
This edition applies to Version 4, Release 3, Modification 0 of IBM® z/VM (product number 5739-A03) and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC24-6021-00.

© Copyright International Business Machines Corporation 1987, 2002. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
# Contents

**Preface** ........................................... ix
  Who Should Read This Book .................. ix
  What You Should Know before Reading This Book  ix
  What This Book Contains ................. ix
  How to Use This Book .................... xi
    How the Term "internet" Is Used in This Book  xi
  Where to Find More Information ....... xi
  Service Information .................... xi
  Understanding Syntax Diagrams .......... xi
  How Numbers Are Used in This Book ..... xiv
  How to Send Your Comments to IBM .... xiv

**Summary of Changes** ............................ xv
  First Edition for z/VM Version 4 (October 2001) . xv
  z/VM C Socket API .............................. xv
  First Edition for z/VM Version 3 (February 2001) . xv
  IP Multicast .............................. xv
  Other Changes ............................ xv

**Chapter 1. z/VM C Socket Application**
**Programming Interface** ..................... 1
  TCP/IP Network Communication ............ 2
    Transport Protocols .................... 2
  What is a Socket? ........................ 3
    Address Families ..................... 4
    Socket Types ........................ 4
    Domain-specific Socket Addresses .... 5
  Client/Server Conversation ............... 8
    Server Perspective for AF_INET ..... 9
    Client Perspective for AF_INET ... 11
  Typical TCP Socket Session .............. 11
  Typical UDP Socket Session .............. 12
  Locating the Server's Port .............. 13
  Network Application Example .......... 13
  z/VM C Socket Implementation .......... 19
    Header Files ........................ 19
    Multithreading ...................... 20
    POSIX Signals and Thread Cancellation 21
    Sockets and Their Relationship to Other POSIX Functions ............. 22
    Miscellaneous Implementation Notes ........ 23
    Incompatibilities with the VM TCP/IP C Sockets Library .......... 23
    Incompatibilities with z/OS and OS/390 C Sockets ................. 26
    Incompatibilities with the Berkeley Socket Implementation ........ 26
  Compiling and Linking a Sockets Program . 26
    Compiling and Linking a z/VM C Sockets Program ............. 27
    Compiling and Linking a TCP/IP C Sockets Program ............ 29
  Running a Sockets Program .............. 31

  Preparing to Run a Sockets Program .... 31
  Using Environment Variables ............ 31
  Running a Program Residing in the BFS . 33
  Running a Program Residing on an Accessed Minidisk or SFS Directory 33
  C Sockets Quick Reference ............. 35
  TCP Client Program ..................... 36
  TCP Server Program .................... 38
  UDP Client Program .................... 40
  UDP Server Program .................... 41

**Chapter 2. TCP/UDP/IP API (Pascal Language)** ............................ 43
  Software Requirements .................... 43
  Data Structures .......................... 44
    Connection State ..................... 44
    Connection Information Record ....... 45
    Socket Record ......................... 46
    Notification Record ................... 46
    File Specification Record .......... 53
  Using Procedure Calls .................... 53
    Notifications ........................ 53
    TCP/UDP Initialization Procedures .... 54
    TCP/UDP Termination Procedure ....... 54
    Handling External Interrupts ......... 55
    TCP Communication Procedures ....... 55
    Ping Interface ........................ 55
    Monitor Procedures .................... 56
    UDP Communication Procedures ....... 56
    Raw IP Interface ....................... 56
    Timer Routines ........................ 57
    Host Lookup Routines .................. 57
    AddUserNote .......................... 58
    Other Routines ........................ 58
  Procedure Calls .......................... 58
    BeginTcpIp .......................... 58
    ClearTimer ............................ 59
    CreateTimer ........................... 59
    DestroyTimer .......................... 59
    EndTcpIp .............................. 59
    GetHostNumber ......................... 60
    GetHostResol .......................... 60
    GetHostString .......................... 61
    GetIdentity ............................ 61
    GetNextNote ........................... 61
    GetMsg .............................. 62
    Handle ............................ 62
    LocalAddress ........................... 63
    IsLocalHost ............................ 63
    MonCommand ............................ 64
    MonQuery ............................. 65
    NotifyIo .............................. 66
    PingRequest ........................... 67
    RawIpClose ............................ 67
    RawIpOpen ............................. 68
Chapter 5. Remote Procedure Calls 149

The RPC Interface 149

Portmapper 151

Contacting Portmapper 152

Target Assistance 152

RPCGEN Command 152

enum clnt_stat Structure. 153

Porting 154

Accessing System Return Messages 154

Printing System Return Messages 154

Enumerations 154

Compiling, Linking, and Running an RPC Program 154

RPC Global Variables 155

rpc_createerr 155

svc_fds 155

svc_fdset 156

Remote Procedure and eXternal Data Representation Calls 156

auth_destroy() 156

authnone_create() 156

authunix_create() 156

authunix_create_default() 157

callrpc() 157

clnt_broadcast() 158

clnt_call() 159

clnt_control() 160

clnt_create() 161

clntraw_create() 161

clnt_freeres() 161

clnt_geterr() 162

clnt_pcreateerror() 162

clnt_permot() 163

clntraw_create() 164

clnt_spcreateerror() 164

clnt_sperrno() 164

clntraw_create() 164

clntraw_create() 165

clnttcp_create() 165

clntudp_create() 166

get_myaddress() 167

getrpcport() 167

pmap_getmaps() 168

pmap_getport() 168

pmap_rmtcall() 169

pmap_set() 169

pmap_unset() 170

registerrpc() 170

svc_destroy() 171

svc_freargs() 171

svc_getargs() 172

svc_getcaller() 172

svc_getreq() 173

svc_getreqset() 173

svc_register() 173

svc_run() 174

svc_sendreply() 174

svc_unregister() 175

svcerr_auth() 175

svcerr_decode() 176

svcerr_noproc() 176

svcerr_noprog() 176

svcerr_systemerr() 177

svcerr_weakauth() 177

svcrw_create() 178

svctcp_create() 178

svcudp_create() 178

xdr_accepted_reply() 179

xdr_array() 179

xdr_authunix_parms() 180

xdr_bool() 180

xdr_bytes() 180

xdr_callhdr() 181

xdr_callmsg() 181

xdr_double() 182

xdr_enum() 182

xdr_float() 183

xdr_inline() 184

xdr_int() 184

xdr_long() 184

xdrOpaque() 185

xdr_opaque_auth() 185

xdr_pmap() 186

xdr_pmaplist() 186

xdr_pointer() 186

xdr_reference() 187

xdr_rejected_reply() 187

xdr_replymsg() 188

xdr_short() 188

xdr_string() 189

xdr_u_int() 189

xdr_u_long() 189

xdr_u_short() 190

xdr_union() 190

xdr_vector() 191

xdr_void() 191

xdr_wrapstring() 192

xdrmem_create() 192

dcrec_create() 192

dcrec_endofrecord() 193

dcrec_eof() 193

dcrec_skiprecord() 194

xrdstdio_create() 194

xprt_register() 194

xprt_unregister() 195

Sample RPC Programs 195

Running the Geneserv server and Genesend client 195

Running the Rawex program 196

RPC Geneserv Client 196

RPC Geneserv Server 197

RPC Rawex Raw Data Stream 198
## Chapter 6. X Window System Interface 201

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What Is Provided</td>
<td>201</td>
</tr>
<tr>
<td>Software Requirements</td>
<td>201</td>
</tr>
<tr>
<td>Using the X Window System Interface in the VM Environment</td>
<td>201</td>
</tr>
<tr>
<td>Application Resource File</td>
<td>203</td>
</tr>
<tr>
<td>Identifying the Target Display</td>
<td>204</td>
</tr>
<tr>
<td>Creating an Application</td>
<td>204</td>
</tr>
<tr>
<td>Generating X-Window System Applications</td>
<td>204</td>
</tr>
<tr>
<td>X Window System Subroutines</td>
<td>206</td>
</tr>
<tr>
<td>Opening and Closing a Display</td>
<td>206</td>
</tr>
<tr>
<td>Creating and Destroying Windows</td>
<td>206</td>
</tr>
<tr>
<td>Manipulating Windows</td>
<td>207</td>
</tr>
<tr>
<td>Changing Window Attributes</td>
<td>207</td>
</tr>
<tr>
<td>Obtaining Window Information</td>
<td>208</td>
</tr>
<tr>
<td>Obtaining Properties and Atoms</td>
<td>208</td>
</tr>
<tr>
<td>Manipulating Window Properties</td>
<td>208</td>
</tr>
<tr>
<td>Setting Window Selections</td>
<td>208</td>
</tr>
<tr>
<td>Manipulating Colormaps</td>
<td>209</td>
</tr>
<tr>
<td>Manipulating Color Cells</td>
<td>209</td>
</tr>
<tr>
<td>Creating and Freeing Pixmaps</td>
<td>209</td>
</tr>
<tr>
<td>Manipulating Graphics Contexts</td>
<td>210</td>
</tr>
<tr>
<td>Clearing and Copying Areas</td>
<td>211</td>
</tr>
<tr>
<td>Drawing Lines</td>
<td>211</td>
</tr>
<tr>
<td>Filling Areas</td>
<td>211</td>
</tr>
<tr>
<td>Loading and Freeing Fonts</td>
<td>212</td>
</tr>
<tr>
<td>Querying Character String Sizes</td>
<td>212</td>
</tr>
<tr>
<td>Drawing Text</td>
<td>213</td>
</tr>
<tr>
<td>Transferring Images</td>
<td>213</td>
</tr>
<tr>
<td>Manipulating Cursors</td>
<td>213</td>
</tr>
<tr>
<td>Handling Window Manager Functions</td>
<td>214</td>
</tr>
<tr>
<td>Manipulating Keyboard Settings</td>
<td>214</td>
</tr>
<tr>
<td>Controlling the Screen Saver</td>
<td>215</td>
</tr>
<tr>
<td>Manipulating Hosts and Access Control</td>
<td>215</td>
</tr>
<tr>
<td>Handling Events</td>
<td>216</td>
</tr>
<tr>
<td>Enabling and Disabling Synchronization</td>
<td>216</td>
</tr>
<tr>
<td>Using Default Error Handling</td>
<td>217</td>
</tr>
<tr>
<td>Communicating with Window Managers</td>
<td>217</td>
</tr>
<tr>
<td>Manipulating Keyboard Event Functions</td>
<td>218</td>
</tr>
<tr>
<td>Manipulating Regions</td>
<td>219</td>
</tr>
<tr>
<td>Using Cut and Paste Buffers</td>
<td>220</td>
</tr>
<tr>
<td>Querying Visual Types</td>
<td>220</td>
</tr>
<tr>
<td>Manipulating Images</td>
<td>221</td>
</tr>
<tr>
<td>Manipulating Bitmaps</td>
<td>221</td>
</tr>
<tr>
<td>Using the Resource Manager</td>
<td>221</td>
</tr>
<tr>
<td>Manipulating Display Functions</td>
<td>222</td>
</tr>
<tr>
<td>Extension Routines</td>
<td>225</td>
</tr>
<tr>
<td>MIT Extensions to X</td>
<td>225</td>
</tr>
<tr>
<td>Associate Table Functions</td>
<td>226</td>
</tr>
<tr>
<td>Miscellaneous Utility Routines</td>
<td>227</td>
</tr>
<tr>
<td>X Authorization Routines</td>
<td>230</td>
</tr>
<tr>
<td>X Intrinsics Routines</td>
<td>230</td>
</tr>
<tr>
<td>Athena Widget Support</td>
<td>240</td>
</tr>
<tr>
<td>Extension Routines</td>
<td>242</td>
</tr>
<tr>
<td>MIT Extensions to X</td>
<td>243</td>
</tr>
<tr>
<td>Associate Table Functions</td>
<td>243</td>
</tr>
<tr>
<td>Miscellaneous Utility Routines</td>
<td>243</td>
</tr>
<tr>
<td>X Authorization Routines</td>
<td>243</td>
</tr>
<tr>
<td>X Window System Toolkit</td>
<td>243</td>
</tr>
<tr>
<td>Application Resources</td>
<td>245</td>
</tr>
<tr>
<td>Athena Widget Set</td>
<td>246</td>
</tr>
</tbody>
</table>

## Chapter 7. Kerberos Authentication System 253

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authentication Server</td>
<td>253</td>
</tr>
<tr>
<td>Name Structures</td>
<td>253</td>
</tr>
<tr>
<td>Tickets and Authenticators</td>
<td>254</td>
</tr>
<tr>
<td>Communicating with the Authentication Server</td>
<td>254</td>
</tr>
<tr>
<td>Ticket-Granting Server</td>
<td>255</td>
</tr>
<tr>
<td>Accessing a Service</td>
<td>255</td>
</tr>
<tr>
<td>Kerberos Database</td>
<td>256</td>
</tr>
<tr>
<td>Administration Server</td>
<td>256</td>
</tr>
<tr>
<td>Kerberos C Language Applications Library</td>
<td>257</td>
</tr>
<tr>
<td>Kerberos Routines Reference</td>
<td>258</td>
</tr>
<tr>
<td>Client Commands</td>
<td>258</td>
</tr>
<tr>
<td>Applications</td>
<td>259</td>
</tr>
<tr>
<td>Kerberos Routines</td>
<td>259</td>
</tr>
<tr>
<td>krb_get_cred()</td>
<td>259</td>
</tr>
<tr>
<td>krb_knto1n()</td>
<td>259</td>
</tr>
<tr>
<td>krb_mk_err()</td>
<td>260</td>
</tr>
<tr>
<td>krb_mk_priv()</td>
<td>260</td>
</tr>
<tr>
<td>krb_mk_req()</td>
<td>261</td>
</tr>
<tr>
<td>krb_mk_safe()</td>
<td>261</td>
</tr>
<tr>
<td>krb_rd_err()</td>
<td>262</td>
</tr>
<tr>
<td>krb_rd_priv()</td>
<td>263</td>
</tr>
<tr>
<td>krb_rd_req()</td>
<td>264</td>
</tr>
<tr>
<td>krb_rd_safe()</td>
<td>265</td>
</tr>
<tr>
<td>krb_recvauth()</td>
<td>265</td>
</tr>
<tr>
<td>krb_sendauth()</td>
<td>266</td>
</tr>
<tr>
<td>Sample Kerberos Programs</td>
<td>268</td>
</tr>
<tr>
<td>Kerberos Client</td>
<td>268</td>
</tr>
<tr>
<td>Kerberos Server</td>
<td>270</td>
</tr>
</tbody>
</table>

## Chapter 8. SNMP Agent Distributed Program Interface 275

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNMP Agents and Subagents</td>
<td>275</td>
</tr>
<tr>
<td>Processing DPI Requests</td>
<td>276</td>
</tr>
<tr>
<td>Processing a GET Request</td>
<td>277</td>
</tr>
<tr>
<td>Processing a SET Request</td>
<td>277</td>
</tr>
<tr>
<td>Processing a GET_NEXT Request</td>
<td>277</td>
</tr>
<tr>
<td>Processing a REGISTER Request</td>
<td>278</td>
</tr>
<tr>
<td>Processing a TRAP Request</td>
<td>278</td>
</tr>
<tr>
<td>Compiling and Linking</td>
<td>278</td>
</tr>
<tr>
<td>SNMP DPI Reference</td>
<td>279</td>
</tr>
<tr>
<td>DPI Library Routines</td>
<td>279</td>
</tr>
<tr>
<td>DPIdebug()</td>
<td>279</td>
</tr>
<tr>
<td>fDPIparse()</td>
<td>280</td>
</tr>
<tr>
<td>mkDPIlist()</td>
<td>280</td>
</tr>
<tr>
<td>mkDPIregister()</td>
<td>281</td>
</tr>
<tr>
<td>mkDPIresponse()</td>
<td>281</td>
</tr>
<tr>
<td>mkDPIset()</td>
<td>282</td>
</tr>
<tr>
<td>mkDPItrap()</td>
<td>283</td>
</tr>
<tr>
<td>mkDPItrape()</td>
<td>284</td>
</tr>
<tr>
<td>Example of an Extended Trap</td>
<td>284</td>
</tr>
<tr>
<td>pDPIpacket()</td>
<td>285</td>
</tr>
<tr>
<td>query_DPI_port()</td>
<td>286</td>
</tr>
</tbody>
</table>
Preface


This book contains information about the following application programming interfaces (API):
- C sockets
- Pascal
- Virtual Machine Communication Facility (VMCF)
- Remote Procedure Calls (RPCs)
- X Window System
- Kerberos Authentication System
- Simple Network Management Protocol (SNMP) agent distributed program interface
- Conversational Monitor System (CMS) command interface to the name server
- Simple Mail Transfer Protocol (SMTP)

The descriptive information in the chapters is supplemented with appendixes that contain sample programs and quick references.

For comments and suggestions about this book, use the Reader’s Comment Form located at the back of this book. This form gives instructions on submitting your comments by mail, by FAX, or by electronic mail.

Who Should Read This Book

This book is intended for users and programmers who are familiar with z/VM and the Control Program (CP) and the Conversational Monitor System (CMS) components. You should also be familiar with the C or Pascal programming language and the specific Application Programming Interface (API) that you are using.

What You Should Know before Reading This Book

Before using this book, you should be familiar with z/VM, CP, and CMS. In addition, TCP/IP Level 430 for VM should already be installed and customized for your network.

What This Book Contains

You should read this book when you want to use the application programming interfaces that are available in TCP/IP Level 430 for VM.

Chapter 1, “z/VM C Socket Application Programming Interface” on page 1 describes how to write, compile, and run programs that use the z/VM C socket API.
Chapter 2, “TCP/UDP/IP API (Pascal Language)” on page 43, describes how to use the Pascal language application program interface to write application programs for the TCP, UDP, and IP layers of the TCP/IP protocol suite.

Chapter 3, “Virtual Machine Communication Facility Interface” on page 95, describes how to communicate directly with the TCPIP virtual machine using Virtual Machine Communication Facility calls.

Chapter 4, “Inter-User Communication Vehicle Sockets” on page 117, describes how to use the (IUCV) socket. While not every C socket library function is provided, all of the basic operations necessary to communicate with other socket programs are present.

Chapter 5, “Remote Procedure Calls” on page 149, describes the Remote Procedure Call protocol, which permits remote execution of subroutines across a TCP/IP network.

Chapter 6, “X Window System Interface” on page 201, describes the X Window System and subroutines for TCP/IP Level 430 for VM.

Chapter 7, “Kerberos Authentication System” on page 253, describes the Kerberos Authentication System and the routines for writing authentication programs.

Chapter 8, “SNMP Agent Distributed Program Interface” on page 275, describes the Simple Network Management Protocol (SNMP) agent distributed program interface (DPI).

Chapter 9, “SMTP Virtual Machine Interfaces” on page 309, describes the communication interfaces to the SMTP virtual machine.

Chapter 10, “Telnet Exits” on page 341, describes the Telnet server exits that provide CP command simulation, TN3270E printer management, and system access control when Telnet connections are established with your host.

Chapter 11, “FTP Server Exit” on page 345, describes the FTP Server Exit.

Appendix A, “TCPLOAD EXEC” on page 353, provides information about using the TCPLOAD EXEC to generate an executable module from a compiled program.

Appendix B, “Pascal Return Codes” on page 357, lists the system return codes as they apply to Pascal calls and provides their numeric value and description.

Appendix C, “C API System Return Codes” on page 361, lists the system return codes and provides their numeric value and a description.

Appendix D, “Well-Known Port Assignments” on page 365, This appendix 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 ETC SERVICES file.

Appendix E, “Related Protocol Specifications” on page 369, lists the related protocol specifications concerning TCP/IP Level 430 for VM.

Appendix E, “Abbreviations and Acronyms” on page 373, lists and defined the abbreviations and acronyms used in this book.
How to Use This Book

Read this book to learn about the application programming interfaces.

How the Term “internet” Is Used in This Book

In this book, an internet is a logical collection of networks supported by routers, gateways, bridges, hosts, and various layers of protocols, which permit the network to function as a large, virtual network.

Note: The term “internet” is used as a generic term for a TCP/IP network, and should not be confused with the Internet, which consists of large national backbone networks (such as MILNET, NSFNet, and CREN) and a myriad of regional and local campus networks worldwide.

Where to Find More Information

Appendix F, “Abbreviations and Acronyms” on page 373, lists the abbreviations and acronyms that are used throughout this book.

The “Glossary” on page 381, defines terms used throughout this book that are associated with TCP/IP communication in an internet environment.

For more information about related publications, see the books listed in the “Bibliography” on page 399.

PDF Links to Other Books

The PDF version of this book provides links to other IBM books by file name. The name of the PDF file for an IBM book is unique and identifies the book and its edition. The book links provided in this book are for the editions (PDF file names) that were current when this PDF file was generated. Newer editions of some books (with different file names) may exist. A PDF link from this book to another book works only when a PDF file with the requested file name resides in the same directory as this book.

Service Information

The IBM Software Support Center provides you with telephone assistance in problem diagnosis and resolution. You can call the IBM Software Support Center at anytime; you will receive a return call within eight business hours (Monday—Friday, 8:00 a.m.—5:00 p.m., local customer time). The number to call is: 1-800-237-5511.

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

Understanding Syntax Diagrams

This section describes how to read the syntax diagrams in this book.

Getting Started: To read a syntax diagram, follow the path of the line. Read from left to right and top to bottom.

• The ➤➤ symbol indicates the beginning of a syntax diagram.
• The ➞ symbol, at the end of a line, indicates that the syntax diagram continues on the next line.
• The ▶ symbol, at the beginning of a line, indicates that a syntax diagram continues from the previous line.
• The ➞ symbol indicates the end of a syntax diagram.

Syntax items (for example, a keyword or variable) may be:
• Directly on the line (required)
• Above the line (default)
• Below the line (optional).

### Syntax Diagram Description

**Abbreviations:**

Uppercase letters denote the shortest acceptable abbreviation. If an item appears entirely in uppercase letters, it cannot be abbreviated.

You can type the item in uppercase letters, lowercase letters, or any combination.

In this example, you can enter KEYWO, KEYWOR, or KEYWORD in any combination of uppercase and lowercase letters.

**Symbols:**

You must code these symbols exactly as they appear in the syntax diagram.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>:</td>
<td>Colon</td>
</tr>
<tr>
<td>,</td>
<td>Comma</td>
</tr>
<tr>
<td>=</td>
<td>Equal Sign</td>
</tr>
<tr>
<td>-</td>
<td>Hyphen</td>
</tr>
<tr>
<td>0</td>
<td>Parentheses</td>
</tr>
<tr>
<td>.</td>
<td>Period</td>
</tr>
</tbody>
</table>

**Variables:**

Highlighted lowercase items (*like this*) denote variables.

In this example, *var_name* represents a variable you must specify when you code the KEYWORD command.
**Syntax Diagram Description**

<table>
<thead>
<tr>
<th>Repetition:</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>An arrow returning to the left means that the item can be repeated.</td>
<td>![repeat]</td>
</tr>
<tr>
<td>A character within the arrow means you must separate repeated items with that character.</td>
<td>![repeat]</td>
</tr>
<tr>
<td>A footnote (1) by the arrow references a limit that tells how many times the item can be repeated.</td>
<td>![repeat]</td>
</tr>
</tbody>
</table>

### Notes:
1. Specify repeat up to 5 times.

<table>
<thead>
<tr>
<th>Required Choices:</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>When two or more items are in a stack and one of them is on the line, you must specify one item.</td>
<td>![A B C]</td>
</tr>
<tr>
<td>In this example, you must choose A, B, or C.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optional Choice:</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>When an item is below the line, the item is optional. In this example, you can choose A or nothing at all.</td>
<td>![A]</td>
</tr>
<tr>
<td>When two or more items are in a stack below the line, all of them are optional. In this example, you can choose A, B, C, or nothing at all.</td>
<td>![A B C]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Defaults:</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defaults are above the line. The system uses the default unless you override it. You can override the default by coding an option from the stack below the line.</td>
<td>![A B C]</td>
</tr>
<tr>
<td>In this example, A is the default. You can override A by choosing B or C.</td>
<td></td>
</tr>
</tbody>
</table>
Syntax Diagram Description

Repeatable Choices:

A stack of items followed by an arrow returning to the left means that you can select more than one item or, in some cases, repeat a single item.

In this example, you can choose any combination of A, B, or C.

Syntax Fragments:

Some diagrams, because of their length, must fragment the syntax. The fragment name appears between vertical bars in the diagram. The expanded fragment appears in the diagram after a heading with the same fragment name.

In this example, the fragment is named “A Fragment.”

How Numbers Are Used in This Book

In this book, numbers over four digits are represented in metric style. A space is used rather than a comma to separate groups of three digits. For example, the number sixteen thousand, one hundred forty-seven is written 16 147.

How to Send Your Comments to IBM

Your feedback is important in helping us to provide the most accurate and high-quality information. If you have comments about this book or any other VM documentation, send your comments to us using one of the following methods. Be sure to include the name of the book, the publication number (including the suffix), and the page, section title, or topic you are commenting on.

• Visit the z/VM web site at:

  There you will find the feedback page where you can enter and submit your comments.

