENDING or NMsec_SASTATE_INCOMPLETE. The NMsIPTunEncryptAlg field can have one of the following values: **NMsec_ENCR_NONE (0)** **NMsec_ENCR_NULL (11)** **NMsec_ENCR_DES (18)** **NMsec_ENCR_3DES (3)** **NMsec_ENCR_AES_CBC (12)** AES encryption algorithm in Cipher Block Chaining (CBC) mode. Also see the definition of the NMsIPTunEncryptKeyLength field, which identifies the key length in use. **NMsec_ENCR_AES_GCM_16 (20)** AES encryption algorithm in Galois/Counter Mode (GCM) using a 16-octet IV. Also see the definition of the NMsIPTunEncryptKeyLength field, which identifies the key length in use.
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.
NMsIPTunOutbEncryptSPI	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.
NMsIPTunStartTime	156	4 bytes	Binary	Tunnel start time. Indicates the time at which the tunnel was activated or refreshed, in UNIX format.
NMsIPTunEncryptKeyLength	160	4 bytes	Binary	Encryption key length, in bits for variable-length algorithms. This value is 0 for encryption algorithms that have a fixed key length, such as DES and 3DES, and is a nonzero value for encryption algorithms that have a variable key length, such as AES-CBC and AES-GCM. **Result:** Example values are 128 and 256.

- One section, NMsecIPManualTunnel, describes the attributes that are specific to a manual IP tunnel. This section contains the following data.

Table 48. NMsecIPManualTunnel structure

Field	Offset	Length	Format	Description
NMsIPManTunOutPkt	0	8 bytes	Binary	Outbound packet count for this tunnel
NMsIPManTunInPkt	8	8 bytes	Binary	Inbound packet count for this tunnel
NMsIPManTunOutBytes	16	8 bytes	Binary	Outbound byte count for this tunnel, representing the number of outbound data bytes protected by the tunnel
NMsIPManTunInBytes	24	8 bytes	Binary	Inbound byte count for this tunnel, representing the number of inbound data bytes protected by the tunnel
NMsIPManTunClearDF	32, bit 0	1 bit	Binary	Don't-fragment bit clear indicator. If this bit is set, the IPv4 tunnel mode tunnel clears the DF bit in the outer IP header. If neither the NMsIPManTunClearDF or NMsIPManTunSetDF value is set, the IPv4 tunnel mode tunnel passes through the DF bit from the inner IP header to the outer IP header. This field is not applicable and is always 0 for IPv6 or transport mode tunnels.
NMsIPManTunSetDF	32, bit 1	1 bit	Binary	Don't-fragment bit set indicator. If this bit is set, IPv4 the tunnel mode tunnel sets the DF bit in the outer IP header. If neither the NMsIPManTunClearDF or NMsIPManTunSetDF value is set, the IPv4 tunnel mode tunnel passes the DF bit through from the inner IP header to the outer IP header. This field is not applicable and is always 0 for IPv6 or transport mode tunnels.
NMsIPManTunClearDSCP	32, bit 2	1 bit	Binary	DSCP clear indicator. If this bit is set, tunnel mode tunnel clears the DSCP bit in the outer IP header. If the value of this bit is 0, the tunnel mode tunnel copies the DSCP field from the inner IP header to the outer IP header. This field is not applicable is always 0 for transport mode tunnels.
NMsIPManTunRsvd1	32, bit 3	29 bits	Binary	Reserved bits

NMsec_GET_IPTUNDYNSTACK

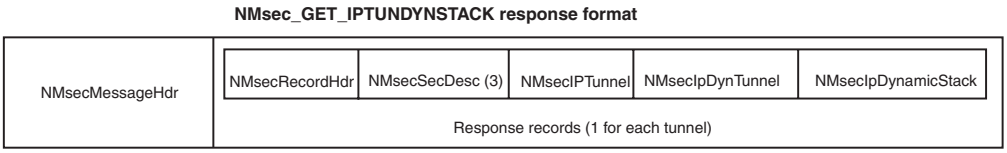


Figure 23. NMsec_GET_IPTUNDYNSTACK response format

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 47 on page 513.

One section, NMsecIPDynTunnel, describes the basic properties of a dynamic IP tunnel. This section contains the following data.

Table 49. NMsecIPDynTunnel structure

Field	Offset	Length	Format	Description
NMsIPDynUDPEncap	0, bit 0	1 bit	Binary	UDP encapsulation indicator. If set, the tunnel uses UDP encapsulation mode.
NMsIPDynLclNAT	0, bit 1	1 bit	Binary	Local NAT indicator. If set, a NAT has been detected in front of the local security endpoint.
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.

Table 49. NMsecIPDynTunnel structure (continued)

Field	Offset	Length	Format	Description
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.
NMsIPDynTransportOpaque	1, bit 5	1 bit	Binary	Opaque transport selector indicator. If set, the dynamic tunnel is protecting data traffic in which the upper layer selectors, source and destination ports, ICMP or ICMPv6 type, and code or IPv6 Mobility header type are not available as a result of fragmentation.
NMsIPDynRsvd1	1, bit 6	18 bits	Binary	Reserved bits.
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.

Table 49. NMsecIPDynTunnel structure (continued)

Field	Offset	Length	Format	Description
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. This field retains its original value for a refreshed tunnel
NMsIPDynActMethod	128	1 byte	Binary	Tunnel activation method. The field can have one of the following values: **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. This field retains its original value for a refreshed tunnel
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.
NMsIPDynSrcNATOA	136	4 bytes	Binary	Source NAT original IP address. NAT original IP addresses 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 an IKEv1 peer did not send a source NAT-OA payload, the value of this field is 0. **Restriction:** An IKEv1 peer at a NAT traversal support level that is prior to RFC3947 is not required to send a source NAT-OA payload.

Table 49. NMsecIPDynTunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIPDynDstNATOA	140	4 bytes	Binary	Destination NAT original IP address. NAT original IP addresses 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 an IKEv1 peer did not send a destination NAT-OA payload, the value of this field is 0. **Restriction:** An IKEv1 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	Low end of source port range for tunnel data, or 0 if the tunnel is not limited to TCP or UDP.
NMsIPDynDstPort	150	2 bytes	Binary	Low end of destination port range for tunnel data, or 0 if the tunnel is not limited to TCP or UDP.
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	
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.

Table 49. NMsecIPDynTunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIPDynMajorVer	218	1 byte	Binary	Major version of the IKE protocol that is in use. Only the low-order 4 bits are used.
NMsIPDynMinorVer	219	1 byte	Binary	Minor version of the IKE protocol that is in use. Only the low-order 4 bits are used.
NMsIPDynType	220	1 byte	Binary	Low end of the ICMP, ICMPv6, or MIPv6 type range for tunnel data, or 0 if the tunnel is not limited to ICMP, ICMPv6, or MIPv6.
NMsIPDynTypeRange	221	1 byte	Binary	High end of the ICMP, ICMPv6, or MIPv6 type range for tunnel data, or 0 if the tunnel is not limited to ICMP, ICMPv6, or MIPv6. A tunnel that applies to all type values is indicated as the range 0 - 255.
NMsIPDynCode	222	1 byte	Binary	Low end of ICMP or ICMPv6 code range for tunnel data, or 0 if the tunnel is not limited to ICMP or ICMPv6.
NMsIPDynCodeRange	223	1 byte	Binary	High end of ICMP or ICMPv6 code range for tunnel data, or 0 if the tunnel is not limited to ICMP or ICMPv6. A tunnel that applies to all code values is indicated as the range 0 - 255.
NMsIPDynSrcPortRange	224	2 bytes	Binary	High end of source port range for tunnel data, or 0 if the tunnel is not limited to TCP or UDP. A tunnel that applies to all source port values is indicated as the range 0 - 65 535.
NMsIPDynDstPortRange	226	2 bytes	Binary	High end of destination port range for tunnel data, or 0 if the tunnel is not limited to TCP or UDP. A tunnel that applies to all destination port values is indicated as the range 0 - 65 535.
NMsIPDynGeneration	228	4 bytes	Binary	Tunnel generation number. The first dynamic tunnel that has a particular tunnel ID is generation 1. Subsequent refreshes of this dynamic tunnel have the same tunnel ID but have higher generation numbers.

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 50. 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.
NMsIPDynStackClearDF	0, bit 1	1 bit	Binary	Don't-fragment bit clear indicator. If this bit is set, the IPv4 tunnel mode tunnel clears the DF bit in the outer IP header. If neither the NMsIPDynStackClearDF or the NMsIPDynStackSetDF value is set, the IPv4 tunnel mode tunnel passes the DF bit from the inner IP header to the outer IP header. This field is not applicable and is always 0 for IPv6 or transport mode tunnels.

Table 50. NMsecIPDynamicStack structure (continued)

Field	Offset	Length	Format	Description
NMsIPDynStackSetDF	0, bit 2	1 bit	Binary	Don't-fragment bit set indicator. If this bit is set, the IPv4 tunnel mode tunnel sets the DF bit in the outer IP header. If neither the NMsIPDynStackClearDF or the NMsIPDynStackSetDF value is set, the IPv4 tunnel mode tunnel passes the DF bit from the inner IP header to the outer IP header. This field is not applicable and is always 0 for IPv6 or transport mode tunnels.
NMsIPDynStackClearDSCP	0, bit 3	1 bit	Binary	DSCP clear indicator. If this bit is set, the tunnel mode tunnel clears the DSCP bit in the outer IP header. If this bit has the value 0, the tunnel mode tunnel copies the DSCP field from the inner IP header to the outer IP header. This field is not applicable and is always 0 for transport mode tunnels.
NmsIPDynStackRsvd1	0, bit 4	28 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

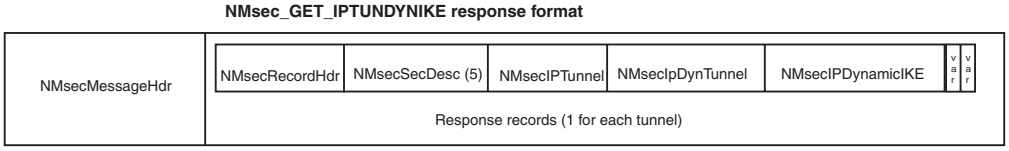


Figure 24. NMsec_GET_IPTUNDYNIKE response format

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 47 on page 513.
- One section, NMsecIPDynTunnel, describes the basic properties of a dynamic IP tunnel. The layout of this section is described in Table 49 on page 518.
- 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 51. NMsecIPDynamicIKE structure

Field	Offset	Length	Format	Description
NMsIPDynIKEIsPendingNew	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.
NMsIPDynIKERsvd1	0, bit 1	31 bits	Binary	Reserved bits.
NMsIPDynIKEFilter	4	48 bytes	EBCDIC	Filter name for the IP filter related to this dynamic tunnel.
NMsIPDynIKEDHGroup	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.
NMsIPDynIKELclIDType	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. The IKEv2 equivalent term for client ID is traffic selector. Although RFC 2407 pertains to IKEv1, section 4.6.2.1 in RFC 2407 can interpret this field value for both IKEv1 and IKEv2. See Appendix H, "Related protocol specifications," on page 991 for information about accessing RFCs.
NMsIPDynIKERmtIDType	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. The IKEv2 equivalent term for client ID is traffic selector. Although RFC 2407 pertains to IKEv1, section 4.6.2.1 in RFC 2407 can interpret this field value for both IKEv1 and IKEv2. See Appendix H, "Related protocol specifications," on page 991 for information about accessing RFCs.

Table 51. NMsecIPDynamicIKE structure (continued)

Field	Offset	Length	Format	Description
NMsIPDynIKEExtState	58	2 bytes	Binary	Extended tunnel state information. The field can have one of the following values: **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. Applies to IKEv1 tunnels only. **NMsec_P2STATE_PENDING_START (4)** Tunnel is awaiting the activation of an IKE tunnel to allow it to begin. See the description of the NMsIPTunState field in Table 47 on page 513 for more succinct state information. **NMsec_P2STATE_HALF_CLOSED (5)** Tunnel is no longer being used by the local endpoint but the delete process has not been acknowledged by the remote endpoint. Applies to IKEv2 tunnels only.

- 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

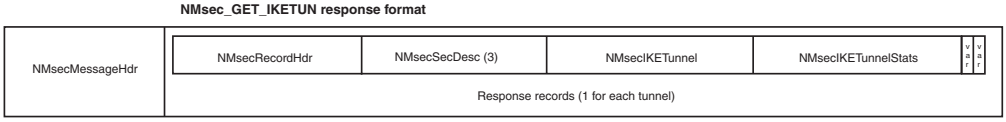


Figure 25. NMsec_GET_IKETUN response format

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 52. NMsecIKETunnel structure

Field	Offset	Length	Format	Description
NMsIKETunIPv6	0, bit 0	1 bit	Binary	IPv6 indicator. If set, the IKE tunnel security endpoints are IPv6 addresses, otherwise they are IPv4
NMsIKETunNATAllowed	0, bit 1	1 bit	Binary	NAT traversal indicator. If set, the NAT traversal function is enabled for this IKE tunnel.
NMsIKETunLclNAT	0, bit 2	1 bit	Binary	Local NAT indicator. If set, a NAT has been detected in front of the local security endpoint.
NMsIKETunRmtNAT	0, bit 3	1 bit	Binary	Remote NAT indicator. If set, a NAT has been detected in front of the remote security endpoint.
NMsIKETunRmtNAPT	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.
NMsIKETunCanInitP1	0, bit 5	1 bit	Binary	IKE tunnel (P1) initiation indicator. If this field is 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.
NMsIKETunFIPS140	0, bit 6	1 bit	Binary	FIPS 140 mode indicator. If this field is set, cryptographic operations for this IKE tunnel are performed using cryptographic algorithms and modules that are designed to meet the FIPS 140 requirements; otherwise, cryptographic algorithms and modules that do not meet the FIPS 140 requirements might be used.
NMsIKETunRsvd1	0, bit 7	25 bits	Binary	Reserved bits.
NMsIKETunID	4	48 bytes	EBCDIC	Tunnel ID for this IKE tunnel.
NMsIKETunKeyExchRule	52	48 bytes	EBCDIC	Key exchange rule name for this IKE tunnel.
NMsIKETunKeyExchAction	100	48 bytes	EBCDIC	Key exchange action name for this IKE tunnel.
NMsIKETunLclEndpt4	148	4 bytes	Binary	IPv4 or IPv6 local security endpoint for this IKE tunnel.
NMsIKETunLclEndpt6	148	16 bytes	Binary	
NMsIKETunRmtEndpt4	164	4 bytes	Binary	IPv4 or IPv6 remote security endpoint for this IKE tunnel.
NMsIKETunRmtEndpt6	164	16 bytes	Binary	
NMsIKETunICookie	180	8 bytes	Binary	The icookie for this IKE tunnel.
NMsIKETunRCookie	188	8 bytes	Binary	The rcookie for this IKE tunnel.
NMsIKETunExchangeMode	196	1 byte	Binary	Tunnel exchange mode. For IKEv1 SAs, the field can have one of the following values: NMsec_IKETUN_EXCHMAIN (2) NMsec_IKETUN_EXCHAGGRESSIVE (4) For IKEv2 SAs, this field is not applicable and the value will be 0.

Table 52. NMsecIKETunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIKETunState	197	1 byte	Binary	Tunnel state. The field can have one of the following values: **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. **NMsec_SASTATE_HALF_CLOSED (6)** Tunnel is no longer being used by the local endpoint but the delete process has not been acknowledged by the remote endpoint. Applies to IKEv2 tunnels only.
NMsIKETunAuthAlg	198	1 byte	Binary	Tunnel authentication algorithm. One of the following values: **NMsec_AUTH_HMAC_MD5 (38)** The tunnel uses HMAC-MD5 authentication with the full 128-bit Integrity Check Value (ICV). This value is applicable only to IKEv1 tunnels. **NMsec_AUTH_HMAC_SHA1 (39)** The tunnel uses HMAC-SHA1 authentication with the full 160-bit ICV. This value is applicable only to IKEv1 tunnels. **NMsec_AUTH_HMAC_MD5_96 (40)** The tunnel uses HMAC-MD5 authentication with ICV truncation to 96 bits. This value is applicable only to IKEv2 tunnels. **NMsec_AUTH_HMAC_SHA1_96 (41)** The tunnel uses HMAC-SHA1 authentication with ICV truncation to 96 bits. This value is applicable only to IKEv2 tunnels. **NMsec_AUTH_HMAC_SHA2_256_128 (7)** The tunnel uses HMAC-SHA2-256 authentication with ICV truncation to 128 bits. **NMsec_AUTH_HMAC_SHA2_384_192 (13)** The tunnel uses HMAC-SHA2-384 authentication with ICV truncation to 192 bits. **NMsec_AUTH_HMAC_SHA2_512_256 (14)** The tunnel uses HMAC-SHA2-512 authentication with ICV truncation to 256 bits. **NMsec_AUTH_AES128_XCBC_96 (9)** The tunnel uses AES128-XCBC authentication with ICV truncation to 96 bits.

Table 52. NMsecIKETunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIKETunEncryptAlg	199	1 byte	Binary	Tunnel encryption algorithm. The field can have one of the following values: **NMsec_ENCR_DES (18)** **NMsec_ENCR_3DES (3)** **NMsec_ENCR_AES_CBC (12)** AES encryption algorithm in Cipher Block Chaining (CBC) mode. Also see the NMsIKETunEncryptKeyLength field, which identifies the key length in use.
NMsIKETunDHGroup	200	4 bytes	Binary	Diffie-Hellman group used to generate keying material for this IKE tunnel.
NMsIKETunPeerAuthMethod	204	1 byte	Binary	Tunnel peer authentication method. The field can have one of the following values: NMsec_IKETUN_PRESHAREDKEY (3) NMsec_IKETUN_RSASIGNATURE (2) NMsec_IKETUN_ECDSA_256 (4) NMsec_IKETUN_ECDSA_384 (5) NMsec_IKETUN_ECDSA_521 (6)
NMsIKETunRole	205	1 byte	Binary	Tunnel role. The field can have one of the following values: NMsec_IKETUN_INITIATOR (1) NMsec_IKETUN_RESPONDER (2)
NMsIKETunNATTLevel	206	1 byte	Binary	NAT traversal support level. The field can have one of the following values: **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. **NMsec_IKETUN_NATTV2 (6)** RFC 5996 support with non-z/OS peer. **NMsec_IKETUN_NATTV2ZOS (7)** RFC 5996 support with z/OS peer.

Table 52. NMsecIKETunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIKETunExtState	207	1 byte	Binary	Extended tunnel state information. The field can have one of the following values: **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. **NMsec_P1STATE_WAIT_AUTH (6)** An SA authorization request is in progress. **NMsec_P1STATE_HALF_CLOSED (7)** Tunnel is no longer being used by the local endpoint but the delete process has not been acknowledged by the remote endpoint. Applies to IKEv2 tunnels only. See the NMsIKETunState field 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.
NMsIKETunRIDType	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.
NMsIKETunStartTime	232	4 bytes	Binary	Tunnel start time. Indicates the time at which the tunnel was activated or refreshed, in UNIX format.
NMsIKETunMajorVer	236	1 byte	Binary	Major version of the IKE protocol that is in use. Only the low-order 4 bits are used.

Table 52. NMsecIKETunnel structure (continued)

Field	Offset	Length	Format	Description
NMsIKETunMinorVer	237	1 byte	Binary	Minor version of the IKE protocol that is in use. Only the low-order 4 bits are used.
NMsIKETunPseudoRandomFunc	238	1 byte	Binary	Pseudo-random function that is used to seed keying material. The field can have one of the following values: • NMsec_AUTH_HMAC_MD5 (38) • NMsec_AUTH_HMAC_SHA1 (39) • NMsec_AUTH_HMAC_SHA2_256 (15) • NMsec_AUTH_HMAC_SHA2_384 (16) • NMsec_AUTH_HMAC_SHA2_512 (17) • NMsec_AUTH_AES128_XCBC (18)
NMsIKETunLocalAuthMethod	239	1 byte	Binary	The authentication method for the local endpoint. The field can have one of the following values: • NMsec_IKETUN_PRESHAREDKEY (3) • NMsec_IKETUN_RSASIGNATURE (2) • NMsec_IKETUN_ECDSA_256 (4) • NMsec_IKETUN_ECDSA_384 (5) • NMsec_IKETUN_ECDSA_521 (6) • NMsec_IKETUN_DS (7)
NMsIKETunReauthInterval	240	4 bytes	Binary	Re-authentication interval. Indicates the number of seconds between re-authentication operations.
NMsIKETunReauthTime	244	4 bytes	Binary	Tunnel re-authentication time. Indicates the time at which the tunnel is re-authenticated, in UNIX format.
NMsIKETunGeneration	248	4 bytes	Binary	Tunnel generation number. The first IKE tunnel that has a particular tunnel ID is generation 1. Subsequent refreshes of this IKE tunnel will have the same tunnel ID but will have higher generation numbers.
NMsIKETunEncryptKeyLength	252	4 bytes	Binary	Encryption key length for variable-length algorithms, in bits. This value is 0 for encryption algorithms that have a fixed key length, such as DES and 3DES, and is a nonzero value for encryption algorithms that have a variable key length, such as AES-CBC. **Result:** Example values are 128 and 256.

- One section, NMsecIKETunStats, indicates various counters and statistics for the IKE tunnel. This section contains the following data.

Table 53. IKE tunnel statistics

Field	Offset	Length	Format	Description
NMsIKETunP2Current	0	4 bytes	Binary	Current count of active dynamic tunnels that are associated with this IKE tunnel.
NMsIKETunP2InProgress	4	4 bytes	Binary	Current count of pending or in-progress dynamic tunnels that are associated with this IKE tunnel.
NMsIKETunP2LclActSuccess	8	4 bytes	Binary	Cumulative count of successful dynamic tunnel activations that were initiated locally for this IKE tunnel.
NMsIKETunP2RmtActSuccess	12	4 bytes	Binary	Cumulative count of successful dynamic tunnel activations that were initiated remotely for this IKE tunnel.
NMsIKETunP2LclActFailure	16	4 bytes	Binary	Cumulative count of failed dynamic tunnel activations that were initiated locally for this IKE tunnel.

Table 53. IKE tunnel statistics (continued)

Field	Offset	Length	Format	Description
NMsIKETunP2RmtActFailure	20	4 bytes	Binary	Cumulative count of failed dynamic tunnel activations that were initiated remotely for this IKE tunnel.
NMsIKETunBytes	24	8 bytes	Binary	Cumulative number of bytes that were protected by this IKE tunnel.
NMsIKETunP1Rexmit	32	8 bytes	Binary	Cumulative number of retransmitted key exchange (phase 1) messages sent for this tunnel over the life of the IKE daemon. This data is cumulative even across TCP/IP restarts.
NMsIKETunP1Replay	40	8 bytes	Binary	Cumulative number of replayed key exchange (phase 1) messages received for this tunnel over the life of the IKE daemon. This data is cumulative even across TCP/IP restarts.
NMsIPIKETunP2Rexmit	48	8 bytes	Binary	Cumulative number of retransmitted QUICKMODE (phase 2) messages sent for this tunnel over the life of the IKE daemon. This data is cumulative even across TCP/IP restarts.
NMsIPIKEStatsP2Replay	56	8 bytes	Binary	Cumulative number of replayed QUICKMODE (phase 2) messages received for this tunnel over the life of the IKE daemon. This data is cumulative even across TCP/IP restarts.

- One variable-length section contains the contents of the local identity used to negotiate the IKE tunnel. Regardless of the type of the identity, the identity is expressed as an EBCDIC string. An IP address is returned in printable form. A key ID is returned as an EBCDIC string of hex values.
- 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. A key ID is returned as an EBCDIC string of hex values.

NMsec_GET_IKETUNCASCADE

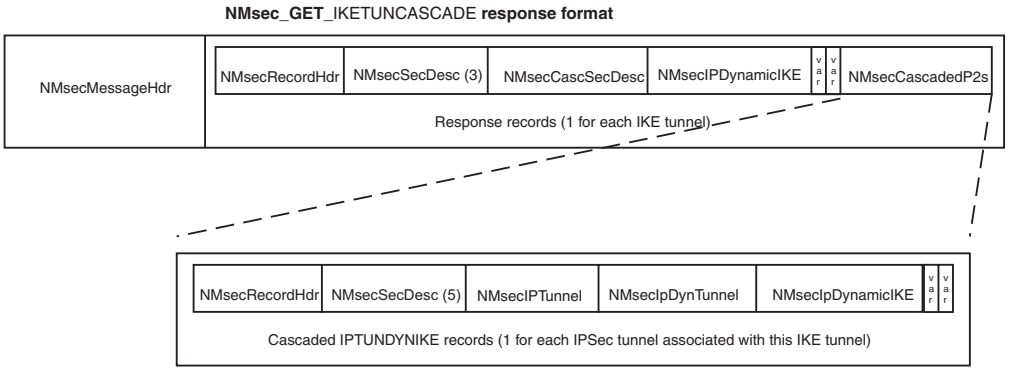


Figure 26. NMsec_GET_IKETUNCASCADE response format

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 525.
- One section, NMsecIKETunStats, describes various counters and statistics for the IKE tunnel. The layout of this section is described in Table 53 on page 530.
- 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 523.

NMsec_GET_IPINTERFACES

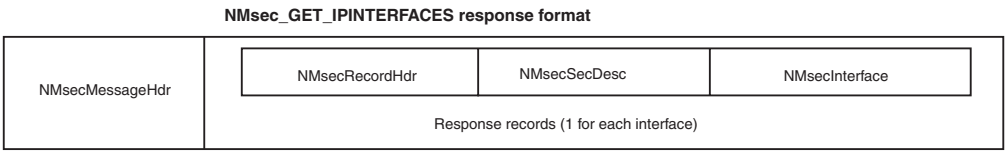


Figure 27. NMsec_GET_IPINTERFACES response format

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 54. NMsecInterface structure

Field	Offset	Length	Format	Description
NMsInterfaceName	0	16 bytes	EBCDIC	Interface name
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

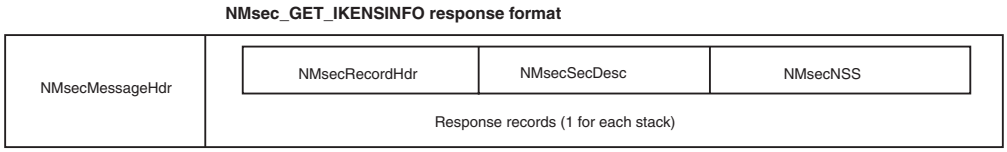


Figure 28. NMsec_GET_IKENSINFO response format

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 55.

Table 55. NMsec_GET_IKENSINFO structure

Field	Offset	Length	Format	Description
NMsIKENSClientName	0	24 bytes	EBCDIC	The stack's NSS client name. The default format is *sysname_stackname*, where *sysname* is the client's MVS system name, and *stackname* 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.
NMsIKENSServerConnectState	50	2 bytes	Binary	NS server connection state. The field can have one of the following values: **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.
NMsIKENSClientAddr	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.

Table 55. NMsec_GET_IKENSINFO structure (continued)

Field	Offset	Length	Format	Description
NMsIKENSServerAddr	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.
NMsIKENSClientPort	84	2 bytes	Binary	The TCP source port of the client connection to the server.
NMsIKENSServerPort	86	2 bytes	Binary	The TCP destination port of the client connection to the server.
NMsIKENSConnTime	88	4 bytes	Binary	UNIX-format timestamp indicating when the client connected to the server.
NMsIKENSLastMsgTime	92	4 bytes	Binary	UNIX-format timestamp indicating when the last message was sent to the server.
NMsIKENSNumFailedReqs	96	8 bytes	Binary	Number of failed requests to server.
NMsIKENSClientAPIVersion	104	1 byte	Binary	The version of the NSS client API that the NSS client is using. • NMsec_NSS_API_VERSION1 (1) - The level of NSS support that is available in z/OS version V1R9 and later. • NMsec_NSS_API_VERSION2 (2) - The level of NSS support that is available in z/OS version V1R10 and later.
NMsIKENSServerAPIVersion	105	1 byte	Binary	The version of the NSS client API that the connected NSS server supports. • The value 0 indicates that this information is not currently available. • NMsec_NSS_API_VERSION1 (1) - The level of NSS support that is available in z/OS version V1R9 and later. • NMsec_NSS_API_VERSION2 (2) - The level of NSS support that is available in z/OS version V1R10 and later.
NMsIKENSClientRsvd2	106	2 bytes	Binary	Reserved

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.

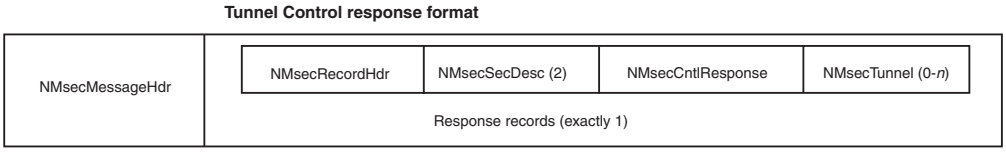


Figure 29. Tunnel control response format

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 56 on page 535. For requests that indicate a specific tunnel, this section does not exist.

Table 56. NMsecTunCntlResponse structure

Field	Offset	Length	Format	Description
NMsTunCRCount	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.

IPSec NMI 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.

IPSec NMI 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 57. 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.
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: Reissue the connection and send a correctly formatted message.

Table 57. Return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
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 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 nonzero 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 twenty input filters. System Action: Request is failed but connection remains open. Response: Send a message with the correct number of input filters.

Table 57. Return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
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.
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.

Table 57. Return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
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 *z/OS UNIX System Services Messages and Codes*.

Network security services (NSS) network management 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 IPSec clients, or monitor NSS clients that are connected to the local NSS server. 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 IPSec clients. The NSS server routes all monitoring and control requests (described in "Local IPSec NMI" on page 480) to NSS IPSec clients, with the exception of the NMsec_GET_STACKINFO and NMsec_GET_IKENSINFO requests. Routing occurs only if the NSS IPSec 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, either for a specified discipline or for all disciplines.

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 IPSec 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.

Network security services NMI: 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.

- For an NSS IPSec 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.
- For an NSS XMLAppliance client, the default value of a client name is left up to the client application's implementation.

Tip: When you override the *clientname* value for an NSS IPSec client, 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 NSS IPSec client monitoring functions (those that request information only from an NSS IPSec 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 IPSec client).
- Access to the remote NSS IPSec 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 IPSec 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.

Network security services NMI: 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 483).

Network security services NMI request and 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 480). 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.

Network security services NMI request messages

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 480). In addition, the NSS server also supports the NMsec_GET_CLIENTINFO request message.

The NMsec_GET_CLIENTINFO request is a monitoring request that obtains a list of NSS clients that are currently connected to the NSS server as well as summary information about each client. This request allows zero or more input filtering records that specify client discipline type. 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.

Network security services NMI response messages

The NSS server supports all of the response messages described for the IKE daemon in "Local IPSec NMI" on page 480 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

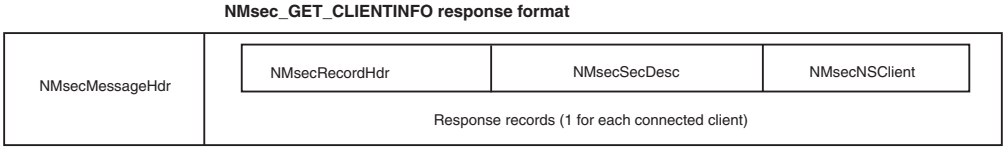


Figure 30. NMsec_GET_CLIENTINFO response format

Each record returned identifies and describes a single NSS client. Each record has a single section, NMsecNSClient, which contains the following data.

Table 58. NMsecNSClient structure

Field	Offset	Length	Format	Description
NMsNSClientName	0	24 bytes	EBCDIC	The name of the NSS client. For z/OS clients, the default format is *sysname_stackname*, where *sysname* is the z/OS system name, and *stackname* is the TCP/IP job name. z/OS clients can override this default form with any valid 24-character string. For non z/OS clients, the format of the client name is determined by the client.
NMsNSClientSysName	24	8 bytes	EBCDIC	This field contains the NSS client's system name. Each NSS client provides this value to the NSS server at connection time. For z/OS NSS clients, this field contains the client's z/OS system name. For other clients, this field can contain a system identifier such as the first 8 bytes of their system's host name.
NMsNSClientStackName	32	8 bytes	EBCDIC	This field contains the NSS client's stack name. Each NSS client provides this value to the NSS server at connection time.
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 NSS client.
NMsNSClientSvcSelCert	84, bit 0	1 bit	Binary	Used in conjunction with the NMsNSClientDiscipline field to indicate which type of certificate service, if any, is selected. If the NMsNSClientDiscipline field is set to NMsec_DISCIPLINE_IPSEC, this field indicates whether the IPSec certificate service is selected. If the NMsNSClientDiscipline field is set to NMsec_DISCIPLINE_XMLAPP, this field indicates whether the XMLAppliance certificate service is selected.
NMsNSClientSvcEnblCert	84, bit 1	1 bit	Binary	Used in conjunction with the NMsNSClientDiscipline field to indicate which type of certificate service, if any, is enabled. If the NMsNSClientDiscipline field is set to NMsec_DISCIPLINE_IPSEC, this field indicates whether the IPSec certificate service is enabled. When the NMsNSClientDiscipline field is set to NMsec_DISCIPLINE_XMLAPP, this field indicates whether the XMLAppliance certificate service is enabled.
NMsNSClientSvcSelMgmt	84, bit 2	1 bit	Binary	The IPSec Network Management Service is selected.
NMsNSClientSvcEnblMgmt	84, bit 3	1 bit	Binary	The IPSec Network Management Service is enabled.

Table 58. NMsecNSClient structure (continued)

Field	Offset	Length	Format	Description
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.
NMsNSClientSvcSelPrivKey	84, bit 5	1 bit	Binary	XMLAppliance private key service is selected.
NMsNSClientSvcEnblPrivKey	84, bit 6	1 bit	Binary	XMLAppliance private key service is enabled.
NMsNSClientSvcSelSAF	84, bit 7	1 bit	Binary	XMLAppliance SAF access service is selected.
NMsNSClientSvcEnblSAF	84, bit 8	1 bit	Binary	XMLAppliance SAF access service is enabled.
NMsNSClientRsvd1	84, bit 9	7 bits	Binary	Reserved
NMsNSClientConnectState	86	2 bytes	Binary	Client connection state. The field can have one of the following values: **NMsec_CLIENTSTATUS_CONNECTPENDING (0x0001)** 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 (0x0002)** The client is connected to the server. **NMsec_CLIENTSTATUS_DISCONNECTPENDING (0x0003)** 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 (0x0004)** 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	For NSS IPSec clients, the number of create signature requests that have been received from the client. For NSS XMLAppliance clients, this number is 0.
NMsNSClientNumreqSigVerify	104	8 bytes	Binary	For NSS IPSec clients, the number of verify signature requests that have been received from the client. For NSS XMLAppliance clients, this number is 0.
NMsNSClientNumreqCACacheRefresh	112	8 bytes	Binary	For NSS IPSec clients, the number of CA cache refreshes that have been requested by the client. For NSS XMLAppliance clients, this number is 0.

Table 58. NMsecNSClient structure (continued)

Field	Offset	Length	Format	Description
NMsNSClientNumNMIForward	120	8 bytes	Binary	For NSS IPSec clients, the number of NMI requests that have been forwarded to the client by the server. For NSS XMLAppliance clients, this number is 0.
NMsNSClientNumreqCertSvc	128	8 bytes	Binary	The number of XMLAppliance certificate service requests made by the client.
NMsNSClientNumreqPrivKeySvc	136	8 bytes	Binary	The number of XMLAppliance private key service requests made by the client.
NMsNSClientNumreqSAFSvc	144	8 bytes	Binary	The number of XMLAppliance SAF access service requests made by the client.
NMsNSClientAPIVersion	152	1 byte	Binary	The version of the NSS client API that the NSS client is using. • NMsec_NSS_API_VERSION1 (0x01) - The level of NSS support that is available in z/OS version V1R9 and later. • NMsec_NSS_API_VERSION2 (0x02) - The level of NSS support that is available in z/OS version V1R10 and later. • NMsec_NSS_API_VERSION3 (0x03) - The level of NSS support that is available in z/OS version V1R11 and later.
NMsNSClientDiscipline	153	1 byte	Binary	NSS discipline: • NMsec_DISCIPLINE_IPSEC (0x01) - Indicates an NSS IPSec client. • NMsec_DISCIPLINE_XMLAPP (0x02) - Indicates an NSS XMLAppliance client.
NMsNSClientRsvd2	154	2 bytes	Binary	Reserved

Network security services NMI 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.

Network security services NMI 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. For more information about return codes and reason codes, see Overview of diagnosing NSS server problems in *z/OS Communications Server: IP Diagnosis Guide*.

Table 59. Request return and reason codes

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
0	0	No error.

Table 59. Request return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
ENXIO (138)	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 (1112)	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 (121)	NMsRsnBadIdent (1)	Invalid message or record identifier supplied in message. System Action: Connection is closed. Response: Reissue the connection and send a correctly formatted message.
EINVAL (121)	NMsRsnBadVersion (2)	Invalid version supplied in message header. System Action: Request is failed but connection remains open. Response: Send a correctly formatted message.
EINVAL (121)	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 (121)	NMsRsnExcessiveSize (4)	Excessive message size. System Action: Connection is closed. Response: Reissue the connection and send a correctly formatted message.
EINVAL (121)	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.
EINVAL (121)	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.

Table 59. Request return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
EINVAL (121)	NMsRsnReservedNonzero (7)	Reserved data in message header, record header, or record data specifies a nonzero 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 (121)	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 (121)	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 (121)	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 (121)	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 (121)	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.

Table 59. Request return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
EINVAL (121)	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 (121)	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 (121)	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.
EACCESS (111)	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.
EACCESS (111)	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 (132)	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.
ENOMEM (132)	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.

Table 59. Request return and reason codes (continued)

Return code (NMsMRc)	Reason code (NMsMRsn)	Description
ETIMEDOUT (1127)	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 (122)	(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 *z/OS UNIX System Services Messages and Codes*.
EMVSERR (157)	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.

Packet and data trace formatting NMI

Records collected from the SYSTCPDA and SYSTCPOT interface described in "Real-time TCP/IP network monitoring NMI" on page 564 can be formatted programmatically with the EZBCTAPI macro. This section describes how the EZBCTAPI interface can be used.

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) or OSAENTA packets (SYSTCPOT), which is described in "Real-time TCP/IP network monitoring NMI" on page 564.

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	V1R7	V1R8	V1R9+
1	Packet Trace	GTCNTL	IPv4	variable	variable	N	N	N
2	X25 Trace	GTCNTL	IPv4	variable	variable	N	N	N
3	Data Trace	GTCNTL	N/A	N/A	variable	N	N	N
4	Packet Trace	PTHDR_T	IPv4	variable	variable	Y	Y	Y

CteFmtId	Description	Header	IP Header	Protocol	Data	V1R7	V1R8	V1R9+
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[1]	PTHDR_T	N/A	N/A	variable	Y	Y	Y
7	OSAENTA trace	PTHDR_T	IPv4 or IPv6	variable	variable	N	Y	Y
[1] EE stands for Enterprise Extender. Read about Enterprise Extender in the Using Enterprise Extender (EE) information in the *z/OS Communications Server: SNA Network Implementation Guide*.								

Note: Record types 1, 2, and 3 are no longer created by TCP/IP.

The ABBREV value of the PKTTRACE, DATTRACE, and OSAENTA commands determines the amount of data that is available. The layout of CTEs in the 64 KB buffer is shown in the following figure.

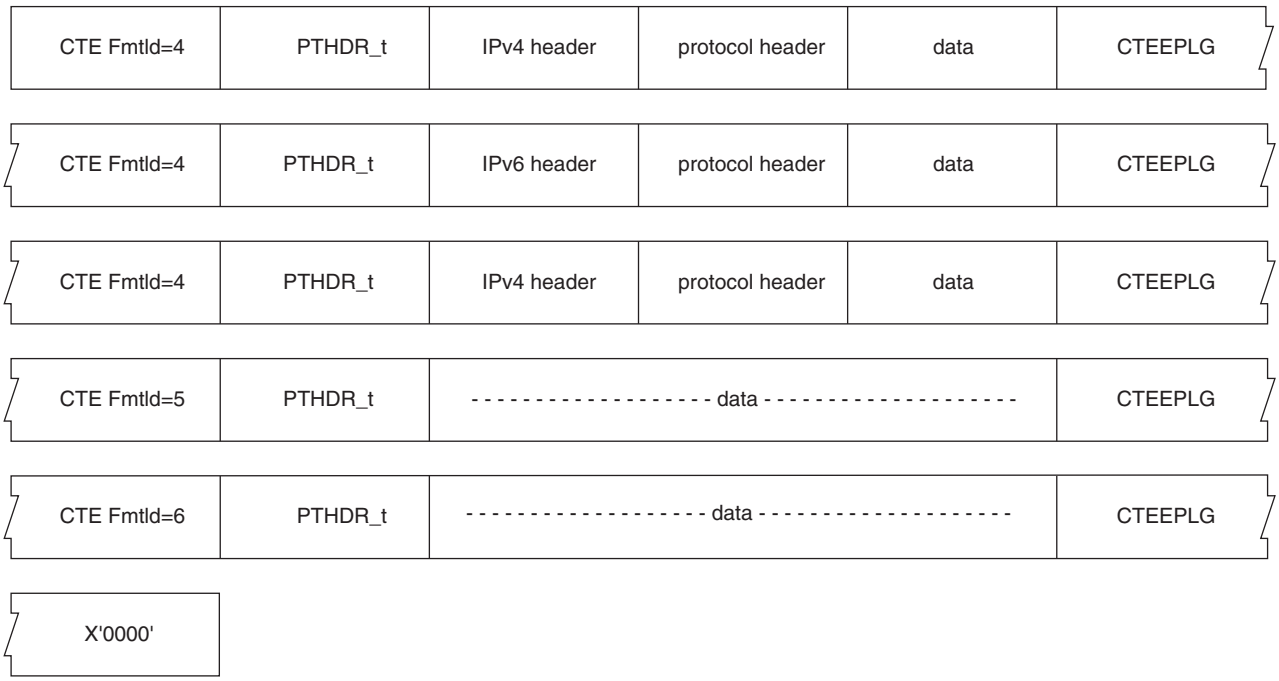


Figure 31. CTE layout

Packet and data trace formatting NMI: 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, OSAENTA, 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 5 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 included 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.

EZBCTAPI NMI 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

EZBCTAPI NMI 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).

  ,ENTRYID=entryId_list    entryId_list: Rx-type address or register (2) - (12).

  ,LDTO=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)
```

EZBCTAPI NMI 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 is 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.

,ENTRYID=entryId_list
Specifies a list of entry identifiers used to to select a subset of CTE entries. The format of the list is fullword count, followed by a list of two fullword pairs. The fullword count contains the number of fullword pairs that follow. The first word of the pair contains the low value of the entry ID and second word contains the high value of the entry ID. For example, to format only type 5 data trace records, use the following:

```
DC    F'1,5,5
```

The count is one pair of words, and the low and high values are both 5.

,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 CVTLSO is used as the default.

,MAXLINE=number

Specifies the location of a word than 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 is 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 561 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 is 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 two lines per trace record. Key fields from each qualifying trace record are 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 is 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		
ENTRYID	I	X				X		
LDTO	I	X				X		
LSO	I	X				X		
NMCTF	I	X	X	X	X			
MAXLINE	I	X				X		
OBTAIN	I	X				X		
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

EZBCTAPI NMI 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).

EZBCTAPI NMI 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.

EZBCTAPI NMI ABEND codes

There are no ABEND codes.

EZBCTAPI NMI 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 60. GPR 0 (and rsncode, if you coded RSNCODE) might contain one of the reason codes shown in Table 60.

Table 60. 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.

Table 60. EZBCTAPI return and reason codes (continued)

Hexadecimal return code (CtApi_IRtnCd)	Hexadecimal reason code (CtApi_IRsnCd)	Meaning
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	*xxxxxxxx*	The interface routine or the analysis routine abended; *xxxxxxxx* is the abend code.
Note: The first line of a new return code is a generic line about the return code.		

Table 61. 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').

EZBCTAPI NMI: 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. See "Real-time NMI: Configuration and enablement" on page 566 for information about authorizing an application.

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 564).
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.

EZBCTAPI NMI: 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
*
* INTIALIZE THE EZBCTAPI PARAMETER LIST
         EZBCTAPI WORKAREA=(8),                                        C
               COMP==CL8'SYSTCPDA',                                    C
               PRTSRV==A(PRINTSRV),                                    C
               OPTIONS=APTO,                                           C
               REPORT=FULL,                                            C
               TIME=LOCAL,                                             C
               USERTOKEN=PRINTTKN,                                     C
               MAXLINE==A(L'PRINTBUF-1),                               C
*              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,                                        C
               ABUFFER,PRIMARYALET,LBUFTKN,                            C
               RETVAL,RETCDE,RETRSN),VL
         L     15,RETVAL
         LTR   15,15
         BNP   EOF                     CLOSE SOCKET AND EXIT
*
* READ IN DATA BUFFERS
*
         ST    15,LBUFTKN
         CALL  EZBTMIC1,(BUFTOKEN,LBUFTKN,RETVAL,RETCDE,RETRSN)
         L     15,RETVAL
         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,                                      C
               MF=(E,CTAPIL)
         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)
*                                      RELEASE STORAGE ABDPL WORK AREA
         CLOSE (PRINTDCB)
         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
RETVAL   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,                      C
               RECFM=FBA,LRECL=133
*
PRINTBUF DS    0CL133                  A PRINT BUFFER
PRINTCC  DC    C' '
PRINTDAT DC    CL132' '
*
PRINTSA  DC    18A(0)                  A SAVE AREA FOR PRINT SERVICE
*
*
*
         EJECT
PRINTSRV 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 62 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 and OSAENTA trace (SYSTCPOT) information in *z/OS Communications Server: IP Diagnosis Guide*.

Table 62. Available EZBYPTO options

Option	Field setting	Field format
ASCII	Pto_Dump=1;Pto_DmpCd=PtoAscii;	Bit flag
BASIC(DETAIL)	Pto_Basic=1;Pto_BasDtl=1;	Bit flag
BASIC(SUMMARY)	Pto_Basic=1;Pto_BasDtl=0;	Bit flag
BOTH	Pto_Dump=1;Pto_DmpCd=PtoBoth;	Bit flag
CLEANUP(*nnnnn*)	Pto_Cleanup=1;Pto_GcIntvl=*nnnnn*;	Bit flag
DEVICE(list)	Pto_Device@=Addr(list);Pto_Device#=*nn*	List of 32-bit word pairs
DISCARD(list)	Pto_Discard@=Addr(list),Pto_Discard#=*nn*	List of 16-bit word pairs
DUMP	Pto_Dump=1;	Bit flag
DUMP(*nnnnn*)	Pto_Dump=1;Pto_MaxDmp=*nnnnn*;	Bit Flag; 31-bit word

Table 62. Available EZBYPTO options (continued)

Option	Field setting	Field format
EBCDIC	Pto_Dump=1;Pto_DmpCd=PtoEbcdic;	Bit flag; Value
ELEMENT(list)	Pto_Element@=Addr(list);Pto_Element=*nn*	List of 32-bit word pairs
ETHTYPE(list)	Pto_EthType@=Addr(list);Pto_EthType#=*nn*	List of 32-bit word pairs
FLAGS(flags)	Pto_Flags@=Addr(Pto_Flagss), Pto_Flags#=size(Pto_Flagss);	16 bytes of bit flag used to select packets
FLAGS(ANY|ALL)	Pto_FlgAny=1	Select a packet that has any flags in Pto_Flagss set.
FORMAT(DETAIL)	Pto_Format=1;Pto_FmtDtl=1;	Bit flags
FORMAT(SUMMARY)	Pto_Format=1;Pto_FmtDtl=0;	Bit flags
FULL	Pto_Dump=1,Pto_Format=1,Pro_FmtDtl=1;	Bit flags
HEX	Pto_Dump=1;Pto_DmpCd=PtoHex;	Bit flags
HPRDIAG	Pto_HprSess=1;Pto_HprRpt=Pto_HprSummary;	Bit flags
INTERFACE	Pto_Links@=Addr(list),Pto_Links#=*nn*	List of 16-byte character strings
IPADDR(list)	Pto_Addr@=Addr(list);Pto_Addr#=*nn*	List of 16-byte byte IPv6 addresses
MACADDR(list)	Pto_MacAddr@=Addr(list);Pto_MacAddr#=*nn*	List of 6-byte Mac addresses
PORT(list)	Pto_Port@=Addr(list);Pto_Port#=*nn*	List of 16-bit word pairs
PROTOCOL(list)	Pto_Proto@=Addr(list);Pto_Proto#=*nn*	List of 32-bit word pairs
REASSEMBLY(*nnnnn*)	Pto_ReAsm=1;Pto_MaxRsm=*nnnnn*	Bit flags
REASSEMBLY(DETAIL)	Pto_ReAsm=1;Pto_RsmSum=0	Bit flags
REASSEMBLY(SUMMARY)	Pto_ReAsm=1;Pto_RsmSum=1	Bit flags
NOREASSEMBLY	Pto_ReAsm=0;	Bit flag
SEGMENT	Pto_Segment=1;	Bit flag
NOSEGMENT	Pto_Segment=0;	Bit flag
SESSION(DETAIL)	Pto_SesRpt=Pto_SesDetail; Pto_Session=1;	Bit flag
SESSION(SUMMARY)	Pto_SesRpt=Pto_SesSummary; Pto_Session=1;	Bit flag
SESSION(STATE)	Pto_SesRpt=Pto_SesState; Pto_Session=1;	Bit flag
SPEED(local,remote)	Pto_LSpeed=*nnn*;Pto_RSpeed=*nnn*	Two 32-bit words
STATISTICS(DETAIL)	Pto_Stats=1;Pto_StcSum=0;	Bit flag
STATISTICS(SUMMARY)	Pto_Stats=1;Pto_StcSum=1;	Bit flag
STREAMS(*nnn*)	Pto_Streams=1;Pto_StrmBuf=*nnn*	Bit flag
STREAMS(DETAIL)	Pto_Streams=1;Pto_StmSum=0;	Bit flag
STREAMS(SUMMARY)	Pto_Streams=1;Pto_StmSum=1;	Bit flag
SUBAREA(list)	Pto_Subarea@=Addr(list);Pto_SubArea#=*nn*	List of 32-bit word pairs
SUMMARY	Pto_Summary=1;	Bit flag
TCID(list)	Pto_Tcid@=Addr(list);Pto_Tcid#=*nn*	List of 8-byte hex strings
TH5SA(list)	Pto_Th5SA@=Addr(list);Pto_Th5SA#=*nn*	List of 8-byte hex strings
TALLY	Pto_Stats=1;Pto_StcSum=0;	Bit flag
VLANID(list)	Pto_VlanId@=Addr(list);Pto_VlanId#=*nn*	List of 32-bit word pairs

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 0, 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.
9. If the first and only discard reason code in the Pto_Discard1(1) field is 65 535 (X'FFFF'), then all packets with a nonzero discard reason code are selected. If one of the discard reason codes is 0, then packets that were not discarded are selected.

Using the packet trace 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) is 0. The record contains the IP header and protocol header (if any). Otherwise the record contains only 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 0. 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.

Real-time TCP/IP network monitoring NMI

Network management applications can use the z/OS Communications Server real-time TCP/IP network monitoring NMI to programmatically obtain data in real time. The network management applications obtain the data by performing the following steps:

- Connect to one of the real-time NMI interfaces. Use the NETMONITOR profile statement to enable these interfaces in the TCP/IP stack.
- Invoke the TMI copy buffer interface to copy the real-time data to application storage.

Table 63 shows the real-time NMI interfaces that are described in this topic.

Table 63. Real-time NMI interfaces

Interface name	Description
SYSTCPDA	Real-time TCP/IP packet and data trace NMI
SYSTCPCN	Real-time TCP connection SMF NMI
SYSTCPOT	Real-time OSAENTA packet trace NMI
SYSTCPSM	Real-time SMF NMI

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 64. Interface descriptions

Interface	Description
Real-time TCP/IP packet and data trace 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 *z/OS Communications Server: IP System Administrator's Commands* for more information about using the **Vary** command.
Real-time TCP connection SMF NMI (SYSTCPCN)	Using this interface, applications can be notified when TCP connections are established or terminated in a near real-time fashion. SYSTCPCN 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 64. Interface descriptions (continued)

Interface	Description
Real-time TCP/IP OSAENTA trace NMI (SYSTCPOT)	Using this interface, applications can obtain copies of network packets and records that are buffered by the TCP/IP OSAENTA trace functions. The OSAENTA Trace function must be enabled using the VARY TCPIP,,OSAENTA command. See *z/OS Communications Server: IP System Administrator's Commands* for more information about using the **Vary** command.
Real-time SMF NMI (SYSTCPSM)	The records provided through the interface are type 119 SMF records. The specific subtypes that are provided are: • FTP client transfer completion records (subtype 3) • TCP/IP profile event record (subtype 4) • TN3270E Telnet server session initiation and termination records (subtypes 20 and 21) • TSO Telnet client connection initiation and termination records (subtypes 22 and 23) • DVIPA status change and DVIPA removed records (subtypes 32 and 33) • DVIPA target added and removed records (subtypes 34 and 35) • DVIPA target server started and ended records (subtypes 36 and 37) • CSSMTP event records (subtypes 48 - 52) • FTP server transfer completion records (subtype 70) • FTP server logon failure records (subtype 72) • IKE tunnel and dynamic tunnel event records (subtypes 73 - 78) • Manual tunnel activation and deactivation records (subtypes 79 and 80) 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 NMI: FTP SMF type 119 subtypes 100-104 record formats" on page 583 for the structures and mappings of records 100 through 104.

Steps for using the real-time NMI

Follow this 2-step process to use interfaces of the real-time NMI to access the data:

1. Connect to the interface. See "Connecting to the AF_UNIX stream socket."
2. Copy the real-time data to application storage. See "Obtaining the real-time data."

Connecting to the AF_UNIX stream socket

The Communications Server TCP/IP stack provides an AF_UNIX stream socket for each of the interfaces (see Table 64 on page 564). The interfaces allow one or more applications to receive notifications for the data that is being collected. The TCP/IP stack acts as the server for these AF_UNIX stream sockets, performing the listen() function call and waiting for incoming connection requests. To use the interface, applications connect to the listening socket. Each interface has a distinct AF_UNIX path name that uniquely identifies the socket that the interface will use. The network management application can connect to one or more interfaces from the same application.

- If the application connects to the SYSTCPDA, SYSTCPOT, or SYSTCPSM interface, it immediately starts receiving applicable data.
- If the application connects to the SYSTCPCN interface, it must send a record to the server to indicate the type of data it requires. Then the application starts receiving applicable data.

Obtaining the real-time data

Each notification record that the application receives over the socket represents a buffer that the TCP/IP stack stores. The actual real-time data is not part of this notification record. After the application receives the entire notification record from the AF_UNIX socket, it must pass this record, along with a user-allocated storage buffer, to the TMI copy buffer interface. For more information about the TMI copy buffer interface, see "Real-time NMI: Copying the real-time data" on page 572.

The TMI copy buffer interface populates the provided storage buffer with the output records. The output records are related to the real-time interface that the input notification record defines. After the application receives the notification over the AF_UNIX socket, it must call the TMI copy buffer interface immediately because the TCP/IP stack stores the buffers in a circular queue and the buffers might be eventually 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 quickly enough to obtain the data before those buffers are overwritten.

For information about the format of these buffers and the records that the buffers contain, see "Real-time NMI: Processing the output records" on page 577.

Real-time NMI: Configuration and enablement

You must enable the real-time interfaces within the TCP/IP stack and authorize the network management applications to the interfaces before the applications can obtain the real-time data.

Perform the following steps to enable the real-time interfaces and authorize the network management applications:

1. Enable the real-time interfaces.

Use the NETMONITOR statement in the TCP/IP profile to enable the real-time interfaces and start the collection of the real-time data. See NETMONITOR statement in *z/OS Communications Server: IP Configuration Reference* for details.

2. Authorize the network management applications. See "Authorizing the applications."

Authorizing the applications

Perform the following steps to authorize applications to use the real-time NMI:

1. Define the security product resource profiles

   An optional resource name is supported for each real-time interface to restrict access to the interfaces. The resource name has the format EZB.NETMGMT.*sysname*.*tcpprocname*.*interface*, where:

   - *sysname* is the MVS system name where the interface is enabled.
   - *tcpprocname* is the job name that is associated with the TCP/IP stack where the interface is enabled.
   - *interface* is the real-time interface name. It can be SYSTCPDA, SYSTCPCN, SYSTCPOT, or SYSTCPSM.

   For examples of the RACF commands that are used to define the real-time interface security product resource names, see sample EZARACF in data set SEZAINST.

2. Permit the user IDs of the applications to access the real-time NMI resources

   After the resource profiles are defined, the user ID that is associated with the network management application must be permitted for READ access to the resources.

   Guideline: The user ID that is referenced for access to the resources is the user ID that is associated with the MVS address space from which the connect() function call or the TMI copy buffer interface invocation was issued. 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 should define the real-time interface resource profiles for the real-time interfaces and permit the user ID of the application for READ access to the profile.

3. Review the authorization verification performed by the real-time NMI

   The authorization verification for the application is different when an application connects to the real-time interface and when it invokes the TMI copy buffer interface. You should review the verification to ensure that your application will be authorized. See "Verifying authorization for applications that connect to the real-time interface" and "Verifying authorization for applications that invoke the TMI copy buffer interface" on page 568 for more information.

Verifying authorization for applications that connect to the real-time interface: The real-time NMI performs the following actions to verify the authorization of an application when the application connects to the real-time interface:

- If a multilevel secure (MLS) environment is active, the real-time interface resource profile must be defined and the user ID that is associated with the application must be permitted for READ access to the profile.
- If the real-time interface resource profile is defined, the user ID that is associated with the application must be permitted for READ access to the profile.
- If the real-time interface resource profile is not defined, the user ID of the application must be defined as a superuser (that is, a user ID with an OMVS

UID of zero or a user ID that is permitted for READ access to the BPX.SUPERUSER resource in the FACILITY class).

Verifying authorization for applications that invoke the TMI copy buffer interface: The real-time NMI performs the following actions to verify the authorization of an application when the application invokes the TMI copy buffer interface:

- The application is APF authorized.
- If the application is not APF authorized, the real-time interface resource profile must be defined and the user ID that is associated with the application must be permitted for READ access to the profile.

Guideline: Because the security product resource profiles can be used for authorization verification when connecting to the interface and when invoking the TMI copy buffer interface, you should use the profiles for authorizing network management applications to use the real-time interfaces.

Real-time NMI: Connecting to the server

The application that is to use 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, *tcpipprocname* is the procedure name used to start TCP/IP.

- Network monitor interface for capturing packet and data trace packets (SYSTCPDA)

  `/var/sock/SYSTCPDA.tcpipprocname`
- Network monitor interface for obtaining TCP connection information (SYSTCPCN)

  `/var/sock/SYSTCPCN.tcpipprocname`
- Network monitor interface for capturing OSAENTA trace packets (SYSTCPOT)

  `/var/sock/SYSTCPOT.tcpipprocname`
- Network monitor interface for obtaining real-time SMF data (SYSTCPSM)

  `/var/sock/SYSTCPSM.tcpipprocname`

Use either the z/OS XL C/C++ API or the z/OS UNIX System Services assembler callable services to open AF_UNIX sockets and connect to the given service.

Real-time NMI: Interacting with the servers

In the case of the TCP connection information service, after connecting to the SYSTCPCN server over AF_UNIX socket, /var/sock/SYSTCPN.*tcpipprocname*, the application must then send a connection request record to the server over the connected socket (see the tmi_conn_request record in "Real-time NMI: Requests sent by the client to the server" on page 569). For the other services, the application does not need to take action.

After the client connects to the desired server (or, in the case of the SYSTCPCN 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 "Records sent by the server to the client: Initialization record" on page 570). 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 "Records sent by the server to the client: Termination record" on page 571) 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, SYSTCPOT, and SYSTCPCN interfaces, large amounts of data can be generated. For SYSTCPDA, do not 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 *z/OS Communications Server: IP System Administrator's Commands* and the packet trace (SYSTCPDA) for TCP/IP stacks information in *z/OS Communications Server: IP Diagnosis Guide* for details. For SYSTCPCN, 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 *z/OS Communications Server: IP Configuration Reference* for details. For SYSTCPOT, see the VARY TCPIP,,OSAENTA information in *z/OS Communications Server: IP System Administrator's Commands*.

Restriction: Except in the case of sending a connection request record for the SYSTCPCN 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.

Real-time NMI: 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 */
};
```

Real-time NMI: Requests sent by the client to the server

For the SYSTCPCN 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 SYSTCPCN 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 SYSTCPCN 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 "Records sent by the server to the client: Termination record" on page 571) with *tmit_termcode* = EINVAL to the client, and closes the connection.

Real-time NMI: 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.

Records sent by the server to the client: Initialization record

After the client connects to the server, the server sends an initialization record to the client. The initialization record can 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 ezbytmih.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_sysn[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 that the client is connected to. This is one of SYSTCPDA, SYSTCPCN, SYSTCPOT, or SYSTCPSM, depending on the server that is 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.

Records sent by the server to the client: Termination record

The termination record is sent when the server closes the connection. The termination record can 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.

Value	Description
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 is closed by the server.

Records sent by the server to the client: Token record

The server sends the tmi_token record when a buffer has been filled with records for the given service. The token record can be recognized as having a *TmiHr_Id* value that is equal to the *TmiHr_PktTok* value (in the case of SYSTCPDA and SYSTCPOT) or the *TmiHr_SmfTok* value (in the case of SYSTCPCN 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 "Real-time NMI: Copying the real-time data."

Real-time NMI: Copying the real-time data

To copy the data buffer to application storage, use one of the following TMI copy buffer interfaces, depending on the language that is used to write your application:

Assember interface
EZBTMIC1 (31-bit AMODE) or EZBTMIC4 (64-bit AMODE)

C/C++ interface
TMI_CopyBuffer function call (invoking EZBTMIC1 or EZBTMIC4)

EZBTMIC1 or EZBTMIC4: Copy real-time data for assembler applications

The EZBTMIC1 and EZBTMIC4 callable interfaces use the *tmi_token* record that was recently read from the AF_UNIX socket as input to locate and copy the data buffer into application storage. Assembler macro EZBYTMIA contains the definitions of the calls to these interfaces. The EZBYTMIA macro is installed in the SEZANMAC data set.

Guideline: EZBTMIC1 is the API that is used by AMODE 31 callers, EZBTMIC4 is the API that is used by AMODE 64 callers. References to the EZBTMIC1 API also apply to the EZBTMIC4 API.

EZBTMIC1 requirements:

Authorization:	Supervisor state or problem state, any PSW key; Caller must be APF authorized or the security product profile for the real-time interface must be defined and the user ID of the application must be permitted for READ access to the profile.
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 must be in the primary address space.

EZBTMIC4 requirements:

Authorization:	Supervisor state or problem state, any PSW key; Caller must be APF authorized or the security product profile for the real-time interface must be defined and the user ID of the application must be permitted for READ access to the profile.
Dispatchable unit mode:	Task
Cross memory mode:	PASN = HASN
AMODE:	64-bit
ASC mode:	Primary mode
Interrupt status:	Enabled for interrupts
Locks:	Unlocked
Control parameters:	All parameters must be addressable by the caller and must be in the primary address space.

EZBTMIC1 format:

```
CALL EZBTMIC1,(Token,
               Bufptr,
               Return_value,
               Return_code,
               Reason_code)
```

EZBTMIC4 format:

```
CALL EZBTMIC1,(Token,
               Bufptr,
               Return_value,
               Return_code,
               Reason_code)
```

EZBTMIC1 or EZBTMIC4 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 is 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 0. 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-bit 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* token, the data buffer contains the unformatted packet trace data records (SYSTCPDA or SYSTCPOT).
- When the token is a *TmiHr_SmfTok* token, 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.
–1	EACCES	The application is not 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*.

EZBTMIC1 or EZBTMIC4 usage notes:

- Compiling and linking
  - Assembler mappings for the various records that flow over the AF_UNIX socket are in macro EZBYTMIA.
  - EZBTMIC1 and EZBTMIC4 are defined as callable stubs in SYS1.CSSLIB.

TMI_CopyBuffer: Copy real-time data for C/C++ applications

The TMI_CopyBuffer() function call uses the *tmi_token* record that is recently read from the AF_UNIX socket as input to locate and copy the data buffer to the buffer that the application provides. Specify this *tmi_token* for the input *token*. The data will be copied to the buffer that *bufptr* points to. The ezbytmih.h header file is installed in the SEZANMAC data set and in the z/OS UNIX file system directory, /usr/include.

For programming requirements for invoking this function, see "EZBTMIC1 requirements" on page 573 if your application uses 31-bit AMODE and see "EZBTMIC4 requirements" on page 573 if your application uses 64-bit AMODE.

TMI_CopyBuffer format:

```
void Tmi_CopyBuffer     (struct tmi_header *token,
                         struct bufptr_t *bufptr,
                         int *retval,
                         int *retcode,
                         int *rsncode);
```

TMI_CopyBuffer 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 is overwritten with the contents of the TMI data buffer if the call is successful.

The *tmi_bufptr* structure is a 12-byte structure that describes the address of the buffer for AMODE 31 callers.

```
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 */
};
```

When _LP64 is defined, the *tmi_bufptr* structure is a 12-byte structure that describes the address of the buffer for AMODE 64 callers.

```
struct tmi_bufptr       /* Buffer pointer */
{
int buf_alet;           /* Buffer ALET, or 0 */
void *buf_addr;         /* Pointer to 64bit bufptr */
};
```

retval The returned value. If successful, TMI_CopyBuffer() returns the number of data bytes copied in *retval*. This length does not include the trailing halfword of zeros copied to the buffer. 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 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 address space of the caller.
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*.

TMI_CopyBuffer 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.

Real-time NMI: Processing the output records

Upon successful completion of the EZBTMIC1, EZBTMIC4, or TMI_CopyBuffer() invocations, the user-supplied buffer is filled with *cte* records. The *cte* record contains the data that is provided by the service being used. Each *cte* record consists of a common portion and a service-specific portion. The common portion of the record encapsulates the service-specific portion. The format of each record is as follows:

```
cte (common portion header)
service-specific portion
cteeplg (common portion epilog)
```

Format of common portion of output records

The common portion of the data record consists of the *cte* and the *cteeplg* structures. 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, which signifies the end of the data in the buffer. The layout of the buffer is as follows:

cte	data	*cte_epilogue*	*cte*	data	*cte_epilogue*	...	*cte*	data	*cte_epilogue*	binary 0

The *cte* is a 16-byte descriptor whose format is as follows (as defined in ezbytmih.h, and in ITTCTE in SYS1.MACLIB):

```
struct cte
{
    unsigned      short       ctelenp;      /* Length of CTE
                                               and cte_epilogue. */
                  short       cteoff;       /* Offset from start of CTE */
     uint32_t                 ctefmtid;     /* Format ID of record */
     uint64_t                 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 portion of output records" on page 578. 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 ezbytmih.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 portion of output records

The following information describes how to process *cte* records for all the real-time services.

Processing the *cte* records for SYSTCPDA and SYSTCPOT: Use the packet and data trace formatting NMI to format records that are collected from the SYSTCPDA and SYSTCPOT real-time NMI interfaces. See "Packet and data trace formatting NMI" on page 547 for more information about using this formatting NMI. The following *ctefmtid* values are supported for the SYSTCPDA and SYSTCPOT interfaces:

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
6 (PTHIdEE)	Described by the PTHDR_T structure	IPv4 or IPv6 EE trace record	VARY TCPIP,,PKTTRACE, SRCP =12000
7 (PTHidNTA)	Described by the PTHDR_T structure	IPv4 and IPv6 OSAENTA trace record	VARY TCPIP,,OSAENTA

Notes:

1. As of the V1R6 release, type 1, 2, and 3 records are no longer written.
2. In V1R11 and later releases, the GtCntl description is not included in EZBCTHDR.

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 PTHDR_T is defined in EZBYPTHA and contains the following information:

```
pth_len           Length of the PTHDR_T structure. This is also the offset to the
                  IP header.
pth_seqnum        Sequence number of this packet trace record
pth_flag          Flag indicators
       PTH_Local     0x80       Src and Dest are local
       PTH_CfTxt     0x40       Confidential was not recorded in data trace record
       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
pth_devty         The type of device represented by the interface being traced.
       PTHLCSE          1       - Ethernet
       PTHLCS8          2       - 802.3 Ethernet
       PTHLCSE8         3       - Ether|802.3
```

```
        PTHLCSTR        4       - Token Ring
        PTHLCSFD        5       - FDDI
        PTHLU62         6       - SNA LU6.2
        PTHHCH          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
|       PTHIQDX          44      - IQDX
        PTH6loopb       51      - IPv6 loopback
        PTH6vipa        52      ifp6vipa
        PTH6ipaqenet    53      ifp6ipaqenet
        PTH6ipaqtr      54      ifp6ipaqtr
        PTH6mpc         55      ifp6mpc
        PTH6ipaqidio    56      ifp6ipaqidio
|       PTHIQDX6         57      IQDX6
pth_tlen          Portion of the payload that is actually traced. If ABBREV was
                  not specified on the trace command then this will be the name
                  as pth_plen. If ABBREV was specified, then it will be this value.
pth_infname       Name of the interface the packet was traced on in EBCDIC
                  character format
pth_jobname       The jobname from the data trace record
pth_DtDomain      Socket domain (AF_INET or AF_INET6)
pth_DtType        Socket type (Sock#_Stream, Sock#_Dgram, Sock#_Raw)
pth_DtProto       Socket protocol number
pth_DtState       Start/End of data flow
PTH_DtStartInb  EQU   6       Data Flow starts for Inbound
PTH_DtStartOutb EQU   7       Data Flow starts for Outbound
PTH_DtTerm      EQU   8       Data Flow ends
pth_time          Stored time of day clock when packet was processed by the trace
pth_src           Hexadecimal source IP address of this packet (IPv6 or IPv4)
pth_dst           Hexadecimal destination IP address of this packet (IPv6 or IPv4)
pth_sport         Hexadecimal source IP port number
pth_dport         Hexadecimal destination IP port
pth_trcnt         Total count of records traced
|pth_IQDX          Packet associated with an IQDX device
pth_tcb           Task control block address of the sender of the outbound. On
                  inbound, this will usually be task associated with the TCP/IP
                  stack
pth_asid          Ascbasid of the sender of the outbound packet. On inbound, this
                  will usually be the asid of the TCP/IP stack
PTH_SeqNr         OSAENTA trace sequence number
Pth_Vlan          OSAENTA Vlan id field
Pth_VlanPri       Vlan priority (0-7)
Pth_VlanId        Vlan Id (0-4095)
PTH_NtaFlag       OSAENTA Flags
      Pth_OutB    1-Outbound, 0-Inbound
      Pth_Lpar    1=Lpar_to_Lpar
      Pth_DD      Data Device is valid
      Pth_gVlan   VLAN frame
      PTH_LS      Large Send
      Pth_QHdr    Qdio header present
      Pth_Exhdr   1=Extension header
      Pth_Layer2  1=Layer2 0=Layer3
pth_lost          Total lost record count
pth_plen          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_InfIx           Interface index (PKTT)
PTH_Cid             Communication Id(PKTD)
PTH_DevId           Device Id       (PKTL)
PTH_DropRsn         Packet discard reason code.  pth_DropRsn can be compared
                    with the discard reason code that is provided by EZBYCODE
                    in sys1.sezanmac. A packet can be traced twice, once at
                    the lower level IP layer, and again as a discarded packet in
                    an upper level protocol layer of TCPIP. This value is 0 if the
                    packet was not discarded.
PTH_OffSegLen       Length of each of the first N-1 segments being offloaded, not
                    including headers - i.e. the MSS (meaningful only when
                    the PTH_Seg_Offload flag is on)
PTH_NxtHopAddr      Hexadecimal next hop IP address for outbound packets (IPv6 or
                    IPv4)
PTH_NxtHopLen       Length of next hop address field
PTH_NxtHopKey       Key of next hop address field
PTH_Ext3Len         Length of the extension header
PTH_Ext3Key         Extension header key
PTH_Ext3QID         OSA Express QID - The identifier of the input queue on which
                    this packet was received. The QID value is 1 when this packet
                    was received on the primary input queue.  The QID value is
                    greater than 1 when this packet was received on an ancillary
                    input queue using QDIO inbound workload queueing. For an
                    ancillary input queue the queue type is indicated in the
                    PTH_Ext3QueueType field.
PTH_Ext3QueueType   QDIO Inbound Workload Queueing Ancillary Queue Type - The queue
                    type when the QID is an ancillary input queue. Queue types are
                    represented by the following values:
       PTHMIQBULKDATA   2    Bulk Data
       PTHMIQSYSDIST    3    Sysplex Distributor
       PTHMIQEE         4    Enterprise Extender
```

IPv4 addresses in *pth_src* and *pth_dst* are prefixed with x'000000000000', x'00000000FFFE' or x'00000000FFFF'.

Processing SYSTCPDA and SYSTCPOT trace records in a buffer: The EZBTMIC1 call or the TMI_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* is 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 *ctelenp*=1 KB	PTHDR_T structure *pth_seqnum*=0 *pth_tlen*=64 KB *pth_time*=Time X	trace data (IP packet X) contains IP headers	*ctelente*=1 KB

Second TMI_CopyBuffer issued; a complete buffer received:

Trace Records			
Record1 for IP packet Y CTE structure *ctelenp*=1 KB	PTHDR_T structure *pth_seqnum*=0 *pth_tlen*=ip payload length (less than 1 KB)	trace data (IP packet Y) contains IP headers	*ctelente*=1 KB
Record2 for IP packet X CTE structure *ctelenp=nnn*	PTHDR_T structure *pth_seqnum*=1 *pth_tlen*=64 KB *pth_time*=Time X	trace data (IP packet X continued) no headers	*ctelente=nnn*

Last TMI_CopyBuffer() issued; a partial buffer received:

Trace Records			
Record *n* for IP packet X CTE structure *ctelenp=nnn*	PTHDR_T structure *pth_seqnum*=n-1 *pth_tlen*=64 KB *pth_time*=Time X	trace data (IP packet X continued) No headers	*ctelente=nnn*

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 is a complete type 119 SMF record, of subtype 1 or 2.

Additionally, if requested, the server fills 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 are indistinguishable from newly established connections (except that the connection establishment timestamp is 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 is always 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 preexisting 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 completion (subtype 3)
- TCP/IP profile event (subtype 4)
- 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)
- DVIPA removed (subtype 33)
- DVIPA status change (subtype 32)
- DVIPA target added (subtype 34)
- DVIPA target removed (subtype 35)
- DVIPA target server ended (subtype 37)
- DVIPA target server started (subtype 36)
- CSSMTP configuration record (subtype 48)
- CSSMTP connection record (subtype 49)
- CSSMTP mail message record (subtype 50)
- CSSMTP spool file record (subtype 51)
- CSSMTP statistical record (subtype 52)
- FTP server transfer completion (subtype 70)
- FTP server logon failure (subtype 72)
- IPSec IKE Tunnel Activation/Refresh (subtype 73)
- IPSec IKE Tunnel Deactivation/Expire (subtype 74)
- IPSec Dynamic Tunnel Activation/Refresh (subtype 75)
- IPSec Dynamic Tunnel Deactivation (subtype 76)
- IPSec Dynamic Tunnel Added (subtype 77)
- IPSec Dynamic Tunnel Removed (subtype 78)
- IPSec Manual Tunnel Activation (subtype 79)
- IPSec Manual Tunnel Deactivation (subtype 80)
- FTP server transfer initialization (subtype 100)
- FTP client transfer initialization (subtype 101)
- FTP client login failure records (subtype 102)
- FTP client session records (subtype 103)
- FTP server session records (subtype 104)

For the real-time SMF data server, the *ctefmtid* for the CTE is always equal to the subtype of the SMF record (one of the values listed above) following the CTE in the data buffer. Table 65 lists the structures and macros for mapping the SMF 119 record subtypes that are delivered by these interfaces.

Table 65. SMF 119 record subtypes

Subtype	C/C++	Assembler macro
3, 20, 21, 22, 23, 32, 33, 34, 35, 36, 37, 48, 49, 50, 51,52, 70, 72, 73, 74, 75, 76, 77, 78, 79, 80	SEZANMAC(EZASMF) /usr/include/ezasmf.h	SYS1.MACLIB(EZASMF77)
4	SEZANMAC(EZBNMMPC) /usr/include/ezbnmmpc.h	SEZANMAC(EZBNMMPA)
100, 101, 102, 103, 104	SEZANMAC(EZANMFTC)	SEZANMAC(EZANMFTA)

See Appendix E, "Type 119 SMF records," on page 743 for the formats of SMF type 119 records.

Restriction: The FTP type 119 records of subtypes 100 through 104 are available only across the real-time SMF NMI interface, SYSTCPSM, and are not available in the MVS SMF data sets. Therefore, the record formats for these subtypes are not included in Appendix E, "Type 119 SMF records," on page 743. For information about processing these SMF records, see "Real-time SMF NMI: FTP SMF type 119 subtypes 100-104 record formats."

Real-time SMF NMI: FTP SMF type 119 subtypes 100-104 record formats

The FTP SMF type 119 records of subtypes 100 through 104 are available only across the real-time SMF NMI interface, SYSTCPSM, and are not available in the MVS SMF data sets. However, these SMF records have the same format as all other SMF 119 records do. To understand this format, see "Common Type 119 SMF record format" on page 745. To view the format of the TCP/IP Common identification section in the record, see "Common TCP/IP identification section" on page 748. To understand the format of the self-defining sections and record-specific data in these SMF records, use the information in this section.

Real-time SMF NMI: FTP server transfer initialization record (subtype 100)

Table 66 shows the FTP server transfer initialization self-defining section of SMF record.

Table 66. 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

Table 66. FTP server transfer initialization 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 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 “Common TCP/IP identification section” on page 748 for the contents of the TCP/IP stack identification section.				

Table 67 on page 585 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 67. 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 H, "Related protocol specifications," on page 991 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
93	SMF119FT_FSIMode	1	Transmission mode • B: Block • C: Compressed • S: Stream

Table 67. FTP server transfer initialization record section (continued)

Offset	Name	Length	Description
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: 0cyydddF). If the start date is not available, the value specified is 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: 0cyydddF). If the end date is not available, the value specified is 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 (z/OS version V1R8 and later)
132	SMF119FT_FSIDConnID	4	TCP connection ID of FTP data connection, or zero (z/OS version V1R8 and later)
136	SMF119FT_FSISessionID	15	FTP activity logging session ID (z/OS version V1R8 and later)
151	Reserved	1	Reserved (z/OS version V1R8 and later)

Table 68 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 68. FTP server hostname section

Offset	Name	Length	Description
0	SMF119FT_Hostname	*n*	Hostname

Table 69 on page 587 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 is included for Rename File Transfer operations.

Table 69. FTP server MVS data set name section

Offset	Name	Length	Description
0	SMF119FT_MVSDataSet	44	MVS Data set name

Table 70 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 70. 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	*n*	z/OS UNIX file name
2+*n*	SMF119FT_HFSLen2	2	Length of second z/OS UNIX file name (0 if only one z/OS UNIX file name is being reported)
4+*n*	SMF119FT_HFSName2	*m*	z/OS UNIX file name

Table 71 displays the FTP server security section.

Table 71. 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 71. FTP server security section (continued)

Offset	Name	Length	Format	Description
4 (X'4')	SMF119FT_FSProtoLevel	8	EBCDIC	Protocol level (present only if Protocol Mechanism is TLS or AT-TLS). Possible values are: • SSLV2 • SSLV3 • TLSV1 • TLSV1.1
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, TLSV1, or TLSV1.1: • 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_FSProtoBufSize	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).
38 (X'26')	SMF119FT_FSFips140	1	Binary	FIPS 140 status • X'00': FIPS 140 off • X'01': FIPS 140 on

Real-time SMF NMI: FTP client transfer initialization record (subtype 101)

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 is 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 (z/OS version V1R5 and later)
64 (x'40')	SMF119S4Len	2	Binary	Length of FTP client security section (z/OS version V1R5 and later)
66 (x'42')	SMF119S4Num	2	Binary	Number of FTP client security sections (z/OS version V1R5 and later)
68 (x'44')	SMF119S5Off	4	Binary	Offset to FTP client user name section (z/OS version V1R8 and later)
72 (x'48')	SMF119S5Len	2	Binary	Length of FTP client user name section (z/OS version V1R8 and later)
74 (x'4A')	SMF119S5Num	2	Binary	Number of FTP client user name sections (z/OS version V1R8 and later)
* See "Common TCP/IP identification section" on page 748 for the contents of the TCP/IP stack identification section.				

Table 72 on page 590 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 72. FTP client transfer initialization record section

Offset	Name	Length	Description
0	SMF119FT_FCICmd	4	FTP subcommand (according to RFC 959; see Appendix H, "Related protocol specifications," on page 991 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 Uuser 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 hundreth of a second, since midnight [using Coordinated Universal Time (UTC)]
104	SMF119FT_FCISDate	4	Start date of data connection (format: *0cyydddF*). If the start date is not available, the value specified is X'0000000F'.
108	SMF119FT_FCICSTime	4	Start time of control connection, in a hundreth of a second, since midnight [using Coordinated Universal Time (UTC)]. FTP session start time.

Table 72. FTP client transfer initialization record section (continued)

Offset	Name	Length	Description
112	SMF119FT_FCICSSDate	4	Start date of the control connection (format *0cyydddF*). If the start date is not available, the value specified is 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 (z/OS version V1R8 and later)
132	SMF119FT_FCIDConnID	4	TCP connection ID of FTP data connection (z/OS version V1R8 and later)

Table 73 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 73. FTP client associated data set name section

Offset	Name	Length	Description
0	SMF119FT_FCFileName	*n*	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 74 describes the FTP client SOCKS section, which is present only if the connection passes through a SOCKS server.

Table 74. 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 75 on page 592 describes the FTP client security section.

Table 75. 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 TLS or AT-TLS). Possible values are: • SSLV2 • SSLV3 • TLSV1 • TLSV1.1

Table 75. 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, TLSV1, or TLSV1.1 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).
38 (X'26')	SMF119FT_FCFips140	1	Binary	FIPS 140 status • X'00': FIPS 140 off • X'01': FIPS 140 on

Table 76 describes the FTP client user name section.

Table 76. FTP client user name section

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FCUserID	*n*	EBCDIC	User name or user ID used to log into the FTP server.

Real-time SMF NMI: FTP client login failure record (subtype 102)

Table 77 on page 594 describes the FTP client login failure self-defining section of the SMF record.

Table 77. 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
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 (z/OS version V1R8 and later)
64 (x'40')	SMF119S4Len	2	Binary	Length of FTP client user name section (z/OS version V1R8 and later)
66 (x'42')	SMF119S4Num	2	Binary	Number of FTP client user name sections (z/OS version V1R8 and later)
* See "Common TCP/IP identification section" on page 748 for the contents of the TCP/IP identification section.				

Table 78 shows the client login failure session section, which follows the TCP/IP identification section.

Table 78. 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)

Table 78. Client login failure session section (continued)

Offset	Name	Length	Format	Description
36(x'24')	SMF119FT_FCLUserID	8	EBCDIC	Local user ID
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 *z/OS Communications Server: IP User's Guide and Commands*. 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_FCLCConnID	4	Binary	TCP connection ID of FTP control connection

Table 79 describes the FTP client SOCKS section, which is present only if the connection passes through a SOCKS server.

Table 79. 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 80 defines the FTP client login failure security section.

Table 80. 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 80. 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 • TLSV1.1

Table 80. 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, TLSV1, or TLSV1.1 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).
38 (X'26')	SMF119FT_FCLFips140	1	Binary	FIPS 140 status • X'00': FIPS 140 off • X'01': FIPS 140 on

Table 81 shows the FTP client user name section.

Table 81. FTP client user name section

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FCLUserID	*n*	EBCDIC	User name or user ID used to log on to the FTP server

Real-time SMF NMI: FTP client session record (subtype 103)

Table 82 shows the FTP client session record self-defining section.

Table 82. 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
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 "Common TCP/IP identification section" on page 748 for the contents of the TCP/IP identification section.				

Table 83 shows the FTP client session section.

Table 83. 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

Table 83. FTP client session section (continued)

Offset	Name	Length	Format	Description
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.
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 hundreths of a second, since midnight [using Coordinated Universal Time (UTC)].
56 (X'38')	SMF119FT_FCNSDate	4	Binary	Session start date (format: *0cyydddF*). If the date is not available, the value specified is X'0000000F'.
60 (X'3C')	SMF119FT_FCNETime	4	Binary	Session end time, in one hundreths 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: *0cyydddF*). 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 84 shows the FTP client SOCKS section, which is present only if the connection passes through a SOCKS server.

Table 84. FTP client SOCKS section

Offset	Name	Length	Description
0	SMF119FT_FCNIP	16	SOCKS server IP address
16	SMF119FT_FCNPort	2	SOCKS server port number

Table 84. FTP client SOCKS section (continued)

Offset	Name	Length	Description
18	SMF119FT_FCNProt	1	SOCKS protocol version. Possible values are: • X'01' SOCKS Version 4 • X'02' SOCKS Version 5

Table 85 shows the FTP client security section:

Table 85. 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 • TLSV1.1

Table 85. 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, TLSV1, or TLSV1.1 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_FCNProtoBufSize	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).
38 (X'26')	SMF119FT_FCNFips140	1	Binary	FIPS 140 status • X'00': FIPS 140 off • X'01': FIPS 140 on

Table 86 shows the FTP client session user name section.

Table 86. FTP client session user name section

Offset	Name	Length	Format	Description
0(x'0')	SMF119FT_FCNUserID	*n*	EBCDIC	User name or user ID used to log into the FTP server.

Real-time SMF NMI: FTP server session record (subtype 104)

Table 87 describes the FTP server session record.

Table 87. 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
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 "Common TCP/IP identification section" on page 748 for the contents of the TCP/IP identification section.				

Table 88 describes the server session section.

Table 88. 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

Table 88. Server session section (continued)

Offset	Name	Length	Format	Description
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.
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 hundreths of a second since midnight [using Coordinated Universal Time (UTC)].
56 (X'38')	SMF119FT_FSRSDate	4	Binary	Session start date (format: *0cyydddF*). If the date is not available, the value specified is X'0000000F'. (FTP sessions start date).
60 (X'3C')	SMF119FT_FSRETime	4	Binary	Session end time in hundreths 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: *0cyydddF*). If the date is not available, the value specified is X'0000000F'. This field is defined only when the value of the SMF119FT_FSREvent field is T.
68 (X'44')	SMF119FT_FSRCConnID	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 89 describes the FTP server security section.

Table 89. 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_FSNCProtect	1	EBCDIC	Control Connection Protection Level. Possible values are: • N: None • C: Clear • S: Safe • P: Private
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 • TLSV1 • TLSV1.1

Table 89. FTP server security section (continued)

Offset	Name	Length	Format	Description
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, TLSV1, or TLSV1.1: • 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).
38 (X'26')	SMF119FT_FSNFips140	1	Binary	FIPS 140 status • X'00': FIPS 140 off • X'01': FIPS 140 on

Resolver NMI (EZBREIFR)

z/OS Communications Server provides a high-speed low-overhead callable programming interface for network management applications to access data related to the resolver. Use the EZBREIFR network management interface to monitor resolver configuration information and GLOBALTCPIPDATA file contents.

This section describes the details for invoking the EZBREIFR interface with the defined input parameters and for processing the output it provides. The following topics are addressed:

- "Resolver NMI: Overview" on page 606
- "Resolver NMI: Configuration and enablement" on page 606
- "Resolver NMI: Using the EZBREIFR requests" on page 606

Resolver NMI: Overview

You can invoke the EZBREIFR interface to return data related to the resolver at a given point in time. You cannot specify any filters on the resolver NMI requests to limit the returned information to a specific set of information.

You can obtain the resolver setup information, including the contents of the GLOBALTCPIPDATA file if one is defined, from the resolver.

Resolver NMI: Configuration and enablement

There is no configuration required to enable the resolver interface.

Resolver NMI: Using the EZBREIFR requests

This topic describes the program requirements for invoking the resolver callable NMI and it includes examples of the invocations.

EZBREIFR requirements

Table 90 identifies the authorization requirements for EZBREIFR requests.

Table 90. EZBREIFR requests

Minimum authorization	Supervisor state, or system key, or APF authorization
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

EZBREIFR format

For C/C++ callers, invoke EZBREIFR as shown in the following example:

```
EZBREIFR(RequestResponseBuffer,
             &RequestResponseBufferAlet,
             &RequestResponseBufferLength,
             &ReturnValue,
             &ReturnCode,
             &ReasonCode);
```

For assembler callers, invoke EZBREIFR as shown in the following example:

```
CALL EZBREIFR,(RequestResponseBuffer,
                   RequestResponseBufferAlet,
                   RequestResponseBufferLength,
                   ReturnValue,
                   ReturnCode,
                   ReasonCode)
```

EZBREIFR parameters

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 (NMSHeader), as specified in the EZBRENMC header file. When the request completes successfully, the storage contains an output response in the same format.

RequestResponseBufferAlet
Supplied parameter.

Type:	Integer
Length:	Fullword

The name of a fullword that contains the access list entry token (ALET) of the *RequestResponseBuffer* parameter. If a nonzero ALET is specified, the ALET must represent a valid entry in the dispatchable unit access list (DU-AL) of the caller.

RequestResponseBufferLength
Supplied parameter.

Type:	Integer
Length:	Fullword

The name of a fullword that contains the length of request/response buffer.

If the buffer length is too short to contain all of the requested information, the request fails with the return code ENOBUFS. The length that is required to contain all of the information is provided in the NMSHBytesNeeded field of the NMSHeader data structure of the response. If the buffer length is not the minimum size for the request, the request fails with the return code ENOBUFS; however, the value that is required is not provided in the NMSHBytesNeeded field. The minimum size is the length of the NMSHeader data structure.

ReturnValue
Returned parameter.

Type:	Integer
Length:	Fullword

The name of a fullword in which the EZBREIFR service returns one of the following:

- 0 or positive integer, if the request is successful. A value greater than 0 specifies the number of output data bytes copied to the response buffer. See "Resolver NMI: Request and response formats" on page 609 for additional details about 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 EZBREIFR service stores the return code (errno). The EZBREIFR service returns the *ReturnCode* parameter only if the *ReturnValue* value is -1.

ReasonCode
Returned parameter.

Type: Integer

Length: Fullword

The name of a fullword in which the EZBREIFR service stores the reason code (errnojr). The EZBREIFR service returns the *ReasonCode* parameter only if the *ReturnValue* value is -1. The *ReasonCode* parameter further qualifies the *ReturnCode* value.

The EZBREIFR service sets the return codes and reason codes described in Table 91. See *z/OS UNIX System Services Messages and Codes* for the hexadecimal values of the return and reason codes.

Table 91. EZBREIFR service return codes and reason codes

ReturnValue	ReturnCode	ReasonCode	Meaning
0	0	0	The request was successful.
-1	ENOBUFS	JRBuffTooSmall	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. If the buffer was large enough for a complete NMSHeader to be returned, the NMSHeader NMSHBytesNeeded field might contain the buffer size to return all of the requested information. See the description of the RequestResponseBufferLength parameter for an explanation of when the NMSHBytesNeeded value is provided.
-1	EACCES	JRSAFNotAuthorized	The request was not successful. The caller is not authorized.
-1	EAGAIN	JRInactive	The request was not successful. The resolver is 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.

Table 91. EZBREIFR service return codes and reason codes (continued)

ReturnValue	ReturnCode	ReasonCode	Meaning
-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. An invalid value was specified in the request/response header, or a filter was provided on the request.
-1	EMVSERR	JRUnexpectedErr	The request was not successful. An unexpected error occurred.

An application can use any of the following methods to invoke the EZBREIFR service:

- Issue a LOAD macro to obtain the EZBREIFR service entry point address, and then issue a CALL macro specifying that address. The EZBREIFR load module must reside in a linklist dataset (for example, the SEZALOAD load library of TCP/IP), or in the LPA.
- Issue a LINK macro to invoke the EZBREIFR service. The EZBREIFR load module must reside in a link list data set (for example, the SEZALOAD load library of TCP/IP), or in the link pack area (LPA).
- Link-edit EZBREIFR directly into the application load module, and then issue a CALL macro specifying EZBREIFR. Include SYS1.CSSLIB(EZBREIFR) in the application load module link-editing.
- For 64-bit C/C++ applications, link-edit the EZBREIF4 program directly into the application load module, and then issue a CALL macro specifying EZBREIFR. Include SYS1.CSSLIB(EZBREIF4) in the application load module link-editing.

Resolver NMI: Request and response formats

Resolver NMI requests and responses share a common format. A resolver NMI request or response consists of a header followed by zero or more records.

The structures described in the following tables are part of the Network Management Service component.

Table 92 describes the fields in the buffer header (NMSHeader) structure.

Table 92. Buffer header (NMSHeader) structure

Field	Offset decimal	Length in bytes	Format	Description
NMSHeaderIdent	0	4	EBCDIC	Header identifier. Set to NMSHeaderIdentifier (EBCDIC 'NMSH').
NMSHeaderLength	4	4	Binary	Length of record. For a request, the client application should set this field to be the length of the NMS header. The server returns the same value in the response data.
NMSHVersion	8	2	Binary	Network monitor version. Currently, only version 1 is supported by this interface (NMSHVersion1).

Table 92. Buffer header (NMSHeader) structure (continued)

Field	Offset decimal	Length in bytes	Format	Description
NMSHType	10	2	Binary	Network monitor type. For a request, this indicates the type of the request. For a response, this indicates the type of response data and is identical to the request type. See "Resolver NMI: Requests" on page 611 for a description of the request types.
NMSHBytesNeeded	12	4	Binary	Length of buffer required to contain all requested data. For a request, the server ignores this field. For a response, the server sets this field to the number of bytes that the client application must provide to obtain all the resolver configuration data. If the client application provides a smaller buffer than the value that is specified in the NMSHBytesNeeded field, the server provides only the records that fit in the buffer.
Reserved	16	20	N/A	N/A
NMSHInputDataDescriptors	36	12	Binary	Input section descriptors. The client sets this field to 0 on input because EZBREIFR does not accept input section descriptors on the request. This descriptor is described by the NMSTriplet structure. See Table 93 for details.
NMSHSetup	48	16	Binary	Output record descriptor. The client application should set this field to 0 on input. The server completes this field with information describing the records that contain the setup record data. This descriptor is described by the NMSQuadruplet structure. See Table 94 on page 611 for details.
NMSHGlobalTcpip	64	16	Binary	Output record descriptor. The client application should set this field to 0 on input. The server completes this field with information describing the records that contain the global TCPIP.DATA record data. This descriptor is described by the NMSQuadruplet structure. See Table 94 on page 611 for details.

Table 93 describes the fields in the triplet (NMSTriplet) structure.

Table 93. Triplet (NMSTriplet) structure

Field	Offset decimal	Length in bytes	Format	Description
NMSTOffset	0	4	Binary	Offset to section
NMSTLength	4	4	Binary	Length of each section
NMSTNumber	8	4	Binary	Number of sections

Table 94 on page 611 describes the fields in the quadruplet (NMSQuadruplet) structure.

Table 94. Quadruplet (NMSQuadruplet) structure

Field	Offset decimal	Length in bytes	Format	Description
NMSQOffset	0	4	Binary	Offset to section
NMSQLength	4	4	Binary	Length of each section
NMSQNumber	8	4	Binary	Number of each section
NMSQMatch	12	4	Binary	Number of sections that matched filters

A buffer 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 in the record, followed by one or more sections that contain the actual record data.

Table 95 describes the fields in the record header (NMSRecHeader) structure.

Table 95. Record header (NMSRecHeader) structure

Field	Offset decimal	Length in bytes	Format	Description
NMSRecHdrIdent	0	4	EBCDIC	Record identifier; always the value NMSR.
NMSRecType	4	4	EBCDIC	Record type. One of two values: **RSET** setup record data **RGTD** global TCPIP.DATA record data
NMSRecLength	8	4	Binary	Total record length
NMSRecNumber	12	4	Binary	Number of sections descriptors

Each section descriptor is mapped by triplet (NMSTriplet) structure; see Table 93 on page 610.

The records for a request or response can differ in length because some data is present or absent, or because there is variable-length data. 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. If applications check the validity of a section length, the applications should always test for a length that is greater than or equal to the expected length to ensure that applications are compatible with future releases

Resolver NMI: Requests

The GetResolverConfig (NMSHGetResolverConfig) request is supported by EZBREIFR. The request obtains resolver setup information, including the contents of the GLOBALTCPIPDATA file, if that file is defined. The request constant, which is specified in the NMSHType field in the NMSHeader data structure, follows the request name.

The general format of a resolver NMI request consists of a request header. There are currently no input filters defined for EZBREIFR; the NMSHInputDataDescriptors field should be set to 0. If you specify a filter on an EZBREIFR request, the request fails.

Resolver NMI: Responses

The GetResolverConfig (NMSHGetResolverConfig) response is provided by EZBREIFR. The response returns information about resolver setup information, including the contents of the GLOBALTCPIPDATA file, if that file is defined. The response constant, which is specified in the NMSHType field in the NMSHeader data structure, follows the response name.

The general format of a resolver NMI response is as follows:

- The response header that is mapped by buffer header (NMSHeader) structure; see Table 93 on page 610
- One or more response records

The response header contains the number of bytes required to contain all the requested data (NMSHBytesNeeded). When the return code is ENOBUFS, use the number of bytes value to allocate a larger request or response buffer and reissue the request.

GetResolverConfig response contents: The GetResolverConfig response consists of two records:

- A setup record
- Optionally, a global TCPIP.DATA record

A global TCPIP.DATA record is provided whenever the GLOBALTCPIPDATA resolver setup statement is specified in the resolver setup file, unless the resolver is unable to access the file. For example, the global TCPIP.DATA record might be provided if the MVS sequential file specified on the GLOBALTCPIPDATA statement is being edited when the resolver attempts to read the file contents. The resolver notifies you of the failure to read the global TCPIP.DATA file contents by issuing message EZZ9297E UNABLE TO ACCESS FILE filename - RC returncode. See *z/OS Communications Server: IP Messages Volume 4 (EZZ, SNM)* for information about this message and for suggestions about how to correct the file access problems.

A setup record consists of a record header that contains six section descriptors (see Table 96 for details) and one section of resolver configuration data.

Table 96. Setup record (NMSSetupRecord) structure

Field	Offset decimal	Length in bytes	Format	Description
NMSSetupHeader	0	16	Binary	Setup record heading mapping for setup record. See Table 95 on page 611 for details.
NMSSetupDataSection	16	12	Binary	Setup data record section descriptor. The server completes this field with information describing the setup record data section. This descriptor is described by the NMSTriplet structure; see Table 93 on page 610 for details.

Table 96. Setup record (NMSSetupRecord) structure (continued)

Field	Offset decimal	Length in bytes	Format	Description
NMSSetupGlobalTcpipData	28	12	Binary	Setup global TCPIP.DATA data record section descriptor. The server completes this field with information describing the setup file name section that maps the GLOBALTCPIPDATA statement. This descriptor is described by the NMSTriplet structure; see Table 93 on page 610 for details.
NMSSetupDefaultTcpipData	40	12	Binary	Setup default TCPIP.DATA data record section descriptor. The server completes this field with information describing the setup file name section that maps the DEFAULTTCPIPDATA statement. This descriptor is described by the NMSTriplet structure; see Table 93 on page 610 for details.
NMSSetupGlobalIpnodes	52	12	Binary	Setup global IPNODES record section descriptor. The server completes this field with information describing the setup file name section that maps the GLOBALIPNODES statement. This descriptor is described by the NMSTriplet structure; see Table 93 on page 610 for details.
NMSSetupDefaultIpnodes	64	12	Binary	Setup default IPNODES record section descriptor. The server completes this field with information describing the setup file name section that maps the DEFAULTIPNODES statement. This descriptor is described by the NMSTriplet structure; see Table 93 on page 610 for details.
NMSSetupFile	76	12	Binary	Resolver setup file name. The server completes this field with the name of the optional resolver setup file name (either an MVS data set or a z/OS UNIX file) that contains resolver configuration statements. This descriptor is described by the NMSTriplet structure; see Table 93 on page 610 for details.

Table 97 on page 614 describes the fields in the setup record data section (NMSSetupData) structure.

Table 97. Setup record data section (NMSSetupData) structure

Field	Offset decimal	Length in bytes	Format	Description
NMSSOptions	0	4	Binary	Options that were specified in the resolver setup file, if one is in use. • bit 0 = CACHE • bit 1 = NOCACHE • bit 2 = CACHESIZE • bit 3 = COMMONSEARCH • bit 4 = NOCOMMONSEARCH • bit 5 = DEFAULTIPNODES • bit 6 = DEFAULTTCPIPDATA • bit 7 = GLOBALIPNODES • bit 8 = GLOBALTCPIPDATA • bit 9 = MAXTTL • bit 10 = UNRESPONSIVETHRESHOLD • bit 11= AUTOQUIESCE (see Note)
NMSSDefaults	4	4	Binary	Options that were not specified in the resolver setup file but that have default settings that were used. • bit 0 = CACHE • bit 2 = CACHESIZE • bit 4 = NOCOMMONSEARCH • bit 9 = MAXTTL • bit 10 = UNRESPONSIVETHRESHOLD
NMSSFlags	8	4	Binary	Flag options that indicate which resolver functions are active, based on the contents of the resolver setup file. • bit 0 = CACHE • bit 1 = NOCACHE • bit 3 = COMMONSEARCH • bit 4 = NOCOMMONSEARCH • bit 11= AUTOQUIESCE (see Note)
Reserved	12	4	N/A	N/A
NMSSetupCacheSize	16	8	Binary	CACHESIZE value, in numbers of bytes
NMSSetupMaxTTL	24	4	Binary	MAXTTL value, in seconds
NMSSetupUnresponsiveThreshold	28	4	Binary	UNRESPONSIVETHRESHOLD value
Note: If the GLOBALTCPIPDATA statement is not coded, although the NMSSOptions field indicates that the AUTOQUIESCE operand is set in the resolver setup file, the NMSSFlags field will indicate that the AUTOQUIESCE function is not active.				

There are zero to five instances of setup record file name section (NMSSetupFileNames) structure; see Table 98 on page 615 for details about the fields. Each of the instances either maps the resolver setup file name or one of the following four resolver setup statements that specify a file name:

- GLOBALTCPIPDATA
- DEFAULTTCPIPDATA
- GLOBALIPNODES
- DEFAULTIPNODES

You must use the section descriptors that are described in Table 96 on page 612 to identify the use of a given instance of the record file name section structure.

Table 98. Setup record file name section (NMSSetupFileNames) structure

Field	Offset decimal	Length in bytes	Format	Description
NMSSetupFileName	0	Variable	EBCDIC	Setup record file name

A global TCPIP.DATA record consists of the following sections:

- A record header that contains four section descriptors; see Table 99 for details.
- One section of TCPIP.DATA configuration information from the GLOBALTCPIPDATA file; see Table 100 on page 617 for details.
- Zero or more Domain Name System (DNS) name server IP addresses; see Table 101 on page 619 for details.

  Each DNS address represents a value that is coded on either a NSINTERADDR or a NAMESERVER TCPIP.DATA statement. Use the settings in the NMSGOptions field to identify which TCPIP.DATA statement was used to code the addresses returned in the resolver NMI response. The resolver can return up to sixteen DNS address sections.
- Zero or more domain search names; see Table 102 on page 619 for details.

  Each domain search name represents a value coded on a DOMAIN, DOMAINORIGIN, or SEARCH TCPIP.DATA statement. Use the settings in the NMSGOptions field to identify which TCPIP.DATA statement was used to set the values mapped in the resolver NMI response. There can be up to six domain search name sections. Names shorter than the value specified for the NMSTLength in the NMSGtdNameSection field are padded with null characters.
- Zero or more DCBS table names; see Table 103 on page 619 for details.

  Each table name represents a value coded on the LOADDBCSTABLES TCPIP.DATA statement. The resolver can return up to nine table name sections. Table names shorter than 8 characters are padded with null characters.

Table 99. Global TCPIP.DATA record (NMSGtdRecord) structure

Field	Offset decimal	Length in bytes	Format	Description
NMSGtdHeader	0	16	Binary	Record heading mapping for global TCPIP.DATA record. See Table 95 on page 611 for details.
NMSGtdDataSection	16	12	Binary	Resolver global TCPIP.DATA data section. The server completes the field with information describing the Network Management global TCPIP.DATA record data section. This descriptor is described by the NMSTriplet structure; see Table 93 on page 610 for details.
NMSGtdNsadSection	28	12	Binary	Resolver global TCPIP.DATA record NAMESERVER values section. The server completes the field with information describing the global TCPIP.DATA record DNS addresses section. This descriptor is described by the NMSTriplet structure; see Table 93 on page 610 for details.

Table 99. Global TCPIP.DATA record (NMSGtdRecord) structure (continued)

Field	Offset decimal	Length in bytes	Format	Description
NMSGtdNameSection	40	12	Binary	Resolver global TCPIP.DATA record SEARCH values section. The server completes the field with information describing the global TCPIP.DATA record domain search names section. This descriptor is described by the NMSTriplet structure; see Table 93 on page 610 for details.
NMSGtdLoadDbcsTables	52	12	Binary	Resolver global TCPIP.DATA record LOADDBCSTABLES values. The server completes the field with information describing global TCPIP.DATA record DCBS table names section. This descriptor is described by the NMSTriplet structure; see Table 93 on page 610 for details.

Table 100 on page 617 describes the fields in the global TCPIP.DATA record data section (NMSGtdData) structure.

Table 100. Global TCPIP.DATA record data section (NMSGtdData) structure

Field	Offset decimal	Length in bytes	Format	Description
NMSGOptions	0	8	Binary	Options specified in the global TCPIP.DATA file. • Bit 0 = ALWAYSWTO NO • Bit 1 = ALWAYSWTO YES • Bit 2 = DATASETPREFIX • Bit 3 = DOMAIN • Bit 4 = DOMAINORIGIN • Bit 5 = HOSTNAME • Bit 6 = LOADDBCSTABLES • Bit 7 = LOOKUP DNS • Bit 8 = LOOKUP DNS LOCAL • Bit 9 = LOOKUP LOCAL • Bit 10 = LOOKUP LOCAL DNS • Bit 11 = MESSAGECASE MIXED • Bit 12 = MESSAGECASE UPPER • Bit 13 = NOCACHE • Bit 14 = NAMESERVER • Bit 15 = NSINTERADDR • Bit 16 = NSPORTADDR • Bit 17 = OPTIONS DEBUG • Bit 18 = OPTIONS NDOTS • Bit 19 = RESOLVERTIMEOUT • Bit 20 = RESOLVERUDPRETRIES • Bit 21 = RESOLVEVIA TCP • Bit 22 = RESOLVEVIA UDP • Bit 23 = SEARCH • Bit 24 = SOCKDEBUG • Bit 25 = SOCKNOTESTSTOR • Bit 26 = SOCKTESTSTOR • Bit 27 = SORTLIST • Bit 28 = TCPIPJOBNAME • Bit 29 = TCPIPUSERID • Bit 30 = TRACE RESOLVER • Bit 31 = TRACE SOCKET
NMSGDefaults	8	8	Binary	The default values of the options specified in the global TCPIP.DATA file. • Bit 0 = ALWAYSWTO NO • Bit 2 = DATASETPREFIX • Bit 5 = HOSTNAME • Bit 8 = LOOKUP DNS LOCAL • Bit 11 = MESSAGECASE MIXED • Bit 13 = NOCACHE • Bit 16 = NSPORTADDR • Bit 18 = OPTIONS NDOTS • Bit 19 = RESOLVERTIMEOUT • Bit 20 = RESOLVERUDPRETRIES • Bit 22 = RESOLVEVIA UDP • Bit 25 = SOCKNOTESTSTOR • Bit 28 = TCPIPJOBNAME

Table 100. Global TCPIP.DATA record data section (NMSGtdData) structure (continued)

Field	Offset decimal	Length in bytes	Format	Description
NMSGFlags	16	8	Binary	Flag options that indicate which capabilities are in effect, based on the contents of the global TCPIP.DATA file. • Bit 0 = ALWAYSWTO NO • Bit 1 = ALWAYSWTO YES • Bit 7 = LOOKUP DNS • Bit 8 = LOOKUP DNS LOCAL • Bit 9 = LOOKUP LOCAL • Bit 10 = LOOKUP LOCAL DNS • Bit 11 = MESSAGECASE MIXED • Bit 12 = MESSAGECASE UPPER • Bit 13 = NOCACHE • Bit 17 = OPTIONS DEBUG • Bit 21 = RESOLVEVIA TCP • Bit 22 = RESOLVEVIA UDP • Bit 24 = SOCKDEBUG • Bit 25 = SOCKNOTESTSTOR • Bit 26 = SOCKTESTSTOR • Bit 30 = TRACE RESOLVER • Bit 31 = TRACE SOCKET
NMSGtdHostName	24	64	EBCDIC	HOSTNAME value. The name is padded with null characters.
NMSGtdOptionsNdots	88	2	Binary	OPTIONS NDOTS value.
NMSGtdNsPortAddr	90	2	Binary	NSPORTADDR value.
NMSGtdResolverTimeout	92	4	Binary	RESOLVERTIMEOUT value, in seconds.
NMSGtdResolverTimeoutMsecs	96	4	Binary	RESOLVERTIMEOUT value, in milliseconds.
NMSGtdResolverUdpRetries	100	4	Binary	RESOLVERUDPRETRIES value
NMSGtdSortList (4)	104	32	Binary	Up to four SORTLIST values. Each individual SORTLIST entry contains the following two values: • Bits 0 - 31 = SORTLIST IPv4 address • Bits 32 - 63 = SORTLIST network mask
NMSGtdTcpipNames	136	9	EBCDIC	TCPIPJOBNAME or TCPIPUSERID value. Use the settings in the NMSGOptions field or the NMSGDefaults field to identify which statement was used. The name is padded with null characters.
NMSGtdDatasetPrefix	145	27	EBCDIC	DATASETPREFIX value. The name is padded with null characters.
Reserved	172	20	N/A	N/A

Guideline: Do not specify OPTIONS DEBUG or TRACE RESOLVER in your global TCPIP.DATA file.

Table 101 on page 619 describes the fields in the global TCPIP.DATA record DNS addresses (NMSGtdDnsAddresses) structure.

Table 101. Global TCPIP.DATA record DNS addresses (NMSGtdDnsAddresses) structure

Field	Offset decimal	Length in bytes	Format	Description
NMSGtdSockAddrs	0	28	Socket address	DNS names server IP addresses.

Table 102 describes the fields in the global TCPIP.DATA record structure.

Table 102. Global TCPIP.DATA record structure

Field	Offset decimal	Length in bytes	Format	Description
NMSGtdSearchNameSize	0	2	Binary	Length of domain search name.
NMSGtdSearchName	2	Variable	EBCDIC	Domain search name.

Table 103 describes the fields in the global TCPIP.DATA record DCBS table names section (NMSGtdDbcsNames).

Table 103. Global TCPIP.DATA record DCBS table names section (NMSGtdDbcsNames)

Field	Offset decimal	Length in bytes	Format	Description
NMSGtdDbcsName	0	8	EBCDIC	DBCS table name.

Resolver NMI: Request and response data structures

The resolver NMI request and response data structures for C/C++ and assembler programs are located as shown in Table 104. The header files and macros are included in the SEZANMAC data set. The header file is also included in the z/OS UNIX file system directory, /usr/include. When you compile or assemble a program in an MVS batch job, the SEZANMAC data set must be available in the MVS batch job concatenation.

Table 104. Location of resolver NMI request and response data structures for C/C++ and assembler programs

Header file for C/C++ programs	Macros for assembler programs	Contents
EZBRENMC	EZBRENMA	The resolver callable NMI (EZBREIFR) request and response data structure definitions.

Resolver NMI: Examples

The following C/C++ code fragment shows how to format a request to obtain current resolver configuration information:

```
/*********************************************************************/
/*                                                                   */
/* NMI data definitions                                              */
/*                                                                   */
/*********************************************************************/
typedef struct {
NMSHeader NMIheader;
} 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.NMSHeaderIdent=NMSHIDENTIFIER;
NMIbuffer->NMIheader.NMSHeaderLength=sizeof(NMSHeader);
NMIbuffer->NMIheader.NMSHVersion=NMSHVERSION1;
NMIbuffer->NMIheader.NMSHType=NMSHGETRESOLVERCONFIG;
NMIbuffer->NMIheader.NMSHBytesNeeded=0;
/*********************************************************************/
/*                                                                   */
/* Resolver NMI does not currently have filters                      */
/*                                                                   */
/*********************************************************************/
NMIbuffer->NMIheader.NMSHInputDataDescriptors.\
  NMSHFilters.NMSTOffset=0;
NMIbuffer->NMIheader.NMSHInputDataDescriptors.\
  NMSHFilters.NMSTLength=0;
NMIbuffer->NMIheader.NMSHInputDataDescriptors.\
  NMSHFilters.NMSTNumber=0;
/*********************************************************************/
/*                                                                   */
/* Invoke NMI service                                                */
/*                                                                   */
/*********************************************************************/
NMService(NMIbuffer,&NMIalet,&NMIlength,&RV,&RC,&RSN);
```

The following assembler code fragment shows how to format a request to obtain current resolver configuration information:

```
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
         BAKR  R14,R0             SAVE REGISTER
         LARL  R11,STATIC_AREA    LTORG AND STATIC AREA
         USING (STATIC_AREA,STATIC_AREA_END),R11    ADDRESSEBILITY
         LA    R0,WORKL                LENGTH OF DYNAMIC STORAGE AREA
         GETMAIN R,LV=(0)              ALLOCATE DYNAMIC STORAGE
         LR    R13,R1                  WHERE DID WE FIND IT
         USING WORK,R13                ADDRESSABILITY FOR IT
         L     R0,=A(262344)           256K + 200 BYTES
         GETMAIN RU,LV=(R0),LOC=ANY    ALLOCATE NMI BUFFER AREA
         LR    R2,R1                   SAVE ITS ADDRESS
*********************************************************************
*                                                                   *
* FORMAT THE HEADER                                                 *
*                                                                   *
```

```
*********************************************************************
         USING NMSHEADER,R2                BASE IT ON OBTAINED STORAGE
         XC    NMSHEADER(NMSHEADERSIZE),NMSHEADER     CLEAR HEADER AREA
         MVC   NMSHEADERIDENT,=A(NMSHEADERIDENTIFIER) MOVE IN ITS ID
         MVC   NMSHEADERLENGTH,=A(NMSHEADERSIZE)      SET HEADER SIZE
         L     R0,=A(NMSHCURRENTVER)                  CURRENT VERSION
         STH   R0,NMSHVERSION                         INTO HEADER FIELD
         L     R0,=A(NMSHGETRESOLVERCONFIG)        RESOLVER CONFIG TYPE
         STH   R0,NMSHTYPE                            INTO HEADER FIELD
         XC    NMSHBYTESNEEDED,NMSHBYTESNEEDED        CLEAR BYTESNEEDED
*********************************************************************
*                                                                   *
*    RESOLVER NMI DOES NOT CURRENTLY HAVE FILTERS                   *
*                                                                   *
*********************************************************************
         XC    NMSHFILTERS,NMSHFILTERS                CLEAR IT
*********************************************************************
* INVOKE RESOLVER NMI SERVICE                                       *
*                                                                   *
*********************************************************************
         CALL  EZBREIFR,                                            *
               ((R2),NMIALET,NMILENGTH,RV,RC,RSN),MF=(E,WORK512)
         PR                      RETURN
         DROP  R2
         DROP  R13
         EJECT
STATIC_AREA DS 0D
MINUSONE  DC   F'-1'           RETURN VALUE INDICATES FAILURE
NMIALET   DC   A(0)            NO ALET USED
NMILENGTH DC   A(8192)         8K IS BIG ENOUGH
         LTORG
STATIC_AREA_END DS 0D
WORK     DSECT
WORK512  DS    XL512           PARM LIST AREA
RV       DC    F'0'            RETURN VALUE
RC       DC    F'0'            RETURN CODE
RSN      DC    A(0)            REASON CODE
         LTORG ,
WORKL    EQU   *-WORK
         EZBRENMA DSECT=YES,LIST=YES,TITLE=NO,NMSLIST=YES
```

SMF records

Installations can use Systems Management Facilities (SMF) records for purposes such as performance management, capacity planning, auditing, and accounting. z/OS Communications Server provides the following SMF record types:

- Type 109, Syslogd SMF records
- Type 118, event and interval records for TCP/IP and several applications; these records are no longer enhanced
- Type 119, event and interval records for TCP/IP and several applications; these records are continuing to be enhanced

These records can be processed from the MVS SMF data sets or from SMF exits. Some of the records can also be processed programmatically by using the real-time SMF NMI. See "Real-time TCP/IP network monitoring NMI" on page 564 for more information. See Appendix C, "Type 109 SMF records," on page 731, Appendix D, "Type 118 SMF records," on page 733, and Appendix E, "Type 119 SMF records," on page 743 for detailed layouts of the records.

For general information about the uses of SMF records see *z/OS Communications Server: IP Configuration Guide*.

SMF type 109 records

SMF type 109 records are created by the syslog daemon (syslogd). TCP/IP server applications and components use syslogd to write log messages and trace messages. If $SMF is specified as a destination for the messages, then syslogd creates an SMF type 109 record that contains the message. For information about configuring syslogd to create SMF type 109 records, see the information about the syslog daemon in *z/OS Communications Server: IP Configuration Reference*. See Appendix C, "Type 109 SMF records," on page 731 for the layout of the record.

SMF type 118 records

TCP/IP can create SMF type 118 records for certain events.

Tip: You should use SMF type 119 records instead of SMF type 118 records. SMF type 118 records have been stabilized and no new information is being added to them. SMF type 119 records include all the latest enhancements to SMF information created by TCP/IP.

If you are running multiple stacks, SMF does not always allow you to distinguish among them. Consider the following issues:

- There is no stack identity in SMF type 118 records. SMF records that are written by the system address space or by standard servers may be identified as belonging to one stack or another, based on address space naming conventions.
- SMF records written by client address spaces cannot be identified as belonging to a single stack based on the address space naming conventions used in standard servers.
- The only technique currently available to distinguish among records written by various client address spaces is to assign unique SMF type 118 record subtype intervals to each stack:
  - FTP server: One or nine subtypes in FTP.DATA
  - Telnet server: Two subtypes on TELNETPARMS
  - API: Two subtypes on SMFPARMS
  - FTP, Telnet client: One subtype on SMFPARMS

If you choose to assign subtypes, there will be an obvious impact on your local accounting programs. SMF type 118 subtype changes and additions must be coordinated with persons responsible for managing the use of SMF.

SMF type 118 records do not support IPv6 addresses. Thus, if you choose to exploit IPv6 in your environment, migrate your SMF processing to use the SMF type 119 records, which do support IPv6 addresses.

An external mapping (EZASMF76 macro) is available for customers to parse the SMF type 118 records that TCP/IP generates. EZASMF76 produces assembler level DSECTs for the Telnet (server and client), FTP (server and client), and API SMF records.

To create the Telnet SMF Record layout, code:

```
EZASMF76 TELNET=YES
```

To create the FTP SMF Record layout, code:

```
EZASMF76 FTP=YES
```

To create the API SMF Record layout, code:

```
EZASMF76 API=YES
```

SMF type 119 records

SMF type 119 records contain unique stack identification sections designed to eliminate the confusion of the type 118 records. They provide uniformity of date and time (UTC), common record format (self-defining section and TCP/IP identification section), and support for IPv6 addresses and expanded field sizes (64 bit versus 32 bit) for some counters. The following kinds of SMF type 119 records are available:

- TCP connection initiation and termination
- UDP socket close
- TCP/IP interface and server port statistics
- TCP/IP stack start/stop
- FTP server transfer completion
- FTP server logon failure
- FTP client transfer completion
- TN3270E Telnet server session initiation and termination
- Telnet client connection initiation and termination
- IKE tunnel activation, refresh, and expire
- Dynamic tunnel activation, refresh, installation, and removal
- Manual tunnel activation and deactivation
- TCP/IP profile
- CSSMTP processing of configuration files, spool files, mail messages, connections and statistical records
- DVIPAs and sysplex distributor targets

The SMF type 119 records utilize a common structure. Each record is organized as follows:

- SMF header
- Self-defining section containing pointers to:
  - TCP/IP identification section (identifies system, stack etc)
  - Sections containing the data for the record

You can parse the SMF type 119 records that TCP/IP generates by using macros and header files.

- For assembler applications, use the following macros:
  - EZASMF77, which is installed in SYS1.MACLIB.
  - EZBNMMPA, which is installed in TCP/IP data set SEZANMAC. This macro is needed only for the TCP/IP profile record.
- For C/C++ applications, use the following header files:
  - ezasmf.h
  - ezbnmmpc.h

    This header file is needed only for the TCP/IP profile record.

  These header files are installed in TCP/IP data set SEZANMAC, and in the /usr/include file system directory.

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 always contains CSM storage pool statistics and CSM summary information and can optionally contain CSM storage owner statistics.

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.

SNA network monitoring NMI 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.

SNA network monitoring NMI: 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 `NO`. The default for this start option is `NO`, 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 NO
  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=NO
  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=NO
  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

SNA network monitoring NMI: 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 closes 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 monitoring NMI request/response format" for details.

SNA network monitoring 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 564, which return a token to a response buffer that the application must use as input to one of the TMI copy buffer interfaces, to obtain the formatted response data.

The NMI request and response mappings are provided for programming to this interface.

SNA network monitoring NMI 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 and optionally, about CSM storage owner 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 Table 105 on page 627. 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.

Table 105. EE connection request filter parameters

Local Hostname	An EBCDIC name, right-padded with nulls or blanks if less than 64 characters long (applicable to CS for z/OS version 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 z/OS Communications Server 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 z/OS Communications Server V1R5 and later releases only).

– An EE Summary Request cannot contain any filter elements; no filters are applicable to an EE summary request.
– 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 (1 - 4 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, the value 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.

– A CSM Statistics Request can contain 1 – 4 filter elements to request CSM storage ownership statistics. To request statistics about all users that own CSM storage, include a filter element in the request that has an ASID set to the value 0. To request statistics about a subset of users that own CSM storage, supply filter elements that include a nonzero value filter parameter in Table 106 on page 628.

  If no filter element is provided on the CSM Statistics Request, no CSM storage ownership statistics are included in the response. The CSM Global Pool Output Section record and the CSM Summary Output Section record are always returned as part of a CSM Statistics Response, regardless of whether filters are included on the request or not.

Restriction: ASID filter parameters are applicable only to z/OS Communications Server V1R11 and later. If the initialization record that was received by the client when the connection was opened specifies that CSM Statistics Request filters are not supported by this VTAM level (any VTAM level prior to z/OS V1R11), then the server rejects any request that contains a filter on the CSM Statistics Request.

ASID	A 16-bit integer. Specify the ASID value 0 to request all storage owner statistics.

Table 106 shows which filter parameters are required, optional, or not applicable (N/A) for each request type. If you specify inapplicable filters for a particular request type, an EE Connection Request, HPR Connection Request, or CSM Statistics Request, they are ignored. EE Summary Requests that contain filter elements are rejected by VTAM.

Table 106. Required filter parameters

Request Type	Local IP Address or Hostname	Remote IP Address or Hostname	RTP PU name or Partner CP name	COS name	ASID
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	N/A
EE Summary Request	N/A	N/A	N/A	N/A	N/A
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	N/A
CSM Statistics Request	N/A	N/A	N/A	N/A	Optional; if a filter is provided, an ASID value is required

Every valid request record that is 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 the start of the request header to this data indicated in the Input Triplet)

SNA network monitoring NMI 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 the value 0 on responses that contain 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 that contain information that passed the request filter checks.

The general format of an NMI response section record is as follows:

- 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 that are associated with this response section record.

An application that navigates 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 Statistics Response
  1. One CSM Global Pool Output Section record
  2. One CSM Summary Output Section record
  3. Zero or one CSM Storage Owner Output Section record

Every response record sent by VTAM to the client looks 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)

SNA network monitoring NMI 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 included in SYS1.MACLIB. This data set must be available in the concatenation when compiling or assembling a part that makes use of these definitions. For an example of the mappings of the request and response data structures, see sample SNA network monitoring NMI mappings.

SNA network monitoring NMI 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 • CSM Statistics filters supported (1 bit) – 0 = Filters are not accepted on the CSM Statistics request – 1 = Filters are accepted on the CSM Statistics request • Reserved (5 bits): '00000'B
Reserved (15 bytes): 0

SNA network monitoring NMI 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

Enterprise Extender termination record format

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

SNA network monitoring NMI 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)

Enterprise Extender Summary IP Address Output Record

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)

SNA network monitoring NMI 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): 6
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)

Enterprise Extender connection-specific output record

Start of one or more Enterprise Extender connection associated RTP PU name sections (one instance per RTP PU that is using this EE connection)
Start of one or more Enterprise Extender connection health verification sections (one instance per route that is used by the EE connection, included only if EE health verification is available)
Start of Enterprise Extender connection health verification policy-based routing (PBR) section (one instance, included only if the EE connection is using PBR and it is available)

SNA network monitoring NMI 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
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)

HPR Connection Specific Output Record

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)

SNA network monitoring NMI 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 Section record that contains multiple CSM Global Buffer Pool Data data records (CSMPoolGData), one per pool • CSM Summary Output Section record that contains a single CSM Summary Data record (CSMSummGData) that represents CSM system-wide summary information • (optionally) CSM Storage Owner Output Section record that contains one or more CSM Storage Owner Data records (CSMStorOData), one per reported owner

CSM Global Pool Output Section 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

CSM Global Pool Output Section 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 CSM Global Buffer Pool Data records (CSMPoolGDdata), one per CSM pool

CSM Summary Output Section 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 Summary Data record (CSMSummGData), one single system wide record

CSM Storage Owner Output Section Record

Record Identifier (4 characters): CSMO
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 Storage Owner Data records (CSMStorOData), one per reported owner

NMI request errors

The following table describes the errors in an NMI request for which VTAM sends a termination record with the given return code and reason code, and then closes the connection.

Return Code	Reason Code	Meaning
EINVAL (121)	X'00007110'	Request header too short.
EINVAL (121)	X'00007111'	Unsupported version number in request header.
EINVAL (121)	X'00007112'	Triplet format is not valid; first request section is not contiguous to request header.
EINVAL (121)	X'00007112'	Triplet format is not valid; length of the filter element is insufficient for the request.
EINVAL (121)	X'00007113'	Length of request header plus length of request sections does not equal total length of request.

Return Code	Reason Code	Meaning
EINVAL (121)	X'00007114'	Eyecatcher in request header is not valid.

The following table describes the error in an NMI request for which VTAM returns a negative response of the same type as the request. VTAM leaves the connection active after returning the negative response for these errors.

Return Code	Reason Code	Meaning
EINVAL (121)	X'00007115'	Unrecognized request type.
EINVAL (121)	X'00007116'	Too many filter elements (request sections) included for request type.
EINVAL (121)	X'00007117'	Too few filter elements (request sections) included for request type.
EINVAL (121)	X'00007118'	Undefined filter parameter indicator set in filter element.
EINVAL (121)	X'00007119'	Required filter parameter missing from filter element.
EINVAL (121)	X'0000711A'	Unsupported filter parameter indicator set in filter element.
EINVAL (121)	X'0000711B'	Request not valid for HPR or EE information in a pure subarea VTAM node.
EINVAL (121)	X'0000711C'	Request not valid for HPR or EE information if VTAM was started with HPR=NONE start option.

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
- Monitor TCP/IP sysplex networking data
- Monitor TCP/IP stack profile statement settings
- Monitor TCP/IP stack interface and global statistics

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:

- "EZBNMIFR overview" on page 638
- "EZBNMIFR: Configuration and enablement" on page 639
- "Using the EZBNMIFR requests" on page 639
- "TCP/IP NMI request format" on page 642
- "TCP/IP NMI response format" on page 653

EZBNMIFR overview

You can invoke the EZBNMIFR interface to perform two types of requests: poll-type requests and action-type requests.

EZBNMIFR: Poll-type requests

For poll-type requests, EZBNMIFR is a callable interface that returns data related to the TCP/IP stack at a given point in time. In most cases, the caller can specify filters that limit the returned data to a specific set of information.

Poll-type requests enable you to obtain the following types 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
- Sysplex XCF data
- Dynamic VIPA addresses
- Dynamic VIPA port distribution
- Dynamic VIPA routes
- Dynamic VIPA connections
- TCP/IP profile statement settings
- Interface attributes and IP addresses
- Interface standard and extended statistics
- TCP/IP stack global statistics

EZBNMIFR: Action-type requests

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 DRop/**-D** command, which can be invoked from the TSO, z/OS UNIX, and MVS operator environments. 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 addresses and local port for UDP endpoints.

Tip: When a TCP connection or UDP endpoint is dropped, the associated socket is not closed. The application that owns the associated socket must close the socket.

The following action-type request tells the TCP/IP stack to perform an action:

- Drop TCP connections or UDP endpoints

EZBNMIFR: 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 DRop/**-D**, the user ID can use the EZBNMIFR callable interface to drop a TCP connection or UDP endpoint.

Using the EZBNMIFR requests

This material describes how to use EZBNMIFR requests with the TCP/IP stack or the TN3270E Telnet server.

EZBNMIFR 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 the execution key of the caller

EZBNMIFR format

Invoke EZBNMIFR, as follows.

For C/C++ callers:

```
NWMServices(JobName,
          RequestResponseBuffer,
          &RequestResponseBufferAlet,
          &RequestResponseBufferLength,
          &ReturnValue,
          &ReturnCode,
          &ReasonCode);
```

For assembler callers:

```
CALL EZBNMIFR,(JobName,
               RequestResponseBuffer,
               RequestResponseBufferAlet,
               RequestResponseBufferLength,
               ReturnValue,
               ReturnCode,
               ReasonCode)
```

EZBNMIFR parameters

JobName

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. If the buffer length is too short to contain all of the requested information, the request fails with the return code ENOBUFS. The length that is needed to contain all of the information is provided in the NWMHeader data structure of the response, in the NWMBytesNeeded field. If the buffer length is not the minimum size for the request, the request fails with the return code ENOBUFS, but the value that is needed is not provided in the NWMBytesNeeded field. The minimum size is the combined length of the NWMHeader data structure and any input filters.

ReturnValue
Returned parameter.

Type: Integer

Length: Fullword

The name of a fullword in which the EZBNMIFR service returns one of the following values:

- 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 NMI response format" on page 653 for additional details about 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	ENOBUFS	JRBuffTooSmall	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. If the buffer was large enough for some information to be returned, the NWMHeader NWMBytesNeeded field might contain the buffer size needed to return all of the requested information. See the description of the RequestResponseBufferLength parameter for an explanation of when the NWMBytesNeeded value is provided.

ReturnValue	ReturnCode	ReasonCode	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	EAGAIN	JRMustBeSysplex	The request was not successful. The target TCP/IP stack has not joined a sysplex.
-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 the 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 the 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 NMI 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. The request constant, which is specified in the NWMType field in the NWMHeader data structure, follows the request name. Some requests support filters. See "Filter request section" on page 646 for a description of each filter and the information about which filters are supported by each request.

- **GetConnectionDetail (NWMTcpConnType)**

  Use this request to obtain information about active TCP connections.

  Guideline: When you use filters with this request, you can experience a performance improvement in retrieving the connection details if every filter contains a 4-tuple (local address, local port, remote address and remote port) for a connection. Additional filter values can be specified in each filter along with the 4-tuple.

- **GetDVIPAConnRTab (NWMDvConnRTabType)**

  Use this request to obtain information about dynamic virtual IP addresses (DVIPA) connections. This call returns a list of IPv4 and IPv6 DVIPA TCP connections. Entries are returned for the following:

  – All DVIPA interfaces for which MOVEABLE IMMEDIATE or NONDISRUPTIVE was specified.
  – On a sysplex distributor routing stack, every connection that is being routed through this distributor.
  – On a stack taking over a DVIPA, every connection to the DVIPA.
  – On a sysplex distributor target stack or a stack that is in the process of giving up a DVIPA, every connection for which the stack is an endpoint.

  If none of the these apply, then an empty response buffer is returned with a successful reason value, return code, and reason code. If the invoked TCP/IP stack has not joined a sysplex, then return value -1, return code EAGAIN, and reason code JRMustBeSysplex are returned without any other data.

- **GetDVIPAList (NWMDvListType)**

  Use this request to obtain information about dynamic virtual IP addresses (DVIPAs). This request returns a list of all IPv4 and IPv6 DVIPAs for the invoked TCP/IP stack. For each DVIPA, the MVS system name, TCP/IP job name, and various status information are returned.

- **GetDVIPAPortDist (NWMDvPortDistType)**

  Use this request to obtain information about dynamic virtual IP address (DVIPA) port distribution. This request returns a list of IPv4 and IPv6 distributed DVIPAs and ports. For each distributed DVIPA and port pair, one or more entries are returned for each target TCP/IP stack. If the invoked TCP/IP stack has not joined a sysplex, then return value -1, return code EAGAIN, and reason code JRMustBeSysplex are returned without any other data. If the TCP/IP stack is not a distributing stack, then an empty response buffer is returned with a successful return value, return code, and reason code. If the same DVIPA and port pair are affected by more than one QOS Policy, then an entry with the same DVIPA and port is returned for each QOS policy.

- **GetDVIPARoute (NWMDvRouteType)**

  Use this request to obtain information about dynamic virtual IP address (DVIPA) routes. This request returns a list of information that is defined on VIPAROUTE profile statements. Each entry includes the dynamic XCF address of a target TCP/IP stack and the corresponding target IP address that is used to route connection requests to that TCP/IP stack. Output is returned only by a distributing TCP/IP stack, or by a backup TCP/IP stack for a distributed DVIPA when the backup TCP/IP stack is assuming ownership of the distributed

DVIPA. If the invoked TCP/IP stack has not joined a sysplex, then return value -1, return code EAGAIN, and reason code JRMustBeSysplex are returned without any other data. If the invoked TCP/IP stack is neither a distributing stack nor a backup stack, then an empty response buffer is returned with a successful return value, return code, and reason code.

- **GetGlobalStats (NWMGlobalStatsType)**

  Use this request to obtain TCP/IP stack global statistics for IP, ICMP, TCP, and UDP processing. The statistics that are returned by the request are similar to those in the output of the Netstat STATS/-S report. This request does not support filtering.

- **GetIfs (NWMIfsType)**

  Use this request to obtain TCP/IP stack interface attributes and IP addresses. The attributes and IP address information that are returned by the request are similar to those in the output of the Netstat DEVLINKS/-d and HOME/-h reports. Detailed attribute information is supported only for strategic interface types. The strategic interface types are:

  – Loopback
  – OSA-Express QDIO Ethernet
  – HiperSockets
  – Multipath Channel Point-to-Point
  – Static VIPA

  Dynamic VIPA interfaces are also strategic interfaces but their attributes can be obtained from the dynamic VIPA (DVIPA) NMI requests that are previously described in this topic. For non-strategic interface types, only the following information is provided:

  – Interface name from the LINK profile statement
  – Interface index
  – Associated device name from the DEVICE profile statement
  – Interface type
  – Interface status at the DEVICE and LINK level
  – Time stamp of last interface status change at the LINK level

  This request does not support filtering.

- **GetIfStats (NWMIfsStatsType)**

  Use this request to obtain TCP/IP stack interface statistics. The statistics that are returned by the request are similar to those in the output of the Netstat DEVLINKS/-d report with the addition of SNMP counters that are defined in the IF-MIB. For information about the IF-MIB, see RFC 2233. For information about how to access RFCs, see Appendix H, "Related protocol specifications," on page 991. Statistics are provided for all strategic interface types except for VIPA interfaces; the stack does not maintain counters for VIPA interfaces. This request also provides a time stamp of when the counters were last reset. This request does not support filtering.

- **GetIfStatsExtended (NWMIfsStatsExtType)**

  Use this request to obtain data link control (DLC) tuning statistics for datapath devices that are used by active OSA-Express QDIO Ethernet and HiperSockets interfaces. The statistics that are returned by the request are similar to those in the output of the VTAM TNSTATS function and the SMF type 50 record. Counters are provided for each read and write queue for each datapath device. Because of performance considerations, the counters are not maintained by default as part of TCP/IP stack initialization. The first GetIfStatsExtended

request causes the counters to be maintained for all active interfaces. Therefore, the read and write queue counters can be 0 in the response for the first request.

This request also provides a time stamp of when the counters were last reset. The counters are reset if all the interfaces that are using a datapath device are deactivated. This request does not support filtering.

- **GetProfile (NWMProfileType)**

  Use this request to obtain information about the current TCP/IP profile statement settings. This request does not support filtering. To detect changes to the profile statement settings, callers can use this callable request to obtain an initial set of current profile settings, and then do one of the following actions:

  – Repeat the request, over a time interval, comparing returned data from a previous response to the returned data from the last response.

  – Obtain the SMF Type 119 subtype 4 TCP/IP profile event records. These records provide information about changes to the profile settings that are made by using the VARY TCPIP,,OBEYFILE command processing. The records are created if they are requested on the SMFCONFIG or NETMONITOR profile statements. If the records are requested on the SMFCONFIG profile statement, they are written to the MVS SMF data sets. If the records are requested on the NETMONITOR profile statement, they can be obtained from the real-time SMF data network management interface (NMI). For more information about the real-time SMF NMI, see "Real-time TCP/IP network monitoring NMI" on page 564. For more information about the TCP/IP profile SMF record, see "TCP/IP profile event record (subtype 4)" on page 763. The SMF record might be created even if some errors occurred during the VARY TCIP,,OBEYFILE command processing. To determine whether profile changes actually occurred, application programs that process these records must compare the sections of changed information to the previous profile settings.

- **GetStorageStatistics (NWMStgStatsType)**

  Use this request to obtain information about TCP/IP storage utilization. This request does not support filtering.

- **GetSysplexXCF (NWMSyXcfType)**

  Use this request to obtain information about all TCP/IP stacks in the subplex. This request returns a list of all TCP/IP stacks in the same subplex as the invoked TCP/IP stack. For each TCP/IP stack, the MVS system name and one or more dynamic XCF IP addresses are returned. There are no filters defined for this request. If the invoked TCP/IP stack has not joined a sysplex, then return value -1, return code EAGAIN, and reason code JRMustBeSysplex are returned without any other data.

- **GetTCPListeners (NWMTcpListenType)**

  Use this request to obtain information about active TCP listeners.

- **GetTnMonitorGroups (NWMTnMonGrpType)**

  Use this request to obtain information about TN3270E Telnet server monitor groups.

- **GetTnConnectionData (NWMTnConnType)**

  Use this request to obtain information about TN3270E Telnet server connection performance data.

- **GetUDPTable (NWMUdpConnType)**

  Use this request to obtain information about active UDP sockets.

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.

Filter request section

For requests that support filters, you can use filters to limit the data that is returned to data that matches the specified filter values. Not all filters are supported for all requests.

The following request types do not support any filters. If you specify filters for these requests, the filters are ignored.

- GetGlobalStats
- GetIfs
- GetIfStats
- GetIfStatsExtended
- GetProfile
- GetStorageStatistics
- GetSysplexXCF
- GetTnMonitorGroups

The following table describes all possible filters.

Table 107. Available EZBNMIFR poll-type request filters

Filter item	Filter item value
Application data	An EBCDIC character string (right-padded with blanks if less 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 Appendix F, "Application data," on page 917 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 18, "Miscellaneous programming interfaces," on page 713 for more information about associating application data with a socket.
Application name	An EBCDIC application name (right-padded with blanks if less 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, the value 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.
ASID	A 16-bit address space number of a socket application address space.

Table 107. Available EZBNMIFR poll-type request filters (continued)

Filter item	Filter item value
Destination XCF IP address and family	A 32-bit IPv4 address or a 128-bit IPv6 address. The destination XCF IP address family field must also be set to indicate whether the destination XCF IP address filter value is an IPv4 address or an IPv6 address. For IPv4 addresses, the destination XCF IP address filter value can be specified as either an IPv4 address (for example, 9.1.2.3) or an IPv4-mapped IPv6 address (for example, ::FFFF:9.1.2.3). 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 (::). The destination XCF IP address family field must be set to AF_INET for an IPv4 address or AF_INET6 for an IPv6 address.
Destination XCF IP address prefix	A 16-bit signed binary value that specifies the number of destination XCF IP address bits to use. For example, the value 12 specifies that the first 12 bits of a destination XCF IP address are compared to the first 12 bits of the destination XCF IP address filter value. The value 0 specifies that all address bits are compared. A value greater than 32 for an IPv4 address, or greater than 128 for an IPv6 address, specifies that all address bits are compared.
Dynamic virtual IP address and family	A 32-bit IPv4 address or a 128-bit IPv6 address. The dynamic VIPA address family field must also be set to indicate whether the DVIPA filter value is an IPv4 address or an IPv6 address. For IPv4 addresses, the DVIPA filter value can be specified as either an IPv4 address (for example, 9.1.2.3) or an IPv4-mapped IPv6 address (for example, ::FFFF:9.1.2.3). 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 (::). The dynamic virtual IP address family field must be set to AF_INET for an IPv4 address or AF_INET6 for an IPv6 address.
Dynamic virtual IP address port	A 16-bit unsigned binary port number.
Dynamic virtual IP address prefix	A 16-bit signed binary value that specifies the number of dynamic virtual IP address bits to use. For example, the value 12 means that the first 12 bits of a dynamic VIPA are compared to the first 12 bits of the dynamic VIPA filter value. The value 0 means that all address bits are compared. A value greater than 32 for an IPv4 address, or greater than 128 for an IPv6 address, means that all address bits are compared.
Interface name	An EBCDIC interface name (right-padded with blanks if less than 16 characters in length) of an IPv4 or IPv6 interface. The interface name can have wildcard characters. Use a question mark (?) as a wildcard for a single character, and use an asterisk (*) as a wildcard for zero or more characters. For example, the value A?C* matches all interface 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 that have anything other than the character C in the third position.

Table 107. Available EZBNMIFR poll-type request filters (continued)

Filter item	Filter item value
Local or source IP address	A 32-bit IPv4 address or a 128-bit IPv6 address. The local or source 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 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 (::).
Local or source IP address prefix	A 16-bit signed binary value that specifies the number of local or source IP address bits to use. For example, the value 12 means that the first 12 bits of a local or source IP address are compared to the first 12 bits of the local IP address filter value. The value 0 means that all address bits are compared. A value greater than 32 for an IPv4 address, or greater than 128 for an IPv6 address, means that all address bits are compared.
Local or source 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, the value 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.
Remote or destination IP address	A 32-bit IPv4 address or a 128-bit IPv6 address. The remote or destination 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 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 (::).
Remote or destination IP address prefix	A 16-bit signed binary value specifying the number of remote or destination IP address bits to use. For example, the value 12 means that the first 12 bits of a remote or destination IP address are compared to the first 12 bits of the remote IP address filter value. The value 0 means that all address bits are compared. A value greater than 32 for an IPv4 address, or greater than 128 for an IPv6 address, means that all address bits are compared.

Table 107. Available EZBNMIFR poll-type request filters (continued)

Filter item	Filter item value
Resource ID	A 32-bit unsigned binary TCP/IP resource identifier (`Client ID` in Netstat displays).
Resource name	An EBCDIC job name, right-padded with blanks if less then 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, the value 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.
Remote or destination port	A 16-bit unsigned binary port number.
Server resource ID	A 32-bit unsigned binary TCP/IP resource identifier of the related server listening connection.
Target IP address and family	A 32-bit IPv4 address or a 128-bit IPv6 address. The target IP address family field must also be set to indicate whether the target IP address filter value is an IPv4 address or an IPv6 address. For IPv4 addresses, the destination XCF 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). 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 (::). The target IP address family field must be set to AF_INET for an IPv4 address or AF_INET6 for an IPv6 address.
Target IP address prefix	A 16-bit signed binary value that specifies the number of target IP address bits to use. For example, the value 12 means that the first 12 bits of a target IP address are compared to the first 12 bits of the target IP address filter value. The value 0 means that all address bits are compared. A value greater than 32 for an IPv4 address, or greater than 128 for an IPv6 address, means that all address bits are compared.

You can specify 1 – 4 filter elements. Each filter element can contain any combination of the items that are listed in Table 107 on page 646. A filter element that does not have any applicable items matches all data for the request. The data must match all items that are specified in a filter element to pass that filter check; data must pass at least one filter check to be selected.

If you do not specify any filters (triplet offset field is 0, or triplet element count field is 0, or triplet element length field is 0), then the caller is requesting all information that is applicable to that request.

The following list shows the applicable filter items that each request type supports. If you specify inapplicable filters for a particular request type, they are ignored.

- GetConnectionDetail
  - Application data
  - ASID
  - Local or source IP address
  - Local or source IP address prefix
  - Local or source port

  - Remote or destination IP address
  - Remote or destination IP address prefix
  - Remote or destination port
  - Resource ID
  - Resource name
  - Server resource ID
- GetDVIPAConnRTab
  - Destination XCF IP address and family
  - Destination XCF IP address prefix
  - Local or source IP address
  - Local or source IP address prefix
  - Local or source port
  - Remote or destination IP address
  - Remote or destination IP address prefix
  - Remote or destination port
- GetDVIPAList
  - Dynamic virtual IP address and family
  - Dynamic virtual IP address prefix
  - Interface name
- GetDVIPAPortDist
  - Destination XCF IP address and family
  - Destination XCF IP address prefix
  - Dynamic virtual IP address and family
  - Dynamic virtual IP address port
  - Dynamic virtual IP address prefix
- GetDVIPARoute
  - Destination XCF IP address and family
  - Destination XCF IP address prefix
  - Target IP address and family
  - Target IP address prefix
- GetTCPListeners
  - Application data
  - ASID
  - Local or source IP address
  - Local or source IP address prefix
  - Local or source port
  - Resource ID
  - Resource name
- GetTnConnectionData
  - Application name
  - Local or source IP address
  - Local or source IP address prefix
  - Local or source port
  - Lu name
  - Monitor group identifier

  - Remote or destination IP address
  - Remote or destination IP address prefix
  - Remote or destination port
  - Resource ID
  - Server resource ID
- GetUDPTable
  - ASID
  - Local or source IP address
  - Local or source IP address prefix
  - Local or source port
  - Resource ID
  - Resource name

Filter 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.

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 108. 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.
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 (::). The sockaddr length field value must be set to the correct length for the specified socket family. 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.
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 (::). The sockaddr length field value must be set to the correct length for the specified socket family. This desriptor 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 NMI response format" on page 653 for information about processing the result of a DropConnection request.

TCP/IP NMI response format

The following list describes the general format of the response:

- The response header, which is defined by the NWMHeader structure, 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 topics.
- The request sections.
- The response output. See the following topics about the poll-type and action-type response output for a description.

  Tip: Connection elements for TN3270E Telnet server connection performance data are returned only if the connection is being monitored by a MonitorGroup that is mapped to the connection. See Connection monitoring mapping statement in *z/OS Communications Server: IP Configuration Guide* for details.

Processing poll-type request responses:

The format of the response output depends on the specific request.

- The following requests return one or more response section elements of the same type.

Table 109. Poll-type request responses

Request	Response
GetConnectionDetail	NWMTCPConnEntry (assembler), NWMConnEntry (C/C++)
GetDVIPAConnRTab	NWMDvConnRTabEntry
GetDVIPAList	NWMDvListEntry
GetDVIPAPortDist	NWMDvPortDistEntry
GetDVIPARoute	NWMDvRouteEntry
GetIfStats	NWMIfStatsEntry
GetStorageStatistics	NWMStgStatEntry
GetSysplexXCF	NWMSyXcfEntry
GetTCPListeners	NWMTCPListenEntry
GetTnMonitorGroups	NWMTnMonGrpEntry
GetTnConnectionData	NWMTnConnEntry
GetUDPTable	NWMUDPConnEntry

- The following requests return one or more records. Each record begins with an NWMRecHeader structure that describes the record. See the specific request topics for a detailed description of the response output of each request.
  - GetGlobalStats
  - GetIfs
  - GetIfStatsExtended
  - GetProfile

The response output is described by the response section quadruplet in the NWMHeader structure. The quadruplet consists of the following fields:

- The offset, in bytes, of the first response section element or record. This offset is relative to the beginning of the response buffer.

- The number of elements in the response section or the number of records that are returned.
- The length of a response section element for requests that return one or more response section elements of the same type. For requests that return one or more records, the value of this field is 0. The NWMRecHeader structure for each returned record contains the actual length of each record.
- The total number of elements that passed the requested filter checks.

The response header contains 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.

GetGlobalStats response format:

For the GetGlobalStats request, the output is returned as one record. The response section quadruplet contains the following values:

- The offset, which is in the response buffer, of the output record.
- The length of each element. It is always 0.
- The number of elements in the response section. It is set to 1 to indicate that only one record was returned.
- The total number of matching elements. It is set to 1 because filters are not supported.

The output record consists of the following fields:

- Record header. The record header is mapped by the NWMRecHdr structure and consists of the following fields:
  - An EBCDIC identifier
  - The total length of the record
  - The number of section descriptors (mapped by the NWMTriple structure) that are present in this record
- Section descriptor triplets for each set of statistic counters. The returned statistic counters are similar to the counters in the output of the Netstat STATS/-S report. See Netstat STATS/-S report in *z/OS Communications Server: IP System Administrator's Commands* for a description of each field. The following section descriptors that describe the returned information for each section type are always included:
  - IP counters section - A section for IPv4 counters is always returned. If the TCP/IP stack is IPv6-enabled, a section for IPv6 counters is also returned.
  - IP general counters section - Only one section of this type is included.
  - TCP counters section - Only one section of this type is included.
  - UDP counters section - Only one section of this type is included.
  - ICMP global counters section - A section for IPv4 counters is always returned. If the TCP/IP stack is IPv6-enabled, a section for IPv6 counters is also returned.
  - ICMP type counters section - One section is returned for each ICMP and ICMPv6 type. For more information about these types, see http://www.iana.org/assignments/icmp-parameters and http://www.iana.org/assignments/icmpv6-parameters.
- IP counters sections (NWMIpStatsEntry) - Sections for IPv4 and IPv6 counters.
- IP general counters section (NWMIpGenStatsEntry)
- TCP counters section (NWMTcpStatsEntry)

- UDP counters section (NWMUdpStatsEntry)
- ICMP global counters sections (NWMIcmpStatsEntry)
- ICMP type counters sections (NWMIcmpTypeStatsEntry)

GetIfs response format:

For the GetIfs request, the output is returned as one record per interface. The response section quadruplet contains the following values:

- The offset, which is in the response buffer, of the first output record.
- The length of each element. It is always 0.
- The number of elements in the response section. It is set to the total number of records that are returned.
- The total number of matching elements. It is set to the number of records that are returned because filters are not supported.

All fields that contain EBCDIC values are padded with EBCDIC blanks (x'40') and are set to EBCDIC blanks if the field does not contain a value.

Each output record consists of the following fields:

- Record header. The record header is mapped by the NWMRecHdr structure and consists of the following fields:
  - An EBCDIC identifier
  - The total length of the record
  - The number of section descriptors (mapped by the NWMTriple structure) that are present in this record
- Section descriptor triplets. Two section descriptors that describe the returned information for each section type are always included:
  - Base section - Only one section of this type is included per interface.
  - IP address section - Only one section of this type is included for every IP address for the interface. If an interface does not have an IP address, the section descriptor triplet fields are all set to 0.
- Base section (NWMIfEntry). This section provides the interface name, status, and attributes.
- One or more IP address sections (NWMIpadEntry)

GetIfStatsExtended response format:

For the GetIfStatsExtended request, the data link control (DLC) tuning statistics output is returned as one record per data subchannel address that is used by an OSA-Express QDIO ethernet or HiperSockets interface. The response section quadruplet contains the following values:

- The offset, which is in the response buffer, of the first output record.
- The length of each element. It is always 0.
- The number of elements in the response section. It is set to the total number of records that are returned.
- The total number of matching elements. It is set to the number of records that are returned because filters are not supported.

All fields that contain EBCDIC values are padded with EBCDIC blanks (x'40').

Each output record consists of the following fields:

- Record header. The record header is mapped by the NWMRecHdr structure and consists of the following fields:
  - An EBCDIC identifier
  - The total length of the record
  - The number of section descriptors (mapped by the NWMTriple structure) that are present in this record
- Section descriptor triplets. Four section descriptors that describe the information that is returned for each section type are always included:
  - Base section - Only one section of this type is included per data subchannel address.
  - Interface section - One section of this type is included for each interface that shares the data subchannel address.
  - Read queue counters section - There is one of these sections per read queue supported for the data subchannel address. For more information about the OSA-Express read queues, see QDIO inbound workload queueing in *z/OS Communications Server: IP Configuration Guide*. HiperSockets interfaces only support one read queue.
  - Write queue counters section - One section of this type is included for each of one to four possible write priority queues that are supported for the data subchannel address.
- Base section (NWMIFStExtBaseEntry). This section provides information about the data, read control, write control subchannel addresses, the TRLE name, and OSA-Express ports. This section also includes a time stamp of when the counters were last reset.
- Interface section (NWMIFStExtIntfEntry)
- Read queue sections (NWMIfStExtReadEntry)
- Write queue sections (NWMIfStExtWriteEntry)

GetProfile response format:

For the GetProfile request, the output is returned as one record. The response section quadruplet contains the following values:

- Offset is the offset, into the response buffer, of a GetProfile record header.
- The length of each element is always 0.
- The number of elements in the response section is always 1 to indicate that only one record was returned.
- The total number of matching elements is always 1, because filters are not supported.

The record header is mapped by the NWMRecHdr structure. The header consists of the following fields:

- An EBCDIC identifier
- The total length of the record
- The number of section descriptors (triplets) that are present in this record. Twenty-one section descriptors are always returned. The section descriptor triplets are mapped by the NWMTriple structure.

The section descriptors (triplets) immediately follow the record header, and the sections immediately follow the section descriptors. If there is no profile information for a section, the section descriptor triplet fields for that section all contain 0.

The section structures in the GetProfile response are identical to the section structures in the TCP/IP profile SMF 119 subtype 4 event records. If you already have an application that processes the SMF record section structures, you can also use it for processing the GetProfile response section structures. See "TCP/IP profile event record (subtype 4)" on page 763 for a layout of this SMF record.

In the GetProfile response, the Profile Information Common and Data Set Name sections primarily contain information about the initial profile, not about the last change to the profile; however, the following fields contain the date and time of the last change to the profile:

- NMTP_PICOChangeTime
- NMTP_PICOChangeDate

Processing action-type request responses:

Processing the response for the DropConnection action-type request is described in this section.

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 nonnegative 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 nonnegative 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 110. 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.
ENOENT	JRGETCONNERR	The connection was not in the correct state for retrieving.
EAGAIN	JRUDPNOTUP	TCP/IP was not initialized

Table 110. Return code values (continued)

NWMDropConnRC	NWMDropConnRSN	Description
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 GetUDPTable 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.

TCP/IP NMI 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/C++ programs	Macros for assembler programs	Contents
EZBNMRHC	EZBNMRHA	The NMI request and response data structure definitions.
EZBNMMPC	EZBNMMPA	The GetProfile request data structure definitions for the sections of profile information in the response.

These header files and macros are included in the SEZANMAC data set and the header files are also included in the z/OS UNIX file system directory, /usr/include. When you compile or assemble a program in an MVS batch job, the SEZANMAC data set must be available in the MVS batch job concatenation. For an example of the mappings of the request and response data structures, see sample EZBNMIFR mappings.

TCP/IP NMI 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 example" on page 651):

```
 /*********************************************************************/
 /*                                                                   */
 /* 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 z/OS version V1R7 and later, the current version is 2. Applications coded with NWMVERSION1 (as in example 1) will have the version accepted in z/OS version V1R4 and later. However, applications coded with NWMCURRENTVER and compiled using the version 2 headers work only on z/OS version V1R7 and later releases. Applications using NWMCURRENTVER in z/OS version 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 NMIDropConnEntry                                       */
  /*                                                                   */
  /*********************************************************************/
  NMIbuffer.NMIDropConnEntry.NWMDropConnIdent=NWMDROPCONNIDENTIFIER;
  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.MWMDropConnEntry.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 example" on page 651):

```
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=@DYNSIZE,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,(R13),NMIalet,NMIlength,RV,RC,RSN)
         STORAGE RELEASE,LENGTH=@DYNSIZE,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
```

Network management 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 111. File storage locations

Interface	File name	Type	Library
Real-time TCP/IP packet and data trace NMI	EZBYTMIA (1)	MACRO	SEZANMAC
Real-time OSAENTA packet and data trace NMI	EZBYTMIH (1)	H	SEZANMAC and the z/OS UNIX /usr/include directory.
Packet and data trace formatting NMI	EZBCTAPI	MACRO	SEZANMAC
	EZBYPTO	MACRO	
	EZBYPTHA	MACRO	
	EZBCTHDR	MACRO	
	EZBYCTHH	H	SEZANMAC and the z/OS UNIX /usr/include directory.
	EZBYPTHH	H	
Real-time TCP connection SMF NMI	EZBYTMIA (1)	MACRO	SEZANMAC
	EZBYTMIH (1)	H	SEZANMAC and the z/OS UNIX /usr/include directory.
	EZASMF77	MACRO	SYS1.MACLIB
	EZASMF (2)	H	SEZANMAC and the z/OS UNIX /usr/include directory.
Allow applications to obtain SMF records for FTP, IPSec and CSSMTP	EZBYTMIA (1)	MACRO	SEZANMAC
	EZBYTMIH (1)	H	SEZANMAC
	EZASMF77	MACRO	SYS1.MACLIB
	EZASMF (2)	H	SEZANMAC and the z/OS UNIX /usr/include directory.
	EZANMFTA	MACRO	
	EZANMFTC	H	SEZANMAC
Real-time SMF NMI	EZASMF77	MACRO	SYS1.MACLIB
	EZASMF (2)	H	SEZANMAC and the z/OS UNIX /usr/include directory.

Table 111. File storage locations (continued)

Interface	File name	Type	Library
TCP/IP callable NMI	EZBNMRHA	MACRO	SEZANMAC
	EZBNMRHC (2)	H	SEZANMAC and the z/OS UNIX /usr/include directory.
	EZBNMMPA	MACRO	SEZANMAC
	EZBNMMPC (2)	H	SEZANMAC and the z/OS UNIX /usr/include directory.
Resolver callable NMI	EZBRENMA	MACRO	SEZANMAC
	EZBRENMC	H	SEZANMAC and the z/OS UNIX /usr/include directory
SNA network monitoring NMI	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	SEZANMAC and the z/OS UNIX /usr/include directory.
	EZANMFTA	MACRO	SEZANMAC
	EZANMFTC	H	
Local IPSec NMI	EZBNMSEA	MACRO	SEZANMAC
	EZBNMSEC	H	SEZANMAC and the z/OS UNIX /usr/include directory.
	EZBNMIV2	H	SEZANMAC and the z/OS UNIX /usr/include directory.

Table 111. File storage locations (continued)

Interface	File name	Type	Library
Network security services (NSS) NMI	EZBNMSEA	MACRO	SEZANMAC
	EZBNMSEC	H	SEZANMAC and the z/OS UNIX /usr/include directory.
	EZBNMIV2	H	SEZANMAC and the z/OS UNIX /usr/include directory.
(1) Part used for multiple functions. (2) These parts require the XL C/C++ Run-Time functions, macros, and header files.			

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 that requires a secure connection, AT-TLS provides encryption and decryption of data based on policy statements that are 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 *z/OS Communications Server: IP Configuration Guide* and the Policy Agent information in *z/OS Communications Server: IP Configuration Reference*.

Most applications do not need any awareness of the security negotiations and encryption that is 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 that are 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 112. 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 112. 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 SIOCTTLSCTL ioctl blocks during the initial handshake if the socket is in blocking mode. If the socket is non-blocking, SIOCTTLSCTL 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 SIOCTTLSCTL 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 *z/OS Communications Server: IP CICS Sockets Guide* for more information on configuring TCP/IP CICS socket support.

Using the SIOCTTLSCTL ioctl

An application uses the SIOCTTLSCTL 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 SIOCTTLSCTL 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 SIOCTTLSCTL 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 SIOCTTLSCTL 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 SIOCTTLSCTL ioctl with option TTLS_QUERY_ONLY. If a blocking socket is used, the final status is contained in the returned SIOCTTLSCTL. Ensure that your server application checks the SIOCTTLSCTL 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 nonzero.

Stopping AT-TLS on a connection

Use the SIOCTTLSCTL 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 SIOCTTLSCTL 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 SIOCTTLSCTL 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 SIOCTTLSCTL 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 SIOCTTLSCTL 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 SIOCTTLSCTL ioctl" on page 673 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 SIOCTTLSCTL 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 SIOCTTLSCTL 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 TTLSK_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 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 673, and Figure 34 on page 673 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.

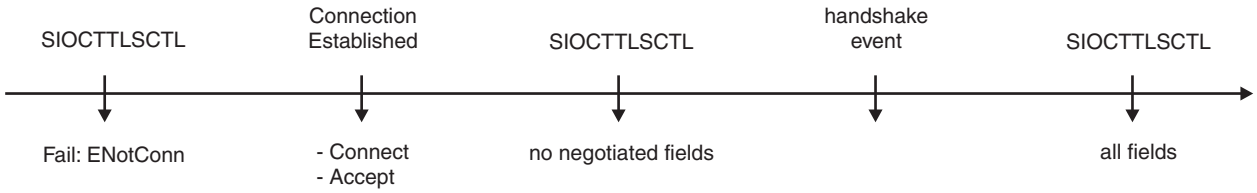


Figure 32. SIOCTTLSCTL with TTLS_Query_Only

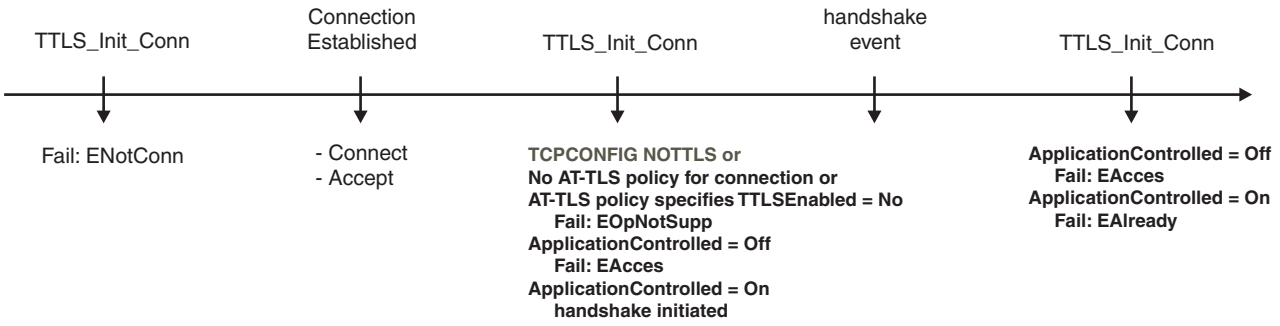


Figure 33. SIOCTTLSCTL with TTLS_Init_Connection

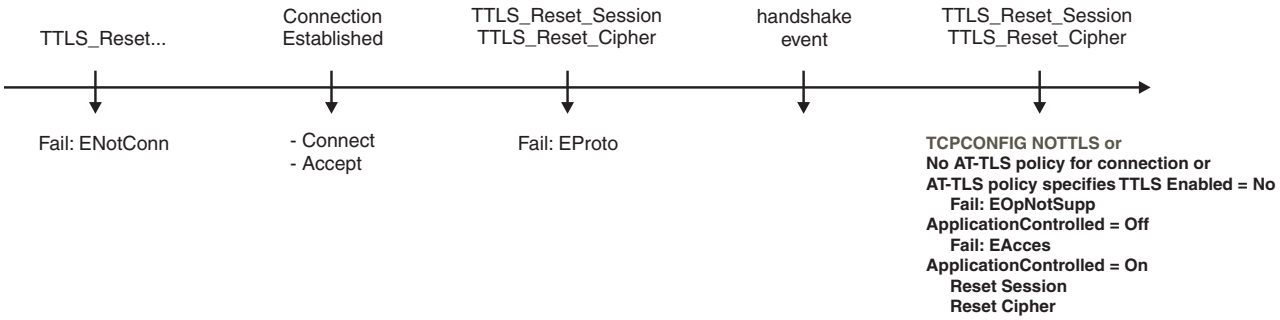


Figure 34. SIOCTTLSCTL with TTLS_Reset_Session or TTLS_Reset_Cipher

Coding the SIOCTTLSCTL ioctl

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 *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 *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for coding BPX1IOC or BPX4IOC.

PL/I Include EZBZTLS1 mapping. See *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 EZBZTLSB mapping. See *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 *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 *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 SIOCTTLSCTL Ioctl requests.
- Any field not used must be set to 0.
- If the additional buffer is required for the SIOCTTLSCTL 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 *ENOBUFS* with the required buffer size specified in *TTLSi_Cert_Len*.

SIOCTTLSCTL (X'C038D90B')

The SIOCTTLSCTL ioctl provides an interface for the application to query and control AT-TLS for the connection. The following data items are returned by theSIOCTTLSCTL 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 that is 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 1.0 is in use.

TTLS_PROT_TLSV1_1 (0x0302)
TLS version 1.1 is in use.

TTLSi_FIPS140
Indicates whether Federal Information Processing Standard (FIPS) 140 is in effect. Valid values are:

TTLS_FIPS140_OFF (0x00)
FIPS 140 support is not in effect.

TTLS_FIPS140_ON (0x01)
FIPS 140 support is in effect.

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 Services System SSL 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 *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 SIOCTTLSCTL 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 ENOBUFS and the required buffer size is returned in *TTLSi_Cert_Len*.

The following optional behaviors can be requested on the SIOCTTLSCTL 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 TTLSHeader 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 is 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 nonzero value. This option is valid only if the HandshakeRole value is Server or ServerWithClientAuth. Any non-SSL data received ends the handshake request and leaves the connection state nonsecure.

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 Verion 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 SIOCTTLSCTL returns.

Rule: Set request quadruplets must immediately follow the TTLSHeader 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 113. TTLSHeader control block structure

TTLSHeader fixed section
TTLSHeader optional Set request quadruplets
TTLSHeader optional Get request quadruplets
Beginning of set data area
Beginning of get data area

TTLSHeader fixed section: The TTLSHeader fixed section contains control information that describes the Set and Get requests that follow. The fixed section is defined by the TTLSHeader structure as shown in Table 114.