• Send your comments by electronic mail to one of the following addresses:

  Internet: vmpub@us.ibm.com

  IBMLink™: GDLVME(PUBRCF)

• Fill out the Readers’ Comments form at the back of this book and return it by mail, by fax (1–607–752–2327), or by giving it to an IBM representative. If the form is missing, you can mail your comments to the following address:

  IBM Corporation
  Information Development
  Department G60G
  1701 North Street
  Endicott, New York 13760-5553
  USA
Summary of Changes

This section describes the technical changes made in this edition of the book and in previous editions. For your convenience, the changes made in this edition are identified in the text by a vertical bar (|) in the left margin. This edition may also include minor corrections and editorial changes that are not identified.


This edition contains updates for the General Availability of TCP/IP Level 430 as well as miscellaneous service updates.

First Edition for z/VM Version 4 (October 2001)

This edition contains updates for the General Availability of TCP/IP Level 420.

z/VM C Socket API

A new z/VM C sockets library within Language Environment provides C socket support for new or existing applications that use Language Environment services. z/VM C sockets are intended as replacements for the existing VM TCP/IP C sockets. Although TCP/IP C sockets are still supported for compatibility, z/VM C sockets is the preferred API. Existing TCP/IP C socket applications may need to recompile to use the z/VM C socket functions, but no source code changes are required.

The z/VM C socket support includes:

- New z/VM C socket function calls. Chapter 1, “z/VM C Socket Application Programming Interface” on page 1 describes how to write, compile, and run programs that use the z/VM C sockets library. The z/VM C socket functions are documented in the z/VM: Language Environment 1.8 C Run-Time Library Reference (This is a new book, which replaces the C for VM/ESA: Library Reference.)
- A new version of the TCP/IP for z/VM Remote Procedure Call (RPC) function library, VMRPC, to be used in conjunction with the z/VM C sockets. How to use this library is documented in Chapter 5, “Remote Procedure Calls” on page 149.

First Edition for z/VM Version 3 (February 2001)

This edition contains updates for the General Availability of TCP/IP Level 3A0.

IP Multicast

Support for IP Multicast has been added for the C sockets application program interfaces getssockopt and setsockopt.

Other Changes

- Miscellaneous service updates were added since the previous release.
- XCLIENT sample programs were removed.
- The chapter on NCS was removed. The product is now unsupported.
Chapter 1. z/VM C Socket Application Programming Interface

This chapter describes the z/VM C socket application programming interface (API). z/VM C sockets are C Language functions that closely correspond to the sockets used by UNIX applications that use the Berkeley Software Distribution (BSD) 4.4.

z/VM C sockets are intended as replacements for VM TCP/IP C sockets (formerly documented in this chapter). Although TCP/IP C sockets are still supported for compatibility, the z/VM C socket API is preferred.

This chapter describes how to write, compile, and run applications that use z/VM C sockets. Existing applications that use the VM TCP/IP C sockets library may continue to do so without any modification. To use the z/VM C socket functions, existing TCP/IP C socket applications may need to be recompiled, but no source changes are required. Instructions are provided in this chapter.

Notes:
1. To run programs that use z/VM C sockets, you must have Language Environment (supplied with z/VM) installed on your system. Language Environment provides header files and the object code and run-time library for the z/VM C socket functions.
   To compile programs that use the z/VM C socket API, you also need the IBM C for VM/ESA (C/VM™) Compiler, Version 3 (5654-033) Release 1.
   For specific program requirements, see the z/VM: General Information manual.
2. This chapter provides a guide to using the z/VM C socket API. For complete reference information on the z/VM C socket functions, see the z/VM: Language Environment 1.8 C Run-Time Library Reference.

This chapter contains the following sections:
- “TCP/IP Network Communication” on page 2 defines some of the basic networking terms.
- “What is a Socket?” on page 3 provides an overview of socket programming concepts.
- “Client/Server Conversation” on page 8 shows how a client and server use sockets to exchange information.
- “Network Application Example” on page 13 shows how sockets are used in a network application program.
- “z/VM C Socket Implementation” on page 19 explains how z/VM has implemented the support for C sockets. This section also explains the incompatibilities between z/VM C sockets and VM TCP/IP C sockets.
- “Compiling and Linking a Sockets Program” on page 24 describes how to compile and link programs to use the z/VM C sockets library.
- “Running a Sockets Program” on page 31 describes how to run programs that use the z/VM C sockets library.
- “C Sockets Quick Reference” on page 35 lists the z/VM C socket calls.
- “TCP Client Program” on page 36 shows an example of a TCP client program using z/VM C sockets.
- “TCP Server Program” on page 38 shows an example of a TCP server program using z/VM C sockets.
TCP/IP Network Communication

Network communication, or “internetworking”, defines a set of protocols that allow application programs to talk with each other without regard to the hardware and operating systems where they are run. Internetworking allows application programs to communicate independently of their physical network connections.

TCP/IP is an internetworking technology and is named after its two main protocols: Transmission Control Protocol (TCP), and Internet Protocol (IP). You should also be familiar with the following basic internetworking terms:

- **client**: A process that requests services on the network.
- **server**: A process that responds to a request for service from a client.
- **datagram**: A basic unit of information, consisting of one or more data packets, which are passed across an internet at the transport level.
- **packet**: The unit or block of a data transaction between a computer and its network. A packet usually contains a network header, at least one high-level protocol header, and data blocks. Generally, the format of data blocks does not affect how packets are handled. Packets are the exchange medium used at the Internetwork layer to send data through the network.

Transport Protocols

There are two general types of transport protocols:

- **A connectionless protocol** treats each datagram as independent from all others. Each datagram must contain all the information required for its delivery. An example of such a protocol is User Datagram Protocol (UDP). UDP is a datagram-level protocol built directly on the IP layer and used for application-to-application programs on a TCP/IP host. UDP does not guarantee data delivery, and is therefore considered unreliable. Application programs that require reliable delivery of streams of data should use TCP.
- **A connection-oriented protocol** requires that hosts establish a logical connection with each other before communication can take place. This connection is sometimes called a “virtual circuit”, although the actual data flow uses a packet-switching network. A connection-oriented exchange includes three phases:
  1. Start the connection.
  2. Transfer data.
  3. End the connection.

An example of such a protocol is Transmission Control Protocol (TCP). TCP provides a reliable vehicle for delivering packets between hosts on an internet. TCP breaks a stream of data into datagrams, sends each one individually using IP, and reassembles the datagrams at the destination node. If any datagrams are lost or damaged during transmission, TCP detects this and re-sends the missing or damaged datagrams. The data stream that is received is therefore a reliable copy of the original.
What is a Socket?

A socket can be thought of as an endpoint in a two-way communication channel. Socket routines create the communication channel, and the channel carries data between application programs either locally or over networks. Each socket open by a process — like any open file in a POSIX process — has a unique (within the process) number associated with it called a “file descriptor”, an integer that designates a socket and allows the application program to refer to it when needed.

Using an electrical analogy, you can think of the communication channel as the electrical wire with its plug and the port, or socket, as the electrical socket or outlet, as shown in Figure 1.

Figure 1 shows many application programs running on a client and many application programs on a server. When the client starts a socket call, a socket connection is made between an application on the client and an application on the server.

Another analogy used to describe socket communication is a telephone conversation. Dialing a phone number from your telephone is similar to starting a socket connection. The telephone switching unit knows where to logically make the correct switch to complete the call at the remote location. During your telephone conversation, this connection is present and information is exchanged. After you hang up, the connection is broken and you must start it again. The client uses the connect() function call to start the logical switch mechanism to connect to the server.

User processes ask the sockets library to create a socket when one is needed. The sockets library returns an integer, the file descriptor that the application uses every time it wants to refer to that socket.
Sockets perform in many respects like UNIX files or devices, so they can be used with such traditional operations as read() or write(). For example, after two application programs create sockets and open a connection between them, one program can use write() to send a stream of data, and the other can use read() to receive it. Because each file or socket has a unique descriptor, the system knows exactly where to send and to receive the data.

Address Families

The z/VM C socket API supports three address families (also called domains):

AF_INET AF_INET (internet domain) sockets provide a means of communicating between application programs that are on different systems using the TCP and UDP transport protocols provided by a TCP/IP product. This address family supports both stream and datagram sockets. TCP/IP for z/VM must be configured for you to be able to use this address family.

AF_IUCV AF_IUCV (VM IUCV) sockets provide communication between processes on a single VM system, or on a group of systems that share IUCV connectivity. VM IUCV sockets allow interprocess communication within VM independent of TCP/IP. The AF_IUCV domain supports only stream sockets.

AF_UNIX AF_UNIX sockets (also called local sockets) provide communication between processes on a single VM system, or on a group of systems that share a single Byte File System (BFS) server. UNIX domain sockets allow interprocess communication within VM independent of TCP/IP. On z/VM, the AF_UNIX domain supports only stream sockets.

The primary difference between VM IUCV sockets and UNIX sockets is how partners are identified (for example, how they are named).

Socket Types

The z/VM C socket API provides application programs with an interface that hides the details of the physical network. The API supports stream sockets, datagram sockets, and raw sockets, each providing different services for application programs. Stream and datagram sockets interface to the network layer protocols, and raw sockets interface to the network interface layers. You choose the most appropriate interface for an application.

Stream Sockets

The stream sockets interface provides a connection-oriented service. After the partner applications connect, the data sent on stream sockets acts like a stream of information. There are no boundaries between data, so communicating processes must agree on their own mechanism to distinguish information. For example, the process sending information could first send the length of the data, followed by the data itself. The process receiving information reads the length and then loops, reading data until all of it has been transferred. Stream sockets guarantee delivery of the data in the order it was sent and without duplication. The stream socket interface provides a reliable connection-oriented service. Data is sent without errors or duplication and is received in the same order as it is sent. Flow control is built in, to avoid data overruns. No boundaries are imposed on the data; the data is considered to be a stream of bytes.

Stream sockets are the most-commonly used, because the burden of transferring the data reliably is handled by the system rather than by the application.
Datagram Sockets
The datagram socket interface provides a connectionless service. Datagrams are sent as independent packets. The service provides no guarantees; datagrams can be lost, duplicated, and can arrive out of order. The size of a datagram is limited to the size that can be sent in a single transaction.

Raw Sockets
The raw socket interface provides direct access to lower layer protocols, such as the AF_INET Internet Protocol (IP) and Internet Control Message Protocol (ICMP). You can use raw sockets to test new protocol implementations. You can extend the socket interface; you can define new socket types to provide additional services. Because they isolate you from the communication functions of the different protocol layers, socket interfaces are largely independent of the underlying network. In the AF_INET address family, stream sockets interface to TCP, datagram sockets interface to UDP, and raw sockets interface to ICMP and IP.

Guidelines for Using Socket Types
The following criteria will help you choose the appropriate socket type for an application program.

If you are communicating with an existing application program, you must use the same protocol as the existing application program. For example, if you want to communicate with an application that uses TCP, you must use AF_INET stream sockets. For new application programs, you should consider the following factors:

- **Reliability**: Stream sockets provide the most reliable connection. Datagram sockets are unreliable, because datagrams can be discarded, corrupted, or duplicated during transmission. This may be acceptable if the application program does not require reliability, or if the application program implements the reliability on top of the sockets interface. The trade-off is the improved performance available with datagram sockets.

- **Performance**: The overhead associated with reliability, flow control, packet reassembly, and connection maintenance degrade the performance of stream sockets in comparison with datagram sockets.

- **Data transfer**: Datagram sockets impose a limit on the amount of data transferred in a single transaction. If you send less than 2048 bytes at a time, use datagram sockets. As the amount of data in a single transaction increases, use stream sockets.

If you are writing a new protocol on top of IP, or wish to use the ICMP protocol, then you must use raw sockets.

Domain-specific Socket Addresses
The following sections describe the different ways to address processes who communicate with each other using sockets.

**Address Families**
Each address family defines a different style of addressing. All hosts in the same address family use the same scheme for addressing socket endpoints. The AF_INET address family identifies processes by IP address and port number. The AF_UNIX address family identifies processes by file name in the Byte File System. The AF_IUCV address family identifies processes by VM user ID and application name.

**Socket Address**
A socket address is defined by the `sockaddr` structure in the `sys/socket.h` header file. The structure has three fields, as shown in the following example:
struct sockaddr {
    unsigned char sa_len;
    unsigned char sa_family;
    char sa_data[14]; /* variable length data */
};

The `sa_len` field contains the length of the entire `sockaddr` structure, in bytes. The `sa_family` field contains a value identifying the address family. It is AF_INET for the internet domain, AF_UNIX for the UNIX domain, and AF_IUCV for the IUCV domain. The `sa_data` field is different for each address family. Each address family defines its own structure, which can be overlaid on the `sockaddr` structure. See "Addressing within the AF_INET Domain", "Addressing within the AF_UNIX Domain" on page 7, and "Addressing within the AF_IUCV Domain" on page 8.

### Addressing within the AF_INET Domain

Before discussing the contents of the AF_INET `sockaddr` structure, the following terms must be introduced.

**Internet Addresses:** Internet addresses are 32-bit quantities that represent a network interface. Every internet address within an administered AF_INET domain is unique. On the other hand, it is not necessary that every host have a single internet address; in fact, a host has as many internet addresses as it has network interfaces.

**Ports:** A port distinguishes between different application programs using the same AF_INET network interface. It is an additional qualifier used by the system software to get data to the correct application program. Physically, a port is a 16-bit integer. Some ports are reserved for particular application programs or protocols and are called well-known ports.

**Network byte order and host byte order:** Ports and addresses are always specified in calls to the socket functions using the network byte order convention. This convention is a method of sorting bytes that is independent of specific machine architectures. Host byte order, on the other hand, sorts bytes in the manner which is most natural to the host software and hardware. There are two common host byte order methods:

- **Little-endian** byte ordering places the least significant byte first. This method is used in Intel microprocessors, for example.
- **Big-endian** byte ordering places the most significant byte first. This method is used in IBM z/Architecture™ and S/390® mainframes and Motorola microprocessors, for example.

The network byte order is defined to always be big-endian, which may differ from the host byte order on a particular machine. Using network byte ordering for data exchanged between hosts allows hosts using different architectures to exchange address information without confusion because of byte ordering. The following C functions allow the application program to switch numbers easily back and forth between the host byte order and network byte order without having to first know what method is used for the host byte order:

- `htonl()` translates an unsigned long integer into network byte order.
- `htons()` translates an unsigned short integer into network byte order.
- `ntohl()` translates an unsigned long integer into host byte order.
- ` ntohs()` translates an unsigned short integer into host byte order.
See Figure 5 on page 14, Figure 7 on page 15, and Figure 8 on page 15 for examples of using the htons() call to put port numbers into network byte order.

**AF_INET addresses:** A socket address in the AF_INET address family is defined by the sockaddr_in structure, which is defined in the netinet/in.h header file:

```c
typedef unsigned long in_addr_t;
struct in_addr {
    in_addr_t s_addr;
};
struct sockaddr_in {
    unsigned char sin_len; /* length of sockaddr struct */
    unsigned char sin_family; /* addressing family */
    unsigned short sin_port; /* port number */
    struct in_addr sin_addr; /* IP address */
    unsigned char sin_zero[8]; /* unassigned */
};
```

The sin_len field is set to either 0 or sizeof(struct sockaddr_in) when providing a sockaddr_in structure to the sockets library. Both values are treated the same. When the sockets library provides a sockaddr_in structure to the application, the sin_len field is set to sizeof(struct sockaddr_in).

The sin_family field is set to AF_INET.

The sin_port field is set to the port to which the process is bound, in network byte order.

The sin_addr field is set to the internet address (IP address) of the interface to which the process is bound, in network byte order.

The sin_zero field is not used and must be set to all zeros.

**Addressing within the AF_UNIX Domain**

A socket address in the AF_UNIX address family is defined by the sockaddr_un structure, which is defined in the sys/un.h header file:

```c
struct sockaddr_un {
    unsigned char sun_len; /* length of sockaddr struct */
    unsigned char sun_family; /* addressing family */
    char sun_path[108]; /* file name */
};
```

When the application provides a sockaddr_un structure to the sockets library, the sun_len field should be set to either 0 or a value greater than or equal to SUN_LEN(&sa), where sa is the name of the sockaddr_un variable, but less than or equal to sizeof(struct sockaddr_un). The SUN_LEN() macro, which is defined in sys/un.h, evaluates to an expression which returns the total length of the used portion of the sockaddr_un structure, when sun_path has been filled in with a null-terminated file name. The length returned by SUN_LEN() does not include the terminating null character. If a 0 is specified for sun_len, the sockaddr length provided on the specific socket function call determines how long the path name is. If sun_len is nonzero, the lesser of sun_len and the provided length is used. In either case, if a null character appears in the string before the given length, the path name is considered to end there. When the sockets library provides a sockaddr_un structure to the application, the sun_len field is set to SUN_LEN(&sa)+1, where sa is the name of the sockaddr_un variable. This length thus includes the null byte which terminates the file name.

The sun_family field is set to AF_UNIX.
C Sockets API

The **sun_path** field contains the name of the file which represents the open socket. It need not be null delimited, although it is recommended that it is, so that the SUN_LEN() macro can be used. A file by this name will be created in the Byte File System by the bind() function call, and must exist there for the connect() function call to succeed. Because the Byte File System contains the file, the form of the path name should follow the POSIX conventions. Generally, an absolute path name (one that begins with a slash) should be specified, so that the client and the server can both use the same path name to identify the file. If an AF_UNIX socket is not yet bound when a client calls the connect() function, it will be bound to the null path name string (for example, the string ""). In this case, no file is created in the Byte File System.

For more information about the Byte File System, see the *z/VM: OpenExtensions User’s Guide*.

**Addressing within the AF_IUCV Domain**

A socket address in the AF_IUCV address family is defined by the **sockaddr_iucv** structure, which is defined in the **saiucv.h** header file:

```
struct sockaddr_iucv {
    unsigned char siucv_len; /* length of sockaddr struct */
    unsigned char siucv_family; /* addressing family */
    unsigned short siucv_port; /* port number */
    unsigned long siucv_addr; /* address */
    unsigned char siucv_nodeid[8]; /* nodeid to connect to */
    unsigned char siucv_userid[8]; /* userid to connect to */
    unsigned char siucv_name[8]; /* iucvname for connect */
};
```

The **siucv_len** field is set to either 0 or sizeof(struct sockaddr_iucv) when providing a sockaddr_iucv structure to the sockets library. Both values are treated the same. When the sockets library provides a sockaddr_iucv structure to the application, the **siucv_len** field is set to sizeof(struct sockaddr_iucv).

The **siucv_family** field is set to AF_IUCV.

The **siucv_port**, **siucv_addr**, and **siucv_nodeid** fields are reserved for future use. The **siucv_port** and **siucv_addr** fields must be zeroed. The **siucv_nodeid** field must be set to exactly eight blank characters.

The **siucv_userid** field is set to the VM user ID of the application which owns the address. This field must be eight characters long, padded with blanks on the right. It cannot contain the null character.

The **siucv_name** field is set to the application name by which the socket is known. A server advertises a particular application name, and this is the name used by the client to connect to the server. The recommended form of the name contains eight characters, padded with blanks to the right.

For more information about IUCV, see the *z/VM: CMS Application Development Guide for Assembler*.

**Client/Server Conversation**

The client and server exchange data using a number of socket functions. They can send data using send(), sendto(), sendmsg(), write(), or writev(). They can receive data using recv(), recvfrom(), recvmsg(), read(), or readv(). The following is an example of the send() and recv() calls:
send(s, addr_of_data, len_of_data, 0);
recv(s, addr_of_buffer, len_of_buffer, 0);

The send() and recv() functions specify the socket \( s \) on which to communicate, the address in memory of the buffer that contains, or will contain, the data \((addr\_of\_data, addr\_of\_buffer)\), the size of this buffer \((len\_of\_data, len\_of\_buffer)\), and a flag that tells how the data is to be sent. Using the flag 0 tells TCP/IP to transfer the data normally. The server uses the socket that is returned from the accept() call. The client uses the socket that is returned from the socket() call.

These functions return the amount of data that was either sent or received. Because stream sockets send and receive information in streams of data, it can take more than one call to send() or recv() to transfer all the data. It is up to the client and server to agree on some mechanism of signaling that all the data has been transferred.

When the conversation is over, both the client and server call the close() function to end the connection. The close() function deallocates the socket, freeing its space in the table of connections. To end a connection with a specific client, the server closes the socket returned by accept(). If the server closes its original socket, the “listening” socket, it can no longer accept new connections, but it can still converse with the clients it is connected to. The following is an example of the close() call:

close(s);

**Server Perspective for AF_INET**

Before the server can accept any connections with clients, it must register itself with TCP/IP and “listen” for client requests on a specific port.

**socket()**
The server must first allocate a socket. This socket provides an endpoint that clients connect to.

Opened sockets are identified by file descriptors, like any open files in a POSIX environment. The programmer calls the socket() function to allocate a new socket, as shown in the following example:

socket(AF_INET, SOCK_STREAM, 0);

The socket() function requires the address family (AF_INET), the type of socket (SOCK_STREAM), and the particular networking protocol to use (when 0 is specified, the system automatically uses the appropriate protocol for the specified socket type). A new socket is allocated and its file descriptor is returned.

**bind()**
At this point, an entry in the table of communications has been reserved for your application program. However, the socket has no port or IP address associated with it until you use the bind() function, which requires three parameters:

- The socket the server was just given
- The number of the port on which the server wishes to provide its service
- The IP address of the network connection on which the server is listening (to understand what is meant by “listening”, see “listen()” on page 10).

The server puts the port number and IP address into a sockaddr_in structure, passing it and the socket file descriptor to the bind() function. For example:
struct sockaddr_in sa;
;
bind(s, (struct sockaddr *) &sa, sizeof sa);

listen()
After the bind, the server has specified a particular IP address and port. Now it must notify the system that it intends to listen for connections on this socket. The listen() function puts the socket into passive open mode and allocates a backlog queue of pending connections. In passive open mode, the socket is open for clients to contact. For example:
listen(s, backlog_number);

The server gives the file descriptor of the socket on which it will be listening and the number of requests that can be queued (the backlog_number). If a connection request arrives before the server can process it, the request is queued until the server is ready.

accept()
Up to this point, the server has allocated a socket, bound the socket to an IP address and port, and issued a passive open. The next step is for the server to actually establish a connection with a client. The accept() call blocks the server until a connection request arrives, or, if there are connection requests in the backlog queue, until a connection is established with the first client in the queue. The following is an example of the accept() call:

struct sockaddr_in sa;
int addrlen;...

client_sock = accept(s, (struct sockaddr *) &sa, &addrlen);

The server passes the file descriptor of its socket to the accept() call. When the connection is established, the accept() call creates a new socket representing the connection with the client, and returns its file descriptor. When the server wishes to communicate with the client or end the connection, it uses the file descriptor of this new socket, client_sock. The original socket s is now ready to accept connections with other clients. The original socket is still allocated, bound, and opened passively. To accept another connection, the server calls accept() again. By repeatedly calling accept(), the server can establish many connections simultaneously.

select()
The server is now ready to start handling requests on this port from any client with the server’s IP address and port number. If the server handles just one client at a time, it can just start sending or receiving data. A server is not limited to one active socket, though. Often a server processes requests from several clients at the same time, and additionally listens for new clients wanting to establish connections. For maximum performance, such a server should either create a new thread to handle each client request, or set all of its sockets to “nonblocking” mode, so that a delay in handling one client request does not affect other client requests. Using nonblocking mode allows a single-threaded server to operate only on those sockets that are ready for communication. The select() call allows an application program to test for activity on a group of sockets.

Note: The select() function can also be used with other descriptors, such as file descriptors, pipes, or character special files such as the tty.

To allow you to test any number of sockets with just a single call to select(), place the file descriptors of the sockets to test into a “bit set”, passing the bit set to the
select() call. A bit set is a string of bits where each possible member of the set is on or off. If the member’s bit is off, the member is not in the set. If the member’s bit is on, the member is in the set. If the socket with file descriptor 3 is a member of a bit set, then the bit that represents it is on.

The following macros are provided to manipulate the bit sets:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD_ZERO</td>
<td>Clears the whole bit set</td>
</tr>
<tr>
<td>FD_SET</td>
<td>Sets the bit corresponding to a particular file descriptor</td>
</tr>
<tr>
<td>FD_CLR</td>
<td>Clears the bit corresponding to a particular file descriptor</td>
</tr>
<tr>
<td>FD_ISSET</td>
<td>Tests whether the bit corresponding to a particular file descriptor is set or cleared</td>
</tr>
</tbody>
</table>

To be active, a socket must be ready for reading data or for writing data, or an exceptional condition must have occurred. Therefore, the server specifies three bit sets of file descriptors in its call to the select() function: one bit set for file descriptors on which to receive data, another for file descriptors on which to write data, and one for sockets with exception conditions. The select() call tests each file descriptor in each bit set for activity and returns only those file descriptors that are active.

A server that processes many clients simultaneously can be written so that it processes only those clients that are ready for activity.

**Client Perspective for AF_INET**

The client first issues the socket() function call to allocate a socket on which to communicate:

```c
socket(AF_INET, SOCK_STREAM, 0);
```

To connect to the server, the client places the port number and the IP address of the server into a `sockaddr_in` structure, like the one used by the server of its bind() call. If the client does not know the server’s IP address, but does know the server’s host name, the gethostbyname() function may be called to translate the host name into its IP address. The client then calls connect():

```c
struct sockaddr_in sa;
connect(s, (struct sockaddr_in *) &sa, sizeof sa);
```

When the connection is established, the client uses its socket to communicate with the server.

**Typical TCP Socket Session**

You can use TCP sockets for both passive (server) and active (client) processes. Whereas some functions are necessary for both types, some are role-specific. After you make a connection, it exists until you close the socket. During the connection, data is either delivered or an error code is returned by TCP/IP.

See [Figure 2 on page 12](#) for the general sequence of calls to be followed for most socket routines using TCP, or stream sockets.
Typical UDP Socket Session

UDP socket processes, unlike TCP socket processes, are not clearly distinguished by server and client roles. The distinction is between connected and unconnected sockets. An unconnected socket can be used to communicate with any host; but a connected socket, because it has a dedicated destination, can send data to, and receive data from, only one host.

Both connected and unconnected sockets send their data over the network without verification. Consequently, after a packet has been accepted by the UDP interface, the arrival and integrity of the packet cannot be guaranteed.

See Figure 3 for the general sequence of calls to be followed for most socket routines using UDP, or datagram, sockets.
Locating the Server’s Port

In the client/server model, the server provides a resource by listening for clients on a particular port. Such application programs as FTP, SMTP, and Telnet listen on a well-known port, a port reserved for use by a specific application program or protocol. However, for your own client/server application programs, you need a method of assigning port numbers to represent the services you intend to provide. One general method of defining services and their ports is to enter them into the ETC SERVICES file. The programmer uses the getservbyname() function to determine the port for a particular service. If the port number for a particular service changes, only the ETC SERVICES file must be modified.

Note: TCP/IP for z/VM is shipped with an ETC SERVICES file containing such well-known services as FTP, SMTP, and Telnet.

Network Application Example

The following steps illustrates using socket functions in an AF_INET network application program.
C Sockets API

Note: Error checking has been omitted from the examples. Error checking is very important, and has been omitted only to avoid complicating the examples.

1. First, an application program must open a socket using the socket() call, as shown in Figure 4.

```c
int s;
::
s = socket(AF_INET, SOCK_STREAM, 0);
```

**Figure 4. An Application Using socket()**

This example allocates a socket `s` in the AF_INET address family, with socket type SOCK_STREAM and protocol 0. Passing 0 for the protocol chooses the default, which for the AF_INET domain and SOCK_STREAM type is IPPROTO_TCP. The supported values for the socket domain, type, and protocol are defined in the *netinet/in.h* header file.

If successful, the socket() call returns a positive integer called a file descriptor that is used in subsequent function calls to identify the socket.

2. After an application program creates a socket, it can explicitly bind a unique address to the socket, as shown in Figure 5.

```c
int rc;
int s;
struct sockaddr_in myname;

/* Clear the structure to be sure that the sin_len and */
/* sin_zero fields are clear */
memset(&myname, 0, sizeof myname);
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = INADDR_ANY; /* any interface */
myname.sin_port = htons(5001);
::
rc = bind(s, (struct sockaddr *) &myname, sizeof myname);
```

**Figure 5. An Application Using bind()**

This example binds the socket with file descriptor `s` to port 5001, allowing it to accept connections from any interface available to the host in the internet domain. Servers must bind to an address and port to become accessible to the network. Also shown in this example is a handy utility routine called htons(), which takes a short integer (like a port number) in host byte order and returns it in network byte order.

3. After binding to a socket, a server that uses stream sockets must indicate its readiness to accept connections from clients. The server does this with the listen() call, as shown in Figure 6.

```c
int s;
int rc;
::
rc = listen(s, 5);
```

**Figure 6. An Application Using listen()**
This example tells TCP/IP that the server is ready to begin accepting connections, and that a maximum of five connection requests can be queued for the server. Additional requests are ignored.

4. Clients using stream sockets begin a connection request by calling connect(), as shown in Figure 7.

```
int s;
struct sockaddr_in servername;
int rc;
:
memset(&servername, 0, sizeof servername);
servername.sin_family = AF_INET;
servername.sin_addr.s_addr = inet_addr("129.5.24.1");
servername.sin_port = htons(5001);
:
rc = connect(s, (struct sockaddr *) &servername, sizeof servername);
```

**Figure 7. An Application Using connect()**

This example attempts to connect the socket with file descriptor `s` to the server with an address specified in the `servername` variable. This could be the server that was used in Figure 5 on page 14. After a successful return, the socket with file descriptor `s` is associated with the connection to the server. This example also uses another handy utility routine, `inet_addr()`, which takes an internet address in dotted-decimal form and returns it as a long integer in network byte order.

**Figure 8** shows another example of the connect() call. It uses the utility routine `gethostbyname()` to find the internet address of the host rather than using `inet_addr()` with a specific address.

```
int rc;
int s;
char *hostname = "jphhost.ibm.com";
struct sockaddr_in servername;
struct hostent *hp;

hp = gethostbyname(hostname);
:
/* Clear the structure to be sure that the sin_len and sin_zero fields are clear. */
memset(&servername, 0, sizeof servername);
servername.sin_family = AF_INET;
servername.sin_addr.s_addr = *(in_addr_t *) hp->h_addr;
servername.sin_port = htons(5001);
:
rc = connect(s, (struct sockaddr *) &servername, sizeof servername);
```

**Figure 8. A connect() Function Using gethostbyname()**

5. Servers using stream sockets accept a connection request with the accept() call, as shown in Figure 9 on page 16.
If connection requests are not pending on the socket with file descriptor \( s \), the `accept()` call blocks the server (unless \( s \) is in nonblocking mode). When a connection request is accepted, the socket, the name of the client, and length of the client name are returned, along with a file descriptor representing a new socket. The new socket is associated with the client that began the connection, and \( s \) is again available to accept new connections.

Clients and servers have many calls from which to choose for data transfer. The `send()` and `recv()`, `readv()` and `writev()`, and `read()` and `write()` calls can be used only on sockets that are in the connected state. The `sendto()` and `recvfrom()`, and `sendmsg()` and `recvmsg()` calls can be used at any time on datagram sockets. Figure 10 illustrates the use of `send()` and `recv()`.

This example shows an application program sending data on a connected socket and receiving data in response. The flags field can be used to specify additional options to `send()` or `recv()`, such as sending out-of-band data. (In this case no flags are being used, so 0 is passed.)

If the socket is not in a connected state, additional address information must be passed to `sendto()` and can be optionally returned from `recvfrom()`. An example is shown in Figure 11 on page 17.
The sendto(), recvfrom(), sendmsg(), and recvmsg() calls take additional parameters that allow the caller to specify the recipient of the data or to be notified of the sender of the data. Usually, sendto(), recvfrom(), sendmsg(), and recvmsg() are used for datagram sockets, and send() and recv() are used for stream sockets.

8. The writev(), readv(), sendmsg(), and recvmsg() calls provide the additional features of “scatter” and “gather” buffers, two related operations where data is received and stored in multiple buffers (scatter data), and then taken from multiple buffers and transmitted (gather data). The writev() and sendmsg() calls gather the data and send it. The readv() and recvmsg() calls receive data and scatter it into multiple buffers.

9. Applications can handle multiple file descriptors. In such situations, use the select() call to determine the file descriptors that have data to be read, those that are ready for data to be written, and those that have pending exceptional conditions. Figure 12 on page 18 is an example of how the select() call is used.

```c
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
struct sockaddr_in to;
struct sockaddr_in from;
int addrlen;
int s;...
memset(&to, 0, sizeof to);
to.sin_family = AF_INET;
to.sin_addr.s_addr = inet_addr("129.5.24.1");
to.sin_port = htons(5001);
bytes_sent = sendto(s, data_sent, sizeof data_sent, 0,
              (struct sockaddr *) &to, sizeof to);
;
addrlen = sizeof from; /* must be initialized */
bytes_received = recvfrom(s, data_received,
              sizeof data_received, 0, (struct sockaddr *) &from, &addrlen);
```

Figure 11. An Application Using sendto() and recvfrom()
In this example, the application program uses bit sets to indicate that the sockets are being tested for certain conditions and also indicates a timeout. If the timeout parameter is a null pointer, the select() call blocks until a socket becomes ready. If the timeout parameter is nonnull, select() waits up to this amount of time for at least one socket to become ready on the indicated conditions. This is useful for application programs servicing multiple connections that cannot afford to block, waiting for data on one connection.

In addition to select(), application programs can use the fcntl() or ioctl() calls to help perform asynchronous (nonblocking) socket operations. An example of the use of the ioctl() call is shown in Figure 13.

In this example, the socket with file descriptor $s$ is placed into nonblocking mode. When this file descriptor is passed as a parameter to calls that would block, such as recv() when data is not present, it causes the call to return with an error code, and the global $errno$ value is set to EWOULDBLOCK or EAGAIN. Setting the mode of the socket to be nonblocking allows an application program to continue processing without becoming blocked.
11. A socket with file descriptor s is deallocated with the close() call, as shown in [Figure 14].

```c
int rc;
int s;
rc = close(s);
```

Figure 14. An Application Using close()

z/VM C Socket Implementation

The following sections describe some important implementation details of the z/VM C socket API.

Header Files

Most of the socket header files used by the z/VM C sockets library are shipped with Language Environment. The only header file that is unique to the z/VM library is `saiucv.h`, which contains the `sockaddr` structure definition for AF_IUCV sockets.

_PROJE_SOCKETS Preprocessor Symbol

In general, the Language Environment header files are sensitive to whether the _PROJE_SOCKETS preprocessor symbol has been defined. In order to use the Language Environment header files for sockets programming, you must define the _PROJE_SOCKETS preprocessor symbol before you include any Language Environment header files. You can do this in your program by placing a statement similar to the following at the top of each source file:

```c
#define _OE_SOCKETS
```

Alternatively, you can cause the _PROJE_SOCKETS preprocessor symbol to be defined by the compiler, by using the -D option on the c89 command line. See "Compiling and Linking a Sockets Program" on page 26 for more information on compiling sockets programs.

Function Prototypes

Although they contain function prototypes for all of the POSIX.1 functions, the Language Environment header files do not contain prototypes for all of the socket functions. Specifically, when _PROJE_SOCKETS is defined, the following socket functions are available, but have no prototypes provided:

<table>
<thead>
<tr>
<th>Header File</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys/socket.h</td>
<td>accept(), bind(), connect(), getpeername(), getsockname(), getsockopt(), listen(), recv(), recvfrom(), recvmsg(), send(), sendmsg(), sendto(), setsockopt(), shutdown(), socket()</td>
</tr>
<tr>
<td>netdb.h</td>
<td>endhostent(), endnetent(), endprotoent(), endserverent(), getservent(), getservent(), resolve(), setprotoent(), setservent()</td>
</tr>
<tr>
<td>arpa/inet.h</td>
<td>inet_pton(), inet_pton()</td>
</tr>
<tr>
<td>sys/uio.h</td>
<td>readv(), writev()</td>
</tr>
</tbody>
</table>

Because the socket functions were designed to be useful for any networking interface, the types of the parameters declared for the functions do not always exactly match the types of the arguments provided. In one sense, therefore, it is convenient that prototypes are not always provided, as it reduces the number of possible compiler warning messages because of type mismatches. On the other
hand, socket programs may contain subtle bugs because of misunderstandings about the type definitions of the function parameters, so care should be taken when coding a function call. One way to ensure care is to use prototypes and to explicitly cast function arguments when necessary (and only when necessary). For example, the connect() function call accepts a pointer to a sockaddr structure, but the sockaddr structure is a generic structure not associated with any particular address family. A program using AF_INET sockets might provide a pointer to a sockaddr_in structure for this connect() parameter. Because these two pointer types are not compatible, an explicit cast should be used on the function call to convert the sockaddr_in pointer into a generic sockaddr pointer.

**Suppressing Function Prototypes**

If you are porting a large program that you know is coded correctly, you may at first receive a lot of type-mismatch compiler errors if the program is not coded to explicitly cast function arguments to their proper types. This is common when porting code from other systems, because not all systems provide function prototypes for the socket functions. To avoid correcting the function calls to perform the explicit cast operations, you can define the _NO_PROTO preprocessor symbol before including any header files. If the _NO_PROTO preprocessor symbol is defined before including any Language Environment or z/VM header file, function prototypes will be suppressed, or at least modified to omit type declarations for the function arguments. Use of this preprocessor symbol will avoid the compiler warning messages and make porting easier, but be aware that it may also obscure coding errors in the program.

If you wish to define the _NO_PROTO preprocessor symbol, you can do so by placing a statement similar to the following at the top of each source file of your program:

```c
#define _NO_PROTO
```

Alternatively, you can cause the _NO_PROTO preprocessor symbol to be defined by the compiler, by using the -D option on the c89 command line. See “Compiling and Linking a Sockets Program” on page 26 for more information on compiling sockets programs.

**Multithreading**

The z/VM C sockets library is a multithreading sockets library. This means that you can write programs to exploit the z/VM multithreading capabilities provided for POSIX programs, and still use socket functions without worrying about socket calls by one thread interfering with calls by another thread, or about the entire process being blocked just because one thread is blocked.

The z/VM C sockets library protects its internal data structures with mutexes, and uses thread-local data areas, where necessary, to ensure that socket calls by different threads can occur “concurrently”. The z/VM C sockets library is careful to never hold one of these internal mutexes when it might block for a substantial period of time, so multiple threads can use socket functions with as much concurrency as possible.

Some function calls in the z/VM C sockets library are not “primitive” socket function calls, however. For example, the gethostbyname() function call is really a procedure which tries to resolve a host name by reading data from local files and by communicating with Domain Name Servers in the network.
Multithreading versus Nonblocking Sockets

In a single-threaded program, a server that wants to handle concurrent requests from multiple clients usually sets all of its sockets to nonblocking mode, so that if a socket call on behalf of one client can not be processed immediately, other client requests are not delayed. In a multithreaded server, another approach is available. Instead of setting the sockets to be nonblocking, the server can create a separate thread to handle each client request. If a call to a socket function by one thread blocks, only that client request is affected; other threads are free to continue processing requests from other clients. Using multiple threads can therefore simplify the programming model, because each thread can concentrate on a single client without worrying about any other client. Either approach is available with the z/VM C sockets library.

Conflicts Between Socket Calls

When one thread of a multithreaded program is issuing a socket function call for a given socket, other threads are restricted from issuing certain socket function calls against that same socket. These restrictions are enforced by the TCP/IP service virtual machine. The following list describes the restrictions for each type of socket call:

- Multiple read-type calls (read(), recv(), recvfrom(), or recvmsg()) and multiple write-type calls (write(), writev(), send(), sendto(), or sendmsg()) for the same socket can be in progress simultaneously. The read-type calls are satisfied in the order they are issued, independently of the write-type calls. Similarly, the write-type calls are satisfied in the order they are issued, independently of the read-type calls.
- Multiple accept() calls for the same listening stream socket can be in progress simultaneously. They are satisfied in the order they are issued.
- Multiple select() calls referring to any combination of sockets (or other file descriptors) may be in progress simultaneously. When the state of a socket or other file descriptor changes, all active select() calls are checked; any that are then satisfied will return.
- Calls other than the read-type, write-type, accept(), and select() calls may not be in progress simultaneously for the same socket. For example, your program must wait for a write-type call to complete (or interrupt it) before issuing a close() call for the same socket.

If your program violates one of these restrictions, the function call that violates it will fail with an ECONFLICT error, except a close() call, which will fail with an EAGAIN error.

POSIX Signals and Thread Cancellation

The POSIX.1 standard greatly enhances the ANSI C Language definition by defining and guaranteeing certain aspects of signal processing. For example, many POSIX.1 function calls are defined to unblock, returning the EINTR error code, if a signal is delivered to a thread while it is blocked in a function call. POSIX.1 also defines what function calls a program may safely make while in a signal handler. Similarly, the (draft) POSIX.1c threading standard defines the conditions under which a thread may be “cancelled” by a call to the pthread_cancel() function, and what function calls are considered to be “cancellation points”. The intent of this section is to define these attributes for the z/VM C sockets library.

Any socket function call which blocks may return EINTR if interrupted by a signal. That is, if a signal is caught by a thread which is blocked in a call to a socket function, that socket function will unblock and return the EINTR error code when the signal handler returns. The z/VM C sockets library blocks signal delivery in
places when it is holding any internal mutexes, so signal delivery will occur only when the z/VM C sockets library can tolerate it. However, the following strict restriction does exist: **It is not supported for a signal handler to use longjmp() or siglongjmp() to exit from a signal handler.** In order for the z/VM C sockets library to properly recover from being interrupted, the signal handler must return, allowing the interrupted function call to resume from the point of interruption. This restriction only exists for signal handlers which might run because the thread was interrupted in the middle of a call to a socket function.

All z/VM C socket functions are async signal safe, and may thus be called without restriction from signal handlers.

All z/VM C socket functions are defined to possibly be thread cancellation points, as defined in the (draft) POSIX.1c threading standard, and no socket functions are defined to be async cancel safe. Any socket function which blocks will be a cancellation point. Although it is not supported by POSIX, it is safe for a cancellation cleanup handler to use longjmp() or siglongjmp() to exit, even if a function in the z/VM C sockets library was interrupted by the thread cancellation request.

**Note:** The difference between using longjmp() from a cancellation cleanup handler and using longjmp() from a signal handler is that in the case of cancellation, the z/VM C sockets library uses a cancellation cleanup handler of its own (which gets invoked before the application’s cleanup handler) to clean up outstanding socket activity.

**Sockets and Their Relationship to Other POSIX Functions**

The z/VM C sockets library allocates file descriptors for sockets from the same pool of numbers that CMS uses for other open files in the POSIX environment. Among other things, this means that non-socket-specific POSIX functions can be called for file descriptors allocated to sockets. For example, the fstat() function can be called to retrieve information about an open socket. In the case of a socket, the st_mode field of the stat structure returned will indicate that the file descriptor is a socket. The S_ISSOCK() macro, defined in sys/stat.h, can be used to test the st_mode field. The S_ISSOCK() macro is analogous to the S_ISREG(), S_ISFIFO(), and other related macros provided in sys/stat.h to test for other file types.

Examples of other POSIX functions which can be called for socket file descriptors are chmod(), chown(), dup(), and dup2(). Note that usingfcntl(), dup(), or dup2(), it is possible to open several file descriptors for the same socket. When this is done, all of the file descriptors are considered to be equivalent, in the sense that none of them has any special status over the rest. A socket is closed when the last file descriptor which refers to a socket is closed.

When using AF_UNIX sockets, files are created by the bind() function call in the Byte File System, CMS’s implementation of a POSIX-compliant file system. These files cannot be opened with the open() function, and are used only by the connect() function in the z/VM C sockets library. If stat() or lstat() is used on one of these files, the st_mode field in the return stat structure indicates that the file is a socket. The S_ISSOCK() macro can be used to test for this file type.

**Note:** Certain other function calls in the BPX layer of interfaces may report these files to be “external links”.
Miscellaneous Implementation Notes

The following are some miscellaneous points to consider when writing socket programs for the z/VM C sockets library:

1. Most of the socket functions are defined to return an EFAULT error if an address is passed which cannot be used by the sockets library. In certain cases, your program may receive a signal such as SIGSEGV instead. A multithreading library such as the z/VM C sockets library has difficulty prechecking for all of the conditions that could cause an EFAULT error, so invalid addresses may sometimes be used instead of causing EFAULT. In the worst case, if using AF_INET sockets, the library’s IUCV connection with the TCP/IP service virtual machine may be severed by TCP/IP when it receives an IUCV error because of an invalid address or length.

2. If you receive an error from a sockets function call, you may call the perror() or strerror() functions to translate the errno into an error message. These routines can decode socket error codes as well as error codes from non-socket functions. For compatibility with the VM TCP/IP C sockets library, the z/VM C sockets library defines a tcperror() function which invokes perror(). A small difference between tcperror() and perror() is that if the sockets library encountered an error during its most recent IUCV communication on the invoking thread, the tcperror() function will report that error as well as the one represented in the errno variable. It is not necessary to use tcperror() unless this additional information concerning IUCV errors is needed.

3. If you wish to use the AF_UNIX or AF_IUCV addressing domains, the virtual machines that wish to connect to each other need sufficient authorization in their user directory entries to allow the connection to be established. Usually this is handled by placing a statement in the user directory entry of the server virtual machine like the following:

   IUCV ALLOW

   This statement tells CP to allow any virtual machine to establish an IUCV connection to the server.

   Another possibility is to place a statement like the following in the user directory entry for each client virtual machine:

   IUCV ANY

   This statement tells CP to allow this virtual machine to establish an IUCV connection with any other virtual machine.

   Note: Technically, this requirement is true for AF_INET connections too, but is usually not mentioned because the TCP/IP server virtual machine is expected to have an IUCV ALLOW statement in its directory entry, which permits any client virtual machine to establish a connection with it.

Incompatibilities with the VM TCP/IP C Sockets Library

The goal of the z/VM C sockets library is to allow easier porting of UNIX programs that use sockets, and to provide a sockets API which can coexist with, and is more compatible with, the POSIX.1 API. To achieve this goal, it was often necessary to introduce incompatibilities with the TCP/IP C sockets library, because it has many incompatibilities with typical UNIX implementations. The following are some of the incompatibilities between the VM TCP/IP C sockets library and the z/VM C sockets library:

1. The names of the socket header files differ a great deal between the two libraries. For example, the z/VM C sockets library does not have a manifest.h
header file, and you should not attempt to include one in your program. Another example is that the old \texttt{bsdtypes.h} header file has been replaced with a \texttt{sys/types.h} header file. The \texttt{time.h} header file has been replaced with two header files, \texttt{time.h} and \texttt{sys/time.h}. It is necessary for you to include the correct one (or both, if necessary) in your program. Be sure to use the header file names required by the functions as documented in this reference guide. Do not omit the path name prefixes which are documented. For example, do not include \texttt{time.h} when you really should be including \texttt{sys/time.h}. The path name prefixes are significant.

2. Most of the header files to be used with the z/VM C sockets library are provided with Language Environment. Because those header files support several levels of socket functionality (on z/OS), it is necessary for all z/VM C socket programs to declare the level of functionality they want by defining the \_OE_SOCKETS preprocessor symbol before including any Language Environment header files. Failure to do so will usually cause several confusing compilation error messages. One way to define this preprocessor symbol is to place a statement similar to the following at the top of each source file of your program:

\begin{verbatim}
#define _OE_SOCKETS
\end{verbatim}

Alternatively, you can cause the \_OE_SOCKETS preprocessor symbol to be defined by the compiler, by using the `-D` option on the c89 command line. See "Compiling and Linking a Sockets Program" on page 26 for more information on compiling sockets programs.

3. The BSD 4.4 UNIX system introduced a new field into the sockaddr structures used by many socket functions. For each socket address family, there is a sockaddr\_xx structure which contains fields that define the address. For example, in the AF_INET address family, the structure is called sockaddr\_in and primarily contains an IP address and port number. In the AF_UNIX address family, the structure is called sockaddr\_un and primarily contains a file name. There is also a generic structure called sockaddr.

In the TCP/IP C sockets library definitions, such structures do not contain self-defining length fields. Each socket function that accepts a sockaddr structure also accepts a length, so there is really no need for them to contain lengths within the structure. In the 4.4 BSD UNIX implementation, however, there is now a length field in the sockaddr structure, so the Language Environment header files have them too. The z/VM C sockets library uses them, therefore, when processing those functions. When a sockaddr structure is given to the z/VM C sockets library, the length fields are handled as follows:

- For AF_INET sockets, the library verifies that the length field is either 0 or sizeof(struct sockaddr\_in). If the library sees a zero length, it assumes that the application does not know about length fields, and uses sizeof(struct sockaddr\_in) instead. If it sees any other length value, it rejects the socket request with EINVAL.
- For AF_UNIX sockets, if the length field is nonzero, the library uses it to limit how much of the file name is examined; a zero length field is ignored.
- For AF_IUCV sockets, if the length field is nonzero, the library checks to make sure it is equal to sizeof(struct sockaddr\_iucv).

If you do not explicitly initialize the sockaddr length field, then, depending on how the storage is allocated, you might have an unintended value there, and get unexpected EINVAL errors. This is more of a problem for AF_UNIX sockets than for AF_INET sockets. The reason for this is that an AF_INET sockaddr structure already contains a field which must be zeros, so most robust
applications use memset() to zero the entire sockaddr structure before filling it in. Because the z/VM C sockets library treats a zero sockaddr_in length field the same as if sizeof(struct sockaddr_in) were specified, robust AF_INET applications need no changes to deal with sockaddr length fields. AF_UNIX sockaddr structures have no fields which must be zero, however, so it is less likely that the structure will be cleared before filling it in, especially since the full size is so much bigger. Having uninitialized data in that length field might cause the socket library to use less of the file name than you intend.