Table 114. TTLSHeader fixed section

Field	Offset	Length	Format	Description
TTLSHeaderIdent	0	8 bytes	EBCDIC	TTLSHeader identifier; set to TTLSHdr_Ident(EBCDIC 'TTLSHDR ')
TTLSHdrRsvd1	8	8 bytes	Binary	Reserved; set to 0
TTLSHdr_BytesNeeded	16	4 bytes	Binary	Length of the buffer needed to contain TTLSHeader, TTLSQuadruplet 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
TTLSHdr_SetCount	20	4 bytes	Binary	Number of Set requests in the buffer. Each Set request is represented by a TTLSQuadruplet
TTLSHdr_GetCount	24	4 bytes	Binary	Number of Get requests in the buffer. Each Get request is represented by a TTLSQuadruplet
TTLSHdrRsvd2	28	16 bytes	Binary	Reserved; set to 0

TTLSHeader 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 TTLSQuadruplet structure as shown in Table 115.

Table 115. TTLSQuadruplet structure

Field	Offset	Length	Format	Description
TTLSQ_Key	0	4 bytes	Binary	Constant identifying the request
TTLSQ_Offset	4	4 bytes	Binary	Offset to the first value for the request, measured in bytes from the start of the TTLSHeader 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.
TTLSQ_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.
TTLSQ_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 116. Get request structure

Identifier	Constant	Length	Format	Description
TTLSK_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 TTLSQ_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 TTLSQ_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.

Table 116. Get request structure (continued)

Identifier	Constant	Length	Format	Description
TTLSK_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 TTLSQ_Rcode: **0** The request completed successfully.
TTLSK_TTLSRule_Name	4002	48 bytes, ending with a null character	EBCDIC	Returns the name of the TTLSRule field that is mapped to the connection. The following value can be returned in TTLSQ_Rcode: **0** The request completed successfully.
TTLSK_GroupAction_Name	4003	48 bytes, ending with a null character	EBCDIC	Returns the name of the TTLSGroupAction field that is mapped to the connection. The following value can be returned in TTLSQ_Rcode: **0** The request completed successfully.
TTLSK_EnvironmentAction_Name	4004	48 bytes, ending with a null character	EBCDIC	Returns the name of the TTLSEnvironmentAction field that is mapped to the connection. The following value can be returned in TTLSQ_Rcode: **0** The request completed successfully.
TTLSK_ConnectionAction_Name	4005	48 bytes, ending with a null character	EBCDIC	Returns the name of the TTLSConnectionAction field that is mapped to the connection. The following value can be returned in TTLSQ_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 TTLSHeader structure, pointed to by the TTLSI_BufPtr pointer on the SIOCTTLSCTL ioctl request to validate the server's certificate against these host names:

Table 117. Example TTLSHeader structure

TTLSHeader			
Field	**Offset**	**Format**	**Value**
TTLSHeaderIdent	0	EBCDIC	TTLSHDR

Table 117. Example TTLSHeader structure (continued)

TTLSHeader			
Field	**Offset**	**Format**	**Value**
TTLSHdrRsvd1	8	Binary	00000000 00000000
TTLSHdr_BytesNeeded	16	Binary	00000000
TTLSHdr_SetCount	20	Binary	00000000
TTLSHdr_GetCount	24	Binary	00000002
TTLSHdrRsvd2	28	Binary	00000000 00000000 00000000 00000000
TTLSQuadruplet			**Get Request buffer**
TTLSQ_Key	48	Binary	00000FA0
TTLSQ_Offset	52	Binary	00000050
TTLSQ_Length	56	Binary	00000000
TTLSQ_Rcode	60	Binary	00000000
TTLSQuadruplet			**Get Request buffer**
TTLSQ_Key	64	Binary	00000FA0
TTLSQ_Offset	68	Binary	0000006B
TTLSQ_Length	72	Binary	00000000
TTLSQ_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 TTLSHeader structure would be returned to the application:

Table 118. Example returned TTLSHeader structure

TTLSHeader			
Field	**Offset**	**Format**	**Value**
TTLSHeaderIdent	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
TTLSQuadruplet			**Get Request buffer**
TTLSQ_Key	48	Binary	00000FA0
TTLSQ_Offset	52	Binary	00000050
TTLSQ_Length	56	Binary	00000000
TTLSQ_Rcode	60	Binary	00000001
TTLSQuadruplet			**Get Request buffer**
TTLSQ_Key	64	Binary	00000FA0
TTLSQ_Offset	68	Binary	0000006B
TTLSQ_Length	72	Binary	00000001

Table 118. Example returned TTLSHeader structure (continued)

TTLSHeader			
Field	**Offset**	**Format**	**Value**
TTLSQ_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

SIOCTTLSCTL ioctl 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 TTLSEnabled=No.

EAcces
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 .

SIOCTTLSCTL ioctl coding examples

The following examples show sample code for building and issuing the SIOCTTLSCTL ioctl.

SIOCTTLSCTL ioctl assembler example

The following sample assembler code builds an SIOCTTLSCTL 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 SIOCTTLSCTL 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                  X
               S=SOCDESCA,                                           X
               COMMAND='SIOCTTLSCTL',                                X
               REQARG=TTLS_IOCTL,                                    X
               RETARG=TTLS_IOCTL,                                    X
               REQAREA=REQAREA,   IN CASE WE ARE DOING EXITS OR ECBS X
               ERRNO=ERRNO,       RETURN ERRNO HERE                  X
               RETCODE=RETCODE,   RETURN RETCODE HERE                X
               ERROR=ERROR        ABEND IF MACRO ERROR
*
...

BUFFERA  DS    CL1024
..
         EZBZTLSP DSECT=NO        TTLS ioctl structure
...
```

SIOCTTLSCTL ioctl PL/I example

The following sample PL/I code builds and issues an SSIOCTTLSCTL ioctl that requests secure connection initialization and the return of the partner's certificate in the provided buffer.

```
...
 /* get the SIOCTTLSCTL ioctl mapping and constants  */
 % include EZBZTLS1;
...
 /* area to return the certificate data if available */
 DCL CERTIF  CHAR(1000) INIT(''B);
....
```

```
/* allocate storage for the SIOCTTLSCTL 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 ezasoket(IOCTL,
               SOCKET,
               SIOCTTLSCTL,   /* 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;
```

SIOCTTLSCTL ioctl COBOL example

The following sample COBOL code builds and issues an SIOCTTLSCTL ioctl that requests the initialization of the secure connection.

```
...
      *===============*
       Data Division.
      *===============*
      *----------------------------------------------------------------*
      * Variables used by the SIOCTTLSCTL IOCTL call                   *
      *----------------------------------------------------------------*
       01  ttls-ioctl-data.
           COPY EZBZTLSB.
...
      *================================================*
       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 'EZASOKET' using soket-ioctl socket-descriptor-new
               SIOCTTLSCTL
               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.
```

SIOCTTLSCTL ioctl C example

The following sample C code builds and issues an SIOCTTLSCTL ioctl that requests initialization of the secure connection and the return of the partner's certificate in the provided buffer.

```
...
#include "ezbztlsc.h"               /* SIOCTTLSCTL ioctl             */
...
  struct TTLS_IOCTL ioc;            /* ioctl data structure          */
  char buff[1000];                  /* buffer for certificate        */
...
  /* issue the SIOCTTLSCTL 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,SIOCTTLSCTL,(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. Trusted TCP connections

z/OS TCP/IP stacks within a sysplex or a subplex communicate using the cross-system coupling facility (XCF). You can use XCF to exchange security information between application endpoints, which creates a trusted TCP connection. For more information about TCP/IP in a sysplex, see *z/OS Communications Server: IP Configuration Guide*.

An application end point can retrieve either or both of the following kinds of information for a partner end point:

- Sysplex-specific connection routing information

  An application can retrieve connection routing information by invoking the SO_CLUSTERCONNTYPE socket option or the SIOCGPARTNERINFO ioctl call. For more information, see "Sysplex-specific connection routing information."
- Partner security credentials

  An application can retrieve partner security credentials by invoking the SIOCGPARTNERINFO ioctl call, optionally preceded by the SIOCSPARTNERINFO ioctl call. Partner security credentials can include the partner user ID, partner user security token (UTOKEN), or both. For more information, see "Partner security credentials" on page 690.

Sysplex-specific connection routing information

When all of the TCP/IP stacks in a sysplex are initialized and in a steady state, they exchange information within the sysplex, such that each stack recognizes all of the IP addresses that are 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 to all other stacks. For any TCP connections, a stack can determine from the partner IP address whether the stack that supports the partner application is part of the same sysplex, and whether the stack resides in the same MVS image as the local stack.

Requirement: A TCP socket connection is required for an application to retrieve connection routing information.

An application can use the SO_CLUSTERCONNTYPE socket option or the SIOCGPARTNERINFO ioctl to obtain sysplex-specific connection routing information for a sockets application, which might enable the application to offer better function, performance, and scalability. The application collects and reports this information only when you specifically request it, so that an application that does not need the information does not incur the expense. The SO_CLUSTERCONNTYPE socket option and the SIOCGPARTNERINFO ioctl perform similarly when the sockets application is the listening (server) application or the initiating (client) application.

The SO_CLUSTERCONNTYPE socket option and the SIOCGPARTNERINFO ioctl return indicators to a sockets application when a connection is established. Table 119 on page 688 lists the indicators and the potential benefits that TCP sockets applications can gain from this information about a partner.

Table 119. Indicators and potential benefits of connection routing information

Partner indicator	Potential benefit
Same MVS image	Partners can share memory information that is costly to generate (for example, security contexts).
Same sysplex	Parameter conversions can be avoided because both sides of the connection use the same machine architectures and data representation (z/OS).
Internal	Application security might cost less (for example, applications might not encrypt data).

The internal indicator is returned only when the partner is part of the same sysplex, the data flows to the partner are over a link or interface that is never exposed outside of the sysplex, and the link or interface is one of the following types:

- CTC
- HiperSockets interface (iQDIO)
- MPCPTP (including XCF and IUTSAMEH)
- OSA-Express QDIO with CHPID type OSX or OSM
- Loopback
- Both connection partners are owned by the same multihomed stack

An application can use the internal indicator, for example, to avoid the cost of encrypting and decrypting data. If an application establishes a connection for which SSL is the appropriate protection mechanism when the partner is 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 implement the following logic.

Immediately after connection setup, but before the SSL handshake is initiated, use the SO_CLUSTERCONNTYPE socket option or the SIOCGPARTNERINFO ioctl to retrieve the connection routing information.

- If the internal indicator is not returned, initiate the SSL handshake with the appropriate levels of encryption specified (negotiated) between the two connection end points.
- If the internal indicator is returned and the partner security credentials are available (see "Partner security credentials" on page 690 for more information), retrieve the partner security information by using the SIOCGPARTNERINFO ioctl and optionally refrain from using SSL to protect the data.
- If the internal indicator is returned, but the partner security credentials are not available, initiate the SSL handshake as usual to authenticate the partner and retrieve the user ID that is associated with the digital certificate of the partner. After the internal indicator was returned, optionally specify only null encryption as an encryption choice. Because support for null encryption is a required feature of SSL and the SSL handshake is not destined to fail for architectural (IETF RFC) reasons. Then the partner determines whether a negotiated null encryption is acceptable to the partner or the connection should be closed.

Although the expensive SSL is not avoided, you can avoid encryption and decryption of the data that is exchanged between the partners, as appropriate. If

the applications are doing bulk data transfer, and normal encryption is triple-DES, the savings in CPU cycles can be considerable.

Results:

- The internal indicator is not returned if the destination IP address for a connection (the partner's IP address) is a dynamic VIPA (DVIPA) or distributed DVIPA residing in the sysplex, because traffic for these connections can be forwarded to the target TCP/IP stacks over links or interfaces that are external to the sysplex.
- If the TCP/IP stacks within a sysplex have been partitioned into subplexes, they do not appear to each other to be in the same image or sysplex. Results from the SO_CLUSTERCONNTYPE socket option or the SIOCGPARTNERINFO ioctl are impacted in the following ways:
  - Same sysplex indicator

    If two stacks in the same sysplex belong to different TCP/IP subplexes, this indicator is not set. Communication between these two stacks must cross a network interface, as opposed to using an XCF connection.
  - Same MVS image indicator

    If two stacks on the same MVS system belong to different TCP/IP subplexes, this indicator is not set.
  - Internal indicator

    If two TCP/IP stacks are in separate subplexes, this indicator is not set for connections that are using CTC, MPCPTP (including XCF and IUTSAMEH), OSA-Express QDIO with CHPID type OSX, or the HiperSockets function. Each stack is not aware of the IP addresses for the partner stack because the stacks belong to different subplexes.

    For connections that use loopback, OSA-Express QDIO with CHPID type OSM, or one of the local interfaces on a stack, this indicator is set.

Steps for retrieving connection routing information

This topic describes how to retrieve sysplex-specific connection routing information to a socket application.

Before you begin: A TCP socket connection in a sysplex environment is required. For information about TCP/IP in a sysplex, see *z/OS Communications Server: IP Configuration Guide*.

Perform the following steps to retrieve connection routing information:

1. After the connect call in the client application, issue the SO_CLUSTERCONNTYPE socket option or the SIOCGPARTNERINFO ioctl.
2. After the accept call in the server application, issue the SO_CLUSTERCONNTYPE socket option or the SIOCGPARTNERINFO ioctl.

For more information about the SO_CLUSTERCONNTYPE socket option, see “Coding the SO_CLUSTERCONNTYPE socket option” on page 693. For more information about the SIOCGPARTNERINFO ioctl, see “SIOCGPARTNERINFO (X'C000F612')” on page 696.

Partner security credentials

Applications in a sysplex can exchange security information over a TCP sockets connection to establish a trusted TCP connection between the applications. z/OS sockets partners can use the retrieved security information to perform access control checks on a TCP connection.

Use the SIOCGPARTNERINFO ioctl to enable an application to retrieve the security credentials of a partner. Partner security credentials can include the partner user ID, partner user security token (UTOKEN), or both. For information about what is provided in the UTOKEN by the ICHRUTKN macro, see *z/OS Security Server RACF Data Areas*.

You can also issue the SIOCSPARTNERINFO ioctl, before you issue the SIOCGPARTNERINFO ioctl, to avoid suspending an application on the SIOCGPARTNERINFO ioctl. The SIOCSPARTNERINFO ioctl sets an indicator to retrieve the partner security credentials during connection setup and saves the information. You must issue the SIOCSPARTNERINFO ioctl first to allow an application to issue a SIOCGPARTNERINFO ioctl without suspending the application, or at least to minimize the time needed to retrieve the information. If you do not issue the SIOCSPARTNERINFO ioctl, the partner security credentials are not retrieved until you issue the SIOCGPARTNERINFO ioctl, which requires a suspension to retrieve the data.

Steps for retrieving partner security credentials

This topic describes how to retrieve partner security credentials to create a trusted TCP connection.

Before you begin:

- A TCP socket connection in a sysplex environment is required. For information about TCP/IP in a sysplex, see *z/OS Communications Server: IP Configuration Guide*.
- You need to determine whether your application is APF authorized or is authorized to run in supervisor state, or you need to know which users run the application to retrieve partner security credentials.
- You need to decide on a common security domain name within your sysplex or subplex.
- You need to determine whether your application can be suspended when you are retrieving partner security credentials.

Perform the following steps to retrieve partner security credentials:

1. Set up proper authorization for your application using one of the following methods:
   - Set up your application so that it is APF authorized or is authorized to run in supervisor state.
   - Provide access to specific users by defining security product authority in the SERVAUTH class for the following profile:

     `EZB.IOCTL.`*`sysname`*`.`*`tcpprocname`*`.PARTNERINFO`

     The *sysname* value is the system name that is defined in the sysplex, and the *tcpprocname* value is the TCP/IP procedure name.

     Tip: You can specify a wildcard on segments of the profile name.

Requirement: Grant at least READ access to this profile to permit a user to retrieve partner security credentials.

2. Define security product authority for the profile EZBDOMAIN in the SERVAUTH class within the sysplex that is to use trusted TCP connections. Specify the same security domain name in the APPLDATA field.

   ```
   RDEFINE SERVAUTH EZBDOMAIN APPLDATA('security_domain_name')
   ```

   Rules:

   - The security domain name is limited to 255 characters.
   - The security domain name is not case sensitive.

   Tip: The security domain name is not required when you are using the SIOCGPARTNERINFO ioctl to retrieve information from a partner on the same stack.

   Results:

   - If the security domain name is not defined or does not match, then the request fails and the partner security credentials are not returned.
   - Verification of the security domain name occurs only the first time that partner security credentials are retrieved by the SIOCGPARTNERINFO or SIOCSPARTNERINFO ioctl in each connection.

3. Code the appropriate ioctl calls for the client and server applications.

   - For the client application:
     a. Optionally, issue the SIOCSPARTNERINFO ioctl before the connect call to avoid suspending your application while the partner security credentials are being retrieved.
     b. Issue the SIOCGPARTNERINFO ioctl after the connect call. Optionally, when you are using the SIOCSPARTNERINFO ioctl, specify the PI_Timeout value 0 on the SIOCGPARTNERINFO ioctl to indicate that your application cannot be suspended while the partner security credentials are being retrieved.
   - For the server application:
     a. Optionally, issue the SIOCSPARTNERINFO ioctl before the listen call to avoid suspending your application while the partner security credentials are being retrieved.
     b. Issue the SIOCGPARTNERINFO ioctl after the accept call. Optionally, when you are using the SIOCSPARTNERINFO ioctl, specify the PI_Timeout value 0 on the SIOCGPARTNERINFO ioctl to indicate that your application cannot be suspended while the partner security credentials are being retrieved.

   Issue the SIOCSPARTNERINFO ioctl with the value PI_REQTYPE_SET_PARTNERDATA. For more information about the SIOCSPARTNERINFO ioctl, see “SIOCSPARTNERINFO (X'8004F613')” on page 694.

   You can issue the SIOCGPARTNERINFO ioctl with the PI_Reqtype value set to PI_REQTYPE_PARTNER_USERID, PI_REQTYPE_PARTNER_UTOKEN, or both, to retrieve the partner user ID, partner user security token (UTOKEN), or both. For more information about the SIOCGPARTNERINFO ioctl, see “SIOCGPARTNERINFO (X'C000F612')” on page 696. For information about what is provided in the UTOKEN by the ICHRUTKN macro, see *z/OS Security Server RACF Data Areas*.

Programming requirements for the SO_CLUSTERCONNTYPE socket option

You can use the following APIs to retrieve connection routing information using the SO_CLUSTERCONNTYPE socket option:

- UNIX System Services Assembler Callable Service (BPX1IOC or BPX4IOC)
- TCPIP C socket API [getsockopt()]

Restriction: The following APIs are not supported for the SO_CLUSTERCONNTYPE socket option:

- Language Environment C/C++ socket
- X/Open Transport Interface (XTI)
- Pascal API
- Macro API (EZASMI)
- CALL instruction API (EZASOKET) supporting COBOL, PL/I, and System/370 assembler languages
- REXX socket API
- CICS C socket calls
- CICS CALL instruction API (EZASOKET - by including EZACICAL or EZACICSO)
- IMS CALL instruction API (EZASOKET)

Programming requirements for the SIOCGPARTNERINFO and SIOCSPARTNERINFO ioctl calls

You can use the following APIs to retrieve connection routing information, partner security credentials, or both by using the SIOCGPARTNERINFO ioctl (and optionally by using the SIOCSPARTNERINFO ioctl prior to the SIOCGPARTNERINFO ioctl):

- Language Environment C/C++ socket:
  - SIOCGPARTNERINFO call [w_ioctl()]
  - SIOCSPARTNERINFO call [ioctl()]
- UNIX System Services Assembler Callable Service (BPX1IOC or BPX4IOC)
- Trusted TCP connections API for Java

  Install the 31-bit or 64-bit software development kit (SDK) for Java, Java 2 Technology Edition, V5 or later.
- Macro API (EZASMI)
- CALL instruction API (EZASOKET) supporting COBOL, PL/I, and System/370 assembler languages
- REXX socket API

  Restriction: For the SIOCGPARTNERINFO ioctl, this API supports only the request types PI_REQTYPE_CONNTYPE and PI_REQTYPE_PARTNER_USERID. For more information about the SIOCGPARTNERINFO ioctl, see "SIOCGPARTNERINFO (X'C000F612')" on page 696.
- CICS C socket calls
- CICS CALL instruction API (EZASOKET - by including EZACICAL or EZACICSO)
- IMS CALL instruction API (EZASOKET)

Restriction: The following APIs are not supported for the SIOCGPARTNERINFO and SIOCSPARTNERINFO calls:

- TCPIP C socket API
- X/Open Transport Interface (XTI)
- Pascal API

Table 120 describes the programming requirements for trusted TCP connection APIs.

Table 120. Programming requirements for trusted TCP connection APIs

Function	Requirement
Authorization	Supervisor state or problem state, any PSW key. For special authorization requirements when retrieving partner security credentials, see "Steps for retrieving partner security credentials" on page 690.
Dispatchable unit mode	Task.
SRB mode	Some APIs can be invoked in SRB mode. All APIs can be invoked in TCB mode.
Cross-memory mode	The API can be invoked in only 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 you issue these calls.
Functional recovery routine (FRR)	Do not invoke the API with an FRR set, which can cause system recovery routines to be bypassed and severely damage the system.
Storage	Storage acquired for the purpose of containing data that is returned from an API call must be obtained in the same key as that of the application program status word (PSW) at the time of the call.
Nested API calls	You cannot issue nested SIOCGPARTNERINFO ioctl calls when you are requesting partner security credentials.
Addressability mode (AMODE)	ALL APIs can be invoked in 31-bit addressability mode. Unix System Services assembler callable service, Language Environment C/C++ socket, and JAVA can be invoked in 64-bit addressability mode.

Coding the SO_CLUSTERCONNTYPE socket option

For all supported programming languages, the SO_CLUSTERCONNTYPE socket option returns a 32-bit value that is associated with a socket. For more information and for general coding guidelines for the SO_CLUSTERCONNTYPE socket option, see the following sources:

- *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*
  - C socket API
- *z/OS UNIX System Services Programming: Assembler Callable Services Reference*
  - Callable services descriptions (BPX1IOC or BPX4IOC)

Coding the SIOCSPARTNERINFO and SIOCGPARTNERINFO ioctl calls

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 Java API classes are installed in the directory /usr/include/java_classes, and the dynamic link library (DLL) files are installed in the directory /usr/lib. The following programming languages are supported:

Assembler Include the EZBPINFA mapping.

C Include the EZBPINFC header file.

Java Include the EZBTrustedPartner.jar file.

PL/I Include the EZBPINF1 mapping.

COBOL Include the EZBPINFB mapping.

REXX No mapping or header file is used.

For general coding guidelines for the sockets application ioctl calls, see the following sources:

- *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*
  - Callable services descriptions (BPX1IOC or BPX4IOC)
- *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*
  - C socket calls (EZACIC07 or EZACIC17, which calls EZASOKET with entry in EZACICAL)
  - 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*
  - CALL instruction API (EZASOKET) supporting COBOL, PL/I, and System/370 assembler languages
- Javadoc information that is included in the EZBTrustedPartnerdoc.jar file, which is installed in the directory /usr/include/java_classes

  Download the jar file to a workstation, unpack it, and read it in a web browser. The trusted TCP connections API for Java includes the ConnectionType and PartnerInfo classes.

  Optionally for logging, see the information for the java.util.logging package; the trusted TCP connections API for Java uses the Java logging API (java.util.logging.Logger) to generate debug information.

SIOCSPARTNERINFO (X'8004F613')

The SIOCSPARTNERINFO ioctl sets an indicator to retrieve the partner security credentials during connection setup and saves the information. You must issue the SIOCSPARTNERINFO ioctl first to allow an application to issue a

SIOCGPARTNERINFO ioctl without suspending the application, or at least to minimize the time needed to retrieve the information.

Result in a common INET environment: In a common INET (CINET) environment, which enables multiple TCP/IP stacks to run in a single logical partition (LPAR), the SIOCSPARTNERINFO ioctl is sent to all active stacks.

Input to SIOCSPARTNERINFO

The following input item is accepted by the SIOCSPARTNERINFO ioctl:

PI_REQTYPE_SET_PARTNERDATA (X'01')
Indicates that the partner security credentials are to be retrieved during connection setup.

Output from SIOCSPARTNERINFO

No data items are returned by the SIOCSPARTNERINFO ioctl. The 31-bit input value that is passed on the call is saved and used during connection setup.

SIOCSPARTNERINFO return values

The SIOCSPARTNERINFO ioctl has the following possible return values:

0 Successful completion.

-1 An error occurred. For return codes and reason codes, see the following information:

- For the Language Environment, Java, or UNIX System Services APIs, see Table 121.
- For the Macro, Call instruction, or Rexx APIs, see *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*.
- For the CICS API, see *z/OS Communications Server: IP CICS Sockets Guide*.
- For the IMS API, see *z/OS Communications Server: IP IMS Sockets Guide*.

Table 121. SIOCSPARTNERINFO ioctl return codes for the Language Environment, Java, and UNIX System Services APIs

Return code and reason code	Problem	Action
EBADF No reason code	The specified socket descriptor is not valid.	Check and modify the socket descriptor.
EINVAL JrRequestTypeErr	The specified 31-bit input value is not valid.	Check and modify the input value. For the correct values, see the EZBPINFA or EZBPINFC files in SEZANMAC.
ENOPARTNERINFO JrNoSecDomain	The security domain name is not defined.	Define the security domain name.
EOPNOTSUPP JrAlreadyConn	The socket is already in the connected state. The request must be issued before the listen call or the connect call.	Check and modify the socket descriptor, or close the connection and reissue the call.
EOPNOTSUPP JrListenAlreadyDone	The listen call has already been issued for the socket. The request must be issued before the listen call.	Check and modify the socket descriptor, or close the socket or connection and reissue the call.
EPROTOTYPE JrSocketTypeNotSupported	The requested socket type is not supported.	Check and modify the socket descriptor.

SIOCGPARTNERINFO (X'C000F612')

The SIOCGPARTNERINFO ioctl provides an interface for an application to retrieve security information about its partner. The following information can be retrieved:

- Connection routing information
- Security credentials

  Partner security credentials can include the following:

  – Address-space user ID
  – Task-level user ID, if available
  – Address-space user security token (UTOKEN)
  – Task-level UTOKEN, if available

  For information about what is provided in the UTOKEN by the ICHRUTKN macro, see *z/OS Security Server RACF Data Areas*.

Results:

- Connection routing information is always returned.
- Security credentials are returned if they are requested and available.

Input to SIOCGPARTNERINFO

Set the following input items:

PI_Version
Set the version field to PI_VERSION_1 or PI_VERSION_CURRENT to indicate the version of the control block that is passed.

PI_ReqType
Indicates the type of data that is requested on this ioctl.

PI_REQTYPE_CONNTYPE (X'00')
Requests connection routing information. This is the default value.

PI_REQTYPE_PARTNER_USERID (X'01')
Requests connection routing information and the partner user ID.

PI_REQTYPE_PARTNER_UTOKEN (X'02')
Requests connection routing information and the partner UTOKEN. A UTOKEN is an encapsulation or representation of the security characteristics of a user.

Rule: You can request multiple items by adding request type values. For example, to request both the user ID and the UTOKEN, specify X'03'.

PI_TimeOut
When retrieving the partner security credentials, the PI_TimeOut value is used to set the timeout value, in seconds. If the request is not completed during this time, an error is returned. The valid range is 0 - 60.

Rule: The value 0 is valid only if you have issued the SIOCSPARTNERINFO ioctl. Use the SIOCSPARTNERINFO ioctl and the value 0 for PI_TimeOut on the SIOCGPARTNERINFO ioctl for an application that cannot be suspended.

Results:

- The timeout value is not used when only connection routing information is being retrieved.
- The timeout value is not used when partner information from a partner on the same stack is being retrieved.

Restriction: You cannot use a select mask to determine when an ioctl is complete, because an ioctl is not affected by whether the socket is running in blocking or nonblocking mode. If the ioctl times out, you need to reissue the ioctl to retrieve the partner security credentials.

PI_BufLen
Indicates the size of the buffer that is passed. Obtain a buffer that is large enough to hold the returning data, including the extension data portion if it is required.

Rules:

- Set the PI_BufLen value to match the ioctl input parameter argument length, and to be at least equal to the PI_FIXED_SIZE value.
- When you are retrieving the UTOKEN, set the PI_Buflen value to the PI_FIXED_SIZE value plus the PI_UTOKEN_EXT_SIZE value.

Output from SIOCGPARTNERINFO

The following output fields can be returned.

PI_Status
Indicates the type of information that is returned, which includes connection routing information, and can also include the partner user ID, the partner UTOKEN, or both.

PI_STATUS_CONNTYPE (x'01')
The connection routing information is returned in the PI_ConnType field.

PI_STATUS_PARTNER_USERID (x'02')
The address-space user ID of the partner and the length of this user ID are returned in the PI_UserID_AS and PI_UserID_Len_AS fields. The task-level user ID of the partner, if available, and the length of this user ID are returned in the PI_UserID_TL and PI_UserID_Len_TL fields.

PI_STATUS_PARTNER_UTOKEN (x'04')
The address-space UTOKEN of the partner and the length of this UTOKEN are returned in the PI_Utoken_AS and PI_Utoken_Len_AS fields. The task-level UTOKEN of the partner, if available, and the length of this UTOKEN are returned in the PI_Utoken_TL and PI_Utoken_Len_TL fields.

Multiple flags can be set, indicating that multiple items are returned. For example, if both PI_STATUS_CONNTYPE (x'01') and PI_STATUS_PARTNER_USERID (x'02') are returned for the PI_Status field, both connection routing information and the partner user ID are returned. The output fields that are returned depend on the flag values set in the PI_Status field:

- If PI_STATUS_CONNTYPE (x'01') is returned in the PI_Status field:

  PI_ConnType
  This value is the sysplex-specific connection routing information for a sockets application.

  PI_CONNTYPE_NOCONN (X'00')
  The socket is not connected.

  PI_CONNTYPE_NONE (X'01')
  The socket is active, but the partner is not in the same sysplex. If this indicator is set, the following three indicators are 0.

PI_CONNTYPE_SAME_CLUSTER (X'02')
The connection partner is in the same sysplex.

PI_CONNTYPE_SAME_IMAGE (X'04')
The connection partner is in the same MVS image. If this indicator is set, PI_CONNTYPE_SAME_CLUSTER is also set. If the connection partner is a distributed DVIPA, the same image bit is not set to on because the exact hosting stack is unknown.

PI_CONNTYPE_INTERNAL (X'08')
The communication from this node to the stack that hosts the partner application is not sent over links or interfaces outside of the sysplex. To determine whether both ends of the connection flow over internal links or interfaces, the partner application must also issue this ioctl, and both ends can exchange their results from this ioctl call (through an application-dependent method).

If this value is returned to an application, any subsequent rerouting decision due to failure of the current route results in either an alternate internal route or failure of the connection with the indication that no route is available. This logic ensures that a connection that an application relies upon to be an internal route does not transparently change to a route that is not internal.

- If PI_STATUS_PARTNER_USERID (x'02') is returned in the PI_Status field:

  PI_UserID_Len_AS
  If PI_REQTYPE_PARTNER_USERID (X'01') was provided as input and the PI_Status field indicates that the partner user ID is returned, this field is set to the length of the address-space user ID that is returned, excluding the trailing blanks.

  PI_UserID_AS
  If PI_REQTYPE_PARTNER_USERID (X'01') was provided as input and the PI_Status field indicates that the partner user ID is returned, this field is set to the address-space user ID that is returned, padded with blanks.

  PI_UserID_Len_TL
  If PI_REQTYPE_PARTNER_USERID (X'01') was provided as input and the PI_Status field indicates that the partner user ID is returned, this field is set to the length of the task-level user ID (if this user ID was returned), excluding the trailing blanks. If PI_UserID_Len_TL is set to 0, then only an address-space user ID is returned.

  PI_UserID_TL
  If PI_REQTYPE_PARTNER_USERID (X'01') was provided as input and the PI_Status field indicates that the partner user ID is returned, this field is set to the task-level user ID (if this user ID was returned), padded with blanks.

- If PI_STATUS_PARTNER_UTOKEN (x'04') is returned in the PI_Status field:

  PI_Ext_Length
  If PI_REQTYPE_PARTNER_UTOKEN (X'02') was provided as input and the PI_Status field indicates that the partner UTOKEN is returned, this field is set to the length of the UTOKEN, which includes the address-space UTOKEN and, if available, the task-level UTOKEN.

PI_Ext_Offset
If PI_REQTYPE_PARTNER_UTOKEN (X'02') was provided as input and the PI_Status field indicates that the partner UTOKEN is returned, this field is set to the offset to the PI_Utoken_Ext structure.

PI_Utoken_Ext
If PI_REQTYPE_PARTNER_UTOKEN (X'02') was provided as input and the PI_Status field indicates that the partner UTOKEN is returned, this structure contains the address-space UTOKEN of the partner, and the task-level UTOKEN if it is available.

PI_Utoken_Len_AS
If PI_REQTYPE_PARTNER_UTOKEN (X'02') was provided as input and the PI_Status field indicates that the partner UTOKEN is returned, this field is set to the length of the address-space UTOKEN that was returned.

PI_Utoken_AS
If PI_REQTYPE_PARTNER_UTOKEN (X'02') was provided as input and the PI_Status field indicates that the partner UTOKEN is returned, this field is set to the address-space UTOKEN.

PI_Utoken_Len_TL
If PI_REQTYPE_PARTNER_UTOKEN (X'02') was provided as input and the PI_Status field indicates that the partner UTOKEN is returned, this field is set to the length of the task-level UTOKEN, if it was returned. If PI_Utoken_Len_TL is set to 0, then only an address-space UTOKEN is returned.

PI_Utoken_TL
If PI_REQTYPE_PARTNER_UTOKEN (X'02') was provided as input and the PI_Status field indicates that the partner UTOKEN is returned, this field is set to the task-level UTOKEN, if it was returned.

For information about what is provided in the UTOKEN by the ICHRUTKN macro, see *z/OS Security Server RACF Data Areas*.

Using the partner information control block

The partner information control block contains control information that describes the SIOCGPARTNERINFO request, as shown in Table 122 and Table 123 on page 700.

Table 122. SIOCGPARTNERINFO ioctl partner information control block structure

Field	Offset	Length in bytes	Format
PI_Version	0	1	Binary
PI_RsvdAvail1 (Reserved; set to 0)	1	3	Binary
PI_ReqType	4	4	Binary
PI_TimeOut	8	4	Binary
PI_BufLen	12	4	Binary
PI_Status	16	4	Binary
PI_ConnType	20	4	Binary
PI_UserID_Len_AS	24	1	Binary
PI_UserID_AS	25	8	EBCDIC
Null character	33	1	Binary

Table 122. SIOCGPARTNERINFO ioctl partner information control block structure (continued)

Field	Offset	Length in bytes	Format
PI_UserID_Len_TL	34	1	Binary
PI_UserID_TL	35	8	EBCDIC
Null character	43	1	Binary
PI_RsvdAvail2 (Reserved; set to 0)	44	24	Binary
PI_Ext_Length	68	4	Binary
PI_Ext_Offset	72	4	Binary

Table 123. SIOCGPARTNERINFO ioctl partner information UTOKEN extension control block structure

Field	Offset	Length in bytes	Format
PI_Utoken_Len_AS	0	1	Binary
PI_Utoken_RsvdAvail1 (Reserved; set to 0)	1	3	Binary
PI_Utoken_AS	4	80	EBCDIC
PI_Utoken_Len_TL	84	1	Binary
PI_Utoken_RsvdAvail2 (Reserved; set to 0)	85	3	Binary
PI_Utoken_TL	88	80	EBCDIC

For a description of each field, see "Input to SIOCGPARTNERINFO" on page 696 and "Output from SIOCGPARTNERINFO" on page 697.

SIOCGPARTNERINFO return values

The SIOCGPARTNERINFO ioctl has the following possible return values:

0 Successful completion.

-1 An error occurred. For return codes and reason codes, see the following information:

- For the Language Environment, Java, or UNIX System Services APIs, see Table 124.
- For the Macro, Call instruction, or Rexx APIs, see *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*.
- For the CICS API, see *z/OS Communications Server: IP CICS Sockets Guide*.
- For the IMS API, see *z/OS Communications Server: IP IMS Sockets Guide*.

Table 124. SIOCGPARTNERINFO ioctl return codes for the Language Environment, Java, and UNIX System Services APIs

Return code and reason code	Problem	Output returned	Action
EACCES JrIoctlAccessAuthorization	The application is not running in supervisor state, is not APF authorized, or is not permitted to the appropriate SERVAUTH profile.	PI_ConnType	Allow the application to issue this ioctl, or provide the user ID with the proper SERVAUTH permission.

Table 124. SIOCGPARTNERINFO ioctl return codes for the Language Environment, Java, and UNIX System Services APIs (continued)

Return code and reason code	Problem	Output returned	Action
EALREADY JrAlreadyInProgress	The request is already in progress. Only one ioctl can be outstanding.	PI_ConnType	Check and modify the socket descriptor, if specified; otherwise, no action is needed.
EBADF No reason code	The specified socket descriptor is not valid.	None	Check and modify the socket descriptor.
EINPROGRESS JrNoSuspend	The ioctl was issued in no-suspend mode after the SIOCSPARTNERINFO ioctl was issued, but the partner security credentials are not currently available.	PI_ConnType	Retry the ioctl at a later time, or issue the ioctl with a timeout value to set the amount of time to wait while the partner security credentials are being retrieved. **Restriction:** You cannot use a select mask to determine when an ioctl is complete, because an ioctl is not affected by whether the socket is running in blocking or nonblocking mode. If the ioctl times out, you need to reissue the ioctl to retrieve the partner security credentials.
EINVAL JrBuffLenInvalid	The input buffer length is below the required minimum length.	None	Check and modify the ioctl parameter argument length to be at least the size of the PI_FIXED_SIZED value, or check and modify the PI_BufLen value to match the ioctl parameter argument length. For the correct values, see the EZBPINFA or EZBPINFC files in SEZANMAC.
EINVAL JrInvalidVersion	The specified PI_Version value is not valid.	None	Check and modify the PI_Version value. For the correct values, see the EZBPINFA or EZBPINFC files in SEZANMAC.
EINVAL JrRequestTypeErr	The specified PI_ReqType value is not valid.	None	Check and modify the PI_ReqType value. For the correct values, see the EZBPINFA or EZBPINFC files in SEZANMAC.
EINVAL JrSecOutofRange	The specified PI_TimeOut value is not valid.	None	Check and modify the PI_TimeOut value. For the correct values, see the EZBPINFA or EZBPINFC files in SEZANMAC.

Table 124. SIOCGPARTNERINFO ioctl return codes for the Language Environment, Java, and UNIX System Services APIs (continued)

Return code and reason code	Problem	Output returned	Action
ENOBUFS JrBuffTooSmall	The supplied buffer was too small.	PI_ConnType, and the amount of storage needed is returned in PI_Buflen	Create a larger input buffer based on the value returned in the PI_Buflen field.
ENOMEM JrNoCsaStorage	TCP/IP cannot process the request because there is insufficient common storage available.	None	Check and modify the amount of common storage that is available.
ENOPARTNERINFO JrInvalidTCPIPStack	The partner resides in a TCP/IP stack that is running a release earlier than V1R12.	PI_ConnType	Ensure that both endpoints reside in TCP/IP stacks that are running V1R12 or any later release.
ENOPARTNERINFO JrNoPartnerInfo	The partner is not in the same sysplex.	PI_ConnType	Check and modify the socket descriptor. However, if the partner is not in the same sysplex, no security credentials can be returned.
ENOPARTNERINFO JrNoSecDomain	The security domain name is not defined.	PI_ConnType	Define the security domain name on both endpoints. After you define the security domain name, the application might need to close the connection if the ioctl is needed.
ENOPARTNERINFO JrNoSuspend	The ioctl was issued in no-suspend mode and the SIOCSPARTNERINFO ioctl has not been issued.	PI_ConnType	Issue the ioctl with a timeout value to set the amount of time to wait while the partner security credentials are being retrieved. **Restriction:** You cannot use a select mask to determine when an ioctl is complete, because an ioctl is not affected by whether the socket is running in blocking or nonblocking mode. If the ioctl times out, you need to reissue the ioctl to retrieve the partner security credentials.
ENOPARTNERINFO JrNotSameSecDomain	Both endpoints do not reside in the same security domain.	PI_ConnType	Check and modify the security domain name for the endpoints. After you correct the security domain name, the application might need to close the connection if the ioctl is needed.

Table 124. SIOCGPARTNERINFO ioctl return codes for the Language Environment, Java, and UNIX System Services APIs (continued)

Return code and reason code	Problem	Output returned	Action
ENOPARTNERINFO JrTimeOut	The wait time for the request has expired, possibly as the result of network problems.	PI_ConnType	Retry the request. **Restriction:** You cannot use a select mask to determine when an ioctl is complete, because an ioctl is not affected by whether the socket is running in blocking or nonblocking mode. If the ioctl times out, you need to reissue the ioctl to retrieve the partner security credentials.
ENOTCONN JrSocketNotCon	The requested socket is not connected.	PI_ConnType with the PI_CONNTYPE_NOCONN value returned	Check and modify the socket descriptor, or reissue the ioctl after the connect call from the client side or the accept call from the server side.
EPROTOTYPE JrSocketTypeNotSupported	The requested socket type is not supported.	None	Check and modify the socket descriptor.

Coding examples – SIOCSPARTNERINFO and SIOCGPARTNERINFO ioctl calls

These examples show sample code for building and issuing the SIOCGPARTNERINFO ioctl and the optional SIOCSPARTNERINFO ioctl.

Assembler example – SIOCGPARTNERINFO ioctl call

The following sample assembler code builds the SIOCGPARTNERINFO ioctl and issues the ioctl using the Macro API (EZASMI). The SIOCGPARTNERINFO ioctl retrieves the connection routing information and the security credentials of a partner.

```
⋮
*********************************************************************
*                                                                   *
*        Issue SIOCGPARTNERINFO IOCTL to retrieve connection        *
*        routing information and the security credentials           *
*        of a partner.                                              *
*                                                                   *
*********************************************************************
*
         USING PARTNERINFO,R6
         MVI   PI_Version,PI_VERSION_1
         MVC   PI_ReqType,=A(PI_REQTYPE_PARTNER_USERID+PI_REQTYPE_PARTNX
               ER_UTOKEN)
         MVC   PI_TIMEOUT,=A(PI_TIMEOUT_MAXIMUM)
         MVC   PI_BUFLEN,=A(PI_FIXED_SIZE+PI_UTOKEN_EXT_SIZE)
 *
         EZASMI TYPE=IOCTL,       Issue Macro                         X
               S=SOCDESCA,        ACCEPT SOCKET                       X
               COMMAND='SIOCGPARTNERINFO',                            X
               REQARG=PARTNERINFO,                                    X
               RETARG=PARTNERINFO,                                    X
               ERRNO=ERRNO,       (Specify ERRNO field)               X
```

```
                RETCODE=RETCODE,   (Specify RETCODE field)              X
                REQAREA=REQAREA,   IN CASE WE ARE DOING EXITS OR ECBS   X
                ERROR=ERROR        Abend if Macro error
*
.
.
.
          EZBPINFA DSECT=NO        SIOCGPARTNERINFO ioctl structure
```

PL/I example – SIOCGPARTNERINFO ioctl call

The following sample PL/I code builds and issues the SIOCGPARTNERINFO ioctl. The SIOCGPARTNERINFO ioctl retrieves the connection routing information and the security credentials of a partner.

```
.
.
.
 /* SIOCGPartnerInfo ioctl mapping and constants                       */
 % include EZBPINF1;
.
.
.
 dcl 1 IoctlBuffer Based,
    3 data char(PI_FIXED_SIZE + PI_UTOKEN_EXT_SIZE);
.
.
.
 /*********************************************************************/
 /* Allocate the IOCTL buffer for SIOCGPARTNERINFO and issue ioctl.   */
 /*********************************************************************/
 allocate ioctlBuffer set(piPtr);

 piPtr->PI_Version  = PI_VERSION_1;
 piPtr->PI_ReqType  = PI_REQTYPE_PARTNER_USERID
                    + PI_REQTYPE_PARTNER_UTOKEN;
 piPtr->PI_TimeOut  = PI_TIMEOUT_MAXIMUM;
 piPtr->PI_BufLen   = PI_FIXED_SIZE + PI_UTOKEN_EXT_SIZE;

 call ezasoket(IOCTL,
               SOCK_STREAM,
               SIOCGPARTNERINFO,      /* SIOCGPARTNERINFO ioctl      */
               piPtr->IoctlBuffer,    /* input buffer                */
               piPtr->IoctlBuffer,    /* output buffer               */
               ERRNO,
               RETCODE);
```

COBOL example – SIOCGPARTNERINFO ioctl call

The following sample COBOL code builds and issues the SIOCGPARTNERINFO ioctl. The SIOCGPARTNERINFO ioctl retrieves the connection routing information and the security credentials of a partner.

```
.
.
.
*===============*
 Data Division.
*===============*
*---------------------------------------------------------------*
* Variables used by the SIOCGPARTNERINFO IOCTL call             *
*---------------------------------------------------------------*
 01  partnerinfo-data pic x.
     COPY EZBPINFB.
 01  FIELD-DEFINE           PIC X(4).
 01  FIELD-DEFINEVALUE redefines field-DEFINE.
   02 defineNum             PIC 9(8) Binary.
 01  FIELD-DEFINE1          PIC X(4).
 01  FIELD-DEFINE1VALUE redefines field-DEFINE1.
   02 defineNum1            PIC 9(8) Binary.
.
.
.
*===============================================*
 Procedure Division.
*===============================================*
.
.
.
*****************************************************************
* Issue IOCTL SIOCGPARTNERINFO                                  *
```

```
*****************************************************************
 SiocGPartnerInfo-Ioctl.
     move PI-VERSION-1 to PI-VERSION.
     move PI-TIMEOUT-MAXIMUM to PI-TIMEOUT.

     move PI-FIXED-SIZE to PI-BUFLEN.
     add  PI-UTOKEN-EXT-SIZE to PI-BUFLEN.

     move PI-REQTYPE-PARTNER-USERID to FIELD-DEFINE.
     move PI-REQTYPE-PARTNER-UTOKEN to FIELD-DEFINE1.
     add defineNum1 to defineNum.
     move FIELD-DEFINE to PI-REQTYPE.

     move soket-ioctl to ezaerror-function.
     Call 'EZASOKET' using soket-ioctl socket-descriptor
         SIOCGPARTNERINFO
         PARTNERINFO PARTNERINFO
         errno retcode.
 SiocGPartnerInfo-Ioctl-Exit.
          Exit.
```

C example – SIOCSPARTNERINFO and SIOCGPARTNERINFO ioctl calls

The following sample C code builds and issues the SIOCSPARTNERINFO and SIOCGPARTNERINFO ioctl calls. The SIOCGPARTNERINFO ioctl retrieves the connection routing information and the security credentials of a partner. The optional SIOCSPARTNERINFO ioctl enables an application to avoid suspending while retrieving partner security credentials with the SIOCGPARTNERINFO ioctl.

```
:
#include <sys/ioctl.h>
#include <termios.h>
:
#include "ezbpinfc.h"               /* SIOCSPARTNERINFO and
                                       SIOCGPARTNERINFO ioctls          */
:
   char *buff     = NULL;
   int   s;
   int   function = PI_REQTYPE_SET_PARTNERDATA;
   int   bufsiz;
:
   /*********************************************************************/
   /* issue the SIOCSPARTNERINFO ioctl to avoid suspending              */
   /*********************************************************************/
   rc = ioctl(s,
              SIOCSPARTNERINFO,
              (char *)&function,
              sizeof(function));
:
   bufsiz = PI_FIXED_SIZE + PI_UTOKEN_EXT_SIZE;
   buff = (char *)__malloc31(bufsiz);

   if (buff != NULL)
   {
     memset(buff,0,bufsiz);
     iocPtr = (struct PartnerInfo *)buff;
     iocPtr->PI_BufLen  = bufsiz;
     iocPtr->PI_Version = PI_VERSION_1;
     iocPtr->PI_ReqType = PI_REQTYPE_PARTNER_USERID
                        + PI_REQTYPE_PARTNER_UTOKEN;
     iocPtr->PI_TimeOut = PI_TIMEOUT_MINIMUM;

     rc = w_ioctl(s,
```

```
                    SIOCGPARTNERINFO,
                    iocPtr->PI_BufLen,
                    (char *)iocPtr);
}
```

Chapter 17. Interfacing with the Digital Certificate Access Server (DCAS)

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 *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 topics.

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.
  - Clients can send multiple requests on a single connection. Use the Correlator field to match requests and responses.

See the Configuring RACF services for Express Logon information in *z/OS Communications Server: IP Configuration Guide* for more details about PassTickets.

The request and response formats are described in the following topic.

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.

Interfacing with the DCAS: Defining the format for request and response specifications

Table 125 contains format 1 request information.

Table 125. 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-*n*	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 *z/OS Security Server RACF Security Administrator's Guide* for information about defining PTKTDATA for applications.		

Table 126 contains format 1 response information.

Table 126. 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 128 on page 709)
12-15	Return Code 3	Extended (see Table 128 on page 709)
16-19	Return Code 4	Extended (see Table 128 on page 709)
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 127 on page 709 contains format 2 request information.

Table 127. 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-*n*	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 *z/OS Security Server RACF Security Administrator's Guide* for information about defining PTKTDATA for applications.		

Table 128. 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 *z/OS Communications Server: IP Diagnosis Guide* 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 *z/OS Communications Server: IP Diagnosis Guide*.

Table 128. 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 us 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 *z/OS Communications Server: IP Diagnosis Guide* 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 *z/OS Security Server RACF Security Administrator's Guide* 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 details about configuring DCAS, see the EXPRESS LOGON using DCAS (Digital Certificate Access Server) information in *z/OS Communications Server: IP Configuration Reference*. The DCAS configuration statements described in Table 129 require coordination between the DCAS client and server.

Table 129. 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 18. Miscellaneous programming interfaces

This topic contains descriptions of the following:

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 *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 Appendix E, "Type 119 SMF records," on page 743 for additional information.
- Through the callable TCP/IP network management interface. See "TCP/IP callable NMI (EZBNMIFR)" on page 637 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.

SIOCSAPPLDATA 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 130. SetApplData

Field name	Size	Description
SetAD_eye1	8	constant SETADEYE1
SetAD_ver	4	constant SETADVER

Table 130. SetApplData (continued)

Field name	Size	Description
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 131. 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.

SIOCSAPPLDATA output

The SIOCSAPPLDATA IOCTL sets the following return codes and reason codes:

Table 132. 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	ENOBUFS	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.

SIOCSAPPLDATA C language example

```
#include <ezbyaplc.h>
char myappldata[SETADBUFLEN+1];       /* extra byte for null string terminator */
SetApplData myIoctlParm;
SetADcontainer myBuffer;

sprintf(myappldata, "@HRSERVR%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);
```

SIOCSMOCTL IOCTL

Applications that use the UNIX System Services optimized Asynchronous Socket I/O option (designated by the AioCommBuff bit in the AIOCB control block) can exploit 64-bit shared memory objects. The application allocates a shared memory object and issues a new IOCTL SIOCSMOCTL that enables TCP/IP to establish access to the memory object or to remove access to the memory object.

Table 133 lists SIOCSMOCTL requirements.

Table 133. SIOCSMOCTL requirements

Minimum authorization:	Executing in supervisor state, in system key, or APF authorized
Dispatchable unit mode:	Task or SRB
Cross memory mode:	PASN=HASN=SASN
Addressing mode:	AMODE31 or AMODE64
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

A SIOCSMOCTL IOCTL can be issued on any type of socket (stream, datagram, or raw), and requires that the application be authorized. After access to shared memory objects is established, the application can use buffers in the memory objects for asynchronous I/O by setting the AioCommBuff bit in the AIOCB control block on any stream socket that it has created. TCP/IP internally associates access to shared memory objects with the socket that was used to issue the SIOCSMOCTL IOCTL; if any shared memory object associations remain when that socket is closed, these memory associations are automatically broken and TCP/IP access to those objects is removed. The application must ensure that the socket that is used to issue the SIOCSMOCTL IOCTL is closed only after all other stream sockets that use buffers in those shared memory objects are closed.

For more information about the use of the BPX1AIO and BPX4AIO services and about the use of the AioCommBuff bit, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

For more information about the use of shared memory objects, see *z/OS MVS Programming: Extended Addressability Guide*.

SIOCSMOCTL input

SIOCSMOCTL input consists of a pointer to a SMOCTL_IOCTL structure that contains the following:

Table 134. SIOCSMOCTL input structure

Data item	Description
SMOCTL_Version	Required input that contains the signed 31-bit version is Version 1 of the SIOCSMOCTL IOCTL.
SMOCTL_Request	Required input that contains the signed 31-bit request type, which can be one of the following: • Attach request (establish access from TCP/IP to the shared memory object • Detach request (remove access from TCP/IP to the shared memory object)
SMOCTL_ObjectAddr	Required input that contains the 64-bit starting address of a shared memory object to be attached or detached.
SMOCTL_IARV64_Retcode	Output field that contains the IARV64 return code.
SMOCTL_IARV64_Rsncode	Output field that contains the IARV64 reason code.

The SIOCSMOCTL IOCTL parameter list for assembler applications is defined in the EZBITSIA macro in the SEZANMAC data set. For C/C++ applications, the parameter list is defined in the header file, ezbitsic.h. This header file is installed in the SEZANMAC data set and in the file system directory, /usr/include.

SIOCSMOCTL output

The SMOCTL_IOCTL structure is updated with status information that pertains to the attach or delete request. The SIOCSMOCTL ioctl sets the return codes and reason codes that are described in Table 135.

Table 135. SIOCSMOCTL return and reason codes

ReturnValue	ReturnCode	ReasonCode	Meaning
0	0	0	The request was successful.
-1	EACCESS	JRIOCTLAccessAuthorization	The request was not successful. The issuer of the IOCTL is not authorized.
-1	EINVAL	JRSocketCallParmError	The request was not successful. The input parameter length is incorrect, the version is not valid, or the request type is not valid.
-1	ENOMEM	JRSmNoStorage	The request was not successful. The attach request failed due to a storage shortage.
-1	EINVAL	JRDuplicateSmoAttach	The request was not successful. For an attach request, the specified shared memory object has already been attached.

Table 135. SIOCSMOCTL return and reason codes (continued)

ReturnValue	ReturnCode	ReasonCode	Meaning
-1	EINVAL	JRSmoNotAttached	The request was not successful. For a detach request, the specified shared memory object is not attached.
-1	EMVSPARM	JRIARV64Error	The request was not successful. For an attach or detach request, the IARV64 services encountered an error. Fields SMOI_IARV64_Retcode and SMOI_IARV64_Rsncode describe the error.

Tips:

- Create shared memory objects in system key (0-7) fetch-protected storage to maintain integrity of the data.
- Create an ancillary socket that is used only to issue SIOCSMOCTL IOCTL requests.

Steps for creating an ancillary socket

Before you begin: Ensure that the ancillary socket is not closed until all stream sockets that might reference those shared memory objects are closed. This can be accomplished by taking advantage of the fact that the UNIX System Services process cleanup service closes sockets sequentially from lowest-numbered socket descriptor to highest-numbered socket descriptor.

Perform the following steps:

1. Issue a getrlimits() request to discover the largest socket descriptor available for the process.

   _____

2. Issue a dup2() request to copy the original ancillary socket descriptor to the largest socket descriptor obtained from the getrlimits() request.

   _____

3. Close the original ancillary socket descriptor.

Applications in a common INET environment

When a generic application in a common INET environment creates a socket, UNIX System Services creates socket sub-instances to each active TCP/IP instance. When the application then issues an IOCTL on its socket, UNIX System Services propagates the IOCTL to all active TCP/IP instances until the sockets all indicate that the IOCTL was successful. If a TCP/IP instance indicates a failure, IOCTL processing stops at that point and a failure return value, return code, and reason code are returned to the application. If the generic application issues a SIOCSMOCTL attach request and encounters a failure, some TCP/IP instances might have access to the shared memory object, and some might not. For problem determination purposes, all TCP/IP instances should gain access to the shared

memory object, or no TCP/IP instances should gain access to it. When a SIOCSMOCTL attach request fails in a common INET environment, the application should immediately issue a SIOCSMOCTL detach request to ensure that no TCP/IP instance has access to the shared memory object and should thereafter not set a value in the AioCommBuff bit in the AIOCB.

A generic application in a common INET environment can use the SOCK#SO_EIOIFNEWTP socket option on its listening socket so that the application is notified when a TCP/IP instance is stopped and restarted. When a TCP/IP instance is recycled, the application's response is to close the listening socket and create a new listening socket, which cause new listening socket sub-instances to each active TCP/IP instance to be created. The application should do the following to ensure that the recycled TCP/IP instance gains access to the shared memory object and that the other TCP/IP instances retain their access to the share memory object:

1. After the new listening socket is created, create a new ancillary socket (which is propagated to all active TCP/IP instances).
2. Issue a SIOCSMOCTL attach request for the shared memory object on the new ancillary socket.
3. Close the original ancillary socket descriptor.
4. Perform the “Steps for creating an ancillary socket” on page 717 to ensure that the new ancillary socket is not closed until all stream sockets that might reference the shared memory object are closed.

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 the TCP_KeepAlive value 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 it is 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, the value 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 the value 0 if keep alive timing is not active.

Unlike the algorithm that is used to issue probes during an SO_KeepAlive cycle, the TCP_KeepAlive function varies the number of probes that are issued before terminating the connection. Probe retry intervals are scaled in proportion to the interval specified, as shown in Table 136 on page 719.

Table 136. 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 x 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. 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 137 lists the well-known port assignments for TCP.

Table 137. 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 137. 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 138 lists the well-known port assignments for UDP.

Table 138. 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	**rvd-control**		rvd control port
2001	**rauth2**		Andrew File System service, for the Venus process
2002	**rfilebulk**		Andrew File System service, for the Venus process

Table 138. 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 B. Programming interfaces for providing classification data to be used in differentiated services policies

Applications and users of TCP/IP networks might have different requirements for the service they receive from those networks. A network that treats all traffic as best effort might 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 *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 might provide services for several different types of requests using a single well-known port. A network administrator might 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 topic 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 about these APIs, see *z/OS XL C/C++ Run-Time Library Reference*.
- z/OS UNIX System Services Assembler Callable services socket APIs. For more information about these APIs, see *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, job name, 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 might 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 might 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 topic:

```
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 *cmsg_len* 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 *cmsg_data* field.
- The *cmsg_level* 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 *cmsg_type* 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 included 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 SENDMSG 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 topic.
  - 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 C. Type 109 SMF records

Table 139 shows the format of syslogd messages as written to SMF.

Table 139. Type 109 SMF record layout

Offsets	Name	Length	Format	Description
0(x'0')	SMF109LEN	2	Binary	Record length (maximum size 32 756). This field and the next field (total of 4 bytes) form the record descriptor word (RDW). The first 2 bytes of this field must contain the logical record length, including the RDW. The second 2 bytes which are in the following field are used for variable block spanned records. If the record is spanned, set these 2 bytes to hexadecimal zeros. Both fields must be filled in before writing the record to the SMF data set.
0(x'2')	SMF109SEG	2	Binary	Segment descriptor (see previous record length field).
0(x'4')	SMF109FLG	1	Binary	System indicators (bits and meaning when set): **0-2** Reserved **3** MVS/SP Version 4 and later. Bits 3, 4, 5, and 6 are on (*). **4** MVS/SP Version 3 and later. Bits 4,5, and 6 are on. **5** MVS/SP Version 2 and later. Bits 5 and 6 are on. **6** VS2. Bit 6 is on. **7** Reserved. Use information located elsewhere in this record to determine the MVS product level.
5(X'5')	SMF109RTY	1	Binary	Record type: 109 (X'6D')
6(X'6')	SMF109TME	4	Binary	Time since midnight, in hundredths of a second, that has elapsed since the record was moved into the SMF buffer. In record types 2 and 3, this field indicates the time that the record was moved to the dump data set.
10(X'A')	SMF109DTE	4	Packed	Date when the record was moved into the SMF buffer. In the form of *00yyddF* or *0cyydddF* [where c is 0 for 19xx and 1 for 20xx, yy is the current year (0-99), dd is the current day (1-366), and F is the sign]. In record types 2 and 3, this field indicates the date that the record was moved into the dump data.
14(X'E')	SMF109SID	4	EBCDIC	System identification (from the SID parameter).
18(X'12')	SMF109LOG	4096	EBCDIC	System logging daemon (syslogd) messages.

Appendix D. Type 118 SMF records

This topic describes the Type 118 SMF records for the Telnet and FTP servers, API calls, FTP and Telnet client calls, and syslogd and contains the following layouts:

- "Standard subtype record numbers"
- "TN3270E Telnet server SMF record layout" on page 734
- "FTP server Type 118 SMF record layout" on page 735
- "SMF record layout for API calls" on page 737
- "SMF record layout for FTP client calls" on page 738
- "SMF record layout for Telnet client calls" on page 740
- "SMF record layout for TCPIPSTATISTICS" on page 740

The EZASMF76 macro can be used to map the TCP/IP SMF records. EZASMF76 produces assembler level DSECTs for the Telnet (Server and Client), FTP (Server and Client), and API SMF records.

See Appendix E, "Type 119 SMF records," on page 743 for a description of the preferred Type 119 SMF records.

To create the Telnet Client SMF record layout, code the following in an assembler program:

```
EZASMF76 TELNET=YES
```

To create the FTP Server SMF record layout, code the following in an assembler program:

```
EZASMF76 FTP=YES
```

To create the API SMF record layout, code the following in an assembler program:

```
EZASMF76 API=YES
```

Standard subtype record numbers

TCP/IP logging of SMF records can be activated through the use of the SMFCONFIG and SMFPARMS statements in the TCP/IP profile. The TCP/IP SMF records written using record Type 118 (x'76') and their standard subtypes are described in this topic.

Guideline: If you use the SMFPARMS statement, you can specify that records be written with nonstandard subtype records. However, you should use the standard subtype records shown in Table 140.

Table 140. Standard subtype record numbers

Record number	Description
1	TCP API initialization
2	TCP API termination
3	FTP client
4	TN3270 client
5	TCP/IP statistics

Table 140. Standard subtype record numbers (continued)

Record number	Description
6-19	Reserved
20	TN3270 server initialization
21	TN3270 server termination
22-69	Reserved
70	FTP server append subcommand
71	FTP server delete subcommand
72	FTP server logon failures
73	FTP server rename
74	FTP server retrieve
75	FTP server store
76–255	Reserved

TN3270E Telnet server SMF record layout

The Type 118 TN3270E Telnet server (Telnet) SNA session record written by the z/OS TN3270E Telnet server has the format shown in Table 141.

Table 141. TN3270E Telnet server SMF record format

Offset	Name	Length	Format	Description
0(x'0')	SMFHEADR			Standard SMF header
4(x'4')	SMFHDFLG	1	Binary	A system indicator that is set to 94 (x'5E').
5(x'5')	SMFHDTYP	1	Binary	Record type (set to 118, or x'76')
22(x'16')	SMFHDSUB	2	Binary	Record subtype
	Telnet data			
24(x'18')	SMFTNTCM	4	EBCDIC	Event type **LOGN** Session initiation **LOGF** Session termination
28(x'1C')	SMFTNTLU	8	EBCDIC	LU name
36(x'24')	SMFTNTAP	8	EBCDIC	Application name
44(x'2C')	SMFTNTIA	4	Binary	Interal logical device address (same for LOGN and LOGF records).
48(x'30')	SMFTNTRA	4	Binary	Remote IP address
52(x'34')	SMFTNTLA	4	Binary	Local IP Address
56(x'38')	SMFTNTST	8	EBCDIC	Started task qualifier name, for example, TCPIP
64(x'40')	SMFTNTHN	8	EBCDIC	TCP/IP host name
72(x'48')		2		Reserved
74(x'4A')	SMFTNTIN	4	Binary	Inbound byte count
78(x'4E')	SMFTNTOU	4	Binary	Outbound byte count
82(x'52')	SMFTNTLF	4	Binary	Time specified in hundredths of a second (LOGF record only)

Table 141. TN3270E Telnet server SMF record format (continued)

Offset	Name	Length	Format	Description
86(x'56')	SMFTNTPD	4	Packed decimal	Julian date (LOGF record only). The date is in the form of 0CYYDDDF, where C is 0 for 19yy and 1 for 20yy, DDD is the day of the year (1-365), and F is the sign.
90(x'5A')	SMFTNTRP	2	Binary	Remote port number
92(x'5C')	SMFTNTLP	2	Binary	Local port number

FTP server Type 118 SMF record layout

The Type 118 SMF record written by the FTP server has the format shown in Table 142.

Table 142. FTP server Type 118 SMF record format

Offset	Name	Length	Format	Description
0(x'0')	SMFHEADR	24		Standard SMF header
4(x'4')	SMFHDFLG	1	Binary	Record flag (set to 66, or x'42')
5(x'5')	SMFHDTYP	1	Binary	Record type (set to 118, or x'76')
22(x'16')	SMFHDSUB	2	Binary	Record subtype
	FTP server data			
24(x'18')	SMFFTPCM	4	EBCDIC	FTP subcommand **APPE** Append **DELE** Delete **LOGN** Login **REN** Rename **RETR** Retrieve **STOR** Store **STOU** Store unique
28(x'1C')	SMFFTPTY	4	EBCDIC	FTP file type (SEQ, JES, SQL)
32(x'20')	SMFFTPSA	4	Binary	Remote (client) IP address (IPv4) or –1 for IPv6
36(x'24')	SMFFTPSL	4	Binary	Local (server) IP address (IPv4) or –1 for IPv6
40(x'28')		8		Reserved
48(x'30')	SMFFTPSU	8	EBCDIC	Local user ID
56(x'38')	SMFFTPFM	1	EBCDIC	Data format **A** ASCII **E** EBICIDIC **I** Image (binary) **B** Double-byte **U** USC-2

Table 142. FTP server Type 118 SMF record format (continued)

Offset	Name	Length	Format	Description
57(x'39')	SMFFTPMO	1	EBCDIC	Mode **S** Stream **B** Block **C** Compressed
58(x'3A')	SMFFTPST	1	EBCDIC	Structure **F** File **R** Record
59(x'3B')	SMFFTPDT	1	EBCDIC	Data set type **P** PDS **S** Sequential **H** HFS
60(x'3C')	SMFFTTRS	4	Binary	Start time of transmission (See Note)
64(x'40')	SMFFTTRE	4	Binary	End time of transmission
68(x'44')	SMFFTTBC	4	Binary	Byte count of transmission
72(x'48')	SMFFTPXD	1	EBCDIC	FTP ID **S** Server
73(x'49')	SMFFTSLR	3	EBCDIC	Last reply sent to the client from the FTP server
76(x'4C')	SMFFTDSN	44	EBCDIC	User ID/Data set name For LOGN records, this is the user ID of the failed login attempt; otherwise, this is the data set name, or up to the first 44 bytes of the z/OS UNIX file name.
120(x'78')	SMFFTMEM	8	EBCDIC	Member name of PDS
128(x'80')		8		Reserved
136(x'88')	SMFFTDS2	44	EBCDIC	Second data set name, if needed (for example, for REN subcommands). For z/OS UNIX files, up to the first 44 bytes of the z/OS UNIX file name.
180(x'B4')	SMFFTMM2	8	EBCDIC	Second member name, if needed (for example, REN subcommands involving PDS files).
188(x'BC')	SMFFTSTC	8	EBCDIC	Started task qualifier
196(x'C4')	SMFFTHST	8	EBCDIC	TCP/IP host name
204(x'CC')	SMFFTSRP	2	Binary	Remote (client) port number
206(x'CE')	SMFFTSLP	2	Binary	Local (server) port number
208(x'D0')	SMFFTOF1	2	Binary	Offset to the first z/OS UNIX file name field
210(x'D2')	SMFFTOF2	2	Binary	Offset to the second z/OS UNIX file name field
212(x'D4')	SMFFTBYF	8	Floating point Hex	Bytes transferred counter. The leftmost byte is an exponent, and other seven bytes are significant bytes.

Table 142. FTP server Type 118 SMF record format (continued)

Offset	Name	Length	Format	Description
220(x'DC')	SMFFTGIG	4	Binary	Bytes transferred, 4 GB increments. Increments with every 4 GB of data transfer, starting from 0.

Notes:

1. The start time of the transmission might be greater than the end time when the transmission began on the previous day.

Two variable-length fields at the end of the record contain z/OS UNIX file names. The variable-length z/OS UNIX name fields have the format shown in Table 143.

Table 143. z/OS UNIX file name (variable length fields appended to end of FTP server record)

Offset	Name	Length	Format	Description
0(x'0')		2	Binary	Length of the z/OS UNIX file name
2(x'2')		*n*	EBCDIC	z/OS UNIX file name (maximum length is 1023 bytes)

SMF record layout for API calls

The SMF record written by API calls for sockets has the format shown in Table 144.

Table 144. API call SMF record format

Offset	Name	Length	Format	Description
0(x'0')	SMFHEADR	24		Standard SMF header
4(x'4')	SMFHDFLG	1	Binary	Record flag (set to 66, or x'42')
5(x'5')	SMFHDTYP	1	Binary	Record type (set to 118, or x'76')
22(x'16')	SMFHDSUB	2	Binary	Record subtype
API data				
24(x'18')	SMFAPIST	4	EBCDIC	Connection status **INIT** Connection initiation **TERM** Connection termination
28(x'1C')	SMFAPILA	4	Binary	Local IPv4 address
32(x'20')	SMFAPIRA	4	Binary	Remote IPv4 address
36(x'24')	SMFAPILP	2	Binary	Local port number
38(x'26')	SMFAPIRP	2	Binary	Remote port number
40(x'28')	SMFAPIIN	4	Binary	Inbound bytes (valid only for TERM records)
44(x'2C')	SMFAPIOU	4	Binary	Outbound bytes (valid only for TERM records)
48(x'30')	SMFAPIUO	2	Binary	Offset to start of an area available for user exit storage
50(x'32')	SMFAPIUL	2	Binary	User area length (See Note 1.)

Table 144. API call SMF record format (continued)

Offset	Name	Length	Format	Description
52(x'34')	SMFAPINM	8	EBCDIC	Job name for: • Interactive TSO API usage; the user's TSO user ID • Batch-submitted jobs; the name of the JOB card • Started procedures; the name of the procedure.
60(x'3C')	SMFAPIJI	8	EBCDIC	JES job identifier
68(x'44')	SMFAPIJS	4	Binary	Connection start time, in hundredths of seconds
72(x'48')	SMFAPIJD	4	Packed	Date connection started. The date is in the form of 0CYYDDDF, where C is 0 for 19yy and 1 for 20yy, DDD is the day of the year (1-365), and F is the sign. For TSO/E, it is the logon enqueue date.
76(x'4C')	SMFAPIUS	52		User area, available for user exit usage (See Note 2.)

Notes:

1. The current maximum length of the user data is 52 bytes. This value could change between TCP/IP releases.
2. The actual displacement of this area could change between TCP/IP releases. Use the values of the user area offset and the user area length fields to access this area correctly.

SMF record layout for FTP client calls

The SMF record written by FTP client calls has the format Table 145.

Table 145. FTP client SMF record format

Offset	Name	Length	Format	Description
0(x'0')	SMFHEADR	24		Standard SMF header
4(x'4')	SMFHDFLG	1	Binary	Record flag (set to 66, or x'42')
5(x'5')	SMFHDTYP	1	Binary	Record type (set to 118, or x'76')
22(x'16')	SMFHDSUB	2	Binary	Record subtype
	FTP client data			
24(x'18')	SMFFTCCM	4	EBCDIC	FTP subcommand **APPE** Append **RETR** Retrieve **STOR** Store
28(x'1C')	SMFFTCCY	4	EBCDIC	Value of the reply to the FTP command
32(x'20')	SMFFTCSA	4	Binary	Local (client) IP address (IPv4) or –1 for IPv6
36(x'24')	SMFFTCSL	4	Binary	Remote (server) IP address (IPv4) or –1 for IPv6
40(x'28')	SMFFTCCP	2	Binary	Local port
42(x'2A')	SMFFTCCF	2	Binary	Remote port
44(x'2C')		4		Reserved

Table 145. FTP client SMF record format (continued)

Offset	Name	Length	Format	Description
48(x'30')	SMFFTCSU	8	EBCDIC	Remote user ID
56(x'38')	SMFFTCFM	1	EBCDIC	Data format: **A** ASCII **E** EBCDIC **I** Image (binary) **B** Double-byte **U** UCS-2
57(x'39')	SMFFTCMO	1	EBCDIC	Transfer mode: **C** Compressed data **S** Stream data **B** Block data
58(x'3A')	SMFFTCST	1	EBCDIC	Structure: **F** File **R** Record
59(x'3B')	SMFFTCDT	1	EBCDIC	Data set type: **P** Partitioned **S** Sequential **H** HFS
60(x'3C')	SMFFTCRS	4	Binary	Start time of transmission, if applicable, in hundredths of seconds.
64(x'40')	SMFFTCRE	4	Binary	End time of transmission
68(x'44')	SMFFTCBC	4	Binary	Byte count, if applicable
72(x'48')	SMFFTCXD	1	EBCDIC	FTP ID: **C** Client
73(x'49')		3		Reserved
76(x'4C')	SMFFTCSN	44	EBCDIC	Local data set name or PDS name (for z/OS UNIX file names, only the first 44 bytes are included).
120(x'78')	SMFFTCEM	8	EBCDIC	Member name for PDS
128(x'80')		60		Reserved
188(x'BC')	SMFFTCTC	8	EBCDIC	User ID of the FTP user
196(x'C4')	SMFFTCHN	8	EBCDIC	Host ID
204(x'CC')	SMFFTCF1	2	Binary	Offset to the first z/OS UNIX file name field
206(x'CE')	SMFFTCF2	2	Binary	Offset to the second z/OS UNIX file name field
208(x'D0')	SMFFTCYF	8	Floating point Hex	Bytes transferred counter. The leftmost byte is an exponent, and other seven bytes are significant bytes.
216(x'D8')	SMFFTCIG	4	Binary	Bytes transferred, 4 GB increments. Increments with every 4 GBs of data transfer, starting from 0.

Two variable-length fields at the end of the record contain z/OS UNIX file names. The variable-length z/OS UNIX name fields have the format shown in Table 146.

Table 146. z/OS UNIX file name (variable length fields appended to end of FTP server record)

Offset	Name	Length	Format	Description
0(x'0')		2	Binary	Length of the z/OS UNIX file name
2(x'2')		*n*	EBCDIC	z/OS UNIX file name (maximum length is 1023 bytes)

SMF record layout for Telnet client calls

The SMF record written by Telnet client calls has the format shown in Table 147.

Table 147. Telnet client SMF record format

Offset	Name	Length	Format	Description
0(x'0')	SMFHEADR	24		Standard SMF header
4(x'4')	SMFHDFLG	1	Binary	Record flag (set to 66, or x'42')
5(x'5')	SMFHDTYP	1	Binary	Record type (set to 118, or x'76')
22(x'16')	SMFHDSUB	2	Binary	Record subtype
	Telnet Client data			
24(x'18')	SMFTNTCM	4	EBCDIC	Event type **LOGN** Session intiation **LOGF** Session termination
28(x'1C')		20		Reserved
48(x'30')	SMFTNTRA	4	Binary	Remote (server) IP address
52(x'34')	SMFTNTLA	4	Binary	Local (client) IP address
56(x'38')	SMFTNTST	8	EBCDIC	Started task qualifier
64(x'40')	SMFTNTHN	8	EBCDIC	NJE node name
72(x'48')		18		Reserved
90(x'5A')	SMFTNTRP	2	Binary	Remote port number
92(x'5C')	SMFTNTLP	2	Binary	Local port number

SMF record layout for TCPIPSTATISTICS

Table 148. SMF record layout for TCPIPSTATISTICS

Offset	Name	Length	Format	Description
0(x'0')	Standard SMF header	24		Standard SMF header
0(x'0')	SMFHDLEN	2	Binary	Record length
2(x'2')	SMFHDSEG	2	Binary	Segment descriptor
4(x'4')	SMFHDFLG	1	Binary	Record flag (set to 66, or x'42')
5(x'5')	SMFHDTYP	1	Binary	Record type (set to 118, or x'76')
6(x'6')	SMFHDTME	4	Binary	Time when record was written
10(x'A')	SMFHDDTE	4	Binary	Date when record was written

Table 148. SMF record layout for TCPIPSTATISTICS (continued)

Offset	Name	Length	Format	Description
14(x'E')	SMFHDSID	4	EBCDIC	System identification
18(x'12')	SMFHDSSI	2	Binary	Subsystem identification
20(x'14')	SMFHDSUB	2	Binary	Record subtype
22(x'16')		2		Reserved
24(x'18')	SMFHDSDL	2	Binary	Length of self-defining area
	Self-defining area			
26(x'1A')	SMF3Off (See Note 1.)	4	Binary	Offset of subsystem area
30(x'1E')	SMF3Len	2	Binary	Length of subsystem area
32(x'20')	SMF3Num	2	Binary	Number of subsystem areas (1)
34(x'22')	SMF3Off	4	Binary	Offset of IP area
38(x'26')	SMF3Len	2	Binary	Length of IP area
40(x'28')	SMF3Num	2	Binary	Number of IP areas (1)
42(x'2A')	SMF3Off	4	Binary	Offset of ICMP area
46(x'2E')	SMF3Len	2	Binary	Length of ICMP area
48(x'30')	SMF3Num	2	Binary	Number of ICMP areas (0)
50(x'32')	SMF3Off	4	Binary	Offset of TCP area
54(x'36')	SMF3Len	2	Binary	Length of TCP area
56(x'38')	SMF3Num	2	Binary	Number of TCP areas (1)
58(x'3A')	SMF3Off	4	Binary	Offset of UDP area
62(x'3E')	SMF3Len	2	Binary	Length of UDP area
64(x'40')	SMF3Num	2	Binary	Number of UDP areas (1)
	Subsystem ID area			
0(x'0')	SMFSubProc	8	EBCDIC	TCP/IP Procname
8(x'8')	SMFSubASID	4	Binary	TCP/IP ASID
12(x'C')	SMFSubTime	8	Binary	TCP/IP Startup TOD
20(x'14')	SMFSubFlag	4	Binary	TCP/IP SMF reason: **x'10'** Last SMF record/Shutdown **x'20'** Last SMF record/End stats **x'40'** SMF Interval record **x'80'** First SMF record
	IP area			
0(x'0')	imirecv	4	Binary	Total received datagrams
4(x'4')	imihdrer	4	Binary	Total discarded datagrams
8(x'8')	imiadrer	4	Binary	Total discarded: address errors
12('C')	imifwddg	4	Binary	Total attempts to forward datagrams
16(x'10')	imiunprt	4	Binary	Total discarded: unknown protocols
20(x'14')	imidisc	4	Binary	Total discarded: other
24(x'18')	imidelvr	4	Binary	Total delivered datagrams
28(x'1C')	imoreqst	4	Binary	Total sent datagrams

Table 148. SMF record layout for TCPIPSTATISTICS (continued)

Offset	Name	Length	Format	Description
32(x'20')	imodisc	4	Binary	Total send discarded: other
36(x'24')	imonorte	4	Binary	Total send discarded: no route
40(x'28')	imrsmtos	4	Binary	Total reassembly timeouts
44(x'2C')	imrsmreq	4	Binary	Total received: reassembly required
48(x'30')	imrsmok	4	Binary	Total datagrams reassembled
52(x'34')	imrsmfld	4	Binary	Total reassembly failed
56(x'38')	imfragok	4	Binary	Total datagrams fragmented
60(x'3C')	imfrgfld	4	Binary	Total discarded: fragments failed
64(x'40')	imrgcre	4	Binary	Total fragments generated
68(x'44')	imrtdisc	4	Binary	Total routing discards
72(x'48')	imrsmmax	4	Binary	Max active reassemblies
76(x'4C')	imrmsact	4	Binary	Num active reassemblies
80(x'50')	imrsmful	4	Binary	Discarding reassembled fragments
TCP area				
0(x'0')	tcp_RtoAlgorithm	4	Binary	Retransmit algorithm
4(x'4')	tcp_RtoMin	4	Binary	Minimum retransmit time (ms)
8(x'8')	tcp_RtoMax	4	Binary	Maximum retransmit time (ms)
12(x'C')	tcp_MaxConn	4	Binary	Maximum connections
16(x'10')	tcp_ActiveOpens	4	Binary	Active opens
20(x'14')	tcp_PassiveOpens	4	Binary	Passive Opens
24(x'18')	tcp_AttemptFails	4	Binary	Open failures
28(x'1C')	tcp_EstabResets	4	Binary	Number of resets
32(x'20')	tcp_CurrEstab	4	Binary	Number of currently established connections
36(x'24')	tcp_InSegs	4	Binary	Input segments
40(x'28')	tcp_OutSegs	4	Binary	Output segments
44(x'2C')	tcp_RetransSegs	4	Binary	Retransmitted segments
48(x'30')	tcp_InErrs	4	Binary	Input errors
52(x'34')	tcp_OutRsts	4	Binary	Number of resets
UDP area				
0(x'0')	usindgrm	4	Binary	Received UDP datagrams
4(x'4')	usnoprts	4	Binary	UDP datagrams with no ports
8(x'8')	usinerrs	4	Binary	Other UDP datagrams not received
12(x'C')	usotdgrm	4	Binary	UDP datagrams sent

Notes:

1. The same fields overlay each (offset, length, number) structure within the self-defining area. The overlay must be appropriately based to reference any single field within the self-defining area.

Appendix E. Type 119 SMF records

This topic describes the Type 119 SMF records created for several TCP/IP functions. The following information is included:

- "IPSec manual tunnel activation record (subtype 79)" on page 914
- "IPSec manual tunnel deactivation record (subtype 80)" on page 915

Mapping SMF records

In order for an application to be able to process SMF 119 records, z/OS Communications Server provides mapping macros and C header files.

Assembler applications

For assembler applications, the macro EZASMF77 (installed in SYS1.MACLIB) produces assembler level DSECTs that can be used to map the various record formats described in this topic. When invoking EZASMF77, the default value creates all the record mappings. EZBNMMPA are in tcpip.SEZANMAC and should be concatenated with SYS1.MACLIB.

To create the mapping for the interval statistic records, code the following:

```
EZASMF77 STAT=YES
```

To create the mapping of the format 119 IPSec SMF records, code the following:

```
EZASMF77 IPSEC=YES
```

Because the YES value is the default for all the EZASMF77 operands, coding EZASMF77 without any operands is equivalent to coding the following:

```
EZASMF77 FTP=YES,API=YES,TELNET=YES,HEADER=YES,STAT=YES,IPSEC=YES,PROF=YES,MAIL=YES,DVIPA=YES
```

Guideline: Code NO for any of the operands to exclude those mappings from the assembler output.

To obtain the mappings for the individual sections of profile data in the format 119 TCP/IP profile SMF event record (subtype 4), include macro EZBNMMPA from MVS data set SEZANMAC.

C/C++ applications

For C/C++ applications, the following header files provide the SMF record mappings:

ezasmf.h
: This header file provides mappings for most of the SMF records.

ezbnmmpc.h
: This header file provides the mappings for the individual sections of profile data in the SMF 119 TCP/IP profile SMF event record (subtype 4).

Both header files are installed in the SEZANMAC MVS data set and in the /usr/include file system directory.

Processing SMF records for IP security

The IPSec SMF record structures were designed to be analogous to the IPSec Network Management Interface (NMI) structures that describe the responses for IP Security information returned by this API. Management applications that currently use or plan to use the IPSec NMI should consider this when designing their applications. The analogous IP Security NMI section names are indicated under the SMF records. The IP Security NMI is described in "Network security services NMI request and response format" on page 540.

Common Type 119 SMF record format

All Type 119 SMF records are in the format shown in Table 149. For related subtypes, see "SMF 119 record subtypes."

Table 149. Records types and subtype information

Offset	Name	Length	Format	Description
0(X'0')	Standard header	24	Binary	SMF system header
0(X'0')		2	Binary	SMF record length
2(X'2')		2	Binary	Segment descriptor
4(X'4')		1	Binary	Record flag
5(X'5')		1	Binary	Record type; is set to 119(X'77').
6(X'6')		4	Binary	SMF system timestamp (is local time)
10(X'A')		4	Packed	SMF system date (is local time)
14(X'D')		4	EBCDIC	SMF system ID
18(X'12')		4	EBCDIC	SMF subsystem ID
22(X'16')		2	Binary	Record subtype
24(X'18')	Self-defining section		Binary	This section indicates how many sections follow, and their location in the record.
...	TCP/IP identification section	64	Binary	This section is present in every record; it describes the TCP/IP stack which issued the record. Its location and size are indicated by the self-defining section.
...	Record-specific data section 1	...	Binary	First record-specific data section. Its location and size are indicated by the self-defining section.
...	Record-specific data section 1, second entry	...	Binary	The self-defining section indicates how many occurrences of each record-specific data section are present in the record.
...	Record-specific data section 2 (optional)	...	Binary	Second record-specific data section.
...	...	...	Binary	...
...	Record-specific data section *n*, first entry (optional)	...	Binary	Last record-specific data section. The self-defining section indicates how many types of data sections there are.
...	...	...	Binary	...

SMF 119 record subtypes

TCP/IP collects SMF information about certain Telnet, FTP, TCP/IP stack, IKE daemon or CSSMTP activity. These records can be generated by the TCP/IP stack, the FTP and Telnet clients and server, the IKE daemon or the CSSMTP client. You can control the collection of these records by using the SMFCONFIG statements in PROFILE.TCPIP, or by using statements in the various application's configuration files. For more information about those statements, see *z/OS Communications Server: IP Configuration Reference*.

All the records described in this topic are written using record type 119 (X'77'), and standard subtype values, at offset 22 (X'16') in SMF record header, are used to

uniquely identify the type of record being collected. Table 150 correlates the subtype information to the type of record being produced.

Table 150. SMF 119 record subtype information and record type

Record subtype	Description	TCP/IP component event	Reason
1(X'1')	"TCP connection initiation record (subtype 1)" on page 750	TCP	Event
2(X'2')	"TCP connection termination record (subtype 2)" on page 751	TCP	Event
3(X'3')	"FTP client transfer completion record (subtype 3)" on page 758	FTPC	Event
4(X'4')	"TCP/IP profile event record (subtype 4)" on page 763	STACK	Event
5(X'5')	"TCP/IP statistics record (subtype 5)" on page 816	STACK	Interval
6(X'6')	"Interface statistics record (subtype 6)" on page 826	IP	Interval
7(X'7')	"Server port statistics record (subtype 7)" on page 830	STACK	Interval
8(X'8')	"TCP/IP stack start/stop record (subtype 8)" on page 832	TCP	Event
9	Reserved		
10(X'A')	"UDP socket close record (subtype 10)" on page 833	UDP	Event
11–19	Reserved		
20(X'14')	"TN3270E Telnet server SNA session initiation record (subtype 20)" on page 835	TN3270S	Event
21(X'15')	"TN3270E Telnet server SNA session termination record (subtype 21)" on page 836	TN3270S	Event
22(X'16')	"TSO Telnet client connection initiation record (subtype 22)" on page 842	TN3270C	Event
23(X'17')	"TSO Telnet client connection termination record (subtype 23)" on page 843	TN3270C	Event
24–31	Reserved		
32(X'20')	"DVIPA status change record (subtype 32)" on page 844	STACK	Event
33(X'21')	"DVIPA removed record (subtype 33)" on page 846	STACK	Event
34(X'22')	"DVIPA target added record (subtype 34)" on page 849	STACK	Event
35(X'23')	"DVIPA target removed record (subtype 35)" on page 850	STACK	Event
36(X'24')	"DVIPA target server started record (subtype 36)" on page 852	STACK	Event
37(X'25')	"DVIPA target server ended record (subtype 37)" on page 854	STACK	Event
38–47	Reserved		

Table 150. SMF 119 record subtype information and record type (continued)

Record subtype	Description	TCP/IP component event	Reason
48(X'30')	"CSSMTP configuration record (CONFIG subtype 48)" on page 855	CSSMTP	Event
49(X'31')	"CSSMTP connection record (CONNECT subtype 49)" on page 861	CSSMTP	Event
50(X'32')	"CSSMTP mail record (MAIL subtype 50)" on page 864	CSSMTP	Event
51(X'33')	"CSSMTP spool file record (SPOOL subtype 51)" on page 868	CSSMTP	Event
52(X'34')	"CSSMTP statistical record (STATS subtype 52)" on page 874	CSSMTP	Interval
53–69	Reserved		
70(X'46')	"FTP server transfer completion record (subtype 70)" on page 879	FTPS	Event
71	Reserved		
72(X'48')	"FTP server logon failure record (subtype 72)" on page 884	FTPS	Event
73(X'29')	"IPSec IKE tunnel activation and refresh record (subtype 73)" on page 888	IKE	Event
74(X'4A')	"IPSec IKE tunnel deactivation and expire record (subtype 74)" on page 894	IKE	Event
75(X'4B')	"IPSec dynamic tunnel activation and refresh record (subtype 75)" on page 897	IKE	Event
76(X'4C')	"IPSec dynamic tunnel deactivation record (subtype 76)" on page 910	IKE	Event
77(X'4D')	"IPSec dynamic tunnel added record (subtype 77)" on page 911	STACK	Event
78(X'4E')	"IPSec dynamic tunnel removed record (subtype 78)" on page 912	STACK	Event
79(X'4F')	"IPSec manual tunnel activation record (subtype 79)" on page 914	STACK	Event
80(X'50')	"IPSec manual tunnel deactivation record (subtype 80)" on page 915	STACK	Event
81-255	Reserved		

Notes:

1. The TCP/IP component indicated is the one reported in the TCP/IP identification section for each record (see the following sections).
2. The Reason indicated determines whether each record is an event record (it is flagged with a reason code of X'08'; in the TCP/IP identification section) or an interval record (it is flagged with one of the six interval reason codes in the TCP/IP identification section).

Standard data format concepts

The following concepts apply to standard data formats:

- Unless specified otherwise, all times are indicated in units of 1/100 seconds since midnight (local time). Certain select times are in MVS TOD clock format.

- All dates are indicated in packed decimal (BCD) form, with digits X'01yydddF'. If no data is available, a date of X'0000000F' is written.
- Interval durations are specified in one of two formats, indicated within the record itself. It can either be in units of 1/100 seconds or a 64-bit integer with bit 51 marking the microsecond.
- All interval-type statistics records (such as TCP/IP statistics) report interval data, rather than total data.

  This behavior for Type 119 records is a change in semantics from type 118 records, which record summary data. For example, while a type 118 record would report "bytes sent to date", a Type 119 record would report "bytes sent since the last recording interval."
- IP addresses

  Most IP addresses are in 128-bit IPv6 format. In this format, IPv4 addresses are reported in IPv4-mapped form; the 4-byte IPv4 address is preceded by 12 bytes, the first 10 of which are 0, and the last two of which are 'FF'x. IPv6 addresses appear in numeric form.

  For the following record subtypes, the IPv4 and IPv6 addresses are defined in the same 16-byte field in the record section. The IPv4 address is reported in the first 4 bytes of the field, and the IPv6 address occupies the whole field. A flag field in the record section indicates whether the field contains an IPv4 or an IPv6 address.
  - Subtype 4 TCP/IP profile
  - Subtypes 32 - 37 DVIPA
  - Subtypes 73 - 80 IPSec
- For information about AT-TLS cipher suite values, see TTLSCipherParms statement in *z/OS Communications Server: IP Configuration Reference*.

Common TCP/IP identification section

Table 151 shows a section that is present in every SMF Type 119 record produced by the TCP/IP stack. Its purpose is to identify the system and stack responsible for producing the record.

Table 151. Common TCP/IP identification section

Offset	Name	Length	Format	Description
0(X'0')	SMF119TI_SYSName	8	EBCDIC	System name from SYSNAME in IEASYSxx
8(X'8')	SMF119TI_SysplexName	8	EBCDIC	Sysplex name from SYSPLEX in COUPLExx
16(X'10')	SMF119TI_Stack	8	EBCDIC	TCP/IP stack name
24(X'18')	SMF119TI_ReleaseID	8	EBCDIC	z/OS Communications Server TCP/IP release identifier

Table 151. Common TCP/IP identification section (continued)

Offset	Name	Length	Format	Description
32(X'20')	SMF119TI_Comp	8	EBCDIC	TCP/IP subcomponent (right padded with blanks): **CSSMTP** CSSMTP client **FTPC** FTP client **FTPS** FTP server **IKE** IKE daemon **IP** IP layer **STACK** Entire TCP/IP stack **TCP** TCP layer **TN3270C** TN3270 client **TN3270S** TN3270 server **UDP** UDP layer
40(X'28')	SMF119TI_ASName	8	EBCDIC	Started task qualifier or address space name of address space that writes this SMF record
48(X'30')	SMF119TI_UserID	8	EBCDIC	User ID of security context under which this SMF record is written
56(X'38')		2	EBCDIC	Reserved
58(X'3A')	SMF119TI_ASID2	2	Binary	ASID of address space that writes this SMF record (in EZASMF77 macro).
58(X'3A')	SMF119TI_ASID	2	Binary	ASID of address space that writes this SMF record (in ezasmf.h).
60(X'3C')	SMF119TI_Reason	1	Binary	Reason for writing this SMF record: • X'C0': Interval record, more records follow • X'80': Interval record, last record in set • X'60': End-of-statistics record, more records follow • X'20': End-of-statistics record, last record in set • X'50': Shutdown starts record, more records follow • X'10': Shutdown starts record, last record in set • X'48': Event record, more records follow • X'08' : Event record, last record in set

Table 151. Common TCP/IP identification section (continued)

Offset	Name	Length	Format	Description
61(X'3D')	SMF119TI_RecordID	1	Binary	Value used by the following SMF 119 records, to correlate several physical records which contain one logical set of information. The SMF 119 record descriptions will explain when the field is used. • TCP/IP profile event record (subtype 4)
62(X'3E')		2	EBCDIC	Reserved

TCP connection initiation record (subtype 1)

The TCP connection initiation record is collected whenever a TCP connection is opened. This record contains pertinent information about the connection available at the time of its opening.

Guidelines:

- Because this information is duplicated in the TCP connection termination record, which contains additional information, you should only collect the TCP connection termination record.
- Because this record is generated for every single TCP connection, significant load can be generated on a server and rapidly fill the SMF data sets. The TCP connection termination record is collected whenever a TCP connection is closed or terminiated. This record contains all pertinent information about the connection, such as elapsed time, bytes transferred, and so on.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the TCP/IP connection initiation record, the TCP/IP stack identification section indicates TCP as the subcomponent and X'08' (event record) as the record reason.

Table 152 shows the TCP connection initiation record self-defining section:

Table 152. TCP connection initiation record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF header	24		Standard SMF header
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2).
26(X'1A')		2	Binary	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 TCP connection initiation section
40(X'28')	SMF119S1Len	2	Binary	Length of TCP connection initiation section

Table 152. TCP connection initiation record self-defining section (continued)

Offset	Name	Length	Format	Description
42(X'2A')	SMF119S1Num	2	Binary	Number of TCP connection initiation sections

Table 153 shows the TCP connection initiation specific section of this SMF record.

Table 153. TCP connection initiation specific section

Offset	Name	Length	Format	Description
0(X'0')	SMF119AP_TIRName	8	EBCDIC	TCP socket resource name (Address space name of address space that established this TCP connection)
8(X'8')	SMF119AP_TIConnID	4	Binary	TCP socket resource ID (connection ID)
12(X'C')	SMF119AP_TIRsv1	4	Binary	Reserved
16(X'10')	SMF119AP_TISubTask	4	Binary	Subtask Name (Address of MVS TCB for the task that owns this connection. Note that this is not the subtask value specified on an INITAPI call.)
20(X'14')	SMF119AP_TIRIP	16	Binary	Remote IP address at time of connection open
36(X'24')	SMF119AP_TILIP	16	Binary	Local IP address at time of connection open
52(X'34')	SMF119AP_TIRPort	2	Binary	Remote port number at time of connection open
54(X'36')	SMF119AP_TILPort	2	Binary	Local port number at time of connection open
56(X'38')	SMF119AP_TITime	4	Binary	Time of day of connection establishment
60(X'3C')	SMF119AP_TIDate	4	Packed	Date of connection establishment
64(X'40')	SMF119AP_TISTCK	8	Binary	STCK of connection establishment

TCP connection termination record (subtype 2)

The TCP connection termination record is collected whenever a TCP connection is closed or aborted. This record contains all pertinent information about the connection, such as elapsed time, bytes transferred, and so on.

Guidelines:

- Because this information duplicates all of the information contained in the TCP connection initiation record, only collect the TCP connection termination record.
- Because this record is generated for every single TCP connection, this can generate significant load on a server and rapidly fill the SMF data sets. Use care when you use it.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the TCP/IP connection termination record, the TCP/IP stack identification section indicates TCP as the subcomponent and X'08' (event record) as the record reason.

Table 154 shows the TCP connection termination self-defining section:

Table 154. TCP connection termination self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF header	24		Standard SMF header
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record
26(X'1A')		2	Binary	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 TCP connection termination section
40(X'28')	SMF119S1Len	2	Binary	Length of TCP connection termination section
42(X'2A')	SMF119S1Num	2	Binary	Number of TCP connection termination sections
44 (X'2C')	SMF119S2Off	4	Binary	Offset to TCP connection termination Telnet information section
48 (X'30')	SMF119S2Len	2	Binary	Length of TCP connection termination Telnet information section
50 (X'32')	SMF119S2Num	2	Binary	Number of TCP connection termination Telnet information sections
52 (X'34')	SMF119S3Off	4	Binary	Offset to TCP connection termination Application Transparent Transport Layer Security (AT-TLS) information section
56 (X'38')	SMF119S3Len	2	Binary	Length of TCP connection termination AT-TLS information section
58 (X'3A')	SMF119S3Num	2	Binary	Number of TCP connection termination AT-TLS information sections
60 (X'3C')	SMF119S4Off	4	Binary	Offset to TCP connection termination ApplData section
64 (X'40')	SMF119S4Len	2	Binary	Length of TCP connection termination ApplData section
66 (X'42')	SMF119S4Num	2	Binary	Number of TCP connection termination ApplData sections

Table 155 on page 753 shows the TCP connection termination specific section of this SMF record.

Table 155. TCP connection termination section

Offset	Name	Length	Format	Description
0(X'0')	SMF119AP_TTRName	8	EBCDIC	TCP socket resource name (Address space name of address space that closed this TCP connection)
8(X'8')	SMF119AP_TTConnID	4	Binary	TCP socket resource ID (connection ID)
12(X'C')	SMF119AP_TTTTLSCS	1	Binary	AT-TLS connection status: • X'01': Connection is not secure • X'02': Connection handshake in progress • X'03': Connection is secure
13(X'D')	SMF119AP_TTTTLSPS	1	Binary	AT-TLS Policy Status: • X'00': Policy status is not known • X'01': AT-TLS function off • X'02': No policy defined for connection • X'03': Policy defined for connection; AT-TLS not enabled • X'04': Policy defined for connection; AT-TLS enabled • X'05': Policy defined for connection; AT-TLS enabled and Application Controlled

Table 155. TCP connection termination section (continued)

Offset	Name	Length	Format	Description
14(X'E')	SMF119AP_TTTermCode	1	Binary	Reason code for connection termination: • X'11': Error occurred during a send using FRCA(AFPA), possibly because the stack is terminating. • X'12': A persistent socket used by FRCA (AFPA) was closed by a FIN. • X'21': The connection was terminated because the stack was terminating. • X'22': Last stack that can own the dynamic VIPA bound to the socket is terminating • X'31': Intrusion detection found the connection to be malicious and closed the connection. • X'32': Connection was denied because of a NetAccess rule. • X'33': ACK received in lastack state. • X'41': The connection was terminated because of an administrator action (for example, using Netstat DRop/-D command or the NMI API). • X'42': The connection was terminated because the local IP address bound by the application has been deleted from the stack. • X'51': The connection from a client was terminated because the application closed the socket before performing an accept(). • X'52': The application using the socket, closed the connection using a close(). • X'53': A pascal routine issued an orderly close request. • X'54': A pascal routine issued a disconnect. • X'55': An error occurred during a pascal accept. • X'61': The connection was terminated because the client sent a reset.

Table 155. TCP connection termination section (continued)

Offset	Name	Length	Format	Description
14(X'E') (continued)				• X'71': The connection was closed because the same packet was being re-transmitted multiple times. • X'72': The connection was closed because the TCP window was reduced to 0 and multiple window probes were not acknowledged. • X'73': The connection was closed because multiple keepalive probes were not acknowledged. • X'74': The connection was terminated because the stack timed out waiting for a fin in the finwait-2 state. • X'75': The connection was terminated because a global TCP stall attack was detected. • X'76': The connection was terminated because a TCP queue size attack was detected.
15(X'F')	SMF119AP_TTRsv2	1	Binary	Reserved
16(X'10')	SMF119AP_TTSubtask	4	Binary	Subtask Name (Address of MVS TCB for the task that owns this connection. This is not the subtask value specified on an INITAPI call.)
20(X'14')	SMF119AP_TTSTime	4	Binary	Time of connection establishment
24(X'18')	SMF119AP_TTSDate	4	Packed	Date of connection establishment
28(X'1C')	SMF119AP_TTETime	4	Binary	Time connection entered TIMEWAIT or LASTACK state.
32(X'20')	SMF119AP_TTEDate	4	Packed	Date connection entered TIMEWAIT or LASTACK state.
36(X'24')	SMF119AP_TTRIP	16	Binary	Remote IP address at time of connection close.
52(X'34')	SMF119AP_TTLIP	16	Binary	Local IP address at time of connection close.
68(X'44')	SMF119AP_TTRPort	2	Binary	Remote port number at time of connection close.
70(X'46')	SMF119AP_TTLPort	2	Binary	Local port number at time of connection close.
72(X'48')	SMF119AP_TTInBytes	8	Binary	Inbound byte count.
80(X'50')	SMF119AP_TTOutBytes	8	Binary	Outbound byte count.
88(X'58')	SMF119AP_TTSWS	4	Binary	Send window size at time of connection close.
92(X'5C')	SMF119AP_TTMSWS	4	Binary	Maximum send window size.
96(X'60')	SMF119AP_TTCWS	4	Binary	Congestion window size at time of connection close.
100(X'64')	SMF119AP_TTSMS	4	Binary	Send segment size at time of connection close.

Table 155. TCP connection termination section (continued)

Offset	Name	Length	Format	Description
104(X'68')	SMF119AP_TTRTT	4	Binary	Round trip time in milliseconds at time of connection close.
108(X'6C')	SMF119AP_TTRVA	4	Binary	Round trip time variance estimator at time of connection close, in milliseconds.
112(X'70')	SMF119AP_TTStatus	1	Binary	Socket status: • X'00': Passive Open (this is a server socket) • X'01': Active Open (this is a client socket)
113(X'71')	SMF119AP_TTTOS	1	Binary	Type of Service (ToS) used by this connection.
114(X'72')	SMF119AP_TTXRT	2	Binary	Number of times retransmission was required for this connection.
116(X'74')	SMF119AP_TTProf	32	EBCDIC	Service profile name.
148(X'94')	SMF119AP_TTPol	32	EBCDIC	Service Policy name at the time of connection close.
180(X'B4')	SMF119AP_TTInSeg	8	Binary	Inbound segment count.
188(X'BC')	SMF119AP_TTOutSeg	8	Binary	Outbound segment count.
196(X'C4')	SMF119AP_TTSSTCK	8	Binary	MVS TOD clock value at time of connection establishment.
204(X'CC')	SMF119AP_TTESTCK	8	Binary	MVS TOD clock value at time connection entered TIMEWAIT or LASTACK state.
212(X'D4')	SMF119AP_TTDupAcksRcvd	4	Binary	Total Number of DUP ACKs received on the connection.

Table 156 shows the TCP connection termination Telnet specific section of this SMF record. This section is present only when the given TCP connection represented a TN3270 Telnet connection.

Table 156. TCP connection termination Telnet section

Offset	Name	Length	Format	Description
0(X'0')	SMF119AP_TTTelLUName	8	EBCDIC	LU name
8(X'8')	SMF119AP_TTTelAppl	8	EBCDIC	Target application name
16(X'10')	SMF119AP_TTTelLogmode	8	EBCDIC	Logmode name
24(X'18')	SMF119AP_TTTelStatus	4	Binary	Status word: • x80000000: Definite response mode • x40000000: The connection is being performance monitored • x00000004: TN3270E mode • x00000002: TN3270 mode • x00000001: Line mode

Table 156. TCP connection termination Telnet section (continued)

Offset	Name	Length	Format	Description
28(X'1C')	SMF119AP_TTTelTermCode	1	Binary	Reason code for closing connection. The socket must be accessible to the TN3270 server to record a reason. (for example, SMF119AP_TTTermCode for this record is X'52'.) See the description of EZZ6034I in *z/OS Communications Server: IP Messages Volume 4 (EZZ, SNM)* for a list of reason codes and their descriptions.
29(X'1D')	SMF119AP_TTTelRsv	3	Binary	Reserved

Table 157 shows the TCP connection termination AT-TLS-specific section of this SMF record.

Restriction: This section is present only when the given TCP connection was secured by AT-TLS (SMF119AP-TTTTLSCS is X'03').

Table 157. TCP connection termination AT-TLS section

Offset	Name	Length	Format	Description
0 (X'0')	SMF119AP_TTTTLSSP	2	Binary	AT-TLS SSL Protocol: • X'0200': SSL Version 2 • X'0300': SSL Version 3 • X'0301': TLS Version 1.0 • X'0302': TLS Version 1.1
2(X'2')	SMF119AP_TTTTLSNC	2	EBCDIC	AT-TLS Negotiated Cipher
4(X'4')	SMF119AP_TTTTLSST	1	Binary	AT-TLS Security Type: • X'01': Client • X'02': Server • X'03': Server with client authentication, ClientAuthType = PassThru • X'04': Server with client authentication, ClientAuthType = Full • X'05': Server with client authentication, ClientAuthType = Required • X'06': Server with client authentication, ClientAuthType = SAFCheck
5(X'5')	SMF119AP_TTTTLSFP	1	Binary	FIPS 140 status • X'00': FIPS 140 off • X'01': FIPS 140 on
6(X'6')	SMF119AP_TTTTLSRSV1	2	Binary	Reserved

Table 158 on page 758 shows the TCP connection termination application-specific section of this SMF record. The ApplData section provides the application-specific information that is associated with a TCP connection. See "SIOCSAPPLDATA

IOCTL" on page 713 for information about how applications can use the SIOCSAPPLDATA ioctl to associate application-specific data with a TCP connection.

This section is present only when the given TCP connection has application data associated with it.

The content of this field is determined by the application that owns the connection. For z/OS Communications Server applications, see the information about Appendix F, "Application data," on page 917 for an explanation of the layout, format, and meaning of this field. For other applications, see the documentation that is supplied by the application. This field typically contains all printable EBCDIC characters, although some applications might include some binary data.

Table 158. TCP connection termination ApplData section

Offset	Name	Length	Format	Description
0 (X'0')	SMF119AP_TTAPPLDATA	40	Varies	For z/OS Communications Server applications, see Appendix F, "Application data," on page 917 for an explanation of the layout, format, and meaning of this field. For other applications, see the documentation that is supplied by the application.

FTP client transfer completion record (subtype 3)

The FTP client transfer completion record is collected when the z/OS FTP client completes processing of one of the following FTP file transfer operations: file appending, file storage, or file retrieval. A common format for the record is used for each FTP file transfer operations, so the record contains an indication of which operation was performed. The record also contains optional sections provided when the file name involved in the transfer operation was an MVS or z/OS UNIX filename, as well as when the FTP operation traversed a SOCKS server in the path from the z/OS client to the FTP server.

The Type 119 FTP client transfer completion record is collected at the same point in file transfer processing as the equivalent Type 118 FTP client SMF record.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the FTP client transfer completion record, the TCP/IP stack identification section indicates FTPC as the subcomponent and X'08' (event record) as the record reason.

Table 159 shows the FTP client transfer completion record self-defining section:

Table 159. FTP client transfer completion record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 3(X'3')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (6)
26(X'1A')		2	Binary	Reserved

Table 159. FTP client transfer completion 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 transfer completion section
40(X'28')	SMF119S1Len	2	Binary	Length of FTP client transfer completion section
42(X'2A')	SMF119S1Num	2	Binary	Number of FTP client transfer completion sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to FTP client transfer completion associated data set name section
48(X'30')	SMF119S2Len	2	Binary	Length of FTP client transfer completion associated data set name section
50(X'32')	SMF119S2Num	2	Binary	Number of FTP client transfer completion associated data set name sections
52(X'34')	SMF119S3Off	4	Binary	Offset to FTP client transfer completion SOCKS section
56(X'38')	SMF119S3Len	2	Binary	Length of FTP client transfer completion SOCKS section
58(X'3A')	SMF119S3Num	2	Binary	Number of FTP client transfer completion SOCKS sections
60 (X'3C')	SMF119S4Off	4	Binary	Offset to FTP client transfer completion security section
64 (X'40')	SMF119S4Len	2	Binary	Length of FTP client transfer completion security section
66 (X'42')	SMF119S4Num	2	Binary	Number of FTP client transfer completion security sections
68 (X'44')	SMF119S5Off	4	Binary	Offset to FTP client transfer completion user name section
72 (X'48')	SMF119S5Len	2	Binary	Length of FTP client transfer completion user name section
74 (X'4A')	SMF119S5Num	2	Binary	Number of FTP client transfer completion user name sections

Table 160 shows the FTP client transfer completion specific section of this SMF record.

Table 160. FTP client transfer completion record section

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FCCmd	4	EBCDIC	FTP command (according to RFC 959)
4(X'4')	SMF119FT_FCFType	4	EBCDIC	Local file type (SEQ, JES, or SQL)

Table 160. FTP client transfer completion record section (continued)

Offset	Name	Length	Format	Description
8(X'8')	SMF119FT_FCDRIP	16	Binary	Remote IP address (data connection)
24(X'18')	SMF119FT_FCDLIP	16	Binary	Local IP address (data connection)
40(X'28')	SMF119FT_FCDRPort	2	Binary	Remote port number (data connection)
42(X'2A')	SMF119FT_FCDLPort	2	Binary	Local port number (data connection)
44(X'2C')	SMF119FT_FCCRIP	16	Binary	Remote IP address (control connection)
60 (X'3C')	SMF119FT_FCCLIP	16	Binary	Local IP address (control connection)
76(X'4C')	SMF119FT_FCCRPort	2	Binary	Remote port number (control connection)
78 (X'4E')	SMF119FT_FCCLPort	2	Binary	Local port number (control connection)
80 (X'50')	SMF119FT_FCRUser	8	EBCDIC	User ID (login name) on server
88(X'58')	SMF119FT_FCLUser	8	EBCDIC	Local User ID
96(X'60')	SMF119FT_FCType	1	EBCDIC	Data format: • A: ASCII • E: EBCDIC • I: Image • B: Double-byte • U: UCS-2
97(X'61')	SMF119FT_FCMode	1	EBCDIC	Transfer mode: • B: Block • C: Compressed • S: Stream
98(X'62')	SMF119FT_FCStruct	1	EBCDIC	Structure: • F: File • R: Record
99(X'63')	SMF119FT_FCDSType	1	EBCDIC	Data set type: • S: SEQ • P: PDS • H: HFS
100(X'64')	SMF119FT_FCSTime	4	Binary	Transmission start time of day
104(X'68')	SMF119FT_FCSDate	4	Packed	Transmission start date
108(X'6C')	SMF119FT_FCETime	4	Binary	Transmission end time of day
112(X'70')	SMF119FT_FCEDate	4	Packed	Transmission end date
116(X'74')	SMF119FT_FCDur	4	Binary	File transmission duration in units of 1/100 seconds
120(X'78')	SMF119FT_FCBytes	8	Binary	Transmission byte count; 64-bit integer
128(X'80')	SMF119FT_FCLReply	4	EBCDIC	Last server reply (3-digit RFC 959 code, left justified)
132(X'84')	SMF119FT_FCM1	8	EBCDIC	PDS member name
140(X'8C')	SMF119FT_FCHostname	8	EBCDIC	Host name
148(X'94')	SMF119FT_FCRS	8	EBCDIC	Reserved for abnormal end info

Table 160. FTP client transfer completion record section (continued)

Offset	Name	Length	Format	Description
156(X'9C')	SMF119FT_FCBytesFloat	8	Floating point hex	z/OS floating point format for transmission byte count
164 (X'A4')	SMF119FT_FCCConnID	4	Binary	TCP connection ID of FTP control connection
168 (X'A8')	SMF119FT_FCDConnID	4	Binary	TCP connection ID of FTP data connection, or 0 if no data connection is active

Table 161 shows the FTP client transfer completion associated data set name section. This section represents the MVS or z/OS UNIX data set name associated with the file transfer.

Table 161. FTP client transfer completion associated data set name section

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FCFileName	*n*	EBCDIC	MVS or z/OS UNIX data set name associated with the file transfer operation. Use the Data Set Type field information in the FTP client transfer completion section to determine the type of file name represented by this value.

Table 162 shows the FTP client transfer completion SOCKS section. This section is present when the FTP operation traverses a SOCKS server on the path between the z/OS FTP client and FTP server.

Table 162. FTP client transfer completion SOCKS section

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FCCIP	16	Binary	IP address of SOCKS server for control connection
16(X'10')	SMF119FT_FCCPort	2	Binary	SOCKS port number (control connection)
18(X'12')	SMF119FT_FCCProt	1	Binary	SOCKS protocol version (control connection): • X'01': SOCKS Version 4 • X'02': SOCKS Version 5

Table 163 shows the FTP client transfer completion security section:

Table 163. FTP client transfer completion security section

Offset	Name	Length	Format	Description
0 (X'0')	SMF119FT_FCMechanism	1	EBCDIC	Protection Mechanism: • N: None • T: TLS • G: GSSAPI • A: AT_TLS

Table 163. FTP client transfer completion security section (continued)

Offset	Name	Length	Format	Description
1 (X'1')	SMF119FT_FCCProtect	1	EBCDIC	Control connection Protection Level: • N: None • C: Clear • S: Safe • P: Private
2 (X'2')	SMF119FT_FCDProtect	1	EBCDIC	Data connection Protection Level: • N: None • C: Clear • S: Safe • P: Private
3 (X'3')	SMF119FT_FCLoginMech	1	EBCDIC	Login Method: • U: Login method is not defined for the FTP client
4 (X'4')	SMF119FT_FCProtoLevel	8	EBCDIC	Protocol level (only present if protocol mechanism is TLS or AT-TLS). Possible values are: • SSLV2 • SSLV3 • TLSV1 • TLSV1.1
12 (X'C')	SMF119FT_FCCipherSpec	20	EBCDIC	Cipher specification (only present 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, TLSV1, or TLSV1.1: • 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

Table 163. FTP client transfer completion security section (continued)

Offset	Name	Length	Format	Description
36(X'24')	SMF119FT_FCCipher	2	EBCDIC	Hexadecimal value of cipher specification (present only if protocol mechanism is TLS or AT-TLS).
38(X'26')	SMF119FT_FCFips140	1	Binary	FIPS 140 status • X'00': FIPS 140 off • X'01': FIPS 140 on

Table 164 shows the FTP client transfer completion user name section.

Table 164. FTP client transfer completion user name section

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FCIUserID	*n*	EBCDIC	User name or user ID used to log into the FTP server.

TCP/IP profile event record (subtype 4)

The TCP/IP profile record provides profile information for the TCP/IP stack. The first or only record always contains the following sections:

- SMF header
- Self-defining section with 21 section triplets
- TCP/IP identification section
- Profile information common section
- Profile information data set name section

See Table 165 on page 766 for a list of all the sections of information that can be provided in this SMF record.

This record is created as an event record during the following processing:

- During the initialization of the stack. In this case, the record contains the complete profile information for the stack.
- If the profile is changed by the use of the VARY TCIP,,OBEYFILE command. In this case, the record contains only changed profile information.
- The NMTP_PICOSecChanged flag bits in the profile information common section indicate which sections actually contain changed information.
- In the self-defining section, the triplet field values are zero for sections for which no information was changed, or for those sections which all the information was deleted from the stack's configuration.
- If deprecated profile statements were specified in the VARY TCPIP,,OBEYFILE command data set, field NMTP_PicoDepChanged indicates which statements were processed. If only deprecated statements were processed, the profile information common and data set name sections are the only sections of information provided in the SMF record. See Table 166 on page 769 for an explanation of deprecated profile statements.
- For the sections that changed, the section in the SMF record contains all of the information for the section. For example, if a network interface was added, the whole interface section is included in the SMF record. Applications need to compare the interface section in the new record with the interface section in the previous record to determine which interface was added.

- If the profile data set referenced by the VARY TCIP,,OBEYFILE command changed the SMFCONFIG setting from PROFILE to NOPROFILE, one final SMF event record is created and written to the MVS SMF data sets to record this change.
- If the profile data set referenced by the VARY TCIP,,OBEYFILE command changed the NETMONITOR SMFSERVICE setting from PROFILE to NOPROFILE, one final SMF event record is created and written to the real-time SMF data network management interface (NMI) to record this change. For more information about the real-time SMF NMI, see "Real-time TCP/IP network monitoring NMI" on page 564.

The SMF record might be created even if some errors occurred during processing the VARY TCIP,,OBEYFILE command. Application programs that process these records must compare the sections of changed information to the previous profile settings to determine if profile changes actually occurred.

Relationship to GetProfile Callable NMI

The information provided by this record is also available from the TCP/IP Callable NMI by invoking this NMI with the GetProfile (NWMPROFILE TYPE) request. The GetProfile request always returns complete profile information. For more information about the GetProfile request output, see "TCP/IP NMI response format" on page 653. There are some minor differences in the information between this SMF record and the GetProfile request output.

Management section
Both the SMF record and the GetProfile request provide a flag bit indicating whether the community name parameter was specified on the SACONFIG profile statement. But, for security reasons, the actual community name value is only returned by the GetProfile request.

Continuing the SMF record

If the information for the record exceeds 32 746 bytes, additional TCP/IP profile records are created to provide all the information. For sections with multiple entries, all the entries that fit in the current record are provided in the current record. Any entries that did not fit in the current record are provided in a subsequent record, along with additional sections that did not fit in the current record.

The value in the SMF119TI_Reason field indicates whether the record is complete or incomplete. If the record is incomplete, it is followed by an additional record or records. The Profile information sections (common and data set name) are included in the first record only in the set of SMF records. The intermediate and final SMF records do not contain the Profile information sections. They contain the TCP/IP identification section only plus the additional sections of configured information.

Two-phase SMF record creation for VIPADYNAMIC/ ENDVIPADYNAMIC profile statement information

You can use the VIPADYNAMIC and ENDVIPADYNAMIC profile statements and their substatements to configure dynamic VIPA and Sysplex Distributor support in the TCP/IP stack. The following sections in the TCP/IP profile SMF record provide information about this configuration:

Dynamic VIPA addresses
Provides configuration information from VIPABACKUP, VIPADEFINE, and VIPARANGE substatements

Dynamic VIPA routing
Provides configuration information from the VIPAROUTE substatement.

Distributed dynamic VIPA
Provides configuration information from the VIPADISTRIBUTE substatement.

Processing of these configuration statements occurs after the normal profile configuration processing, so more than one SMF record is needed to provide the configured information. When a profile data set contains these configuration statements, the resulting TCP/IP profile SMF records are created in two phases:

Phase one
During normal profile configuration processing, the first TCP/IP profile SMF record is created. It contains the following sections of information:

TCP/IP identification section
- The value in field SMF119TI_Reason indicates that the record is incomplete.
- Field SMF119TI_RecordID contains a correlator value, so that you can correlate this first record with the additional record or records that are created during Phase two.

Profile information common section
Field NMTP_PICOSecChanged indicates the sections affected by the statements in the profile data set

Sections of configured information
If other profile statements other than the VIPADYNAMIC and ENDVIPADYNAMIC statement block were specified in the profile data set, their information is provided in the Phase one SMF record.

Phase two
When the VIPADYNAMIC and ENDVIPADYNAMIC profile statements are processed, an additional record or records is created to provide the configured information. These records contain the following sections of information:

TCP/IP identification section
- The value in field SMF119TI_Reason indicates whether the record is complete or incomplete. If more than one additional record is needed to support all the configured information, all the additional records except the final record indicate that the record is still incomplete. In the final record, field SMF119TI_Reason indicates that the record is complete.
- Field SMF119TI_RecordID contains a correlator value, so that you can correlate the record written during Phase one with the additional record or records which are created during Phase two.

Sections of dynamic VIPA and Sysplex Distributor configured information
If other profile statements other than the VIPADYNAMIC and ENDVIPADYNAMIC statement block were specified in the profile data set, their information has already been provided in the Phase one SMF record.

Cancelled configuration information

In some cases, configuration changes are cancelled. For example, if the TCP/IP stack is not currently joined to the sysplex group, and a VARY TCPIP,,OBEYFILE command is issued to change the stack's dynamic VIPA configuration, the requested configuration changes are cancelled. A TCP/IP Profile SMF record is created with the following attributes:

- The NMTP_PICOSecChanged flag bits in the Profile information common section indicate the sections that would have been affected by the configuration change.
- There is one section for each record section that would have been affected by the configuration changes. A flag bit is set in the section to indicate that the requested changes were cancelled. The description of the flag bit explains the reasons why the changes were cancelled.

Configuration changes can be cancelled for the following sections:

- Dynamic VIPA addresses
- Dynamic VIPA routing
- Distributed dynamic VIPA

Data format concepts

The following concepts apply to the fields in the record sections:

- All fields with a binary format are set to binary zeros if there is no value for the field.
- All fields with an EBCDIC format are set to EBCDIC blanks (X'40') if there is no value for the field.
- The value in all fields that use an EBCDIC format is padded with trailing blanks.

TCP/IP profile record self-defining section

Table 165 shows the TCP/IP profile record self-defining section:

Table 165. TCP/IP profile record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (21)
26(X'1A')		2	Binary	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 Profile information common section
40(X'28')	SMF119S1Len	2	Binary	Length of Profile information common section
42(X'2A')	SMF119S1Num	2	Binary	Number of Profile information common sections

Table 165. TCP/IP profile record self-defining section (continued)

Offset	Name	Length	Format	Description
44(X'2C')	SMF119S2Off	4	Binary	Offset of Profile information common data set name section
48(X'30')	SMF119S2Len	2	Binary	Length of Profile information common data set name section
50(X'32')	SMF119S2Num	2	Binary	Number of Profile information common data set name sections
52(X'34')	SMF119S3Off	4	Binary	Offset to Autolog procedure section
56(X'38')	SMF119S3Len	2	Binary	Length of Autolog procedure section
58(X'3A')	SMF119S3Num	2	Binary	Number of Autolog procedure sections
60 (X'3C')	SMF119S4Off	4	Binary	Offset to IPv4 IP configuration section
64 (X'40')	SMF119S4Len	2	Binary	Length of IPv4 IP configuration section
66 (X'42')	SMF119S4Num	2	Binary	Number of IPv4 IP configuration sections
68 (X'44')	SMF119S5Off	4	Binary	Offset to IPv6 IP configuration section
72 (X'48')	SMF119S5Len	2	Binary	Length of IPv6 IP configuration section
74 (X'4A')	SMF119S5Num	2	Binary	Number of IPv6 IP configuration sections
76(X'4C')	SMF119S6Off	4	Binary	Offset to TCP configuration section
80(X'50')	SMF119S6Len	2	Binary	Length of TCP configuration section
82(X'52')	SMF119S6Num	2	Binary	Number of TCP configuration sections
84(X'54')	SMF119S7Off	4	Binary	Offset to UDP configuration section
88(X'58')	SMF119S7Len	2	Binary	Length of UDP configuration section
90(X'5A')	SMF119S7Num	2	Binary	Number of UDP configuration sections
92(X'5C')	SMF119S8Off	4	Binary	Offset to Global configuration section
96(X'60')	SMF119S8Len	2	Binary	Length of Global configuration section
98(X'62')	SMF119S8Num	2	Binary	Number of Global configuration sections
100(X'64')	SMF119S9Off	4	Binary	Offset to Port reservation section
104(X'68')	SMF119S9Len	2	Binary	Length of Port reservation section
106(X'6A')	SMF119S9Num	2	Binary	Number of Port reservation sections
108(X'6C')	SMF119S10Off	4	Binary	Offset to Interface section
112(X'70')	SMF119S10Len	2	Binary	Length of interface section
114(X'72')	SMF119S10Num	2	Binary	Number of interface sections
116(X'74')	SMF119S11Off	4	Binary	Offset to IPv6 address section
120(X'78')	SMF119S11Len	2	Binary	Length of IPv6 address section
122(X'7A')	SMF119S11Num	2	Binary	Number of IPv6 address sections

Table 165. TCP/IP profile record self-defining section (continued)

Offset	Name	Length	Format	Description
124(X'7C')	SMF119S12Off	4	Binary	Offset to routing section
128(X'80')	SMF119S12Len	2	Binary	Length of routing section
130(X'82')	SMF119S12Num	2	Binary	Number of routing sections
138(X'8A')	SMF119S13Num	2	Binary	Number of source IP address sections
140(X'8C')	SMF119S14Off	4	Binary	Offset to management section
144(X'90')	SMF119S14Len	2	Binary	Length of management section
146(X'92')	SMF119S14Num	2	Binary	Number of Management sections
148(X'94')	SMF119S15Off	4	Binary	Offset to IPSec common section
152(X'98')	SMF119S15Len	2	Binary	Length of IPSec common section
154(X'9A')	SMF119S15Num	2	Binary	Number of IPSec common sections
156(X'9C')	SMF119S16Off	4	Binary	Offset to IPSec default rules section
160(X'A0')	SMF119S16Len	2	Binary	Length of IPSec default rules section
162(X'A2')	SMF119S16Num	2	Binary	Number of IPSec default rules sections
164(X'A4')	SMF119S17Off	4	Binary	Offset to network access section
168(X'A8')	SMF119S17Len	2	Binary	Length of network access section
170(X'AA')	SMF119S17Num	2	Binary	Number of network access sections
172(X'AC')	SMF119S18Off	4	Binary	Offset to dynamic VIPA (DVIPA) address section
176(X'B0')	SMF119S18Len	2	Binary	Length of dynamic VIPA (DVIPA) address section
178(X'B2')	SMF119S18Num	2	Binary	Number of dynamic VIPA (DVIPA) address sections
180(X'B4')	SMF119S19Off	4	Binary	Offset to DVIPA routing section
184(X'B8')	SMF119S19Len	2	Binary	Length of DVIPA routing section
186(X'BA')	SMF119S190Num	2	Binary	Number of DVIPA routing sections
188(X'BC')	SMF119S20Off	4	Binary	Offset to distributed DVIPA section
192(X'C0')	SMF119S20Len	2	Binary	Length of distributed DVIPA section
194(X'C2')	SMF119S20Num	2	Binary	Number of distributed DVIPA sections
196(X'C4')	SMF119S21Off	4	Binary	Offset to default address selection policy section
200(X'C8')	SMF119S21Len	2	Binary	Length of default address selection policy section
202(X'CA')	SMF119S21Num	2	Binary	Number of default address selection policy sections

TCP/IP profile record TCP/IP stack identification section

"Common TCP/IP identification section" on page 748 shows the contents of the TCP/IP stack identification section. For the TCP/IP profile record, the TCP/IP stack identification section indicates STACK as the subcomponent. The record reason field is set to one of the following bit values:

- X'08' (event record)

- X'48' (event record incomplete, more records follow)

TCP/IP profile record profile information common section

This section provides some general TCP/IP stack values and information about the last time the profile was changed. There is only one of these sections in the record.

The NMTP_PICODepStmts and NMTP_PICODepChanged fields

Flags in these fields are set when deprecated profile statements are processed. Deprecated profile statements are those whose function is considered to be non-strategic. There are also some network interface types that are considered to be non-strategic. The flags for deprecated interface, IPv4 IP address, or route changes in the NMTP_PICODepStmts and NMTP_PICODepChanged fields are set when any profile statement, except the PRIMARYINTERFACE statement, is processed for one of the following non-strategic network interface types:

- ATM (includes ATMARPSV, ATMLIS, ATMPVC profile statements)
- CLAW
- CTC
- HCH
- LCS
- MPCIPA/IPAQTR
- MPCOSA
- SNA LU0 and LU6.2
- X.25
- CDLC

The PRIMARYINTERFACE setting is provided in the IPv4 configuration section. Non-strategic network interface types are supported for this setting.

Table 166 shows the profile information common section.

Table 166. Profile information common section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_PICOEye	4	EBCDIC	PICO eyecatcher
4(X'4')	NMTP_PICOStartTime	8	Binary	Time TCP/IP stack was started (TOD clock value)
12(X'C')	NMTP_PICOStartDate	4	Packed	Date TCP/IP stack was started
16(X'10')	NMTP_PICOChangeTime	8	Binary	Time the TCP/IP stack's profile was last changed (TOD clock value) by a VARY TCPIP,,OBEYFILE command.
24(X'18')	NMTP_PICOChangeDate	4	Packed	Date the TCP/IP stack's profile was last changed by a VARY TCPIP,,OBEYFILE command.
28(X'1C')	NMTP_PICOChangeRsn	1	Binary	Reason for last profile change: • NMTP_PICOChangeRsn_OBEYFILE(1) - VARY TCPIP,,OBEYFILE command

Table 166. Profile information common section (continued)

Offset	Name	Length	Format	Description
29(X'1D')	NMTP_PICOFlags	1	Binary	Miscellaneous flags: **X'80', NMTP_PICOProfComplete:** Record contains complete profile information. If set, the record was created either during TCP/IP initialization or, by way of VARY TCPIP,,OBEYFILE where SMF TCP/IP profile record support was activated. Field NMTP_PICOSecChanged is zero if the record was created during initialization.
30(X'1E')		2	Binary	Reserved
32(X'20')	NMTP_PICODepStmts	2	Binary	Flag that indicates which deprecated profile statements were specified in the initial profile: **X'80000000', NMTP_PICODepStIntf** DEVICE/LINK/ BSDROUTINGPARMS for non-strategic interfaces. **X'40000000', NMTP_PICODepStHome:** HOME for non-strategic interfaces. **X'20000000', NMTP_PICODepStRoute:** GATEWAY or BEGINROUTES for non-strategic interfaces. **X'10000000', NMTP_PICODepStSMF:** SMFCONFIG TYPE118 or SMFPARMS **X'08000000', NMTP_PICODepStTrans:** TRANSLATE **X'04000000', NMTP_PICODepStSMParms:** VIPASMPARMS

Table 166. Profile information common section (continued)

Offset	Name	Length	Format	Description
34(X'22')	NMTP_PICODepChanged	2	Binary	Flag which indicates which deprecated profile statements were changed: **X'80000000', NMTP_PICODepChIntf:** DEVICE/LINK/ BSDROUTINGPARMS for non-strategic interfaces. **X'40000000', NMTP_PICODepChHome:** HOME for non-strategic interfaces. **X'20000000', NMTP_PICODepChRoute:** GATEWAY or BEGINROUTES for non-strategic interfaces. **X'10000000', NMTP_PICODepChSMF:** SMFCONFIG TYPE118 or SMFPARMS **X'08000000', NMTP_PICODepChTrans:** TRANSLATE **X'04000000', NMTP_PICODepChSMParms:** VIPASMPARMS
36(X'24')	NMTP_PICOSecChanged	4	Binary	Flag that indicates which sections were changed. The following flags are only set if the record was created due to a profile change. • X'80000000', NMTP_PICOSecAutolog • X'40000000', NMTP_PICOSecV4Cfg • X'20000000', NMTP_PICOSecV6Cfg • X'10000000', NMTP_PICOSecTCPCfg • X'08000000', NMTP_PICOSecUDPCfg • X'04000000', NMTP_PICOSecGblCfg • X'02000000', NMTP_PICOSecPort • X'01000000', NMTP_PICOSecIntf • X'00800000', NMTP_PICOSecIPA6 • X'00400000', NMTP_PICOSecRoute • X'00200000', NMTP_PICOSecSrcip • X'00100000', NMTP_PICOSecMgmt • X'00080000', NMTP_PICOSecIPSecCm • X'00040000', NMTP_PICOSecIPSecRules • X'00020000', NMTP_PICOSecNetacc • X'00008000', NMTP_PICOSecDVCfg • X'00004000', NMTP_PICOSecDVRoute • X'00002000', NMTP_PICOSecDistDV • X'00001000', NMTP_PICOSecDasp
40(X'28')	NMTP_PICOConsName	8	EBCDIC	Name of console from which VARY TCPIP,,OBEYFILE command was issued.

Table 166. Profile information common section (continued)

Offset	Name	Length	Format	Description
48(X'30')	NMTP_PICOSysplexGrpName	8	EBCDIC	Sysplex group name. The value is created when the TCP/IP stack joins the sysplex group. Because the stack joins the sysplex group after the initial profile is processed, the SMF record created during initial profile processing does not contain the sysplex group name. If the TCP/IP stack has never joined the sysplex group since it was initialized, this field is set to zeros.
56(X'38')	NMTP_PICOUserToken	80	Binary	RACF user security token of user responsible for change. For a mapping of the fields, see the RUTKN data area in *z/OS Security Server RACF Data Areas*.

TCP/IP profile record profile information data set name section

This section provides a list of the data sets used for the initial profile and the data sets used for the last VARY TCPIP,,OBEYFILE command processing. There can be multiple sections in the record, one per data set name.

Table 167 shows the Profile information data set name section.

Table 167. Profile information data set name section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_PIDSEye	4	EBCDIC	PIDS eyecatcher
4(X'4')	NMTP_PIDSFlag	1	Binary	Indicates whether data set was used for the initial profile or for a profile change, and whether it was the main profile data set or was specified on an INCLUDE profile statement. **X'80', NMTP_PIDSChange** Change data set. If set, the data set was used to change the profile. If not set, the data set was used for the initial profile. **X'40', NMTP_PIDSInclude** Include data set. If set, the data set was specified on an INCLUDE statement. If not set, the data set was the main data set.
5(X'5')		1	Binary	Reserved
6(X'6')	NMTP_PIDSName	54	EBCDIC	The data set name value is padded with trailing blanks.

TCP/IP profile record autolog procedure section

This section provides a list of the started procedures to be autologged and their attributes. There can be multiple sections in the record, one per autologged procedure.

Table 168 shows the Autolog procedure section.

Table 168. Autolog procedure section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_ALPREye	4	EBCDIC	ALPR eyecatcher
4(X'4')	NMTP_ALPRName	8	EBCDIC	Procedure name to be started. The procedure name value is padded with trailing blanks.
12(X'C')	NMTP_ALPRJobName	8	EBCDIC	Job name assigned to reserved port for the started procedure. The job name value is padded with trailing blanks.
20(X'14')	NMTP_ALPROptions	2	Binary	Procedure options: • X'8000', NMTP_ALPRDelayDvipa: DELAYSTART DVIPA • X'4000', NMTP_ALPRDelayTtls: DELAYSTART TTLS
22(X'16')		2	Binary	Reserved
24(X'18')	NMTP_ALPRParmStr	115	EBCDIC	The parmstring value, padded with trailing blanks.
139(X'8B')	NMTP_ALPRWaitTime	1	Binary	Wait time

TCP/IP profile record IPv4 configuration section

This section provides IPv4 layer configuration information from the IPCONFIG, ARPAGE, and PRIMARYINTERFACE profile statements. There is only one of these sections in the record.

Table 169 shows the IPv4 configuration section.

Table 169. IPv4 configuration section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_V4CFEye	4	EBCDIC	V4CF eyecatcher

Table 169. IPv4 configuration section (continued)

Offset	Name	Length	Format	Description
4(X'4')	NMTP_V4CFFlags	4	Binary	IPCONFIG flags: **X'80000000'** NMTP_V4CFCLAWDblNoop: If set, the CLAW channel programs have 2 NOP CCWs at the end. **X'40000000'** NMTP_V4CFDatagramFwd: If set, the stack is forwarding datagrams and field NMTP_V4CFFwdMultipPkt indicates if a multipath per packet algorithm is being used for forwarded packets. If not set, the stack is not forwarding datagrams. **X'20000000'** NMTP_V4CFFwdMultipPkt: This flag is only valid if flag NMTP_V4CFDatagramFwd is set. If the NMTP_V4CFFwdMultipPkt flag is set, the stack is forwarding datagrams using a multipath per packet algorithm. If not set, the stack is not using a multipath algorithm when forwarding datagrams. **X'10000000'** NMTP_V4CFDynamicXcf: If set, dynamic XCF interfaces are defined and the following fields contain dynamic XCF configured values: • NMTP_V4CFDynXcfAddr • NMTP_V4CFDynXcfCostMetric • NMTP_V4CFDynXcfMask • NMTP_V4CFDynXcfSecClass

Table 169. IPv4 configuration section (continued)

Offset	Name	Length	Format	Description
4(X'4') (Cont)				**X'08000000'** NMTP_V4CFFormatLong: If set, the Netstat command displays the report output in long format. This flag is always set for IPv6-enabled stacks. **X'04000000'** NMTP_ V4CFIgnoreRedirectCfg: If set, IGNOREREDIRECT was specified on the IPCONFIG profile statement. **X'02000000'** NMTP_ V4CFIgnoreRedirectAct: If set, the stack is ignoring ICMP redirects and the NMTP_V4CFIgnRedirectRsn field indicates the reason why this setting is in effect. **X'01000000'** NMTP_V4CFIPSecurity: If set, IP security is enabled. **X'00800000'** NMTP_V4CFIQDIORouting: If set, IQDIO routing is enabled. **X'0040000'** NMTP_V4CFMultipPerConn: If set, the stack is using a multipath per connection routing selection algorithm for outbound IP traffic. **X'0020000'** NMTP_V4CFMultipPerPkt: If set, the stack is using a multipath per packet routing selection algorithm for outbound IP traffic. **X'00100000'** NMTP_V4CFPathMtuDisc: If set, Path MTU discovery is in effect. **X'00080000'** NMTP_V4CFSourceVipa: If set, the stack uses the appropriate VIPA IP address as the source IP address for outbound packets.

Table 169. IPv4 configuration section (continued)

Offset	Name	Length	Format	Description
4(X'4') (Cont)				**X'00040000'** NMTP_V4CFStopClawErr: If set, the stack stops channel programs when a CLAW error is detected. **X'00020000'** NMTP_V4CFSysplexRouting: If set, the stack communicates interface changes to the workload manager. **X'00010000'** NMTP_V4CFTCPSourceVipa: If set, and NMTP_V4CFSourceVipa is also set, the stack uses the address in field V4CFTcpSrcVipaAddr as the source IP address for outbound TCP connections. **X'00008000'** NMTP_V4CFQDIOAcc: If set, the QDIO accelerator function is enabled. If flag NMTP_V4CFDatagramFwd is set, then the function is enabled for all IP traffic. If flag NMTP_V4CFDatagramFwd is not set, the function is enabled for Sysplex Distributor IP traffic only. **X'00004000'** NMTP_V4CFChkOffload: If set, IP, UDP and TCP checksum processing is offloaded to an OSA-Express feature. **X'00002000'** NMTP_V4CFSegOffload: If set, TCP segmentation is offloaded to an OSA-Express feature.
8(X'8')	NMTP_V4CFArpTimeout	4	Binary	ARP cache timeout in seconds. If the value was configured, then it was either specified on the ARPAGE statement, or on the ARPTO parameter of the IPCONFIG statement.
12(X'C')	NMTP_V4CFDevRetry	4	Binary	Device retry duration in seconds
16(X'10')	NMTP_V4CFTcpSrcVipaAddr	4	Binary	VIPA source IP address for outbound TCP connections. If flags NMTP_V4CFSourceVipa and NMTP_V4CFTCPSourceVipa are set, this address is used as the source IP address.
20(X'14')	NMTP_V4CFDynXcfAddr	4	Binary	Dynamic XCF IP address. This field is only valid if the NMTP_V4CFDynamicXcf flag is set.
24(X'18')	NMTP_V4CFDynXcfCostMetric	1	Binary	Dynamic XCF cost metric. This field is only valid if the NMTP_V4CFDynamicXcf flag is set.

Table 169. IPv4 configuration section (continued)

Offset	Name	Length	Format	Description
25(X'19')	NMTP_V4CFDynXcfMask	1	Binary	Dynamic XCF number of mask bits. This field is only valid if the NMTP_V4CFDynamicXcf flag is set.
26(X'1A')	NMTP_V4CFDynXcfSecClass	1	Binary	Dynamic XCF security class. This field is only valid if the NMTP_V4CFDynamicXcf flag is set.
27(X'1B')	NMTP_V4CFQDIOPriority	1	Binary	IQDIO routing priority. This field is only valid if either the NMTP_IQDIORouting flag or the NMTP_QDIOAcc flag is set.
28(X'1C')	NMTP_V4CFIgnRedirectRsn	1	Binary	For one of the following reasons is why the NMTP_ V4CFIgnoreRedirectAct flag is set: • NMTP_V4CFIgnRedRsn_CFG(1) - Set by configuration • NMTP_V4CFIgnRedRsn_OMP(2) - Set due to OMPROUTE • NMTP_V4CFIgnRedRsn_IDS(3) - Set due to IDS ICMP redirect policy This field is only valid if the NMTP_ V4CFIgnoreRedirectAct flag is set.
29(X'1D')	NMTP_V4CFReasmTimeout	1	Binary	Reassembly timeout in seconds
30(X'1E')	NMTP_V4CFTTL	1	Binary	Time to live
31(X'1F')		1	Binary	Reserved
32(X'20')	NMTP_V4CFPrimaryIntfName	16	EBCDIC	Name of the primary interface. The primary interface could have been configured on a PRIMARYINTERFACE profile statement, or the stack could have selected a default primary interface.

TCP/IP profile record IPv6 configuration section

This section provides IPv6 layer configuration information from the IPCONFIG6 profile statement. There is only one of these sections in the record.

Table 170 shows the IPv6 configuration section.

Table 170. TCP/IP profile record IPv6 configuration section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_V6CFEye	4	EBCDIC	V6CF eyecatcher

Table 170. TCP/IP profile record IPv6 configuration section (continued)

Offset	Name	Length	Format	Description
4(X'4')	NMTP_V6CFFlags	4	Binary	IPCONFIG6 Flags: **X'80000000', NMTP_V6CFDatagramFwd:** If set, the stack is forwarding datagrams and field NMTP_V6CFFwdMultipPkt indicates if a multipath per packet algorithm is being used for forwarded packets. If not set, the stack is not forwarding datagrams **X'40000000', NMTP_V6CFFwdMultipPkt:** This flag is only valid if flag NMTP_V6CFDatagramFwd is set. If the NMTP_V6CFFwdMultipPkt flag is set, the stack is forwarding datagrams using a multipath per packet algorithm. If not set, the stack is not using a multipath algorithm when forwarding datagrams. **X'20000000', NMTP_V6CFDynamicXcf** If set, dynamic XCF interfaces are defined and the following fields contain dynamic XCF configured values: • NMTP_V6CFDynXcfAddr • NMTP_V6CFDynXcfPfxRteLen • NMTP_V6CFDynXcfSecClass **X'10000000', NMTP_V6CFDynXcfIntfIDFlg:** If set, field NMTP_V6CFDynXcfIntfID contains the specified interface ID value. **X'08000000', NMTP_V6CFDynXcfSrcVipaIfNameFlg:** If set, field NMTP_V6CFDynXcfSrcVipaIfName contains the specified source VIPA interface name. **X'04000000', NMTP_ V6CFIgnoreRedirectCfg:** If set, IGNOREREDIRECT was specified on the IPCONFIG6 profile statement. **X'02000000', NMTP_ V6CFIgnoreRedirectAct:** If set, the stack is ignoring ICMPv6 redirects and the NMTP_V6CFIgnRedirectRsn field indicates the reason why this setting is in effect **X'01000000', NMTP_ V6CFIgnoreRtrHopLimit:** If set, the stack is ignoring hop limits received in router advertisements.

Table 170. TCP/IP profile record IPv6 configuration section (continued)

Offset	Name	Length	Format	Description
4(X'4') (Cont)				**X'00800000', NMTP_V6CFIPSecurity:** If set, IP security is enabled. **X'00400000', NMTP_V6CFMultipPerConn:** If set, the stack is using a multipath per connection routing selection algorithm for outbound IP traffic. **X'00200000', NMTP_V6CFMultipPerPkt:** If set, the stack is using a multipath per packet routing selection algorithm for outbound IP traffic. **X'00100000', NMTP_V6CFSourceVipa:** If set, the TCP/IP stack uses the appropriate VIPA IP address as the source IP address for outbound packets **X'00080000', NMTP_V6CFTCPSourceVipa:** If set, and NMTP_V6CFSourceVipa is also set, the stack uses the interface in field V6CFTcpSrcVipaIntfName to determine the source IP address for outbound TCP connections. **X'00040000', NMTP_V6CFTempAddrs:** If set, the TCP/IP stack generates IPv6 temporary addresses for IPAQENET6 OSA-Express QDIO interfaces for which stateless address autoconfiguration is enabled. When this flag is set, the following fields contain life time values for the generated addresses: • NMTP_V6CFTempAddrsPrefLifeTime • NMTP_V6CFTempAddrsValidLifeTime **X'00020000'** NMTP_V6CFChkOffload: If set, UDP and TCP checksum processing is offloaded to an OSA-Express feature. **X'00010000'** NMTP_V6CFSegOffload: If set, TCP segmentation is offloaded to an OSA-Express feature.
8(X'8')	NMTP_V6CFDynXcfIntfID	8	Binary	Dynamic XCF interface ID. This field is only valid if the NMTP_V6CFDynXcfIntfIDFlg flag is set.
16(X'10')	NMTP_V6CFDynXcfAddr	16	Binary	Dynamic XCF IP address. This field is only valid if the NMTP_V6CFDynamicXcf flag is set.
32(X'20')	NMTP_V6CFDynXcfSrcVipaIntfName	16	EBCDIC	Dynamic XCF source VIPA interface name. This field is only valid if the NMTP_V6CFDynXcfSrcVipaIfNameFlg flag is set.
48(X'30')	NMTP_V6CFTcpSrcVipaIntfName	16	EBCDIC	The VIPA interface name that is used for source IP address selection for outbound TCP connections. This field is valid only if flags NMTP_V6CFSourceVipa and NMTP_V6CFTCPSourceVipa are set.
64(X'40')	NMTP_V6CFDynXcfPfxRteLen	1	Binary	Dynamic XCF prefix route length. This field is only valid if the NMTP_V6CFDynamicXcf flag is set. If a prefix route length was not specified, then the value is zero.
65(X'41')	NMTP_V6CFDynXcfSecClass	1	Binary	Dynamic XCF security class. This field is valid only if the NMTP_V6CFDynamicXcf flag is set.

Table 170. TCP/IP profile record IPv6 configuration section (continued)

Offset	Name	Length	Format	Description
66(X'42')	NMTP_V6CFHopLimit	1	Binary	Hop limit for outbound packets.
67(X'43')	NMTP_V6CFIcmpErrLimit	1	Binary	Number of ICMPv6 error messges sent per second to a particular IPv6 destination.
68(X'44')	NMTP_V6CFIgnRedirectRsn	1	Binary	The following are reasons that the NMTP_V6CFIgnoreRedirectAct flag is set: • NMTP_V6CFIgnRedRsn_CFG(1) - Set by configuration • NMTP_V6CFIgnRedRsn_OMP(2) - Set due to OMPROUTE • NMTP_V6CFIgnRedRsn_IDS(3) - Set due to IDS ICMPv6 redirect policy This field is valid only if the NMTP_V6CFIgnoreRedirectAct flag is set.
69(X'45')	NMTP_V6CFOSMSecClass	1	Binary	OSM security class. This field is valid only when flag NMTP_V6CFIPSecurity is set.
70(X'46')		2	Binary	Reserved
72(X'48')	NMTP_V6CFTempAddrsPrefLifeTime	2	Binary	Preferred life time for temporary addresses, specified in hours. This field is valid only if the NMTP_V6CFTempAddrs flag is set.
74(X'4A')	NMTP_V6CFTempAddrsValidLifeTime	2	Binary	Valid life time for temporary addresses, specified in hours. This field is valid only if the NMTP_V6CFTempAddrs flag is set.

TCP/IP profile record TCP configuration section

This section provides TCP layer configuration information from the TCPCONFIG and SOMAXCONN profile statements. There is only one of these sections in the record.

Table 171 shows the TCP layer configuration section.

Table 171. TCP layer configuration section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_TCCFEye	4	EBCDIC	TCCF eyecatcher

Table 171. TCP layer configuration section (continued)

Offset	Name	Length	Format	Description
4(X'4')	NMTP_TCCFFlags	2	Binary	Flags: **X'8000', NMTP_TCCFDelayAcks:** If set, an acknowledgment is delayed when a packet is received for this port, or range of ports, with the PUSH bit on in the TCP header. If not set, the acknowledgment is returned immediately. **X'4000', NMTP_TCCFRestrictLowPorts:** If set, access to TCP port numbers 1-1 023 are restricted. **X'2000', NMTP_TCCFSendGarbage:** If set, keepalive packets contain one byte of random data. If not set, keepalive packets contain no data. **X'1000', NMTP_TCCFTimeStamp:** If set, the TCP layer engages in TCP timestamp negotiation during connection setup. **X'0800', NMTP_TCCFTtls:** If set, the AT-TLS function is active.
6(X'6')	NMTP_TCCFFinWait2Time	2	Binary	The number of seconds a TCP connection should remain in the FINWAIT2 state.
8(X'8')	NMTP_TCCFInterval	2	Binary	The default TCP keepalive interval, in minutes.
10(X'A')		2	Binary	Reserved
12(X'C')	NMTP_TCCFSoMaxConn	4	Binary	The maximum number of connection requests queued for any listening socket.
16(X'10')	NMTP_TCCFMaxRcvBufSize	4	Binary	The maximum receive buffer size, in bytes, that an application can set using the Setsockopt socket function call.
20(X'14')	NMTP_TCCFRcvBufSize	4	Binary	The default receive buffer size, in bytes, for applications which do not set a size using the Setsockopt socket function call.
24(X'18')	NMTP_TCCFSendBufSize	4	Binary	The default send buffer size, in bytes, for applications that do not set a size using the Setsockopt socket function call.

TCP/IP profile record UDP configuration section

This section provides TCP/IP profile record UDP configuration section. There is only one of these sections in the record.

Table 172 shows the TCP/IP profile record UDP configuration section.

Table 172. TCP/IP profile record UDP configuration section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_UDCFEye	4	EBCDIC	UDCF eyecatcher
4(X'4')	NMTP_UDCFFlags	1	Binary	Flags: **X'80', NMTP_UDCFRestrictLowPorts:** If set, access to UDP port numbers 1-1023 is restricted. **X'40', NMTP_UDCFChkSum:** If set, the UDP layer performs checksum processing. **X'20', NMTP_UDCFQueueLimit:** If set, UDP limits queued incoming datagrams to 2000 per socket.
5(X'5')		3	Binary	Reserved
8(X'8')	NMTP_UDCFRcvBufSize	2	Binary	The default UDP receive buffer size, in bytes, for applications that do not set a size using the Setsockopt socket function call.
10(X'A')	NMTP_UDCFSendBufSize	2	Binary	The default UDP send buffer size, in bytes, for applications that do not set a size using the Setsockopt socket function call.

TCP/IP profile record Global configuration section

This section provides Global configuration information from the GLOBALCONFIG profile statement. There is only one of these sections in the record.

Table 173 shows the TCP/IP profile record Global configuration section.

Table 173. TCP/IP profile record Global configuration section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_GBCFEye	4	EBCDIC	GBCF eyecatcher

Table 173. TCP/IP profile record Global configuration section (continued)

Offset	Name	Length	Format	Description
4(X'4')	NMTP_GBCFFlags	2	Binary	Flags: **X'8000', NMTP_GBCFExpBindPortRange:** If set, fields NMTP_GBCFExpBindPortRangeBegNum and NMTP_GBCFExpBindPortRangeEndNum contain the beginning and ending port numbers of the range of reserved TCP ports in the sysplex. **X'4000', NMTP_GBCFIqdMultiWrite:** If set, multiple write support is enabled for HiperSockets interfaces. **X'2000', NMTP_GBCFMlsCheckTerminate:** If set, the stack terminates if multi-level secure configuration inconsistencies are encountered. **X'1000', NMTP_GBCFSegOffload:** If set, TCP segmentation is offloaded to an OSA-Express feature. **Guideline:** This flag is deprecated. Use NMTP_V4CFSegOffload instead. **X'0800', NMTP_GBCFTcpipStats:** If set, several counters are written to the CFGPRINT DD data set when the TCP/IP stack terminates. **X'0400', NMTP_GBCFZiip:** If set, field NMTP_GBCFZiipOptions indicates for which workloads CPU cycles are displaced to a zIIP. **X'0200', NMTP_GBCFWlmPriorityQ:** If set, the following fields indicate the OSA-Express QDIO priority values that are assigned for packets associated with WLM service classes and for forwarded packets according to the control values for the WLMPRIORITYQ parameter: • NMTP_GBCFWPQCV0Pri • NMTP_GBCFWPQCV1Pri • NMTP_GBCFWPQCV2Pri • NMTP_GBCFWPQCV3Pri • NMTP_GBCFWPQCV4Pri • NMTP_GBCFWPQCV5Pri • NMTP_GBCFWPQCV6Pri • NMTP_GBCFWPQFwdPri

Table 173. TCP/IP profile record Global configuration section (continued)

Offset	Name	Length	Format	Description
6(X'6')	NMTP_GBCFSysMonOptions	2	Binary	The following are sysplex monitor subparameter settings: **X'8000', NMTP_GBCFSysMonAutoRejoin:** If set, the stack automatically rejoins the sysplex group after problems that caused it to leave the sysplex group are resolved. **X'4000', NMTP_GBCFSysMonDelayJoin:** If set, the stack delays joining the sysplex group until OMPROUTE is active. **X'2000', NMTP_GBCFSysMonDynRoute:** If set, the TCP/IP stack monitors the presence of dynamic routes over those network interfaces for which the MONSYSPLEX parameter was specified. This setting is dynamically changed if the MONINTERFACE or NOMONINTERFACE subparameters are specified. **X'1000', NMTP_GBCFSysMonMonIntf:** If set, the TCP/IP stack monitors the status of network interfaces for which the MONSYSPLEX parameter was specified. **X'0800', NMTP_GBCFSysMonRecovery:** If set, the TCP/IP stack issues error messagess, leaves the sysplex group, and deletes all DVIPA interfaces when a sysplex problem is detected. **X'0400', NMTP_GBCFSysMonNoJoin:** If set, the TCP/IP stack does not join the sysplex group until the V TCPIP,,SYSPLEX,JOINGROUP command is issued.
8(X'8')	NMTP_GBCFIqdVlanId	2	Binary	VLAN ID for the dynamic XCF HiperSockets interface. If not specified the value is 0.
10(X'A')	NMTP_GBCFSysWlmPoll	1	Binary	The number of seconds used by the sysplex distributor and its target servers, when polling WLM for new weight values.
11(X'B')	NMTP_GBCFZiipOptions	1	Binary	Workloads whose CPU cycles should be displaced to a zIIP. This field is only valid if the NMTP_GBCFZiip flag is set. The following are valid values: **X'80', NMTP_GBCFZiipIPSecurity:** If set, CPU cycles for IPSec workloads are displaced to a zIIP, when possible. **X'40', NMTP_GBCFZiipIqdioMultiWrite:** If set, CPU cycles for large TCP outbound messages are displaced to a zIIP
12(X'C')	NMTP_GBCFSysMonTimerSecs	2	Binary	The number of seconds used by the sysplex monitor function to react to problems with needed sysplex resources.
14(X'E')	NMTP_GBCFXcfGroupId	2	EBCDIC	The 2-digit suffix used to generate the sysplex group name that the TCP/IP stack joins. If not specified the value is zero.
16(X'10')	NMTP_GBCFExpBindPortRangeBegNum	2	Binary	If flag NMTP_GBCFExpBindPortRange is set, this field contains the beginning port number in the reserved range.

Table 173. TCP/IP profile record Global configuration section (continued)

Offset	Name	Length	Format	Description
18(X'12')	NMTP_GBCFExpBindPortRangeEndNum	2	Binary	If flag NMTP_GBCFExpBindPortRange is set, this field contains the ending port number in the reserved range.
20(X'14')	NMTP_GBCFMaxRecs	4	Binary	Configured maximum records value for the D TCPIP,,NETSTAT command. The value range is 1 - 65 535. The value 65 536 indicates that the * (asterisk) value was specified. This means all records.
24(X'18')	NMTP_GBCFEcsaLimit	4	Binary	The maximum ECSA storage size in bytes that can be used by the TCP/IP stack.
28(X'1C')	NMTP_GBCFPoolLimit	4	Binary	The maximum private storage size in bytes that can be used in the TCP/IP address space.
32(X'20')	NMTP_GBCFWPQCV0Pri	1	Binary	The OSA-Express QDIO priority value that is assigned to packets represented by control value 0. This field is valid only if flag NMTP_GBCFWlmPriorityQ is set.
33(X'21')	NMTP_GBCFWPQCV1Pri	1	Binary	The OSA-Express QDIO priority value that is assigned to packets represented by control value 1. This field is valid only if flag NMTP_GBCFWlmPriorityQ is set.
34(X'22')	NMTP_GBCFWPQCV2Pri	1	Binary	The OSA-Express QDIO priority value that is assigned to packets represented by control value 2. This field is valid only if flag NMTP_GBCFWlmPriorityQ is set.
35(X'23')	NMTP_GBCFWPQCV3Pri	1	Binary	The OSA-Express QDIO priority value that is assigned to packets represented by control value 3. This field is valid only if flag NMTP_GBCFWlmPriorityQ is set.
36(X'24')	NMTP_GBCFWPQCV4Pri	1	Binary	The OSA-Express QDIO priority value that is assigned to packets represented by control value 4. This field is valid only if flag NMTP_GBCFWlmPriorityQ is set.
37(X'25')	NMTP_GBCFWPQCV5Pri	1	Binary	The OSA-Express QDIO priority value that is assigned to packets represented by control value 5. This field is valid only if flag NMTP_GBCFWlmPriorityQ is set.
38(X'26')	NMTP_GBCFWPQCV6Pri	1	Binary	The OSA-Express QDIO priority value that is assigned to packets represented by control value 6. This field is valid only if flag NMTP_GBCFWlmPriorityQ is set.
39(X'27')	NMTP_GBCFWPQFwdPri	1	Binary	The OSA-Express QDIO priority value that is assigned to forwarded packets. This field is valid only if flag NMTP_GBCFWlmPriorityQ is set.
40(X'28')	NMTP_GBCFAutoIQDX	1	Binary	AutoIQDX settings. If no flag bits are set, the NOAUTOIQDX parameter value is in effect. **X'02', NMTP_GBCFAutoIQDX_NoLargeData:** If this flag bit is set, dynamic IQDX interfaces are used for all eligible traffic, except for TCP data traffic that is sent with socket transmissions of 32 K or larger. **X'01', NMTP_GBCFAutoIQDX_AllTraffic:** If this flag bit is set, dynamic IQDX interfaces are used for all eligible traffic to the intraensemble data network.
41(X'29')		7	Binary	Reserved

TCP/IP profile record Port section

This section provides information from the PORT and PORTRANGE profile statements, regarding reserved ports and access to unreserved ports. There can be multiple sections in the record, one per PORT or PORTRANGE profile statement.

Table 174 shows the TCP/IP profile record port section.

Table 174. TCP/IP profile record port section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_PORTEye	4	EBCDIC	PORT eyecatcher
4(X'4')	NMTP_PORTFlags	1	Binary	Flags: **X'80'** NMTP_PORTIPv6: If set, the BIND parameter was specified with an IPv6 IP address. **X'40'** NMTP_PORTRange: If set, this entry represents a range of reserved ports. **X'20'** NMTP_PORTUnrsv: If set, this entry applies to unreserved ports. For unreserved port entries: • Field NMTP_PORTBegNum is zero • Flag field NMTP_PORTUnrsvOptions provides settings specific to unreserved ports. **X'10'** NMTP_PORTTCP: If set, this entry applies to TCP applications. If this flag is not set, the entry applies to UDP applications.
5(X'5')	NMTP_PORTUseType	1	Binary	Type of use for the port or ports: **NMTP_PORTUTReserved(1)** None of the ports can be used by any user for the protocol (TCP or UDP) specified on this entry. This type only applies to reserved port entries. **NMTP_PORTUTAuthport(2)** The ports can only be used by the FTP server, when the server is configured to use PASSIVEDATAPORTS. This type only applies to reserved port entries which were reserved as a range (flag NMTP_PORTRange is set). **NMTP_PORTUTJobname(3)** The specified or unreserved port(s) can only be used based on an MVS job name value. If this use type value is set, then field NMTP_PORTJobName contains the job name value.