A simple method to check code you are porting for proper length-field handling is to search for places that initialize the “family” field, which is called sin_family for AF_INET sockets, and sun_family for AF_UNIX sockets. If there is a call to memset() just before this code to clear the entire structure, you are probably safe. If not, you should fill in the length field. For AF_INET sockets, either fill it in with 0 or sizeof(struct sockaddr_in). For AF_UNIX sockets, either fill it in with 0 or a value greater than or equal to SUN_LEN(&sa), where sa is the name of the sockaddr_un variable, but less than or equal to sizeof(struct sockaddr_un). See the bind() and connect() functions in the z/VM: Language Environment 1.8 C Run-Time Library Reference for more information and examples of how to initialize the sockaddr length fields.

4. The z/VM C sockets library supports the selectex() function call. No WAITECBs are done, because the CMS OS WAIT and POST are not multitasking-aware and are not interruptable. Instead, the ECB post bits are checked directly during a polling loop that processes socket and file descriptors. If a set post bit is found, then selectex() stops processing and returns to its caller.

Consider replacing the usage of ECBs with POSIX constructs such as condition variables. You can then create a thread that waits on the condition variable, and when the condition being waited for has really occurred, it could signal a thread blocked in a select() call, if necessary, to cause it to exit the select(). In may cases, the signal is not even necessary; the thread that waited on the condition variable could process the event itself without disturbing the other threads.

5. If your program calls tcperror() instead of perror(), you must define the constant _OPEN_SOURCE as follows:

```
#define _OPEN_SYS_SOCK_EXT
```

This will cause tcperror() to be mapped to perror().

6. If you used the TCP/IP C sockets library, and did not call the maxdesc() function to increase the number of sockets you could open, your program was guaranteed that no socket descriptor would be greater than 49. This guarantee may have been exploited by programs, allowing them to set the FD_SETSIZE preprocessor symbol to a very small value. The FD_SETSIZE preprocessor symbol is used by the sys/time.h header file to control how much storage it takes to hold an fd_set, as used by the select() function call. When using the z/VM C sockets library, there is no relationship between file descriptor numbers as allocated by CMS and the value of the FD_SETSIZE preprocessor symbol. Another change to keep in mind is that the default FD_SETSIZE in the Language Environment sys/time.h header file is 2048, much larger than the default of 256 in the TCP/IP bsdtypes.h header file.

7. Programs compiled with the TCP/IP C sockets library header files must be recompiled before they can be link edited to the z/VM C sockets library. The two sets of header files do not produce object-compatible code. For example,
C Sockets API

the external symbol names associated with socket functions have changed, and the errno mapping is quite different.

8. If your program includes BSD header files (bsdtypes.h, bsdtime.h, bsdtocms.h), you must remove those includes. The z/VM C sockets library covers BSD functions, but it does not provide those header files.

9. In the TCP/IP C sockets library, getdtablesize() returns the maximum number of socket descriptors. In the z/VM C sockets library, getdtablesize() functions as it really should, returning the maximum number of file descriptors. The z/VM C sockets library provides getstablesize() for determining the maximum number of socket descriptors. If an existing TCP/IP application that uses getdtablesize() is being rebuilt with the z/VM C sockets library, to get the same results as before you must either use the APPTYPE environment variable or change the getdtablesize() call to getstablesize().

10. The TCP/IP remote procedure calls library (RPCLIB) cannot be used with the z/VM C sockets library. Use the VMRPC library instead. For more information about RPC, see Chapter 5, “Remote Procedure Calls” on page 149.

Incompatibilities with z/OS and OS/390 C Sockets

The z/VM C socket API is equivalent to the OS/390 Language Environment 2.5 sockets subset, except the following functions have not been implemented in z/VM:

- accept_and_recv()
- aio_read()
- aio_write()
- poll()
- send_file()
- socketpair()
- srx_np()

Incompatibilities with the Berkeley Socket Implementation

The following list summarizes some of the differences between the z/VM C socket implementation and the Berkeley socket implementation:

1. The z/VM ioctl() implementation may be different from the current Berkeley ioctl() implementation.

2. The z/VM getsockopt() and setsockopt() calls support only a subset of the options available, and only for the AF_INET address family.

3. z/VM C sockets do not work with the X-windows library for TCP/IP. If your application program uses this X-windows library, you must use the TCP/IP C socket API instead of the z/VM C socket API.

4. In the z/VM C socket API, the AF_UNIX address family does not support datagram sockets or nonblocking mode.

Compiling and Linking a Sockets Program

This section describes how to compile and link-edit C Language programs that use the z/VM C sockets library. Other books with information on compiling and linking C Language applications are:

- C for VM/ESA- Programming Guide
- Language Environment for OS/390 & VM: Programming Guide

Note: An existing application that currently uses the VM TCP/IP C sockets library (COMMTXT) may continue to do so in exactly the same way it did before, without any modification. Also, the application may also continue to use the
Compiling and Linking a z/VM C Sockets Program

To compile z/VM C socket programs, you must have the IBM C for VM/ESA Compiler, Version 3 (5654-033) Release 1, and IBM Language Environment (supplied with z/VM) installed on your z/VM system. In order to use AF_INET sockets, you must have TCP/IP installed and running.

To compile and link-edit a z/VM C sockets application program, use the `c89` utility. You must make sure that `c89` has access to the files it needs to compile and link-edit. The VM-unique header files reside on the CMS S-disk. The Language Environment object code and header files reside on the Y-disk.

Another aspect to ensuring that `c89` has all required files available is to make sure that you have a Byte File System mounted and available. The files and directories in this Byte File System must be arranged in the manner done by the BFS installation procedures. Specifically, the `/usr/include/sys/time.h` file is assumed to be an external link of type CMSDATA to the SYS_TIME H file.

The `c89` program can be run from the CMS command line (or equivalent) or from within a POSIX command shell, if you are running with a command shell. The syntax is the same in both cases. For example, if you wish to compile the `testprog.c` file in your current Byte File System (BFS) directory and bind the socket function stubs to it, use a command like the following:

```
c89 -o testprog -D_OE_SOCKETS testprog.c
```

Depending on the TCP/IP functions your application uses, additional libraries (listed in Table 1) may be required. For example, if `testprog.c` uses RPC functions, the command would be:

```
c89 -o testprog -D_OE_SOCKETS testprog.c -l//VMRPC
```

<table>
<thead>
<tr>
<th>TXTLIB File</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMRPC</td>
<td>Remote procedure calls</td>
</tr>
<tr>
<td>SCEELKED</td>
<td>C APIs</td>
</tr>
<tr>
<td>SCEEOBJ</td>
<td>C writable static variables</td>
</tr>
</tbody>
</table>

The previous examples assume that `testprog.c` is the only source file in the program, and that the `_OE_SOCKETS` preprocessor symbol is not defined in the source file itself. If it is, then do not specify the `-D` option. To avoid having to type the `-D` option all the time, or including it in your `make` file, put the following statement in the beginning of every source file of your program, before it includes its first header file:

```
#define _OE_SOCKETS
```

If you want to use the Language Environment extended socket and bulkmode support, define the feature test macro `_OPEN_SYS_SOCK_EXT` using a preprocessor directive, either on the `c89` compile command line:

```
c89 filename -D_OPEN_SYS_SOCK_EXT
```
or in the source code before including any header files:
#define _OPEN_SYS_SOCK_EXT

For more information about this macro, see the z/VM: Language Environment 1.8 C
Run-Time Library Reference.

Sometimes defining _OE_SOCKETS in the source program itself is inconvenient
because, for example, you are porting many source files from another system, and
you would rather not change them all. In this case, define _OE_SOCKETS on the
c89 command line with the -D option, either by hand or in your make file, if you
are using the make utility. For more information about the make utility, see z/VM:
OpenExtensions Advanced Application Programming Tools.

The -o option in the example above tells c89 to store the final executable file with
the name testprog. This overrides the default name of a.out.

When you compile, be very careful not to have the disk containing the header files
for TCP/IP accessed ahead of the disk containing the Language Environment
header files. Because both TCP/IP and Language Environment ship socket header
files with the same names, it is important to use the correct Language Environment
header files, and not the TCP/IP files. No TCP/IP header files are needed to
compile z/VM C socket programs. However, if you are using RPC functions, then
header files on the TCP/IP disk are required.

If you would rather have c89 produce a MODULE file on a CMS minidisk or
accessed SFS directory, then specify something like the following on your c89
command:
c89 -o //testprog -D_OE_SOCKETS testprog.c

This will cause c89 to create a CMS file called TESTPROG MODULE A instead of an
executable file in the BFS.

If the source file itself is on a CMS minidisk or accessed SFS directory, then use a
c89 command like the following:
c89 -D_OE_SOCKETS //testprog.c

This example adds // to the front of the name of the source file, and removes the
-o option. It adds the // because the source file resides on a CMS minidisk. The -o
option is removed because when the testprog.c source file is on a minidisk, the
c89 default name for the executable file is //testprog.module.a, so there is no need
to specify it explicitly.

Note: It is not necessary to use uppercase letters in the name, type, or mode of a
CMS file when the file ID is preceded with // . The file ID is converted to
uppercase automatically.

If your program is composed of several source files, for example progmile1.c and
progmile2.c, you can use either of the two following sequences to produce an
executable file.
c89 -c -D_OE_SOCKETS progmile1.c
c89 -c -D_OE_SOCKETS progmile2.c
c89 -o testprog progmile1.o progmile2.o

or

c89 -o testprog -D_OE_SOCKETS progmile1.c progmile2.c
In the first sequence, the source files are first compiled (the -c option prevents \texttt{c89} from trying to link edit them) and then link-edited in a separate \texttt{c89} command. In the second sequence, the source files are compiled and link-edited in one command. The point being demonstrated in this example is that the -D option is needed only for the compilation step.

Many other variations of the \texttt{c89} command are possible. See the \texttt{z/VM} OpenExtensions Command Reference for a complete description of the \texttt{c89} command. After you have created an executable program, you can use the OPENVM GET and OPENVM PUT commands to move it back and forth between a CMS minidisk or accessed SFS directory and the Byte File System. See the \texttt{z/VM: OpenExtensions Command Reference} for information on those commands.

### Compiling and Linking a TCP/IP C Sockets Program

If you want to recompile and relink an existing application that was built with VM TCP/IP C sockets, you have three choices:

- Convert the program to use the z/VM C sockets library
- Recompile and relink using the z/VM C sockets library with minimal changes to the program source
- Recompile using the TCP/IP C sockets library

#### Converting Your Program to Use z/VM C Sockets

To convert a TCP/IP C sockets program to use z/VM C sockets:

1. Go to "Incompatibilities with the VM TCP/IP C Sockets Library" on page 23. Make the necessary changes to your program to resolve the incompatibilities.
2. Go to "Compiling and Linking a z/VM C Sockets Program" on page 27 and follow the instructions.

#### Using the z/VM C Sockets Library with Minimal Changes to Program Source

You can recompile and relink your VM TCP/IP C sockets program to use the z/VM C sockets library (SCEELKED) with little or no source code modification:

- If your program uses remote procedure calls, you must use the VMRPC library instead of RPCLIB.
- If your program includes BSD header files (\texttt{bsdtypes.h}, \texttt{bsdt ime.h}, \texttt{bsdto cms.h}), you must remove those includes. Language Environment covers BSD functions, but it does not provide those header files.
- Define the feature test macro \_TCPVM SOCKETS using a preprocessor directive, either on the \texttt{c89} command line:

  \begin{verbatim}
  c89 filename -D_TCPVM SOCKETS
  \end{verbatim}

  or in the source code before including any header files:
  \begin{verbatim}
  #define _TCPVM SOCKETS
  \end{verbatim}

#### Recompiling with the TCP/IP C Sockets Library

To recompile with the TCP/IP C sockets library:

1. Access the TCP/IP Client-code minidisk (usually TCPMAINT 592) \textit{ahead} of the disk that contains the Language Environment header files (usually the Y-disk) to avoid conflicts.
2. Establish the C development environment:

   a. Access the C compiler
   b. Issue GLOBAL LOADLIB SCEERUN.
3. Compile your program. Make sure that the preprocessor symbol VM is defined; if it is not already defined in your program, you can specify it on the compile command:

```bash
CC myprog (DEF(VM)
```

With OpenExtensions, you can also use the `c89` command or the `make` utility.

4. Select the link libraries your application needs and put them on a GLOBAL TXTLIB command. COMMTXT and SCEELKED are the minimum required:

```bash
GLOBAL TXTLIB COMMTXT SCEELKED
```

Additional libraries (listed in Table 2) may be required, depending on the functions your application uses. For example, programs that use RPC must issue:

```bash
GLOBAL TXTLIB COMMTXT RPCLIB SCEELKED
```

Note that the Language Environment text library, SCEELKED, should always be listed last.

<table>
<thead>
<tr>
<th>TXTLIB File</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMTXT</td>
<td>TCP/IP C sockets and Pascal API</td>
</tr>
<tr>
<td>RPCLIB</td>
<td>Remote procedure calls</td>
</tr>
<tr>
<td>BPLDBM</td>
<td>Kerberos</td>
</tr>
<tr>
<td>KDB</td>
<td>Kerberos</td>
</tr>
<tr>
<td>KRBl</td>
<td>Kerberos</td>
</tr>
<tr>
<td>DES</td>
<td>Kerberos</td>
</tr>
<tr>
<td>XIILIB</td>
<td>Xlib, Xmu, Xext, and Xau routines</td>
</tr>
<tr>
<td>OLDXLIB</td>
<td>X Release 10 compatibility routines</td>
</tr>
<tr>
<td>XTLIB</td>
<td>X Intrinsics</td>
</tr>
<tr>
<td>XAWLIB</td>
<td>Athena widget set</td>
</tr>
<tr>
<td>XMLIB</td>
<td>OSF/Motif-based widget set</td>
</tr>
<tr>
<td>DPLIB</td>
<td>SNMP DPI</td>
</tr>
<tr>
<td>SCEELKED</td>
<td>C APIs</td>
</tr>
<tr>
<td>SCEEOBJ</td>
<td>C writable static variables</td>
</tr>
</tbody>
</table>

5. Link-edit your programs into an executable module. The sample applications in this book are built using the TCPLOAD utility. Your own applications should be built using the CMOD command. For example,

```bash
TCLOAD sample@cc
```

or

```bash
CMOD myprog1 myprog2 (AUTO
```

Complete information on compiling and link-editing C programs can be found in the `C for VM/ESA: Programming Guide`. For information about TCPLOAD, see Appendix A, “TCPLOAD EXEC” on page 353.
Running a Sockets Program

After building your executable sockets program, the next step is to run the program. Before you do, however, some preparation may be necessary. In addition, you may want to consider using environment variables to affect certain aspects of the execution. There are also differences between running a program from the BFS and running it from an accessed minidisk or SFS directory.

Preparing to Run a Sockets Program

If your program uses AF_INET sockets, then you should access the TCP/IP “client” minidisk or SFS directory that contains the TCP/IP configuration files. Usually you can LINK to TCPIP 592 to access the disk. Your installation may have assigned a VMLINK nickname to this minidisk (for example, TCP/IP). Issue a VMLINK command (with no arguments) to see if one has been assigned. In the compilation step, it was noted that this disk contains some header files with the same names as Language Environment and VM-unique header files. If you might compile your program again after running it, be sure to access the TCP/IP client disk at a mode after the disk that contains the Language Environment header files.

To run your POSIX program, it should reside on an accessed minidisk or SFS directory, or in a mounted BFS file system. As described below, if it resides on an accessed minidisk or SFS directory, you can run it by either typing the name of the executable module on the CMS command line, or by using the OPENVM RUN command. If it resides in the BFS, you can run it by either typing the name of the executable file on a POSIX shell command line, or by using the OPENVM RUN command. Before running a module that resides on an accessed minidisk or SFS directory, either by using the OPENVM RUN command or by simply typing the name of the module on the CMS command line, you must establish the proper run-time Language Environment load library with the following CMS command:

GLOBAL LOADLIB SCEERUN

Note: If you have a TCP/IP C sockets application that you have recompiled and relinked with the z/VM C sockets library, with no source changes, but you want the maxdesc() and getdtablesizet() functions to operate the same way they did in the TCP/IP C socket API, you must set the APPTYPE environment variable (to the value OLDAPP) before running your program. For example:

GLOBALV SELECT CENV SETLP APPTYPE OLDAPP

This will cause the maxdesc() default to be 50 and getdtablesizet() to return a maximum of 50.

Using Environment Variables

Environment variables can be used to affect certain aspects of the execution of a z/VM C sockets program. If the z/VM C sockets program is executed from the OpenExtensions shell, the shell controls the contents of the program’s environment. For example, the following shell command could be used to set the APPTYPE environment variable:

export APPTYPE=OLDAPP

If the z/VM C sockets program is being run from the CMS command line (or equivalent), then the global variables existing in the CENV group managed by the
GLOBALV command are used as the environment variables for the process. In this case, a CMS command like the following could be used to temporarily set the APPTYPE environment variable:

GLOBALV SELECT CENV SET APPTYPE OLDAPP

Note: Be aware, however, that some of the environment variables described below accept values which are case sensitive, and which will often be set to lowercase values. It can be difficult, using the GLOBALV command, to set lowercase values, because commands typed in from the CMS command line are automatically uppercased by CMS before processing. One way to set the variable to a mixed-case value is to issue the GLOBALV command from a REXX exec with "Address Command" in effect.

Some of the environment variables described below are set to values which represent file names. For these environment variables, the given file names are interpreted as POSIX-style file names, which means that case is significant, and that the file name is interpreted as residing in the Byte File System unless you precede the file name with two slashes. To specify the name of a file which resides on a minidisk or accessed SFS directory instead of in the BFS, precede the name of the file with two slashes, and separate the CMS file name and type (and mode, if specified) with a period.

The following environment variables can be used to affect the execution of the Language Environment sockets library:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPTYPE</td>
<td>This environment variable forces the z/VM C socket functions maxdesc() and getdtablesize() to return the same values that those functions returned in the TCP/IP C socket API.</td>
</tr>
<tr>
<td>HOSTALIASES</td>
<td>This environment variable tells the socket library resolver code to use the named file when searching for aliases for AF_INET host names. For example, setting the variable to the string /etc/aliases tells the resolver to use the /etc/aliases file when needed. By default, no aliases files is used by the resolver. If you intend to transfer an aliases file into the BFS using the OPENVM PUTBFS command, be sure to specify the BFSLINE NL option, or let it default.</td>
</tr>
<tr>
<td>X_ADDR</td>
<td>This environment variable tells the socket library resolver code to use the named file in place of the HOSTS ADDRINFO file, which contains information about AF_INET networks known to this host. For example, setting the variable to the string /etc/addrs tells the resolver to use the /etc/addrs file in place of the default file, which is //HOSTS.ADDRINFO. If you intend to transfer the HOSTS ADDRINFO file into the BFS using the OPENVM PUTBFS command, be sure to specify the BFSLINE NONE option. This environment variable is used by the gethostbyaddr() function call, the getnetent() function call, and several others. Note: When using the TCP/IP C sockets library, the format of this environment variable is X-ADDR.</td>
</tr>
<tr>
<td>X_SITE</td>
<td>This environment variable tells the socket library resolver code to use the...</td>
</tr>
</tbody>
</table>
named file in place of the HOSTS SITEINFO file, which contains information about AF_INET hosts known to this host. For example, setting the variable to the string /etc/hosts tells the resolver to use the /etc/hosts file in place of the default file, which is //HOSTS.SITEINFO. If you intend to transfer the HOSTS SITEINFO file into the BFS using the OPENVM PUTBFS command, be sure to specify the BFSLINE NONE option. This environment variable is used by the gethostbyname() function call, the gethostent() function call, and several others.

**Note:** When using the TCP/IP C sockets library, the format of this environment variable is **X-SITE**.

**X_XLATE**

This environment variable tells the socket library resolver code to use the named file in place of the STANDARD TCPXLBIN file, which contains ASCII to EBCDIC and EBCDIC to ASCII translation tables for use by the resolver when sending or receiving information from an AF_INET network. For example, setting the variable to the string /etc/xlate tells the resolver to use the /etc/xlate file in place of the default file, which is //STANDARD.TCPXLBIN. If you intend to transfer the STANDARD TCPXLBIN file into the BFS using the OPENVM PUTBFS command, be sure to specify the BFSLINE NONE option. This environment variable is used by the gethostbyname() and gethostbyaddr() function calls.

**Note:** When using the TCP/IP C sockets library, the format of this environment variable is **X-XLATE**.

### Running a Program Residing in the BFS

If you are using a POSIX command shell, and the executable file is in your path, then simply type the name of the program to run it:

```
testprog
```

or

```
/dirname/dirname/.../testprog
```

In this scenario, the shell will spawn a process to run your program. Any program which is spawned is automatically considered by CMS to be a POSIX program, and will have access to OpenExtensions services.

If you are not using a POSIX command shell, then use the OPENVM RUN command to execute the program. For example:

```
OPENVM RUN /dirname/dirname/.../testprog
```

In this scenario, the OPENVM RUN command will spawn a process to run your program. As before, your program will automatically have access to OpenExtensions services. Be aware that path name arguments to the OPENVM RUN command are case sensitive.

### Running a Program Residing on an Accessed Minidisk or SFS Directory

If the executable file is on an accessed minidisk or SFS directory, then there are several ways to execute it.
The OPENVM RUN command, which was mentioned earlier to run a program residing in the BFS, can also run a MODULE file. To do so, uppercase the file name on the command line, as follows:

```
OPENVM RUN TESTPROG
```

If the executable file is on an accessed minidisk or SFS directory, you can type the name of the module on the CMS command line to run it, but you must also tell CMS that the program is a POSIX program and should be given access to the OpenExtensions services.

The following are the two techniques for establishing your program as a POSIX program when you run it:

1. Specify the Language Environment POSIX(ON) run-time option on the CMS command line. In order to be able to pass run-time options to a program, the EXECOPS compiler option must be in effect when the program is compiled. Because it is the default setting, EXECOPS is in effect unless overridden with the NOEXECOPS option. Specify the run-time options by separating them from the program arguments with a slash (/) as you run your program:

```
testprog runopt1 runopt2 ... / arg1 arg2 arg3 ...
```

To specify the POSIX(ON) run-time option, use a command like the following:

```
testprog posix(on)/ arg1 arg2 arg3 ...
```

2. Specify the Language Environment POSIX(ON) run-time option by putting it in the source file containing the main function. If you plan to run the program often by simply typing its name on the CMS command line, the most convenient way to get the program recognized as a POSIX program is to place a pragma like the following in the source file which contains the main function:

```
#pragma runopts(posix(on))
```

With this pragma, you never need to type the POSIX(ON) run-time option.

When you run a program by typing its name on the CMS command line, and the EXECOPS compiler option was in effect when you compiled your program (it is by default), then everything before the first slash, if there is a slash on the command line, will be interpreted as a run-time option. Because POSIX path names often contain slashes, this can cause program arguments to be misinterpreted as run-time options if your program accepts a POSIX path name as an argument. To avoid this, consider placing a pragma like the following in the source file containing the main function:

```
#pragma runopts(noexecops,posix(on))
```

This will prevent POSIX path names from accidentally being interpreted as run-time options, and cause the POSIX(ON) run-time option to always be in effect.

Using pragma statements such as the ones discussed above is necessary only when you intend to run your program from the CMS command line by typing its name. If you use OPENVM RUN to run the program, or run it from a POSIX command shell, all operands are interpreted as program arguments, and the program is automatically treated as a POSIX program.
C Sockets Quick Reference

This section provides brief descriptions of the z/VM C socket calls. For additional information about these socket functions, see the [z/VM: Language Environment 1.8 C Run-Time Library Reference](#).

<table>
<thead>
<tr>
<th>Socket() Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accept()</td>
<td>Accepts a connection request from a foreign host.</td>
</tr>
<tr>
<td>bind()</td>
<td>Assigns a local address to a socket.</td>
</tr>
<tr>
<td>close()</td>
<td>Closes a socket.</td>
</tr>
<tr>
<td>connect()</td>
<td>Requests a connection to a foreign host.</td>
</tr>
<tr>
<td>endhostent()</td>
<td>Closes the HOSTS SITEINFO file.</td>
</tr>
<tr>
<td>endnetent()</td>
<td>Closes the HOSTS ADDRINFO file.</td>
</tr>
<tr>
<td>endprotoent()</td>
<td>Closes the ETC PROTO file.</td>
</tr>
<tr>
<td>endservent()</td>
<td>Closes the ETC SERVICES file.</td>
</tr>
<tr>
<td>fcntl()</td>
<td>Controls socket operating characteristics.</td>
</tr>
<tr>
<td>getclientid()</td>
<td>Returns the identifier by which the calling application is known to the TCPIP virtual machine.</td>
</tr>
<tr>
<td>gethostbyaddr()</td>
<td>Returns information about a host specified by an address.</td>
</tr>
<tr>
<td>gethostbyname()</td>
<td>Returns information about a host specified by a name.</td>
</tr>
<tr>
<td>gethostent()</td>
<td>Returns the next entry in the HOSTS SITEINFO file.</td>
</tr>
<tr>
<td>gethostid()</td>
<td>Returns the unique identifier of the current host.</td>
</tr>
<tr>
<td>gethostname()</td>
<td>Returns the standard name of the current host.</td>
</tr>
<tr>
<td>getnetbyaddr()</td>
<td>Returns the network entry specified by address.</td>
</tr>
<tr>
<td>getnetbyname()</td>
<td>Returns the network entry specified by name.</td>
</tr>
<tr>
<td>getnetent()</td>
<td>Returns the next entry in the HOSTS ADDRINFO file.</td>
</tr>
<tr>
<td>getpeername()</td>
<td>Returns the name of the peer connected to a socket.</td>
</tr>
<tr>
<td>getprotobyname()</td>
<td>Returns a protocol entry specified by name.</td>
</tr>
<tr>
<td>getprotobynumber()</td>
<td>Searches the ETC PROTO file for a specified protocol number.</td>
</tr>
<tr>
<td>getprotoent()</td>
<td>Returns the next entry in the ETC PROTO file.</td>
</tr>
<tr>
<td>getservbyname()</td>
<td>Returns a service entry specified by name.</td>
</tr>
<tr>
<td>getservbyport()</td>
<td>Returns a service entry specified by port number.</td>
</tr>
<tr>
<td>getservent()</td>
<td>Returns the next entry in the SERVICES file.</td>
</tr>
<tr>
<td>getsockname()</td>
<td>Obtains the local socket name.</td>
</tr>
<tr>
<td>getsockopt()</td>
<td>Gets options associated with sockets in the AF_INET domain.</td>
</tr>
<tr>
<td>givesocket()</td>
<td>Tells TCPIP to make the specified socket available to a takesocket() call issued by another application.</td>
</tr>
<tr>
<td>htons()</td>
<td>Translates host byte order to network byte order for a long integer.</td>
</tr>
<tr>
<td>inet_addr()</td>
<td>Constructs an internet address from character strings set in standard dotted-decimal notation.</td>
</tr>
<tr>
<td>inet_lnaof()</td>
<td>Returns the local network portion of an internet address.</td>
</tr>
</tbody>
</table>
### Table 3. C Sockets Quick Reference (continued)

<table>
<thead>
<tr>
<th>Socket() Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inet_makeaddr()</td>
<td>Constructs an internet address from a network number and a local address.</td>
</tr>
<tr>
<td>inet_netof()</td>
<td>Returns the network portion of the internet address in network byte order.</td>
</tr>
<tr>
<td>inet_network()</td>
<td>Constructs a network number from character strings set in standard dotted-decimal notation.</td>
</tr>
<tr>
<td>inet_ntoa()</td>
<td>Returns a pointer to a string in dotted-decimal notation.</td>
</tr>
<tr>
<td>ioctl()</td>
<td>Performs special operations on a socket.</td>
</tr>
<tr>
<td>listen()</td>
<td>Indicates that a stream socket is ready for a connection request from a foreign host.</td>
</tr>
<tr>
<td>maxdesc()</td>
<td>Allows socket numbers to extend beyond default range of 0 - 49.</td>
</tr>
<tr>
<td>ntohl()</td>
<td>Translates network byte order to host byte order for a long integer.</td>
</tr>
<tr>
<td>ntohss()</td>
<td>Translates network byte order to host byte order for a short integer.</td>
</tr>
<tr>
<td>read()</td>
<td>Reads a set number of bytes into a buffer.</td>
</tr>
<tr>
<td>readv()</td>
<td>Obtains data from a socket and reads this data into specified buffers.</td>
</tr>
<tr>
<td>recv()</td>
<td>Receives messages from a connected socket.</td>
</tr>
<tr>
<td>recvfrom()</td>
<td>Receives messages from a datagram socket, regardless of its connection status.</td>
</tr>
<tr>
<td>recvmsg()</td>
<td>Receives messages on a socket into an array of buffers.</td>
</tr>
<tr>
<td>select()</td>
<td>Monitors activity on a set of sockets.</td>
</tr>
<tr>
<td>selectex()</td>
<td>Monitors activity on a set of different sockets.</td>
</tr>
<tr>
<td>send()</td>
<td>Transmits messages to a connected socket.</td>
</tr>
<tr>
<td>sendmsg()</td>
<td>Sends messages on a socket from an array of buffers.</td>
</tr>
<tr>
<td>sendto()</td>
<td>Transmits messages to a datagram socket, regardless of its connection status.</td>
</tr>
<tr>
<td>sethostent()</td>
<td>Opens the HOSTS SITEINFO file at the beginning.</td>
</tr>
<tr>
<td>setnetent()</td>
<td>Opens the HOSTS ADDRINFO file at the beginning.</td>
</tr>
<tr>
<td>setprotoent()</td>
<td>Opens the ETC PROTO file at the beginning.</td>
</tr>
<tr>
<td>setservent()</td>
<td>Opens the ETC SERVICES file at the beginning.</td>
</tr>
<tr>
<td>setsockopt()</td>
<td>Sets options associated with a socket in the AF_INET domain.</td>
</tr>
<tr>
<td>shutdown()</td>
<td>Shuts down all or part of a full-duplex connection.</td>
</tr>
<tr>
<td>socket()</td>
<td>Requests that a socket be created.</td>
</tr>
<tr>
<td>takesocket()</td>
<td>Acquires a socket from another application.</td>
</tr>
<tr>
<td>write()</td>
<td>Writes a set number of bytes from a buffer to a socket.</td>
</tr>
<tr>
<td>writev()</td>
<td>Writes data in the buffers specified by an array of iovec structures.</td>
</tr>
</tbody>
</table>

### TCP Client Program

The following is an example of a C socket TCP client program.
/* Include Files. */
#define VM
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>

/* Client Main. */
main(argc, argv)
int argc;
char **argv;
{
  unsigned short port; /* port client will connect to */
  char buf[12]; /* data buffer for sending and receiving */
  struct hostent *hostnm; /* server host name information */
  struct sockaddr_in server; /* server address */
  int s; /* client socket */

  /* Check Arguments Passed. Should be hostname and port. */
  if (argc != 3)
    {
      fprintf(stderr, "Usage: %s hostname port\n", argv[0]);
      exit(-1);
    }

  /* The host name is the first argument. Get the server address. */
  hostnm = gethostbyname(argv[1]);
  if (hostnm == (struct hostent *) 0)
    {
      fprintf(stderr, "Gethostbyname failed\n");
      exit(-1);
    }

  /* The port is the second argument. */
  port = (unsigned short) atoi(argv[2]);

  /* Put a message into the buffer. */
  strcpy(buf, "the message");

  /* Put the server information into the server structure. */
  server.sin_family = AF_INET;
  server.sin_port = htons(port);
  server.sin_addr.s_addr = *((unsigned long *)hostnm->h_addr);

  /* Get a stream socket. */
  if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    { perror("Socket()");}}
**C Sockets API**

```c
exit(-1);
}

/*
 * Connect to the server.
 */
if (connect(s, &server, sizeof(server)) < 0)
{
    perror("Connect()");
    exit(-1);
}

if (send(s, buf, sizeof(buf), 0) < 0)
{
    perror("Send()");
    exit(-1);
}

/*
 * The server sends back the same message. Receive it into the buffer.
 */
if (recv(s, buf, sizeof(buf), 0) < 0)
{
    perror("Recv()");
    exit(-1);
}

/*
 * Close the socket.
 */
close(s);

printf("Client Ended Successfully\n");
exit(0);
}
```

**TCP Server Program**

The following is an example of a C socket TCP server program.

```c
#include <arpa/inet.h>
#include <netinet/in.h>
#include <socket.h>
#include <stdio.h>

/* Server Main. */
main(argc, argv)
    int argc;
    char **argv;
{
    unsigned short port; /*port server binds to*/
    char buf[12]; /*buffer for sending and receiving data*/
    struct sockaddr_in client; /*client address information*/
    struct sockaddr_in server; /*server address information*/
    int s; /*socket for accepting connections*/
    int ns; /*socket connected to client*/
    int namelen; /*length of client name*/
```
/*
 * Check arguments. Should be only one: the port number to bind to.
 */
if (argc != 2)
{
 fprintf(stderr, "Usage:%s port\n", argv[0]);
 exit(-1);
}
/*
 * First argument should be the port.
 */
port = (unsigned short)atoi(argv[1]);
/*
 * Get a socket for accepting connections.
 */
if ((s = socket(AF_INET,SOCK_STREAM, 0)) < 0)
{
 perror("Socket()");
 exit(-1);
}
/*
 * Bind the socket to the server address.
 */
server.sin_family = AF_INET;
server.sin_port = htons(port);
server.sin_addr.s_addr = INADDR_ANY;
if (bind(s, &server, sizeof(server)) < 0)
{
 perror("Bind()");
 exit(-1);
}
/*
 * Listen for connections. Specify the backlog as 1.
 */
if (listen(s, 1) != 0)
{
 perror("Listen()");
 exit(-1);
}
/*
 * Accept a connection.
 */
namelen = sizeof(client);
if ((ns = accept(s, &client,&namelen)) == -1)
{
 perror("Accept()");
 exit(-1);
}
/*
 * Receive the message on the newly connected socket.
 */
if (recv(ns, buf, sizeof(buf),0) == -1)
{
 perror("Recv()");
 exit(-1);
}
C Sockets API

```c
if (send(ns, buf, sizeof(buf),0) < 0)
{
    perror("Send()");
    exit(-1);
}

close(ns);
close(s);

printf("Server ended successfully\n");
exit(0);
```

UDP Client Program

The following is an example of a C socket UDP client program.

```c
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>

main(argc, argv)
int argc;
char **argv;
{

    int s;
    unsigned short port;
    struct sockaddr_in server;
    char buf[32];

     * Convert the port from ascii to integer and then from host byte
     * order to network byte order.
     */
    if(argc != 3)
    {
        printf("Usage: %s <host address> <port> \n",argv[0]);
        exit(1);
    }
    port = htons(atoi(argv[2]));

    /* Create a datagram socket in the internet domain and use the
     * default protocol (UDP).
     */
    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("socket()");
        exit(1);
    }

    /* Set up the server name */
    server.sin_family = AF_INET; /* Internet Domain */
    server.sin_port = port; /* Server Port */
    server.sin_addr.s_addr = inet_addr(argv[1]); /* Server's Address */

    strcpy(buf, "Hello");

    /* Send the message in buf to the server */
    if (sendto(s, buf, (strlen(buf)+1), 0, &server, sizeof(server)) < 0)
    {
        perror("sendto()");
    }
```
UDP Server Program

The following is an example of a C socket UDP server program.

```c
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>

main()
{
    int s, namelen, client_address_size;
    struct sockaddr_in client, server;
    char buf[32];

    /* Create a datagram socket in the internet domain and use the */
    /* default protocol (UDP). */
    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("socket()");
        exit(1);
    }

    /* Bind my name to this socket so that clients on the network can */
    /* send me messages. (This allows the operating system to demultiplex */
    /* messages and get them to the correct server) */
    /* Set up the server name. The internet address is specified as the */
    /* wildcard INADDR_ANY so that the server can get messages from any */
    /* of the physical internet connections on this host. (Otherwise we */
    /* would limit the server to messages from only one network */
    /* interface.) */
    server.sin_family = AF_INET; /* Server is in Internet Domain */
    server.sin_port = 0; /* Use any available port */
    server.sin_addr.s_addr = INADDR_ANY; /* Server's Internet Address */
    if (bind(s, &server, sizeof(server)) < 0)
    {
        perror("bind()");
        exit(2);
    }

    /* Find out what port was really assigned and print it */
    namelen = sizeof(server);
    if (getsockname(s, (struct sockaddr *) &server, &namelen) < 0)
    {
        perror("getsockname()");
        exit(3);
    }

    printf("Port assigned is %d\n", ntohs(server.sin_port));

    /* Receive a message on socket s in buf of maximum size 32 */
```
C Sockets API

* from a client. Because the last two parameters
* are not null, the name of the client will be placed into the
* client data structure and the size of the client address will
* be placed into client_address_size.
*/
client_address_size = sizeof(client);

if(recvfrom(s, buf, sizeof(buf), 0, (struct sockaddr *) &client,
    &client_address_size) <0)
{
    perror("recvfrom()");
    exit(4);
}
/*
* Print the message and the name of the client.
* The domain should be the internet domain (AF_INET).
* The port is received in network byte order, so we translate it to
* host byte order before printing it.
* The internet address is received as 32 bits in network byte order
* so we use a utility that converts it to a string printed in
* dotted decimal format for readability.
*/
printf("Received message %s from domain %s port %d internet" 
"address %s\n", 
    buf,
    (client.sin_family == AF_INET?"AF_INET":"UNKNOWN"),
    ntohs(client.sin_port),
    inet_ntoa(client.sin_addr));

/*
* Deallocate the socket.
*/
close(s);
Chapter 2. TCP/UDP/IP API (Pascal Language)

This chapter describes the Pascal language application program interface (API) provided with TCP/IP for z/VM. This interface allows programmers to write application programs that use the TCP, UDP, and IP layers of the TCP/IP protocol suite.

You should have experience in Pascal language programming and be familiar with the principles of internetwork communication to use the Pascal language API.

Your program uses procedure calls to initiate communication with the TCPIP virtual machine. Most of these procedure calls return with a code that indicates success, or the type of failure incurred by the call. The TCPIP virtual machine starts asynchronous communication by sending you notifications.

The general sequence of operations is:

1. Start TCP/UDP/IP service (BeginTcpIp, StartTcpNotice).
2. Specify the set of notifications that TCP/UDP/IP may send you (Handle).
3. Establish a connection (TcpOpen, UdpOpen, RawIpOpen, TcpWaitOpen).
   If using TcpOpen, you must wait for the appropriate notification that a connection has been established.
4. Transfer data buffer to or from the TCPIP virtual machine (TcpSend, TcpFSend, TcpWaitSend, TcpReceive, TcpFRReceive, TcpWaitReceive, UdpSend, UdpNReceive, RawIpSend, RawIpReceive).
   Note: TcpWaitReceive and TcpWaitSend are synchronous calls.
5. Check the status returned from the TCPIP virtual machine in the form of notifications (GetNextNote).
6. Repeat the data transfer operations (steps 4 and 5) until the data is exhausted.
7. Terminate the connection (TcpClose, UdpClose, RawIpClose).
   If using TcpClose, you must wait for the connection to terminate.
8. Terminate the communication service (EndTcpIp).

Control is returned to you, in most instances, after the initiation of your request. When appropriate, some procedures have alternative wait versions that return only after completion of the request. The bodies of the Pascal procedures are in the TCPIP ATCPPSRC file.

A sample program is supplied with the TCP/IP program, see "Sample Pascal Program" on page 91.

Software Requirements

To develop programs in Pascal that interface directly to the TCP, UDP, and IP protocol boundaries, you require the IBM VS Pascal Compiler & Library (5668-767).
Data Structures

Programs containing Pascal language API calls must include the appropriate data structures. The data structures are declared in the CMCOMM COPY and CMCLIEN COPY. The CMCOMM and CMCLIEN are included in the ALLMACRO MACLIB shipped with TCP/IP. To include these files in your program source, enter:

```
#include CMCOMM
#include CMCLIEN
```

Additional include statements are required in programs that use certain calls. The following list shows the members of the ALLMACRO MACLIB that need to be included for the various calls.

- CMRESGLB for GetHostResol
- CMINTER for GetHostNumber, GetHostString, IsLocalAddress, and IsLocalHost.

The load modules are in the TCPIP COMMTXT file. Include this file in your GLOBAL TXTLIB command when you are creating a load module to link an application program.

Connection State

ConnectionState is the current state of the connection. For the Pascal declaration of the ConnectionStateType data type, see Figure 15. ConnectionStateType is used in StatusInfoType and NotificationInfoType. It defines the client program's view of the state of a TCP connection, in a form more readily usable than the formal TCP connection state defined by RFC 793. For the mapping between TCP states and ConnectionStateType, see Table 4 on page 45.

```
ConnectionStateType =
(  
  CONNECTIONclosing,
  LISTENING,
  NONEXISTENT,
  OPEN,
  RECEIVINGonly,
  SENDINGonly,
  TRYINGtoOPEN
);
```

*Figure 15. Pascal Declaration of Connection State Type*

**CONNECTIONclosing**
Indicates that no more data can be transmitted on this connection, because it is going through the TCP connection closing sequence.

**LISTENING**
Indicates that you are waiting for a foreign site to open a connection.

**NONEXISTENT**
Indicates that a connection no longer exists.

**OPEN**
Indicates that data can go either way on the connection.

**RECEIVINGonly**
Indicates that data can be received, but cannot be sent on this connection, because the client has done a TcpClose.
SENDINGonly
Indicates that data can be sent out, but cannot be received on this connection, because the foreign application has done a TcpClose or equivalent.

TRYINGtoOPEN
Indicates that you are trying to contact a foreign site to establish a connection.

Table 4. TCP Connection States

<table>
<thead>
<tr>
<th>TCP State</th>
<th>ConnectionStateType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td>NONEXISTENT</td>
</tr>
<tr>
<td>LAST-ACK, CLOSING, TIME-WAIT</td>
<td>If there is incoming data that the client program has not received, then RECEIVINGonly, else CONNECTIONclosing.</td>
</tr>
<tr>
<td>CLOSE-WAIT</td>
<td>If there is incoming data that the client program has not received, then OPEN, else SENDINGonly.</td>
</tr>
<tr>
<td>ESTABLISHED</td>
<td>OPEN</td>
</tr>
<tr>
<td>FIN-WAIT-1, FIN-WAIT-2</td>
<td>RECEIVINGonly</td>
</tr>
<tr>
<td>LISTEN</td>
<td>LISTENING</td>
</tr>
<tr>
<td>SYN-SENT, SYN-RECEIVED</td>
<td>TRYINGtoOPEN</td>
</tr>
</tbody>
</table>

Connection Information Record
The connection information record is used as an operand in several of the procedure calls. It enables you and the TCP/IP program to exchange information about the connection. The Pascal declaration is shown in Figure 16. For more information about the use of each field, see “TcpOpen and TcpWaitOpen” on page 82 and “TcpStatus” on page 85.

Connection
Specifies a number identifying the connection that is described. This connection number is different from the connection number displayed by the NETSTAT command. For more information about the NETSTAT command, see TCP/IP User’s Guide.

OpenAttemptTimeout
Specifies the number of seconds that TCP continues to attempt to open a
connection. You specify this number. If the limit is exceeded, TCP stops trying to open the connection and shuts down any partially open connection.

**Security, Compartment, Precedence**
Specifies entries used only when working within a multilevel secure environment.

**BytesToRead**
Specifies the number of data bytes received from the foreign host by TCP, but not yet delivered to the client. TCP maintains this value.

**UnackedBytes**
Specifies the number of bytes sent by your program, but not yet sent to the foreign TCP, or the number of bytes sent to the foreign TCP, but not yet acknowledged.

**LocalSocket**
 Specifies the local internet address and local port. Together, these form one end of a connection. The foreign socket forms the other end. For the Pascal declaration of the SocketType record, see Figure 17.

**ForeignSocket**
Specifies the foreign, or remote, internet address and its associated port. These form one end of a connection. The local socket forms the other end.

### Socket Record

```pascal
InternetAddressType = UnsignedIntegerType;
PortType = UnsignedHalfWordType;
SocketType = record
  Address: InternetAddressType;
  Port: PortType;
end;
```

*Figure 17. Pascal Declaration of Socket Type*

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Specifies the internet address.</td>
</tr>
<tr>
<td>Port</td>
<td>Specifies the port.</td>
</tr>
</tbody>
</table>

### Notification Record

The notification record is used to provide event information. You receive this information by using the GetNextNote call. For more information, see "GetNextNote" on page 61. It is a variant record; the number of fields is dependent on the type of notification. For the Pascal declaration of this record, see Figure 18 on page 47.
NotificationInfoType =
  record
  Connection: ConnectionType;
  Protocol: ProtocolType;
  case NotificationTag: NotificationEnumType of
    BUFFERspaceAVAILABLE:
      (AmountOfSpaceInBytes: integer);
    CONNECTIONstateCHANGED:
      (NewState: ConnectionStateType;
       Reason: CallReturnCodeType);
    DATAdelivered:
      (BytesDelivered: integer;
       LastUrgentByte: integer;
       PushFlag: Boolean);
    EXTERNALinterrupt:
      (RuptCode: integer);
    RECEIVEError:
      (ReceiveTurnCode: CallReturnCodeType;
       ReceiveRequestErr: Boolean);
    FSENDresponse:
      (SendTurnCode: CallReturnCodeType;
       SendRequestErr: Boolean);
    IOinterrupt:
      (DeviceAddress: integer;
       UnitStatus: UnsignedByteType;
       ChannelStatus: UnsignedByteType);
    IUCVinterrupt:
      (IUCVResponseBuf: IUCVBufferType);
    PINGresponse:
      (PingTurnCode: CallReturnCodeType;
       ElapsedTime: TimeStampType);
  end;

Figure 18. Notification Record (Part 1 of 2)
Connection
Indicates the client’s connection number to which the notification applies. In the case of USERdefinedNOTIFICATION, this field is as supplied by the user in the AddUserNote call.

Protocol
In the case of USERdefinedNOTIFICATION, this field is as supplied by the user in the AddUserNote call. For all other notifications, this field is reserved.

NotificationTag
Is the type of notification being sent, and a set of fields dependent on the value of the tag. Possible tag values relevant to the TCP/UDP/IP interface and the corresponding fields are:

BUFFERspaceAVAILABLE
Notification given when space becomes available on a connection for which TcpSend previously returned NObufferSPACE. For more information about these procedures, see "TcpSend, TcpSend, and TcpWaitSend" on page 79.

AmountOfSpaceInBytes
Indicates the minimum number of bytes that the TCP/IP service has available for buffer space for this connection. The actual amount of buffer space might be more than this number.
CONNECTIONStateCHANGED
Indicates that a TCP connection has changed state.

NewState
Indicates the new state for this connection.

Reason
Indicates the reason for the state change. This field is meaningful only if the NewState field has a value of NONEXISTENT.

Notes:
1. The following is the sequence of state notifications for a connection.
   - For active open:
     - OPEN
     - RECEIVINGonly or SENDINGonly
     - CONNECTIONclosing
     - NONEXISTENT.
   - For passive open:
     - TRYINGtoOPEN
     - OPEN
     - RECEIVINGonly or SENDINGonly
     - CONNECTIONclosing
     - NONEXISTENT.

   Your program should be prepared for any intermediate step or steps to be skipped.

2. The normal TCP connection closing sequence can lead to a connection staying in CONNECTIONclosing state for up to two minutes, corresponding to the TCP state TIME-WAIT.

3. Possible Reason codes giving the reason for a connection changing to NONEXISTENT are:
   - OK (means normal closing)
   - UNREACHABLEnetwork
   - TIMEOUTopen
   - OPENrejected
   - REMOTErset
   - WRONGsecORprc
   - UNEXPECTEDsyn
   - FATALerror
   - KILLEDbyCLIENT
   - TIMEOUTconnection
   - TCPipSHUTDOWN
   - DROPPEDbyOPERATOR.

DATAdelivered
Notification given when your buffer (named in an earlier TcpReceive or TcpFReceive request) contains data.

Note: The data delivered should be treated as part of a byte-stream, not as a message. There is no guarantee that the data sent in one TcpSend (or equivalent) call on the foreign host is delivered in a single DATAdelivered notification, even if the PushFlag is set.
**Pascal Language**

**BytesDelivered**
Indicates the number of bytes of data delivered to you.

**LastUrgentByte**
Indicates the number of bytes of urgent data remaining, including data just delivered.

**PushFlag**
TRUE if the last byte of data was received with the push bit set.

**EXTERNALInterrupt**
Notification given when a simulated external interrupt occurs in your virtual machine. The Connection and Protocol fields are not applicable.

**RuptCode**
The interrupt type.

**FRECEIVEerror**
Notification given in place of DATA delivered when a TcpFReceive that initially returned OK has terminated without delivering data.

**ReceiveTurnCode**
Specifies the reason the TcpFReceive has failed or was canceled. If `ReceiveRequestErr` is set to FALSE, ReceiveTurnCode contains the same reason as the Reason field in the CONNECTION stateCHANGED with NewState set to NONEXISTENT notification for this connection (see 2 on page 49). ReceiveTurnCode could be OK, if the connection closed normally.

**ReceiveRequestErr**
If TRUE, the TcpFReceive was rejected during initial processing. If FALSE, the TcpFReceive was initially accepted, but was terminated because of connection closing.

**Note:** Normally, you do not need to take any action upon receipt of this notification with `ReceiveRequestErr` set to FALSE, because your program receives a CONNECTION stateCHANGED notification informing it that the connection has been terminated.

**FSENDresponse**
Notification given when a TcpFSend request is completed, successfully or unsuccessfully.

**SendTurnCode**
Indicates the status of the send operation.

**SendRequestErr**
If TRUE, the TcpFSend was rejected during initial processing or during retry after buffer space became available. If FALSE, the TcpFSend was canceled because of connection closing.
**IOinterrupt**
Notification given when a simulated I/O interrupt occurs in your virtual machine. The Connection and Protocol fields are not applicable.