Table 174. TCP/IP profile record port section (continued)

Offset	Name	Length	Format	Description
6(X'6')	NMTP_PORTRsvOptions	2	Binary	If this is a reserved port entry and field NMTP_PORTUseType is set to NMTP_PORTUTJobname, this field contains the options for reserved ports. **X'8000'** NMTP_PORTRAutolog: If set, autolog monitoring is in effect for this port or range of ports. If not set, autolog monitoring is not in effect for this port. **X'4000'** NMTP_PORTRDelayAcks: If set, an acknowledgment is delayed when a packet is received for this port, or range of ports, with the PUSH bit on in the TCP header. If not set, the acknowledgment is returned immediately. **X'2000'** NMTP_PORTRSharePort: If set, TCP connections can be distributed to multiple listeners, listening on the same combination of port and interface. **X'1000'** NMTP_PORTRSharePortWlm: If set, TCP connections can be distributed to multiple listeners, listening on the same combination of port and interface, using WLM server-specific recommendations. **X'0800'** NMTP_PORTRBind: If set, the BIND parameter was specified for the port entry, and fields NMTP_PORTBindAddr4 or NMTP_PORTBindAddr6 contain the specified IP address. **X'0400'** NMTP_PORTRSaf: If set, a SAF resource name was specified for the port entry, and field NMTP_PORTSafName contains the name.
8(X'8')	NMTP_PORTBegNum	2	Binary	Contains one of the following values: • The reserved port number, if this is a reserved port entry and flag NMTP_PORTRange is not set. • The beginning reserved port number in the range, if this is a reserved port entry and flag NMTP_PORTRange is set. • Zeros, if this is an unreserved port entry (flag NMTP_PORTUnrsv is set).
10(X'A')	NMTP_PORTEndNum	2	Binary	If flag NMTP_PORTUnrsv is not set, this field contains one of the following values: • If flag NMTP_PORTRange is not set, this field is set to zero. • If flag NMTP_PORTRange is set, this field contains the ending reserved port number in the range.

Table 174. TCP/IP profile record port section (continued)

Offset	Name	Length	Format	Description
12(X'C')	NMTP_PORTUnrsvOptions	1	Binary	Options for unreserved ports. These flags are only set for unreserved port entries (flag NMTP_PORTUnrsv is set in field NMTP_PORTFlags): **X'80'** NMTP_PORTUDeny: If set, access to unreserved ports is denied for the protocol (TCP or UDP) specified on this entry. **X'40'** NMTP_PORTUSaf: If set, a SAF resource name was specified for the port entry, and field NMTP_PORTSafName contains the name. **X'20'** NMTP_PORTUWhenListen: If set, access to the port is checked when a TCP server application issues a Listen socket function call involving a user-specified unreserved port. **X'10'** NMTP_PORTUWhenBind: If set, access to the port is checked when an application issues a Bind socket function call involving a user-specified unreserved port.
		3	Binary	Reserved
16(X'10')	NMTP_PORTJobName	8	EBCDIC	If the NMTP_PORTUseType value is NMTP_PORTUTJobname, this field contains the MVS job name value associated with the port entry, padded with trailing blanks.
24(X'18')	NMTP_PORTSafName	8	EBCDIC	If flags NMTP_PORTRSaf or NMTP_PORTUSaf are set, this field contains the SAF resource name, padded with trailing blanks.
32(X'20')	NMTP_PORTBindAddr4	4	Binary	If flag NMTP_PORTRBind is set in the NMTP_PORTRsvOptions field, this field contains one of the following values: • If the NMTP_PORTIPv6 flag bit is not set, this field contains the IPv4 IP address specified on the BIND parameter. • If the NMTP_PORTIPv6 flag bit is set, this field contains the IPv6 IP address specified on the BIND parameter.
32(X'20')	NMTP_PORTBindAddr6	16	Binary	

TCP/IP profile record interface section

This section provides network interface information from the DEVICE, LINK, HOME, BSDROUTINGPARMS, and INTERFACE profile statements. For IPv4 interfaces, the IP address is included in the interface information. Only the subnet mask value from the BSDROUTINGPARMS statement is included in the interface information. For IPv6 interfaces, the IP addresses are provided in the IPv6 address section.

There can be multiple sections in the record, one per interface. Information from DEVICE, LINK, HOME, and BSDROUTINGPARMS statements for an interface is

combined into one section. If more than one additional IPv4 loopback IP address has been configured, there are multiple sections for the IPv4 loopback interface, one per additional IP address.

Information for only the following types of network interfaces is provided in this section:

Loopback
The loopback section is provided only if additional loopback IP addresses besides the default address, 127.0.0.1, have been configured.

OSA-Express QDIO Ethernet
MPCIPA/IPAQENET or IPAQENET6

HiperSockets
MPCIPA/IPAQIDIO or IPAQIDIO6

Static MPC Point-to-point
MPCPTP or MPCPTP6

Static VIPA
VIRTUAL or VIRTUAL6

Information for dynamic XCF and dynamic VIPA interfaces is not supported in this section. Information for dynamic XCF interfaces can be found in the IPv4 and IPv6 configuration sections. Information for dynamic VIPA interfaces can be found in the dynamic VIPA address section.

If other types of network interfaces are defined to the TCP/IP stack, their presence is indicated by a flag bit in the NMTP_PICODepStmts and NMTP_PICODepChanged fields of the profile information common section.

Table 175 shows the TCP/IP profile record interface section.

Table 175. TCP/IP profile record interface section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_INTFEye	4	EBCDIC	INTF eyecatcher

Table 175. TCP/IP profile record interface section (continued)

Offset	Name	Length	Format	Description
4(X'4')	NMTP_INTFFlags	4	Binary	Flags: **X'80000000', NMTP_INTFIPv6:** IPv6 indicator. If set, this entry is an IPv6 interface, otherwise this entry is an IPv4 interface. **X'40000000', NMTP_INTFDefIntf:** If set, the interface was defined by the INTERFACE statement; otherwise, the interface was defined by DEVICE and LINK statements. **X'20000000', NMTP_INTFIntfIDFlg:** If set, an IPv6 interface ID was specified. Field NMTP_INTFIntfID contains the interface ID value. **X'10000000', NMTP_INTFAutoRestart:** This flag only applies to non-VIRTUAL interfaces defined by DEVICE and LINK profile statements. If set, either AUTORESTART was specified or, the interface is using the same OSA-Express port, MPCPTP TRLE, or HiperSockets CHPID as an IPv6 interface, so the AUTORESTART parameter has been set by default. **X'08000000', NMTP_INTFIpBcast:** If set, IPBCAST was specified. **X'04000000', NMTP_INTFVlanIDFlg:** If set, VLANID was specified. Field NMTP_INTFVlanID contains the VLAN ID value. **X'02000000', NMTP_INTFMonSysplex:** If set, MONSYSPLEX was specified. **X'01000000', NMTP_INTFDynVlanReg:** If set, DYNVLANREG was specified. **X'00800000', NMTP_INTFVmac:** If set, VMAC was specified. Field NMTP_VmacAddr contains the virtual MAC address. **X'00400000', NMTP_INTFVmacAddrFlg:** If set, the VMAC parameter was specified with a virtual MAC address. If not set, the VMAC parameter was specified without a virtual MAC address. The OSA-Express QDIO feature generates the virtual MAC address. Field NMTP_VmacAddr contains the virtual MAC address.

Table 175. TCP/IP profile record interface section (continued)

Offset	Name	Length	Format	Description
4(X'4') (Cont)				**X'00200000', NMTP_INTFVmacRtLcl:** If set, VMAC was specified with the ROUTELCL subparameter. If not set, and flag NMTP_INTFVmac is set, then the ROUTEALL subparameter is in effect. **X'00100000', NMTP_INTFCheckSum:** If set, inbound checksum calculation is being performed. This flag only applies to MPCPTP interfaces. **X'00080000', NMTP_INTFSrcVipaIfNameFlg:** If set, SOURCEVIPAINTERFACE was specified. Field NMTP_INTFSrcVipaIntfName contains the specified source VIPA interface name. **X'00040000', NMTP_INTFTempPrefix:** If set, TEMPPREFIX was specified. Field NMTP_INTFTempPfxType indicates the type of IPv6 temporary address which was requested. **X'00020000', NMTP_INTFIsolate:** If set, ISOLATE was specified. This flag only applies to IPAQENET interfaces defined by the INTERFACE profile statement and to IPAQENET6 interfaces. **X'00010000', NMTP_INTFOptLatMode:** Indicates whether optimized latency mode (OLM parameter) was requested or is in effect. If set, and the interface is not active, the OLM parameter was specified for the interface. If set, and the interface is active, then the OLM setting is in effect for the interface. This flag applies to only IPAQENET interfaces defined by the INTERFACE profile statement and to IPAQENET6 interfaces. **X'00008000', NMTP_INTFChpIDFlg:** If set, an optional CHPID value was specified for an interface that was defined by the INTERFACE statement. The CHPID value is in NMTP_INTFChpID field.

Table 175. TCP/IP profile record interface section (continued)

Offset	Name	Length	Format	Description
8(X'8')	NMTP_INTFType	1	Binary	Type of interface: **NMTP_INTFTLOOPB(1):** Loopback (LOOPBACK/LOOPBACK6) **NMTP_INTFTOSAETH(2):** OSA-Express QDIO Ethernet (IPAQENET/IPAQENET6) **NMTP_INTFTHIPERSOCK(3):** HiperSockets (IPAQIDIO/IPAQIDIO6) **NMTP_INTFTPTP(4):** MPC Point-to-point (MPCPTP/ MPCPTP6) **NMTP_INTFTVIRTUAL(5):** Static Virtual (VIRTUAL/VIRTUAL6)
9(X'9')	NMTP_INTFRtrType	1	Binary	Router type. This field is only valid when the NMTP_INTFType field value is NMTP_INTFTOSAETH. **NMTP_INTFRTNON(1):** NONROUTER **NMTP_INTFRTPRI(2):** PRIROUTER **NMTP_INTFRTSEC(3):** SECROUTER
10(X'A')	NMTP_INTFReadStorType	1	Binary	Read storage amount type. This field is only valid when the NMTP_INTFType field value is NMTP_INTFTOSAETH or NMTP_INTFTHIPERSOCK. **NMTP_INTFRSGLOBAL(1):** GLOBAL **NMTP_INTFRSMAX(2):** MAX **NMTP_INTFRSAVG(3):** AVG **NMTP_INTFRSMIN(4):** MIN
11(X'B')	NMTP_INTFInbPerfType	1	Binary	Inbound performance type. This field is only valid when the NMTP_INTFType field value is NMTP_INTFTOSAETH. **NMTP_INTFIPBAL(1):** BALANCED **NMTP_INTFIPDYN(2):** DYNAMIC **NMTP_INTFIPMINCPU(3):** MINCPU **NMTP_INTFIPMINLAT(4):** MINLATENCY
12(X'C')	NMTP_INTFSecClass	1	Binary	SECCLASS value.

Table 175. TCP/IP profile record interface section (continued)

Offset	Name	Length	Format	Description
13(X'D')	NMTP_INTFChpID	1	Binary	CHPID value. This field is only valid for the following interface types: • IPv6 interfaces where the NMTP_INTFType field value is NMTP_INTFTHIPERSOCK. • Interfaces for which the NMTP_INTFChpIDFlg flag is set.
14(X'E')	NMTP_INTFDupAddrDet	1	Binary	DUPADDRDET count. This field is only valid for IPv6 interfaces, where the NMTP_INTFType field value is NMTP_INTFTOSAETH.
15(X'F')	NMTP_INTFIPv4Mask	1	Binary	IPv4 Subnet number of mask bits from INTERFACE or BSDROUTINGPARMS statement. If subnet mask specified on BSDROUTINGPARMS but overridden by OMPROUTE, this field is zero.
16(X'10')	NMTP_INTFTempPfxType	1	Binary	TEMPPREFIX type. This field is only valid for IPv6 interfaces where flag NMTP_INTFTempPfx is set, and the NMTP_INTFType field value is NMTP_INTFTOSAETH. **NMTP_INTFTTALL(1):** ALL **NMTP_INTFTTPFX(2):** Prefix specified **NMTP_INTFTTNONE(3):** NONE **NMTP_INTFTTDIS(4):** Temporary IPv6 address generation is disabled due to multiple Duplicate Address Detection (DAD) failures.
17(X'11')	NMTP_INTFDynTypes	1	Binary	Indicates the dynamic inbound performance types. This field is set only when the NMTP_INTFInbPerfType field is set to NMTP_INTFIPDYN and the interface was defined by an INTERFACE statement. • X'80', NMTP_INTFDYNWRKLDQ: If set, INBPERF DYNAMIC WORKLOADQ was configured.
18(X'12')	NMTP_INTFChpIDType	1	Binary	The CHPID type of the OSA-Express QDIO Ethernet interface. This field is valid only for interfaces where the NMTP_INTFType field value is NMTP_INTFTOSAETH and the interface was defined by an INTERFACE profile statement (flag NMTP_INTFDefIntf is set). **NMTP_INTFCTOSD(1):** OSD indicates an external data network CHPID type **NMTP_INTFCTOSX(2):** OSX indicates an intraensemble data network CHPID type
19(X'13')		1	Binary	Reserved

Table 175. TCP/IP profile record interface section (continued)

Offset	Name	Length	Format	Description
20(X'14')	NMTP_INTFVlanID	2	Binary	VLAN ID. This field is only valid when flag NMTP_INTFVlanIDFlg is set and the NMTP_INTFType field value is NMTP_INTFTOSAETH or NMTP_INTFTHIPERSOCK.
22(X'16')	NMTP_INTFMtu	2	Binary	MTU value. This field is only valid when flag NMTP_INTFDefIntf is set, and the NMTP_INTFType field value is NMTP_INTFTOSAETH or NMTP_INTFTHIPERSOCK.
24(X'18')	NMTP_INTFIPv4Addr	4	Binary	If flag NMTP_INTFIPv6 is not set, this field is the IPv4 IP address from the HOME or INTERFACE statement. If an IP address has not yet been configured for the interface, this field is set to zeros.
28(X'1C')	NMTP_INTFIfIndex	4	Binary	The interface index, which is a small, positive number assigned to the interface when it is defined to the TCP/IP stack. For interfaces defined by DEVICE and LINK statements, this is the interface index of the LINK.
32(X'20')	NMTP_INTFVmacAddr	6	Binary	Virtual MAC address. This field is only valid if flag NMTP_INTFVmac is set. The field contains one of the following values: • If flag NMTP_INTFVmacAddrFlg is set, the field contains the configured virtual MAC address. • If flag NMTP_INTFVmacAddrFlg is not set, and the interface is active, the field contains the virtual MAC address generated by the OSA-Express QDIO feature, when the interface was activated. If the interface is not yet active, then the field is set to zeros.
38(X'26')		2	Binary	Reserved
40(X'28')	NMTP_INTFIntfID	8	Binary	IPv6 interface ID value. This field is only valid if flag NMTP_INTFIntfIDFlg is set.
48(X'30')	NMTP_INTFName	16	EBCDIC	Interface name. For interfaces defined by DEVICE and LINK statements, this is the LINK name; otherwise, it is the interface name defined on the INTERFACE statement.
64(X'40')	NMTP_INTFAssocName	16	EBCDIC	One of the following associated names: • DEVICE name for interfaces defined with the LINK profile statement. For IPAQENET interfaces defined with the LINK statement, this is also the OSA-Express port name. For MPCPTP interfaces defined with the LINK statement, this is also the TRLE name. • PORTNAME value from the IPAQENET/IPAQENET6 INTERFACE statement. • TRLENAME value from the MPCPTP6 profile statement.

Table 175. TCP/IP profile record interface section (continued)

Offset	Name	Length	Format	Description
80(X'50')	NMTP_INTFSrcVipaIntfName	16	EBCDIC	Source VIPA interface name from the INTERFACE profile statement. This field is only valid if flag NMTP_INTFSrcVipaIfNameFlg is set.

TCP/IP profile record IPv6 address section

This section provides configured IPv6 addresses, prefixes, and temporary address prefixes from the IPv6 INTERFACE profile statements. The other IPv6 interface attributes defined on the INTERFACE statement are provided in the Interface section of the SMF record.There can be multiple IPv6 address sections in the record, one per IPv6 address or prefix.

Table 176 shows the TCP/IP profile record IPv6 address section.

Table 176. TCP/IP profile record IPv6 address section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_IPA6Eye	4	EBCDIC	IPA6 eyecatcher
4(X'4')		4	Binary	Flags: **X'80', NMTP_IPA6Deprecated:** If set, the address or address prefix has been deprecated by the DEPRADDR parameter.
5(X'5')	NMTP_IPA6Type	1	Binary	Type of entry: **NMTP_IPA6TADDR(1):** Configured address **NMTP_IPA6TPFX(2):** Configured address prefix **NMTP_IPA6TTEMPPFX(3):** Configured temporary address prefix
6(X'6')	NMTP_IPA6PfxLen	1	Binary	Prefix length. This field is only valid when the NMTP_IPA6Type is either NMTP_IPA6TPFX or NMTP_IPA6TTEMPPFX.
7(X'7')		1	Binary	Reserved
8(X'8')	NMTP_IPA6IfIndex	4	Binary	The interface index of the interface to which the IPv6 address is assigned. This is a small, positive number assigned to the interface when it is defined to the TCP/IP stack.
12(X'C')	NMTP_INTFSecClass	4	Binary	Reserved
16(X'10')	NMTP_IPA6IntfName	16	EBCDIC	Associated interface name.
32(X'20')	NMTP_IPA6Addr	16	Binary	Address or prefix.

TCP/IP profile record Routing section

This section provides configured routing information from the BEGINROUTES statement block and the GATEWAY statement. There can be multiple sections in the record, one per ROUTE substatement or GATEWAY route entry.

Table 177 on page 796 shows the TCP/IP profile record routing section.

Table 177. TCP/IP profile record routing section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_ROUTEEye	4	EBCDIC	ROUT eyecatcher
4(X'4')	NMTP_ROUTFlags	2	Binary	Flags: **X'8000', NMTP_ROUTIPv6:** IPv6 indicator. If set, this is an IPv6 route. **X'4000', NMTP_ROUTDefault:** If set, this is a default route so there is no destination IP address. **X'2000', NMTP_ROUTNextHop:** If set, a next hop address was specified. **X'1000', NMTP_ROUTDelayAcks:** If set, DELAYACKS was specified. **X'0800', NMTP_ROUTReplaceable:** If set, REPLACEABLE was specified. **X'0400', NMTP_ROUTReplaced:** If set, this is a replaceable static route which has been replaced by a dynamic route. This route is not currently being used by the TCP/IP stack.
6(X'6')	NMTP_ROUTMtu	2	Binary	MTU size
8(X'8')	NMTP_ROUTDestPfxLen	1	Binary	Destination prefix length for both IPv4 or IPv6 destination addresses. This value is set to the maximum IPv4 (32) or IPv6(128) value in the following cases: • If the HOST parameter was specified as the IPv4 address mask or IPv6 prefix length. • A prefix length was not specified.
9(X'9')		3	Binary	Reserved
12(X'C')	NMTP_ROUTIfIndex	4	Binary	Interface index of interface over which route is defined.
16(X'10')	NMTP_ROUTMaxRetranTime	4	Binary	Maximum retransmission time in milliseconds.
20(X'14')	NMTP_ROUTMinRetranTime	4	Binary	Minimum retransmission time in milliseconds.
24(X'18')	NMTP_ROUTRoundTripGain	2	Binary	Round trip gain percentage in thousandths of seconds.
26(X'1A')	NMTP_ROUTVarGain	2	Binary	Variance gain percentage in thousandths of seconds.
28(X'1C')	NMTP_ROUTVarMultiplier	4	binary	Variance multiplier value in thousandths if seconds.
32(X'20')	NMTP_ROUTIntfName	16	EBCDIC	Name of interface over which route is defined, padded with trailing blanks.
48(X'30')	NMTP_ROUTDestAddr4	4	Binary	One of the following values: • If the NMTP_ROUTIPv6 flag is not set, this field contains the IPv4 destination IP address. • If the NMTP_ROUTIPv6 flag is set, this field contains the IPv6 destination IP address.

Table 177. TCP/IP profile record routing section (continued)

Offset	Name	Length	Format	Description
48(X'30')	NMTP_ROUTDestAddr6	16	Binary	One of the following values: • If the NMTP_ROUTIPv6 flag is not set, this field contains the IPv4 destination IP address. • If the NMTP_ROUTIPv6 flag is set, this field contains the IPv6 destination IP address.
64(X'40')	NMTP_ROUTNextHopAddr4	4	Binary	Next hop IP address. This field is only valid if flag NMTP_ROUTNextHop is set. The value is one of the following: • If the NMTP_ROUTIPv6 flag is not set, this field contains the IPv4 next hop IP address. • If the NMTP_ROUTIPv6 flag is set, this field contains the IPv6 next hop IP address.
64(X'40')	NMTP_ROUTNextHopAddr6	16	Binary	Next hop IP address. This field is only valid if flag NMTP_ROUTNextHop is set. The value is one of the following: • If the NMTP_ROUTIPv6 flag is not set, this field contains the IPv4 next hop IP address. • If the NMTP_ROUTIPv6 flag is set, this field contains the IPv6 next hop IP address.

TCP/IP profile record source IP section

This section provides source IP address information from the SRCIP profile statement. There can be multiple sections in the record, one per SRCIP DESTINATION or JOBNAME substatements.

Table 178 shows the source IP section.

Table 178. TCP/IP profile record source IP section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_SRCIEye	4	EBCDIC	SRCI eyecatcher
4(X'4')	NMTP_SRCIType	1	Binary	Type of entry: **NMTP_SRCITDest(1)** Destination **NMTP_SRCITJob(2)** Job name

Table 178. TCP/IP profile record source IP section (continued)

Offset	Name	Length	Format	Description
5(X'5')	NMTP_SRCIFlags	1	Binary	Flags: **X'80'** NMTP_SRCIIPv6: IPv6 indicator. If set IP addresses are IPv6, otherwise IP addresses are IPv4. **X'40'** NMTP_SRCISrcIfName: Source IP address identifier in field NMTP_SRCISrc6 is an IPv6 interface name. **X'30'** NMTP_SRCIBoth: Both job name Clients and Servers **X'20'** NMTP_SRCIClients: Job name Clients **X'10'** NMTP_SRCIServers: Job name Servers **X'08'** NMTP_SRCITempAddrs: If the flag is set and default source IP address selection is performed, an IPv6 temporary address is preferred over an IPv6 public address. **X'04'** NMTP_SRCIPubAddrs: If the flag is set and default source IP address selection is performed, an IPv6 public address is preferred over an IPv6 temporary address.
6(X'6')		1	Binary	Reserved
7(X'7')	NMTP_SRCIDestPfxLen	1	Binary	Destination prefix length for both IPv4 or IPv6 destination addresses. This value is zero if a prefix length was not specified.
8(X'8')	NMTP_SRCIJobName	8	EBCDIC	If the NMTP_SRCIType value is Job name, this field contains the specified job name, padded with trailing blanks.
16(X'10')	NMTP_SRCIDestAddr4	4	Binary	One of the following values: • If the NMTP_SRCIType value is Destination and the NMTP_SRCIIPv6 flag is not set, this field contains the IPv4 destination IP address. • If the NMTP_SRCIType value is Destination and the NMTP_SRCIIPv6 flag is set, this field contains the IPv6 destination IP address.

Table 178. TCP/IP profile record source IP section (continued)

Offset	Name	Length	Format	Description
16(X'10')	NMTP_SRCIDestAddr6	16	Binary	One of the following values: • If the NMTP_SRCIType value is Destination and the NMTP_SRCIIPv6 flag is not set, this field contains the IPv4 destination IP address. • If the NMTP_SRCIType value is Destination and the NMTP_SRCIIPv6 flag is set, this field contains the IPv6 destination IP address.
32(X'20')	NMTP_SRCISrcAddr4	4	Binary	One of the following values: **IPv4 source IP address** If the NMTP_SRCIIPv6 flag is not set, this field contains the IPv4 source IP address. **IPv6 source IP address** If the NMTP_SRCIIPv6 flag is set, but the NMTP_SRCISrcIfName and NMTP_SRCITempAddrs flags are not set, this field contains the IPv6 source IP address. **IPv6 source interface name** If both the NMTP_SRCIIPv6 and NMTP_SRCISrcIfName flags are set, this field contains the IPv6 source interface name, padded with trailing blanks.
32(X'20')	NMTP_SRCISrcAddr6	16	Binary	One of the following values: **IPv4 source IP address** If the NMTP_SRCIIPv6 flag is not set, this field contains the IPv4 source IP address. **IPv6 source IP address** If the NMTP_SRCIIPv6 flag is set, but the NMTP_SRCISrcIfName and NMTP_SRCITempAddrs flags are not set, this field contains the IPv6 source IP address. **IPv6 source interface name** If both the NMTP_SRCIIPv6 and NMTP_SRCISrcIfName flags are set, this field contains the IPv6 source interface name, padded with trailing blanks.

Table 178. TCP/IP profile record source IP section (continued)

Offset	Name	Length	Format	Description
32(X'20')	NMTP_SRCISrcIntfName	16	EBCDIC	One of the following values: **IPv4 source IP address** If the NMTP_SRCIIPv6 flag is not set, this field contains the IPv4 source IP address. **IPv6 source IP address** If the NMTP_SRCIIPv6 flag is set, but the NMTP_SRCISrcIfName and NMTP_SRCITempAddrs flags are not set, this field contains the IPv6 source IP address. **IPv6 source interface name** If both the NMTP_SRCIIPv6 and NMTP_SRCISrcIfName flags are set, this field contains the IPv6 source interface name, padded with trailing blanks.

TCP/IP profile record management section

This section provides network management information from the NETMONITOR, SACONFIG, and SMFCONFIG profile statements. For the SMFCONFIG profile statement, only the Type 119 SMF record parameter settings are provided. For the SACONFIG profile statement the community name value from the COMMUNITY parameter is not provided due to security considerations; however, the flag bit, NMTP_MGMTSACommunity, is set if a community name was specified. There is only one of these sections in the record.

Table 179 shows the TCP/IP profile record management section.

Table 179. TCP/IP profile record management section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_MGMTEye	4	EBCDIC	MGMT eyecatcher

Table 179. TCP/IP profile record management section (continued)

Offset	Name	Length	Format	Description
4(X'4')	NMTP_MGMTSmf119Types	4	Binary	SMF 119 record types requested: **X'8000', NMTP_MGMT119FtpClient** FTP client **X'4000', NMTP_MGMT119IfStats** Interface statistics **X'2000', NMTP_MGMT119IpSec** IPSec **X'1000', NMTP_MGMT119PortStats** Port statistics **X'0800', NMTP_MGMT119Profile** Profile **X'0400', NMTP_MGMT119TcpInit** TCP connection initiation **X'0200', NMTP_MGMT119TcpipStats** TCP/IP statistics **X'0100', NMTP_MGMT119TcpStack** TCP/IP stack initiation and termination **X'0080', NMTP_MGMT119TcpTerm** TCP connection termination **X'0040', NMTP_MGMT119TN3270Client** TSO Telnet client connection initiation and termination **X'0020', NMTP_MGMT119UdpTerm** UDP endpoint termination **X'0010', NMTP_MGMT119DVIPA** Dynamic VIPAs
8(X'8')	NMTP_MGMTNetMonServices	1	Binary	NETMONITOR services requested: **X'80', NMTP_MGMTNMPktTrace** Packet trace **X'40', NMTP_MGMTNMTcpConn** TCP connection **X'20', NMTP_MGMTNMSmf** SMF records **X'10', NMTP_MGMTNMNTATrace** OSAENTA trace

Table 179. TCP/IP profile record management section (continued)

Offset	Name	Length	Format	Description
9(X'9')	NMTP_MGMTNetMonSmfRecs	1	Binary	SMFSERVICE records requested: **X'80', NMTP_MGMTNMSmfIPSec** IP Sec **X'40', NMTP_MGMTNMSmfProfile** Profile **X'20', NMTP_MGMTNMSmfCSSmtp** CSSMTP **X'10', NMTP_MGMTNMSmfCSMail** CSMail **X'08', NMTP_MGMTNMSmfDVIPA** Dynamic VIPAs
10(X'A')	NMTP_MGMTNetMonMinLife	1	Binary	If flag NMTP_MGMTNMTcpConn is set, this field contains the NETMONITOR TCPCONNSERVICE MINLIEFTIME value.
11(X'B')	NMTP_MGMTSAFlags	1	Binary	SACONFIG flags: **X'80', NMTP_MGMTSAEnabled** If set, the TCP/IP subagent is enabled. If not set, the TCP/IP subagent is disabled. **X'40', NMTP_MGMTSAOsaEnabled** If set, OSA support is enabled. If not set, OSA support is disabled. **X'20', NMTP_MGMTSASetsEnabled** If set, Set support is enabled. If not set, Set support is disabled. **X'10', NMTP_MGMTSACommunity** If set, a community name was specified.
12(X'C')	NMTP_MGMTSAAgent	2	Binary	SACONFIG Agent port number.
14(X'E')	NMTP_MGMTSAOsasf	2	Binary	SACONFIG OSASF port number.
16(X'10')	NMTP_MGMTSACacheTime	2	Binary	SACONFIG Cache time
18(X'12')		2	Binary	Reserved
20(X'14')	NMTP_MGMTSACommName	32	EBCDIC	SACONFIG Community name, padded with trailing blanks. Due to security concerns, this value is not provided in the SMF record. But if a community name value was specified, flag NMTP_MGMTSACommunity is set.

TCP/IP profile record IPSec common section

This section provides configured common information from the IPSEC profile statement. It does not provide any information about configured default filter rules. Use the IPSec rule section to obtain the default filter rule information. There is only one of these sections in the record.

Table 180 shows the TCP/IP profile record IPSec Common section.

Table 180. TCP/IP profile record IPSec Common section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_IPSCEye	4	EBCDIC	IPSC eyecatcher
4(X'4')	NMTP_IPSCFlags	1	Binary	Flags: **X'80', NMTP_IPSCDVIPSec:** If set, DVIPSEC was specified. **X'40', NMTP_IPSCLogEnable:** If set, LOGENABLE was specified. **X'20', NMTP_IPSCLogImplicit:** If set, LOGIMPLICIT was specified.
5(X'5')		3	Binary	Reserved

TCP/IP profile record IPSec rule section

This section provides the default filter rule information that is configured on the IPSEC profile statement. There can be multiple sections in the record, one per default filter rule.

Table 181 shows the TCP/IP profile record IPSec Rule section.

Table 181. TCP/IP profile record IPSec Rule section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_IPSREye	4	EBCDIC	IPSR eyecatcher

Table 181. TCP/IP profile record IPSec Rule section (continued)

Offset	Name	Length	Format	Description
4(X'4')	NMTP_IPSRFlags	2	Binary	Flags: **X'8000', NMTP_IPSRIPv6:** If set, addresses are in IPv6 format. **X'4000', NMTP_IPSRSrcAddrDef:** If set, a source address was specified and fields NMTP_IPSRSrcAddr4 or NMTP_IPSRSrcAddr4 contain the address. If not set, any source address matches the rule. **X'2000', NMTP_IPSRDestAddrDef:** If set, a destination address was specified and fields NMTP_IPSRDestAddr4 or NMTP_IPSRDestAddr4 contain the address. If not set, any destination address matches the rule. **X'1000', NMTP_IPSRLog:** If set, LOG was specified. **X'0800', NMTP_IPSRProtoDef:** If set, a protocol value was specified and field NMTP_IPSRProto contains the value. If not set, any protocol value matches the rule. **X'0400', NMTP_IPSRSrcPortDef:** If set, a source port was specified and field NMTP_IPSRSrcPort contains the port number. If not set, any source port number matches the rule. **X'0200', NMTP_IPSRDestPortDef:** NMTP_IPSRDestPortDef: If set, a destination port was specified and field NMTP_IPSRDestPort contains the port number. If not set, any destination port number matches the rule. **X'0100', NMTP_IPSRTypeDef:** If set, an ICMP, ICMPv6, or OSPF type was specified and field NMTP_IPSRType contains the type value. If not set, any type matches the rule for the specified or defaulted protocol.

Table 181. TCP/IP profile record IPSec Rule section (continued)

Offset	Name	Length	Format	Description
4(X'4') (Cont.)				**X'0080', NMTP_IPSRCodeDef:** If set, an ICMP or ICMPv6 code was specified and field NMTP_IPSRCode contains the code value. If not set, any code matches the rule for the specified or defaulted protocol and type.
6(X'6')	NMTP_IPSRSrcPfxLen	1	Binary	Source address prefix length. If a prefix was not specified, this field is set to zero.
7(X'7')	NMTP_IPSRDestPfxLen	1	Binary	Destination address prefix length. If a prefix was not specified, this field is set to zero.
8(X'8')	NMTP_IPSRProto	1	Binary	If the flag NMTP_IPSRProtoDef is set, this field contains the protocol value.
9(X'9')	NMTP_IPSRType	1	Binary	If the flag NMTP_IPSRTypeDef is set, this field contains the ICMP/ICMPv6/OSPF type value.
10(X'A')	NMTP_IPSRCode	1	Binary	If the flag NMTP_IPSRCodeDef is set, this field contains the ICMP/ICMPv6 code value.
11(X'B')	NMTP_IPSRRoutingType	1	Binary	One of the following ROUTING type values: **NMTP_IPSRRTLOCAL(1):** ROUTING LOCAL **NMTP_IPSRRTROUTED(2):** ROUTING ROUTED **NMTP_IPSRRTEITHER(3):** ROUTING EITHER
12(X'C')	NMTP_IPSRSecClass	1	Binary	SECCLASS value.
13(X'D')		3	Binary	Reserved
16(X'10')	NMTP_IPSRSrcPort	2	Binary	If the flag NMTP_IPSRSrcPortDef is set, this field contains the source port number.
18(X'12')	NMTP_IPSRDestPort	2	Binary	If the flag NMTP_IPSRDestPortDef is set, this field contains the destination port number.
20(X'14')	NMTP_IPSRSrcAddr4	4	Binary	If the flag NMTP_IPSRSrcAddrDef is set, this field contains one of the following values: • If the NMTP_IPSRIPv6 flag is not set, this field contains the source address in IPv4 format • If the NMTP_IPSRIPv6 flag is set, this field contains the source address in IPv6 format.

Table 181. TCP/IP profile record IPSec Rule section (continued)

Offset	Name	Length	Format	Description
20(X'14')	NMTP_IPSRSrcAddr6	16	Binary	One of the following values: • If the NMTP_SRCIType value is Destination and the NMTP_SRCIIPv6 flag is not set, this field contains the IPv4 destination IP address. • If the NMTP_SRCIType value is Destination and the NMTP_SRCIIPv6 flag is set, this field contains the IPv6 destination IP address.
36(X'24')	NMTP_IPSRDestAddr4	4	Binary	If the flag NMTP_IPSRDestAddrDef is set, this field contains one of the following values: • If the NMTP_IPSRIPv6 flag is not set, this field contains the destination address in IPv4 format. • If the NMTP_IPSRIPv6 flag is set, this field contains the destination address in IPv6 format.
36(X'24')	NMTP_IPSRDestAddr6	16	Binary	If the flag NMTP_IPSRDestAddrDef is set, this field contains one of the following values: • If the NMTP_IPSRIPv6 flag is not set, this field contains the destination address in IPv4 format. • If the NMTP_IPSRIPv6 flag is set, this field contains the destination address in IPv6 format..

TCP/IP profile record network access section

This section provides network access control information from the NETACCESS profile statement. There can be multiple sections in the record, one per NETACCESS network substatement.

Table 182 shows the TCP/IP profile record network access section.

Table 182. TCP/IP profile record network access section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_NETAEye	4	EBCDIC	NETA eyecatcher

Table 182. TCP/IP profile record network access section (continued)

Offset	Name	Length	Format	Description
4(X'4')	NMTP_NETAFlags	1	Binary	NETACCESS flags: **X'80', NMTP_NETAIPv6** If set IP addresses are IPv6, otherwise IP addresses are IPv4. **X'40', NMTP_NETAInBound** If set, inbound network access control checking is in effect. **X'20', NMTP_NETAOutBound** If set, outbound network access control checking is in effect. **X'10', NMTP_NETADefault** If set, this is a DEFAULT entry. **X'08', NMTP_NETADefaultHome** If set, this is a DEFAULTHOME entry.
5(X'5')	NMTP_NETANetwPfxLen	1	Binary	Network address prefix length for the IPv4 or IPv6 network value.
6(X'6'		2	Binary	Reserved
8(X'8')	NMTP_NETASafName	8	EBCDIC	SAF resource name, padded with trailing blanks.
16(X'10')	NMTP_NETANetwAddr4	4	Binary	One of the following values: • If the NMTP_NETAIPv6 flag is not set, and this is not a DEFAULT or DEFAULTHOME entry, this field contains the IPv4 network value. The network value is the IPv4 network address ANDed with the prefix length. • If the NMTP_NETAIPv6 flag is set, and this is not a DEFAULT or DEFAULTHOME entry, this field contains the IPv6 network value. The network value is the IPv6 network address ANDed with the prefix length.

Table 182. TCP/IP profile record network access section (continued)

Offset	Name	Length	Format	Description
16(X'10')	NMTP_NETANetwAddr6	16	Binary	One of the following values: • If the NMTP_NETAIPv6 flag is not set, and this is not a DEFAULT or DEFAULTHOME entry, this field contains the IPv4 network value. The network value is the IPv4 network address ANDed with the prefix length. • If the NMTP_NETAIPv6 flag is set, and this is not a DEFAULT or DEFAULTHOME entry, this field contains the IPv6 network value. The network value is the IPv6 network address ANDed with the prefix length.

TCP/IP profile record dynamic VIPA (DVIPA) address section

This section provides information about dynamic VIPA (DVIPA) address and interfaces, from the following VIPADYNAMIC profile substatements:

- VIPABACKUP
- VIPADEFINE
- VIPARANGE

There can be multiple sections in the record, one per each of the above profile substatements. If requested configuration changes for this section were cancelled, then:

- Only one section is provided in the record.
- Flag NMTP_DVCFChgCancelled is set. If this flag is set, no other information is provided in the section.

Table 183 shows the dynamic VIPA address section.

Table 183. TCP/IP profile record dynamic VIPA (DVIPA) address section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_DVCFEye	4	EBCDIC	DVCF eyecatcher

Table 183. TCP/IP profile record dynamic VIPA (DVIPA) address section (continued)

Offset	Name	Length	Format	Description
4(X'4')	NMTP_DVCFFlags	2	Binary	DVIPA flags: **X'8000', NMTP_DVCFChgCancelled:** If set, pending configuration changes for this section were cancelled because the stack is not currently joined to the sysplex group. If this flag is set, no other information is provided in this section. **X'4000', NMTP_DVCFIPv6:** If set, IP addresses are IPv6; otherwise, IP addresses are IPv4. **X'2000', NMTP_DVCFMoveImmed:** This flag is only valid when the value of NMTP_DVCFType is Backup or Define. If set, the DVIPA can be immediately moved to another stack when the other stack requests ownership of it, but existing connections are preserved. If this flag is not set, the DVIPA can not move to another stack until all current connections have ended. **X'1000', NMTP_DVCFMoveNonDisrupt:** This flag is only valid if the value of NMTP_DVCFType is Range. If set, the DVIPA can be immediately moved to another stack when the other stack requests ownership of it, but existing connections are preserved. If this flag is not set: • A subsequent BIND on another stack for the same DVIPA address fails. A subsequent SIOCSVIPA ioctl on another stack succeeds and the DVIPA is deleted from this stack. Any connections to the DVIPA on this offset are terminated. • A subsequent SIOCSVIPA ioctl on another stack succeeds and the DVIPA is deleted from this stack. Any connections to the DVIPA on this stack are terminated.

Table 183. TCP/IP profile record dynamic VIPA (DVIPA) address section (continued)

Offset	Name	Length	Format	Description
4(X'4') (Cont)				**X'0800', NMTP_DVCFCpcScope** If set, the DVIPA cannot be moved to or taken over by another TCP/IP stack that is in a different central processor complex (CPC). This flag is only valid if field NMTP_DVCFType is set to Backup or Define. **X'0400', NMTP_DVCFTier1** If set, the DVIPA is used to distribute incoming requests to non-z/OS targets. This flag is only valid if field NMTP_DVCFType is set to Backup or Define. **X'0200', NMTP_DVCFTier2** If set, the DVIPA is used to distribute incoming requests from Tier1 targets to server applications and the DVIPA cannot be moved to or taken over by another TCP/IP stack that is in a different CPC. This flag is only valid if field NMTP_DVCFType is set to Backup or Define. **X'0100', NMTP_DVCFServMgr** If set, and this DVIPA is distributed, MultiNode Load Balancing (MNLB) is performed as part of the distribution. This flag is not supported. **X'0080', NMTP_DVCFDeactivated** If set, the associated DVIPA address is currently deactivated. DVIPA addresses and interfaces can be deactivated by way of the VARY TCPIP,,SYSPLEX,DEACTIVATE command. This flag is only valid if NMTP_DVCFType is Backup or Define. **X'0040', NMTP_DVCFSAFNameSet** If set, the SAF parameter is specified on the VIPARANGE statement. Field NMTP_DVCFSAFName contains the SAF parameter value.
6(X'6')	NMTP_DVCFType	1	Binary	DVIPA entry type: **NMTP_DVCFBackup(1)** Backup **NMTP_DVCFDefine(2)** Define **NMTP_DVCFRange(3)** Range
7(X'7')	NMTP_DVCFBackupRank	1	Binary	If the NMTP_DVCFType value is Backup, this field contains the rank value.

Table 183. TCP/IP profile record dynamic VIPA (DVIPA) address section (continued)

Offset	Name	Length	Format	Description
8(X'8')	NMTP_DVCFPfxLen	8	EBCDIC	One of the following values: • If the NMTP_DVCFType value is Define or Backup, and the NMTP_DVCFIPv6 flag is not set, this field contains the IPv4 subnet prefix length. • If the NMTP_DVCFType value is Define or Backup, and the NMTP_DVCFIPv6 flag is set, this field contains the IPv6 subnet prefix length. • If the NMTP_DVCFType value is Range, and the NMTP_DVCFIPv6 flag is not set, this field contains the prefix length used to create the IPv4 VIPARANGE prefix. • If the NMTP_DVCFType value is Range, and the NMTP_DVCFIPv6 flag is set, this field contains the prefix length used to create the IPv6 VIPARANGE prefix. • 0 if a prefix length was not specified.
9(X'9')		7	Binary	Reserved
16(X'10')	NMTP_DVCFAddr4	4	Binary	One of the following values: • If the NMTP_DVCFIPv6 flag is not set, this field contains the IPv4 DVIPA IP address. • If the NMTP_DVCFIPv6 flag is set, this field contains the IPv6 DVIPA IP address.
16(X'10)	NMTP_DVCFAddr6	16	Binary	One of the following values: • If the NMTP_DVCFIPv6 flag is not set, this field contains the IPv4 DVIPA IP address. • If the NMTP_DVCFIPv6 flag is set, this field contains the IPv6 DVIPA IP address.
32(X'20')	NMTP_DVCFIntfName	16	EBCDIC	If the NMTP_DVCFIPv6 flag is set, this field contains the IPv6 DVIPA interface name, padded with trailing blanks.
48(X'30')	NMTP_DVCFSAFName	8	EBCDIC	If the NMTP_DVCFSAFNameSet flag is set, this field contains the name specified on the SAF parameter of the VIPARANGE statement, padded with trailing blanks.

TCP/IP profile record dynamic VIPA (DVIPA) routing section

This section provides information about routes configured for dynamic VIPA (DVIPA) distribution on the VIPADYNAMIC VIPAROUTE profile substatement.

There can be multiple sections in the record, one for each VIPAROUTE substatement. If requested configuration changes for this section were cancelled, then:

- Only one section is provided in the record
- Flag NMTP_DVRTChgCancelled is set. If this flag is set, no other information is provided in the section.

Table 184 on page 812 shows the dynamic VIPA routing section.

Table 184. TCP/IP profile record dynamic VIPA (DVIPA) routing section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_DVRTEye	4	EBCDIC	DVRT eyecatcher
4(X'4')	NMTP_DVRTFlags	1	Binary	DVIPA route flags: **X'80', NMTP_DVRTChgCancelled:** If set, pending configuration changes for this section were cancelled because the stack is not currently joined to the sysplex group. If this flag is set, no other information is provided in this section. **X'40', NMTP_DVRTIPv6:** If set, IP addresses are IPv6; otherwise, IP addresses are IPv4.
5(X'5')		3	Binary	Reserved
8(X'8')	NMTP_DVRTDynXcfAddr4	4	Binary	One of the following values: • If the NMTP_DVRTIPv6 flag is not set, this field contains the IPv4 dynamic XCF IP address. • If the NMTP_DVRTIPv6 flag is set, this field contains the IPv6 dynamic XCF IP address.
8(X'8')	NMTP_DVRTDynXcfAddr6	16	Binary	One of the following values: • If the NMTP_DVRTIPv6 flag is not set, this field contains the IPv4 dynamic XCF IP address. • If the NMTP_DVRTIPv6 flag is set, this field contains the IPv6 dynamic XCF IP address.
24(X'18')	NMTP_DVRTTargetAddr4	4	Binary	One of the following values: • If the NMTP_DVRTIPv6 flag is not set, this field contains the IPv4 target IP address. • If the NMTP_DVRTIPv6 flag is set, this field contains the IPv6 target IP address.
24(X'18')	NMTP_DVRTTargetAddr6	16	Binary	One of the following values: • If the NMTP_DVRTIPv6 flag is not set, this field contains the IPv4 target IP address. • If the NMTP_DVRTIPv6 flag is set, this field contains the IPv6 target IP address.

TCP/IP profile record distributed dynamic VIPA (DVIPA) section

This section provides information about distributed TCP connection processing for dynamic VIPA (DVIPA) interfaces. This information is configured on the VIPADYNAMIC VIPADISTRIBUTE profile statement. There can be multiple sections in the record. Each section represents one distributed dynamic VIPA, per one distributed port, per one destination to a target TCP/IP stack or non-z/OS target.

If requested configuration changes for this section were cancelled, then the following occurs:

- Only one section is provided in the record.
- Flag NMTP_DDVSChgCancelled is set. If this flag is set, no other information is provided in the section.

Table 185 shows the Distributed dynamic VIPA section.

Table 185. TCP/IP profile record Distributed dynamic VIPA (DVIPA) section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_DDVSEye	4	EBCDIC	DDVS eyecatcher
4(X'4')	NMTP_DDVSFlags	2	Binary	Distributed DVIPA flags: **X'8000', NMTP_DDVSChgCancelled:** If set, pending configuration changes for this section were cancelled because the stack is not currently joined to the sysplex group. If this flag is set, no other information is provided in this section. **X'4000', NMTP_DDVSIPv6:** If set, this is an IPv6 entry; otherwise, it is an IPv4 entry. **X'2000', NMTP_DDVSPort:** If set, the PORT parameter was specified and field NMTP_DDVSDistPortNum contains the distributed port number. **X'1000', NMTP_DDVSDestipAll:** If set, connections to the DVIPA address can be distributed to all stacks connected to this stack by way of a dynamic XCF interface of the same protocol type (IPv4 or IPv6) as the DVIPA address. If flag NMTP_DDVSTier2 is set, connections can only be distributed to targets on the same CPC as the Tier2 distributor. **X'0800', NMTP_DDVSOptLocal:** If set, target stacks should normally process new connection requests locally instead of sending them to the sysplex distributor stack, depending on the OPTLOCAL value in field NMTP_DDVSOptLocalValue. **X'0400', NMTP_DDVSSysplexPorts:** If set, coordinated sysplex-wide ephemeral port assignment is activated for the distributed DVIPA on all stacks where the DVIPA is defined.

Table 185. TCP/IP profile record Distributed dynamic VIPA (DVIPA) section (continued)

Offset	Name	Length	Format	Description
4(X'4') (Cont)				**X'0200', NMTP_DDVSTier1:** If this parameter is set, and the NMTP_DDVSTier1Gre flag is set, incoming connection requests to the distributed DVIPA are distributed to non-z/OS targets. If this parameter is set, and the NMTP_DDVSTier1Gre flag is not set, incoming connection requests to the distributed DVIPA are distributed to z/OS targets. • If NMTP_DDVSIPv6 is not set, the NMTP_DDVSDestipAddr4 field contains the IPv4 target IP address. • If NMTP_DDVSIPv6 is set, the NMTP_DDVSDestipAddr6 fields contain the IPv6 target IP address. • The NMTP_DDVSTierGroupName field contains the TIER1 group name. • If the NMTP_DDVSTier1Gre flag is set, the NMTP_DDVSControlPortNum field contains the control port number. **X'0100', NMTP_DDVSTier1Gre:** If set and NMTP_DDVSIPv6 is not set, generic routing encapsulation (GRE) is used to distribute requests to IPv4 tier 1 non-z/OS targets. If set and NMTP_DDVSIPv6 is set, IPv6 routing encapsulation is used to distribute requests to IPv6 tier 1 non-z/OS targets. This flag can be set only if flag NMTP_DDVSTier1 is set. **X'0080', NMTP_DDVSTier2:** If set, the DVIPA is used to distribute incoming requests from tier 1 targets to server applications. The NMTP_DDVSTierGroupName field contains the TIER2 group name. **X'0040', NMTP_DDVSDeactivated:** If set, the associated distributed DVIPA is currently deactivated. DVIPA distribution can be deactivated by using the VARY TCPIP,,SYSPLEX,DEACTIVATE command to deactivate the corresponding DVIPA address. **X'0020', NMTP_DDVSSrvTypePreferred:** When the value of NMTP_DDVSDistMethod is HotStandby, this flag is set if the server type is Preferred: **1** This is the preferred server. **0** This is not the preferred server. **X'0010', NMTP_DDVSSrvTypeBackup:** When the value of NMTP_DDVSDistMethod is HotStandby, this flag is set if the server type is Backup: **1** This is a backup server. **0** This is not a backup server. **X'0008', NMTP_DDVSAutoSwitchBack:** When the value of NMTP_DDVSDistMethod is HotStandby, this flag is the AUTOSWITCHBACK setting: **1** AUTOSWITCHBACK is configured. **0** NOAUTOSWITCHBACK is configured **X'0004', NMTP_DDVSHealthSwitch:** When the value of NMTP_DDVSDistMethod is HotStandby, this flag is the HEALTHSWITCH setting: **1** HEALTHSWITCH is configured. **0** NOHEALTHSWITCH is configured.

Table 185. TCP/IP profile record Distributed dynamic VIPA (DVIPA) section (continued)

Offset	Name	Length	Format	Description
6(X'6')	NMTP_DDVSDistMethod	1	Binary	One of the following distribution methods: **NMTP_DDVSBaseWlm(1)** BaseWlm **NMTP_DDVSRoundRobin(2)** RoundRobin **NMTP_DDVSServerWlm(3)** ServerWlm **NMTP_DDVSWeightedActive(4)** WeightedActive **NMTP_DDVSTargetControlled(5)** TargetControlled **NMTP_DDVSHotStandby(6)** HotStandby
7(X'7')	NMTP_DDVSBWProcTypeCp	1	Binary	When the value of NMTP_DDVSDistMethod is BaseWlm, this field contains the proportion of the workload that is expected to use conventional processors.
8(X'8')	NMTP_DDVSBWProcTypeZaap	1	Binary	When the value of NMTP_DDVSDistMethod is BaseWlm, this field contains the proportion of the workload that is expected to use zAAP processors.
9(X'9')	NMTP_DDVSBWProcTypeZiip	1	Binary	When the value of NMTP_DDVSDistMethod is BaseWlm, this field contains the proportion of the workload that is expected to use zIIP processors.
10(X'A')	NMTP_DDVSSWProcXcostZaap	1	Binary	When the value of NMTP_DDVSDistMethod is ServerWlm, this field contains the crossover cost of running the targeted zAAP workload on a conventional processor instead of the zAAP processor.
11(X'B')	NMTP_DDVSSWProcXcostZiip	1	Binary	When the value of NMTP_DDVSDistMethod is ServerWlm, this field contains the crossover cost of running the targeted zIIP workload on a conventional processor instead of the zIIP processor.
12(X'C')	NMTP_DDVSSWIlWeighting	1	Binary	When the value of NMTP_DDVSDistMethod is ServerWlm, this field contains the weighting factor WLM uses when comparing displaceable capacity at different importance levels (IL's) as it determines a SERVERWLM recommendation for each system.
13(X'D')	NMTP_DDVSWADestipWeight	1	Binary	When the value of NMTP_DDVSDistMethod is WeightedActive, this field contains the weight used by the distributor to determine the proportion of active connections on this target.
14(X'E')	NMTP_DDVSOptLocalValue	1	Binary	If flag NMTP_DDVSOptLocal is set, this field contains the OPTLOCAL value.
15(X'F')	NMTP_DDVSBackupRank	1	Binary	When the flag NMTP_DDVSSrvTypeBackup is set, this field contains the rank of the backup server.
16(X'10')		2	Binary	Reserved
18(X'12')	NMTP_DDVSTimedAffinity	2	Binary	The number of seconds during which connection requests from a client are routed to the same target server. This value is only valid if the NMTP_DDVSOptlocal flag is not set.
20(X'14')	NMTP_DDVSControlPortNum	2	Binary	If flag NMTP_DDVSTier1 is set, this field contains the destination port number to be used when establishing a control connection to the Tier1 target.
22(X'16')	NMTP_DDVSDistPortNum	2	Binary	If flag NMTP_DDVSPort is set, this field contains the port number for one of the distributed ports.
24(X'18')	NMTP_DDVSTierGroupName	16	EBCDIC	If either flag NMTP_DDVSTier1 or flag NMTP_DDVSTier2 is set, this field contains the group name.
40(X'28')	NMTP_DDVSDistAddr	4	Binary	One of the following values: • If the NMTP_DDVSIPv6 flag is not set, this field contains the IPv4 distributed DVIPA IP address. • If the NMTP_DDVSIPv6 flag is set, this field contains the IPv6 distributed DVIPA interface name.
40(X'28')	NMTP_DDVSDistIntfName	16	EBCDIC	One of the following values: • If the NMTP_DDVSIPv6 flag is not set, this field contains the IPv4 distributed DVIPA IP address. • If the NMTP_DDVSIPv6 flag is set, this field contains the IPv6 distributed DVIPA interface name.
56(X'38')	NMTP_DDVSDestipAddr4	4	Binary	If the flag NMTP_DDVSDestipAll is not set, this field contains one of the destinations to which connections requests are sent. If the NMTP_DDVSIPv6 flag is set, this field contains an IPv6 IP address, otherwise it contains an IPv4 IP address. The address is one of the following values: • If the NMTP_DDVSTier1Gre flag is not set, a dynamic XCF IP address to a target stack. • If the NMTP_DDVSTier1GRE flag is set, a non- z/OS target's IP address.

Table 185. TCP/IP profile record Distributed dynamic VIPA (DVIPA) section (continued)

Offset	Name	Length	Format	Description
56(X'38')	NMTP_DDVSDestipAddr6	16	Binary	If the flag NMTP_DDVSDestipAll is not set, this field contains one of the destinations to which connections requests are sent. If the NMTP_DDVSIPv6 flag is set, this field contains an IPv6 IP address, otherwise it contains an IPv4 IP address. The address is one of the following values: • If the NMTP_DDVSTier1Gre flag is not set, a dynamic XCF IP address to a target stack. • If the NMTP_DDVSTier1GRE flag is set, a non- z/OS target's IP address.

TCP/IP profile record policy table for IPv6 default address selection section

This section provides information about the policy table for IPv6 default address selection. This information is configured on the DEFADDRTABLE profile statement. Multiple sections can be in the record, one per policy.

Table 186. TCP/IP profile record policy table for IPv6 default address selection section

Offset	Name	Length	Format	Description
0(X'0')	NMTP_DASPEye	4	EBCDIC	Eyecatcher
4(X'4')	NMTP_DASPPrefix	16	Binary	IPv6 address prefix
20(X'14')	NMTP_DASPPfxLen	1	Binary	IPv6 prefix length
21(X'15')	NMTP_DASPRsv	3	Binary	Reserved
24(X'18')	NMTP_DASPPrecedence	2	Binary	Policy precedence
26(X'1A')	NMTP_DASPLabel	2	Binary	Policy label

TCP/IP statistics record (subtype 5)

The TCP/IP statistics record is collected at user specified intervals. The record provides data about IP, TCP, UDP, and ICMP activity in the reporting TCP stack during the previous recording interval. The cumulative value for each statistic reported is obtained by summing the values reported for the statistic in the individual TCP/IP statistics interval records. If TCP/IP statistics recording is turned off dynamically, or the TCP stack terminates, a final TCP/IP statistics record is generated to report close-out statistics.

The Type 119 TCP/IP statistics record is generated using the same user specified interval time value as the equivalent Type 118 TCPIPSTATISTICS record.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the TCP/IP statistics record, the TCP/IP stack identification section indicates STACK as the subcomponent and X'08' (event record), X'20' (recording stop), or X'10' (recording shutdown) as the record reason.

Table 187 shows the TCP/IP statistics record self-defining section:

Table 187. SMF records: TCP/IP statistics record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 5(X'5')
Self-defining section				
24(X'5')	SMF119SD_TRN	2	Binary	Number of triplets in this record (7)

Table 187. SMF records: TCP/IP statistics record self-defining section (continued)

Offset	Name	Length	Format	Description
26(X'1A')		2	Binary	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 IPv4 IP statistics section
40(X'28')	SMF119S1Len	2	Binary	Length of IPv4 IP statistics section
42(X'2A')	SMF119S1Num	2	Binary	Number of IPv4 IP statistics sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to TCP statistics section
48(X'30')	SMF119S2Len	2	Binary	Length of TCP statistics section
50(X'32')	SMF119S2Num	2	Binary	Number of TCP statistics sections
52(X'34')	SMF119S3Off	4	Binary	Offset to UDP statistics section
56(X'38')	SMF119S3Len	2	Binary	Length of UDP statistics section
58(X'3A')	SMF119S3Num	2	Binary	Number of UDP statistics sections
60(X'3C')	SMF119S4Off	4	Binary	Offset to IPv4 ICMP statistics section
64(X'40')	SMF119S4Len	2	Binary	Length of IPv4 ICMP statistics section
66(X'42')	SMF119S4Num	2	Binary	Number of IPv4 ICMP statistics sections
68 (X'44')	SMF119S5Off	4	Binary	Offset to IPv6 IP statistics section
72 (X'48')	SMF119S5Len	2	Binary	Length of IPv6 IP statistics section
74 (X'4A')	SMF119S5Num	2	Binary	Number of IPv6 IP statistics sections
76 (X'4C')	SMF119S6Off	4	Binary	Offset to IPv6 ICMP statistics section
80 (X'50')	SMF119S6Len	2	Binary	Length of IPv6 ICMP statistics section
82 (X'52')	SMF119S6Num	2	Binary	Number of IPv6 ICMP statistics sections
84 (X'54')	SMF119S7Off	4	Binary	Offset to storage statistics section
88 (X'58')	SMF119S7Len	2	Binary	Length of storage statistics section
90 (X'5A')	SMF119S7Num	2	Binary	Number of storage statistics sections

Table 188 shows the IP statistics section:

Table 188. IP statistics section

Offset	Name	Length	Format	Description
0(X'0')	SMF119AP_TSIPDuration	8	Binary	Duration of recording interval in microseconds, where bit 51 is equivalent to one microsecond
8(X'8')	SMF119AP_TSIPRecData	4	Binary	Number of datagrams received
12(X'C')	SMF119AP_TSIPDscData	4	Binary	Number of input datagrams discarded due to errors in their IP headers

Table 188. IP statistics section (continued)

Offset	Name	Length	Format	Description
16(X'10')	SMF119AP_TSIPDscDAddr	4	Binary	Number of input datagrams discarded because the IP address in their IP header's destination field was not valid
20(X'14')	SMF119AP_TSIPAttFwdData	4	Binary	Number of attempts to forward datagrams
24(X'18')	SMF119AP_TSIPDscDUnkPr	4	Binary	Number of datagrams discarded because of an unknown or unsupported protocol
28(X'1C')	SMF119AP_TSIPDscDOth	4	Binary	Number of input datagrams discarded that are not accounted for in another input discard counter
32(X'20')	SMF119AP_TSIPDlvData	4	Binary	Number of datagrams delivered
36(X'24')	SMF119AP_TSIPXData	4	Binary	Number of datagrams transmitted
40(X'28')	SMF119AP_TSIPXDscOth	4	Binary	Number of outbound transmitted datagrams discarded, due to reasons other than no route being available
44(X'2C')	SMF119AP_TSIPXDscRoute	4	Binary	Number of outbound transmitted datagrams discarded, due to no route being available
48(X'30')	SMF119AP_TSIPTimeouts	4	Binary	Number of reassembly timeouts
52(X'34')	SMF119AP_TSIPRecDRsbm	4	Binary	Number of received datagrams requiring assembly
56(X'38')	SMF119AP_TSIPRsmb	4	Binary	Number of datagrams reassembled
60(X'3C')	SMF119AP_TSIPFailRsmb	4	Binary	Number of failed reassembly attempts
64(X'40')	SMF119AP_TSIPRecFgmt	4	Binary	Number of fragmented datagrams received
68(X'44')	SMF119AP_TSIPDscDFgmt	4	Binary	Number of discarded datagrams due to fragmentation failures
72(X'48')	SMF119AP_TSIPXFgmt	4	Binary	Number of fragments generated
76(X'4C')	SMF119AP_TSIPRouteDisc	4	Binary	Number of routing discards
80(X'50')	SMF119AP_TSIPMaxRsmb	4	Binary	Maximum active number of reassemblies
84(X'54')	SMF119AP_TSIPCurRsmb	4	Binary	Number of currently active reassemblies
88(X'58')	SMF119AP_TSIPRsmbFlags	4	Binary	Reassembly flags
92(X'5C')	SMF119AP_TSIPInCalls	4	Binary	Number of inbound calls from device layer
96(X'60')	SMF119AP_TSIPInUerrs	4	Binary	Number of received frame unpacking
100(X'64')	SMF119AP_TSIPIDMem	4	Binary	Number of discarded datagrams, due to memory shortages
104(X'68')	SMF119AP_TSIPODSync	4	Binary	Number of transmitted datagrams discarded, due to Sync errors
108(X'6C')	SMF119AP_TSIPODAsyn	4	Binary	Number of transmitted datagrams discarded, due to Async errors

Table 188. IP statistics section (continued)

Offset	Name	Length	Format	Description
112(X'70')	SMF119AP_TSIPODMem	4	Binary	Number of transmitted datagrams discarded due to memory shortages

Table 189 shows the TCP statistics section:

Table 189. TCP statistics section

Offset	Name	Length	Format	Description
0(X'0')	SMF119AP_TSTCDuration	8	Binary	Duration of recording interval in microseconds, where bit 51 is equivalent to one microsecond
8(X'8')	SMF119AP_TSTCAlg	4	Binary	Retransmission algorithm
12(X'C')	SMF119AP_TSTCMinRet	4	Binary	Minimum retransmission time, in milliseconds
16(X'10')	SMF119AP_TSTCMxRet	4	Binary	Maximum retransmission time, in milliseconds
20(X'14')	SMF119AP_TSTCMxCon	4	Binary	Maximum TCP connections
24(X'18')	SMF119AP_TSTCOpenConn	4	Binary	Number of active open connections
28(X'1C')	SMF119AP_TSTCPassConn	4	Binary	Number of passive open connections
32(X'20')	SMF119AP_TSTCOFails	4	Binary	Number of open connection failures
36(X'24')	SMF119AP_TSTCConReset	4	Binary	Number of connection resets
40(X'28')	SMF119AP_TSTCEstab	4	Binary	Number of current establishments
44(X'2C')	SMF119AP_TSTCInSegs	4	Binary	Number of input TCP segments
48(X'30')	SMF119AP_TSTCOSegs	4	Binary	Number of output TCP segments
52(X'34')	SMF119AP_TSTCRxSegs	4	Binary	Number of retransmitted segments
56(X'38')	SMF119AP_TSTCInErrs	4	Binary	Number of input errors
60(X'3C')	SMF119AP_TSTCReset	4	Binary	Number of resets
64(X'40')	SMF119AP_TSTCConCls	4	Binary	Number of TCP connections closed
68(X'44')	SMF119AP_TSTCConAttD	4	Binary	Number of TCP connection attempts discarded

Table 189. TCP statistics section (continued)

Offset	Name	Length	Format	Description
72(X'48')	SMF119AP_TSTCTWRef	4	Binary	Number of TCP Timewait connections refused
76(X'4C')	SMF119AP_TSTCHOKAck	4	Binary	Number of header predictions (OK for ACK)
80(X'50')	SMF119AP_TSTCHOKDat	4	Binary	Number of header predictions (OK for Data)
84(X'54')	SMF119AP_TSTCIDupAck	4	Binary	Number of duplicate ACKs received
88(X'58')	SMF119AP_TSTCDscChecksum	4	Binary	Number of received packets discarded due to bad checksum values
92(X'5C')	SMF119AP_TSTCDscLen	4	Binary	Number of received packets discarded due to bad header length
96(X'60')	SMF119AP_TSTCDscInsData	4	Binary	Number of received packets discarded due to insufficient data
100(X'64')	SMF119AP_TSTCDscOldTime	4	Binary	Number of received packets discarded due to old timestamp information
104(X'68')	SMF119AP_TSTCICmpDupSeg	4	Binary	Number of received complete duplicate segments
108(X'6C')	SMF119AP_TSTCIPartDupSeg	4	Binary	Number of received partial duplicate segments
112(X'70')	SMF119AP_TSTCICmpSegsWin	4	Binary	Number of complete segments received after window closure
116(X'74')	SMF119AP_TSTCIPartSegsWin	4	Binary	Number of partial segments received after window closure
120(X'78')	SMF119AP_TSTCIOOrder	4	Binary	Number of out of order segments received
124(X'7C')	SMF119AP_TSTCISegCls	4	Binary	Number of segments received after the TCP connection closed

Table 189. TCP statistics section (continued)

Offset	Name	Length	Format	Description
128(X'80')	SMF119AP_TSTCIWinPr	4	Binary	Number of received window probes
132(X'84')	SMF119AP_TSTCIWinUp	4	Binary	Number of received window updates
136(X'88')	SMF119AP_TSTCOWinPr	4	Binary	Number of transmitted window probes
140(X'8C')	SMF119AP_TSTCOWinUp	4	Binary	Number of transmitted window updates
144(X'90')	SMF119AP_TSTCODlAck	4	Binary	Number of transmitted delayed ACKs
148(X'94')	SMF119AP_TSTCOKApr	4	Binary	Number of transmitted keepalive probes
152(X'98')	SMF119AP_TSTCRxTim	4	Binary	Number of retransmitted timeouts
156(X'9C')	SMF119AP_TSTCRxMTU	4	Binary	Number of retransmitted Path MTU discovery packets
160(X'A0')	SMF119AP_TSTCPathM	4	Binary	Number of Path MTUs beyond retransmit limit
164(X'A4')	SMF119AP_TSTCDropPr	4	Binary	Number of TCP connections dropped due to probes
168(X'A8')	SMF119AP_TSTCDropKA	4	Binary	Number of TCP connections dropped while in KeepAlive state
172(X'AC')	SMF119AP_TSTCDropF2	4	Binary	Number of TCP connections dropped while in FINWAIT2 state
176(xB'0')	SMF119AP_TSTCDropRx	4	Binary	Number of TCP connections dropped due to retransmits

Table 190 shows the UDP statistics section:

Table 190. UDP statistics section

Offset	Name	Length	Format	Description
0(X'0')	SMF119AP_TSUDDuration	8	Binary	Duration of recording interval in microseconds, where bit 51 is equivalent to one microsecond

Table 190. UDP statistics section (continued)

Offset	Name	Length	Format	Description
8(X'8')	SMF119AP_TSUDRecData	8	Binary	Number of UDP datagrams received
16(X'10')	SMF119AP_TSUDRecNoPort	4	Binary	Number of UDP datagrams received with no port defined
20(X'14')	SMF119AP_TSUDNoRec	4	Binary	Number of other UDP datagrams not received
24(X'18')	SMF119AP_TSUDXmtData	8	Binary	Number of UDP datagrams sent

Table 191 shows the ICMP statistics section:

Table 191. ICMP statistics section

Offset	Name	Length	Format	Description
0(X'0')	SMF119AP_TSICDuration	8	Binary	Duration of recording interval in microseconds, where bit 51 is equivalent to one microsecond
8(X'8')	SMF119AP_TSICInMsg	4	Binary	Number of inbound ICMP messages
12(X'C')	SMF119AP_TSICInError	4	Binary	Number of inbound ICMP error messages
16(X'10')	SMF119AP_TSICInDstUnreach	4	Binary	Number of inbound ICMP destination unreachable messages
20(X'14')	SMF119AP_TSICInTimeExcd	4	Binary	Number of inbound ICMP time exceeded messages
24(X'18')	SMF119AP_TSICInParmProb	4	Binary	Number of inbound ICMP parameter problem messages
28(X'1C')	SMF119AP_TSICInSrcQuench	4	Binary	Number of inbound ICMP source quench messages
32(X'20')	SMF119AP_TSICInRedirect	4	Binary	Number of inbound ICMP redirect messages
36(X'24')	SMF119AP_TSICInEcho	4	Binary	Number of inbound ICMP echo request messages
40(X'28')	SMF119AP_TSICInEchoRep	4	Binary	Number of inbound ICMP echo reply messages
44(X'2C')	SMF119AP_TSICInTstamp	4	Binary	Number of inbound ICMP timestamp request messages
48(X'30')	SMF119AP_TSICInTstampRep	4	Binary	Number of inbound ICMP timestamp reply messages
52(X'34')	SMF119AP_TSICInAddrMask	4	Binary	Number of inbound ICMP address mask request messages
56(X'38')	SMF119AP_TSICInAddrMRep	4	Binary	Number of inbound ICMP address mask reply messages
60(X'3C')	SMF119AP_TSICOutMsg	4	Binary	Number of outbound ICMP messages
64(X'40')	SMF119AP_TSICOutError	4	Binary	Number of outbound ICMP error messages

Table 191. ICMP statistics section (continued)

Offset	Name	Length	Format	Description
68(X'44')	SMF119AP_TSICOutDstUnreach	4	Binary	Number of outbound ICMP destination unreachable messages
72(X'48')	SMF119AP_TSICOutTimeExcd	4	Binary	Number of outbound ICMP time exceeded messages
76(X'4C')	SMF119AP_TSICOutParmProb	4	Binary	Number of outbound ICMP parameter problem messages
80(X'50')	SMF119AP_TSICOutSrcQuench	4	Binary	Number of outbound ICMP source quench messages
84(X'54')	SMF119AP_TSICOutRedirect	4	Binary	Number of outbound ICMP redirect messages
88(X'58')	SMF119AP_TSICOutEcho	4	Binary	Number of outbound ICMP echo request messages
92(X'5C')	SMF119AP_TSICOutEchoRep	4	Binary	Number of outbound ICMP echo reply messages
96(X'60')	SMF119AP_TSICOutTstamp	4	Binary	Number of outbound ICMP timestamp request messages
100(X'64')	SMF119AP_TSICOutTstampRep	4	Binary	Number of outbound ICMP timestamp reply messages
104(X'68')	SMF119AP_TSICOutAddrMask	4	Binary	Number of outbound ICMP address mask request messages
108(X'6C')	SMF119AP_TSICOutAddrMRep	4	Binary	Number of outbound ICMP address mask reply messages

Table 192 shows the IPv6 IP statistics section:

Table 192. IPv6 IP statistics section

Offset	Name	Length	Format	Description
0 (X'00')	SMF119AP_TSP6Duration	8	Binary	Duration of recording interval in microseconds, where bit 51 is equivalent to one microsecond
8(X'08')	SMF119AP_TSP6RecData	4	Binary	Number of IPv6 datagrams received
12(X'0C')	SMF119AP_TSP6DscData	4	Binary	Number of input IPv6 datagrams discarded due to errors in their IP header
16(X'10')	SMF119AP_TSP6DscAddr	4	Binary	Number of input IPv6 datagrams discarded because the IP address in their IP header's destination field was not valid
20(X'14')	SMF119AP_TSP6AttFwdData	4	Binary	Number of attempts to forward IPv6 datagrams
24(X'18')	SMF119AP_TSP6DscDUnkPr	4	Binary	Number of IPv6 datagrams discarded because of an unknown or unsupported protocol
28(X'1C')	SMF119AP_TSP6DscDOth	4	Binary	Number of input IPv6 datagrams discarded that are not accounted for in another input discard counter

Table 192. IPv6 IP statistics section (continued)

Offset	Name	Length	Format	Description
32(X'20')	SMF119AP_TSP6DlvData	4	Binary	Number of IPv6 datagrams delivered
36(X'24')	SMF119AP_TSP6XData	4	Binary	Number of IPv6 datagrams transmitted
40(X'28')	SMF119AP_TSP6XDscOth	4	Binary	Number of IPv6 outbound datagrams discarded, due to reasons other than no route being available
44(X'2C')	SMF119AP_ TSP6XDscRoute	4	Binary	Number of IPv6 outbound datagrams discarded, due to no route being available
48(X'30')	SMF119AP_TSP6Timeouts	4	Binary	Number of IPv6 reassembly timeouts
52(X'34')	SMF119AP_ TSP6RecDRsmb	4	Binary	Number of received IPv6 datagrams requiring reassembly
56(X'38')	SMF119AP_TSP6Rsmb	4	Binary	Number of received IPv6 datagrams reassembled
60(X'3C')	SMF119AP_TSP6FailRsmb	4	Binary	Number of failed reassembly attempts on IPv6 datagrams
64(X'40')	SMF119AP_TSP6RecFgmt	4	Binary	Number of fragmented IPv6 datagrams received
68(X'44')	SMF119AP_ TSP6DscDFgmt	4	Binary	Number of IPv6 datagrams discarded due to fragmentation failure
72(X'48')	SMF119AP_TSP6XFgmt	4	Binary	Number of IPv6 datagram fragments generated
76(X'4C')	SMF119AP_ TSP6RouteDisc	4	Binary	Number of IPv6 routing discards

Table 193 shows the IPv6 ICMP statistics section:

Table 193. IPv6 ICMP statistics section

Offset	Name	Length	Format	Description
0 (X'00')	SMF119AP_TSC6Duration	8	Binary	Duration of recording interval in microseconds, where bit 51 is equivalent to one microsecond
8(X'08')	SMF119AP_TSC6InMsg	4	Binary	Number of inbound IPv6 ICMP messages
12(X'0C')	SMF119AP_TSC6InError	4	Binary	Number of inbound IPv6 ICMP error messages
16(X'10')	SMF119AP_ TSC6InDstUnreach	4	Binary	Number of inbound IPv6 ICMP destination unreachable messages
20(X'14')	SMF119AP_ TSC6InTimeExcd	4	Binary	Number of inbound IPv6 ICMP time exceeded messages
24(X'18')	SMF119AP_ TSC6InParmProb	4	Binary	Number of inbound IPv6 ICMP parameter problem messages
28(X'1C')	SMF119AP_ TSC6InAdmProhib	4	Binary	Number of inbound IPv6 ICMP administratively prohibited messages
32(X'20')	SMF119AP_ TSC6InPktTooBig	4	Binary	Number of inbound IPv6 ICMP packet too big messages

Table 193. IPv6 ICMP statistics section (continued)

Offset	Name	Length	Format	Description
36(X'24')	SMF119AP_TSC6InEcho	4	Binary	Number of inbound IPv6 ICMP echo request messages
40(X'28')	SMF119AP_ TSC6InEchoRep	4	Binary	Number of inbound IPv6 ICMP echo reply messages
44(X'2C')	SMF119AP_ TSC6InRtSolicit	4	Binary	Number of inbound IPv6 ICMP router solicitation messages
48(X'30')	SMF119AP_TSC6InRtAdv	4	Binary	Number of inbound IPv6 ICMP router advertisement messages
52(X'34')	SMF119AP_ TSC6InNbSolicit	4	Binary	Number of inbound IPv6 ICMP neighbor solicitation messages
56(X'38')	SMF119AP_TSC6InNbAdv	4	Binary	Number of inbound IPv6 ICMP neighbor advertisement messages
60(X'3C')	SMF119AP_ TSC6InRedirect	4	Binary	Number of inbound IPv6 ICMP redirect messages
64(X'40')	SMF119AP_TSC6InGrpMemQry	4	Binary	Number of inbound IPv6 ICMP multicast listener discovery membership query messages
68(X'44')	SMF119AP_ TSC6InGrpMemRsp	4	Binary	Number of inbound IPv6 ICMP multicast listener discovery membership reply messages
72(X'48')	SMF119AP_ TSC6InGrpMemRed	4	Binary	Number of inbound IPv6 ICMP multicast listener discovery membership reduction messages
76(X'4C')	SMF119AP_TSC6OutMsg	4	Binary	Number of outbound IPv6 ICMP messages
80 (X'50')	SMF119AP_TSC6OutError	4	Binary	Number of outbound IPv6 ICMP error messages
84 (X'54')	SMF119AP_ TSC6OutDstUnrch	4	Binary	Number of outbound IPv6 ICMP destination unreachable messages
88 (X'58')	SMF119AP_ TSC6OutTimeExcd	4	Binary	Number of outbound IPv6 ICMP time exceeded messages
92 (X'5C')	SMF119AP_ TSC6OutParmProb	4	Binary	Number of outbound IPv6 ICMP parameter problem messages
96 (X'60')	SMF119AP_ TSC6OutAdmProhib	4	Binary	Number of outbound IPv6 ICMP administratively prohibited messages
100 (X'64')	SMF119AP_TSC6OutPktTooBig	4	Binary	Number of outbound IPv6 ICMP packet too big messages
104 (X'68')	SMF119AP_TSC6OutEcho	4	Binary	Number of outbound IPv6 ICMP echo request messages
108 (X'6C')	SMF119AP_ TSC6OutEchoRep	4	Binary	Number of outbound IPv6 ICMP echo reply messages
112 (X'70')	SMF119AP_ TSC6OutRtSolicit	4	Binary	Number of outbound IPv6 ICMP router solicitation messages

Table 193. IPv6 ICMP statistics section (continued)

Offset	Name	Length	Format	Description
116 (X'74')	SMF119AP_TSC6OutRtAdv	4	Binary	Number of outbound IPv6 ICMP router advertisement messages
120 (X'78')	SMF119AP_ TSC6OutNbSolicit	4	Binary	Number of outbound IPv6 ICMP neighbor solicitation messages
124 (X'7C')	SMF119AP_ TSC6OutNbAdv	4	Binary	Number of outbound IPv6 ICMP neighbor advertisement messages
128 (X'80')	SMF119AP_ TSC6OutRedirect	4	Binary	Number of outbound IPv6 ICMP redirect messages
132 (X'84')	SMF119AP_ TSC6OutGrpMemQry	4	Binary	Number of outbound IPv6 ICMP multicast listener discovery membership query messages
136 (X'88')	SMF119AP_ TSC6OutGrpMemRsp	4	Binary	Number of outbound IPv6 ICMP multicast listener discovery membership report messages
140 (X'8C')	SMF119AP_ TSC6OutGrpMemRed	4	Binary	Number of outbound IPv6 ICMP multicast listener discovery membership reduction messages

Table 194 shows the storage statistics section.

Table 194. Storage statistics section

Offset	Name	Length	Format	Description
0(X'0')	SMF119AP_TSSTECSACurrent	8	Binary	Current number of ECSA storage bytes allocated
8(X'8')	SMF119AP_TSSTECSAFree	8	Binary	Current number of ECSA storage bytes allocated but not in use
16(X'10')	SMF119AP_TSSTPrivateCurrent	8	Binary	Current number of authorized private subpool storage bytes allocated
24(X'18')	SMF119AP_TSSTPrivateFree	8	Binary	Current number of authorized private subpool storage bytes allocated but not in use.

Interface statistics record (subtype 6)

The Interface statistics record is collected at user specified intervals. The record provides data about user-defined interfaces (LINK or INTERFACE statements), one interface specific section per defined interface. Only non-VIPA and non-loopback interfaces are included in the SMF record, and any interface in the process of being deleted from the stack at the time of interval reporting is likewise ignored.

Each interface specific section reports statistical data about the interface for the previous recording interval. To determine a cumulative value for a given statistic

reported, the user must sum the values reported for the statistic in the individual Interface statistics interval records. If interface statistics recording is turned off dynamically, or the TCP stack terminates, a final interface statistics record is generated to report close-out data. If a given LINK or INTERFACE statement is deleted during a recording interval, any data related to that interface during the recording interval is lost (for example, is not reported in the next interval record).

Depending on the number of interfaces, this report can be spread across multiple records, in which case the self-defining section for each record specifies the content layout of that particular record.

There is no Type 118 record equivalent to the link interface statistics record.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the interface statistics record, the TCP/IP stack identification section indicates IP as the subcomponent and one of the six possible interval record reason settings, depending on if the reporting is due to interval expiration, statistics collection termination, or collection shutdown, and whether one or more physical records are needed to report all the interface statistics.

Table 195 shows the interface statistics record self-defining section:

Table 195. Interface statistics record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 6(X'6')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (3)
26(X'1A')		2	Binary	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 first interface section
40(X'28')	SMF119S1Len	2	Binary	Length of each interface section
42(X'2A')	SMF119S1Num	2	Binary	Number of interface sections
44 (X'2C')	SMF119S2Off	4	Binary	Offset to first IPv6 additional HOME IP address section
48 (X'30')	SMF119S2Len	4	Binary	Length of each IPv6 additional HOME IP address section
50 (X'32')	SMF119S2Num	2	Binary	Number of IPv6 additional HOME IP address sections

Table 196 on page 828 shows the interface statistics specific record (one per LINK or INTERFACE definition):

Table 196. Interface statistics section

Offset	Name	Length	Format	Description
0(X'0')	SMF119SP_IFDuration	8	Binary	Duration of recording interval in microseconds, where bit 51 is equivalent to one microsecond
8(X'8')	SMF119IS_IFLnkHome	16	Binary	Interface HOME address. For IPv6 interfaces, additional addresses might be specified in subsequent HOME IP address sections.
24(X'18')	SMF119IS_IFName	16	EBCDIC	Link or interface name
40(X'28')	SMF119IS_IFDevName	16	EBCDIC	Device name
56(X'38')	SMF119IS_IFDesc	18	EBCDIC	Interface Description (TCPIP PROFILE keyword for LINK or INTERFACE type.) Possible values include: • ATM • CDLC • CTC • ETHERnet • ETHEROR802.3 • FDDI • HCH • IBMTR • IP • IPAQENET • IPAQIDIO • IPAQTR • MPCPTP • OSAENET • OSAFDDI • SAMEHOST • Unknown • 802.3 • IPAQENET6 • IPAQIDIO6 • MPCPTP6 • IPAQIQDX • IPAQIQDX6
74(X'4A')		2	Binary	Reserved
76(X'4C')	SMF119IS_IFActualMtu	4	Binary	MTU size
80(X'50')	SMF119IS_IFSPeed	4	Binary	Speed **Guideline:** If the interface speed exceeds X'FFFFFFFF', then this field contains X'FFFFFFFF'. If this field contains X'FFFFFFFF', then use the SMF119IS_IFHSpeed field to determine the interface speed.
84(X'54')	SMF119IS_IFHSpeed	4	Binary	HSpeed

Table 196. Interface statistics section (continued)

Offset	Name	Length	Format	Description
88(X'58')	SMF119IS_IFInBytes	8	Binary	Number of inbound bytes
96(X'60')	SMF119IS_IFInUniC	8	Binary	Number of inbound Unicast packets
104(X'68')	SMF119IS_IFInBroadC	8	Binary	Number of inbound broadcast packets
112(X'70')	SMF119IS_IFInMultiC	8	Binary	Number of inbound multicast packets
120(X'78')	SMF119IS_IFInDisc	4	Binary	Number of inbound discarded packets
124(X'7C')	SMF119IS_IFInError	4	Binary	Number of inbound packets in error
128(X'80')	SMF119IS_IFInUProt	4	Binary	Number of inbound packets with unknown protocol.
132(X'84')	SMF119IS_IFOutBytes	8	Binary	Number of outbound bytes
140(X'8C')	SMF119IS_IFOutUniC	8	Binary	Number of outbound Unicast packets
148(X'94')	SMF119IS_IFOutBroadC	8	Binary	Number of outbound broadcast packets
156(X'9C')	SMF119IS_IFOutMultiC	8	Binary	Number of outbound multicast packets
164(X'A4')	SMF119IS_IFOutDisc	4	Binary	Number of outbound discarded packets
168(X'A8')	SMF119IS_IFOutError	4	Binary	Number of outbound packets in error
172(X'AC')	SMF119IS_IFOQL	4	Binary	Current output queue length
176(X'B0')	SMF119IS_IfIQDXName	16	EBCDIC	For IPAQENET and IPAQENET6 interfaces that are defined with CHPIDTYPE OSX and with an associated IQDX interface, this field is the associated IQDX interface name. Otherwise, this field is blank and the following four counters are not valid.
192(X'C0')	SMF119IS_IfInIQDXBytes	8	Binary	Number of inbound bytes that were received over the associated IQDX interface. This field is valid only if the SMF119IS_IfIQDXName field is not blank.
196(X'C4')	SMF119IS_IfInIQDXUniC	8	Binary	Number of inbound Unicast packets that were received over the associated IQDX interface. This field is valid only if the SMF119IS_IfIQDXName field is not blank.
200(X'C8')	SMF119IS_IfOutIQDXBytes	8	Binary	Number of outbound bytes that were sent over the associated IQDX interface. This field is valid only if the SMF119IS_IfIQDXName field is not blank.
204(X'CC')	SMF119IS_IfOutIQDXUniC	8	Binary	Number of outbound Unicast packets that were sent over the associated IQDX interface. This field is valid only if the SMF119IS_IfIQDXName field is not blank.

Table 197 on page 830 shows the HOME IP Address section:

Table 197. HOME IP Address section

Offset	Name	Length	Format	Description