**DeviceAddress**
This address corresponds to the DEVICE statement.

**UnitStatus**
Specifies the status returned by the device.

**ChannelStatus**
Specifies the status returned by the channel.

**IUCVinterrupt**
Notification given when a simulated IUCV interrupt occurs in your virtual machine. The Connection and Protocol fields are not applicable.

**IUCVResponseBuf**
Contains the information returned from the application.

**PINGresponse**
Notification given when a PINGresponse is received.

**PingTurnCode**
Specifies the status of the ping operation.

**ElapsedTime**
Indicates the time elapsed between the sending of a request and the reception of a response. This time does not include the time spent in the simulated Virtual Machine Communication Facility (VMCF) communication between your program and the TCPIP virtual machine. This field is valid only if PingTurnCode has a value of OK.

**RAWIPpacketsDELIVERED**
Notification given when your buffer (indicated in an earlier RawIpReceive request) contains a datagram. Only one datagram is delivered on each notification. Your buffer contains the entire IP header, plus as much of the datagram as fits in your buffer.

**RawIpDataLength**
Specifies the actual data length delivered to your buffer. If this is less than RawIpFullLength, the datagram was truncated.

**RawIpFullLength**
Specifies the length of the packet, from the TotalLength field of the IP header.

**RAWIPspaceAVAILABLE**
When space becomes available after a client does a RawIpSend and receives a NObufferSPACE return code, the client receives this notification to indicate that space is now available.

**RawIpSpaceInBytes**
Specifies the amount of space available always equals the maximum size IP datagram.

**RESOURCESavailable**
Notice given when resources needed for a TcpOpen or
TcpWaitOpen are available. This notification is sent only if a previous TcpOpen or TcpWaitOpen returned ZEROresources.

**SMSGreceived**

Notification given when one or more Special Messages (Smgs) arrive. The GetSmsg call is used to retrieve queued Smgs. For information on the SMSG command, see [TCP/IP User’s Guide](#).

**TIMERexpired**

Notification given when a timer set through SetTimer expires.

**Datum**

Indicates the data specified when SetTimer was called.

**AssociatedTimer**

Specifies the address of the timer that expired.

**UDPdatagramDELIVERED**

Notification given when your buffer, indicated in an earlier UdpNReceive or UdpReceive request, contains a datagram. Your buffer contains the datagram excluding the UDP header.

**Note:** If UdpReceive was used, your buffer contains the entire datagram excluding the header, with the length indicated by DataLength. If UdpNReceive was used, and DataLength is less than FullLength, your buffer contains a truncated datagram. The reason is that the length of your buffer was too small to contain the entire datagram.

**DataLength**

Specifies the length of the data delivered to your buffer.

**ForeignSocket**

Specifies the source of the datagram.

**FullLength**

Specifies the length of the entire datagram, excluding the UDP header. This field is set only if UdpNReceive was used.

**UDPdatagramSPACEavailable**

Notification given when buffer space becomes available for a datagram for which UdpSend previously returned NObufferSPACE because of insufficient resources.

**UDPresourcesAVAILABLE**

Notice given when resources needed for a UdpOpen are available. This notification is sent only if a previous UdpOpen returned UDPzeroRESOURCES.

**URGENTpending**

Notification given when there is urgent data pending on a TCP connection.

**BytesToRead**

Indicates the number of incoming bytes not yet delivered to the client.
Pascal Language

UrgentSpan
Indicates the number of undelivered bytes to the last known urgent pointer. No urgent data is pending if this is negative.

USERdefinedNOTIFICATION
Notice generated from data passed to AddUserNote by your program.

UserData
A 40-byte field supplied by your program through AddUserNote. The Connection and Protocol fields are also set from the values supplied to AddUserNote.

File Specification Record
The file specification record is used to fully specify a file. The Pascal declaration is shown in Figure 19.

SpecOfFile_subtype =
record
  Owner: DirectoryNameType;
  Case SpecOfSystem_subtype of
    VM:
      (VirtualAddress:VirtualAddressType;
       NewVirtualAddress:VirtualAddressType;
       DiskPassword: DirectoryNameType;
       Filename: DirectoryNameType;
       Filetype: DirectoryNameType;
       Filemode: FileModeType)
    MVS:
      (* The MVS declaration is listed here. *)
  );
end;

Figure 19. Pascal Declaration of File Specification Record

Using Procedure Calls
Your program uses procedure calls to initiate communication with the TCPIP virtual machine. Most of these procedure calls return with a code, which indicates success or the type of failure incurred by the call. For an explanation of the return codes, see Table 87 on page 357.

Before invoking any of the other interface procedures, use BeginTcpIp or StartTcpNotice to start up the TCP/UDP/IP service. Once the TCP/UDP/IP service has begun, use the Handle procedure to specify a set of notifications that the TCP/UDP/IP service can send you. To terminate the TCP/UDP/IP service, use the EndTcpIp procedure.

Notifications
The TCPIP virtual machine sends you notifications to inform you of asynchronous events. Also, some notifications are generated in your virtual machine by the TCP interface. Notifications can be received only after BeginTcp or StartTcpNotice.
The notifications are received by the TCP interface and kept in a queue. Use GetNextNote to get the next notification. The notifications are in Pascal variant record form. For more information, see Figure 18 on page 47.

The following table provides a short description of the Notification procedure calls and gives the page number where each call’s detailed description is located.

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetNextNote</td>
<td>Retrieves the next notification.</td>
<td>61</td>
</tr>
<tr>
<td>Handle</td>
<td>Specifies the types of notifications that your program can process.</td>
<td>62</td>
</tr>
<tr>
<td>NotifyIo</td>
<td>Requests that an IOinterrupt notification be sent to you when an I/O interrupt occurs on a given virtual address.</td>
<td>57</td>
</tr>
<tr>
<td>Unhandle</td>
<td>Specifies notification types that your program can no longer process.</td>
<td>57</td>
</tr>
<tr>
<td>UnNotifyIo</td>
<td>Indicates that you no longer wish to be sent a notification when an I/O interrupt occurs on a given virtual address.</td>
<td>57</td>
</tr>
</tbody>
</table>

TCP/UDP Initialization Procedures

The UDP initialization procedures affect all present and future connections. Use these procedures to initialize the TCP/IP environment for your program.

The following table provides a short description of the TCP/UDP Initialization procedure calls and gives the page number where each call’s detailed description is located.

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TcpNameChange</td>
<td>Identifies the name of the virtual machine running the TCP/IP program when the virtual machine has a name other than TCPIP.</td>
<td>81</td>
</tr>
<tr>
<td>BeginTcpIp</td>
<td>Establishes communication with the TCP/IP services.</td>
<td>58</td>
</tr>
<tr>
<td>StartTcpNotice</td>
<td>Establishes communication with the TCP/IP services.</td>
<td>74</td>
</tr>
</tbody>
</table>

TCP/UDP Termination Procedure

The Pascal API has one termination procedure call. You should use the EndTcpIp call when you have finished with the TCP/IP services.

The following table provides a short description of the TCP/UDP Termination procedure call and gives the page number where the call’s detailed description is located.

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndTcpIp</td>
<td>Terminates communication with the TCP/IP services.</td>
<td>59</td>
</tr>
</tbody>
</table>
Handling External Interrupts

The handling external interrupts procedures allow you to pass simulated external interrupts to the TCP interface. You must call the StartTcpNotice initialization routine before you can begin using the external interrupt calls.

The following table provides a short description of the Handling External Interrupts and gives the page number where each call’s detailed description is located.

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TcpExtRupt</td>
<td>Notifies the TCP interface of the arrival of a simulated external interrupt.</td>
<td>76</td>
</tr>
<tr>
<td>RTcpExtRupt</td>
<td>A version of TcpExtRupt.</td>
<td>71</td>
</tr>
<tr>
<td>TcpVmcfRupt</td>
<td>Notifies the TCP interface of the arrival of a simulated VMCF interrupt.</td>
<td>86</td>
</tr>
<tr>
<td>RTcpVmcfRupt</td>
<td>A version of TcpVmcfRupt.</td>
<td>71</td>
</tr>
</tbody>
</table>

TCP Communication Procedures

The TCP communication procedures apply to a particular client connection. Use these procedures to establish a connection and to communicate. You must call either the BeginTcpIp or the StartTcpNotice initialization routine before you can begin using TCP communication procedures.

The following table provides a short description of the TCP communication procedures and gives the page number where each call’s detailed description is located.

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TcpOpen</td>
<td>Initiates a TCP connection.</td>
<td>82</td>
</tr>
<tr>
<td>TcpWaitOpen</td>
<td>Initiates a TCP connection and waits for the establishment of the connection.</td>
<td>82</td>
</tr>
<tr>
<td>TcpFSend</td>
<td>Sends TCP data.</td>
<td>79</td>
</tr>
<tr>
<td>TcpSend</td>
<td>Sends TCP data.</td>
<td>79</td>
</tr>
<tr>
<td>TcpWaitSend</td>
<td>Sends TCP data and waits until TCPIP accepts it.</td>
<td>79</td>
</tr>
<tr>
<td>TcpFReceive</td>
<td>Establishes a buffer to receive TCP data.</td>
<td>76</td>
</tr>
<tr>
<td>TcpReceive</td>
<td>Establishes a buffer to receive TCP data.</td>
<td>76</td>
</tr>
<tr>
<td>TcpWaitReceive</td>
<td>Establishes a buffer to receive TCP data and waits for the reception of the data.</td>
<td>76</td>
</tr>
<tr>
<td>TcpClose</td>
<td>Begins the TCP one-way closing sequence.</td>
<td>75</td>
</tr>
<tr>
<td>TcpAbort</td>
<td>Shuts down a TCP connection immediately.</td>
<td>75</td>
</tr>
<tr>
<td>TcpStatus</td>
<td>Obtains the current status of a TCP connection.</td>
<td>85</td>
</tr>
</tbody>
</table>

Ping Interface

The Ping interface lets a client send an ICMP echo request to a foreign host. You must call either the BeginTcpIp or the StartTcpNotice initialization routine before you can begin using the Ping Interface.
The following table provides a short description of the Ping interface procedures and gives the page number where each call’s detailed description is located.

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PingRequest</td>
<td>Sends an Internet Control Message Protocol (ICMP) echo request.</td>
<td>67</td>
</tr>
</tbody>
</table>

**Monitor Procedures**

Two monitor procedures, MonCommand and MonQuery, provide a mechanism for querying and controlling the TCPIP virtual machine.

The following table provides a short description of the Monitor procedures and gives the page number where each call’s detailed description is located.

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MonCommand</td>
<td>Instructs TCP to read a specific file and execute the commands that it contains.</td>
<td>64</td>
</tr>
<tr>
<td>MonQuery</td>
<td>Performs control functions and retrieves internal TCPIP control blocks.</td>
<td>65</td>
</tr>
</tbody>
</table>

**UDP Communication Procedures**

The UDP communication procedures describe the programming interface for the User Datagram Protocol (UDP) provided in the TCP/IP product.

The following table provides a short description of the UDP communication procedures and gives the page number where each call’s detailed description is located.

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UdpOpen</td>
<td>Requests communication with UDP on a specified socket.</td>
<td>87</td>
</tr>
<tr>
<td>UdpSend</td>
<td>Sends a UDP datagram to a specified foreign socket.</td>
<td>89</td>
</tr>
<tr>
<td>UdpNReceive</td>
<td>Notifies the TCPIP virtual machine that you are willing to receive UDP datagram data.</td>
<td>87</td>
</tr>
<tr>
<td>UdpReceive</td>
<td>Notifies the TCPIP virtual machine that you are willing to receive UDP datagram data.</td>
<td>88</td>
</tr>
<tr>
<td>UdpClose</td>
<td>Terminates use of a UDP socket.</td>
<td>86</td>
</tr>
</tbody>
</table>

**Raw IP Interface**

The Raw IP interface lets a client program send and receive arbitrary IP packets on any IP protocol except TCP and UDP. Only one client can use any given protocol at one time. Only clients in the obey list can use the Raw IP interface. For further information about the obey list, see [TCP/IP Planning and Customization](#).

The following table provides a short description of the Raw IP interface procedures and gives the page number where each call’s detailed description is located.
Table 13. Pascal Language Interface Summary—Raw IP Interface

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RawIpOpen</td>
<td>Informs the TCPIP virtual machine that the client wants to send and receive IP packets of a specified protocol.</td>
<td>68</td>
</tr>
<tr>
<td>RawIpReceive</td>
<td>Specifies a buffer to receive raw IP packets of a specified protocol.</td>
<td>58</td>
</tr>
<tr>
<td>RawIpSend</td>
<td>Sends raw IP packets of a specified protocol.</td>
<td>59</td>
</tr>
<tr>
<td>RawIpClose</td>
<td>Informs the TCPIP virtual machine that the client no longer handles the protocol.</td>
<td>57</td>
</tr>
</tbody>
</table>

Timer Routines

The timer routines are used with the TCP/UDP/IP interface. You must call either the BeginTcpIp or the StartTcpNotice initialization routine before you can begin using the timer routines.

The following table provides a short description of the Timer routines and gives the page number where each call’s detailed description is located.

Table 14. Pascal Language Interface Summary—Timer Routines

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateTimer</td>
<td>Allocates a timer.</td>
<td>59</td>
</tr>
<tr>
<td>ClearTimer</td>
<td>Resets a timer.</td>
<td>59</td>
</tr>
<tr>
<td>SetTimer</td>
<td>Sets a timer to expire after a specified interval.</td>
<td>74</td>
</tr>
<tr>
<td>DestroyTimer</td>
<td>Deallocates a timer.</td>
<td>59</td>
</tr>
</tbody>
</table>

Host Lookup Routines

The host lookup routines (with the exception of GetHostResol ) are declared in the CMINTER member of the ALLMACRO MACLIB. The host lookup routine GetHostResol is declared in the CMRESGLB member of the ALLMACRO MACLIB. Any program using these procedures must include CMINTER or CMRESGLB after the include statements for CMCOMM and CMCLIEN.

The following table provides a short description of the host lookup routines and gives the page number where each call’s detailed description is located.

Table 15. Pascal Language Interface Summary—Host Lookup Routines

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetHostName</td>
<td>Converts a host name to an internet address using static tables.</td>
<td>51</td>
</tr>
<tr>
<td>GetHostResol</td>
<td>Converts a host name to an internet address using a domain name resolver.</td>
<td>51</td>
</tr>
<tr>
<td>GetHostString</td>
<td>Converts an internet address to a host name using static tables.</td>
<td>51</td>
</tr>
<tr>
<td>GetIdentity</td>
<td>Returns environment information.</td>
<td>61</td>
</tr>
<tr>
<td>IsLocalAddress</td>
<td>Determines if an internet address is local.</td>
<td>53</td>
</tr>
<tr>
<td>IsLocalHost</td>
<td>Determines if a host name is local, remote, loopback, or unknown.</td>
<td>53</td>
</tr>
</tbody>
</table>
Pascal Language

AddUserNote

The AddUserNote procedure can be called to add a USERdefinedNOTIFICATION notification to the note queue and wake up GetNextNote if it is waiting for a notification. For more information, see "RTcpExtRupt" on page 71 and "RTcpVmcfRupt" on page 71.

Other Routines

The following table provides a short description of these procedure calls and gives the page number where the detailed description is located.

Table 16. Pascal Language Interface Summary—Other Routines

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetSmsg</td>
<td>Retrieves one queued special message (Smsg).</td>
<td>62</td>
</tr>
<tr>
<td>ReadXlateTable</td>
<td>Reads a binary translation table file.</td>
<td>70</td>
</tr>
<tr>
<td>SayCalRe</td>
<td>Converts a return code into a descriptive message.</td>
<td>72</td>
</tr>
<tr>
<td>SayConSt</td>
<td>Converts a connection state into a descriptive message.</td>
<td>72</td>
</tr>
<tr>
<td>SayIntAd</td>
<td>Converts an internet address into a name or dotted-decimal form.</td>
<td>72</td>
</tr>
<tr>
<td>SayIntNum</td>
<td>Converts an internet address into its dotted-decimal form.</td>
<td>73</td>
</tr>
<tr>
<td>SayNotEn</td>
<td>Converts a notification enumeration type into a descriptive message.</td>
<td>73</td>
</tr>
<tr>
<td>SayPorTy</td>
<td>Converts a port number into a descriptive message or into EBCDIC.</td>
<td>73</td>
</tr>
<tr>
<td>SayProTy</td>
<td>Converts the protocol type into a descriptive message or into EBCDIC.</td>
<td>73</td>
</tr>
<tr>
<td>AddUserNote</td>
<td>Adds a USERdefinedNOTIFICATION notification to the note queue.</td>
<td>58</td>
</tr>
</tbody>
</table>

Procedure Calls

This section provides the syntax, operands, and other appropriate information for each Pascal procedure call supported by TCP/IP for VM.

BeginTcpIp

The BeginTcpIp procedures inform the TCPIP virtual machine that you want to start using its services. If your program handles simulated external interrupts itself, use StartTcpNotice rather than BeginTcpIp. For information about simulated external interrupt support, see Chapter 3, "Virtual Machine Communication Facility Interface" on page 93.

procedure BeginTcpIp

    var ReturnCode: integer
    );
    external;

Operand       Description
ReturnCode     Indicates success or failure of call. Possible return values are:
• OK
• ABNORMALcondition
• fatalerror
• NOtcpIPservice
• TCPipshutdown
• VIRTUALmemoryTOOsmall

For a description of Pascal ReturnCodes, see Appendix B, "Pascal Return Codes" on page 357.

ClearTimer
The ClearTimer procedure resets the timer to prevent it from timing out.

```
procedure ClearTimer
(
    T: TimerPointerType
); external;
```

Operand Description

T Specifies a timer pointer, as returned by a previous CreateTimer call.

CreateTimer
The CreateTimer procedure allocates a timer. The timer is not set in any way. For the procedure to activate the timer, see "SetTimer" on page 74.

```
procedure CreateTimer
(
    var T: TimerPointerType
); external;
```

Operand Description

T Sets to a timer pointer that can be used in subsequent SetTimer, ClearTimer, and DestroyTimer calls.

DestroyTimer
The DestroyTimer procedure deallocates or frees a timer that you created.

```
procedure DestroyTimer
(
    var T: TimerPointerType
); external;
```

Operand Description

T Specifies a timer pointer, as returned by a previous CreateTimer call.

EndTcpIp
The EndTcpIp procedure releases ports and protocols in use that are not permanently reserved. It causes TCP to clean up any data structures it has associated with you. Use EndTcpIp when you have finished with the TCP/IP services.
GetHostNumber

The GetHostNumber procedure resolves a host name into an internet address.

GetHostNumber uses a table lookup to convert the name of a host to an internet address, and returns this address to the HostNumber field. When the name is a dotted-decimal number, GetHostNumber returns the integer represented by that dotted-decimal. The dotted-decimal representation of a 32-bit number has one decimal integer for each of the 4 bytes, separated by dots. For example, 14.0.0.7 for X'0E000007'. For information about how to create host lookup tables, see TCP/IP Planning and Customization.

The HostNumber field is set to NOhost if the host is not found.

Operand Description
Name Specifies the name or dotted-decimal number to be converted.
HostNumber Set to the converted address, or NOhost if conversion fails.

GetHostResol

The GetHostResol procedure resolves a host name into an internet address by using a name server.

GetHostResol passes the query to the remote name server through the resolver. The name server converts the name of a host to an internet address, and returns this address in the HostNumber field. If the name server does not respond or does not find the name, the host name is converted to a host number by table lookup. When the name is a dotted-decimal number, the integer represented by that dotted-decimal is returned. The dotted-decimal representation of a 32-bit number has one decimal integer for each of the 4 bytes, separated by dots. For example, 14.0.0.7 for X'0E000007'.

The HostNumber field is set to NOhost if the host is not found.

Operand Description
Name Specifies the name or dotted-decimal number to be converted.
HostNumber Set to the converted address, or NOhost if conversion fails.
GetHostString

The GetHostString procedure uses a table lookup to convert an internet address to a host name, and returns this string in the Name field. The first host name found in the lookup is returned. If no host name is found, a gateway or network name is returned. If no gateway or network name is found, a null string is returned.

```
procedure GetHostString
   (   Address: InternetAddressType;
       var Name: SiteNameType
   );
external;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Specifies the address to be converted.</td>
</tr>
<tr>
<td>Name</td>
<td>Set to the corresponding host, gateway, or network name, or to null string if no match found.</td>
</tr>
</tbody>
</table>

GetIdentity

The GetIdentity procedure returns the following information:
- The user ID of the VM user
- The host machine name
- The network domain name
- The user ID of the TCPIP virtual machine.

The host machine name and domain name are extracted from the HOSTNAME and DOMAINORIGIN statements, respectively, in the `user_id` DATA file. If the `user_id` DATA file does not exist, the TCPIP DATA file is used. If a HOSTNAME statement is not specified, then the default host machine name is the name specified by the TCP/IP installer during installation. See [TCP/IP Planning and Customization](#). The TCPIP virtual machine user ID is extracted from the TCPIPUuserid statement in the `user_id` DATA file; if the statement is not specified, the default is TCPIP.

```
procedure GetIdentity
   (   var UserId: DirectoryNameType;
       var HostName, DomainName: String;
       var TcpIpServiceName: DirectoryNameType;
       var Result: integer
   );
external;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserId</td>
<td>Specifies the user ID of the VM user or the job name of a batch job that has invoked GetIdentity.</td>
</tr>
<tr>
<td>HostName</td>
<td>Specifies the host machine name.</td>
</tr>
<tr>
<td>DomainName</td>
<td>Specifies the network domain name.</td>
</tr>
<tr>
<td>TcpIpServiceName</td>
<td>Specifies the user ID of the TCPIP virtual machine.</td>
</tr>
</tbody>
</table>

GetNextNote

The GetNextNote procedure retrieves notifications from the queue. This procedure returns the next notification queued for you.
GetNextNote

procedure GetNextNote
(
  var Note: NotificationInfoType;
  ShouldWait: Boolean;
  var ReturnCode: integer
);
external;

Operand Description

Note Indicates that the next notification is stored here when ReturnCode is OK.

ShouldWait Sets ShouldWait to TRUE if you want GetNextNote to wait until a notification becomes available. Set ShouldWait to FALSE if you want GetNextNote to return immediately. When ShouldWait is set to FALSE, ReturnCode is set to NOoutstandingNOTIFICATIONS if no notification is currently queued.

ReturnCode Indicates the success or failure of the call. Possible return values are:
  • OK
  • NOoutstandingNOTIFICATIONS
  • NOTyetBEGUN

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

GetSmsg

The GetSmsg procedure is called by your program after receiving an SMSGreceived notification. Each call to GetSmsg retrieves one queued Smsg. Your program should exhaust all queued Smsgs, by calling GetSmsg repeatedly until the Success field returns with a value of FALSE. After a value of FALSE is returned, do not call GetSmsg again until you receive another SMSGreceived notification.

For information about the SMSG command, see TCP/IP User’s Guide

procedure GetSmsg
(
  var Smsg: SmsgType;
  var Success: Boolean;
);
external;

Operand Description

Smsg Set to the returned Smsg if Success is set to TRUE.

Success TRUE if Smsg returned, otherwise FALSE.

Handle

The Handle procedure specifies that you want to receive notifications in the given set. You must always use it after calling the BeginTcpIp procedure and before accessing the TCP/IP services. This Pascal set can contain any of the NotificationEnumType values shown in Figure 18 on page 47.
**Handle**

```pascal
procedure Handle(   
    Notifications: NotificationSetType;  
    var ReturnCode: integer  
);  
external;
```

**Operand Description**

**Notifications**  
Specifies the set of notification types to be handled.

**ReturnCode**  
Indicates the success or failure of the call. Possible return values are:
- OK
- NOTyetBEGUN
- TCPipSHUTDOWN
- ABNORMALcondition
- FATALerror

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

**LocalAddress**

The IsLocalAddress procedure queries the TCPIP virtual machine to determine whether the HostAddress is one of the addresses recognized for this host. If the address is local, it returns OK. If the address is not local, it returns NONlocalADDRESS.

```pascal
procedure IsLocalAddress(   
    HostAddress: InternetAddressType;  
    var ReturnCode: integer  
);  
external;
```

**Operand Description**

**HostAddress**  
Specifies the host address to be tested.

**ReturnCode**  
Indicates whether the host address is local, or may indicate an error. Possible return values are:
- OK
- NONlocalADDRESS
- TCPipSHUTDOWN
- ABNORMALcondition
- FATALerror

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

**IsLocalHost**

The IsLocalHost procedure returns the correct host class for Name, which may be a host name or a dotted-decimal address.

The host classes are:

<table>
<thead>
<tr>
<th>Host Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOSTlocal</td>
<td>Specifies an internet address for the local host.</td>
</tr>
</tbody>
</table>
**HOSTloopback**
Specifies one of the dummy internet addresses used to designate various levels of loopback testing.

**HOSTremote**
Specifies a known host name for some remote host.

**HOSTunknown**
Specifies an unknown host name (or other error).

---

**Operand Description**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td>Specifies the host name.</td>
</tr>
<tr>
<td><strong>Class</strong></td>
<td>Specifies the host class.</td>
</tr>
</tbody>
</table>

---

**MonCommand**

The MonCommand procedure instructs the TCPIP virtual machine to read a specific file and execute the commands found there. This procedure updates TCPIP internal tables and parameters while the TCPIP virtual machine is running. For example, the type and destination of run-time tracing can be modified dynamically using MonCommand. This procedure is used by the OBEYFILE command. For more information about the OBEYFILE command, see [TCP/IP Planning and Customization](#). You must be in the TCPIP obey list to use the MonCommand procedure.

**Operand Description**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FileSpec</strong></td>
<td>Specifies a file in a manner that allows access to that file. The TCPIP virtual machine must be authorized to access the file. The SpecOfFileType record is listed in <a href="#">Figure 19 on page 53</a>.</td>
</tr>
</tbody>
</table>
| **ReturnCode** | Indicates the success or failure of the call. Possible return values are:  
  - OK  
  - ABNORMALcondition  
  - ERRORinPROFILE  
  - HASnoPASSWORD  
  - INCORRECTpassword  
  - INVALIDuserID  
  - INVALIDvirtualADDRESS  
  - MINIDISKinUSE  
  - MINIDISKnotAVAILABLE  
  - NOTyetBEGUN  
  - PROFILEnotFOUND  
  - SOFTWAREerror  
  - TCPipSHUTDOWN |

---

**IsLocalHost**

**HOSTloopback**
Specifies one of the dummy internet addresses used to designate various levels of loopback testing.

**HOSTremote**
Specifies a known host name for some remote host.

**HOSTunknown**
Specifies an unknown host name (or other error).

---

**Operand Description**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td>Specifies the host name.</td>
</tr>
<tr>
<td><strong>Class</strong></td>
<td>Specifies the host class.</td>
</tr>
</tbody>
</table>

---

**MonCommand**

The MonCommand procedure instructs the TCPIP virtual machine to read a specific file and execute the commands found there. This procedure updates TCPIP internal tables and parameters while the TCPIP virtual machine is running. For example, the type and destination of run-time tracing can be modified dynamically using MonCommand. This procedure is used by the OBEYFILE command. For more information about the OBEYFILE command, see [TCP/IP Planning and Customization](#). You must be in the TCPIP obey list to use the MonCommand procedure.

**Operand Description**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FileSpec</strong></td>
<td>Specifies a file in a manner that allows access to that file. The TCPIP virtual machine must be authorized to access the file. The SpecOfFileType record is listed in <a href="#">Figure 19 on page 53</a>.</td>
</tr>
</tbody>
</table>
| **ReturnCode** | Indicates the success or failure of the call. Possible return values are:  
  - OK  
  - ABNORMALcondition  
  - ERRORinPROFILE  
  - HASnoPASSWORD  
  - INCORRECTpassword  
  - INVALIDuserID  
  - INVALIDvirtualADDRESS  
  - MINIDISKinUSE  
  - MINIDISKnotAVAILABLE  
  - NOTyetBEGUN  
  - PROFILEnotFOUND  
  - SOFTWAREerror  
  - TCPipSHUTDOWN |

---

**IsLocalHost**

**HOSTloopback**
Specifies one of the dummy internet addresses used to designate various levels of loopback testing.

**HOSTremote**
Specifies a known host name for some remote host.

**HOSTunknown**
Specifies an unknown host name (or other error).

---

**Operand Description**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td>Specifies the host name.</td>
</tr>
<tr>
<td><strong>Class</strong></td>
<td>Specifies the host class.</td>
</tr>
</tbody>
</table>

---

**MonCommand**

The MonCommand procedure instructs the TCPIP virtual machine to read a specific file and execute the commands found there. This procedure updates TCPIP internal tables and parameters while the TCPIP virtual machine is running. For example, the type and destination of run-time tracing can be modified dynamically using MonCommand. This procedure is used by the OBEYFILE command. For more information about the OBEYFILE command, see [TCP/IP Planning and Customization](#). You must be in the TCPIP obey list to use the MonCommand procedure.

**Operand Description**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FileSpec</strong></td>
<td>Specifies a file in a manner that allows access to that file. The TCPIP virtual machine must be authorized to access the file. The SpecOfFileType record is listed in <a href="#">Figure 19 on page 53</a>.</td>
</tr>
</tbody>
</table>
| **ReturnCode** | Indicates the success or failure of the call. Possible return values are:  
  - OK  
  - ABNORMALcondition  
  - ERRORinPROFILE  
  - HASnoPASSWORD  
  - INCORRECTpassword  
  - INVALIDuserID  
  - INVALIDvirtualADDRESS  
  - MINIDISKinUSE  
  - MINIDISKnotAVAILABLE  
  - NOTyetBEGUN  
  - PROFILEnotFOUND  
  - SOFTWAREerror  
  - TCPipSHUTDOWN |
MonCommand

- UNAUTHORIZEDUser
- UNIMPLEMENTEDRequest

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

MonQuery

The MonQuery procedure obtains status information, or requests TCPIP to perform certain actions. This procedure is used by the NETSTAT command. For more information about the NETSTAT command, see TCP/IP User’s Guide.

procedure MonQuery
{
  QueryRecord: MonQueryRecordType;
  Buffer: integer;
  BufSize: integer;
  var ReturnCode: integer;
  var Length: integer
};

external;

Operand  Description
Buffer    Specifies the address of the buffer to receive data.
BufSize   Specifies the size of the buffer.
ReturnCode Indicates the success or failure of the call.
Length    Specifies the length of the data returned in the buffer.

QueryRecord Sets up a QueryRecord to specify the type of status information to be retrieved. The MonQueryRecordType is shown in Figure 20.

MonQueryRecordType =
record
  case QueryType: MonQueryType of
    QUERYhome, QUERYgateways, QUERYcontrolBLOCKS,
    QUERYstartTIME, QUERYtelnetSTATUS,
    QUERYdevicesANDlinks,
    QUERYhomeONLY: ();
    QUERYudpPORTowner:
    {   
      QueryPort: PortType
    };        
    COMMANDcpCMD:
    {   
      CpCmd: WordType
    };        
    COMMANDdropCONNECTION:
    {   
      Connection: ConnectionType
    };        
  end; { MonQueryRecordType }

Figure 20. Monitor Query Record

The only QueryType values available for your use are:

QUERYhomeONLY

Used to obtain a list of the home internet addresses recognized by your TCPIP virtual machine. Your program sets the Buffer to the address of a variable of type HomeOnlyListType, and the BufSize to its length. When
MonQuery

MonQuery returns, Length is set to the length of the Buffer that was used, if ReturnCode is OK. Divide the Length by size of (InternetAddressType) to get the number of the home addresses that are returned.

COMMANDdropCONNECTION

Used to instruct the TCPIP virtual machine to drop a TCP connection. The connection is reset, and the client process owning the connection is sent a NONEXISTENT notification with the Reason field set to DROPPEDbyOPERATOR. Your program sets the Connection field to the number of the connection to be dropped. The connection number is the number displayed by the NETSTAT CONN or the NETSTAT TELNET command, and is not the same number used to refer to the connection by the client program that owns the connection. For information about the NETSTAT command, see TCP/IP User's Guide. The virtual machine running your program that uses COMMANDdropCONNECTION must be in the TCPIP virtual machine.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror
- NOTyetBEGUN
- TCPipSHUTDOWN
- UNAUTHORIZEDUser
- UNIMPLEMENTEDrequest

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

NotifyIo

The NotifyIo procedure is used to request that an IOinterrupt notification be sent to you when an I/O interrupt occurs on a given virtual address. You can specify that you wish notifications on up to 10 different virtual device addresses (by means of individual NotifyIo calls). This notification is intended for unsolicited interrupts, not for interrupts showing the completion of a channel program.

procedure NotifyIo
(
    DeviceAddress: integer;
    var ReturnCode: integer;
); external;

Operand Description

DeviceAddress

Specifies the address of the device for which IOinterrupt notifications are to be generated.

ReturnCode

Indicates success or failure of the call. Possible return values are:

- OK
- TOOmanyOPENS
- SOFTWAREerror

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.
PingRequest

The PingRequest procedure sends an ICMP echo request to a foreign host. When a response is received or the time-out limit is reached, you receive a PingResponse notification.

The PingRequest procedure is used by the PING user command. For more information about the PING command, see TCP/IP Planning and Customization.

```
procedure PingRequest
(   ForeignAddress: InternetAddressType;
    Length: integer;
    Timeout: integer;
    var ReturnCode: integer
); external;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForeignAddress</td>
<td>Specifies the address of the foreign host to be pinged.</td>
</tr>
<tr>
<td>Length</td>
<td>Specifies the length of the ping packet, excluding the IP header. The range of values for this field are 8 to 512 bytes.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Specifies how long to wait for a response, in seconds.</td>
</tr>
<tr>
<td>ReturnCode</td>
<td>Indicates the success or failure of the call. Possible return values are:</td>
</tr>
<tr>
<td></td>
<td>• OK</td>
</tr>
<tr>
<td></td>
<td>• ABNORMALcondition</td>
</tr>
<tr>
<td></td>
<td>• BADlengthARGUMENT</td>
</tr>
<tr>
<td></td>
<td>• CONNECTIONalreadyEXISTS</td>
</tr>
<tr>
<td></td>
<td>• NObufferSPACE</td>
</tr>
<tr>
<td></td>
<td>• NOTyetBEGUN</td>
</tr>
</tbody>
</table>

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

Note: CONNECTIONalreadyEXISTS, in this context, means a ping request is already outstanding.

RawIpClose

The RawIpClose procedure tells the TCPIP virtual machine that the client does not handle the protocol any longer. Any queued incoming packets are discarded.

When the client is not handling the protocol, a return code of NOSuchCONNECTION is received.

```
procedure RawIpClose
(   ProtocolNo: integer;
    var ReturnCode: integer
); external;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProtocolNo</td>
<td>Specifies the number of the IP protocol.</td>
</tr>
</tbody>
</table>
RawIpClose

**ReturnCode**  Indicates the success or failure of the call. Possible return values are:

- OK
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPIpSHUTDOWN
- UNAUTHORIZEDuser

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

**RawIpOpen**

The RawIpOpen procedure tells the TCPIP virtual machine that the client wants to send and receive packets of the specified protocol.

You cannot use protocols 6 and 17. They specify the TCP (6) and UDP (17) protocols. When you specify 6, 17, or a protocol that has been opened by another virtual machine, you receive the LOCALportNOTavailable return code.

```
procedure RawIpOpen
  (  ProtocolNo: integer;
    var ReturnCode: integer
  );
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProtocolNo</td>
<td>Specifies the number of the IP protocol.</td>
</tr>
<tr>
<td>ReturnCode</td>
<td>Indicates the success or failure of the call. Possible return values are:</td>
</tr>
<tr>
<td></td>
<td>- OK</td>
</tr>
<tr>
<td></td>
<td>- LOCALportNOTavailable</td>
</tr>
<tr>
<td></td>
<td>- NOTyetBEGUN</td>
</tr>
<tr>
<td></td>
<td>- SOFTWAREerror</td>
</tr>
<tr>
<td></td>
<td>- TCPIpSHUTDOWN</td>
</tr>
<tr>
<td></td>
<td>- UNAUTHORIZEDuser</td>
</tr>
</tbody>
</table>

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

**Note:** You can open the ICMP protocol, but your program receives only those ICMP packets that are not interpreted by the TCPIP virtual machine.

**RawIpReceive**

The RawIpReceive procedure specifies a buffer to receive Raw IP packets of the specified protocol. You get the notification RAWIPpacketsDELIVERED when a packet is put in the buffer.
RawIpReceive

procedure RawIpReceive
{
    ProtocolNo: integer;
    Buffer: Address31Type;
    BufferLength: integer;
    var ReturnCode: integer
}

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

RawIpSend

The RawIpSend procedure sends IP packets of the given protocol number. The entire packet, including the IP header, must be in the buffer. The TCPIP virtual machine uses the total length field of the IP header to determine where each packet ends. Subsequent packets begin at the next doubleword (8-byte) boundary within the buffer.

The packets in your buffer are transmitted as is with the following exceptions.

- They can be fragmented. The fragment offset and flag fields in the header are filled.
- The version field in the header is filled.
- The checksum field in the header is filled.
- The source address field in the header is filled.

You get the return code NOsuchCONNECTION if the client is not handling the protocol, or if a packet in the buffer has another protocol. The return code BADlengthARGUMENT is received when:

- The DataLength is less than 40 bytes or more than 8K bytes.
- NumPackets is 0.
- A packet is greater than 2048 bytes.
- All packets do not fit into DataLength.

A ReturnCode value of NObufferSPACE indicates that the data is rejected because TCPIP is out of buffers. When buffer space is available, the notification RAWIPspaceAVAILABLE is sent to the client.
**RawIpSend**

```pascal
procedure RawIpSend
(
  ProtocolNo: integer;
  Buffer: Address31Type;
  DataLength: integer;
  NumPackets: integers;
  var ReturnCode: integer
); external;
```

**Operand Description**

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProtocolNo</td>
<td>Specifies the number of the IP protocol.</td>
</tr>
<tr>
<td>Buffer</td>
<td>Specifies the address of your buffer containing packets to send.</td>
</tr>
<tr>
<td>DataLength</td>
<td>Specifies the total length of data in your buffer.</td>
</tr>
<tr>
<td>NumPackets</td>
<td>Specifies the number of packets in your buffer.</td>
</tr>
<tr>
<td>ReturnCode</td>
<td>Indicates the success or failure of the call. Possible return values are:</td>
</tr>
<tr>
<td></td>
<td>• OK</td>
</tr>
<tr>
<td></td>
<td>• BADlengthARGUMENT</td>
</tr>
<tr>
<td></td>
<td>• NObufferSPACE</td>
</tr>
<tr>
<td></td>
<td>• NOsuchCONNECTION</td>
</tr>
<tr>
<td></td>
<td>• NOTyetBEGUN</td>
</tr>
<tr>
<td></td>
<td>• SOFTWAREerror</td>
</tr>
<tr>
<td></td>
<td>• TCPIPshutdown</td>
</tr>
<tr>
<td></td>
<td>• UNAUTHORIZEDuser</td>
</tr>
</tbody>
</table>

For a description of Pascal ReturnCodes, see Appendix B, "Pascal Return Codes" on page 357.

**Note:** If your buffer contains multiple packets to send, some of the packets may have been sent even if ReturnCode is not OK.

**ReadXlateTable**

The ReadXlateTable procedure reads the binary translation table file specified by TableName, and fills in the AtoETable and EtoATable translation tables.

```pascal
procedure ReadXlateTable
(
  var TableName: DirectoryNameType;
  var AtoETable: AtoEType;
  var EtoATable: EtoAType;
  var TranslateTableSpec: SpecOfFiletype;
  var ReturnCode: integer
); external;
```

**Operand Description**

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TableName</td>
<td>Specifies the name of the translate table.</td>
</tr>
</tbody>
</table>

ReadXlateTable tries to read TableName TCPXLBIN. If that file exists but it has a bad format, ReadXlateTable returns with a ReturnCode FILEformatINVALID. If user_id TCPXLBIN does not exist, ReadXlateTable tries to read TCPIP TCPXLBIN. ReturnCode reflects the status of reading that file.
ReadXlateTable

AtoETable
Contains an ASCII-to-EBCDIC table if the return code is OK.

EtoATable
Contains an EBCDIC-to-ASCII table if the return code is OK.

TranslateTableSpec
If ReturnCode is OK, TranslateTableSpec contains the complete specification of the file that ReadXlateTable used. If the ReturnCode is not OK, TranslateTableSpec contains the complete specification of the last file that ReadXlateTable tried to use.

ReturnCode
Indicates the success or failure of the call. Possible return values are:
- OK
- ERRORopeningORreadingFILE
- FILEformatINVALID

RTcpExtRupt

The RTcpExtRupt procedure is a version of the TcpExtRupt Pascal procedure and can be called directly from an assembler interrupt handler.

Note: The content of this section is Internal Product Information and must not be used as programming interface information.

The following is a sample of the assembler calling sequence.

```
LA R13,PASCSAVE
LA R1,EXTPARM
L R15,=V(RTCPEXTR)
BALR R14,R15
.
.
RUPTCODE DS H Store interrupt code here before calling XTCPEXTR
PASCSAVE DS 18F Register save area
ENV DC F'0' Zero initially. It will be filled with an environment address. Pass it unchanged in subsequent calls to RTCPEXTR.
EXTPARM DC A(ENV)
DC A(RUPTCODE)
```

The RTcpExtRupt procedure has no operands.

RTcpVmcfRupt

The RTcpVmcfRupt procedure is a version of the TcpVmcfRupt Pascal procedure and can be called directly from an assembler interrupt handler.

Note: The content of this section is Internal Product Information and must not be used as programming interface information.

The following is a sample assembler calling sequence.
SayCalRe

The SayCalRe function returns a printable string describing the return code passed in CallReturn.

```
function SayCalRe
    CallReturn: integer
):
    WordType;
    external;
```

Operand  Description
CallReturn  Specifies the return code to be described.

SayConSt

The SayConSt function returns a printable string describing the connection state passed in State. For example, if SayConSt is invoked with the type identifier RECEIVINGonly, it returns the message *Receiving only*.

```
function SayConSt
    (    State: ConnectionStateType
):
    WordType;
    external;
```

Operand  Description
State  Specifies the connection state to be described.

SayIntAd

The SayIntAd function converts the internet address specified by InternetAddress to a printable string. The address is looked up in HOSTS ADDRINFO file, and the name is returned if found. If it is not found, the dotted-decimal format of the address is returned.

```
function SayIntAd
    (    InternetAddress: InternetAddressType
):
    WordType;
    external;
```

Operand  Description
InternetAddress  Specifies the internet address to be converted.
SayIntNum

The SayIntNum function converts the internet address specified by InternetAddress to a printable string, in dotted-decimal form.

```
function SayIntNum
  (  
    InternetAddress: InternetAddressType  
  ):  
  Wordtype;  
  external;
```

Operand    Description
InternetAddress    Specifies the internet address to be converted.

SayNotEn

The SayNotEn function returns a printable string describing the notification enumeration type passed in Notification. For example, if SayNotEn is invoked with the type identifier EXTERNALinterrupt, it returns the message, Other external Interrupt received.

```
function SayNotEn
  (  
    Notification: NotificationEnumType  
  );  
  Wordtype;  
  external;
```

Operand    Description
Notification    Specifies the notification enumeration type to be described.

SayPorTy

The SayPorTy function returns a printable string describing the port number passed in Port, if it is a well-known port number such as the Telnet port. Otherwise, the EBCDIC representation of the number is returned.

```
function SayPorTy
  (  
    Port: PortType  
  ):  
  WordType;  
  external;
```

Operand    Description
Port    Specifies the port number to be described.

SayProTy

The SayProTy function converts the protocol type specified by Protocol to a printable string, if it is a well-known protocol number such as 6 (TCP). Otherwise, the EBCDIC representation of the number is returned.

```
function SayProTy
  (  
    Protocol: ProtocolType  
  );  
  WordType;  
  external;
```
SetTimer

The SetTimer procedure sets a timer to expire after a specified time interval. Specify the amount of time in seconds. When it times out, you receive the TIMERexpired notification, which contains the data and the timer pointer.

Note: This procedure resets any previous time interval set on this timer.

```
procedure SetTimer
(  T: TimerPointerType;
    AmountOfTime: integer;
    Data: integer
); external;
```

### Operand Description

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Specifies a timer pointer, as returned by a previous CreateTimer call.</td>
</tr>
<tr>
<td>AmountOfTime</td>
<td>Specifies the time interval in seconds.</td>
</tr>
<tr>
<td>Data</td>
<td>Specifies an integer value to be returned with the TIMERexpired notification.</td>
</tr>
</tbody>
</table>

StartTcpNotice

The StartTcpNotice procedure establishes your own external interrupt handler. Use this procedure rather than BeginTcpIp when you want to handle simulated external interrupts yourself.

If your program does not use simulated VMCF, set the ClientDoesVmcf parameter to FALSE. For more information about the simulated Virtual Machine Communication Facility interface, see Chapter 3, “Virtual Machine Communication Facility Interface” on page 95. Later, when your program receives a simulated external interrupt that it does not handle, including a VMCF interrupt, inform the TCP interface by calling TcpExtRupt. The TCP interface then processes the interrupt.

If your program uses simulated VMCF itself, set the ClientDoesVmcf parameter to TRUE. Your program must use the VMCF AUTHORIZE function to establish a VMCF interrupt buffer. Later, when your program receives a VMCF interrupt that it does not handle, inform the TCP interface by calling TcpVmcfRupt with the address of your VMCF interrupt buffer. When your program receives a non-VMCF simulated external interrupt that it does not handle, call TcpExtRupt, as explained previously.

```
procedure StartTcpNotice
(  ClientDoesVmcf: Boolean;
  var ReturnCode: integer
); external;
```

### Operand Description

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientDoesVmcf</td>
<td>Specifies whether your program uses simulated VMCF itself.</td>
</tr>
<tr>
<td>ReturnCode</td>
<td>Specifies the return code from the TCP interface.</td>
</tr>
</tbody>
</table>
ClientDoesVmcf

Set to FALSE if your program does not use simulated VMCF. Otherwise, set to TRUE.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- ALREADYclosing
- NOtcpIPservice
- VIRTUALmemoryTOOsmall
- FATALerror

For a description of Pascal ReturnCodes, see Appendix B, "Pascal Return Codes" on page 357.

TcpAbort

The TcpAbort procedure shuts down a specific connection immediately. Data sent by your application on the aborted connection can be lost. TCP sends a reset packet to notify the foreign host that you have aborted the connection, but there is no guarantee that the reset will be received by the foreign host.

```pascal
procedure TcpAbort
(
  Connection: ConnectionType;
  var ReturnCode: integer
);
external;
```

Operand Description

Connection

Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the Connection field of the StatusInfoType record.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror
- NOsuchCONNECTION
- NOTyetBEGUN
- TCPlpSHUTDOWN

The connection is fully terminated when you receive the notification CONNECTIONstateCHANGED with the NewState field set to NONEXISTENT.

For a description of Pascal ReturnCodes, see Appendix B, "Pascal Return Codes" on page 357.

TcpClose

The TcpClose procedure begins the TCP one-way closing sequence. During this closing sequence, you, the local client, cannot send any more data. Data can be delivered to you until the foreign application also closes. TcpClose also causes all data sent on that connection by your application, and buffered by TCPIP, to be sent to the foreign application immediately.
TcpClose

procedure TcpClose
    (    
    Connection: ConnectionType;
    var  ReturnCode: integer
    );
    external;

Operand   Description
Connection Specifies the connection number, as returned by TcpOpen or
TcpWaitOpen in the Connection field of the StatusInfoType record.
ReturnCode Indicates the success or failure of the call. Possible return values
are:
    • OK
    • ABNORMALcondition
    • ALREADYclosing
    • NOsuchCONNECTION
    • NOTyetBEGUN
    • TCPipSHUTDOWN

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes”
on page 357.

Notes:
  1. If you receive the notification CONNECTIONstateCHANGED with a NewState
    of SENDINGonly, the remote application has done TcpClose (or equivalent
    function) and is receiving only. Respond with TcpClose when you have finished
    sending data on the connection.
  2. The connection is fully closed when you receive the notification
    CONNECTIONstateCHANGED, with a NewState field set to NONEXISTENT.

TcpExtRupt

Use the TcpExtRupt procedure when:
  1. You initiated the TCP/IP service by calling StartTcpNotice with
     ClientDoesVmcf set to TRUE, and your external interrupt handler receives a
     non-VMCF interrupt not handled by your program. For the handling of VMCF
     interrupts, see “TcpVmcfRupt” on page 86.
  2. You initiated the TCP/IP service by calling StartTcpNotice with
     ClientDoesVmcf set to FALSE, and your external interrupt handler receives any
     interrupt not handled by your program.

RTcpExtRupt is a version of TcpExtRupt. For more information, see “RTcpExtRupt”
on page 71 and “RTcpVmcfRupt” on page 71.

procedure TcpExtRupt
    (    
    const   RuptCode: integer
    );
    external;

Operand   Description
RuptCode   Specifies the external interrupt code you received.

TcpFReceive, TcpReceive, and TcpWaitReceive

TcpFReceive and TcpReceive are the asynchronous ways of specifying a buffer to
receive data for a given connection. Both procedures return to your program
TcpFReceive, TcpReceive, and TcpWaitReceive

Immediately. A return code of OK means that the request has been accepted. When received data has been placed in your buffer, your program receives a DATAdelivered notification. If your program uses TcpFReceive, it can receive an FRECEIVEError notification rather than DATAdelivered, indicating that the receive request was rejected, or that it was initially accepted but was later canceled because of connection closing.

TcpWaitReceive is the synchronous interface for receiving data from a TCP connection. TcpWaitReceive does not return to your program until data has been received into your buffer, or until an error occurs. Therefore, it is not necessary that TcpWaitReceive receive a notification when data is delivered. The BytesRead parameter is set to the number of bytes received by the data delivery, but if the number is less than zero, the parameter indicates an error.

TcpReceive uses a complete VMCF transaction (SEND by your virtual machine followed by REJECT by the TCPIP virtual machine) to tell the TCPIP virtual machine that your program is ready to receive, and another complete VMCF transaction (SEND by TCPIP virtual machine followed by RECEIVE by your virtual machine) to deliver the received data. In contrast, the entire TcpFReceive cycle is completed in one VMCF transaction. The TCP interface does a VMCF SEND/RECEIVE to inform TCPIP that your program is ready to receive. This transaction remains uncompleted until data is ready to be placed in your buffer. At that time the TCPIP virtual machine does a VMCF REPLY, completing the transaction.

TcpFReceive requires fewer VMCF transactions to receive data, thus increasing efficiency. The disadvantage is that each outstanding TcpFReceive means an outstanding VMCF transaction. You are limited to 50 outstanding VMCF transactions (for each virtual machine), thus 50 outstanding TcpFReceives.

With TcpReceive, you are not subject to the limit of 50 outstanding receives (for each virtual machine). The disadvantage is that there are twice as many VMCF transactions involved in receiving data, thus more overhead.

The only programming difference between TcpFReceive and TcpReceive is the generation of FRECEIVEError notifications for TcpFReceive.