0 (X'0')	SMF119IS_IFAddIntfName	16	EBCDIC	Interface name, used to correlate this additional address to the interface statistics record in Table 196 on page 828
16 (X'10')	SMF119IS_IFAddIntfHome	16	Binary	Additional interface HOME address

Server port statistics record (subtype 7)

The Port Statistics record, as an interval record, periodically records statistics on ports that have been configured with the PORT statement in the TCP/IP PROFILE. All ports that were defined by the PORTRANGE statement, ports for which the RESERVED flag has been set, or ports that were defined by the PORT UNRSV statement are excluded.

Each TCP or UDP port's activity is reported; connection information is provided for TCP ports, and traffic information is provided for UDP ports.

Depending on the number of reserved ports, this report might actually be spread across multiple records.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the server Port Statistics record, the TCP/IP stack identification section indicates STACK as the subcomponent and one of the six possible interval record reason settings, depending on if the reporting is due to interval expiration, statistics collection termination, or collection shutdown, and whether one or more physical records are needed to report all the Port statistics.

Table 198 shows the server port statistics record self-defining section:

Table 198. Server port statistics record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 7(X'7')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (3)
26(X'1A')		2	Binary	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 first TCP server port section
40(X'28')	SMF119S1Len	2	Binary	Length of each TCP server port section
42(X'2A')	SMF119S1Num	2	Binary	Number of TCP server port sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to first UDP server port section

Table 198. Server port statistics record self-defining section (continued)

Offset	Name	Length	Format	Description
48(X'30')	SMF119S2Len	2	Binary	Length of each UDP server port section
50(X'32')	SMF119S2Num	2	Binary	Number of UDP server port sections

Table 199 shows TCP server Port statistics specific section (one per reserved port definition).

Table 199. TCP server port statistics section

Offset	Name	Length	Format	Description
0(X'0')	SMF119SP_TCDuration	8	Binary	Duration of recording interval in microseconds, where bit 51 is equivalent to one microsecond
8(X'8')	SMF119SP_TCRName	8	EBCDIC	Server socket resource name (the name specified on the PORT reservation statement)
16(X'10')	SMF119SP_TCBindIP	16	Binary	For bind-specific port reservations: the local IP address
32(X'20')	SMF119SP_TCPort	2	Binary	Port number
34(X'22')		2	Binary	Reserved
36(X'24')	SMF119SP_TCConn	4	Binary	Number of successful connection establishments
40(X'28')	SMF119SP_TCBinds	4	Binary	Number of socket binds to this port reservation
44(X'2C')	SMF119SP_TCBusySrv	4	Binary	Number of connection requests rejected due to server Busy conditions
48(X'30')	SMF119SP_TCSynAttack	4	Binary	Number of connection requests rejected due to SYN Attack detect conditions
52(X'34')	SMF119SP_TCHighwater	4	Binary	Highest number of active TCP connections
56(X'38')	SMF119SP_TCNumConns	4	Binary	Number of active TCP connections

Table 200 shows the UDP server port statistics record (one per reserved port definition being collected):

Table 200. UDP server port statistics section

Offset	Name	Length	Format	Description
0(X'0')	SMF119SP_UDDuration	8	Binary	Duration of recording interval
8(X'8')	SMF119SP_UDRName	8	EBCDIC	Server socket resource name (the name specified on the PORT reservation statement)
16(X'10')	SMF119SP_UDBindIP	16	Binary	For bind-specific port reservations: the local IP address
32(X'20')	SMF119SP_UDPort	2	Binary	Port number
34(X'22')		2	Binary	Reserved
36(X'24')	SMF119SP_UDIDgrams	8	Binary	Number of inbound UDP datagrams to server port

Table 200. UDP server port statistics section (continued)

Offset	Name	Length	Format	Description
44(X'2C')	SMF119SP_UDODgrams	8	Binary	Number of outbound UDP datagrams from server port
52(X'34')	SMF119SP_UDIBytes	8	Binary	Number of inbound bytes
60(X'3C')	SMF119SP_UDOBytes	8	Binary	Number of outbound bytes

TCP/IP stack start/stop record (subtype 8)

The TCP/IP stack start/stop record is collected when an individual TCP/IP stack becomes available for processing and when the stack ceases to be available for processing. The record can be used as a beginning and ending bookmark with which to delineate all other SMF recording activity for a given TCP/IP stack.

Guideline: There is no Type 118 record equivalent for the TCP/IP stack start/stop record.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the TCP/IP stack start/stop record, the TCP/IP stack identification section indicates TCP as the subcomponent and X'08' (event record) as the record reason.

Table 201 shows the TCP/IP stack start/stop record self-defining section:

Table 201. TCP/IP stack start/stop record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 8(X'8')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2)
26(X'1A')		2	Binary	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 TCP/IP start/stop section
40(X'28')	SMF119S1Len	2	Binary	Length of TCP/IP start/stop section
42(X'2A')	SMF119S1Num	2	Binary	Number of TCP/IP start/stop sections

Table 202 on page 833 shows the TCP/IP stack start/stop specific section of this SMF record.

Table 202. TCP/IP stack start/stop record section

Offset	Name	Length	Format	Description
0(X'0')	SMF119TC_STType	1	Binary	Event type: • X'80': Stack start up • X'40': Stack termination • X'20': Stack unplanned termination
1(X'1')	SMF119TC_STFlags	1	Binary	Event flags: • X'80': IPv6 supported on this stack • X'40': IPSEC configured on this stack • X'20': IPSEC6 configured on this stack
2(X'2')		2		Reserved
4(X'4')	SMF119TC_STTime	4	Binary	Time of day stack startup or termination
8(X'8')	SMF119TC_STDate	4	Packed	Date of stack startup or termination
12(X'C')	SMF119TC_STECSAMax	8	Binary	Maximum number of ECSA storage bytes allocated since the TCP/IP stack was started
20(X'14')	SMF119TC_STECSALimit	8	Binary	Maximum number of ECSA storage bytes allowed, as specified on the GLOBALCONFIG statement in the TCP/IP profile. The value 0 indicates that there is no limit.
28(X'1C')	SMF119TC_STPrivateMax	8	Binary	Maximum number of authorized private subpool storage bytes allocated since the TCP/IP stack was started.
36(X'24')	SMF119TC_STPrivateLimit	8	Binary	Maximum number of authorized private subpool storage bytes allowed, as specified on the GLOBALCONFIG statement in the TCP/IP profile. The value 0 indicates that there is no limit.

UDP socket close record (subtype 10)

The UDP socket close record is collected whenever a UDP socket is closed (note that this is not collected for individual datagrams sent using the sendto API call). This record contains pertinent information about the socket, such as timestamps for its opening and closing, and bytes flowing through the socket.

Guidelines:

- The socket's partner information is contained in this record; however, this only documents the partner at the time of socket close. Hence, this information would be more meaningful for a client UDP application than a server UDP application.
- Because this record is generated for every single UDP socket, this can generate significant load on a server and rapidly fill the SMF data sets. Care should be exercised in its use.
- The local IP address is 0.0.0.0 unless the application explicitly binds to a local IP address on this socket.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the UDP socket close record, the TCP/IP Sstack identification section indicates UDP as the subcomponent and X'08' (event record) as the record reason.

Table 203 shows the UDP socket close record self-defining section:

Table 203. UDP socket close record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 10(X'A')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2)
26(X'1A')		2	Binary	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 UDP socket close section
40(X'28')	SMF119S1Len	2	Binary	Length of UDP socket close section
42(X'2A')	SMF119S1Num	2	Binary	Number of UDP socket close sections

Table 204 shows the UDP socket close specific section of this SMF record.

Table 204. UDP socket close record section

Offset	Name	Length	Format	Description
0(X'0')	SMF119UD_UCRname	8	EBCDIC	UDP socket resource name (address space name of address space that opens this socket)
8(X'8')	SMF119UD_UCConnID	4	Binary	UDP socket resource ID (connection ID)
12(X'C')	SMF119UD_UCSubTask	4	Binary	Subtask ID. This is the task TCB for the task owning the socket.
16(X'10')	SMF119UD_UCOTime	4	Binary	Time of day of socket open
20(X'14')	SMF119UD_UCODate	4	Packed	Date of socket open
24(X'18')	SMF119UD_UCCTime	4	Binary	Time of day of socket close
28(X'1C')	SMF119UD_UCCDate	4	Packed	Date of socket close
32(X'20')	SMF119UD_UCRIP	16	Binary	Remote IP of last datagram received on socket
48(X'30')	SMF119UD_UCLIP	16	Binary	Local IP address at time of socket close
64(X'40')	SMF119UD_UCRPort	2	Binary	Remote port of last datagram received on socket
66(X'42')	SMF119UD_UCLPort	2	Binary	Local port number at time of socket close

Table 204. UDP socket close record section (continued)

Offset	Name	Length	Format	Description
68(X'44')	SMF119UD_UCType	1	Binary	UDP Socket Type: • X'01': Standard • X'02': Enterprise Extender
69(X'45')	SMF119UD_UCReason	1	Binary	Reason for socket close: • X'01': Normal • X'02': Abnormal: application error or stack termination
70(X'46')		2	Binary	Reserved
72(X'48')	SMF119UD_UCInDgrams	8	Binary	Number of inbound UDP datagrams
80(X'50')	SMF119UD_UCOutDgrams	8	Binary	Number of outbound UDP datagrams
88(X'58')	SMF119UD_UCInBytes	8	Binary	Number of inbound bytes
96(X'60')	SMF119UD_UCOutBytes	8	Binary	Number of outbound bytes

TN3270E Telnet server SNA session initiation record (subtype 20)

The Type 119 TN3270E Telnet server (Telnet) SNA session initiation record is collected when the z/OS TN3270E Telnet server establishes a SNA session with a Telnet client. The information in this record relates to a given LU-LU session, and not to the TCP/IP Telnet connection; for example, if multiple LU-LU sessions use the same Telnet connection, separate SNA session initiation records for each LU-LU session are reported.

The Type 119 Telnet SNA session initiation record is collected at the same point in session processing as the equivalent Type 118 TN3270E Telnet server "LOGN" SMF record.

Guideline: Because the Telnet SNA session initiation record contains a subset of the information that the Telnet SNA session termination record contains, you should collect only the Telnet SNA session termination records.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the Telnet SNA session initiation record, the TCP/IP stack identification section indicates TN3270S as the subcomponent and X'08' (event record) as the record reason. Table 205 shows the Telnet SNA initiation-specific section of this SMF record.

Table 205. TN3270E Telnet server SNA session initiation record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 20(X'14')
Self-defining section				
24(X'0')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2)
26(X'18')		2	Binary	Reserved
28(X'1A')	SMF119IDOff	4	Binary	Offset to TCP/IP identification section
32(X'1C')	SMF119IDLen	2	Binary	Length of TCP/IP identification section

Table 205. TN3270E Telnet server SNA session initiation record self-defining section (continued)

Offset	Name	Length	Format	Description
34(X'20')	SMF119IDNum	2	Binary	Number of TCP/IP identification sections
36(X'24')	SMF119S1Off	4	Binary	Offset to Telnet SNA session initiation section
40(X'28')	SMF119S1Len	2	Binary	Length of Telnet SNA session initiation section
42(X'2A')	SMF119S1Num	2	Binary	Number of Telnet SNA session initiation sections

Table 206 shows the Telnet SNA session initiation section (TCP/IP identification section). TN3270S is the subcomponent, and X'08' (event record) is the record reason.

Table 206. TN3270E Telnet server SNA session initiation section

Offset	Name	Length	Format	Description
0(X'0')	SMF119TN_NILU	8	EBCDIC	Telnet LU name
8(X'8')	SMF119TN_NIAppl	8	EBCDIC	Host application name
16(X'10')	SMF119TN_NILdev	4	Binary	Telnet server internal logical device number
20(X'14')	SMF119TN_NIRIP	16	Binary	Remote IP address
36(X'24')	SMF119TN_NILIP	16	Binary	Local IP address
52(X'30')	SMF119TN_NIRPort	2	Binary	Remote (client) port number
54(X'34')	SMF119TN_NILPort	2	Binary	Local port number
56(X'38')	SMF119TN_NITime	4	Binary	Time of day of session initiation
60(X'3C')	SMF119TN_NIDate	4	Packed	Date of session initiation

TN3270E Telnet server SNA session termination record (subtype 21)

The Type 119 TN3270E Telnet server (Telnet) SNA session termination record is collected when the z/OS TN3270E Telnet server terminates a SNA session with a Telnet client. The information in this record is associated with a given LU-LU session, and not to the TCP/IP Telnet connection; for example, if multiple LU-LU sessions use the same Telnet connection, separate SNA session termination records are reported for each LU-LU session.

The Type 119 Telnet SNA Session termination record is collected at the same point in session processing as the equivalent Type 118 TN3270E Telnet server "LOGF" SMF record.

Guideline: Because the Telnet SNA session termination record contains a superset of the information that the Telnet SNA session initiation record contains, you should collect only the Telnet SNA session termination records.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the Telnet SNA session termination record, the TCP/IP stack identification section indicates TN3270S as the subcomponent and X'08' (event record) as the record reason.

Table 207 shows the Telnet SNA session termination record self-defining section:

Table 207. TN3270E Telnet server SNA session termination record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 21(X'15')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (5)
26(X'1A')		2	Binary	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 Telnet SNA session termination section
40(X'28')	SMF119S1Len	2	Binary	Length of Telnet SNA session termination section
42(X'2A')	SMF119S1Num	2	Binary	Number of Telnet SNA session termination sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to TN3270 server host name section
48(X'30')	SMF119S2Len	2	Binary	Length of TN3270 server host name section
50(X'32')	SMF119S2Num	2	Binary	Number of TN3270 server host name sections
52(X'34')	SMF119S3Off	4	Binary	Offset to TN3270 server session performance data section
56(X'38')	SMF119S3Len	2	Binary	Length of TN3270 server session performance data section
58(X'3A')	SMF119S3Num	2	Binary	Number of TN3270 server session performance data sections
60(X'3C')	SMF119S4Off	4	Binary	Offset to TN3270 server session time bucket performance data section
64(X'40')	SMF119S4Len	2	Binary	Length of TN3270 server session time bucket performance data section
66(X'42')	SMF119S4Num	2	Binary	Number of TN3270 server session time bucket performance data sections

Table 208 shows the Telnet SNA session termination section of this SMF record.

Table 208. TN3270E Telnet server SNA session termination section

Offset	Name	Length	Format	Description
0(X'0')	SMF119TN_NTLU	8	EBCDIC	Telnet LU name
8(X'8')	SMF119TN_NTAppl	8	EBCDIC	Host application name
16(X'10')	SMF119TN_NTLdev	4	Binary	Telnet internal logical device number
20(X'14')	SMF119TN_NTRIP	16	Binary	Remote (client) IP address

Table 208. TN3270E Telnet server SNA session termination section (continued)

Offset	Name	Length	Format	Description
36(X'24')	SMF119TN_NTLIP	16	Binary	Local (Telnet) IP address
52(X'34')	SMF119TN_NTRPort	2	Binary	Remote (client) port number
54(X'36')	SMF119TN_NTLPort	2	Binary	Local (Telnet) port number
56(X'38')	SMF119TN_NTHostNm	8	EBCDIC	TCP/IP Host name
64(X'40')	SMF119TN_NTInByte	8	Binary	Inbound byte count
72(X'48')	SMF119TN_NTOutByte	8	Binary	Outbound byte count
80(X'50')	SMF119TN_NTiTime	4	Binary	Time of session initiation
84(X'54')	SMF119TN_NTiDate	4	Packed	Date of session initiation
88(X'58')	SMF119TN_NTtTime	4	Binary	Time of session termination
92(X'5C')	SMF119TN_NTtDate	4	Packed	Date of session termination
96(X'60')	SMF119TN_NTDur	4	Binary	Session duration in units of 1/100 seconds
100(X'64')	SMF119TN_NTSType	1	Binary	Telnet session type: • 0: UNKNOWN • 1: TN3270 • 2: TN3270E • 3: LINEMODE • 4: DBCSTRANSFORM • 5: BINARY
101(X'65')	SMF119TN_NTLUSel	1	Binary	Telnet LU selection method: • 0: LU chosen by server • 1: LU chosen by client

Table 208. TN3270E Telnet server SNA session termination section (continued)

Offset	Name	Length	Format	Description
102(X'66')	SMF119TN_NTSSL	1	Binary	SSL status: • 0: No SSL session • 1: Server authentication only • 2: Server and client authentication (REQUIRED/SSLCERT): – If AT-TLS policy (REQUIRED), then check SAF, and user ID is not required to be returned. – If TN profile control (SSLCERT), then no SAF. • 3: Server and client authentication (SAFCHECK/SAFCERT): – If AT-TLS policy (SAFCHECK), then SAF check requires user ID returned. – If TN profile control (SAFCERT), then SAF check requires user ID returned. • 4: Server and client authentication (FULL): – AT-TLS policy only. Optional client certificate. SSL cert if provided. SAF check, user ID is not required. • 5: Server and client authentication (PASSTHRU) – AT-TLS policy only. Optional client certificate. No SSL cert if provided. SAF check, user ID is not required.
103(X'67')		1	Binary	Reserved
104(X'68')	SMF119TN_NTCopt	1	Binary	Telnet connection options negotiated for this connection: • 1000 0000: TN3270E • 0100 0000: Terminal type • 0010 0000: End of Record • 0001 0000: Transmit binary • 0000 1000: Echos • 0000 0100: Suppress go ahead • 0000 0010: Timemark • 0000 0001: New Environment TN3270E connection options negotiated for this connection. More than one of these options can be set.
105(X'69')		1	Binary	Reserved

Table 208. TN3270E Telnet server SNA session termination section (continued)

Offset	Name	Length	Format	Description
106(X'6A')	SMF119TN_NT32opt	2	Binary	TN3270E connection options negotiated for this connection. **First Byte**: • 1000 0000: Bind image • 0100 0000: SysRequest • 0010 0000: Responses • 0001 0000: SCS control codes • 0000 1000: DCS control codes • 0000 0100: Contention Resolution • 0000 0010: FMH Support • 0000 0001: SNA Sense Support **Second Byte**: • 1000 0000: Suppress Header Byte Doubling • 0xxx xxxx: Reserved TN3270E connection options negotiated for this connection. More than one of these options can be set.
108(X'6C')	SMF119TN_NTRCode	8	EBCDIC	Session termination reason code. The values in this field are the same as those displayed in message EZZ6034I as value for the object variable.
116(X'74')	SMF119TN_NTLMode	8	EBCDIC	SNA logmode
124(X'7C')	SMF119TN_NTDevt	20	EBCDIC	Telnet device type

Table 209 shows the Telnet server host name section. This section is optional and is present if HNGROUP was applicable for this connection.

Table 209. TN3270E Telnet server host name section

Offset	Name	Length	Format	Description
0(X'0')	SMF119TN_NTHostname	n	EBCDIC	Host name associated with this session

Table 210 shows the TN3270E Telnet server round trip performance section. This section is optional and is present when performance data is being collected for this connection as a result of a MONITORGROUP being mapped to this connection.

Table 210. TN3270E Telnet server Round Trip Performance section

Offset	Name	Length	Format	Description
0(X'0')	SMF119TN_NTRRts	4	Binary	Sum of round trip times for this session in milliseconds
4(X'4')	SMF119TN_NTRIPRts	4	Binary	Sum of IP portion of round trip times for this session in milliseconds

Table 210. TN3270E Telnet server Round Trip Performance section (continued)

Offset	Name	Length	Format	Description
8(X'8')	SMF119TN_NTRCountTrans	4	Binary	Count of transactions used to measure round trip times for this session
12(X'C')	SMF119TN_NTRCountIP	4	Binary	Count of IP transactions used to measure the IP portion of the round trip time
16(X'10')	SMF119TN_NTRElapsRndTrpSq	8	Binary	The sum of the square of each round trip time
24(X'18')	SMF119TN_NTRElapsIpRtSq	8	Binary	The sum of the square of each IP portion of round trip time
32(X'20')	SMF119TN_NTRElapsSnaRtSq	8	Binary	The sum of the square of each SNA portion of round trip time
40(X'28')	SMF119TN_NTRGrpIndex	4	Binary	The index into the master MonitorGroup table this connection is using
44(X'2C')	SMF119TN_NTRDR	1	Binary	Indicator how IP trip time is measured: • '80'x: Definite Response used • '40'x: Timemark used
45(X'2D')		3	Binary	Reserved

Table 211 shows the Telnet SNA session termination time bucket performance data section. This section is optional and is present if performance data is being collected for this connection as a result of a MONITORGROUP being mapped to this connection and time bucket data has been requested. The upper boundary of one bucket is the lower boundary of the next bucket. A transaction is added to a bucket when its round trip time falls within the bounds of that bucket.

Table 211. TN3270E Telnet server time bucket performance section

Offset	Name	Length	Format	Description
0(X'0')	SMF119TN_NTBucketBndry1	4	Binary	Upper boundary for bucket 1 in milliseconds
4(X'4')	SMF119TN_NTBucketBndry2	4	Binary	Upper boundary for bucket 2 in milliseconds
8(X'8')	SMF119TN_NTBucketBndry3	4	Binary	Upper boundary for bucket 3 in milliseconds
12(X'C')	SMF119TN_NTBucketBndry4	4	Binary	Upper boundary for bucket 4 in milliseconds
16(X'10')	SMF119TN_NTBucket1Rts	4	Binary	Number of transactions with a round trip time meeting bucket 1 criteria
20(X'14')	SMF119TN_NTBucket2Rts	4	Binary	Number of transactions with a round trip time meeting bucket 2 criteria
24(X'18')	SMF119TN_NTBucket3Rts	4	Binary	Number of transactions with round trip time meeting bucket 3 criteria

Table 211. TN3270E Telnet server time bucket performance section (continued)

Offset	Name	Length	Format	Description
28(X'1C')	SMF119TN_NTBucket4Rts	4	Binary	Number of transactions with a round trip time meeting bucket 4 criteria
32(X'20')	SMF119TN_NTBucket5Rts	4	Binary	Number of transactions with a round trip time that exceeds bucket 4 time

TSO Telnet client connection initiation record (subtype 22)

The TSO Telnet client connection Initiation record is collected at the establishment of a connection using the TSO Telnet client. This denotes the connection, rather than a particular session. This record contains pertinent information about the connection available at the time of its opening.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the TSO Telnet client connection initiation record, the TCP/IP stack identification section indicates TN3270C as the subcomponent and X'08' (event record) as the record reason.

Table 212 shows the TSO Telnet client connection initiation record self-defining section:

Table 212. TSO Telnet client connection initiation section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 22(X'16')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2)
26(X'1A')		2	Binary	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 TSO Telnet client connection initiation section
40(X'28')	SMF119S1Len	2	Binary	Length of TSO Telnet client connection initiation section
42(X'2A')	SMF119S1Num	2	Binary	Number of TSO Telnet client connection initiation sections

Table 213 shows the TSO Telnet client connection initiation identification section of this SMF record.

Table 213. TSO Telnet client connection initiation record TCP/IP identification section

Offset	Name	Length	Format	Description
0(X'0')	SMF119TN_CIRIP	16	Binary	Remote (server) IP address
16 (X'10')	SMF119TN_CILIP	16	Binary	Local IP address

Table 213. TSO Telnet client connection initiation record TCP/IP identification section (continued)

Offset	Name	Length	Format	Description
32(X'20')	SMF119TN_CIRPort	2	Binary	Remote (server) port number
34 (X'22')	SMF119TN_CILPort	2	Binary	Local port number
36 (X'24')	SMF119TN_CITime	4	Binary	Time of day of session initiation
40(X'28')	SMF119TN_CIDate	4	Packed	Date of session initiation

TSO Telnet client connection termination record (subtype 23)

The TSO Telnet client connection termination record is collected at the termination of a connection using the TSO Telnet client. This denotes the connection, rather than a particular session. This record contains all pertinent information about the connection, such as elapsed time, bytes transferred, and so on.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the TSO Telnet client connection termination record, the TCP/IP stack identification section indicates TN3270C as the subcomponent and X'08' (event record) as the record reason.

Table 214 shows the TSO Telnet client connection termination record self-defining section:

Table 214. TSO Telnet client connection termination record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 23(X'17')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2)
26(X'1A')		2	Binary	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 TSO Telnet client connection termination section
40(X'28')	SMF119S1Len	2	Binary	Length of TSO Telnet client connection termination section
42(X'2A')	SMF119S1Num	2	Binary	Number of TSO Telnet client connection termination sections

Table 215 shows the TSO Telnet client connection termination specific section of this SMF record.

Table 215. TSO Telnet client connection termination section

Offset	Name	Length	Format	Description
0(X'0')	SMF119TN_CTRIP	16	Binary	Remote (server) IP address
16 (X'10')	SMF119TN_CTLIP	16	Binary	Local IP address

Table 215. TSO Telnet client connection termination section (continued)

Offset	Name	Length	Format	Description
32(X'20')	SMF119TN_CTRPort	2	Binary	Remote (server) port number
34 (X'22')	SMF119TN_CTLPort	2	Binary	Local port number
36 (X'24')	SMF119TN_CTNJENode	8	EBCDIC	NJE Node Name
44(X'2C')	SMF119TN_CTInBytes	8	Binary	Inbound byte count
52(X'34')	SMF119TN_CTOutBytes	8	Binary	Outbound byte count
60(X'3C')	SMF119TN_CTiTime	4	Binary	Time of day of session initiation
64(X'40')	SMF119TN_CTiDate	4	Packed	Date of session initiation
68(X'44')	SMF119TN_CTtTime	4	Binary	Time of day of session termination
72(X'48')	SMF119TN_CTtDate	4	Packed	Date of session termination
76(X'4C')	SMF119TN_CTDur	4	Binary	Telnet client session duration in 1/100 seconds
80(X'50')	SMF119TN_CTCOpt	1	Binary	Telnet connection options negotiated for this connection: • x000 0000: Reserved • 0100 0000: Terminal type • 0010 0000: End of record • 0001 0000: Transmit binary • 0000 1000: Echos • 0000 0100: Suppress go ahead • 0000 00xx: Reserved
81(X'51')		3	Binary	Reserved
84(X'54')	SMF119TN_CTDevt	20	EBCDIC	Telnet device type

DVIPA status change record (subtype 32)

The dynamic virtual IP address (DVIPA) status change record is created when a DVIPA is defined and when the status of a DVIPA changes on a TCP/IP stack. This record is not created when the TCP/IP stack leaves the sysplex group and a DVIPA becomes inactive. Creation of this SMF record is controlled by the following:

- DVIPA and NODVIPA parameters on the SMFCONFIG profile statement

  If DVIPA is specified, the record is created and written to the SMF MVS data sets.
- DVIPA and NODVIPA options of the SMFSERVICE parameter on the NETMONITOR profile statement

  If DVIPA is specified, the record is created and written to the real-time TCP/IP network monitoring NMI. For more information about this NMI, see "Real-time TCP/IP network monitoring NMI" on page 564.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the DVIPA status change record, the TCP/IP stack identification section indicates STACK as the subcomponent and X'08' (event record) as the record reason.

Table 216 on page 845 shows the DVIPA status change record self-defining section:

Table 216. DVIPA status change record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 32(X'20')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2)
26(X'1A')		2	Binary	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 DVIPA status change section
40(X'28')	SMF119S1Len	2	Binary	Length of DVIPA status change section
42(X'2A')	SMF119S1Num	2	Binary	Number of DVIPA status change sections

Table 217 shows the DVIPA status change specific section of this SMF record:

Table 217. DVIPA status change section

Offset	Name	Length	Type	Description
0(X'0')	SMF119DV_SCIPAddr4	4	Binary	If SMF119DV_SCFlags_IPv6 is not set, this field contains the IPv4 DVIPA address.
0(X'0')	SMF119DV_SCIPAddr6	16	Binary	If SMF119DV_SCFlags_IPv6 is set, this field contains the IPv6 DVIPA address.
16(X'10')	SMF119DV_SCFlags	1	Binary	Various flags: • X'80', SMF119DV_SCFlags_IPv6: If set, this record describes an IPv6 DVIPA address.
17(X'11')	SMF119DV_SCOrigin	1	Binary	The origin of this DVIPA and how it was configured to the stack: • X'01', SMF119DV_Orig_Unknown A value of X'01' should not occur and represents an error. • X'02', SMF119DV_Orig_Backup • X'03', SMF119DV_Orig_Define • X'04', SMF119DV_Orig_RangeBIND • X'05', SMF119DV_Orig_RangeIOCTL • X'06', SMF119DV_Orig_DistTarget

Table 217. DVIPA status change section (continued)

Offset	Name	Length	Type	Description
18(X'12')	SMF119DV_SCStatus	1	Binary	The status of this DVIPA on the stack: • X'01', SMF119DV_Stat_Unknown A value of X'01' should not occur and represents an error. • X'02', SMF119DV_Stat_Active • X'03', SMF119DV_Stat_Backup • X'04', SMF119DV_Stat_Moving • X'05', SMF119DV_Stat_Quiescing • X'06', SMF119DV_Stat_Deact • X'07', SMF119DV_Stat_DeactLG • X'08', SMF119DV_Stat_DeactAuto • X'09', SMF119DV_Stat_InactLG • X'0A', SMF119DV_Stat_InactAuto
19(X'13')	SMF119DV_SCOptions	1	Binary	Flags indicating DVIPA options specified: • X'80', SMF119DV_Opt_MoveImmed • X'40', SMF119DV_Opt_MoveIdle • X'20', SMF119DV_Opt_MoveNonDis • X'10', SMF119DV_Opt_MoveDisrupt This field does not apply to DVIPAs whose SMF119DV_SCOrigin field is set to SMF119DV_Orig_DistTarget.
20(X'14')	SMF119DV_SCRank	2	Binary	The rank of this stack in the chain of backup stacks for this DVIPA. For entries where the SMF119DV_SCOrigin value is not SMF119DV_Orig_Backup or SMF119DV_Orig_Define, this field does not apply and is set to X'FFFF'.
22(X'16')		2	Binary	Reserved
24(X'18')	SMF119DV_SCActTime	4	Binary	DVIPA activation time, or the time when this DVIPA was activated on the local stack, either because the stack is the owner of the DVIPA or because the stack is a target for this DVIPA.
28(X'1C')	SMF119DV_SCActDate	4	Packed	DVIPA activation date

DVIPA removed record (subtype 33)

The dynamic virtual IP address (DVIPA) removed record is created when a DVIPA is removed from a TCP/IP stack. Creation of this SMF record is controlled by the following:

- DVIPA and NODVIPA parameters on the SMFCONFIG profile statement

  If DVIPA is specified, the record is created and written to the SMF MVS data sets.

- DVIPA and NODVIPA options of the SMFSERVICE parameter on the NETMONITOR profile statement

  If DVIPA is specified, the record is created and written to the real-time TCP/IP network monitoring NMI. For more information about this NMI, see "Real-time TCP/IP network monitoring NMI" on page 564.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the DVIPA removed record, the TCP/IP stack identification section indicates STACK as the subcomponent and X'08' (event record) as the record reason.

Table 218 shows the DVIPA removed record self-defining section:

Table 218. DVIPA removed record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 33(X'21')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2)
26(X'1A')		2	Binary	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 DVIPA removed section
40(X'28')	SMF119S1Len	2	Binary	Length of DVIPA removed section
42(X'2A')	SMF119S1Num	2	Binary	Number of DVIPA removed sections

Table 219 shows the DVIPA removed specific section of this SMF record:

Table 219. DVIPA removed section

Offset	Name	Length	Type	Description
0(X'0')	SMF119DV_RmIPAddr4	4	Binary	If SMF119DV_RmFlags_IPv6 is not set, this field contains the IPv4 DVIPA address.
0(X'0')	SMF119DV_RmIPAddr6	16	Binary	If SMF119DV_RmFlags_IPv6 is set, this field contains the IPv6 DVIPA address.
16(X'10')	SMF119DV_RmFlags	1	Binary	Various flags: • X'80', SMF119DV_RmFlags_IPv6: If set, this record describes an IPv6 DVIPA address.

Table 219. DVIPA removed section (continued)

Offset	Name	Length	Type	Description
17(X'11')	SMF119DV_RmOrigin	1	Binary	The origin of this DVIPA and how it was configured to the stack: • X'01', SMF119DV_Orig_Unknown A value of X'01' should not occur and represents an error. • X'02', SMF119DV_Orig_Backup • X'03', SMF119DV_Orig_Define • X'04', SMF119DV_Orig_RangeBIND • X'05', SMF119DV_Orig_RangeIOCTL • X'06', SMF119DV_Orig_DistTarget
18(X'12')	SMF119DV_RmStatus	1	Binary	The status of this DVIPA on the stack before it was removed: • X'01', SMF119DV_Stat_Unknown A value of X'01' should not occur and represents an error. • X'02', SMF119DV_Stat_Active • X'03', SMF119DV_Stat_Backup • X'04', SMF119DV_Stat_Moving • X'05', SMF119DV_Stat_Quiescing • X'06', SMF119DV_Stat_Deact • X'07', SMF119DV_Stat_DeactLG • X'08', SMF119DV_Stat_DeactAuto • X'09', SMF119DV_Stat_InactLG • X'0A', SMF119DV_Stat_InactAuto
19(X'13')	SMF119DV_RmOptions	1	Binary	Flags indicating DVIPA options specified: • X'80', SMF119DV_Opt_MoveImmed • X'40', SMF119DV_Opt_MoveIdle • X'20', SMF119DV_Opt_MoveNonDis • X'10', SMF119DV_Opt_MoveDisrupt
20(X'14')	SMF119DV_RmRank	2	Binary	The rank of this stack in the chain of backup stacks for this DVIPA. For entries where the SMF119DV_RmOrigin value is not SMF119DV_Orig_Backup or SMF119DV_Orig_Define, this field does not apply and is set to X'FFFF'.
22(X'16')		10	Binary	Reserved

DVIPA target added record (subtype 34)

The dynamic virtual IP address (DVIPA) target added record is created by a sysplex distributor stack when it determines that a designated target stack is active. A separate record is created for each DVIPA and port.

Creation of this SMF record is controlled by the following:

- DVIPA and NODVIPA parameters on the SMFCONFIG profile statement

  If DVIPA is specified, the record is created and written to the SMF MVS data sets.
- DVIPA and NODVIPA options of the SMFSERVICE parameter on the NETMONITOR profile statement

  If DVIPA is specified, the record is created and written to the real-time TCP/IP network monitoring NMI. For more information about this NMI, see "Real-time TCP/IP network monitoring NMI" on page 564.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the DVIPA target added record, the TCP/IP stack identification section indicates STACK as the subcomponent and X'08' (event record) as the record reason.

Table 220 shows the DVIPA target added record self-defining section:

Table 220. DVIPA target added record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 34(X'22')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2)
26(X'1A')		2	Binary	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 DVIPA target added section
40(X'28')	SMF119S1Len	2	Binary	Length of DVIPA target added section
42(X'2A')	SMF119S1Num	2	Binary	Number of DVIPA target added sections

Table 221 shows the DVIPA target added specific section of this SMF record:

Table 221. DVIPA target added section

Offset	Name	Length	Type	Description
0(X'0')	SMF119DV_TAIPAddr4	4	Binary	If SMF119DV_TAFlags_IPv6 is not set, this field contains the IPv4 DVIPA address.

Table 221. DVIPA target added section (continued)

Offset	Name	Length	Type	Description
0(X'0')	SMF119DV_TAIPAddr6	16	Binary	If SMF119DV_TAFlags_IPv6 is set, this field contains the IPv6 DVIPA address.
16(X'10')	SMF119DV_TADxcfAddr4	4	Binary	If SMF119DV_TAFlags_IPv6 is not set, this field contains the IPv4 dynamic XCF address of the target stack that was added.
16(X'10')	SMF119DV_TADxcfAddr6	16	Binary	If SMF119DV_TAFlags_IPv6 is set, this field contains the IPv6 dynamic XCF address of the target stack that was added.
32(X'20')	SMF119DV_TAFlags	1	Binary	Various flags: • X'80', SMF119DV_TAFlags_IPv6: If set, DVIPA address and dynamic XCF address are IPv6. • X'40', SMF119DV_TAFlags_DestIPAll: If set, DESTIP ALL was specified on the VIPADISTRIBUTE DEFINE statement. • X'20', SMF119DV_TAFlags_DynPorts: If set, dynamic ports were specified for this VIPADISTRIBUTE DEFINE statement.
33(X'21')		1	Binary	Reserved
34(X'22')	SMF119DV_TAPort	2	Binary	The DVIPA distributed port number. If dynamic ports are in use for this target, this port number is 0.
36(X'24')		12	Binary	Reserved

DVIPA target removed record (subtype 35)

The dynamic virtual IP address (DVIPA) target removed record is created by a sysplex distributor stack when an active target stack is removed from distribution.

Creation of this SMF record is controlled by the following:

- DVIPA and NODVIPA parameters on the SMFCONFIG profile statement

  If DVIPA is specified, the record is created and written to the SMF MVS data sets.
- DVIPA and NODVIPA options of the SMFSERVICE parameter on the NETMONITOR profile statement

  If DVIPA is specified, the record is created and written to the real-time TCP/IP network monitoring NMI. For more information about this NMI, see “Real-time TCP/IP network monitoring NMI” on page 564.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the DVIPA target removed record, the TCP/IP stack identification section indicates STACK as the subcomponent and X'08' (event record) as the record reason.

Table 222 shows the DVIPA target removed record self-defining section:

Table 222. DVIPA target removed record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 35(X'23')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2)
26(X'1A')		2	Binary	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 DVIPA target removed section
40(X'28')	SMF119S1Len	2	Binary	Length of DVIPA target removed section
42(X'2A')	SMF119S1Num	2	Binary	Number of DVIPA target removed sections

Table 223 shows the DVIPA target removed specific section of this SMF record:

Table 223. DVIPA target removed section

Offset	Name	Length	Type	Description
0(X'0')	SMF119DV_TRIPAddr4	4	Binary	If SMF119DV_TRFlags_IPv6 is not set, this field contains the IPv4 DVIPA address.
0(X'0')	SMF119DV_TRIPAddr6	16	Binary	If SMF119DV_TRFlags_IPv6 is set, this field contains the IPv6 DVIPA address.
16(X'10')	SMF119DV_TRDxcfAddr4	4	Binary	If SMF119DV_TRFlags_IPv6 is not set, this field contains the IPv4 dynamic XCF address of the target stack that was removed.
16(X'10')	SMF119DV_TRDxcfAddr6	16	Binary	If SMF119DV_TRFlags_IPv6 is set, this field contains the IPv6 dynamic XCF address of the target stack that was removed.

Table 223. DVIPA target removed section (continued)

Offset	Name	Length	Type	Description
32(X'20')	SMF119DV_TRFlags	1	Binary	Various flags: • X'80', SMF119DV_TRFlags_IPv6: If set, DVIPA address and dynamic XCF address are IPv6. • X'40', SMF119DV_TRFlags_DestIPAll: If set, DESTIP ALL was specified on the VIPADISTRIBUTE DELETE statement. • X'20', SMF119DV_TRFlags_DynPorts: If set, dynamic ports were specified for this target.
33(X'21')		1	Binary	Reserved
34(X'22')	SMF119DV_TRPort	2	Binary	The DVIPA distributed port number. If dynamic ports are in use for this DVIPA, this port number might be 0.
36(X'24')		12	Binary	Reserved

DVIPA target server started record (subtype 36)

The dynamic virtual IP address (DVIPA) target server started record is created by a sysplex distributor stack when it receives notification from a target stack that a server has opened a listening socket on a distributed port, or that a server has been resumed using the V TCPIP,,SYSPLEX,RESUME command.

Creation of this SMF record is controlled by the following:

- DVIPA and NODVIPA parameters on the SMFCONFIG profile statement

  If DVIPA is specified, the record is created and written to the SMF MVS data sets.
- DVIPA and NODVIPA options of the SMFSERVICE parameter on the NETMONITOR profile statement

  If DVIPA is specified, the record is created and written to the real-time TCP/IP network monitoring NMI. For more information about this NMI, see "Real-time TCP/IP network monitoring NMI" on page 564.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the DVIPA target server started record, the TCP/IP stack identification section indicates STACK as the subcomponent and X'08' (event record) as the record reason.

Table 224 shows the DVIPA target server started record self-defining section:

Table 224. DVIPA target server started record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 36(X'24')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2)

Table 224. DVIPA target server started record self-defining section (continued)

Offset	Name	Length	Format	Description
26(X'1A')		2	Binary	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 DVIPA target server started section
40(X'28')	SMF119S1Len	2	Binary	Length of DVIPA target server started section
42(X'2A')	SMF119S1Num	2	Binary	Number of DVIPA target server started sections

Table 225 shows the DVIPA target server started specific section of this SMF record:

Table 225. DVIPA target server started section

Offset	Name	Length	Type	Description
0(X'0')	SMF119DV_TSSIPAddr4	4	Binary	If SMF119DV_TSSFlags_IPv6 is not set, this field contains the IPv4 DVIPA address.
0(X'0')	SMF119DV_TSSIPAddr6	16	Binary	If SMF119DV_TSSFlags_IPv6 is set, this field contains the IPv6 DVIPA address.
16(X'10')	SMF119DV_TSSDxcfAddr4	4	Binary	If SMF119DV_TSSFlags_IPv6 is not set, this field contains the IPv4 destination XCF address of the target stack on which the target server was added.
16(X'10')	SMF119DV_TSSDxcfAddr6	16	Binary	If SMF119DV_TSSFlags_IPv6 is set, this field contains the IPv6 destination XCF address of the target stack on which the target server was added.
32(X'20')	SMF119DV_TSSFlags	1	Binary	Various flags: • X'80', SMF119DV_TSSFlags_IPv6: If set, DVIPA address and destination XCF address are IPv6.
33(X'21')		1	Binary	Reserved
34(X'22')	SMF119DV_TSSPort	2	Binary	The DVIPA distributed port number.
36(X'24')	SMF119DV_TSSReadyCount	4	Binary	The number of servers on the indicated target stack that are ready to service connection requests for the indicated port.
40(X'28')		8	Binary	Reserved

DVIPA target server ended record (subtype 37)

The dynamic virtual IP address (DVIPA) target server ended record is created by a sysplex distributor stack when it receives notification from a target stack that a server has closed a listening socket on a distributed port, or that a server has been quiesced using the V TCPIP,,SYSPLEX,QUIESCE command.

Creation of this SMF record is controlled by the following:

- DVIPA and NODVIPA parameters on the SMFCONFIG profile statement

  If DVIPA is specified, the record is created and written to the SMF MVS data sets.
- DVIPA and NODVIPA options of the SMFSERVICE parameter on the NETMONITOR profile statement

  If DVIPA is specified, the record is created and written to the real-time TCP/IP network monitoring NMI. For more information about this NMI, see "Real-time TCP/IP network monitoring NMI" on page 564.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the DVIPA target server ended record, the TCP/IP stack identification section indicates STACK as the subcomponent and X'08' (event record) as the record reason.

Table 226 shows the DVIPA target server ended record self-defining section:

Table 226. DVIPA target server ended record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 37(X'25')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2)
26(X'1A')		2	Binary	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 DVIPA target server ended section
40(X'28')	SMF119S1Len	2	Binary	Length of DVIPA target server ended section
42(X'2A')	SMF119S1Num	2	Binary	Number of DVIPA target server ended sections

Table 227 on page 855 shows the DVIPA target server ended specific section of this SMF record:

Table 227. DVIPA target server ended section

Offset	Name	Length	Type	Description
0(X'0')	SMF119DV_TSEIPAddr4	4	Binary	If SMF119DV_TSEFlags_IPv6 is not set, this field contains the IPv4 DVIPA address.
0(X'0')	SMF119DV_TSEIPAddr6	16	Binary	If SMF119DV_TSEFlags_IPv6 is set, this field contains the IPv6 DVIPA address.
16(X'10')	SMF119DV_TSEDxcfAddr4	4	Binary	If SMF119DV_TSEFlags_IPv6 is not set, this field contains the IPv4 destination XCF address of the target stack on which the target server was added.
16(X'10')	SMF119DV_TSEDxcfAddr6	16	Binary	If SMF119DV_TSEFlags_IPv6 is set, this field contains the IPv6 destination XCF address of the target stack on which the target server was added.
32(X'20')	SMF119DV_TSEFlags	1	Binary	Various flags: • X'80', SMF119DV_TSEFlags_IPv6: If set, DVIPA address and destination XCF address are IPv6.
33(X'21')		1	Binary	Reserved
34(X'22')	SMF119DV_TSEPort	2	Binary	The DVIPA distributed port number.
36(X'24')	SMF119DV_TSEReadyCount	4	Binary	The number of servers on the indicated target stack that are ready to service connection requests for the indicated port.
40(X'28')		8	Binary	Reserved

CSSMTP configuration record (CONFIG subtype 48)

This record is written at initialization of the CSSMTP application. The content reflects what is in the actual configuration file and not the result of name resolution. The following MODIFY commands will cause a configuration record to be written if values are changed: MODIFY REFRESH, MODIFY LOG_LEVEL, and MODIFY USEREXIT.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. The TCP/IP identification section indicates CSSMTP as the subcomponent and X'08' (event record) as the record reason.

The field SMF119TI_Stack name is blank unless the -p parameter is used to start the CSSMTP application. If the record is written to NMI, the field SMF119TI_Stack in the NMI record contains the stack name that the record was written to. This is a non-connection oriented SMF record.

Table 228 on page 856 shows the CSSMTP configuration record self-defining section:

Table 228. CSSMTP configuration record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 48(X'30')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (6)
26(X'1A')		2	Binary	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 CSSMTP common information section
40(X'28')	SMF119S1Len	2	Binary	Length of CSSMTP common information section
42(X'2A')	SMF119S1Num	2	Binary	Number of CSSMTP common information sections
44(X'2C')	SMF119S2Off	4	Binary	Offset of CSSMTP configuration section
48(X'30')	SMF119S2Len	2	Binary	Length of CSSMTP configuration section
50(X'32')	SMF119S2Num	2	Binary	Number of CSSMTP configuration sections
52(X'34')	SMF119S3Off	4	Binary	Offset to CSSMTP target server section
56(X'38')	SMF119S3Len	2	Binary	Length of CSSMTP target server section
58(X'36')	SMF119S3Num	2	Binary	Number of CSSMTP target server sections
60(X'3C')	SMF119S4Off	4	Binary	Offset to CSSMTP configuration data section
64(X'40')	SMF119S4Len	2	Binary	Length of CSSMTP configuration data section
66(X'42')	SMF119S4Num	2	Binary	Number of CSSMTP configuration data sections
68(X'44')	SMF119S5Off	4	Binary	Offset to CSSMTP command data section
72(X'48')	SMF119S5Len	2	Binary	Length of CSSMTP command data section
74(X'4A')	SMF119S5Num	2	Binary	Number of CSSSMTP command data sections

Table 229 on page 857 shows the CSSMTP common information section. This section identifies the CSSMTP JOB that created this SMF record. It is found in subtypes 48, 49, 50, 51 and 52.

Table 229. CSSMTP common information

Offset	Name	Length	Type	Description
0(X'0')	SMF119ML_CI	36	STRUCTURE	CSSMTP common information
0(X'0')	SMF119ML_CI_JMR	24	STRUCTURE	Job Management Record. See Standard SMF Record Header in *z/OS MVS System Management Facilities (SMF)* for detailed information about Job Management Record format.
0(X'0')	SMF119ML_CI_JOB	8	EBCDIC	Jobname
8(X'8')	SMF119ML_CI_Entry	4	Binary	Time since midnight, in hundredths of a second, that the reader recognized the CSSMTP JOB card (for this job).
12(X'C')	SMF119ML_CI_EDate	4	Packed	Date when the reader recognized the CSSMTP JOB card (for this job), in the form 0cyydddF.
16 (X'10')	SMF119ML_CI_USEID	8	EBCDIC	User-defined identification field (taken from common exit parameter area, not from USER=parameter on job statement).
24 (X'18')	SMF119ML_CI_EXTWRT	8	EBCDIC	External writer name
32(X'20')	SMF119ML_CI_Jes	4	EBCDIC	JES subsystem name

Table 230 shows the CSSMTP configuration section when CSSMTP was started or from a successful MODIFY REFRESH command. This data section appears as section 2.

Table 230. CSSMTP started or from MODIFY REFRESH command

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_CF	144	STRUCTURE	CSSMTP configuration
0(X'0')	SMF119ML_CF_Flags	4	BIT(32)	Configuration flags
	SMF119ML_CF_Targets		1...	Target servers updated
	SMF119ML_CF_Ntarget		.1..	Non-target data updated
	SMF119ML_CF_LLupdat		..1.	LogLevel updated
	SMF119ML_CF_Warning		...1	Warning issued on update
	SMF119ML_CF_Modify		.... 1...	MODIFY REFRESH update
	SMF119ML_CF_rsvd05		.... .1..	Reserved
	SMF119ML_CF_rsvd06		.... ..1.	Reserved
	SMF119ML_CF_rsvd07		1	Reserved
1(X'1')	SMF119ML_CF_rsvd10		1...	Reserved
	SMF119ML_CF_IpV4ONly		.1..	IPv4 stack. TCP/IP stack is IPV4 only.
	SMF119ML_CF_rsvd1x		..11 1111	Reserved
2(X'2')	SMF119ML_CF_SmfConfg		1...	SMF119 Config 0-No 1-Yes
	SMF119ML_CF_SmfConn		.1..	SMF119 Connect 0-No 1-Yes
	SMF119ML_CF_SmfMail		..1.	SMF119 Mail 0-No 1-Yes
	SMF119ML_CF_SmfSpool		...1	SMF119 Spool 0-No 1-Yes
	SMF119ML_CF_SmfStats		.... 1...	SMF119 Stats 0-No 1-Yes

Table 230. CSSMTP started or from MODIFY REFRESH command (continued)

Offset	Name(Dim)	Length	Type	Description
	SMF119ML_CF_rsvd25		.... .1..	Reserved
	SMF119ML_CF_rsvd26		.... ..1.	Reserved
	SMF119ML_CF_rsvd27		1	Reserved
3(X'3')	SMF119ML_CF_rsvd3x	1	BIT(8)	Reserved
4(X'4')	SMF119ML_CF_CfgPidId	4	Binary	Process ID value
8(X'8')	SMF119ML_CF_BadSpool	4	Binary	Value from the BadSpoolDisp statement: 0-ML_CF_BADSPOOLDISP_HOLD 1-ML_CF_BADSPOOLDISP_DELETE
12(X'C')	SMF119ML_CF_ChkPtSz	4	Binary	Check Point Size Limit (statement CkpPointSizeLimit)
16(X'10')	SMF119ML_CF_ExtWrt	8	EBCDIC	External Writer name (statement ExtWrtName)
24(X'18')	SMF119ML_CF_Tcpip	8	EBCDIC	TCPIP name parameter
32(X'20')	SMF119ML_CF_JesJobSz	4	Binary	JESJobSize
36(X'24')	SMF119ML_CF_JesMsgSz	4	Binary	JESMsgSize
40(X'28')	SMF119ML_CF_LogLevel	4	Binary	LogLevel
44(X'2C')	SMF119ML_CF_Report	4	Binary	Report statement settings: 0-ML_CF_REPORT_SYSOUT 1-ML_CF_REPORT_NONE 2-ML_REPORT_ADMIN
48(X'30')	SMF119ML_CF_RtyCount	4	Binary	Retry count value from the RetryLimit statement
52(X'34')	SMF119ML_CF_RtyIntvl	4	Binary	Retry interval value from the RetryLimit statement
56(X'38')	SMF119ML_CF_AnyCmd	4	Binary	Timeout AnyCmd
60(X'3C')	SMF119ML_CF_ConnRty	4	Binary	Timeout ConnectRetry
64(X'40')	SMF119ML_CF_DataBlk	4	Binary	Timeout DataBlock
68(X'44')	SMF119ML_CF_DataCmd	4	Binary	Timeout DATACmd
72(X'48')	SMF119ML_CF_DataEOM	4	Binary	Timeout DataTerm
76(X'4C')	SMF119ML_CF_InitMsg	4	Binary	Timeout InitalMsg
80(X'50')	SMF119ML_CF_MailCmd	4	Binary	Timeout MAILCmd
84(X'54')	SMF119ML_CF_RCPTCmd	4	Binary	Timeout RCPTCmd
88(X'58')	SMF119ML_CF_ChkPnt	4	Binary	Checkpoint options: 0-ML_CF_CHK_WARMSTART 1-ML_CF_CHK_COLDSTART 2-ML_CF_CHK_NOTAVAILABLE
92(X'5C')	SMF119ML_CF_CfgCP	20	EBCDIC	Configuration file code page from the CSSMTP_CODEPAGE_CONFIG environment variable
112(X'70')	SMF119ML_CF_CodePage	20	EBCDIC	TRANSLATE

Table 230. CSSMTP started or from MODIFY REFRESH command (continued)

Offset	Name(Dim)	Length	Type	Description
132(X'84')	SMF119ML_CF_RtnTo	4	Binary	Undeliverable ReturnToMailFrom 0-No 1-Yes
136(X'88')	SMF119ML_CF_DeadAct	4	Binary	Undeliverable DeadLetterAction 0-ML_CF_DEADLETTERACTION_STORE 1-ML_CF_DEADLETTERACTION_DELETE
140(X'8C')	SMF119ML_CF_UserExit	4	Binary	Userexit version: 0-ML_CF_USEREXIT_NONE - SMTP user exit not configured or not active. 2-ML_CF_USEREXIT_VERSION2 3-ML_CF_USEREXIT_VERSION3
144(X'90')	SMF119ML_CF_ErtAge	4	Binary	Extended retry age (in days)
148(X'94')	SMF119ML_CF_ErtIntvl	4	Binary	Extended retry interval (in minutes)
152(X'98')	SMF119ML_CF_JESSynMax	4	Binary	Maximum number of syntax errors that are acceptable in a JES spool file

Table 231 shows the target servers that are configured. There is one entry for each target server. This data section appears as section 3.

Table 231. CSSMTP target servers

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_TS	36	STRUCTURE	Target Servers
0(X'0')	SMF119ML_TS_IPaddr	16	EBCDIC	IPv6 address (Type=TargetIP)
0(X'0')	SMF119ML_TS_IPPfx	12	EBCDIC	00000000000000000000FFFF - IPv4-mapped address
12(X'C')	SMF119ML_TS_IPaddr4	4	Binary	IpV4 address
16(X'10')	SMF119ML_TS_Port	2	Binary	Connecting target server port number
18(X'12')	SMF119ML_TS_Type	2	Binary	Type of target server 0=TargetIP 1=TargetName, IP address value is always zero 2=TargetMX, IP address value is always zero
20(X'14')	SMF119ML_TS_MsgSize	4	Binary	Maximum message size
24(X'18')	SMF119ML_TS_Secure	4	Binary	Value from the SECURE statement: 0-No 1-Yes
28(X'1C')	SMF119ML_CF_MaxMsg	4	Binary	Maximum number of messages sent per connection

Table 231. CSSMTP target servers (continued)

Offset	Name(Dim)	Length	Type	Description
32(X'20')	SMF119ML_TS_ConnLim	4	Binary	Number of concurrent connections limit

Table 232 describes the variable length data elements in the configuration. This data section appears as section 4.

Table 232. CSSMTP configuration data

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_CD	4	STRUCTURE	Configuration Data
0(X'0')	SMF119ML_CD_Len	2	Binary	Configuration data length (including the length of this header)
2(X'2')	SMF119ML_CD_Key	2	Binary	Configuration data key (See Table 233)
4(X'4')	SMF119ML_CD_Data	0	EBCDIC	Configuration data string

Table 233 describes the value and meaning for the various configuration data strings.

Table 233. CSSMTP configuration data keys

Data type (SMF119ML_CD_Key)	Data length (SMF119ML_CD_Len)	Format	Description (SMF119ML_CD_Data)
SMF119ML_CD_CfgFile (32)	1-1024	EBCDIC	Configuration file name
SMF119ML_CD_CkpFile (33)	1-44	EBCDIC	Checkpoint data set name
SMF119ML_CD_DeadDir (34)	1-512	EBCDIC	Dead letter directory
SMF119ML_CD_LogFile (35)	1-1024	EBCDIC	Log file name
SMF119ML_CD_Madmin1 (36)	1-320	EBCDIC	Mail administrator 1 mailbox
SMF119ML_CD_Madmin2 (37)	1-320	EBCDIC	Mail administrator 2 mailbox
SMF119ML_CD_Madmin3 (38)	1-320	EBCDIC	Mail administrator 3 mailbox
SMF119ML_CD_Madmin4 (39)	1-320	EBCDIC	Mail administrator 4 mailbox
SMF119ML_CD_DomName (40)	1-256	EBCDIC	Domain name
SMF119ML_CD_HostName (41)	1-64	EBCDIC	Host name
SMF119ML_CD_TargSrv1 (42)	1-256	EBCDIC	Target server 1 statement value
SMF119ML_CD_TargSrv2 (43)	1-256	EBCDIC	Target server 2 statement value
SMF119ML_CD_TargSrv3 (44)	1-256	EBCDIC	Target server 3 statement value
SMF119ML_CD_TargSrv4 (45)	1-256	EBCDIC	Target server 4 statement value
SMF119ML_CD_MailDir (46)	1-512	EBCDIC	Extended retry mail directory

Table 234 describes the identity of the command that started CSSMTP or modified the configuration. This data section appears as section 5.

Table 234. CSSMTP configuration command

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_CM	88	STRUCTURE	Command data

Table 234. CSSMTP configuration command (continued)

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_CM_CnsName	8	EBCDIC	Name of the console that issued the command
8(X'8')	SMF119ML_CM_UToken	80	Binary	ICHRUTKN User token

CSSMTP connection record (CONNECT subtype 49)

This record is written at the end of each client connection with a target server. It contains the statistics about the amount of traffic and mail messages that are carried on the connection.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the CSSMTP connection record, the TCP/IP stack identification section indicates CSSMTP as the subcomponent and X'08' (event record) as the record reason. The field SMF119TI_Stack contains the name of the TCP/IP stack for the connection. If the value in the SMF119ML_CS_TrmCd field indicates that the connection is not established, then fields such as SMF119TI_Stack, SMF119AP_TIConnID and other fields that are associated with the connection are not set. This is a connection oriented SMF record.

Table 235 shows the CSSMTP connection record self-defining section.

Table 235. CSSMTP connection record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 48(X'30')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (4)
26(X'1A')		2	Binary	Reserved
28(X'1C')	SMF119IDOff	4	Binary	Offset to TCP/IP identification section...see table
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 CSSMTP common information section...see table
40(X'28')	SMF119S1Len	2	Binary	Length of CSSMTP common information section
42(X'2A')	SMF119S1Num	2	Binary	Number of CSSMTP common information sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to CSSMTP connection identification section see table
48(X'30')	SMF119S2Len	2	Binary	Length of CSSMTP connection identification section
50(X'32')	SMF119S2Num	2	Binary	Number of CSSMTP connection identification sections
52(X'34')	SMF119S3Off	4	Binary	Offset to CSSMTP connection statistics section see table

Table 235. CSSMTP connection record self-defining section (continued)

Offset	Name	Length	Format	Description
56(X'38')	SMF119S3Len	2	Binary	Length of CSSMTP connection statistics section
58(X'36')	SMF119S3Num	2	Binary	Number of CSSMTP connection statistics sections

Table 229 on page 857 in "CSSMTP configuration record (CONFIG subtype 48)" on page 855 shows the CSSMTP common information section. This section identifies the CSSMTP JOB that created this SMF record. It is found in subtypes 48, 49, 50, 51 and 52.

Table 236 shows the CSSMTP connection identification data.

Table 236. CSSMTP connection identification data

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_CN	72	STRUCTURE	Connection Identification
0(X'0')	SMF119ML_CN_LIP	16	Binary	Local IP address
0(X'0')	SMF119ML_CN_LIPPfx	12	Binary	00000000000000000000FFFF
12(X'C')	SMF119ML_CN_LIP4	4	Binary	IPv4 Address
16(X'10')	SMF119ML_CN_RIP	16	Binary	Rmote IP address
16(X'10')	SMF119ML_CN_RIPPfx	12	Binary	00000000000000000000FFFF
28(X'1C')	SMF119ML_CN_RIP4	4	Binary	IPv4 Address
32(X'20')	SMF119ML_CN_LPort	2	Binary	Local port address
34(X'22')	SMF119ML_CN_RPort	2	Binary	Remote port address
36(X'24')	SMF119ML_CN_ConnId	4	Binary	TCP/IP connection ID
40(X'28')	SMF119ML_CN_STIME	4	Binary	Time the connection started (hundredths of seconds since midnight)
44(X'2C')	SMF119ML_CN_SDATE	4	Packed	Date the connection started 0CYYDDDF
48(X'30')	SMF119ML_CN_ETIME	4	Binary	Time the connection ended (hundredths of seconds since midnight)
52(X'34')	SMF119ML_CN_EDATE	4	Packed	Date the connection ended 0CYYDDDF
56(X'38')	SMF119ML_CN_DUR	4	Binary	Duration of connection (hundredths of seconds since midnight)
60(X'3C')	SMF119ML_CN_MsgSize	4	Binary	Maximum message size
64(X'40')	SMF119ML_CN_TLSSSP	2	Binary	AT-TLS SSL protocol: • X'0200': SSL Version 2 • X'0300': SSL Version 3 • X'0301': AT-TLS Version 1 • X'0302': AT-TLS Version 1.1
66(X'42')	SMF119ML_CN_TTLSSNC	2	EBCDIC	AT-TLS negotiated cipher

Table 236. CSSMTP connection identification data (continued)

Offset	Name(Dim)	Length	Type	Description
68(X'44')	SMF119ML_CN_TTLSFP	1	Binary	AT-TLS FIPS 140 status: • X'00': FIPS 140 off • X'01': FIPS 140 on
69(X'45')	SMF119ML_CN_Flags	3	Binary	Flags
69(X'45')	SMF119ML_CN_FLAG1	1	Binary	Flags X'80': If this flag is on, the protocol is ESMTP. Otherwise, the protocol is SMTP.
70(X'46')	SMF119ML_CN_Rsvd2x	1	Binary	Reserved
71(X'47')	SMF119ML_CN_Rsvd3x	1	Binary	Reserved

Table 237 describes the CSSMTP connection statistics data.

Table 237. CSSMTP connection statistics data

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_CS	96	STRUCTURE	Connection Statistics
0(X'0')	SMF119ML_CS_SendByt	8	Binary	Number of bytes outbound
8(X'8')	SMF119ML_CS_RcvdByt	8	Binary	Number of bytes inbound
16(X'10')	SMF119ML_CS_MsgSent	8	Binary	Number of sent mail messages
20(X'14')	SMF119ML_CS_GRcpts	4	Binary	Number of recipients accepted
24(X'18')	SMF119ML_CS_FRcpts	4	Binary	Number of recipients not accepted

Table 237. CSSMTP connection statistics data (continued)

Offset	Name(Dim)	Length	Type	Description
28(X'1C')	SMF119ML_CS_TrmCd	4	Binary	Connection ending status: • SMF119ML_CS_TRM_OK 00 - Connect normal close • SMF119ML_CS_TRM_SOCKET 01 - Socket function error • SMF119ML_CS_TRM_RESET 02 - Server reset connection • SMF119ML_CS_TRM_OVERRUN 03 - Buffer overrun error • SMF119ML_CS_TRM_4XX 04 - 4xx Reply • SMF119ML_CS_TRM_5XX 05 - 5xx Reply • SMF119ML_CS_TRM_XXX 06 - unknown reply • SMF119ML_CS_TRM_CONVERT 07 - ICONV error • SMF119ML_CS_TRM_CONNERR 08 - Connect failed • SMF119ML_CS_TRM_SECURE 09 - StartTLS command failed • SMF119ML_CS_TRM_MAXMSG 10 - Maximum number of messages • SMF119ML_CS_TRM_CONNECT 11 - Connection wait timeout • SMF119ML_CS_TRM_INITMSG12 - Initial message time out • SMF119ML_CS_TRM_13 13 - Reserved • SMF119ML_CS_TRM_MAILCMD 14 - Mail command time out • SMF119ML_CS_TRM_RCPTCMD 15 - RCPT command time out • SMF119ML_CS_TRM_DATACMD 16 - DATA command time out • SMF119ML_CS_TRM_DATABUF 17 - Data buffer time out • SMF119ML_CS_TRM_DATATRM 18 - End of message time out • SMF119ML_CS_TRM_ANYCMD 19 - Any command timeout
32(X'20')	SMF119ML_CS_ErrTxt	64	EBCDIC	Last error text on SMTP command that caused the connection to be closed

CSSMTP mail record (MAIL subtype 50)

This record is written when each mail message completes processing. It contains the statistics and information about each mail message. It also indicates the success or failure to send the mail message.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the CSSMTP mail record, the TCP/IP stack identification section indicates CSSMTP as the subcomponent and X'08' (event record) as the record reason. The field SMF119TI_Stack name is blank unless the -p parameter is used to start the CSSMTP application. If this record is written to NMI, the field SMF119TI_Stack in the NMI record contains the stack name that the record was written to. This is a non-connection oriented SMF record.

Table 238 shows the CSSMTP mail record self-defining section.

Table 238. CSSMTP mail record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 48(X'30')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (5)
26(X'1A')		2	Binary	Reserved
28(X'1C')	SMF119IDOff	4	Binary	Offset to TCP/IP identification section...see table
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 CSSMTP common information section...see table
40(X'28')	SMF119S1Len	2	Binary	Length of CSSMTP common information section
42(X'2A')	SMF119S1Num	2	Binary	Number of CSSMTP common information sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to spool identification section see table
48(X'30')	SMF119S2Len	2	Binary	Length of spool identification section
50(X'32')	SMF119S2Num	2	Binary	Number of spool identification sections
52(X'34')	SMF119S3Off	4	Binary	Offset to mail data section see table
56(X'38')	SMF119S3Len	2	Binary	Length of mail data section
58(X'36')	SMF119S3Num	2	Binary	Number of mail data sections
60(X'3C')	SMF119S4Off	4	Binary	Offset to mail header section see table
64(X'40')	SMF119S4Len	2	Binary	Length of mail header section
66(X'42')	SMF119S4Num	2	Binary	Number of mail header sections

Table 229 on page 857 in "CSSMTP configuration record (CONFIG subtype 48)" on page 855 shows the CSSMTP common information section. This section identifies the CSSMTP JOB that created this SMF record. It is found in subtypes 48, 49, 50, 51 and 52.

Table 239 on page 866 shows the CSSMTP common spool information section. This section identifies the spool job that created the sysout file. It is found in subtypes 50 and 51.

Table 239. CSSMTP spool identification

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_SI	72	STRUCTURE	Spool Identification
0(X'0')	SMF119ML_SI_JMR	24	EBCDIC	Job Management Record. See Standard SMF Record Header in *z/OS MVS System Management Facilities (SMF)* for detailed information about Job Management Record format.
0(X'0')	SMF119ML_SI_Job	8	EBCDIC	Jobname
8(X'8')	SMF119ML_SI_Entry	4	Binary	JES reader entry time - time since midnight, in hundredths of a second, that the reader recognized the JOB card (for this job).
12(X'C')	SMF119ML_SI_EDate	4	Packed	JES reader entry date 0CYYDDDF - date when the reader recognized the JOB card (for this job), in the form 0cyydddF.
16(X'10')	SMF119ML_SI_USEID	8	EBCDIC	User-defined identification field (taken from common exit parameter area, not from USER=parameter on job statement).
24(X'18')	SMF119ML_SI_JobId	8	EBCDIC	Job Id of selected job
32(X'20')	SMF119ML_SI_SYS	8	EBCDIC	System name of the MVS image where the job output was created
40(X'28')	SMF119ML_SI_XEQ	8	EBCDIC	NJE node where job executed
48(X'30')	SMF119ML_SI_CRER	8	EBCDIC	Owning user id of data set
56(X'38')	SMF119ML_SI_TKID	4	Binary	JES task ID
60(X'3C')	SMF119ML_SI_Jnum	4	Binary	JES job number in binary
64(X'40')	SMF119ML_SI_Dsky	4	Binary	JES dataset key
68(X'44')	SMF119ML_SI_Dsnm	4	Binary	JES dataset number

Table 240 describes the CSSMTP mail data section.

Table 240. CSSMTP mail data section

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_MI	60	STRUCTURE	Mail Identification
0(X'0')	SMF119ML_MI_STime	4	Binary	Time mail was read from JES - Hundredths of seconds
4(X'4')	SMF119ML_MI_SDate	4	Packed	Date mail was read from JES - in 0CYYDDDF format
8(X'8')	SMF119ML_MI_ETime	4	Binary	Time mail was completed - Hundredths of seconds
12(X'C')	SMF119ML_MI_EDate	4	Packed	Date mail was completed, in 0CYYDDDF format
16(X'10')	SMF119ML_MI_Dur	4	Binary	Time mail was in progress, in hundredths of seconds
20(X'14')	SMF119ML_MI_ID	4	Binary	Mail message number in spool file

Table 240. CSSMTP mail data section (continued)

Offset	Name(Dim)	Length	Type	Description
24(X'18')	SMF119ML_MI_Type	4	Binary	Type of mail message - type values: • SMF119ML_MI_TYPE_RegNote = 1 mail message is regular type created by customer • SMF119ML_MI_TYPE_UndelNote = 2 mail message is error note created by customer • SMF119ML_MI_TYPE_Report = 3 mail message is a CSSMTP error report • SMF119ML_MI_TYPE_UMNOTIF = 4 mail message is a undeliverable mail notification
28(X'1C')	SMF119ML_MI_Rsvd1	4	Binary	Reserved
32(X'20')	SMF119ML_MI_BYCT	8	Binary	Body byte count
40(X'28')	SMF119ML_MI_RLoc	4	Binary	Record location of MAIL command in spool file
44(X'2C')	SMF119ML_MI_Rcpts	4	Binary	Number of total recipients
48(X'30')	SMF119ML_MI_FRcpts	4	Binary	Number of failed recipients
52(X'34')	SMF119ML_MI_Retry	4	Binary	Number of retry attempts
56(X'38')	SMF119ML_MI_Flags	4	BIT(32)	Flags
	SMF119ML_MI_ESMTP		1...	EHLO(RFC 2821) command
	SMF119ML_MI_TLS		.1..	STARTTLS command
	SMF119ML_MI_Finis		..1.	Mail was completed without errors
	SMF119ML_MI_Error		...1	Mail was completed with errors
	SMF119ML_MI_ERetry		.... 1...	Mail was saved for extended retry
	SMF119ML_MI_Rsv04		.... 1...	Reserved
	SMF119ML_MI_Rsv05		.... .1..	Reserved
	SMF119ML_MI_Rsv06		.... ..1.	Reserved
	SMF119ML_MI_MHFul		1	The SMF record is full. Data in the mail header section was truncated.
57(X'39')	SMF119ML_MI_From		1...	Mail contains a From header specified in the spool file
	SMF119ML_MI_To		.1..	Mail contains a To header specified in the spool file
	SMF119ML_MI_Date		..1.	Mail contains a Date header specified in the spool file
	SMF119ML_MI_MsgID		...1	Mail contains a msg-ID specified in the spool file
	SMF119ML_MI_Subj		.... 1...	Mail contains a subject specified in the spool file
	SMF119ML_MI_Rsv15		.... .1..	Reserved
	SMF119ML_MI_Rsv16		.... ..1.	Reserved
	SMF119ML_MI_Rsv17		1	Reserved
58(X'3A')	SMF119ML_MI_Rsv2x	1	BIT(8)	Reserved

Table 240. CSSMTP mail data section (continued)

Offset	Name(Dim)	Length	Type	Description
59(X'3B')	SMF119ML_MI_Rsv3x	1	BIT(8)	Reserved

Table 241 describes the values of the various mail headers in the mail. They are encoded as variable length strings.

Table 241. CSSMTP mail header sections

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_MH	*	STRUCTURE	Mail header
0(X'0')	SMF119ML_MH_Len	2	Binary	Mail header length
2(X'2')	SMF119ML_MH_Key	2	Binary	Mail header type value
4(X'4')	SMF119ML_MH_Data	*	EBCDIC	Mail header data string

Table 242 describes the values of the various mail commands and header keys.

Table 242. CSSMTP mail commands and header keys

Data type (SMF119ML_MH_Key)	Data length (SMF119ML_MH_Len)	Format	Description (SMF119ML_MH_Data)
SMF119ML_MH_FROM (1)	1-256	EBCDIC	Mail box address of MAIL FROM: command
SMF119ML_MH_RCPT (2)	1-256	EBCDIC	Mail box address of RCPT TO: command
SMF119ML_MH_RCPTRPY (3)	1-512	EBCDIC	Error reply text to previous RCPT TO: command (See Note)
SMF119ML_MH_SUBJ (4)	1-233	EBCDIC	Subject: *subject text*
SMF119ML_MH_DATE (5)	1-47	EBCDIC	Date: *date value*
SMF119ML_MH_MSGID (7)	1-143	EBCDIC	Message-id: *value*
SMF119ML_MH_CMDTXT (8)	1-512	EBCDIC	Text of SMTP command in error
SMF119ML_MH_RPYTXT (9)	1-512	EBCDIC	Server reply to the SMTP command in error
SMF119ML_MH_ERRTXT (10)	1-512	EBCDIC	Text of error message not associated with SMTP command processing
Note: If this field contains a single F, the recipient did not receive the mail as the result of the reason in the general error field (SMF119ML_MH_ERRTXT) or command error field (SMF119ML_MH_CMDTXT/ SMF119ML_MH_RPYTXT).			

CSSMTP spool file record (SPOOL subtype 51)

This record is written when all the mail messages have been processed. It contains information about the spool file and statistics about the mail messages that are processed.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the CSSMTP spool file record, the TCP/IP stack identification section indicates the stack name with blanks, CSSMTP as the subcomponent, and X'08' (event record) as the record reason. The name of field SMF119TI_Stack will be blank unless the -p parameter is used to start the CSSMTP application. If the

record is written to NMI, the field SMF119TI_Stack in the NMI record contains the stack name that the record was written to. This is a non-connection oriented SMF record.

Table 243 shows the CSSMTP spool file record self-defining section.

Table 243. CSSMTP spool file record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 51(X'33')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (6)
26(X'1A')		2	Binary	Reserved
28(X'1C')	SMF119IDOff	4	Binary	Offset to TCP/IP identification section...see table
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 CSSMTP common information section...see table
40(X'28')	SMF119S1Len	2	Binary	Length of CSSMTP common information section
42(X'2A')	SMF119S1Num	2	Binary	Number of CSSMTP common information sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to spool identification section see table
48(X'30')	SMF119S2Len	2	Binary	Length of spool identification section
50(X'32')	SMF119S2Num	2	Binary	Number of spool identification sections
52(X'34')	SMF119S3Off	4	Binary	Offset to spool job section see table
56(X'38')	SMF119S3Len	2	Binary	Length of spool job section
58(X'36')	SMF119S3Num	2	Binary	Number of spool job sections
60(X'3C')	SMF119S4Off	4	Binary	Offset to spool statistics section see table
64(X'40')	SMF119S4Len	2	Binary	Length of spool statistics section
66(X'42')	SMF119S4Num	2	Binary	Number of spool statistics section
68(X'44')	SMF119S5Off	4	Binary	Offset to spool accounting section see table
72(X'48')	SMF119S5Len	2	Binary	Length of spool accounting section
74(X'4A')	SMF119S5Num	2	Binary	Number of spool accounting sections

Table 229 on page 857 in “CSSMTP configuration record (CONFIG subtype 48)” on page 855 shows the CSSMTP common information section. This section identifies the CSSMTP JOB that created this SMF record. It is found in subtypes 48, 49, 50, 51 and 52.

See Table 239 on page 866 in "CSSMTP mail record (MAIL subtype 50)" on page 864 for the contents of the common Spool Identification section. This section identifies the spool job that created the sysout file. It is found in subtypes 50 and 51.

Table 244 describes the JES SSI information for the spool file job.

Table 244. CSSMTP spool job

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_SJ	140	STRUCTURE	Spool Job data
0(X'0')	SMF119ML_SJ_BYCT	8	Binary	Byte count
8(X'8')	SMF119ML_SJ_LNCT	4	Binary	Line count
12(X'C')	SMF119ML_SJ_PRCD	8	EBCDIC	Data set procname
20(X'14')	SMF119ML_SJ_STPD	8	EBCDIC	Data set stepname
28(X'1C')	SMF119ML_SJ_DDND	8	EBCDIC	Data set DD name
36(X'24')	SMF119ML_SJ_PNAM	20	EBCDIC	Programmer name from job
56(X'38')	SMF119ML_SJ_NOTN	8	EBCDIC	Job notify node
64(X'40')	SMF119ML_SJ_NOTU	8	EBCDIC	Job notify user ID
72(X'48')	SMF119ML_SJ_CLAR	1	EBCDIC	Sysout class of data set
73(X'49')	SMF119ML_SJ_LSAB	3	Binary	Last abend code for the job that created the spool file (JES 2 only)
76(X'4C')	SMF119ML_SJ_DSN	44	EBCDIC	Data set name of the spool file
120(X'78')	SMF119ML_SJ_NACT	8	EBCDIC	Network accounting number
128(X'80)'	SMF119ML_SJ_UserExit	4	Binary	User exit version 0-SMF119ML_SJ_USEREXIT_NONE 2-SMF119ML_SJ_USEREXIT_VERSION2 3-SMF119ML_SJ_USEREXIT_VERSION3
132(X'84')	SMF119ML_SJ_QTime	4	Binary	Time spool file queued to JES - Hundredths of second
136(X'88')	SMF119ML_SJ_QDate	4	Packed	Date spool file queued to JES - 0CYYDDDF

Table 245 describes the spool job statistics.

Table 245. CSSMTP spool job statistics

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_SS	100	STRUCTURE	Spool Job statistics
0(X'0')	SMF119ML_SS_STime	4	Binary	Time when CSSMTP started to read the spool data set - Hundredths of seconds since midnight
4(X'4')	SMF119ML_SS_SDate	4	Packed	Date when CSSMTP started to read the spool data set - 0CYYDDDF
8(X'8')	SMF119ML_SS_RTime	4	Binary	Time when CSSMTP completed reading the spool data set. Hundredths of seconds since midnight

Table 245. CSSMTP spool job statistics (continued)

Offset	Name(Dim)	Length	Type	Description
12(X'C')	SMF119ML_SS_RDate	4	Packed	Date when CSSMTP completed reading the spool data set - 0CYYDDDF
16(X'10')	SMF119ML_SS_RcdCnt	4	Binary	Number of spool file records CSSMTP read
20(X'14')	SMF119ML_SS_ETime	4	Binary	Time all mail is processed for this spool data set. Hundredths of seconds since midnight
24(X'18')	SMF119ML_SS_EDate	4	Packed	Date all mail is processed for this spool data set 0CYYDDDF
28(X'1C')	SMF119ML_SS_MAIL	4	Binary	Total number of mail messages found in the spool data set.
32(X'20')	SMF119ML_SS_Good	4	Binary	Number of mail messages successfully sent
36(X'24')	SMF119ML_SS_Udv	4	Binary	Number of undeliverable mails resulting from spool data set processing
40(X'28')	SMF119ML_SS_Dead	4	Binary	Number of Deadletter mail resulting from spool data set processing
44(X'2C')	SMF119ML_SS_Rcpt	4	Binary	Total number of recipients (RCPTs) in the spool data set
48(X'30')	SMF119ML_SS_CRcpt	4	Binary	Total number of recipients (RCPTs) sent successfully in the spool data set
52(X'34')	SMF119ML_SS_URcpt	4	Binary	Total number of recipients (RCPTs) that are undeliverable
56(X'38')	SMF119ML_SS_Skip	4	Binary	Number of mail skipped due to user exit or restart (checkpointing)
60(X'3C')	SMF119ML_SS_Err	4	Binary	Number of mails with syntax errors in jes spool data set
64(X'40')	SMF119ML_SS_Bsize	8	Binary	Total size in bytes of all mail headers and bodies processed by CSSMTP for the JES spool data set

Table 245. CSSMTP spool job statistics (continued)

Offset	Name(Dim)	Length	Type	Description
72(X'48')	SMF119ML_SS_RtnCd	4	Binary	Processing return codes: SMF119ML_SS_Alloc 07 JES sysout allocation failed SMF119ML_SS_OPEN 08 Open failed for sysout file SMF119ML_SS_REQERR 10 IEFSSREQ failed SMF119ML_SS_APIERR 11 IEFSSREQ SSS2 API failed SMF119ML_SS_ICONV 13 Conversion table open error SMF119ML_SS_EMPTY 14 Empty data set SMF119ML_SS_JESSIZE 15 size exceeds JesJobSize SMF119ML_SS_SAF 16 Access is not authorized SMF119ML_SS_TRANSLATE 18 Translation error SMF119ML_SS_NOEBCDIC 19 Unknown translation table SMF119ML_SS_USEREXIT 21 The return code from the CSSMTP user exit indicates that the processing of the spool file should stop SMF119ML_SS_NOMAIL 22 The spool file does not contain any mail transactions SMF119ML_SS_JESCLOSE 23 The JES spool file was not properly closed by JES and the file data might be incomplete SMF119ML_SS_IOERROR 24 An I/O error occurred during reading the spool file SMF119ML_SS_MAXERROR 25 Maximum number of syntax errors in spool file was reached
76(X'4C')	SMF119ML_SS_Flags	4	BIT(32)	Flags
	SMF119ML_SS_TLS		1...	JES spool data set contained STARTTLS command
	SMF119ML_SS_Finis		.1..	CSSMTP completed processing the spool data set

Table 245. CSSMTP spool job statistics (continued)

Offset	Name(Dim)	Length	Type	Description
	SMF119ML_SS_ErrRpt		..1.	Spool data set was generated by CSSMTP for error report
	SMF119ML_SS_Hold		...1	Final disposition of data set 1 - HOLD 0 - DELETE
	SMF119ML_SS_Error		.... 1...	One or more syntax errors were found when the spool file was processed
	SMF119ML_SS_Rsv05		.... .1..	Reserved
	SMF119ML_SS_Rsv06		.... ..1.	Reserved
	SMF119ML_SS_Rsv07		1	Reserved
77(X'4D')	SMF119ML_SS_Xmit		1...	Spool data set is in NetData format
	SMF119ML_SS_RStrt		.1..	Spool data set was restarted due to checkpointing
	SMF119ML_SS_Rsv12		..1.	Reserved
	SMF119ML_SS_Rsv13		...1	Reserved
	SMF119ML_SS_Rsv14		.... 1...	Reserved
	SMF119ML_SS_Rsv15		.... .1..	Reserved
	SMF119ML_SS_Rsv16		.... ..1.	Reserved
	SMF119ML_SS_Rsv17		1	Reserved
78(X'4E')	SMF119ML_SS_RStc		1...	Data set created by started task
	SMF119ML_SS_RTsc		.1..	Data set created by time sharing user
	SMF119ML_SS_RJob		..1.	Data set created by batch job
	SMF119ML_SS_Rsv23		...1	Reserved
	SMF119ML_SS_Rsv24		.... 1...	Reserved
	SMF119ML_SS_Rsv25		.... .1..	Reserved
	SMF119ML_SS_Rsv26		.... ..1.	Reserved
	SMF119ML_SS_Rsv27		1	Reserved
79(X'4F')	SMF119ML_SS_Flag4	1	BIT(8)	Reserved
80(X'50')	SMF119ML_SS_EMail	4	Binary	Number of mail messages saved for extended retry
84(X'54')	SMF119ML_SS_ERcpt	4	Binary	Number of recipients to be retried in mail messages saved for extended retry

Table 246 on page 874 describes the CSSMTP SMF job accounting information section. It contains information about the source spool file. If there is no accounting information (the number of entries is zero), then this section is not present.

Accounting information is in SMF format as it is in type 5 and type 30 SMF records. For more information about the accounting field, see *z/OS MVS System Management Facilities (SMF)*.

```
AL1(number-of-pairs-that-follow) followed by 1 or more fields
AL1(length),CLlength'string' A length of 0 indicates
an omitted field
```

An example of accounting information given the following of (X3600,42,,ANDY):.

```
DC  AL1(4)              Nr of fields
DC  AL1(5),CL5'X3600'   field 1
DC  AL1(2),CL2'42'      field 2
DC  AL1(0)              field 3 (null)
DC  AL1(4),CL4'ANDY'    field 4
```

Table 246. CSSMTP spool job accounting

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_SA	144	STRUCTURE	Spool Job Accounting
0(X'0')	SMF119ML_SA_Cnt	1	Binary	Number of accounting sections
1(X'1')	SMF119ML_SA_Txt	143	EBCDIC	Encoded accounting information

CSSMTP statistical record (STATS subtype 52)

This record is written at the end of each MVS SMF interval and at the termination of CSSMTP application. See *z/OS MVS System Management Facilities (SMF)* for information about setting the SMF interval. It contains global statistics about spool and mail processing, information about the health of the CSSMTP program, and the activity of each of the target server connections during the interval.

Note: Any change in the SMF recording interval is not acted upon until the previous interval expires.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the CSSMTP statistical record, the TCP/IP stack identification section indicates CSSMTP as the subcomponent and X'08' (event record) as the record reason. The field SMF119TI_Stack name is blank unless the -p parameter is used to start the CSSMTP application. If this record is written to NMI, the field SMF119TI_Stack in the NMI record contains the stack name that the record was written to. This is a non-connection oriented SMF record.

Table 247 shows the CSSMTP statistical record self-defining section.

Table 247. CSSMTP statistical record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 48(X'30')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (6)
26(X'1A')		2	Binary	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 CSSMTP common information section
40(X'28')	SMF119S1Len	2	Binary	Length of CSSMTP common information section

Table 247. CSSMTP statistical record self-defining section (continued)

Offset	Name	Length	Format	Description
42(X'2A')	SMF119S1Num	2	Binary	Number of CSSMTP common information sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to CSSMTP statistical data section see table
48(X'30')	SMF119S2Len	2	Binary	Length of CSSMTP statistical data section
50(X'32')	SMF119S2Num	2	Binary	Number of CSSMTP statistical data sections
52(X'34')	SMF119S3Off	4	Binary	Offset to CSSMTP JES statistical section see table
56(X'38')	SMF119S3Len	2	Binary	Length of CSSMTP JES statistical section
58(X'36')	SMF119S3Num	2	Binary	Number of CSSMTP JES statistical sections
60(X'3C')	SMF119S4Off	4	Binary	Offset to CSSMTP health checker statistical section see table
64(X'40')	SMF119S4Len	2	Binary	Length of CSSMTP health checker statistical section
66(X'42')	SMF119S4Num	2	Binary	Number of CSSMPT health checker statistical sections
68(X'44')	SMF119S5Off	4	Binary	Offset to CSSMTP target server statistical section see table
72(X'48')	SMF119S5Len	2	Binary	Length of CSSMTP target server statistical section
74(X'4A')	SMF119S5Num	2	Binary	Number of CSSMTP target server statistical sections

Table 229 on page 857 in "CSSMTP configuration record (CONFIG subtype 48)" on page 855 shows the CSSMTP common information section. This section identifies the CSSMTP JOB that created this SMF record. It is found in subtypes 48, 49, 50, 51 and 52.

Table 248 shows the CSSMTP statistical data section. These values reflect the mail processing that is accumulated over the interval. SMF119ML_ST_LRTQCount is the number of messages on the retry queue at the end of the interval.

Table 248. CSSMTP statistical data

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_ST	64	STRUCTURE	CSSMTP Statistics
0(X'0')	SMF119ML_ST_STime	4	Binary	Time interval started, in hundredths of seconds
4(X'4')	SMF119ML_ST_SDate	4	Packed	Date interval started, in 0CYYDDDF format
8(X'8')	SMF119ML_ST_ETime	4	Binary	Time interval ended, in hundredths of seconds
12(X'C')	SMF119ML_ST_EDate	4	Packed	Date interval ended, in 0CYYDDDF format

Table 248. CSSMTP statistical data (continued)

Offset	Name(Dim)	Length	Type	Description
16(X'10')	SMF119ML_ST_Dur	4	Binary	Duration of the interval, in hundredths of seconds
20(X'14')	SMF119ML_ST_Flags	4	BIT (32)	Application state flags at the end of the interval
	SMF119ML_ST_NoStck		1...	No stack available
	SMF119ML_ST_STargt		.1..	Suspend - no targets available
	SMF119ML_ST_SImmed		..1.	Suspend immediate
	SMF119ML_ST_SDelay		...1	Suspend delay
	SMF119ML_ST_Rsv04		.... 1...	Reserved
	SMF119ML_ST_StgUse		.... .1..	Storage usage high at 95%
	SMF119ML_ST_Rsv06		.... ..1.	Reserved
	SMF119ML_ST_Rsv07		1	Reserved
21(X'15')	SMF119ML_ST_Rsv1x	1	Binary	Reserved
22(X'16')	SMF119ML_ST_Rsv2x	1	Binary	Reserved
23(X'17')	SMF119ML_ST_Rsv3x	1	Binary	Reserved
24(X'18')	SMF119ML_ST_MailCount	8	Binary	Number of new mail messages processed
32(X'20')	SMF119ML_ST_LRTCount	8	Binary	Number of mail messages entered long retry
40(X'28')	SMF119ML_ST_LRTDeadLtrCount	8	Binary	Number of mail messages that have become dead letters
48(X'30')	SMF119ML_ST_LRTQCount	8	Binary	Current number of mail messages on long retry queue
56(X'38')	SMF119ML_ST_UDVCount	8	Binary	Number of mail messages that are undeliverable
64(X'40')	SMF119ML_ST_ErtCount	4	Binary	Current number of mail messages for extended retry
68(X'44')	SMF119ML_ST_ErtQCount	4	Binary	Cumulative total number of mail messages for extended retry
72(X'48')	SMF119ML_ST_ErtUndvl	4	Binary	Number of mail messages made undeliverable by extended retry
76(X'4C')	SMF119ML_ST_ErtError	4	Binary	Number of mail messages dropped by extended retry due to file system errors

Table 249 describes the CSSMTP JES statistical data. This data reflects the values from the JES spool files that completed during the interval.

Table 249. CSSMTP JES statistical data

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_JS	64	STRUCTURE	CSSMTP JES statistics
0(X'0')	SMF119ML_JS_JesFiles	4	Binary	Number of JES spool files completed
4(X'4')	SMF119ML_JS_JesRcdCnt	4	Binary	Number of JES records read from the JES spool files

Table 249. CSSMTP JES statistical data (continued)

Offset	Name(Dim)	Length	Type	Description
8(X'8')	SMF119ML_JS_JesTime	4	Binary	Sum of completed JES spool files processing times (hundreds of seconds)
12(X'C')	SMF119ML_JS_JesScan	4	Binary	Sum of JES scanning time
16(X'10')	SMF119ML_JS_MAIL	4	Binary	Number of mail message found in the spool data sets
20(X'14')	SMF119ML_JS_Good	4	Binary	Number of mail messages in the spool dataset that were successfully sent
24(X'18')	SMF119ML_JS_Udv	4	Binary	Number of mail messages in the spool dataset that were not sent
28(X'1C')	SMF119ML_JS_Dead	4	Binary	Number of dead letter mail resulting from spool dataset processing
32(X'20')	SMF119ML_JS_Rcpt	4	Binary	Number of recipients in the spool datasets
36(X'24')	SMF119ML_JS_CRcpt	4	Binary	Total recipients sent successfully
40(X'28')	SMF119ML_JS_URcpt	4	Binary	Total recipients that are undeliverable
44(X'2C')	SMF119ML_JS_Skip	4	Binary	Number of mail skipped due to user exit or restart
48(X'30')	SMF119ML_JS_Bsize	8	Binary	Total size in bytes of all mail headers and body sections processed by CSSMTP for the JES spool data set
56(X'38')	SMF119ML_JS_SError	4	Binary	Number of mail with syntax error found in the JES spool data sets
60(X'3C')	SMF119ML_JS_RtnCd	4	Binary	Number of JES spool jobs fail with nonzero processing return code
64(X'40')	SMF119ML_JS_EMail	4	Binary	Number of mail messages saved for extended retry
68(X'44')	SMF119ML_JS_ERcpt	4	Binary	Number of recipients to be retried in mail messages saved for extended retry

Table 250 describes the CSSMTP health checker statistics These are the values at the end of the interval.

Table 250. CSSMTP Health checker statistics

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_HC	60	STRUCTURE	Health Check statistics
0(X'0')	SMF119ML_HC_Time	4	Binary	Time of last health check - hundredths of seconds since midnight
4(X'4')	SMF119ML_HC_Date	4	Packed	Date of last health check, in 0CYYDDDF format
8(X'8')	SMF119ML_HC_StgTotal	4	Binary	Total storage region size for CSSMTP
12(X'C')	SMF119ML_HC_StgIFree	4	Binary	Storage available after CSSMTP initialization is completed
16(X'10')	SMF119ML_HC_StgFree	4	Binary	Storage currently not in use
20(X'14')	SMF119ML_HC_StgAlloc	4	Binary	Storage currently in use

Table 250. CSSMTP Health checker statistics (continued)

Offset	Name(Dim)	Length	Type	Description
24(X'18')	SMF119ML_HC_StgPUsed	4	Binary	Storage percent in use
28(X'1C')	SMF119ML_HC_StgFail	4	Binary	Number of storage failures
32(X'20')	SMF119ML_HC_Rsvd1	4	Binary	Reserved
36(X'24')	SMF119ML_HC_DLRPFree	4	Binary	Percentage of file system space that is free system-wide and that can be used to store dead letters
40(X'28')	SMF119ML_HC_DLRPUsed	4	Binary	Percentage of file system space that is used system-wide
44(X'2C')	SMF119ML_HC_JESDUsed	4	Binary	Number of JES DEST tasks busy
48(X'30')	SMF119ML_HC_JESDPerC	8	Binary	Percent of JES DEST Tasks busy
52(X'34')	SMF119ML_HC_JESWUsed	4	Binary	Number of JES writer tasks busy
56(X'38')	SMF119ML_HC_JESWPerC	4	Binary	Percent of JES writer tasks busy
60(X'3C')	SMF119ML_HC_MDirPFree	4	Binary	Percentage of file system space that is free system-wide and that can be used to store extended retry mail messages
64(X'40')	SMF119ML_HC_MDirPUsed	4	Binary	Percentage of file system space that is used system-wide

Table 251 describes the target server statistical data. There is one entry for each defined target server IP address. This is the list that is defined at the interval.

Table 251. Target server statistical data

Offset	Name(Dim)	Length	Type	Description
0(X'0')	SMF119ML_IP	68	STRUCTURE	Target Server statistics
0(X'0')	SMF119ML_IP_IP	16	Binary	Target server IP address
0(X'0')	SMF119ML_IP_IPPfx	12	Binary	00000000000000000000FFFF
12(X'C')	SMF119ML_IP_IP4	4	Binary	IPv4 Address
16(X'10')	SMF119ML_IP_Port	2	Binary	Target server port number
18(X'12')	SMF119ML_IP_Rsvd1	2	Binary	Reserved
20(X'14')	SMF119ML_IP_ConnState	4	Binary	Connection state: • SMF119ML_IP_New - Target server is new so its capabilities are unknown but it is in the configured address list. • SMF119ML_IP_Active - The target server is available. • SMF119ML_IP_Monitoring - The target server is being monitored for a successful connection open and reply from the server. • SMF119ML_IP_NonActive - The target server is not available.
24(X'18')	SMF119ML_IP_MsgSize	4	Binary	Maximum message size
28(X'1C')	SMF119ML_IP_RecvdCount	4	Binary	Total mail messages received
32(X'20')	SMF119ML_IP_SentCount	4	Binary	Total mail messages sent

Table 251. Target server statistical data (continued)

Offset	Name(Dim)	Length	Type	Description
36(X'24')	SMF119ML_IP_ConCount	4	Binary	Connection count
40(X'28')	SMF119ML_IP_ConFailCount	4	Binary	Connection failure count
44(X'2C')	SMF119ML_IP_Rsvd2	4	Binary	Reserved
48(X'30')	SMF119ML_IP_RcvdBytes	8	Binary	Total number of received bytes
56(X'38')	SMF119ML_IP_SentBytes	8	Binary	Total number of sent bytes
64(X'40')	SMF119ML_IP_Flags	4	BIT(32)	Flags
	SMF119ML_IP_ESMTP		1...	ESMTP supported
	SMF119ML_IP_Rsv0x		.111 1111	Reserved
65(X'41')	SMF119ML_IP_Rsv1x	1	BIT(8)	Reserved
66(X'42')	SMF119ML_IP_Rsv2x	1	BIT(8)	Reserved
67(X'43')	SMF119ML_IP_Rsv3x	1	BIT(8)	Reserved

FTP server transfer completion record (subtype 70)

The FTP server transfer completion record is collected when the z/OS FTP server completes processing of one of the following FTP file transfer operations: file appending, file deletion, file storage (includes both store and store unique operations), file retrieval, or file renaming. A common format for the record is used for each FTP file transfer operation, so the record contains an indication of which operation was performed. The record also contains optional sections provided when gethostbyaddr( ) processing was performed during the file transfer operation, as well as when the file names involved in the transfer operation were MVS or z/OS UNIX filenames.

The Type 119 FTP server transfer completion record is collected at the same point in file transfer processing as the equivalent Type 118 FTP server SMF records. The Type 118 records used different record subtypes, as opposed to a field within the SMF record information, to represent the different file transfer operations being recorded.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the FTP server transfer completion record, the TCP/IP stack identification section indicates FTPS as the subcomponent and X'08' (event record) as the record reason.

Table 252 shows the FTP server transfer completion record self-defining section:

Table 252. FTP server transfer completion record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 70(X'46')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (6)
26(X'1A')		2	Binary	Reserved
28(X'1C')	SMF119IDOff	4	Binary	Offset to TCP/IP identification section

Table 252. FTP server transfer completion record self-defining section (continued)

Offset	Name	Length	Format	Description
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 completion section
40(X'28')	SMF119S1Len	2	Binary	Length of FTP server transfer completion section
42(X'2A')	SMF119S1Num	2	Binary	Number of FTP server transfer completion sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to FTP server host name section
48(X'30')	SMF119S2Len	2	Binary	Length of FTP server host name section
50(X'32')	SMF119S2Num	2	Binary	Number of FTP server host name 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
72 (X'48')	SMF119S5Len	2	Binary	Length of FTP server Security section
74 (X'4A')	SMF119S5Num	2	Binary	Number of FTP server Security sections

Table 253 shows the FTP server transfer completion specific section of this SMF record.

Table 253. FTP server transfer completion record section

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FSOper	1	Binary	FTP Operation according to SMF77 subtype classification: • X'01': Append • X'02': Delete • X'03': Rename • X'04': Retrieve • X'05': Store • X'06': Store Unique

Table 253. FTP server transfer completion record section (continued)

Offset	Name	Length	Format	Description
1(X'1')		3	Binary	Reserved
4(X'4')	SMF119FT_FSCmd	4	EBCDIC	FTP command (according to RFC 959+)
8(X'8')	SMF119FT_FSFType	4	EBCDIC	File type (SEQ, JES, or SQL)
12(X'C')	SMF119FT_FSDRIP	16	Binary	Remote IP address (data connection)
28(X'1C')	SMF119FT_FSDLIP	16	Binary	Local IP address (data connection)
44(X'2C')	SMF119FT_FSDRPort	2	Binary	Remote port number (data connection - client)
46(X'2E')	SMF119FT_FSDLPort	2	Binary	Local port number (data connection - server)
48 (X'30')	SMF119FT_FSCRIP	16	Binary	Remote IP address (control connection)
64(X'40')	SMF119FT_FSCLIP	16	Binary	Local IP address (control connection)
80 (X'50')	SMF119FT_FSCRPort	2	Binary	Remote port number (control connection - client)
82 (X'52')	SMF119FT_FSCLPort	2	Binary	Local port number (control connection - server)
84(X'54')	SMF119FT_FSSUser	8	EBCDIC	Client User ID on server
92(X'5C')	SMF119FT_FSType	1	EBCDIC	Data type: • A: ASCII • E: EBCDIC • I: Image • B: Double-byte • U: UCS-2
93(X'5D')	SMF119FT_FSMode	1	EBCDIC	Transmission mode: • B: Block • C: Compressed • S: Stream
94(X'5E')	SMF119FT_FSStruct	1	EBCDIC	Data structure: • F: File • R: Record
95(X'5F')	SMF119FT_FSDsType	1	EBCDIC	Data set type: • S: SEQ • P: PDS • H: HFS
96(X'60')	SMF119FT_FSSTime	4	Binary	Transmission start time of day
100(X'64')	SMF119FT_FSSDate	4	Packed	Transmission start date
104(X'68')	SMF119FT_FSETime	4	Binary	Transmission end time of day
108(X'6C')	SMF119FT_FSEDate	4	Packed	Transmission end date
112(X'70')	SMF119FT_FSDur	4	Binary	File transmission duration in units of 1/100 seconds
116(X'74')	SMF119FT_FSBytes	8	Binary	Transmission byte count; 64-bit integer

Table 253. FTP server transfer completion record section (continued)

Offset	Name	Length	Format	Description
124(X'7C')	SMF119FT_FSLReply	4	EBCDIC	Last reply to client (3-digit RFC 959 code, right justified)
128(X'80')	SMF119FT_FSM1	8	EBCDIC	PDS Member name
136(X'88')	SMF119FT_FSRS	8	EBCDIC	Reserved for abnormal end information
144(X'90')	SMF119FT_FSM2	8	EBCDIC	Second PDS member name (if rename operation)
152(X'98')	SMF119FT_FSBytesFloat	8	Floating point hex	z/OS floating point format for transmission byte count
160 (X'A0')	SMF119FT_FSCConnID	4	Binary	TCP connection ID of FTP control connection
164 (X'A4')	SMF119FT_FSDConnID	4	Binary	TCP connection ID of FTP data connection, or 0
168 (X'A8')	SMF119FT_FSSessionID	15	EBCDIC	FTP activity logging session ID. The activity logging session ID uniquely identifies the FTP session between a client and a server. The identifier is created by combining the job name of the FTP daemon with a 5-digit number in the range 00000 - 99 999.
183 (X'B7')		1	Binary	Reserved

Table 254 shows the FTP server transfer completion host name section. This section is optional and is present if gethostbyaddr operation was performed for the local IP address.

Table 254. FTP server transfer completion record section: Host name

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FSHostname	*n*	EBCDIC	Host Name

Table 255 shows the FTP server transfer completion first associated data set name section. This section represents the server MVS or z/OS UNIX data set name associated with a rename or file transfer. Use the Data Set Type field information in the FTP server transfer completion section to determine the type of filename represented here.

Table 255. FTP server transfer completion record section: First associated data set name

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FSFileName1	*n*	EBCDIC	Server MVS or z/OS UNIX file name associated with the file transfer or rename operation. When the operation is a rename, this is the file or data set original name.

Table 256 on page 883 shows the FTP server transfer completion second associated data set name section. This section represents an MVS or z/OS UNIX data set name associated with the rename operation. Use the Data Set Type field information in the FTP server transfer completion section to determine the type of file name represented here.

Table 256. FTP server transfer completion record section: Second associated data set name

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FSFileName2	*n*	EBCDIC	Second MVS or z/OS UNIX file name associated with a rename. This is the new file or data set name.

Table 257 shows the FTP server security section:

Table 257. 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
4 (X'4')	SMF119FT_FSProtoLevel	8	EBCDIC	(Control connection.) Protocol level (present only if Protocol Mechanism is TLS or AT-TLS) Possible values are: • SSLV2 • SSLV3 • TLSV1 • TLSV1.1

Table 257. FTP server security section (continued)

Offset	Name	Length	Format	Description
12 (X'C')	SMF119FT_FSCipherSpec	20	EBCDIC	(Control connection.) 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, TLSV1, or TLSV1.1: • 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 If this field is blank, the value of SMF119FT_FSCipher for the cipher is used for the connection.
32 (X'20')	SMF119FT_FSProtoBufSize	4	Binary	Negotiated protection buffer size
36(X'24')	SMF119FT_FSCipher	2	EBCDIC	(Control connection.) Hexadecimal value of cipher specification (present only when Protocol Mechanism is TLS or AT-TLS).
38(X'26')	SMF119FT_FSFips140	1	Binary	FIPS 140 status • X'00': FIPS 140 off • X'01': FIPS 140 on

FTP server logon failure record (subtype 72)

The FTP server login failure record is collected when an attempt to log in to the z/OS FTP server completes unsuccessfully. A return code within the SMF record provides information as to the cause of the login failure.

The Type 119 FTP server login failure record is collected at the same point in FTP login processing as the equivalent Type 118 FTP server (subtype X'72') SMF record.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the FTP server logon failure record, the TCP/IP stack identification section indicates FTPS as the subcomponent and X'08' (event record) as the record reason.

Table 258 shows the FTP server logon failure record self-defining section:

Table 258. FTP server logon failure record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	Standard SMF Header	24		Standard SMF header; subtype is 72(X'48')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (3)
26(X'1A')		2	Binary	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 logon failure section
40(X'28')	SMF119S1Len	2	Binary	Length of FTP server logon failure section
42(X'2A')	SMF119S1Num	2	Binary	Number of FTP server logon failure sections
44 (X'2C')	SMF119S2Off	4	Binary	Offset to FTP server logon failure Security section
48 (X'30')	SMF119S2Len	2	Binary	Length of FTP server logon failure Security section
50 (X'32')	SMF119S2Num	2	Binary	Number of FTP server logon failure Security sections

Table 259 shows the FTP server logon failure specific section of this SMF record.

Table 259. FTP server logon failure record: logon failure section

Offset	Name	Length	Format	Description
0(X'0')	SMF119FT_FFRIP	16	Binary	Remote IP address
16(X'10')	SMF119FT_FFLIP	16	Binary	Local IP address
32(X'20')	SMF119FT_FFRPort	2	Binary	Remote port number (Client)
34(X'22')	SMF119FT_FFLPort	2	Binary	Local port number (Server)
36(X'24')	SMF119FT_FFUserID	8	EBCDIC	Client User ID received by server

Table 259. FTP server logon failure record: logon failure section (continued)

Offset	Name	Length	Format	Description
44(X'2C')	SMF119FT_FFReason	1	Binary	Login failure reason: • X'00': FTP session terminated after USERID was processed, but before PASSWORD was entered. • X'01': Password is not valid. • X'02': Password has expired. • X'03': User ID has been revoked. • X'04': User does not have server access. • X'05': FTCHKPWD User exit reject login. • X'06': Excessive bad passwords. • X'07': Group ID process failed. • X'08': User ID is unknown. • X'09': Certificate is not valid • X'0A': Client name associated with certificate or ticket does not match user name.
45(X'2D')		3	Binary	Reserved
48 (X'30')	SMF119FT_FFCConnID	4	Binary	TCP connection ID of FTP control connection
52 (X'34')	SMF119FT_FFSessionID	15	EBCDIC	FTP activity logging session ID. The activity logging session ID uniquely identifies the FTP session between a client and a server. The identifier is created by combining the job name of the FTP daemon with a 5-digit number in the range 00000 - 99 999.
67 (X'49')		1	Binary	Reserved

Table 260 shows the FTP server login failure security section:

Table 260. FTP server login failure security section

Offset	Name	Length	Format	Description
0 (X'0')	SMF119FT_FFMechanism	1	EBCDIC	Protection Mechanism: • N: None • T: TLS • G: GSSAPI • A: AT-TLS
1 (X'1')	SMF119FT_FFCProtect	1	EBCDIC	Control Connection Protection Level: • N: None • C: Clear • S: Safe • P: Private

Table 260. FTP server login failure security section (continued)

Offset	Name	Length	Format	Description
2 (X'2')	SMF119FT_FFDProtect	1	EBCDIC	Data connection protection level: • N: None • C: Clear • S: Safe • P: Private
3 (X'3')	SMF119FT_FFLoginMech	1	EBCDIC	Login Method: • P: Password • C: Certificate • ' ': Login failure occurred before login method was determined. • T: Kerberos ticket
4 (X'4')	SMF119FT_FFProtoLevel	8	EBCDIC	Protocol level (present only if Protocol Mechanism is TLS or AT-TLS) Possible values are: • SSLV2 • SSLV3 • TLSV1 • TLSV1.1
12 (X'C')	SMF119FT_FFCipherSpec	20	EBCDIC	Cipher specification (only present 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, TLSV1, or TLSV1.1: • 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_FFProtBuffSize	4	Binary	Negotiated protection buffer size

Table 260. FTP server login failure security section (continued)

Offset	Name	Length	Format	Description
36(X'24')	SMF119FT_FFCipher	2	EBCDIC	Hexadecimal value of cipher specification (present only if protocol mechanism is TLS or AT-TLS.
38(X'26')	SMF119FT_FFFips140	1	Binary	FIPS 140 status • X'00': FIPS 140 off • X'01': FIPS 140 on

IPSec IKE tunnel activation and refresh record (subtype 73)

The IPSec IKE tunnel activation and refresh record is collected whenever the IKE daemon successfully negotiates an IKE tunnel. This record contains information about the characteristics of the IKE tunnel. If you are using the IPSec Network Management Interface (NMI), the common IKE tunnel section of this SMF record is analogous to the NMsecIKETunnel structure.

Table 261 shows the IPSec IKE tunnel activation/refresh record self-defining section.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. In the interface IKE tunnel activation and refresh record, the TCP/IP stack identification section specifies IKE as the subcomponent and X'08' (event record) as the record reason.

Table 261. IPSec IKE tunnel activation/refresh record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	SMF119_HDR	24		Standard SMF Header; subtype is 73(X'49')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (4)
26(X'1A')		2	Binary	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 common IKE tunnel section
40(X'28')	SMF119S1Len	2	Binary	Length of common IKE tunnel section
42(X'2A')	SMF119S1Num	2	Binary	Number of common IKE tunnel sections
44 (X'2C')	SMF119S2Off	4	Binary	Offset to local ID section
48 (X'30')	SMF119S2Len	2	Binary	Length of local ID section
50 (X'32')	SMF119S2Num	2	Binary	Number of local ID sections
52(X'34')	SMF119S3Off	4	Binary	Offset to remote ID section
56(X'38')	SMF119S3Len	2	Binary	Length of remote ID section
58(X'3A')	SMF119S3Num	2	Binary	Number of remote ID sections

Table 262 shows the IPSec common IKE tunnel specific section.

Table 262. IPSec common IKE tunnel specific section

Offset	Name	Length	Format	Description
0 (X'0')		4	Binary	Common IKE tunnel flags The following list identifies the bits, their names, and meaning. • X'80000000', SMF119IS_IKETunIPv6: The IPv6 indicator. If this bit is set, all IKE tunnel security endpoints are IPv6 addresses. If this bit is not set, the endpoints are IPv4 addresses. • X'40000000', SMF119IS_IKETunNATAllowed: NAT traversal indicator. The NAT traversal function is enabled for this IKE tunnel. • X'20000000', SMF119IS_IKETunLclNAT: Local NAT indicator. A NAT has been detected in front of the local security endpoint. • X'10000000', SMF119IS_IKETunRmtNAT: Remote NAT indicator. A NAT has been detected in front of the remote security endpoint. • X'08000000', SMF119IS_IKETunRmtNAPT: Remote NAPT indicator. An NAPT has been detected in front of the remote security endpoint. **Result:** Some NAPTs might be undetected. In that case, the SMF119IS_IKETunRmtNAT bit is set, but this bit is not set. • X'04000000', SMF119IS_IKETunCanInitP1: IKE tunnel (P1) initiation indicator. The local security endpoint can initiate IKE tunnel negotiations with the remote security endpoint. If this bit is not set, the remote security endpoint must initiate IKE tunnel negotiations. Either side can initiate refreshes. • X'02000000', SMF119IS_IKETunFIPS140: FIPS 140 mode indicator. If this field is set, cryptographic operations for this IKE tunnel are performed using cryptographic algorithms and modules that are designed to meet the FIPS 140 requirements; otherwise, cryptographic algorithms and modules that do not meet the FIPS 140 requirements might be used. • Remaining bits: Reserved
4(X'4')	SMF119IS_IKETunID	48	EBCDIC	Tunnel ID for this IKE tunnel.
52(X'34')	SMF119IS_IKETunKeyExchRule	48	EBCDIC	Key exchange rule name for this IKE tunnel.
100(X'64')	SMF119IS_IKETunKeyExchAction	48	EBCDIC	Key exchange action name for this IKE tunnel.
148(X'94')	SMF119IS_IKETunLclEndpt4	4	Binary	One of the following values: • If SMF119IS_IKETunIPv6 is set, this field is the 16–byte IPv6 local security endpoint for this IKE tunnel. • If SMF119IS_IKETunIPv6 is clear, this field is the 4–byte IPv4 local security endpoint for this IKE tunnel.

Table 262. IPSec common IKE tunnel specific section (continued)

Offset	Name	Length	Format	Description
148(X'94')	SMF119IS_IKETunLclEndpt6	16	Binary	One of the following values: • If SMF119IS_IKETunIPv6 is set, this field is the 16–byte IPv6 local security endpoint for this IKE tunnel. • If SMF119IS_IKETunIPv6 is clear, this field is the 4–byte IPv4 local security endpoint for this IKE tunnel.
164(X'A4')	SMF119IS_IKETunRmtEndpt4	4	Binary	One of the following values: • If SMF119IS_IKETunIPv6 is set, this field is the 16–byte IPv6 remote security endpoint for this IKE tunnel. • If SMF119IS_IKETunIPv6 is clear, this field is the 4–byte IPv4 remote security endpoint for this IKE tunnel.
164(X'A4')	SMF119IS_IKETunRmtEndpt6	16	Binary	One of the following values: • If SMF119IS_IKETunIPv6 is set, this field is the 16–byte IPv6 remote security endpoint for this IKE tunnel. • If SMF119IS_IKETunIPv6 is clear, this field is the 4–byte IPv4 remote security endpoint for this IKE tunnel.
180(X'B4')	SMF119IS_IKETunICookie	8	Binary	The icookie for this IKE tunnel
188(X'BC')	SMF119IS_IKETunRCookie	8	Binary	The rcookie for this IKE tunnel
196(X'C4')	SMF119IS_IKETunExchangeMode	1	Binary	Tunnel exchange mode. For IKEv1 SAs, possible values are: • SMF119IS_IKETUN_EXCHMAIN (2) • SMF119IS_IKETUN_EXCHAGGRESSIVE (4) For IKEv2 SAs, this field is not applicable and is 0.
197(X'C5')	SMF119IS_IKETunState	1	Binary	Tunnel state. Possible values are: • SMF119IS_SASTATE_DEACT(1): Dynamic tunnel is deactivated. This value is valid only for record subtype 74. • SMF119IS_SASTATE_ACTIVE (2): Tunnel is active. This value is valid only for record subtype 73. • SMF119IS_SASTATE_EXPIRED (3): Dynamic tunnel is expired. This value is valid only for record subtype 74.

Table 262. IPSec common IKE tunnel specific section (continued)

Offset	Name	Length	Format	Description
198(X'C6')	SMF119IS_IKETunAuthAlg	1	Binary	Tunnel authentication algorithm. Possible values are: • SMF119IS_AUTH_HMAC_MD5 (38) The tunnel uses HMAC-MD5 authentication with the full 128-bit Integrity Check Value (ICV). This value is applicable only to IKEv1 tunnels. • SMF119IS_AUTH_HMAC_SHA1 (39) The tunnel uses HMAC-SHA1 authentication with the full 160-bit ICV. This value is applicable only to IKEv1 tunnels. • SMF119IS_AUTH_HMAC_MD5_96 (40) The tunnel uses HMAC-MD5 authentication with ICV truncation to 96 bits. This value is applicable only to IKEv2 tunnels. • SMF119IS_AUTH_HMAC_SHA1_96 (41) The tunnel uses HMAC-SHA1 authentication with ICV truncation to 96 bits. This value is applicable only to IKEv2 tunnels. • SMF119IS_AUTH_HMAC_SHA2_256_128 (7) The tunnel uses HMAC-SHA2-256 authentication with ICV truncation to 128 bits. • SMF119IS_AUTH_HMAC_SHA2_384_192 (13) The tunnel uses HMAC-SHA2-384 authentication with ICV truncation to 192 bits. • SMF119IS_AUTH_HMAC_SHA2_512_256 (14) The tunnel uses HMAC-SHA2-512 authentication with ICV truncation to 256 bits. • SMF119IS_AUTH_AES128_XCBC_96 (9) The tunnel uses AES128-XCBC authentication with ICV truncation to 96 bits.
199(X'C7')	SMF119IS_IKETunEncryptAlg	1	Binary	Tunnel encryption algorithm. Possible values are: • SMF119IS_ENCR_DES(18) • SMF119IS_ENCR_3DES(3) • SMF119IS_ENCR_AES_CBC(12) AES encryption algorithm in Cipher Block Chaining (CBC) mode. See SMF119IS_IKETunEncryptKeyLength; it identifies the key length in use.
200(X'C8')	SMF119IS_IKETunDHGroup	4	Binary	Diffie-Hellman group used to generate keying material for this IKE tunnel.
204('xCC')	SMF119IS_IKETunPeerAuthMethod	1	Binary	Tunnel peer authentication method. Possible values are: • SMF119IS_IKETUN_PRESHAREDKEY (3) • SMF119IS_IKETUN_RSASIGNATURE (2) • SMF119IS_IKETUN_ECDSA_256 (4) • SMF119IS_IKETUN_ECDSA_384 (5) • SMF119IS_IKETUN_ECDSA_521 (6)
205(X'CD')	SMF119IS_IKETunRole	1	Binary	Tunnel role. Possible values are: • SMF119IS_IKETUN_INITIATOR (1) • SMF119IS_IKETUN_RESPONDER (2)

Table 262. IPSec common IKE tunnel specific section (continued)

Offset	Name	Length	Format	Description
206(X'CE')	SMF119IS_IKETunNATTLevel	1	Binary	NAT traversal support level. Possible values are: • SMF119IS_IKETUN_NATTNONE (0): No NAT traversal support; support is either not configured or not negotiated. • SMF119IS_IKETUN_NATTRFCD2 (1): RFC 3947 draft 2 support. • SMF119IS_IKETUN_NATTRFCD3 (3): RFC 3947 draft 3 support. • SMF119IS_IKETUN_NATTRFC (4): RFC 3947 support with non-z/OS peer. • SMF119IS_IKETUN_NATTZOS (5): RFC 3947 support with z/OS peer. • SMF119IS_IKETUN_NATTV2 (6): IKEv2 NAT traversal support. • SMF119IS_IKETUN_NATTV2ZOS (7): IKEv2 NAT traversal support with z/OS peer.
207(X'CF')	SMF119IS_IKETunExtState	1	Binary	Extended tunnel state information. Possible values are: • SMF119IS_EXTSASTATE_ACTIVATE (1): This value is a new Phase 1 activation. This value is valid only for record subtype 73. • SMF119IS_EXTSASTATE_REFRESH (2): This value is a Phase 1 refresh. This value is valid only for record subtype 73. The following values are valid only for record subtype 74: • SMF119IS_EXTSASTATE_DEACT (3): This tunnel is deactivated (not as a result of error or negotiation failure). • SMF119IS_EXTSASTATE_PROPOSAL (4): Negotiation failure; no proposal matched the current policy. • SMF119IS_EXTSASTATE_RETRANS (5): Negotiation failure; a retransmit limit was encountered while negotiating this tunnel. • SMF119IS_EXTSASTATE_POLICY (6): Negotiation failure; a policy mismatch other than a proposal mismatch occurred. For example, no valid KeyExchangeRule value was set. • SMF119IS_EXTSASTATE_OTHER (7): Negotiation failure; the data in an ISAKMP packet was not valid, or an internal error occurred.
208(X'D0')	SMF119IS_IKETunLifesize	8	Binary	Tunnel lifesize. If this value is not 0, this value indicates the lifesize limit for the tunnel, in bytes.
216(X'D8')	SMF119IS_IKETunLifetime	4	Binary	Tunnel lifetime. This value indicates the total number of seconds the tunnel remains active.
220(X'DC')	SMF119IS_IKETunLifetimeRefresh	4	Binary	Tunnel lifetime refresh. This value indicates the time at which the tunnel is refreshed (in UNIX format).
224(X'E0')	SMF119IS_IKETunLifetimeExpire	4	Binary	Tunnel lifetime expiration. This value indicates the time at which the tunnel expires (in UNIX format).
228(X'E4')	SMF119IS_IKETunRmtUDPPort	2	Binary	Remote UDP port used for IKE negotiations.

Table 262. IPSec common IKE tunnel specific section (continued)

Offset	Name	Length	Format	Description
230(X'E6')	SMF119IS_IKETunLIDType	1	Binary	ISAKMP identity type for the local security endpoint identity, as defined in RFC 2407. ISAKMP peers exchange and verify identities as part of the IKE tunnel (phase 1) negotiation.
231(X'E7')	SMF119IS_IKETunRIDType	1	Binary	ISAKMP identity type for the remote security endpoint identity, as defined in RFC 2407. ISAKMP peers exchange and verify identities as part of the IKE tunnel (phase 1) negotiation.
232(X'E8')	SMF119IS_IKETunStartTime	4	Binary	Tunnel start time. Indicates the time at which the tunnel was activated or refreshed (in UNIX format).
236(X'EC')	SMF119IS_IKETunMajorVer	1	Binary	Major version of the IKE protocol in use. Only the low-order 4 bits are used.
237(X'ED')	SMF119IS_IKETunMinorVer	1	Binary	Minor version of the IKE protocol in use. Only the low-order 4 bits are used.
238(X'EE')	SMF119IS_IKETunPseudoRandomFunc	1	Binary	Pseudo-random function used for seeding keying material. One of the following values: • SMF119IS_AUTH_HMAC_MD5 (38) • SMF119IS_AUTH_HMAC_SHA1 (39) • SMF119IS_AUTH_HMAC_SHA2_256 (15) • SMF119IS_AUTH_HMAC_SHA2_384 (16) • SMF119IS_AUTH_HMAC_SHA2_512 (17) • SMF119IS_AUTH_AES128_XCBC (18)
239(X'EF')	SMF119IS_IKETunLocalAuthMethod	1	Binary	The authentication method for the local endpoint. One of the following values: • SMF119IS_IKETUN_PRESHAREDKEY (3) • SMF119IS_IKETUN_RSASIGNATURE (2) • SMF119IS_IKETUN_ECDSA_256 (4) • SMF119IS_IKETUN_ECDSA_384 (5) • SMF119IS_IKETUN_ECDSA_521 (6) • SMF119IS_IKETUN_DS (7)
240(X'F0')	SMF119IS_IKETunReauthInterval	4	Binary	Reauthentication interval. Indicates the number of seconds between reauthentication operations.
244(X'F4')	SMF119IS_IKETunReauthTime	4	Binary	Tunnel reauthentication time. Indicates the time at which the tunnel is reauthenticated (in UNIX format).
248(X'F8')	SMF119IS_IKETunGeneration	4	Binary	Tunnel generation number. The first IKE tunnel with a particular tunnel ID has generation 1. Subsequent refreshes of this IKE tunnel have the same tunnel ID, but with higher generation numbers.
252(X'FC')	SMF119IS_IKETunEncryptKeyLength	4	Binary	Encryption key length for variable-length algorithms, in bits. This value is 0 for encryption algorithms that have a fixed key length (such as DES and 3DES) and nonzero for encryption algorithms that have a variable key length (such as AES-CBC). **Result:** Example values are 128 and 256.

Table 263 on page 894 shows the IPSec local ID specific section.

Table 263. IPSec local ID specific section

Offset	Name	Length	Format	Description
0(X'0')	SMF119IS_LocalID	n	EBCDIC	Contents of the local identity used to negotiate the IKE tunnel. Regardless of the identity type, the value is expressed as an EBCDIC string (an IP address is returned in printable form).

Table 264 shows the IPSec remote ID specific section:

Table 264. IPSec remote ID specific section

Offset	Name	Length	Format	Description
0(X'0')	SMF119IS_RemoteID	n	EBCDIC	Contents of the remote identity used to negotiate the IKE tunnel. Regardless of the identity type, the value is expressed as an EBCDIC string (an IP address is returned in printable form).

IPSec IKE tunnel deactivation and expire record (subtype 74)

The IPSec IKE tunnel deactivation record is collected whenever the IKE daemon deactivates an IKE tunnel. This record contains information about the characteristics of the IKE tunnel that is being deleted. If a tunnel is being deactivated as a result of a failure, the values might be unknown. Field values might be unknown because the negotiation has not progressed far enough to determine a value; therefore, those fields are set to the value 0. If you are using the IPSec NMI, the common IKE tunnel section of this SMF record is analogous to the NMsecIKETunnel structure and the IKE counter section is analogous to the NMsecIKETunStats structure.

Result: When an IKE tunnel expires, it is not deleted until all dynamic tunnels that are associated with that tunnel are deleted. Typically, there is one subtype 74 record for the expiration of the IKE tunnel, and there is a second subtype 74 record at a later time for the deletion of the IKE tunnel.

When a TCP/IP stack is stopped, IKE tunnels are not deleted immediately. If an IKE tunnel expires while the stack is stopped, a subtype 74 record is generated for the expiration of that tunnel. However, if the stack restarts before the IKE tunnel expires, the IKE tunnel remains valid and continues to be used until it expires.

Table 265 on page 895 shows the contents of the IPSec IKE tunnel deactivation and expire record self-defining section.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the interface IKE tunnel deactivation and expire record, the TCP/IP stack identification section indicates IKE as the subcomponent and X'08' (event record) as the record reason.

See Table 262 on page 889 for the contents of the common IKE tunnel section.

Table 265. IPSec IKE tunnel deactivation and expire record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	SMF119_HDR	24	EBCDIC	Standard SMF Header; subtype is 74(X'4A')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (5)
26(X'1A')		2	Binary	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 common IKE tunnel section
40(X'28')	SMF119S1Len	2	Binary	Length of common IKE tunnel section
42(X'2A')	SMF119S1Num	2	Binary	Number of common IKE tunnel sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to IKE counter section
48(X'30')	SMF119S2Len	2	Binary	Length of IKE counter section
50(X'32')	SMF119S2Num	2	Binary	Number of IKE counter sections
52(X'34')	SMF119S3Off	4	Binary	Offset to the local ID section
56(X'38')	SMF119S3Len	2	Binary	Length of local ID section
58(X'3A')	SMF119S3Num	2	Binary	Number of local ID sections
60(X'3C')	SMF119S4Off	4	Binary	Offset to remote ID section
64(X'40')	SMF119S4Len	2	Binary	Length of remote ID sections
66(X'42')	SMF119S4Num	2	Binary	Number of remote ID sections

Table 266 lists the IPSec IKE counter specific section.

Table 266. IPSec IKE counter specific section

Offset	Name	Length	Format	Description
0(X'0')	SMF119IS_IKETunP2Current	4	Binary	Current count of active dynamic tunnels associated with this IKE tunnel
4(X'4')	SMF119IS_IKETunP2InProgress	4	Binary	Current count of pending or in progress dynamic tunnels associated with this IKE tunnel

Table 266. IPSec IKE counter specific section (continued)

Offset	Name	Length	Format	Description
8(X'8')	SMF119IS_IKETunP2LclActSuccess	4	Binary	Cumulative count of locally initiated dynamic tunnels that were successfully activated for this IKE tunnel
12(X'C')	SMF119IS_IKETunP2RmtActSuccess	4	Binary	Cumulative count of remotely initiated dynamic tunnel activations that were successfully activated for this IKE tunnel
16(X'10')	SMF119IS_IKETunP2LclActFailure	4	Binary	Cumulative count of failed dynamic tunnel activations that were initiated locally for this IKE tunnel
20(X'14')	SMF119IS_IKETunP2RmtActFailure	4	Binary	Cumulative count of failed dynamic tunnel activations that were initiated for this IKE tunnel
24(X'18')	SMF119IS_IKETunBytes	8	Binary	Cumulative number of bytes protected by this IKE tunnel
32(X'20')	SMF119IS_IKETunP1Rexmit	8	Binary	Cumulative number of retransmitted key exchange (phase 1) messages sent for this tunnel over the life of the IKE daemon. This data is cumulative even across TCP/IP restarts.

Table 266. IPSec IKE counter specific section (continued)

Offset	Name	Length	Format	Description
40(X'28')	SMF119IS_IKETunP1Replay	8	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 a cross TCP/IP restarts.
48(X'30')	SMF119IS_IKETunP2Rexmit	8	Binary	Cumulative number of retransmitted key exchange (phase 2) messages sent for this tunnel over the life of the IKE daemon. This data is cumulative even a cross TCP/IP restarts.
56(X'38')	SMF119IS_IKETunP2Replay	8	Binary	Cumulative number of replayed key exchange (phase 2) messages received for this stack over the life of the IKE daemon. This data is cumulative even a cross TCP/IP restarts.

See Table 263 on page 894 for the contents of the local ID section.

See Table 264 on page 894 for the contents of the remote ID section.

IPSec dynamic tunnel activation and refresh record (subtype 75)

The IPSec dynamic tunnel activation record is collected whenever the IKE daemon successfully negotiates a dynamic tunnel and installs it in the TCP/IP stack. This record contains information about the characteristics of the dynamic tunnel that is to be negotiated. If you are using the IPSec NMI, the common IP tunnel section of this SMF record is analogous to the NMsecIPTunnel structure, the dynamic tunnel section is analogous to the NMsecIPDynTunnel structure, and the IKE dynamic tunnel section is analogous to the NMsecIPDynamicIKE structure.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the IPSec dynamic tunnel activation record, the TCP/IP Stack identification section indicates IKE as the subcomponent and X'08' (event record) as the record reason.

Table 267 lists the contents of the IPSec dynamic tunnel activation record self-defining section.

Table 267. IPSec dynamic tunnel activation record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	SMF119_HDR	24	EBCDIC	Standard SMF Header; subtype is 75(X'4B').
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (6).
26(X'1A')		2	Binary	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 common IP tunnel section.
40(X'28')	SMF119S1Len	2	Binary	Length of common IP tunnel section.
42(X'2A')	SMF119S1Num	2	Binary	Number of common IP tunnel sections.
44(X'2C')	SMF119S2Off	4	Binary	Offset to dynamic tunnel section.
48(X'30')	SMF119S2Len	2	Binary	Length of dynamic tunnel section.
50(X'32')	SMF119S2Num	2	Binary	Number of tunnel sections.
52(X'34')	SMF119S3Off	4	Binary	Offset to IKE dynamic tunnel sections.
56(X'38')	SMF119S3Len	2	Binary	Length of IKE dynamic tunnel section.
58(X'3A')	SMF119S3Num	2	Binary	Number of IKE dynamic tunnel sections.
60(X'3C')	SMF119S4Off	4	Binary	Offset to local client ID section.
64(X'40')	SMF119S4Len	2	Binary	Length of local client ID section.
66(X'42')	SMF119S4Num	2	Binary	Number of local client ID sections.

Table 267. IPSec dynamic tunnel activation record self-defining section (continued)

Offset	Name	Length	Format	Description
68(X'44')	SMF119S5Off	4	Binary	Offset to remote client ID sections.
72(X'48')	SMF119S5Len	2	Binary	Length of remote client ID section.
74(X'5A')	SMF119S5Num	2	Binary	Number of remote client ID sections.

Table 268 lists the IPSec common IP tunnel specific section.

Table 268. IPSec common IP tunnel specific section

Offset	Name	Ln.	Format	Description
0(X'0')	SMF119IS_IPTunID	48	EBCDIC	Tunnel ID
48x'30')	SMF119IS_IPTunVPNAction	48	EBCDIC	Tunnel VPN action name
96(X'60')		4	Binary	IP tunnel flags. The following list identifies the bits, their names, and meaning. • X'80000000', SMF119IS_IPTunFlagIPv6: IPv6 indicator. If set, security endpoint addresses and data endpoint addresses are IPv6; otherwise, they are IPv4. • X'40000000', SMF119IS_IPTunFIPS140: FIPS 140 mode indicator. If this field is set, cryptographic operations for this tunnel are performed using cryptographic algorithms and modules that are designed to meet the FIPS 140 requirements; otherwise, cryptographic algorithms and modules that do not meet the FIPS 140 requirements might be used. • All remaining bits: Reserved
100(X'64')	SMF119IS_IPTunType	1	Binary	Tunnel type. One of the following values: • SMF119IS_IPTUN_MANUAL (1) Manual IP tunnel • SMF119IS_IPTUN_STACK (2) Dynamic IP tunnel, as seen by TCP/IP stack • SMF119IS_IPTUN_IKE (3) Dynamic IP tunnel, as seen by IKE
101(X'65')	SMF119IS_IPTunState	1	Binary	One of the following tunnel states: • SMF119IS_SASTATE_DEACT(1) Dynamic tunnel is deactivated. This value is valid only on record subtypes 76 and 78. • SMF119IS_SASTATE_ACTIVE (2) Manual or dynamic tunnel is active. This value is valid only on record subtype 75. • SMF119IS_SASTATE_EXPIRED (3) Dynamic tunnel is expired. This value is valid only on record subtype 78.
102(X'66')	SMF119IS_IPTunRsvd2	2	Binary	Reserved
104(X'68')	SMF119IS_IPTunLclEndpt4	4	Binary	One of the following values: • If SMF119IS_IPTunFlagIPv6 is set, this field is the 16–byte IPv6 local security endpoint address. • If SMF119IS_IPTunFlagIPv6 is clear, this field is the 4–byte IPv4 local security endpoint address.

Table 268. IPSec common IP tunnel specific section (continued)

Offset	Name	Ln.	Format	Description
104(X'68')	SMF119IS_IPTunLclEndpt6	16	Binary	One of the following values: • If SMF119IS_IPTunFlagIPv6 is set, this field is the 16–byte IPv6 local security endpoint address. • If SMF119IS_IPTunFlagIPv6 is clear, this field is the 4–byte IPv4 local security endpoint address.
120(X'78')	SMF119IS_IPTunRmtEndpt4	4	Binary	One of the following values: • If SMF119IS_IPTunFlagIPv6 is set, this field is the 16–byte IPv6 remote security endpoint address. • If SMF119IS_IPTunFlagIPv6 is clear this field is the 4 byte IPv4 remote security endpoint address.
120(X'78')	SMF119IS_IPTunRmtEndpt6	16	Binary	One of the following values: • If SMF119IS_IPTunFlagIPv6 is set, this field is the 16–byte IPv6 remote security endpoint address. • If SMF119IS_IPTunFlagIPv6 is clear this field is the 4 byte IPv4 remote security endpoint address.
136(X'88')	SMF119IS_IPTunEncapMode	1	Binary	One of the following tunnel encapsulation modes: • SMF119IS_IPTUN_TUNNELMODE (1) • SMF119IS_IPTUN_TRANSPORTMODE (2)
137(X'89')	SMF119IS_IPTunAuthProto	1	Binary	One of the following tunnel authentication protocols: • IPPROTO_AH (51) • IPPROTO_ESP (50)

Table 268. IPSec common IP tunnel specific section (continued)

Offset	Name	Ln.	Format	Description
138(X'8A')	SMF119IS_IPTunAuthAlg	1	Binary	One of the following tunnel authentication alogorithms: • SMF119IS_AUTH_NULL (0) The tunnel uses NULL authentication, or obtains authentication using a combined-mode encryption algorithm (see SMF119IS_IPTunEncryptAlg). • SMF119IS_AUTH_HMAC_MD5(38) The tunnel uses HMAC-MD5 authentication with Integrity Check Value (ICV) truncation to 96 bits. • SMF119IS_AUTH_HMAC_SHA1(39) The tunnel uses HMAC-SHA1 authentication with ICV truncation to 96 bits. • SMF119IS_AUTH_HMAC_SHA2_256_128 (7) The tunnel uses HMAC-SHA2-256 authentication with ICV truncation to 128 bits. • SMF119IS_AUTH_HMAC_SHA2_384_192 (13) The tunnel uses HMAC-SHA2-384 authentication with ICV truncation to 192 bits. • SMF119IS_AUTH_HMAC_SHA2_512_256 (14) The tunnel uses HMAC-SHA2-512 authentication with ICV truncation to 256 bits. • SMF119IS_AUTH_AES128_XCBC_96 (9) The tunnel uses AES128-XCBC authentication with ICV truncation to 96 bits. • SMF119IS_AUTH_AES_GMAC_128 (4) The tunnel uses AES-GMAC authentication with a key length of 128 bits. • SMF119IS_AUTH_AES_GMAC_256 (6) The tunnel uses AES-GMAC authentication with a key length of 256 bits.
139(X'8B')	SMF119IS_IPTunEncryptAlg	1	Binary	One of the following tunnel encryption algorithms: • SMF119IS_ENCR_NONE (0) • SMF119IS_ENCR_NULL (11) • SMF119IS_ENCR_DES (18) • SMF119IS_ENCR_3DES (3) • SMF119IS_ENCR_AES_CBC (12) AES encryption algorithm in Cipher Block Chaining (CBC) mode. See also SMF119IS_IPTunEncryptKeyLength which identifies the key length in use. • SMF119IS_ENCR_AES_GCM_16 (20) AES encryption algorithm in Galois/Counter Mode (GCM) using a 16-octet IV. See SMF119IS_IPTunEncryptKeyLength; it identifies the key length in use.
140(X'8C')	SMF119IS_IPTunInbAuthSPI	4	Binary	Tunnel inbound authentication SPI.
144(X'90')	SMF119IS_IPTunOutbAuthSPI	4	Binary	Tunnel outbound authentication SPI.
148(X'94')	SMF119IS_IPTunInbEncryptSPI	4	Binary	Tunnel inbound encryption SPI.
152(X'98')	SMF119IS_IPTunOutbEncryptSPI	4	Binary	Tunnel outbound encryption SPI.
156(X'9C')	SMF119IS_IPTunStartTime	4	Binary	Indicates the tunnel start time at which the tunnel was activated or refreshed, in UNIX format.

Table 268. IPSec common IP tunnel specific section (continued)

Offset	Name	Ln.	Format	Description
160(X'A0')	SMF119IS_IPTunEncryptKeyLength	4	Binary	Encryption key length for variable-length algorithms, in bits. Zero for encryption algorithms that have a fixed key length (such as DES and 3DES) and nonzero for encryption algorithms that have a variable key length (such as AES-CBC and AES-GCM). **Result:** Example values are 128 and 256.

Table 269 lists the IPSec dynamic tunnel specific section.

Table 269. IPSec dynamic tunnel specific section

Offset	Name	Length	Format	Description
0(X'0')		4	Binary	The following list identifies the bits, their names, and meaning. • X'80000000', SMF119IS_IPDynUDPEncap: UDP encapsulation indicator. The tunnel uses UDP encapsulation mode. • X'40000000', SMF119IS_IPDynLclNAT: Local NAT indicator. A NAT has been detected in front of the local security endpoint. • X'20000000', SMF119IS_IPDynRmtNAT: Remote NAT indicator. A NAT has been detected in front of the remote security endpoint. • X'10000000', SMF119IS_IPDynRmtNAPT: Remote NAPT indicator. An NAPT has been detected in front of the remote security endpoint. **Result:** Some NAPTs might be undetected. In that case, the SMF119IS_IKETunRmtNAT bit is set, but this bit is not set. • X'08000000', SMF119IS_IPDynRmtGW: Remote NAT traversal gateway indicator. The tunnel uses UDP encapsulation and the remote security endpoint is acting as an IPSec gateway.

Table 269. IPSec dynamic tunnel specific section (continued)

Offset	Name	Length	Format	Description
0(X'0') Cont.		Cont.	Cont.	One of the following values: • X'04000000', SMF119IS_IPDynRmtZOS: Remote z/OS indicator. The remote peer has been detected to be z/OS. The remote peer might be running z/OS but it might not be detected as such, if NAT traversal is not enabled. • X'02000000', SMF119IS_IPDynCanInitP2: Dynamic tunnel (P2) 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. • X'01000000', SMF119IS_IPDynSrcIsSingle: Single source address indicator. Traffic source address is indicated by the SMF119IS_IPDynSrcAddr4 or SMF119IS_IPDynSrcAddr6 fields. • X'00800000', SMF119IS_IPDynSrcIsPrefix: Prefixed source address indicator. Traffic source address is indicated by the SMF119IS_IPDynSrcAddr4 or SMF119IS_IPDynSrcAddr6, fields and the source address prefix is indicated by the SMF119IS_IPDynSrcAddrPrefix field. • X'00400000', SMF119IS_IPDynSrcIsRange: Ranged source address indicator. Traffic source address range is indicated by the SMF119IS_IPDynSrcAddr4 and SMF119IS_IPDynSrcAddrRange4 fields, or by the SMF119IS_IPDynSrcAddr6 and SMF119IS_IPDynSrcAddrRange6 fields. • X'00200000', SMF119IS_IPDynDstIsSingle: Single destination address indicator. Traffic destination address is indicated by the SMF119IS_IPDynDstAddr4 or SMF119IS_IPDynDstAddr6 fields.

Table 269. IPSec dynamic tunnel specific section (continued)

Offset	Name	Length	Format	Description
0(X'0') Cont		Cont.	Cont.	One of the following values: • X'000100000', SMF119IS_IPDynDstIsPrefix:Prefixed destination address indicator. Traffic destination address is indicated by the SMF119IS_IPDynDstAddr4 or SMF119IS_IPDynDstAddr6 fields, and destination address prefix is indicated by the SMF119IS_IPDynDstAddrPrefix field. • X'00080000', SMF119IS_IPDynDstIsRange: Ranged destination address indicator. Traffic destination address range is indicated by the SMF119IS_IPDynDstAddr4 and SMF119IS_IPDynDstAddrRange4 fields, or by the SMF119IS_IPDynDstAddr6 and SMF119IS_IPDynDstAddrRange6 field. • X'00040000', SMF119IS_IPDynTransportOpaque: Opaque transport selector indicator. If set, the dynamic tunnel is protecting data traffic where the upper layer selectors, source and destination ports, ICMP or ICMPv6 type and code or IPv6 Mobility header type are not available as a result of fragmentation. • All remaining bits: Reserved
4(X'4')	SMF119IS_IPDynVPNRule	48	EBCDIC	Dynamic VPN rule name for this tunnel. This field is blank if there is no local dynamic VPN rule.
52(X'34')	SMF119IS_IPDynP1TunnelID	48	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.
100(X'64')	SMF119IS_IPDynLifesize	8	Binary	Tunnel lifesize. Nonzero values indicate the lifesize value limit for the tunnel, in bytes.
108(X'6C')	SMF119IS_IPDynLifesizeRefresh	8	Binary	Tunnel lifesize refresh. Nonzero values indicate the lifesize value at which the tunnel is refreshed, in bytes.
116(X'74')	SMF119IS_IPDynLifetimeExpire	4	Binary	Tunnel lifetime. Indicates the time at which the tunnel expires, in UNIX format.
120(X'78')	SMF119IS_IPDynLifetimeRefresh	4	Binary	Tunnel lifetime refresh. Indicates the time at which the tunnel is refreshed, in UNIX format.

Table 269. IPSec dynamic tunnel specific section (continued)

Offset	Name	Length	Format	Description
124(X'7C')	SMF119IS_IPDynVPNLifeExpire	4	Binary	Tunnel VPN lifetime expire. Nonzero values indicate the time at which the tunnel family ceases to be refreshed, in UNIX format. This field retains its original value for a refreshed tunnel.
128(X'80')	SMF119IS_IPDynActMethod	1	Binary	One of the following tunnel activation methods: • SMF119IS_DYNTUN_USER (1): User activation (from the command line). • SMF119IS_DYNTUN_REMOTE (2): Remote activation from IPSec peer. • SMF119IS_DYNTUN_ONDEMAND (3): On-demand activation caused by IP traffic. • SMF119IS_DYNTUN_TAKEOVER (5): SWSA activation as a result of a DVIPA takeover. • SMF119IS_DYNTUN_AUTOACT (6): Auto-activation This field retains its original value for a refreshed tunnel.
129(X'81')	SMF119IS_IPDynRsvd2	3	Binary	Reserved bits
132(X'84')	SMF119IS_IPDynRmtUDPPort	2	Binary	If the tunnel uses UDP encapsulation mode, this value is the IKE UDP port of the remote security endpoint; otherwise, the value is 0.
134(X'86')	SMF119IS_IPDynRsvd3	2	Binary	Reserved bits
136(X'88')	SMF119IS_IPDynSrcNATOA	4	Binary	Source NAT original IP address. NAT original IP addresses 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 an IKEv1 peer did not send a source NAT-OA payload, the value of this field is 0. **Restriction:** An IKEv1 peer at a pre-RFC3947 NAT traversal support level cannot send a source NAT-OA payload.

Table 269. IPSec dynamic tunnel specific section (continued)

Offset	Name	Length	Format	Description
140(X'8C')	SMF119IS_IPDynDstNATOA	4	Binary	Destination NAT original IP address. NAT original IP addresses 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 an IKEv1 peer did not send a source NAT-OA payload, the value of this field is 0. **Restriction:** An IKEv1 peer at a pre-RFC3947 NAT traversal support level cannot send a source NAT-OA payload.
144(X'90')	SMF119IS_IPDynProtocol	1	Binary	Protocol for tunnel data. If the value is 0, the tunnel includes all protocols.
145(X'91')	SMF119IS_IPDynRsvd4	3	Binary	Reserved bits
148(X'94')	SMF119IS_IPDynSrcPort	2	Binary	Low end of source port range for tunnel data or 0 if the tunnel is not limited to TCP or UDP.
150(X'96')	SMF119IS_IPDynDstPort	2	Binary	Low end of destination port range for tunnel data, or 0 if the tunnel is not limited to TCP or UDP.
152(X'98')	SMF119IS_IPDynSrcAddr4	4	Binary	One of the following values: • If the SMF119IS_IPDynSrcIsSingle field is set, this field is the IPv4 or IPv6 source address for tunnel data. • If the SMF119IS_IPDynSrcIsPrefix field is set, this field is the IPv4 or IPv6 source address base for tunnel data. • If the SMF119IS_IPDynSrcIsRange field is set, this field is the low end of the IPv4 or IPv6 source address range for tunnel data.
152(X'98')	SMF119IS_IPDynSrcAddr6	16	Binary	One of the following values: • If SMF119IS_IPTunFlagIPv6 is set, this field is the 16–byte IPv6 local security endpoint address. • If SMF119IS_IPTunFlagIPv6 is clear, this field is the 4–byte IPv4 local security endpoint address.
168(X'A8')	SMF119IS_IPDynSrcAddrRange4	4	Binary	If the SMF119IS_IPDynSrcIsRange field is set, this field is the highest address in the range of the IPv4 or IPv6 source addresses tunnel data.
168(X'A8')	SMF119IS_IPDynSrcAddrRange6	16	Binary	If the SMF119IS_IPDynSrcIsRange field is set, this field is the highest address in the range of the IPv4 or IPv6 source addresses tunnel data.

Table 269. IPSec dynamic tunnel specific section (continued)

Offset	Name	Length	Format	Description
184(X'B8')	SMF119IS_IPDynDstAddr4	4	Binary	One of the following values: • If the SMF119IS_IPDynDstIsSingle field is set, this field is the IPv4 or IPv6 destination address for tunnel data. • If the SMF119IS_IPDynDstIsPrefix field is set, this field is the IPv4 or IPv6 destination address base for tunnel data. • If the SMF119IS_IPDynDstIsRange field is set, this field is the lowest IPv4 or IPv6 destination address in the range for tunnel data.
184(X'B8')	SMF119IS_IPDynDstAddr6	16	Binary	One of the following values: • If the SMF119IS_IPDynDstIsSingle field is set, this field is the IPv4 or IPv6 destination address for tunnel data. • If the SMF119IS_IPDynDstIsPrefix field is set, this field is the IPv4 or IPv6 destination address base for tunnel data. • If the SMF119IS_IPDynDstIsRange field is set, this field is the lowest IPv4 or IPv6 destination address in the range for tunnel data.
200(X'C8')	SMF119IS_IPDynDstAddrRange4	4	Binary	If the SMF119IS_IPDynDstIsRange field is set, this field is the highest IPv4 or IPv6 destination address in the range range for tunnel data.
200(X'C8')	SMF119IS_IPDynDstAddrRange6	16	Binary	If the SMF119IS_IPDynDstIsRange field is set, this field is the highest IPv4 or IPv6 destination address in the range range for tunnel data.
216(X'D8')	SMF119IS_IPDynSrcAddrPrefix	1	Binary	If the SMF119IS_IPDynSrcIsPrefix field is set, this field is the length of the tunnel data source address prefix in bits.
217(X'D9')	SMF119IS_IPDynDstAddrPrefix	1	Binary	If the SMF119IS_IPDynDstIsPrefix field is set, this field is the length of the tunnel data destination address prefix in bits.
218(X'DA')	SMF119IS_IPDynMajorVer	1	Binary	Major version of the IKE protocol in use. Only the low-order 4 bits are used.
219(X'DB')	SMF119IS_IPDynMinorVer	1	Binary	Minor version of the IKE protocol in use. Only the low-order 4 bits are used.
220(X'DC')	SMF119IS_IPDynType	1	Binary	Low end of ICMP, ICMPv6, or MIPv6 type range for tunnel data; otherwise, this value is 0 if the tunnel is not limited to ICMP, ICMPv6, or MIPv6.
221(X'DD')	SMF119IS_IPDynTypeRange	1	Binary	High end of ICMP, ICMPv6, or MIPv6 type range for tunnel data; otherwise this value is 0 if the tunnel is not limited to ICMP, ICMPv6, or MIPv6. A tunnel applying to all type values is indicated as a value in the range 0- 255.

Table 269. IPSec dynamic tunnel specific section (continued)

Offset	Name	Length	Format	Description
222(X'DE')	SMF119IS_IPDynCode	1	Binary	Low end of ICMP or ICMPv6 code range for tunnel data; otherwise this value is 0 if the tunnel is not limited to ICMP or ICMPv6.
223(X'DF')	SMF119IS_IPDynCodeRange	1	Binary	High end of ICMP or ICMPv6 code range for tunnel data; otherwise, this value is 0 if the tunnel is not limited to ICMP or ICMPv6. A tunnel applying to all code values is indicated as a value in the range 0 - 255.
224(X'E0')	SMF119IS_IPDynSrcPortRange	2	Binary	High end of source port range for tunnel data; otherwise this values is 0 if the tunnel is not limited to TCP or UDP. A tunnel applying to all source port values is indicated as a value in the range 0-65 535.
226(X'E2')	SMF119IS_IPDynDstPortRange	2	Binary	High end of destination port range for tunnel data, or 0 if the tunnel is not limited to TCP or UDP. A tunnel applying to all destination port values is indicated as a value in the range 0 - 65 535.
228(X'E4')	SMF119IS_IPDynGeneration	4	Binary	Tunnel generation number. The first dynamic tunnel with a particular tunnel ID has generation 1. Subsequent refreshes of this dynamic tunnel have the same tunnel ID but higher generation numbers.

Table 270 lists the IPSec IKE dynamic tunnel specific section.

Table 270. IPSec IKE dynamic tunnel specific section

Offset	Name	Length	Format	Description
0(X'0')	SMF119IS_IPDynIKERsvd1	4	Binary	Reserved bits.
4(X'4')	SMF119IS_IPDynIKEFilter	48	EBCDIC	Filter name for the IP filter related to this dynamic tunnel.
52(X'34')	SMF119IS_IPDynIKEDHGroup	4	Binary	Diffie-Hellman group used for PFS for this dynamic tunnel, or 0 if phase 2 PFS is not configured.
56(X'38')	SMF119IS_IPDynIKELclIDType	1	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 include the peers' tunnel endpoint addresses. If client identities were not exchanged during negotiation, this field is 0.

Table 270. IPSec IKE dynamic tunnel specific section (continued)

Offset	Name	Length	Format	Description
57(X'39')	SMF119IS_IPDynIKERmtIDType	1	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 include the peers' tunnel endpoint addresses. If client identities were not exchanged during negotiation, this field is 0.
58(X'3A')	SMF119IS_IPDynIKEExtState	2	Binary	One of the following extended tunnel state information types: • SMF119IS_EXTSASTATE_ACTIVATE (1): This is a new Phase 2 activation. This value is valid only on record subtype 75. • SMF119IS_EXTSASTATE_REFRESH (2): This is a Phase 2 refresh. This value is valid only on record subtype 75. • SMF119IS_EXTSASTATE_DEACT (3): This tunnel is deactivated (not caused by an error or negotiation failure). This value is valid only on record subtype 76. The following values are valid only on record subtype 76: • SMF119IS_EXTSASTATE_PROPOSAL (4): Negotiation failure. No proposal matched the current policy. • SMF119IS_EXTSASTATE_RETRANS (5): Negotiation failure. A retransmit limit was exceeded while negotiating this tunnel. • SMF119IS_EXTSASTATE_POLICY (6): Negotiation failure. A policy mismatch other than a proposal mismatch occured. For example, no valid IpFilterRule. • SMF119IS_EXTSASTATE_OTHER (7): Negotiation failure. The data is not valid in an ISAKMP packet or internal error. • SMF119IS_EXTSASTATE_NOINS (8): A stack error prevented this phase 2 SA from being installed.

Table 271 on page 910 lists the IPSec local client ID specific section.

Table 271. IPSec local client ID specific section

Offset	Name	Length	Format	Description
0(X'0')	SMF119IS_LocalClientID	n	EBCDIC	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).

Table 272 lists the IPSec remote client ID specific section.

Table 272. IPSec remote client ID specific section

Offset	Name	Length	Format	Description
0(X'0')	SMF119IS_RemoteClientID	n	EBCDIC	The remote 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).

IPSec dynamic tunnel deactivation record (subtype 76)

The IPSec dynamic tunnel deactivation record is collected whenever the IKE daemon deactivates a dynamic tunnel. This record contains information about the characteristics of the dynamic tunnel about to be deactivated. If a tunnel is being deactivated as a result of a negotiation failure, some of the fields might be unknown. Fields might be unknown because the negotiation has not progressed far enough to determine a value, such fields have the value 0. If you are using the IPSec NMI, the common IP tunnel section of this SMF record is analogous to the NMsecIPTunnel structure, the dynamic tunnel section is analogous to the NMsecIPDynTunnel structure, the IKE dynamic tunnel section is analogous to the NMsecIPDynamicIKE structure.

Result: When a TCP/IP stack is stopped, dynamic tunnels are not immediately deleted from the IKED. Instead, the IKED waits for the stack to restart so that the stack has the opportunity to send a delete message to the IKE peer. At the time the stack is restarted, you see subtype 76 records for IKED deletion of the dynamic tunnels.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the IPSec dynamic tunnel activation record, the TCP/IP stack identification section indicates IKE as the subcomponent and X'08' (event record) as the record reason.

See Table 268 on page 899 for the contents of the common IP tunnel section.

See Table 269 on page 902 for the contents of the dynamic tunnel section.

See Table 270 on page 908 for the contents of the IKE dynamic tunnel section.

See Table 271 for the contents of the local client ID section.

See Table 272 on page 910 for the contents of the remote client ID section.

Table 273 lists the IPSec dynamic tunnel deactivation record self-defining section.

Table 273. IPSec dynamic tunnel deactivation record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	SMF119_HDR	24	EBCDIC	Standard SMF Header; subtype is 76(X'4C')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (6).
26(X'1A')		2	Binary	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 common IP tunnel section
40(X'28')	SMF119S1Len	2	Binary	Length of common IP tunnel section
42(X'2A')	SMF119S1Num	2	Binary	Number of common IP tunnel sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to dynamic tunnel section
48(X'30')	SMF119S2Len	2	Binary	Length of dynamic tunnel section
50(X'32')	SMF119S2Num	2	Binary	Number of dynamic tunnel sections
52(X'34')	SMF119S3Off	4	Binary	Offset to IKE dynamic tunnel section
56(X'38')	SMF119S3Len	2	Binary	Length of IKE dynamic tunnel sections
58(X'3A')	SMF119S3Num	2	Binary	Number of IKE dynamic tunnel sections
60(X'3C')	SMF119S4Off	4	Binary	Offset to local client ID section
64(X'40')	SMF119S4Len	2	Binary	Length of local client ID section
66(X'42')	SMF119S4Num	2	Binary	Number of local client ID sections
68(X'44')	SMF119S5Off	4	Binary	Offset to remote client ID section
72(X'48')	SMF119S5Len	2	Binary	Length of remote client ID section
74(X'4C')	SMF119S5Num	2	Binary	Number of remote client ID sections

IPSec dynamic tunnel added record (subtype 77)

The IPSec dynamic tunnel added record is collected whenever the TCP/IP stack successfully installs a dynamic tunnel. This record contains information about the characteristics of the dynamic tunnel that was installed. This record uses the NMsecIPTunnel, NMsecIPDynTunnel, and SMF119IS_IPDynamicStackAdded structures. If you are using the IPSec NMI, the common IP tunnel section of this SMF record is analogous to the NMsecIPTunnel structure, the dynamic tunnel section is analogous to the NMsecIPDynTunnel structure. There is not an NMI analog to the SMF119IS_IPDynamicStackAdded structure.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the IPSec dynamic tunnel activation record, the TCP/IP stack identification section indicates `STACK` as the subcomponent and X'08' (event record) as the record reason.

See Table 268 on page 899 for the contents of the common IP tunnel section.

See Table 269 on page 902 for the contents of the dynamic tunnel section.

Table 274 lists the IPSec dynamic tunnel added record self-defining section.

Table 274. IPSec dynamic tunnel added record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	SMF119_HDR	24	EBCDIC	Standard SMF header; subtype is 77(X'4D')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (4).
26(X'1A')		2	Binary	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 common IP tunnel section
40(X'28')	SMF119S1Len	2	Binary	Length of common IP tunnel section
42(X'2A')	SMF119S1Num	2	Binary	Number of common IP tunnel sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to dynamic tunnel section
48(X'30')	SMF119S2Len	2	Binary	Length of dynamic tunnel section
50(X'32')	SMF119S2Num	2	Binary	Number of dynamic tunnel sections
52(X'34')	SMF119S3Off	4	Binary	Offset to stack dynamic tunnel added section
56(X'38')	SMF119S3Len	2	Binary	Length of stack dynamic tunnel added sections
58(X'3A')	SMF119S3Num	2	Binary	Number of stack dynamic tunnel added sections

Table 275 lists the IPSec stack dynamic tunnel added specific section.

Table 275. IPSec stack dynamic tunnel added specific section

Offset	Name	Length	Format	Description
0(X'0')		4	Binary	Stack dynamic tunnel added flags. The following list identifies the bits, their names, and meaning: • X'80000000', SMF119IS_DynStackAddedShadow: SWSA shadow indicator. The tunnel is an SWSA shadow tunnel originating from a distributing stack. • 1 - 31, SMF119IS_IPDnStackAddedRsvd1: Reserved bits.

IPSec dynamic tunnel removed record (subtype 78)

The IPSec dynamic tunnel removed record is collected whenever the TCP/IP Stack removes a dynamic tunnel. This record contains information about the characteristics of the dynamic tunnel that was removed. This record uses the NMsecIPTunnel, NMsecIPDynTunnel, and NMsecIPDynamicStack structures. If you are using the IPSec NMI, the common IP tunnel section of this SMF record is

analogous to the NMsecIPTunnel structure, the dynamic tunnel section is analogous to the NMsecIPDynTunnel structure, and the stack dynamic tunnel section is analogous to the NMsecIPDynamicStack structure.

Result: When a TCP/IP stack is stopped, all dynamic tunnels are removed from the stack, and subtype 78 records are generated at that time.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the IPSec dynamic tunnel removed record, the TCP/IP stack identification section indicates `STACK` as the subcomponent and X'08' (event record) as the record reason.

See Table 268 on page 899 for the contents of the common IP tunnel section.

See Table 269 on page 902 for the contents of the dynamic tunnel section.

Table 276 lists the contents of the IPSec dynamic tunnel removed record self-defining section.

Table 276. IPSec dynamic tunnel removed record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	SMF119_HDR	24	EBCDIC	Standard SMF header; subtype is 78(X'4E')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (4).
26(X'1A')		2	Binary	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 common IP tunnel section
40(X'28')	SMF119S1Len	2	Binary	Length of common IP tunnel section
42(X'2A')	SMF119S1Num	2	Binary	Number of common IP tunnel sections
44(X'2C')	SMF119S2Off	4	Binary	Offset to dynamic tunnel section
48(X'30')	SMF119S2Len	2	Binary	Length of dynamic tunnel section
50(X'32')	SMF119S2Num	2	Binary	Number of dynamic tunnel sections
52(X'34')	SMF119S3Off	4	Binary	Offset to stack dynamic tunnel removed section
56(X'38')	SMF119S3Len	2	Binary	Length of stack dynamic tunnel removed sections
58(X'3A')	SMF119S3Num	2	Binary	Number of stack dynamic tunnel removed sections

Table 277 on page 914 lists the contents of the IPSec dynamic tunnel removed specific section.

Table 277. IPSec dynamic tunnel removed specific section

Offset	Name	Length	Format	Description
0(X'0')		4	Binary	Dynamic tunnel removed flags. The following list identifies the bits, their names, and meaning: • X'8000000', SMF119IS_IPDynStackShadow: SWSA shadow indicator. If set, the tunnel is an SWSA shadow tunnel originating from a distributing stack. • 1 - 31, SMF119IS_IPDynStackRsvd1: Reserved bits.
4(X'4')	SMF119IS_IPDynStackLifesizeCur	8	Binary	Current lifesize value. If the tunnel lifesize value is set, this setting represents the current value of the lifesize counter.
12(X'C')	SMF119IS_IPDynStackOutPkt	8	Binary	Outbound packet count for this tunnel. For SWSA tunnels, this value represents this tunnel's outbound packet count only for this particular TCP/IP stack.
20(X'14')	SMF119IS_IPDynStackInPkt	8	Binary	Inbound packet count for this tunnel. For SWSA tunnels, this value represents this tunnel's inbound packet count only for this particular TCP/IP stack.
28(X'1C')	SMF119IS_IPDynStackOutBytes	8	Binary	Outbound byte count for this tunnel, representing the number of outbound data bytes protected by the tunnel. For SWSA tunnels, this value represents this tunnel's outbound byte count only for this particular TCP/IP stack.
36(X'24')	SMF119IS_IPDynStackInBytes	8	Binary	Inbound byte count for this tunnel, representing the number of inbound data bytes protected by the tunnel. For SWSA tunnels, this value represents this tunnel's inbound byte count only for this particular TCP/IP stack.

IPSec manual tunnel activation record (subtype 79)

The IPSec manual tunnel activation record is collected whenever the TCP/IP Stack installs a new manual tunnel. This record contains information about the characteristics of the manual tunnel. If you are using the IPSec NMI, the common IP tunnel section of this SMF record is analogous to the NMsecIPTunnel structure.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the manual tunnel activation record, the TCP/IP stack identification section indicates `STACK` as the subcomponent and X'08' (event record) as the record reason.

See Table 268 on page 899 for the contents of the common IP tunnel section.

Table 278 lists the contents of the IPSec manual tunnel activation record self-defining section.

Table 278. IPSec manual tunnel activation record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	SMF119_HDR	24	EBCDIC	Standard SMF header; subtype is 79(X'4F')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (2)
26(X'1A')		2	Binary	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 section
36(X'24')	SMF119S1Off	4	Binary	Offset to common IP tunnel section
40(X'28')	SMF119S1Len	2	Binary	Length of common IP tunnel section
42(X'2A')	SMF119S1Num	2	Binary	Number of common IP tunnel sections.

IPSec manual tunnel deactivation record (subtype 80)

The IPSec manual tunnel deactivation record is collected whenever the TCP/IP stack deletes a manual tunnel. This record contains information about the characteristics of the manual tunnel and usage statistics. If you are using the IPSec NMI, the common IP tunnel section of this SMF record is analogous to the NMsecIPTunnel structure, the dynamic tunnel section is analogous to the NMsecIPDynTunnel structure, and the manual tunnel section is analogous to the NMsecIPManualTunnel structure.

See Table 151 on page 748 for the contents of the TCP/IP stack identification section. For the manual tunnel deactivation record, the TCP/IP stack identification section indicates `STACK` as the subcomponent and X'08' (event record) as the record reason.

See Table 268 on page 899 for the contents of the common IP tunnel section.

Table 279 lists the contents of the IPSec manual tunnel deactivation record self-defining section.

Table 279. IPSec manual tunnel deactivation record self-defining section

Offset	Name	Length	Format	Description
0(X'0')	SMF119_HDR	24	EBCDIC	Standard SMF Header; subtype is 80(X'50')
Self-defining section				
24(X'18')	SMF119SD_TRN	2	Binary	Number of triplets in this record (3)
26(X'1A')		2	Binary	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 common IP tunnel section

Table 279. IPSec manual tunnel deactivation record self-defining section (continued)

Offset	Name	Length	Format	Description
40(X'28')	SMF119S1Len	2	Binary	Length of common IP tunnel section
42(X'2A')	SMF119S1Num	2	Binary	Number of common IP tunnel sections.
44(X'2C')	SMF119S2Off	4	Binary	Offset to manual tunnel section
48(X'30')	SMF119S2Len	2	Binary	Length of manual tunnel section
50(X'32')	SMF119S2Num	2	Binary	Number of manual tunnel sections

Table 280 lists the contents of the IPSec manual tunnel specific section.

Table 280. IPSec manual tunnel specific section

Offset	Name	Length	Format	Description
0(X'0')	SMF119IS_IPManTunOutPkt	8	Binary	Outbound packet count for this tunnel
8(X'8')	SMF119IS_IPManTunInPkt	8	Binary	Inbound packet count for this tunnel
16(X'10')	SMF119IS_IPManTunOutBytes	8	Binary	Outbound byte count for this tunnel, representing the number of outbound data bytes protected by the tunnel
24(X'18')	SMF119IS_IPManTunInBytes	8	Binary	Inbound byte count for this tunnel, representing the number of inbound data bytes protected by the tunnel

Appendix F. Application data

Application data is data that is associated with a connection through the use of the SIOCSAPPLDATA ioctl socket command. The SIOCSAPPLDATA IOCTL enables applications to associate 40 bytes of application-specific information with TCP sockets the applications own. This application data can also be used to identify socket endpoints in interfaces such as Netstat, SMF, or network management applications. When the SIOCSAPPLDATA IOCTL is issued, the request argument parameter must contain a SetApplData structure as defined by the EZBYAPPL macro. For more information about the SIOCSAPPLDATA IOCTL, see the miscellaneous programming interfaces "Real-time TCP/IP network monitoring NMI" on page 564 information. In the remainder of this topic, this application-specific data is referred to as ApplData.

Identifying application data

User-defined application data is comprised of 40 bytes of data that is used to identify the endpoint with the macro API sockets application. The application data can be obtained from the following:

Netstat reports
The information is displayed conditionally by using the modifier APPLDATA on the ALLConn/-a and COnn/-c reports, and unconditionally on the ALL/-A report. For more information about the Netstat ALL/-A report, Netstat ALLConn/-a report, and the Netstat COnn/-c report, see *z/OS Communications Server: IP System Administrator's Commands*

SMF 119 TCP connection termination record
For more information about the application data written on the SMF 119 record, see Appendix E, "Type 119 SMF records," on page 743..

Network management interfaces
The following network management interfaces (NMIs) support application data:

- The SYSTCPCN service of the "Real-time TCP/IP network monitoring NMI" on page 564 provides application data in SMF 119 TCP connection termination records.
- The GetTCPListeners and GetConnectionDetail requests of the "TCP/IP callable NMI (EZBNMIFR)" on page 637 provide application data and enable callers to filter on application data.

Guidelines:

- The application is responsible for documenting the content, format, and meaning of the ApplData string that associates it with sockets that 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 EZA or EZB are reserved for IBM use. IBM product identifiers begin with a letter in the range A - I.
- Use printable EBCDIC characters for the entire string to enable searching with Netstat filters.

Tip: Separate application data elements with a blank space for easier reading.

The following z/OS applications support application data registration for their connections:

- The z/OS IP CICS socket interface and listener
- The z/OS TN3270 server application data

CICS socket interface and listener application data

The IP CICS socket interface and listener support automatic registration of application data to be associated with the TCP connection. Automatic registration occurs when the following socket commands are run and the underlying MVS subtask is not detached:

- After CONNECT, connect()
- Before GIVESOCKET, givesocket()

  This function is automatic only for the IBM listener. User-written listeners can issue the SIOCSAPPLDATA IOCTL command with their own application data.
- Before LISTEN, listen()
- After TAKESOCKET, takesocket()

The IP CICS socket interface resource manager task-related user exit (TRUE) processes automatic registration when the resource manager makes an additional SIOCSAPPLDATA IOCTL call. This additional call is made only when the APPLDAT or LAPPLD configuration options are specified as YES. The APPLDAT configuration option is global; all socket-enabled transactions are enabled to automatically register application data against their socket endpoints for the socket commands in the previous list. Regardless of how the APPLDAT on the listener is configured, listeners can optionally be enabled or disabled. The IBM listener also automatically registers application data for accepted connections to be given when the application data being registered contains data about the child process expected to take the given socket. The listener's security exit can also enable or prohibit this action.

Although the application data configuration options can be specified with the EZACICD macro and the EZAC configuration transaction, use the EZAO operator transaction to dynamically alter the same options temporarily. In addition, use the EZAO operator transaction to show the current state of application data registration.

z/OS IP FTP client application data

The z/OS FTP client updates its application data for the following events:

- When a control connection is established between the z/OS FTP client and an FTP server
- When a data connection is established between the z/OS FTP client and an FTP server
- After the **user**, **auth**, or **ccc** subcommand completes successfully

FTP client application data format for the control connection

An FTP control connection is established when an FTP client logs into an FTP server. Table 281 shows the format of the application data set by the client for its control connection socket.

Table 281. FTP client application data format for the control connection

Bytes	Description
1 - 8	The component ID of FTP, EZAFTP0C
9	Blank
10	The FTP connection: **C** Control connection
11	Blank
12 - 20	The user name of the FTP client, padded on the right with blanks. This field might be blank if the user is not logged in to the FTP server.
21	Blank
22	Security protection for the connection: **C** Clear **L** Clear, but previously was Private or Safe **P** Private **S** Safe
23	The security method used for the FTP connection if security protection is either Private or Safe; blank otherwise. **T** TLS managed by AT-TLS **G** GSSAPI **F** TLS managed by FTP
24, 25	The security level if security method is TLS; blank otherwise (see Note). **S2** SSLv2 **S3** SSLv3 **T1** TLSv1 **11** TLSv1.1
26, 27	The security cipher used if the security method is TLS managed by FTP or AT-TLS; blank otherwise (see Note).
28	Blank
29	SOCKS server connection: **D** Direct connection; not through a SOCKS server. **S** Connection through a SOCKS server.
30 - 40	Reserved blank
Note: This value is negotiated during the TLS handshake. Another TLS handshake can occur at any time. The value in this record should be considered a snapshot of the current value at the time the FTP client set application data.	

FTP client application data format for the data connection

An FTP data connection is established just before a file transfer, and is closed after the file transfer is complete. The FTP data connection is the format of the application data set by the client for its data connection socket as described in Table 282.

Table 282. FTP client application data format for the control connection

Bytes	Description
1 - 8	The component ID of FTP, EZAFTP0C
9	Blank
10	The FTP connection: **D** Data connection
11	Blank
12 - 20	The username of the FTP client, padded on the right with blanks. This field might be blank if the user is not logged in to the FTP server.
21	Blank
22	Security protection for the connection: **C** Clear **P** Private **S** Safe
23	The security method used for the FTP connection if the security is either Private or Safe; blank otherwise. **T** TLS managed by AT-TLS **G** GSSAPI **F** TLS managed by FTP
24, 25	The security level if security method is TLS; blank otherwise. **S2** SSLv2 **S3** SSLv3 **T1** TLSv1 **11** TLSv1.1
26, 27	The security cipher used if the security method is TLS; blank otherwise (see Note).
28	Blank
29	Data connection type: **P** Active (PORT) **X** Passive (EPSV) **F** Passive (PASV) **T** Active (EPRT) **N** Active, no command (no default)
30	Data transfer direction: **S** Inbound data transfer to this node. **R** Outbound data transfer from this node.

Table 282. FTP client application data format for the control connection (continued)

Bytes	Description
31	File type: **Q** SQL query function **S** Sequential or partitioned data set
32	File location for FTP client: **P** PDS or PDSE data set **S** MVS, but not a PDS or PSDE **H** z/OS UNIX file **-** `*DEV.NULL (NULL directory)`, or client is receiving a directory listing
33	Blank
34	SOCKS connection: **D** Direct connection to FTP server (SOCKS is not in use). **S** Connection through a SOCKS server.
35 - 40	Reserved blank
Note: This value is negotiated during the TLS handshake. Another TLS handshake can occur at any time. The value in this record should be considered a snapshot of the current value at the time the FTP client set application data.	

FTP daemon application data format

The FTP daemon opens a socket to accept connections from FTP clients. Table 283 shows the format of the application data set by the FTP daemon for its listening socket.

Table 283. FTP daemon application data format

Bytes	Description
1 - 8	The component ID of FTP, EZAFTP0D
9	Blank
10	TLSPORT flag: **T** FTP listening port is the TLSPORT

FTP server application data format for the control connection

The FTP server control connection is established when the FTP daemon accepts an incoming connection on its listening socket (the connection is passed from the daemon to the server). Table 284 shows the format of the application data set by the FTP server for its control connection socket.

Table 284. FTP server application data format for the control connection

Bytes	Description
1 - 8	The component ID of FTP, EZAFTP0S
9	Blank

Table 284. FTP server application data format for the control connection (continued)

Bytes	Description
10	The FTP connection: **C** Control connection
11	Blank
12 - 20	The user name used to log into FTP, padded to the right with blanks. This field might be blank if the user is not logged into the FTP server.
21	Blank
22	Security protection for the connection: **C** Clear **L** Clear, but previously was Private or Safe **P** Private **S** Safe
23	The security method used for the FTP connection if security protection is either Private or Safe; Bank otherwise. **T** TLS managed by AT-TLS **G** GSSAPI **F** TLS managed by FTP
24, 25	The security level if security method is TLS and the handshake has completed; blank otherwise (see Note). **S2** SSLv2 **S3** SSLv3 **T1** TLSv1 **11** TLSv1.1
26,27	The security cipher used if the security method is TLS and the handshake has completed; blank otherwise (see Note).
28	Blank
29 - 40	Reserved blank
Note: This value is negotiated during the TLS handshake. Another TLS handshake can occur at any time. When the FTP server next updates the APPLDATA, this value might change.	

FTP server application data format for the data connection

The FTP server establishes a data connection just before a file transfer occurs. The connection is closed when the file transfer is complete. Table 285 shows the format of the application data set by the server for its data connection.

Table 285. FTP server application data for the data connection

Bytes	Description
1 - 8	The component ID of FTP, EZAFTP0S
9	Blank
10	The FTP connection: **D** Data connection

Table 285. FTP server application data for the data connection (continued)

Bytes	Description
11	Blank
12 - 20	The user name of the FTP client, padded to the right with blanks. This field might be blank if the user is not logged into the FTP server.
21	Blank
22	Security protection for the connection: **C** Clear **P** Private **S** Safe
23	The security method used for the FTP connection if security protection is either Private or Safe; Blank otherwise. **T** TLS managed by AT-TLS **G** GSSAPI **F** TLS managed by FTP
24, 25	The security level if security method is TLS and the handshake has completed; blank otherwise (see Note): **S2** SSLv2 **S3** SSLv3 **T1** TLSv1 **11** TLSv1.1
26,27	The security cipher used if the security method is TLS or AT-TLS and the handshake has completed; blank otherwise (see Note).
28	Blank
29	Data connection type: **P** Active (PORT) **X** Passive (EPSV) **F** Passive (PASV) **T** Active (EPRT) **N** Active, no command (this is the default)
30	Data transfer direction: **S** Inbound data transfer to this node **R** Outbound data transfer from this node
31	File type: **D** Directory as the result of a LIST or NLST command **J** JES file **Q** SQL uery function **S** Sequential or partitioned data set

Table 285. FTP server application data for the data connection (continued)

Bytes	Description
32	File location: **P** PDS or PDSE data set **S** MVS but not a PDS or PSDE **H** UNIX file **-** *DEV.NULL (null directory)
33 - 40	Reserved blank
Note: This value is negotiated during the TLS handshake. Another TLS handshake can occur at any time. When the FTP server next updates the APPLDATA, this value might change.	

Application data format for IP CICS sockets

When application data registration is enabled, the IP CICS socket TRUE and listener uses the following application data formats.

CONNECT

The application data registered against a connecting socket is comprised of the elements in Table 286.

Table 286. Registered application data - CONNECT

Bytes	Description
1-8	The component ID of the IP CICS socket interface. For an outbound IP CICS socket client, this data always comprises the characters EZACICSO.
9	Blank
10-13	The CICS/TS transaction identifier. This is the CICS/TS transaction ID that is assigned to the program that issued the CONNECT socket command.
14	Blank
15-21	The task number of the transaction identifier in bytes 10-13.
22	Blank
23-30	The user ID that is assigned to the transaction identifier in bytes 10-13.
31	Blank
32-35	The CICS system name where the transaction is running.
36-40	Blank

This data is registered when a client is connected. The following are examples of the application data that is registered for a client's connected socket. The following example shows the application data registered for a client's connected socket:

```
Col
1.......10...15......23.......32......40
EZACICSO CLI1 0000059 CICSUSR5 CICP
```

Following is an example for application data:

```
EZACICSO CLI1 0000059 CICSUSR5 CICP
```

GIVESOCKET

The application data registered against a socket given to another process by the IBM listener is comprised of elements that are used to identify the GIVESOCKET endpoint. For the IP CICS sockets listener, the elements in Table 287 are used.

Table 287. Registered application data - GIVESOCKET

Bytes	Description
1-8	The component ID of the IP CICS Socket listener. For the IP CICS Sockets listener, this data always comprises the characters EZACIC02.
9	Blank
10-13	The CICS/TS transaction identifier. This is the transaction ID that the listener starts that the listener expects to take the specified socket.
14	Blank
15-21	This data is the task number of the CICS task that gives the accepted socket to a child process.
22	Blank
23-30	The user ID to be assigned to the transaction identifier in bytes 10-13.
31	Blank
32-35	The CICS system name where the transaction is to be assigned.
36-40	Blank

This data is registered for every accepted connection that can be processed by the listeners optional user exit or security exit. The following example shows the application data registered for an accepted connection to be given to a child process:

```
EZACIC02 SRV1 0000021 CICSUSR2 CIC3
```

LISTEN

The application data registered against a passive or listener socket is comprised of the elements shown in Table 288.

Table 288. Registered application data - LISTEN

Bytes	Description
1-8	The component ID of the IP CICS socket interface. For the IP CICS sockets listener, this data always comprises the characters EZACICSO.
9	Blank
10-13	The CICS/TS transaction identifier. This is the CICS/TS transaction ID assigned to the EZACIC02 program or a user-designed listener transaction program.

Table 288. Registered application data - LISTEN (continued)

Bytes	Description
14	Blank
15-21	The task number of the transaction identifier.
22	Blank
23-30	The user ID that is assigned to the transaction identifier in bytes 10-13.
31	Blank
32-35	The CICS system name where the transaction is executing.
36-40	Blank

This data is registered before the listener's listen queues are established so that all connecting sockets inherit the application data. Following are examples of the application data registered for a listener's passive socket. The following example shows the application data registered for a listener's passive socket:

```
EZACICSO CSKL 0000021 CICSUSR1 CICP
```

For application data:

```
EZACICSO CSKL 0000021 CICSUSR1 CICP
```

TAKESOCKET

The application data registered against a socket taken by a child server transaction is comprised of the elements in Table 289.

Table 289. TAKESOCKET

Bytes	Description
1-8	The component ID of the IP CICS Socket interface. For the IP CICS Sockets interface and listener, this data always comprises the characters EZACICSO.
9	Blank
10-13	The CICS/TS transaction identifier. This is the transaction ID that now owns the socket.
14	Blank
15-21	The task number of the transaction identifier in bytes 10-13.
22	Blank
23-30	The user ID that is assigned to the transaction identifier in bytes 10-13.
31	Blank
32-35	The CICS system name where the transaction is running.
36-40	Blank

This data is registered for every socket successfully taken by a child server CICS task. The following are examples of the application data registered for a socket

taken by a child server. The following example shows the application data registered for a socket taken by a child server:

```
EZACICSO SRV1 0000022 CICSUSR2 CIC3
```

For application data:

```
EZACICSO SRV1 0000022 CICSUSR2 CIC3
```

Application data processing

When the IP CICS Socket interface or listener is configured to register application data, the processing shown in Table 290 occurs.

Table 290. Application data processing

APPLDAT value	LAPPLD value (See Note 1)	Security or User exit input (inherited)	Security or User exit output (See Note 2)	Processing
Yes	YES, INHERIT, or unspecified (YES) (See Note 3)	1	1	All socket-enabled transaction programs including specific listeners. Specific accepted connection to be given are registered for the IBM listener.
			0	All socket-enabled transaction programs including specific listeners. But, specific accepted connection to be given is not registered for the IBM listener.
	NO (See Note 4)	0	1	All socket-enabled transaction program excluding specific listeners. But, specific accepted connection to be given is registered for the IBM listener.
			0	All socket-enabled transaction program excluding specific listeners. Specific accepted connection to be given are not registered for the IBM listener.

Table 290. Application data processing (continued)

APPLDAT value	LAPPLD value (See Note 1)	Security or User exit input (inherited)	Security or User exit output (See Note 2)	Processing
NO or unspecified (NO)	YES (See Note 4)	1	1	Only the specific listeners. Specific accepted connection to be given are registered for the IBM listener.
			0	Only the specific listeners. But specific accepted connection to be given are not registered for the IBM listener.
	NO or INHERIT or unspecified (NO) (See Note 3)	0	1	Neither socket enabled transaction program nor specific listeners. But specific accepted connection to be given is registered for the IBM listener.
			0	Neither socket enabled transaction program nor specific listeners. Specific accepted connection to be given is not registered for the IBM listener.

Notes:

1. LAPPLD inherits the value specified by the APPLDAT setting when the LAPPLD parameter is not specified.
2. Reference is made upon the setting made upon return from the IBM listener's security/user exit.
3. When the LAPPLD value is not specified, its value is inherited from the value specified by the listener's interface APPLDAT setting.
4. When the LAPPLD value is different from that specified by the APPLDAT value, the LAPPLD value is used.

The LAPPLD setting is not inherited from APPLDAT; the LAPPLD setting supersedes the APPLDAT value. The security exit byte is inherited from either the APPLDAT or LAPPLD setting. The security exit is then used to change the action taken by the listener when registering application data for the accepted connection.

Application data format for CSSMTP

Forty bytes of application data are available for Netstat reports, SMF119 TCP connection termination reports, or network management interface (NMI) applications. Table 291 on page 929 and Table 292 on page 929 shows the application data format used by CSSMTP. This data is written at the beginning of each successful target server connection.

Table 291. Connections transferring message data

Bytes	Description
1-8	The component ID of the CSSMTP application, EZASMTPC.
9	Blank
10-17	The external writer name used to queue the JES spool files to CSSMTP
18	Blank
19	The type of server connection, otherwise blank if undetermined: S -- SMTP server E -- ESMTP server
20	The security method used for the TCP/IP connection, otherwise blank if undetermined: B - Basic (no security) T - TTLS port managed by AT-TLS
21-22	The security level, otherwise blank if undetermined: T1 - TLSv1 11 - TLSv1.1 S3 - SSLv3 S2 - SSLv2
23-24	The security cipher used, otherwise blank if undetermined.
25	Blank
26-40	Reserved

The following shows an example of the application data after a CSSMTP session has ended.

```
0000000001111111111222222222233333333334
1234567890123456789012345678901234567890
EZASMTPC XYZ      ETT101
```

Table 292. Connections monitoring target servers

Bytes	Description
1-8	The component ID of the CSSMTP application, EZASMTPM.
9	Blank
10-17	The external writer name used to queue the JES spool files to CSSMTP
18	Blank

Table 292. Connections monitoring target servers (continued)

Bytes	Description
19	The type of server connection, otherwise blank if undetermined: S -- SMTP server E -- ESMTP server
20	The security method used for the TCP/IP connection, otherwise blank if undetermined: B - Basic (no security) T - TTLS port managed by AT-TLS
21-22	The security level, otherwise blank if undetermined: T1 - TLSv1 11 - TLSv1.1 S3 - SSLv3 S2 - SSLv2
23-24	The security cipher used, otherwise blank if undetermined.
25	Blank
26-40	Reserved

The following shows an example of the application data after a CSSMTP session has ended.

```
0000000001111111111222222222233333333334
1234567890123456789012345678901234567890
EZASMTPM XYZ      ETT101
```

TN3270E Telnet server application data

The TN3270E Telnet server (Telnet) updates the application data to be applied to the TCP connection when the following events occur:

- When Telnet connection negotiations are complete
- When a SNA session has been established
- When a SNA session has ended

Telnet performs the updates by issuing the SIOCSAPPLDATA IOCTL calls when these events occur.

Application data format for Telnet

The 40 bytes of application data is available for Netstat reports, SMF 119 TCP connection termination reports, or network management interface (NMI) applications. Table 293 on page 931 shows the application data format used by Telnet.

Table 293. Application data format used by Telnet

Bytes	Description
1-8	The component ID of the TN3270E Telnet server, EZBTNSRV.
9	Blank
10-17	The LU name used to represent the client. This can be blank for non-TN3270E connections that do not have a SNA session.
18	Blank
19-26	The SNA application name. This data is present when a SNA session has been established.
27	Blank
28	The Telnet connection mode: • E - TN3270E • 3 - TN3270 • L - Linemode • D - DBCS transform
29	The Client type: • T- Terminal • P - Printer
30	Blank
31	The security method used for the TCP/IP connection: • B - Basic (no security) • S - Secureport managed by Telnet • T - TTLSport managed by AT-TLS
32-33	The security level: • 11 - TLSv1.1 • T1 - TLSv1 • S3 - SSLv3 • S2 - SSLv2
34-35	The security cipher used.
36	Blank
37-40	Reserved blank

The following shows an example of the application data after a SNA session is established.

```
0000000001111111111222222222233333333334
1234567890123456789012345678901234567890
EZBTNSRV TCPM1001 TSO10005 ET TT105
```

Appendix G. 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 197.

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 "X Window System routines: z/OS UNIX System Services support" on page 986.

Software requirements for X Window System interface V11R4 and Motif version 1.1

Application programs using the X Window System API are written in C and should be compiled, linked, and run 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 935 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.

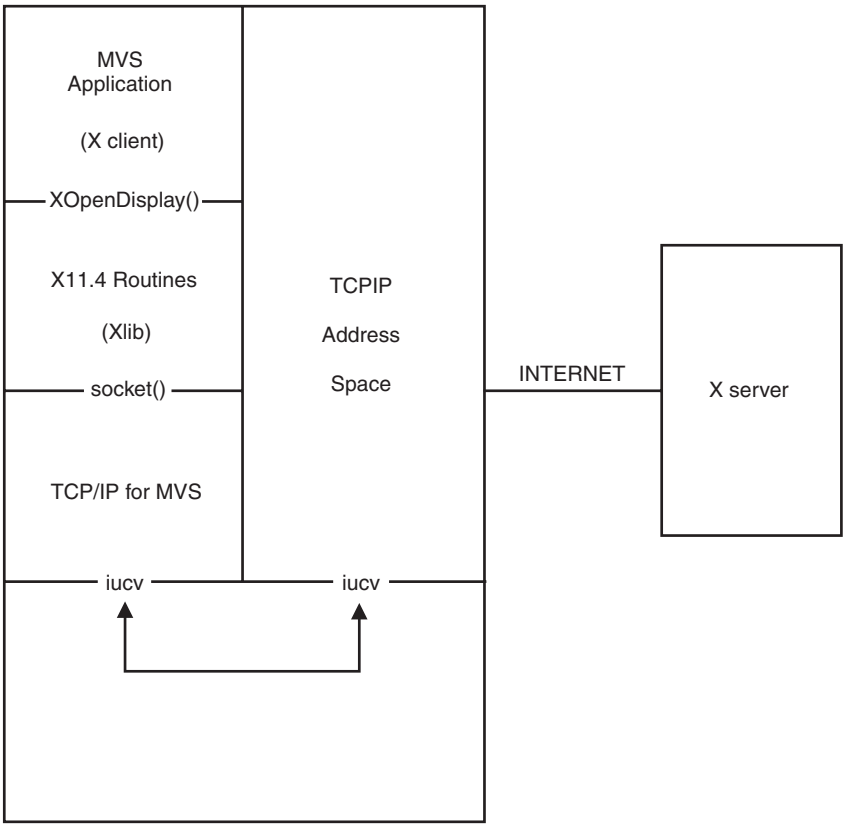


Figure 35. MVS X Window System application to server

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.

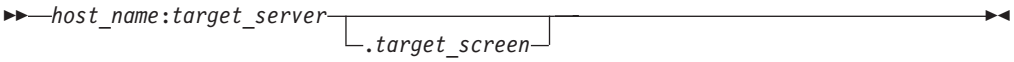
In the MVS environment, external names must be eight characters or less. Many of the X Window System application programming interface names exceed this limit. To support the X API in MVS, all X names longer than eight characters are remapped to unique names using the C compiler preprocessor. This name 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.

X Window System interface in the MVS environment: 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:



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 987.

X Window System interface in the MVS environment: 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 "X Window System routines: z/OS UNIX System Services support" on page 986.

Figure 36 on page 937 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 xrdb program to control the X server resource database in the resource property.

xrdb 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.

X Window System interface in the MVS environment: 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 943) 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 topic 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 Intrinsics header files

The following is a list of X Window System and Xt Intrinsics headers:

```
ap@keysy.h
Atoms.h
Callback.h
CharSet.h
CloseHoo.h
ComposiI.h
ComposiP.h
Composit.h
Constrai.h
ConstraP.h
Converte.h
ConvertI.h
copyrigh.h
Core.h
CoreP.h
cursorfo.h
CurUtil.h
CvtCache.h
DECkeysy.h
DisplayQ.h
Drawing.h
Error.h
EventI.h
extutil.h
fd.h
InitialI.h
Initer.h
IntriniI.h
IntriniP.h
Intrinsi.h
keysym.h
keysymde.h
ks@names.h
Misc.h
MITMisc.h
mitmiscs.h
multibst.h
multibuf.h
Object.h
ObjectP.h
PassivGr.h
poly.h
Quarks.h
RectObj.h
RectObjP.h
region.h
Resource.h
Selectio.h
shape.h
shapestr.h
Shell.h
ShellP.h
StdCmap.h
StdSel.h
StringDe.h
SysUtil.h
Translat.h
VarargsI.h
Vendor.h
VendorP.h
WinUtil.h
X.h
Xatom.h
Xatomtyp.h
Xauth.h
Xct.h
Xext.h
Xkeymap.h
Xlib.h
Xlibint.h
Xlibos.h
Xllglue.h
Xmd.h
Xmu.h
Xos.h
Xproto.h
Xprotost.h
Xresourc.h
Xt@remap.h
Xtos.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
ACommanP.h
AForm.h
AFormP.h
ALabel.h
ALabelP.h
AList.h
AListP.h
AScrollb.h
AScrollP.h
AText.h
ATextP.h
ATextSrP.h
AsciiSin.h
AscSinkP.h
AsciiSrc.h
AscSrcP.h
AsciiTex.h
AscTextP.h
Box.h
BoxP.h
Cardinal.h
Clock.h
ClockP.h
CommandI.h
Dialog.h
DialogP.h
Grip.h
GripP.h
Logo.h
LogoP.h
Mailbox.h
MailboxP.h
MenuButP.h
MenuButt.h
Paned.h
PanedP.h
Scroll.h
Simple.h
SimpleMP.h
SimpleMe.h
SimpleP.h
Sme.h
SmeBSB.h
SmeBSBP.h
SmeLine.h
SmeLineP.h
SmeP.h
StripChP.h
StripCha.h
Template.h
TemplatP.h
TextSink.h
TextSinP.h
TextSrc.h
Toggle.h
ToggleP.h
VPaned.h
Viewport.h
ViewporP.h
XawInit.h
```

Motif header files

The following is a list of headers for the Motif-based widget set:

```
ArrowB.h
ArrowBG.h
ArrowBGP.h
ArrowBP.h
bitmaps.h
BulletBP.h
Bulletin.h
CascaBGP.h
CascadBG.h
CascaBGP.h
CascadBP.h
CascadeB.h
Command.h
CommandP.h
CutPaste.h
CutPastP.h
DialogS.h
DialogSP.h
DrawingA.h
DrawinAP.h
DrawnB.h
DrawnBP.h
FileSB.h
FileSBP.h
Form.h
FormP.h
Frame.h
FrameP.h
Label.h
LabelG.h
LabelGP.h
LabelP.h
List.h
ListP.h
MainW.h
MainWP.h
MenuShel.h
MenuShep.h
MessagBP.h
MessageB.h
PanedW.h
PanedWP.h
PushB.h
PushBG.h
PushBGP.h
PushBP.h
RowColum.h
RowColuP.h
SashP.h
Scale.h
ScaleP.h
ScrollBa.h
ScrollBP.h
Scrolled.h
ScrollWP.h
SelectBP.h
SelectiB.h
SeparaGP.h
SeparatG.h
Separato.h
SeparatP.h
StringSr.h
Text.h
TextInP.h
TextOutP.h
TextP.h
TextSrcP.h
TogglBGP.h
ToggleB.h
ToggleBG.h
ToggleBP.h
Xm.h
XmP.h
```

X Window System interface in the MVS environment: Compiling and linking

You can use several methods to compile, link-edit, and execute your program in MVS. This topic 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
user_id.**MYPROG.C**	Contains user C source programs.
user_id.**MYPROG.C(PROGRAM1)**	Member PROGRAM1 in *user_id*.MYPROG.C partitioned data set.
user_id.**MYPROG.H**	Contains user #include files.
user_id.**MYPROG.OBJ**	Contains object code for the compiled versions of user C programs in *user_id*.MYPROG.C.
user_id.**MYPROG.LOAD**	Contains link-edited versions of user programs in *user_id*.MYPROG.OBJ.

X Window System interface in the MVS environment: Nonreentrant modules

The following lines describe the additions that you must make to the compilation 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(IBMCPP) 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)'
   ```

X Window System interface in the MVS environment: Reentrant modules

The following lines describe the additions that you must make to the compilation 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(IBMCPP) preprocessor symbol. See "X Window System interface in the MVS environment: Compiling and linking" on page 939 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)
//SYSMSGS   DD DSN=C370.V2R1M0.SEDCMSGS(EDCMSGE),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;
//SYSLMOD   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 *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 *z/OS XL C/C++ User's Guide*.

Using sample X Window System programs

This topic contains information about the sample X programs that are 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

For information about running a sample program, see "X Window System interface in the MVS environment: Compiling and linking" on page 939 and "Compiling and linking with z/OS UNIX System Services" on page 987.

X Window System Interface V11r4: Environment variables

Table 294 provides a list of environment variables that can be explicitly set by X Window System Interface V11r4.

Table 294. 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.
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.

Table 294. Environment variables for X Window System Interface V11r4 (continued)

Environment variable	Description
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 "X Window System routines: z/OS UNIX System Services support" on page 986.

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 295. 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.
XDPYINFO C	1. Compile the XDPYINFO C source program. 2. Link-edit the XDPYINFO 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.

Table 295. Building X client modules based on X11 functions (continued)

To build module	Do the following
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.
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 296. 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.

Table 296. Building X client modules based on Xt Intrinsics and Athena Toolkit functions (continued)

To build module	Do the following
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.

X Window System routines: Opening and closing a display

Table 297 provides the routines for opening and closing a display.

Table 297. 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.

X Window System routines: Creating and destroying windows

Table 298 provides the routines for creating and destroying windows.

Table 298. Creating and destroying windows

Routine	Description
XConfigureWindow()	Configures the specified window.
XCreateSimpleWindow()	Creates unmapped InputOutput subwindow.

Table 298. Creating and destroying windows (continued)

Routine	Description
XCreateWindow()	Creates unmapped subwindow.
XDestroySubwindows()	Destroys all subwindows of specified window.
XDestroyWindow()	Unmaps and destroys window and all subwindows.

X Window System routines: Manipulating windows

Table 299 provides the routines for manipulating windows.

Table 299. 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.
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.

X Window System routines: Changing window attributes

Table 300 provides the routines for changing window attributes.

Table 300. 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.

Table 300. Changing window attributes (continued)

Routine	Description
XTranslateCoordinates()	Transforms coordinates between windows.

X Window System routines: Obtaining window information

Table 301 provides the routines for obtaining window information.

Table 301. 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.

X Window System routines: Obtaining properties and atoms

Table 302 provides the routines for obtaining properties and atoms.

Table 302. Properties and atoms

Routine	Description
XGetAtomName()	Gets a name for the specified atom ID.
XInternAtom()	Gets an atom for the specified name.

X Window System routines: Manipulating window properties

Table 303 provides the routines for manipulating the properties of windows.

Table 303. 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.

X Window System routines: Setting window selections

Table 304 provides the routines for setting window selections.

Table 304. Setting window selections

Routine	Description
XConvertSelection()	Converts a selection.
XGetSelectionOwner()	Gets the selection owner.
XSetSelectionOwner()	Sets the selection owner.

X Window System routines: Manipulating colormaps

Table 305 provides the routines for manipulating color maps.

Table 305. 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.

X Window System routines: Manipulating color cells

Table 306 provides the routines for manipulating color cells.

Table 306. 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.
XStoreNamedColor()	Sets a pixel color to the named color.

X Window System routines: Creating and freeing pixmaps

Table 307 provides the routines for creating and freeing pixmaps.

Table 307. Creating and freeing pixmaps

Routine	Description
XCreatePixmap()	Creates a pixmap of a specified size.
XFreePixmap()	Frees all storage associated with specified pixmap.

X Window System routines: Manipulating graphics contexts

Table 308 provides the routines for manipulating graphics contexts.

Table 308. 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.

Table 308. Manipulating graphics contexts (continued)

Routine	Description
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.
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.
XSetTSOrigin()	Sets the tile or stipple origin of the specified GC.

X Window System routines: Clearing and copying areas

Table 309 provides the routines for clearing and copying areas.

Table 309. 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.

X Window System routines: Drawing lines

Table 310 provides the routines for drawing lines.

Table 310. Drawing lines

Routine	Description
XDraw()	Draws an arbitrary polygon or curve that is defined by the specified list of Vertexes as specified in *vlist*.
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.

X Window System routines: Filling areas

Table 311 provides the routines for filling areas.

Table 311. Filling areas

Routine	Description
XFillArc()	Fills single arc in drawable.
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.

X Window System routines: Loading and freeing fonts

Table 312 provides the routines for loading and freeing fonts.

Table 312. 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.

Table 312. Loading and freeing fonts (continued)

Routine	Description
XQueryFont()	Gets information about a loaded font.
XSetFontPath()	Sets the font search path.
XUnloadFont()	Unloads the specified font.

X Window System routines: Querying character string sizes

Table 313 provides the routines for querying the character size of a string.

Table 313. 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.
XTextWidth16()	Gets the width of a 2-byte character string.

X Window System routines: Drawing text

Table 314 provides the routines for drawing text.

Table 314. 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.

X Window System routines: Transferring images

Table 315 provides the routines for transferring images.

Table 315. Transferring images

Routine	Description
XGetImage()	Gets the image from the rectangle in the drawable.

Table 315. Transferring images (continued)

Routine	Description
XGetSubImage()	Copies the rectangle on the display to the image.
XPutImage()	Puts the image from memory into the rectangle in the drawable.

X Window System routines: Manipulating cursors

Table 316 provides the routines for manipulating cursors.

Table 316. 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.

X Window System routines: Handling window manager functions

Table 317 provides the routines for handling the window manager functions.

Table 317. 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.

Table 317. Handling window manager functions (continued)

Routine	Description
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.

X Window System routines: Manipulating keyboard settings

Table 318 provides the routines for manipulating keyboard settings.

Table 318. 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.
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.

X Window System routines: Controlling the screen saver

Table 319 provides the routines for controlling the screen saver.

Table 319. 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.

Table 319. Controlling the screen saver (continued)

Routine	Description
XSetScreenSaver()	Sets the screen saver.

X Window System routines: Manipulating hosts and access control

Table 320 provides the routines for manipulating hosts and toggling the access control.

Table 320. 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.

X Window System routines: Handling events

Table 321 provides the routines for handling events.

Table 321. 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.
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.

Table 321. Handling events (continued)

Routine	Description
XWindowEvent()	Removes the next event that matches the specified window and mask.

X Window System routines: Enabling and disabling synchronization

Table 322 provides the routines for toggling synchronization.

Table 322. Enabling and disabling synchronization

Routine	Description
XSetAfterFunction()	Sets the previous after function.
XSynchronize()	Enables or disables synchronization.

X Window System routines: Using default error handling

Table 323 provides the routines for using the default error handling.

Table 323. 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.

X Window System routines: Communicating with window managers

Table 324 provides the routines for communicating with window managers.

Table 324. 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.

Table 324. Communicating with window managers (continued)

Routine	Description
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.
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.

X Window System routines: Manipulating keyboard event functions

Table 325 on page 961 provides the routines for manipulating keyboard event functions.

Table 325. 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.

X Window System routines: Manipulating regions

Table 326 provides the routines for manipulating regions.

Table 326. 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.
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.

X Window System routines: Using cut and paste buffers

Table 327 provides the routines for using cut and paste buffers.

Table 327. 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.

X Window System routines: Querying visual types

Table 328 provides the routines for querying visual types.

Table 328. 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.

X Window System routines: Manipulating images

Table 329 provides the routines for manipulating images.

Table 329. 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.

X Window System routines: Manipulating bit maps

Table 330 provides the routines for manipulating bit maps.

Table 330. 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.

Table 330. Manipulating bit maps (continued)

Routine	Description
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.

X Window System routines: Using the resource manager

Table 331 provides the routines for using the resource manager.

Table 331. 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.
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.

X Window System routines: Manipulating display functions

Table 332 provides the routines for manipulating display functions.

Table 332. Manipulating display functions

Routine	Description
AllPlanes() XAllPlanes()	Returns all bits suitable for use in plane argument.

Table 332. Manipulating display functions (continued)

Routine	Description
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.
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.

Table 332. Manipulating display functions (continued)

Routine	Description
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.
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.

X Window System routines: 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 333 lists the X Window System Extension Routines and provides a short description of each routine.

Table 333. Extension routines

Routine	Description
XAllocID()	Returns a resource ID that can be used when creating new resources.
XESetCloseDisplay()	Defines a procedure to call when XCloseDisplay is called.
XESetCopyGC()	Defines a procedure to call when a GC is copied.
XESetCreateFont()	Defines a procedure to call when XLoadQueryFont is called.
XESetCreateGC()	Defines a procedure to call when a new GC is created.
XESetError()	Suppresses the call to an external error handling routine and defines an alternative routine for error handling.
XESetErrorString()	Defines a procedure to call when an I/O error is detected.
XESetEventToWire()	Defines a procedure to call when an event must be converted from the host to wire format.
XESetFreeFont()	Defines a procedure to call when XFreeFont is called.
XESetFreeGC()	Defines a procedure to call when a GC is freed.
XESetWireToEvent()	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.

X Window System routines: 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 334 lists the routines that allow an application to use these extensions.

Table 334. 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.

Table 334. MIT extensions to X (continued)

Routine	Description
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.

X Window System routines: 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 335 lists the Associate table functions and provides a short description of each function.

Table 335. 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.

X Window System routines: 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 336 lists the Miscellaneous utility routines and provides a short description of each routine.

Table 336. 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 *item* from the Compound Text string.
XctReset()	Resets the XctData structure to reparse the Compound Text string.
XmuAddCloseDisplayHook()	Adds a callback for the given display.
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.

Table 336. Miscellaneous utility routines (continued)

Routine	Description
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 *official* X Window System logo.
XmuDrawRoundedRectangle()	Draws a rounded rectangle.
XmuFillRoundedRectangle()	Draws a filled rounded rectangle.
XmuGetAtomName()	Returns the name of an Atom.
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.

Table 336. Miscellaneous utility routines (continued)

Routine	Description
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.
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 Window System routines: 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 337 lists the X authorization routines and provides a short description of each routine.

Table 337. 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.

Table 337. Authorization routines (continued)

Routine	Description
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 338.

Table 338. 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
PassivGraI.h	PassivGr.h
ProtocolsP.h	ProtocoP.h
SelectionI.h	SelectiI.h
WindowObjP.h	WindowOP.h

Xt Intrinsics routines

Table 339 provides the Xt Intrinsics routines and a short description of each routine.

Table 339. Xt Intrinsics 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.

Table 339. Xt Intrinsics routines (continued)

Routine	Description
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.
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.

Table 339. Xt Intrinsics routines (continued)

Routine	Description
XtAppSetFallbackResources	Sets the fallback resource list that will be loaded at display initialization time.
XtAppSetSelectionTimeout	Sets the Intrinsics 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.
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.

Table 339. Xt Intrinsics routines (continued)

Routine	Description
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.
XtDisownSelection	Informs the Intrinsics 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.

Table 339. Xt Intrinsics routines (continued)

Routine	Description
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 multiclick 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.
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.

Table 339. Xt Intrinsics routines (continued)

Routine	Description
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.
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.

Table 339. Xt Intrinsics routines (continued)

Routine	Description
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.
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.

Table 339. Xt Intrinsics routines (continued)

Routine	Description
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.
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.

Table 339. Xt Intrinsics routines (continued)

Routine	Description
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.
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.

X Window System toolkit: 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.

X Window System routines: 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
AsciiSrc
AsciiText
Box
Clock
Command
Dialog
Form
Grip
Label
List
Logo
Mailbox
MenuButton
Paned
Scrollbar
Simple
SimpleMenu
Sme (Simple Menu Entry)
SmeBSB (BSB Menu Entry)
SmeLine
StripChart
Text
TextSink
TextSrc
Toggle
VPaned
Viewport

Table 340 provides the Athena widget routines with a short description of each routine.

Table 340. 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.
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.

Table 340. Athena widget routines (continued)

Routine	Description
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.
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.

Table 340. Athena widget routines (continued)

Routine	Description
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 341. 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 341. 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
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

X Window System routines: 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
BulletinBoard
CascadeButton and CascadeButtonGadget
Command
DialogShell
DrawingArea
DrawnButton
Form
Frame
Label and LabelGadget
List
MainWindow
MenuShell
MessageBox
PanedWindow
PushButton and PushButtonGadget
RowColumn
Sash
Scale
ScrollBar
ScrolledWindow
SelectionBox and SelectionDialog
Separator and SeparatorGadget
Text
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 342. 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 342. 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	MessagBP.h

Table 342. Motif header file names (continued)

Motif distribution name	TCP/IP name
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
TraversalI.h	TraversI.h
VirtKeysP.h	VirtKeyP.h

X Window System routines: z/OS UNIX System Services support

The following topics 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 *z/OS XL C/C++ Run-Time Library Reference*.

X Window System routines: 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 *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.

X Window System routines: 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.

X Window System routines: 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 *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 compilation step, make the following changes:

- Change the CPARM parameters to specify one of the following:
  - `CPARM='DEF(IBMCPP),RENT,LO'`
  - `CPARM='DEF(IBMCPP,_POSIX1_ SOURCE=1),RENT,LO'`
  - `CPARM='DEF(IBMCPP,_OPEN_ SYS),RENT,LO'`
  - `CPARM='DEF(IBMCPP,_OPEN_ SOCKETS,_POSIX1_SOURCE=1),RENT,LO'`
  - `CPARM='DEF(IBMCPP,_OPEN_ SOCKETS,_OPEN_SYS),RENT,LO'`

  Note: The recommended CPARMS are:

  ```
  CPARM='DEF(IBMCPP,_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, IBMCPP.

  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(IBMCPP) preprocessor symbol. See "X Window System interface in the MVS environment: Compiling and linking" on page 939 for information about compiling and linking your program in MVS.

  For a complete discussion of all of the AD/Cycle C/370 parameters, see *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.SCEEOBJ:

  ```
  // 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.SCEEOBJ:

```
// DD DSN=SEZAROE2,DISP=SHR
// DD DSN=SEZAROE1,DISP=SHR
```

- To link-edit programs that use theMotif Toolkit functions, add the following lines after the prelink //SYSLIB DD statement for the IBM AD/Cycle C/370 library cee.v1r3m0.SCEEOBJ:

  ```
  // 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 *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:

     ```
        -l"//'
     sys1.SFOMHDRS'"
        -l"//'SEZACMAC'"

        -l"//'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"//'SEZAROE2'"
  ```

- If the Motif Toolkit functions are required, then also specify:

  ```
  -l"//'SEZAROE3'"
  ```

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 945.

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 H. 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 1007 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	*Telnet output carriage-return disposition option* D. Crocker
RFC 653	*Telnet output horizontal tabstops option* D. Crocker
RFC 654	*Telnet output horizontal tab disposition option* D. Crocker
RFC 655	*Telnet output formfeed disposition option* D. Crocker
RFC 657	*Telnet output vertical tab disposition option* D. Crocker
RFC 658	*Telnet output linefeed disposition* D. Crocker
RFC 698	*Telnet extended ASCII option* T. Mock
RFC 726	*Remote Controlled Transmission and Echoing Telnet option* J. Postel, D. Crocker
RFC 727	*Telnet logout option* M.R. Crispin
RFC 732	*Telnet Data Entry Terminal option* J.D. Day
RFC 733	*Standard for the format of ARPA network text messages* D. Crocker, J. Vittal, K.T. Pogran, D.A. Henderson

RFC 734 *SUPDUP Protocol* M.R. Crispin

RFC 735 *Revised Telnet byte macro option* D. Crocker, R.H. Gumpertz

RFC 736 *Telnet SUPDUP option* M.R. Crispin

RFC 749 *Telnet SUPDUP—Output option* B. Greenberg

RFC 765 *File Transfer Protocol specification* J. Postel

RFC 768 *User Datagram Protocol* J. Postel

RFC 779 *Telnet send-location option* E. Killian

RFC 783 *TFTP Protocol (revision 2)* K.R. Sollins

RFC 791 *Internet Protocol* J. Postel

RFC 792 *Internet Control Message Protocol* J. Postel

RFC 793 *Transmission Control Protocol* J. Postel

RFC 820 *Assigned numbers* J. Postel

RFC 821 *Simple Mail Transfer Protocol* J. Postel

RFC 822 *Standard for the format of ARPA Internet text messages* D. Crocker

RFC 823 *DARPA Internet gateway* R. Hinden, A. Sheltzer

RFC 826 *Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware* D. Plummer

RFC 854 *Telnet Protocol Specification* J. Postel, J. Reynolds

RFC 855 *Telnet Option Specification* J. Postel, J. Reynolds

RFC 856 *Telnet Binary Transmission* J. Postel, J. Reynolds

RFC 857 *Telnet Echo Option* J. Postel, J. Reynolds

RFC 858 *Telnet Suppress Go Ahead Option* J. Postel, J. Reynolds

RFC 859 *Telnet Status Option* J. Postel, J. Reynolds

RFC 860 *Telnet Timing Mark Option* J. Postel, J. Reynolds

RFC 861 *Telnet Extended Options: List Option* J. Postel, J. Reynolds

RFC 862 *Echo Protocol* J. Postel

RFC 863 *Discard Protocol* J. Postel

RFC 864 *Character Generator Protocol* J. Postel

RFC 865 *Quote of the Day Protocol* J. Postel

RFC 868 *Time Protocol* J. Postel, K. Harrenstien

RFC 877 *Standard for the transmission of IP datagrams over public data networks* J.T. Korb

RFC 883 *Domain names: Implementation specification* P.V. Mockapetris

RFC 884 *Telnet terminal type option* M. Solomon, E. Wimmers

RFC 885 *Telnet end of record option* J. Postel

RFC 894 *Standard for the transmission of IP datagrams over Ethernet networks* C. Hornig

RFC 896 *Congestion control in IP/TCP internetworks* J. Nagle

RFC 903 *Reverse Address Resolution Protocol* R. Finlayson, T. Mann, J. Mogul, M. Theimer

RFC 904 *Exterior Gateway Protocol formal specification* D. Mills

RFC 919 *Broadcasting Internet Datagrams* J. Mogul

RFC 922 *Broadcasting Internet datagrams in the presence of subnets* J. Mogul

RFC 927 *TACACS user identification Telnet option* B.A. Anderson

RFC 933 *Output marking Telnet option* S. Silverman

RFC 946 *Telnet terminal location number option* R. Nedved

RFC 950 *Internet Standard Subnetting Procedure* J. Mogul, J. Postel

RFC 952 *DoD Internet host table specification* K. Harrenstien, M. Stahl, E. Feinler

RFC 959 *File Transfer Protocol* J. Postel, J.K. Reynolds

RFC 961 *Official ARPA-Internet protocols* J.K. Reynolds, J. Postel

RFC 974 *Mail routing and the domain system* C. Partridge

RFC 1001 *Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods* NetBios Working Group in the Defense Advanced Research Projects Agency, Internet Activities Board, End-to-End Services Task Force

RFC 1002 *Protocol Standard for a NetBIOS service on a TCP/UDP transport: Detailed specifications* NetBios Working Group in the Defense Advanced Research Projects Agency, Internet Activities Board, End-to-End Services Task Force

RFC 1006 *ISO transport services on top of the TCP: Version 3* M.T. Rose, D.E. Cass

RFC 1009 *Requirements for Internet gateways* R. Braden, J. Postel

RFC 1011 *Official Internet protocols* J. Reynolds, J. Postel

RFC 1013 *X Window System Protocol, version 11: Alpha update April 1987* R. Scheifler

RFC 1014 *XDR: External Data Representation standard* Sun Microsystems

RFC 1027 *Using ARP to implement transparent subnet gateways* S. Carl-Mitchell, J. Quarterman

RFC 1032 *Domain administrators guide* M. Stahl

RFC 1033 *Domain administrators operations guide* M. Lottor

RFC 1034 *Domain names—concepts and facilities* P.V. Mockapetris

RFC 1035 *Domain names—implementation and specification* P.V. Mockapetris

RFC 1038 *Draft revised IP security option* M. St. Johns

RFC 1041 *Telnet 3270 regime option* Y. Rekhter

RFC 1042 *Standard for the transmission of IP datagrams over IEEE 802 networks* J. Postel, J. Reynolds

RFC 1043 *Telnet Data Entry Terminal option: DODIIS implementation* A. Yasuda, T. Thompson

RFC 1044 *Internet Protocol on Network System's HYPERchannel: Protocol specification* K. Hardwick, J. Lekashman

RFC 1053 *Telnet X.3 PAD option* S. Levy, T. Jacobson

RFC 1055 *Nonstandard for transmission of IP datagrams over serial lines: SLIP* J. Romkey

RFC 1057 *RPC: Remote Procedure Call Protocol Specification: Version 2* Sun Microsystems

RFC 1058 *Routing Information Protocol* C. Hedrick

RFC 1060 *Assigned numbers* J. Reynolds, J. Postel

RFC 1067 *Simple Network Management Protocol* J.D. Case, M. Fedor, M.L. Schoffstall, J. Davin

RFC 1071 *Computing the Internet checksum* R.T. Braden, D.A. Borman, C. Partridge

RFC 1072 *TCP extensions for long-delay paths* V. Jacobson, R.T. Braden

RFC 1073 *Telnet window size option* D. Waitzman

RFC 1079 *Telnet terminal speed option* C. Hedrick

RFC 1085 *ISO presentation services on top of TCP/IP based internets* M.T. Rose

RFC 1091 *Telnet terminal-type option* J. VanBokkelen

RFC 1094 *NFS: Network File System Protocol specification* Sun Microsystems

RFC 1096 *Telnet X display location option* G. Marcy

RFC 1101 *DNS encoding of network names and other types* P. Mockapetris

RFC 1112 *Host extensions for IP multicasting* S.E. Deering

RFC 1113 *Privacy enhancement for Internet electronic mail: Part I — message encipherment and authentication procedures* J. Linn

RFC 1118 *Hitchhikers Guide to the Internet* E. Krol

RFC 1122 *Requirements for Internet Hosts—Communication Layers* R. Braden, Ed.

RFC 1123 *Requirements for Internet Hosts—Application and Support* R. Braden, Ed.

RFC 1146 *TCP alternate checksum options* J. Zweig, C. Partridge

RFC 1155 *Structure and identification of management information for TCP/IP-based internets* M. Rose, K. McCloghrie

RFC 1156 *Management Information Base for network management of TCP/IP-based internets* K. McCloghrie, M. Rose

RFC 1157 *Simple Network Management Protocol (SNMP)* J. Case, M. Fedor, M. Schoffstall, J. Davin

RFC 1158 *Management Information Base for network management of TCP/IP-based internets: MIB-II* M. Rose

RFC 1166 *Internet numbers* S. Kirkpatrick, M.K. Stahl, M. Recker

RFC 1179 *Line printer daemon protocol* L. McLaughlin

RFC 1180 *TCP/IP tutorial* T. Socolofsky, C. Kale

RFC 1183 *New DNS RR Definitions* C.F. Everhart, L.A. Mamakos, R. Ullmann, P.V. Mockapetris

RFC 1184 *Telnet Linemode Option* D. Borman

RFC 1186 *MD4 Message Digest Algorithm* R.L. Rivest

RFC 1187 *Bulk Table Retrieval with the SNMP* M. Rose, K. McCloghrie, J. Davin

RFC 1188 *Proposed Standard for the Transmission of IP Datagrams over FDDI Networks* D. Katz

RFC 1190 *Experimental Internet Stream Protocol: Version 2 (ST-II)* C. Topolcic

RFC 1191 *Path MTU discovery* J. Mogul, S. Deering

RFC 1198 *FYI on the X window system* R. Scheifler

RFC 1207 *FYI on Questions and Answers: Answers to commonly asked "experienced Internet user" questions* G. Malkin, A. Marine, J. Reynolds

RFC 1208 *Glossary of networking terms* O. Jacobsen, D. Lynch

RFC 1213 *Management Information Base for Network Management of TCP/IP-based internets: MIB-II* K. McCloghrie, M.T. Rose

RFC 1215 *Convention for defining traps for use with the SNMP* M. Rose

RFC 1227 *SNMP MUX protocol and MIB* M.T. Rose

RFC 1228 *SNMP-DPI: Simple Network Management Protocol Distributed Program Interface* G. Carpenter, B. Wijnen

RFC 1229 *Extensions to the generic-interface MIB* K. McCloghrie

RFC 1230 *IEEE 802.4 Token Bus MIB* K. McCloghrie, R. Fox

RFC 1231 *IEEE 802.5 Token Ring MIB* K. McCloghrie, R. Fox, E. Decker

RFC 1236 *IP to X.121 address mapping for DDN* L. Morales, P. Hasse

RFC 1256 *ICMP Router Discovery Messages* S. Deering, Ed.

RFC 1267 *Border Gateway Protocol 3 (BGP-3)* K. Lougheed, Y. Rekhter

RFC 1268 *Application of the Border Gateway Protocol in the Internet* Y. Rekhter, P. Gross

RFC 1269 *Definitions of Managed Objects for the Border Gateway Protocol: Version 3* S. Willis, J. Burruss

RFC 1270 *SNMP Communications Services* F. Kastenholz, ed.

RFC 1285 *FDDI Management Information Base* J. Case

RFC 1315 *Management Information Base for Frame Relay DTEs* C. Brown, F. Baker, C. Carvalho

RFC 1321 *The MD5 Message-Digest Algorithm* R. Rivest

RFC 1323 *TCP Extensions for High Performance* V. Jacobson, R. Braden, D. Borman

RFC 1325 *FYI on Questions and Answers: Answers to Commonly Asked "New Internet User" Questions* G. Malkin, A. Marine

RFC 1327 *Mapping between X.400 (1988)/ISO 10021 and RFC 822* S. Hardcastle-Kille

RFC 1340 *Assigned Numbers* J. Reynolds, J. Postel

RFC 1344 *Implications of MIME for Internet Mail Gateways* N. Bornstein

RFC 1349 *Type of Service in the Internet Protocol Suite* P. Almquist

RFC 1350 *The TFTP Protocol (Revision 2)* K.R. Sollins

RFC 1351 *SNMP Administrative Model* J. Davin, J. Galvin, K. McCloghrie

RFC 1352 *SNMP Security Protocols* J. Galvin, K. McCloghrie, J. Davin

RFC 1353 *Definitions of Managed Objects for Administration of SNMP Parties* K. McCloghrie, J. Davin, J. Galvin

RFC 1354 *IP Forwarding Table MIB* F. Baker

RFC 1356 *Multiprotocol Interconnect on X.25 and ISDN in the Packet Mode* A. Malis, D. Robinson, R. Ullmann

RFC 1358 *Charter of the Internet Architecture Board (IAB)* L. Chapin

RFC 1363 *A Proposed Flow Specification* C. Partridge

RFC 1368 *Definition of Managed Objects for IEEE 802.3 Repeater Devices* D. McMaster, K. McCloghrie

RFC 1372 *Telnet Remote Flow Control Option* C. L. Hedrick, D. Borman

RFC 1374 *IP and ARP on HIPPI* J. Renwick, A. Nicholson

RFC 1381 *SNMP MIB Extension for X.25 LAPB* D. Throop, F. Baker

RFC 1382 *SNMP MIB Extension for the X.25 Packet Layer* D. Throop

RFC 1387 *RIP Version 2 Protocol Analysis* G. Malkin

RFC 1388 *RIP Version 2 Carrying Additional Information* G. Malkin

RFC 1389 *RIP Version 2 MIB Extensions* G. Malkin, F. Baker

RFC 1390 *Transmission of IP and ARP over FDDI Networks* D. Katz

RFC 1393 *Traceroute Using an IP Option* G. Malkin

RFC 1398 *Definitions of Managed Objects for the Ethernet-Like Interface Types* F. Kastenholz

RFC 1408 *Telnet Environment Option* D. Borman, Ed.

RFC 1413 *Identification Protocol* M. St. Johns

RFC 1416 *Telnet Authentication Option* D. Borman, ed.

RFC 1420 *SNMP over IPX* S. Bostock

RFC 1428 *Transition of Internet Mail from Just-Send-8 to 8bit-SMTP/MIME* G. Vaudreuil

RFC 1442 *Structure of Management Information for version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1443 *Textual Conventions for version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1445 *Administrative Model for version 2 of the Simple Network Management Protocol (SNMPv2)* J. Galvin, K. McCloghrie

RFC 1447 *Party MIB for version 2 of the Simple Network Management Protocol (SNMPv2)* K. McCloghrie, J. Galvin

RFC 1448 *Protocol Operations for version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1464 *Using the Domain Name System to Store Arbitrary String Attributes* R. Rosenbaum

RFC 1469 *IP Multicast over Token-Ring Local Area Networks* T. Pusateri

RFC 1483 *Multiprotocol Encapsulation over ATM Adaptation Layer 5* Juha Heinanen

RFC 1514 *Host Resources MIB* P. Grillo, S. Waldbusser

RFC 1516 *Definitions of Managed Objects for IEEE 802.3 Repeater Devices* D. McMaster, K. McCloghrie

RFC 1521 *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies* N. Borenstein, N. Freed

RFC 1535 *A Security Problem and Proposed Correction With Widely Deployed DNS Software* E. Gavron

RFC 1536 *Common DNS Implementation Errors and Suggested Fixes* A. Kumar, J. Postel, C. Neuman, P. Danzig, S. Miller

RFC 1537 *Common DNS Data File Configuration Errors* P. Beertema

RFC 1540 *Internet Official Protocol Standards* J. Postel

RFC 1571 *Telnet Environment Option Interoperability Issues* D. Borman

RFC 1572 *Telnet Environment Option* S. Alexander

RFC 1573 *Evolution of the Interfaces Group of MIB-II* K. McCloghrie, F. Kastenholz

RFC 1577 *Classical IP and ARP over ATM* M. Laubach

RFC 1583 *OSPF Version 2* J. Moy

RFC 1591 *Domain Name System Structure and Delegation* J. Postel

RFC 1592 *Simple Network Management Protocol Distributed Protocol Interface Version 2.0* B. Wijnen, G. Carpenter, K. Curran, A. Sehgal, G. Waters

RFC 1594 *FYI on Questions and Answers— Answers to Commonly Asked "New Internet User" Questions* A. Marine, J. Reynolds, G. Malkin

RFC 1644 *T/TCP — TCP Extensions for Transactions Functional Specification* R. Braden

RFC 1646 *TN3270 Extensions for LUname and Printer Selection* C. Graves, T. Butts, M. Angel

RFC 1647 *TN3270 Enhancements* B. Kelly

RFC 1652 *SMTP Service Extension for 8bit-MIMEtransport* J. Klensin, N. Freed, M. Rose, E. Stefferud, D. Crocker

RFC 1664 *Using the Internet DNS to Distribute RFC1327 Mail Address Mapping Tables* C. Allochio, A. Bonito, B. Cole, S. Giordano, R. Hagens

RFC 1693 *An Extension to TCP: Partial Order Service* T. Connolly, P. Amer, P. Conrad

RFC 1695 *Definitions of Managed Objects for ATM Management Version 8.0 using SMIv2* M. Ahmed, K. Tesink

RFC 1701 *Generic Routing Encapsulation (GRE)* S. Hanks, T. Li, D. Farinacci, P. Traina

RFC 1702 *Generic Routing Encapsulation over IPv4 networks* S. Hanks, T. Li, D. Farinacci, P. Traina

RFC 1706 *DNS NSAP Resource Records* B. Manning, R. Colella

RFC 1712 *DNS Encoding of Geographical Location* C. Farrell, M. Schulze, S. Pleitner D. Baldoni

RFC 1713 *Tools for DNS debugging* A. Romao

RFC 1723 *RIP Version 2—Carrying Additional Information* G. Malkin

RFC 1752 *The Recommendation for the IP Next Generation Protocol* S. Bradner, A. Mankin

RFC 1766 *Tags for the Identification of Languages* H. Alvestrand

RFC 1771 *A Border Gateway Protocol 4 (BGP-4)* Y. Rekhter, T. Li

RFC 1794 *DNS Support for Load Balancing* T. Brisco

RFC 1819 *Internet Stream Protocol Version 2 (ST2) Protocol Specification—Version ST2+* L. Delgrossi, L. Berger Eds.

RFC 1826 *IP Authentication Header* R. Atkinson

RFC 1828 *IP Authentication using Keyed MD5* P. Metzger, W. Simpson

RFC 1829 *The ESP DES-CBC Transform* P. Karn, P. Metzger, W. Simpson

RFC 1830 *SMTP Service Extensions for Transmission of Large and Binary MIME Messages* G. Vaudreuil

RFC 1831 *RPC: Remote Procedure Call Protocol Specification Version 2* R. Srinivasan

RFC 1832 *XDR: External Data Representation Standard* R. Srinivasan

RFC 1833 *Binding Protocols for ONC RPC Version 2* R. Srinivasan

RFC 1850 *OSPF Version 2 Management Information Base* F. Baker, R. Coltun

RFC 1854 *SMTP Service Extension for Command Pipelining* N. Freed

RFC 1869 *SMTP Service Extensions* J. Klensin, N. Freed, M. Rose, E. Stefferud, D. Crocker

RFC 1870 *SMTP Service Extension for Message Size Declaration* J. Klensin, N. Freed, K. Moore

RFC 1876 *A Means for Expressing Location Information in the Domain Name System* C. Davis, P. Vixie, T. Goodwin, I. Dickinson

RFC 1883 *Internet Protocol, Version 6 (IPv6) Specification* S. Deering, R. Hinden

RFC 1884 *IP Version 6 Addressing Architecture* R. Hinden, S. Deering, Eds.

RFC 1886 *DNS Extensions to support IP version 6* S. Thomson, C. Huitema

RFC 1888 *OSI NSAPs and IPv6* J. Bound, B. Carpenter, D. Harrington, J. Houldsworth, A. Lloyd

RFC 1891 *SMTP Service Extension for Delivery Status Notifications* K. Moore

RFC 1892 *The Multipart/Report Content Type for the Reporting of Mail System Administrative Messages* G. Vaudreuil

RFC 1894 *An Extensible Message Format for Delivery Status Notifications*K. Moore, G. Vaudreuil

RFC 1901 *Introduction to Community-based SNMPv2* J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1902 *Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1903 *Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1904 *Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1905 *Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1906 *Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1907 *Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)* J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1908 *Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework* J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 1912 *Common DNS Operational and Configuration Errors* D. Barr

RFC 1918 *Address Allocation for Private Internets* Y. Rekhter, B. Moskowitz, D. Karrenberg, G.J. de Groot, E. Lear

RFC 1928 *SOCKS Protocol Version 5* M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones

RFC 1930 *Guidelines for creation, selection, and registration of an Autonomous System (AS)* J. Hawkinson, T. Bates

RFC 1939 *Post Office Protocol-Version 3* J. Myers, M. Rose

RFC 1981 *Path MTU Discovery for IP version 6* J. McCann, S. Deering, J. Mogul

RFC 1982 *Serial Number Arithmetic* R. Elz, R. Bush

RFC 1985 *SMTP Service Extension for Remote Message Queue Starting* J. De Winter

RFC 1995 *Incremental Zone Transfer in DNS* M. Ohta

RFC 1996 *A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY)* P. Vixie

RFC 2010 *Operational Criteria for Root Name Servers* B. Manning, P. Vixie

RFC 2011 *SNMPv2 Management Information Base for the Internet Protocol using SMIv2* K. McCloghrie, Ed.

RFC 2012 *SNMPv2 Management Information Base for the Transmission Control Protocol using SMIv2* K. McCloghrie, Ed.

RFC 2013 *SNMPv2 Management Information Base for the User Datagram Protocol using SMIv2* K. McCloghrie, Ed.

RFC 2018 *TCP Selective Acknowledgement Options* M. Mathis, J. Mahdavi, S. Floyd, A. Romanow

RFC 2026 *The Internet Standards Process — Revision 3* S. Bradner

RFC 2030 *Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI* D. Mills

RFC 2033 *Local Mail Transfer Protocol* J. Myers

RFC 2034 *SMTP Service Extension for Returning Enhanced Error Codes*N. Freed

RFC 2040 *The RC5, RC5–CBC, RC-5–CBC-Pad, and RC5–CTS Algorithms*R. Baldwin, R. Rivest

RFC 2045 *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies* N. Freed, N. Borenstein

RFC 2052 *A DNS RR for specifying the location of services (DNS SRV)* A. Gulbrandsen, P. Vixie

RFC 2065 *Domain Name System Security Extensions* D. Eastlake 3rd, C. Kaufman

RFC 2066 *TELNET CHARSET Option* R. Gellens

RFC 2080 *RIPng for IPv6* G. Malkin, R. Minnear

RFC 2096 *IP Forwarding Table MIB* F. Baker

RFC 2104 *HMAC: Keyed-Hashing for Message Authentication* H. Krawczyk, M. Bellare, R. Canetti

RFC 2119 *Keywords for use in RFCs to Indicate Requirement Levels* S. Bradner

RFC 2133 *Basic Socket Interface Extensions for IPv6* R. Gilligan, S. Thomson, J. Bound, W. Stevens

RFC 2136 *Dynamic Updates in the Domain Name System (DNS UPDATE)* P. Vixie, Ed., S. Thomson, Y. Rekhter, J. Bound

RFC 2137 *Secure Domain Name System Dynamic Update* D. Eastlake 3rd

RFC 2163 *Using the Internet DNS to Distribute MIXER Conformant Global Address Mapping (MCGAM)* C. Allocchio

RFC 2168 *Resolution of Uniform Resource Identifiers using the Domain Name System* R. Daniel, M. Mealling

RFC 2178 *OSPF Version 2* J. Moy

RFC 2181 *Clarifications to the DNS Specification* R. Elz, R. Bush

RFC 2205 *Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification* R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, S. Jamin

RFC 2210 *The Use of RSVP with IETF Integrated Services* J. Wroclawski

RFC 2211 *Specification of the Controlled-Load Network Element Service* J. Wroclawski

RFC 2212 *Specification of Guaranteed Quality of Service* S. Shenker, C. Partridge, R. Guerin

RFC 2215 *General Characterization Parameters for Integrated Service Network Elements* S. Shenker, J. Wroclawski

RFC 2217 *Telnet Com Port Control Option* G. Clarke

RFC 2219 *Use of DNS Aliases for Network Services* M. Hamilton, R. Wright

RFC 2228 *FTP Security Extensions* M. Horowitz, S. Lunt

RFC 2230 *Key Exchange Delegation Record for the DNS* R. Atkinson

RFC 2233 *The Interfaces Group MIB using SMIv2* K. McCloghrie, F. Kastenholz

RFC 2240 *A Legal Basis for Domain Name Allocation* O. Vaughn

RFC 2246 *The TLS Protocol Version 1.0* T. Dierks, C. Allen

RFC 2251 *Lightweight Directory Access Protocol (v3)* M. Wahl, T. Howes, S. Kille

RFC 2253 *Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names* M. Wahl, S. Kille, T. Howes

RFC 2254 *The String Representation of LDAP Search Filters* T. Howes

RFC 2261 *An Architecture for Describing SNMP Management Frameworks* D. Harrington, R. Presuhn, B. Wijnen

RFC 2262 *Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)* J. Case, D. Harrington, R. Presuhn, B. Wijnen

RFC 2271 *An Architecture for Describing SNMP Management Frameworks* D. Harrington, R. Presuhn, B. Wijnen

RFC 2273 *SNMPv3 Applications* D. Levi, P. Meyer, B. Stewartz

RFC 2274 *User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)* U. Blumenthal, B. Wijnen

RFC 2275 *View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)* B. Wijnen, R. Presuhn, K. McCloghrie

RFC 2279 *UTF-8, a transformation format of ISO 10646* F. Yergeau

RFC 2292 *Advanced Sockets API for IPv6* W. Stevens, M. Thomas

RFC 2308 *Negative Caching of DNS Queries (DNS NCACHE)* M. Andrews

RFC 2317 *Classless IN-ADDR.ARPA delegation* H. Eidnes, G. de Groot, P. Vixie

RFC 2320 *Definitions of Managed Objects for Classical IP and ARP Over ATM Using SMIv2 (IPOA-MIB)* M. Greene, J. Luciani, K. White, T. Kuo

RFC 2328 *OSPF Version 2* J. Moy

RFC 2345 *Domain Names and Company Name Retrieval* J. Klensin, T. Wolf, G. Oglesby

RFC 2352 *A Convention for Using Legal Names as Domain Names* O. Vaughn

RFC 2355 *TN3270 Enhancements* B. Kelly

RFC 2358 *Definitions of Managed Objects for the Ethernet-like Interface Types* J. Flick, J. Johnson

RFC 2373 *IP Version 6 Addressing Architecture* R. Hinden, S. Deering

RFC 2374 *An IPv6 Aggregatable Global Unicast Address Format* R. Hinden, M. O'Dell, S. Deering

RFC 2375 *IPv6 Multicast Address Assignments* R. Hinden, S. Deering

RFC 2385 *Protection of BGP Sessions via the TCP MD5 Signature Option*A. Hefferman

RFC 2389 *Feature negotiation mechanism for the File Transfer Protocol* P. Hethmon, R. Elz

RFC 2401 *Security Architecture for Internet Protocol* S. Kent, R. Atkinson

RFC 2402 *IP Authentication Header* S. Kent, R. Atkinson

RFC 2403 *The Use of HMAC-MD5–96 within ESP and AH* C. Madson, R. Glenn

RFC 2404 *The Use of HMAC-SHA–1–96 within ESP and AH* C. Madson, R. Glenn

RFC 2405 *The ESP DES-CBC Cipher Algorithm With Explicit IV* C. Madson, N. Doraswamy

RFC 2406 *IP Encapsulating Security Payload (ESP)* S. Kent, R. Atkinson

RFC 2407 *The Internet IP Security Domain of Interpretation for ISAKMP*D. Piper

RFC 2408 *Internet Security Association and Key Management Protocol (ISAKMP)* D. Maughan, M. Schertler, M. Schneider, J. Turner

RFC 2409 *The Internet Key Exchange (IKE)* D. Harkins, D. Carrel

RFC 2410 *The NULL Encryption Algorithm and Its Use With IPsec* R. Glenn, S. Kent,

RFC 2428 *FTP Extensions for IPv6 and NATs* M. Allman, S. Ostermann, C. Metz

RFC 2445 *Internet Calendaring and Scheduling Core Object Specification (iCalendar)* F. Dawson, D. Stenerson

RFC 2459 *Internet X.509 Public Key Infrastructure Certificate and CRL Profile* R. Housley, W. Ford, W. Polk, D. Solo

RFC 2460 *Internet Protocol, Version 6 (IPv6) Specification* S. Deering, R. Hinden

RFC 2461 *Neighbor Discovery for IP Version 6 (IPv6)* T. Narten, E. Nordmark, W. Simpson

RFC 2462 *IPv6 Stateless Address Autoconfiguration* S. Thomson, T. Narten

RFC 2463 *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification* A. Conta, S. Deering

RFC 2464 *Transmission of IPv6 Packets over Ethernet Networks* M. Crawford

RFC 2466 *Management Information Base for IP Version 6: ICMPv6 Group* D. Haskin, S. Onishi

RFC 2476 *Message Submission* R. Gellens, J. Klensin

RFC 2487 *SMTP Service Extension for Secure SMTP over TLS* P. Hoffman

RFC 2505 *Anti-Spam Recommendations for SMTP MTAs* G. Lindberg

RFC 2523 *Photuris: Extended Schemes and Attributes* P. Karn, W. Simpson

RFC 2535 *Domain Name System Security Extensions* D. Eastlake 3rd

RFC 2538 *Storing Certificates in the Domain Name System (DNS)* D. Eastlake 3rd, O. Gudmundsson

RFC 2539 *Storage of Diffie-Hellman Keys in the Domain Name System (DNS)* D. Eastlake 3rd

RFC 2540 *Detached Domain Name System (DNS) Information* D. Eastlake 3rd

RFC 2554 *SMTP Service Extension for Authentication* J. Myers

RFC 2570 *Introduction to Version 3 of the Internet-standard Network Management Framework* J. Case, R. Mundy, D. Partain, B. Stewart

RFC 2571 *An Architecture for Describing SNMP Management Frameworks* B. Wijnen, D. Harrington, R. Presuhn

RFC 2572 *Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)* J. Case, D. Harrington, R. Presuhn, B. Wijnen

RFC 2573 *SNMP Applications* D. Levi, P. Meyer, B. Stewart

RFC 2574 *User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)* U. Blumenthal, B. Wijnen

RFC 2575 *View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)* B. Wijnen, R. Presuhn, K. McCloghrie

RFC 2576 *Co-Existence between Version 1, Version 2, and Version 3 of the Internet-standard Network Management Framework* R. Frye, D. Levi, S. Routhier, B. Wijnen

RFC 2578 *Structure of Management Information Version 2 (SMIv2)* K. McCloghrie, D. Perkins, J. Schoenwaelder

RFC 2579 *Textual Conventions for SMIv2* K. McCloghrie, D. Perkins, J. Schoenwaelder

RFC 2580 *Conformance Statements for SMIv2* K. McCloghrie, D. Perkins, J. Schoenwaelder

RFC 2581 *TCP Congestion Control* M. Allman, V. Paxson, W. Stevens

RFC 2583 *Guidelines for Next Hop Client (NHC) Developers* R. Carlson, L. Winkler

RFC 2591 *Definitions of Managed Objects for Scheduling Management Operations* D. Levi, J. Schoenwaelder

RFC 2625 *IP and ARP over Fibre Channel* M. Rajagopal, R. Bhagwat, W. Rickard

RFC 2635 *Don't SPEW A Set of Guidelines for Mass Unsolicited Mailings and Postings (spam*)* S. Hambridge, A. Lunde

RFC 2637 *Point-to-Point Tunneling Protocol* K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, G. Zorn

RFC 2640 *Internationalization of the File Transfer Protocol* B. Curtin

RFC 2665 *Definitions of Managed Objects for the Ethernet-like Interface Types* J. Flick, J. Johnson

RFC 2671 *Extension Mechanisms for DNS (EDNS0)* P. Vixie

RFC 2672 *Non-Terminal DNS Name Redirection* M. Crawford

RFC 2675 *IPv6 Jumbograms* D. Borman, S. Deering, R. Hinden

RFC 2710 *Multicast Listener Discovery (MLD) for IPv6* S. Deering, W. Fenner, B. Haberman

RFC 2711 *IPv6 Router Alert Option* C. Partridge, A. Jackson

RFC 2740 *OSPF for IPv6* R. Coltun, D. Ferguson, J. Moy

RFC 2753 *A Framework for Policy-based Admission Control* R. Yavatkar, D. Pendarakis, R. Guerin

RFC 2782 *A DNS RR for specifying the location of services (DNS SRV)* A. Gubrandsen, P. Vixix, L. Esibov

RFC 2821 *Simple Mail Transfer Protocol* J. Klensin, Ed.

RFC 2822 *Internet Message Format* P. Resnick, Ed.

RFC 2840 *TELNET KERMIT OPTION* J. Altman, F. da Cruz

RFC 2845 *Secret Key Transaction Authentication for DNS (TSIG)* P. Vixie, O. Gudmundsson, D. Eastlake 3rd, B. Wellington

RFC 2851 *Textual Conventions for Internet Network Addresses* M. Daniele, B. Haberman, S. Routhier, J. Schoenwaelder

RFC 2852 *Deliver By SMTP Service Extension* D. Newman

RFC 2874 *DNS Extensions to Support IPv6 Address Aggregation and Renumbering* M. Crawford, C. Huitema

RFC 2915 *The Naming Authority Pointer (NAPTR) DNS Resource Record* M. Mealling, R. Daniel

RFC 2920 *SMTP Service Extension for Command Pipelining* N. Freed

RFC 2930 *Secret Key Establishment for DNS (TKEY RR)* D. Eastlake, 3rd

RFC 2941 *Telnet Authentication Option* T. Ts'o, ed., J. Altman

RFC 2942 *Telnet Authentication: Kerberos Version 5* T. Ts'o

RFC 2946 *Telnet Data Encryption Option* T. Ts'o

RFC 2952 *Telnet Encryption: DES 64 bit Cipher Feedback* T. Ts'o

RFC 2953 *Telnet Encryption: DES 64 bit Output Feedback* T. Ts'o

RFC 2992 *Analysis of an Equal-Cost Multi-Path Algorithm* C. Hopps

RFC 3019 *IP Version 6 Management Information Base for The Multicast Listener Discovery Protocol* B. Haberman, R. Worzella

RFC 3060 *Policy Core Information Model—Version 1 Specification* B. Moore, E. Ellesson, J. Strassner, A. Westerinen

RFC 3152 *Delegation of IP6.ARPA* R. Bush

RFC 3164 *The BSD Syslog Protocol* C. Lonvick

RFC 3207 *SMTP Service Extension for Secure SMTP over Transport Layer Security* P. Hoffman

RFC 3226 *DNSSEC and IPv6 A6 aware server/resolver message size requirements* O. Gudmundsson

RFC 3291 *Textual Conventions for Internet Network Addresses* M. Daniele, B. Haberman, S. Routhier, J. Schoenwaelder

RFC 3363 *Representing Internet Protocol version 6 (IPv6) Addresses in the Domain Name System* R. Bush, A. Durand, B. Fink, O. Gudmundsson, T. Hain

RFC 3376 *Internet Group Management Protocol, Version 3* B. Cain, S. Deering, I. Kouvelas, B. Fenner, A. Thyagarajan

RFC 3390 *Increasing TCP's Initial Window* M. Allman, S. Floyd, C. Partridge

RFC 3410 *Introduction and Applicability Statements for Internet-Standard Management Framework* J. Case, R. Mundy, D. Partain, B. Stewart

RFC 3411 *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks* D. Harrington, R. Presuhn, B. Wijnen

RFC 3412 *Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)* J. Case, D. Harrington, R. Presuhn, B. Wijnen

RFC 3413 *Simple Network Management Protocol (SNMP) Applications* D. Levi, P. Meyer, B. Stewart

RFC 3414 *User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)* U. Blumenthal, B. Wijnen

RFC 3415 *View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)* B. Wijnen, R. Presuhn, K. McCloghrie

RFC 3416 *Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP)* R. Presuhn, J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 3417 *Transport Mappings for the Simple Network Management Protocol (SNMP)* R. Presuhn, J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 3418 *Management Information Base (MIB) for the Simple Network Management Protocol (SNMP)* R. Presuhn, J. Case, K. McCloghrie, M. Rose, S. Waldbusser

RFC 3419 *Textual Conventions for Transport Addresses* M. Daniele, J. Schoenwaelder

RFC 3484 *Default Address Selection for Internet Protocol version 6 (IPv6)* R. Draves

RFC 3493 *Basic Socket Interface Extensions for IPv6* R. Gilligan, S. Thomson, J. Bound, J. McCann, W. Stevens

RFC 3513 *Internet Protocol Version 6 (IPv6) Addressing Architecture* R. Hinden, S. Deering

RFC 3526 *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)* T. Kivinen, M. Kojo

RFC 3542 *Advanced Sockets Application Programming Interface (API) for IPv6* W. Richard Stevens, M. Thomas, E. Nordmark, T. Jinmei

RFC 3566 *The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec* S. Frankel, H. Herbert

RFC 3569 *An Overview of Source-Specific Multicast (SSM)* S. Bhattacharyya, Ed.

RFC 3584 *Coexistence between Version 1, Version 2, and Version 3 of the Internet-standard Network Management Framework* R. Frye, D. Levi, S. Routhier, B. Wijnen

RFC 3602 *The AES-CBC Cipher Algorithm and Its Use with IPsec* S. Frankel, R. Glenn, S. Kelly

RFC 3629 *UTF-8, a transformation format of ISO 10646* R. Kermode, C. Vicisano

RFC 3658 *Delegation Signer (DS) Resource Record (RR)* O. Gudmundsson

RFC 3678 *Socket Interface Extensions for Multicast Source Filters* D. Thaler, B. Fenner, B. Quinn

RFC 3715 *IPsec-Network Address Translation (NAT) Compatibility Requirements* B. Aboba, W. Dixon

RFC 3810 *Multicast Listener Discovery Version 2 (MLDv2) for IPv6* R. Vida, Ed., L. Costa, Ed.

RFC 3947 *Negotiation of NAT-Traversal in the IKE* T. Kivinen, B. Swander, A. Huttunen, V. Volpe

RFC 3948 *UDP Encapsulation of IPsec ESP Packets* A. Huttunen, B. Swander, V. Volpe, L. DiBurro, M. Stenberg

RFC 4001 *Textual Conventions for Internet Network Addresses* M. Daniele, B. Haberman, S. Routhier, J. Schoenwaelder

RFC 4007 *IPv6 Scoped Address Architecture* S. Deering, B. Haberman, T. Jinmei, E. Nordmark, B. Zill

RFC 4022 *Management Information Base for the Transmission Control Protocol (TCP)* R. Raghunarayan

RFC 4106 *The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)* J. Viega, D. McGrew

RFC 4109 *Algorithms for Internet Key Exchange version 1 (IKEv1)* P. Hoffman

RFC 4113 *Management Information Base for the User Datagram Protocol (UDP)* B. Fenner, J. Flick

RFC 4191 *Default Router Preferences and More-Specific Routes* R. Draves, D. Thaler

RFC 4217 *Securing FTP with TLS* P. Ford-Hutchinson

RFC 4292 *IP Forwarding Table MIB* B. Haberman

RFC 4293 *Management Information Base for the Internet Protocol (IP)* S. Routhier

RFC 4301 *Security Architecture for the Internet Protocol* S. Kent, K. Seo

RFC 4302 *IP Authentication Header* S. Kent

RFC 4303 *IP Encapsulating Security Payload (ESP)* S. Kent

RFC 4304 *Extended Sequence Number (ESN) Addendum to IPsec Domain of Interpretation (DOI) for Internet Security Association and Key Management Protocol (ISAKMP)* S. Kent

RFC 4307 *Cryptographic Algorithms for Use in the Internet Key Exchange Version 2 (IKEv2)* J. Schiller

RFC 4308 *Cryptographic Suites for IPsec* P. Hoffman

RFC 4434 *The AES-XCBC-PRF-128 Algorithm for the Internet Key Exchange Protocol* P. Hoffman

RFC 4552 *Authentication/Confidentiality for OSPFv3* M. Gupta, N. Melam

RFC 4678 *Server/Application State Protocol v1* A. Bivens

RFC 4753 *ECP Groups for IKE and IKEv2* D. Fu, J. Solinas

RFC 4754 *IKE and IKEv2 Authentication Using the Elliptic Curve Digital Signature Algorithm (ECDSA)* D. Fu, J. Solinas

RFC 4809 *Requirements for an IPsec Certificate Management Profile* C. Bonatti, Ed., S. Turner, Ed., G. Lebovitz, Ed.

RFC 4835 *Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header* V. Manral

RFC 4862 *IPv6 Stateless Address Autoconfiguration* S. Thomson, T. Narten, T. Jinmei

RFC 4868 *Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec* S. Kelly, S. Frankel

RFC 4869 *Suite B Cryptographic Suites for IPsec* L. Law, J. Solinas

RFC 4941 *Privacy Extensions for Stateless Address Autoconfiguration in IPv6* T. Narten, R. Draves, S. Krishnan

RFC 4945 *The Internet IP Security PKI Profile of IKEv1/ISAKMP, IKEv2, and PKIX* B. Korver

RFC 5014 *IPv6 Socket API for Source Address Selection* E. Nordmark, S. Chakrabarti, J. Laganier

RFC 5095 *Deprecation of Type 0 Routing Headers in IPv6* J. Abley, P. Savola, G. Neville-Neil

RFC 5175 *IPv6 Router Advertisement Flags Option* B. Haberman, Ed., R. Hinden

RFC 5282 *Using Authenticated Encryption Algorithms with the Encrypted Payload of the Internet Key Exchange version 2 (IKEv2) Protocol* D. Black, D. McGrew

RFC 5996 *Internet Key Exchange Protocol Version 2 (IKEv2)* C. Kaufman, P. Hoffman, Y. Nir, P. Eronen

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-ietf-ipngwg-icmp-v3-07
Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification A. Conta, S. Deering

Appendix I. Accessibility

Publications for this product are offered in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when using PDF files, you may view the information through the z/OS Internet Library website or the z/OS Information Center. If you continue to experience problems, send an email to mhvrcfs@us.ibm.com or write to:

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
USA

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 , , and 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 or Library Server versions of z/OS books in the Internet library at www.ibm.com/systems/z/os/zos/bkserv/.

One exception is command syntax that is published in railroad track format, which is accessible using screen readers with the Information Center, as described in "Dotted decimal syntax diagrams."

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the Information Center using a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always

present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- A question mark (?) means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- An exclamation mark (!) means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted

decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

- An asterisk (*) means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

  Notes:

  1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
  2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
  3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the USA.

IBM may not offer all of the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14 Shimotsuruma,, Yamato-Shi
Kanagawa 242-8502 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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.

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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

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_.

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.

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.

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.

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 website at http://www.opengroup.org/publications/catalog/c809.htm for the source of the specification. See *z/OS Communications Server: IP Programmer's Guide and Reference* for details about IBM use of this information.

If you are viewing this information softcopy, the 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.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

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

Adobe and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

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

Other product and service names might be trademarks of IBM or other companies.

Bibliography

This bibliography contains descriptions of the documents in the z/OS Communications Server library.

z/OS Communications Server documentation is available in the following forms:

- Online at the z/OS Internet Library web page at www.ibm.com/systems/z/os/zos/bkserv/
- In softcopy on CD-ROM collections. See "Softcopy information" on page xxxii.

z/OS Communications Server library updates

An index to z/OS Communications Server book updates is at http://www.ibm.com/support/docview.wss?uid=swg21178966. Updates to documents are also available on RETAIN® and in information APARs (info APARs). Go to http://www.ibm.com/software/network/commserver/zos/support to view information APARs. In addition, Info APARs for z/OS documents are in *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.

z/OS Communications Server information

z/OS Communications Server product information is grouped by task in the following tables.

Planning

Title	Number	Description
	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.
	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
	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 .

Title	Number	Description
	SC31-8776	This document presents information for people who want to administer and maintain IP. Use this document in conjunction with the . The information in this document includes: • TCP/IP configuration data sets • Configuration statements • Translation tables • Protocol number and port assignments
	SC31-8777	This document presents the major concepts involved in implementing an SNA network. Use this document in conjunction with the .
	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 .
	SC31-8836	This document contains sample definitions to help you implement SNA functions in your networks, and includes sample major node definitions.
	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
	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.
	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.
	SC31-8779	This document serves as a reference for programmers and operators requiring detailed information about specific operator commands.
	SX75-0124	This document contains essential information about SNA and IP commands.

Customization

Title	Number	Description
	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
	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.
	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.
	SC31-8830	This document is for programmers who want application programs that use the IMS TCP/IP application development services provided by the TCP/IP Services of IBM.
	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.
	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.
	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.)
	SC31-8810	This document provides reference material for the SNA LU 6.2 programming interface for host application programs.
	SC31-8808	This document describes how applications use the communications storage manager.

Title	Number	Description
	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
	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.
	GC23-8588-00	This document explains how to gather the trace data that is collected and stored in the host processor. It also explains how to use the Advanced Communications Function/Trace Analysis Program (ACF/TAP) service aid to produce reports for analyzing the trace data information.
and	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.
and	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
	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
	SC31-8783	This volume contains TCP/IP messages beginning with EZA.
	SC31-8784	This volume contains TCP/IP messages beginning with EZB or EZD.
	SC31-8785	This volume contains TCP/IP messages beginning with EZY.
	SC31-8786	This volume contains TCP/IP messages beginning with EZZ and SNM.
	SC31-8791	This document describes codes and other information that appear in z/OS Communications Server messages.

Index

Special characters

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

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.

Product Number: 5694–A01

Printed in USA

SC31-8787-14