```pascal
procedure TcpFReceive
(  Connection: ConnectionType;
  Buffer: Address31Type;
  BytesToRead: integer;
  var ReturnCode: integer
); external;

procedure TcpReceive
(  Connection: ConnectionType;
  Buffer: Address31Type;
  BytesToRead: integer;
  var ReturnCode: integer
); external;
```
TcpFReceive, TcpReceive, and TcpWaitReceive

procedure TcpWaitReceive
   (Connection: ConnectionType;
    Buffer: Address31Type;
    BytesToRead: integer;
    var BytesRead: integer
   );

Operand Description

Connection Specifies the connection number, as returned by TcpOpen or
TcpWaitOpen in the Connection field of the StatusInfoType record.

Buffer Specifies the address of the buffer to contain the received data.

BytesToRead Specifies the size of the buffer. TCPIP usually buffers the incoming
data until this many bytes are received. Data is delivered sooner if
the sender specified the PushFlag, or if the sender does a TcpClose
or equivalent. The largest usable buffer is 8192 bytes. Specifying
BytesToRead of more than 8192 bytes may not cause an error
return, but only 8192 bytes of the buffer are used.

Note: The order of TcpFReceive or TcpReceive calls on multiple
connections, and the order of DATAdelivered notifications
among the connections, are not necessarily related.

BytesRead Indicates a value when TcpWaitReceive returns. If it is greater than
ZERO, it indicates the number of bytes received into your buffer. If
it is less than or equal to ZERO, it indicates an error.

Possible BytesRead values are:
• OK*
• ABNORMALcondition
• FATALerror
• TIMEOUTopen*
• UNREACHABLEnetwork*
• BADlengthARGUMENT
• NOsuchCONNECTION
• NOTyetBEGUN
• NOTyetOPEN
• OPENrejected*
• RECEIVEstillPENDING
• REMOTEreset*
• UNEXPECTEDsyn*
• WRONGsecORprc*
• DROPPEDbyOPERATOR*
• FATALerror*
• KILLEDbyCLIENT*
• TCPIPSHUTDOWN*
• TIMEOUTconnection*
• REMOTEclose

ReturnCode: Indicates the success or failure of the call. Possible return values are:
• OK
• ABNORMALcondition
• BEGUNlengthARGUMENT
• fatalerror
• NOsuchCONNECTION
TcpFReceive, TcpReceive, and TcpWaitReceive

- NOTyetBEGUN
- NOTyetOPEN
- RECEIVestillPENDING
- REMOTEclose
- TCPipSHUTDOWN

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

Notes:

1. For BytesRead OK, the function was initiated, but the connection is no longer receiving for an unspecified reason. Your program does not have to issue TcpClose, but the connection is not completely terminated until a NONEXISTENT notification is received for the connection.

2. For BytesRead REMOTEclose, the foreign host has closed the connection. Your program should respond with TcpClose.

3. If you receive any of the codes marked with *, the function was initiated but the connection has now been terminated (see [on page 49]). Your program should not issue TcpClose, but the connection is not completely terminated until NONEXISTENT notification is received for the connection.

4. TcpWaitReceive is intended to be used by programs that manage a single TCP connection. It is not suitable for use by multiconnection servers.

5. A return code of TCPipSHUTDOWN can be returned either because the connection initiation has failed, or because the connection has been terminated, because of shutdown. In either case, your program should not issue any more TCP/IP calls.

TcpFSend, TcpSend, and TcpWaitSend

TcpFSend and TcpSend are the asynchronous ways of sending data on a TCP connection. Both procedures return to your program immediately (do not wait under any circumstance).

TcpWaitSend is a simple synchronous method of sending data on a TCP connection. It does not return immediately if the TCPIP virtual machine has insufficient space to accept the data being sent.

TcpFSend and TcpSend differ in the way that they handle VMCF when the TCPIP virtual machine has insufficient buffer space to accept the data being sent. Both start by issuing a VMCF SEND function to transfer your data. Normally, the TCPIP virtual machine issues a VMCF RECEIVE, thus completing the VMCF transaction.

In the case of insufficient buffer space, TCPIP responds to TcpSend with a VMCF REJECT, completing the VMCF transaction (unsuccessfully). When space becomes available, another complete VMCF transaction is performed to send a BUFFERspaceAVAILABLE notification.

In the case of a TcpFSend with insufficient buffer space, TCPIP does not respond to the VMCF SEND until buffer space becomes available, at which time the transaction is completed with a VMCF RECEIVE.

TcpSend returns to your program after the VMCF response from TCPIP is received. In contrast, because the VMCF response from TcpFSend may be delayed, TcpFSend returns before the VMCF response is received. An OK return code from TcpFSend indicates only the successful initiation of the VMCF transaction.
TcpFSend, TcpSend, and TcpWaitSend

The advantage of TcpFSend is that the VMCF transactions necessary to send data are reduced in the case where a program can send data faster than the TCP connection can carry it. Its disadvantages are that it is limited to 50 outstanding VMCF sends and therefore 50 TcpFSends, and is slightly more complicated to use, because you have to wait for an FSENDResponse notification (generated internally by the TCP interface) between successive TcpFSends.

The advantage of TcpSend is that it does not involve an outstanding VMCF transaction. Thus, there is no imposed VMCF-related limit. Also, TcpSend is simpler to use because you can issue successive TcpSends without waiting for a notification. The disadvantage of TcpSend is that it is less efficient than TcpFSend if your program can send data faster than the TCP connection can carry it.

Your program can issue successive TcpWaitSend calls. Buffer shortage conditions are handled transparently. Any errors that occur are likely to be nonrecoverable errors, or are caused by a connection that has terminated.

If you receive any of the codes listed for Reason in the CONNECTIONStateCHANGED notification, except for OK, the connection was terminated for the indicated reason. Your program should not issue a TcpClose, but the connection is not completely terminated until your program receives a NONEXISTENT notification for the connection.

```plaintext
procedure TcpFSend
    (    Connection: ConnectionType;
    Buffer: Address31Type;
    BufferLength: integer;
    PushFlag: Boolean;
    UrgentFlag: Boolean;
    var ReturnCode: integer
    );
  external;

procedure TcpSend
    (    Connection: ConnectionType;
    Buffer: Address31Type;
    BufferLength: integer;
    PushFlag: Boolean;
    UrgentFlag: Boolean;
    var ReturnCode: integer
    );
  external;

procedure TcpWaitSend
    (    Connection: ConnectionType;
    Buffer: Address31Type;
    BufferLength: integer;
    PushFlag: Boolean;
    UrgentFlag: Boolean;
    var ReturnCode: integer
    );
  external;
```

**Operand Description**

Connection  Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the Connection field of the StatusInfoType record.

Buffer  Specifies the address of the buffer containing the data to send.
BufferLength  Specifies the length of data in the buffer. Maximum is 8192.

PushFlag     Forces the data, and previously queued data, to be sent immediately to the foreign application.

UrgentFlag   Marks the data as urgent. The semantics of urgent data is dependent on your application.

Note: Use urgent data with caution. If the foreign application follows the Telnet-style use of urgent data, it may flush all urgent data until a special character sequence is encountered.

ReturnCode   Indicates the success or failure of the call. Possible return values are:
• OK
• ABNORMALcondition
• BADlengthARGUMENT
• CANNOTsendDATA
• FATALerror
• FSENDstillpending
• NObufferSPACE (TcpSend only)
• NOsuchCONNECTION
• NOTyetBEGUN
• NOTyetOPEN
• TCPipSHUTDOWN

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 354.

Notes:
1. A successful TcpFSend, TcpSend, and TcpWaitSend means that TCP has received the data to be sent and stored it in its internal buffers. TCP then puts the data in packets and transmits it when the conditions permit.
2. Data sent in a TcpFSend, TcpSend, or TcpWaitSend request may be split up into numerous packets by TCP, or the data may wait in TCP’s buffer space and share a packet with other TcpFSend, TcpSend, or TcpWaitSend requests.
3. The PushFlag gives the user the ability to affect when TCP sends the data. Setting the PushFlag to FALSE allows TCP to buffer the data and wait until it has enough data to transmit so as to utilize the transmission line more efficiently. There can be some delay before the foreign host receives the data. Setting the PushFlag to TRUE instructs TCP to packetize and transmit any buffered data from previous Send requests along with the data in the current TcpFSend, TcpSend, or TcpWaitSend request without delay or consideration of transmission line efficiency. A successful send does not imply that the foreign application has actually received the data, only that the data will be sent as soon as possible.
4. TcpWaitSend is intended for programs that manage a single TCP connection. It is not suitable for use by multiconnection servers.

TcpNameChange

The TcpNameChange procedure is used if the virtual machine running the TCP/IP program is not using the default name, TCPIP, and is not the same as specified in the TCPIPUSERID statement of the TCPIP DATA file. For more information, see TCP/IP Planning and Customization.
TcpNameChange

If required, this procedure must be called before the BeginTcpIp or the StartTcpNotice procedure.

```pascal
procedure TcpNameChange
    (   NewNameOfTcp: DirectoryNameType
    );
    external;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewNameOfTcp</td>
<td>Specifies the name of the virtual machine running TCP/IP.</td>
</tr>
</tbody>
</table>

TcpOpen and TcpWaitOpen

The TcpOpen or TcpWaitOpen procedures initiate a TCP connection. TcpOpen returns immediately, and connection establishment proceeds asynchronously with your program’s other operations. The connection is fully established when your program receives a CONNECTIONstateCHANGED notification with NewState set to OPEN. TcpWaitOpen does not return until the connection is established, or until an error occurs.

There are two types of TcpOpen calls: passive open and active open. A passive open call sets the connection state to LISTENING. An active open call sets the connection state to TRYINGtoOPEN.

If a TcpOpen or TcpWaitOpen call returns ZEROresources, and your application handles RESOURCESavailable notifications, you receive a RESOURCESavailable notification when sufficient resources are available to process an open call. The first open your program issues after a RESOURCESavailable notification is guaranteed not to get the ZEROresources return code.

```pascal
procedure TcpOpen
    (   var ConnectionInfo: StatusInfoType;
    var ReturnCode: integer
    );
    external;

procedure TcpWaitOpen
    (   var ConnectionInfo: StatusInfoType;
    var ReturnCode: integer
    );
    external;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionInfo</td>
<td>Specifies a connection information record.</td>
</tr>
</tbody>
</table>

Connection

Set this field to UNSPECIFIEDconnection. When the call returns, the field contains the number of the new connection if ReturnCode is OK.

ConnectionState

For active open, set this field to TRYINGtoOPEN. For passive open, set this field to LISTENING.
OpenAttemptTimeout

Set this field to specify how long, in seconds, TCP is to continue attempting to open the connection. If the connection is not fully established during that time, TCP reports the error to you. If you used TcpOpen, you receive a notification. The type of notification that you receive is CONNECTIONstateCHANGED. It has a new state of NONEXISTENT and a reason of TIMEOUTopen. If you used TcpWaitOpen, it returns with ReturnCode set to TIMEOUTopen.

Security

This field is reserved. Set it to DEFAULTsecurity.

Compartment

This field is reserved. Set it to DEFAULTcompartment.

Precedence

This field is reserved. Set it to DEFAULTprecedence.

LocalSocket

Active Open: You can use an address of UNSPECIFIEDAddress (the TCPIP virtual machine uses the home address corresponding to the network interface used to route to the foreign address) and a port of UNSPECIFIEDPort (the TCPIP virtual machine assigns a port number, in the range of 1000 to 65 534). You can specify the address, the port, or both if particular values are required by your application. The address must be a valid home address for your node, and the port must be available (not reserved, and not in use by another application).

Passive Open: You usually specify a predetermined port number, which is known by another program, which can do an active open to connect to your program. Alternatively, you can use UNSPECIFIEDPort to let the TCPIP virtual machine assign a port number, obtain the port number through TcpStatus, and transmit it to the other program through an existing TCP connection or manually. You generally specify an address of UNSPECIFIEDAddress, so that the active open to your port succeeds, regardless of the home addresses to which it was sent.

ForeignSocket

Active Open: The address and port must both be specified, because the TCPIP virtual machine cannot actively initiate a connection without knowing the destination address and port.

Passive Open: If your program is offering a service to anyone who wants it, specify an address of UNSPECIFIEDAddress and a port of UNSPECIFIEDPort. You can specify a particular address and port if you want to accept an active open only from a certain foreign application.

ReturnCode

Indicates the success or failure of the call. Possible return values are:
TcpOpen and TcpWaitOpen

- OK
- ABNORMALcondition
- FATALerror
- CONNECTIONalreadyEXISTS
- DROPPEDbyOPERATOR (TcpWaitOpen only)
- LOCALportNOTavailable
- NOsuchCONNECTION
- NOTyetBEGUN
- OPENrejected (TcpWaitOpen only)
- PARAMlocalADDRESS
- PARAMstate
- PARAMtimeout
- PARAMunspecADDRESS
- PARAMunspecPORT
- REMOTEreset (TcpWaitOpen only)
- SOFTWAREerror
- TCPipSHUTDOWN
- TIMEOUTconnection (TcpWaitOpen only)
- TIMEOUTopen (TcpWaitOpen only)
- TOOmanyOPENS
- UNEXPECTEDsyn (TcpWaitOpen only)
- UNREACHABLEnetwork (TcpWaitOpen only)
- WRONGsecORrpc (TcpWaitOpen only)
- ZEROresources

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

TcpOption

The TcpOption procedure sets an option for a TCP connection.

procedure TcpOption

  connection: ConnectionType
  OptionName: integer;
  OptionValue: integer;
var
  ReturnCode: integer
); external;

Operand Description

Connection Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the Connection field of the StatusInfoType record.

OptionName Specifies the code for the option.

OPTIONTcpKEEPALIVE

If OptionValue is zero, the keep-alive mechanism is deactivated for the connection. If OptionValue is nonzero, the keep-alive mechanism is activated for the connection. This mechanism sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet, the connection state will be changed to NONEXISTENT with TIMEOUTconnection as the reason.

OptionValue Specifies the value for the option.
ReturnCode  Indicates the success or failure of the call. Possible return values are:
  • OK
  • NOSuchCONNECTION
  • NOTyetBEGUN
  • TCPipSHUTDOWN
  • INVALIDrequest

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

TcpStatus

The TcpStatus procedure obtains the current status of a TCP connection. Your program sets the Connection field of the ConnectionInfo record to the number of the connection whose status you want.

```
procedure TcpStatus
(
  var ConnectionInfo: StatusInfoType;
  var ReturnCode: integer
);
```

Operand  Description
---  ---
ConnectionInfo  If ReturnCode is OK, the following fields are returned.

Field  Description
---  ---
OpenAttemptTimeout  If the connection is in the process of being opened (including a passive open), this field is set to the number of seconds remaining before the open is terminated if it has not completed. Otherwise, it is set to WAITforever.

BytesToRead  Specifies the number of bytes of incoming data queued for your program (waiting for TcpReceive, TcpFReceive, or TcpWaitReceive).

UnackedBytes  Specifies the number of bytes sent by your program but not yet sent to the foreign TCP, or the number of bytes sent to the foreign TCP, but not yet acknowledged.

ConnectionState  Specifies the current connection state.

LocalSocket  Specifies the local socket, consisting of a local address and a local port.

ForeignSocket  Specifies the foreign socket, consisting of a foreign address and a foreign port.

ReturnCode  Indicates the success or failure of the call. Possible return values are:
  • OK
  • ABNORMALcondition
  • NOSuchCONNECTION
  • NOTyetBEGUN
TcpStatus

- TCPipSHUTDOWN

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

**Note:** Your program cannot monitor connection state changes exclusively through polling with TcpStatus. It must receive CONNECTIONstateCHANGED notifications through GetNextNote, for the TCP interface to work properly.

**TcpVmcfRupt**

The TcpVmcfRupt procedure is used when you initiate the TCP/IP service by calling StartTcpNotice with ClientDoesVmcf set to TRUE, and your external interrupt handler receives a VMCF interrupt not handled by your program.

RTcpVmcfRupt is a version of TcpVmcfRupt that can be called directly from an assembler interrupt handler. For more information, see “RTcpExtRupt” on page 71 and “RTcpVmcfRupt” on page 71.

```
procedure TcpVmcfRupt
    (VmcfHeaderAddress: integer);
    external;
```

**Operand** | **Description**
--- | ---
VmcfHeaderAddress | Indicates the address of your VMCF interrupt buffer as specified in the VMCF AUTHORIZE function that your program issued at initialization.

**UdpClose**

The UdpClose procedure closes the UDP socket specified in the ConnIndex field. All incoming datagrams on this connection are discarded.

```
procedure UdpClose
    (ConnIndex: ConnectionIndexType;
    var ReturnCode: CallReturnCodeType);
    external;
```

**Operand** | **Description**
--- | ---
ConnIndex | Specifies the ConnIndex value returned from UdpOpen.
ReturnCode | Indicates the success or failure of the call. Possible return values are:
- OK
- ABNORMALcondition
- FATAError
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.
UdpNReceive

The UdpNReceive procedure notifies the TCPIP virtual machine that you can receive UDP datagram data. This call returns immediately. The data buffer is not valid until you receive a UDPdatagramDELIVERED notification.

```pascal
procedure UdpNReceive(
    ConnIndex: ConnectionIndexType;
    BufferAddress: integer;
    BufferLength: integer;
    var ReturnCode: CallReturnCodeType
); external;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnIndex</td>
<td>Specifies the ConnIndex value returned from UdpOpen.</td>
</tr>
<tr>
<td>BufferAddress</td>
<td>Specifies the address of your buffer that will be filled with a UDP datagram.</td>
</tr>
<tr>
<td>BufferLength</td>
<td>Specifies the length of your buffer. If you specify a length larger than 8192 bytes, only the first 8192 bytes are used.</td>
</tr>
<tr>
<td>ReturnCode</td>
<td>Indicates the success or failure of the call. Possible return values are:</td>
</tr>
<tr>
<td></td>
<td>• OK</td>
</tr>
<tr>
<td></td>
<td>• ABNORMALcondition</td>
</tr>
<tr>
<td></td>
<td>• FATALerror</td>
</tr>
<tr>
<td></td>
<td>• NOsuchCONNECTION</td>
</tr>
<tr>
<td></td>
<td>• NOTyetBEGUN</td>
</tr>
<tr>
<td></td>
<td>• RECEIVEstillPENDING</td>
</tr>
<tr>
<td></td>
<td>• TCPipSHUTDOWN</td>
</tr>
</tbody>
</table>

For a description of Pascal ReturnCodes, see Appendix B, "Pascal Return Codes" on page 357.

UdpOpen

The UdpOpen procedure requests acceptance of UDP datagrams on the specified socket and allows datagrams to be sent from the specified socket. When the socket port is unspecified, UDP selects a port and returns it to the socket port field. When the socket address is unspecified, UDP uses the default local address. If specified, the address must be a valid home address for your node.

Note: When the local address is specified, only the UDP datagrams addressed to it are delivered.

If the ReturnCode indicates the open was successful, use the returned ConnIndex value on any further actions pertaining to this UDP socket.

```pascal
procedure UdpOpen(
    var LocalSocket: SocketType;
    var ConnIndex: ConnectionIndexType;
    var ReturnCode: CallReturnCodeType
); external;
```
UdpOpen

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocalSocket</td>
<td>Specifies the local socket (address and port pair).</td>
</tr>
<tr>
<td>ConnIndex</td>
<td>Specifies the ConnIndex value returned from UdpOpen.</td>
</tr>
<tr>
<td>ReturnCode</td>
<td>Indicates the success or failure of the call. Possible return values are:</td>
</tr>
<tr>
<td></td>
<td>• OK</td>
</tr>
<tr>
<td></td>
<td>• ABNORMALcondition</td>
</tr>
<tr>
<td></td>
<td>• FATALerror</td>
</tr>
<tr>
<td></td>
<td>• LOCALportNOTavailable</td>
</tr>
<tr>
<td></td>
<td>• NOTyetBEGUN</td>
</tr>
<tr>
<td></td>
<td>• SOFTWAREerror</td>
</tr>
<tr>
<td></td>
<td>• TCPipSHUTDOWN</td>
</tr>
<tr>
<td></td>
<td>• UDPlocalADDRESS</td>
</tr>
<tr>
<td></td>
<td>• UDPzeroRESOURCES</td>
</tr>
</tbody>
</table>

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

Note: If a UdpOpen call returns UDPzeroRESOURCES, and your application handles UDPresourcesAVAILABLE notifications, you receive a UDPresourcesAVAILABLE notification when sufficient resources are available to process a UdpOpen call. The first UdpOpen your program issues after a UDPresourcesAVAILABLE notification is guaranteed not to get the UDPzeroRESOURCES return code.

UdpReceive

The UdpReceive procedure notifies the TCPIP virtual machine that you are willing to receive UDP datagram data.

UdpReceive is for compatibility with old programs only. New programs should use the UdpNReceive procedure, which allows you to specify the size of your buffer.

If you use UdpReceive, TCPIP can put a datagram of up to 2012 bytes in your buffer. If a larger datagram is sent to your port when UdpReceive is pending, the datagram is discarded without notification.

Note: No data is transferred from the TCPIP virtual machine in this call. It only tells TCPIP that you are waiting for a datagram. Data has been transferred when a UDPdatagramDELIVERED notification is received.

procedure UdpReceive

(  ConnIndex: ConnectionIndexType;
  DatagramAddress: integer;
  var ReturnCode: CallReturnCodeType
); external;

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnIndex</td>
<td>Specifies the ConnIndex value returned from UdpOpen.</td>
</tr>
<tr>
<td>DatagramAddress</td>
<td>Specifies the address of your buffer that will be filled with a UDP datagram.</td>
</tr>
</tbody>
</table>
UdpReceive

ReturnCode  Indicates the success or failure of the call. Possible return values are:
  • OK
  • ABNORMALcondition
  • FATALerror
  • NOsuchCONNECTION
  • NOTyetBEGUN
  • SOFTWAREerror
  • TCPipSHUTDOWN

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

UdpSend

The UdpSend procedure sends a UDP datagram to the specified foreign socket. The source socket is the local socket selected in the UdpOpen that returned the ConnIndex value that was used. The buffer does not include the UDP header. This header is supplied by the TCPIP virtual machine.

When there is no buffer space to process the data, an error is returned. In this case, wait for a subsequent UDPdatagramSPACEavailable notification.

procedure UdpSend
  (  
    ConnIndex: ConnectionIndexType;
    ForeignSocket: SocketType;
    BufferAddress: integer;
    Length: integer;
  
    var ReturnCode: CallReturnCodeType
  );

Operand  Description
ConnIndex  Specifies the ConnIndex value returned from UdpOpen.
ForeignSocket  Specifies the foreign socket (address and port) to whom the datagram is to be sent.
BufferAddress  Specifies the address of your buffer containing the UDP datagram to be sent, excluding UDP header.
Length  Specifies the length of the datagram to be sent, excluding UDP header. Maximum is 8192 bytes.
ReturnCode  Indicates the success or failure of the call. Possible return values are:
  • OK
  • BADlengthARGUMENT
  • NObufferSPACE
  • NOsuchCONNECTION
  • NOTyetBEGUN
  • SOFTWAREerror
  • TCPipSHUTDOWN
  • UDPunspecADDRESS
  • UDPunspecPORT
Unhandle

The Unhandle procedure specifies that you no longer want to receive notifications in the given set.

If you request to unhandle the DATAdelivered notification, the Unhandle procedure returns with a code of INVALIDrequest.

```pascal
procedure Unhandle
  (Notifications: NotificationSetType;
   var ReturnCode: integer
  );
external;
```

Operand Description

Notifications Specifies the set of notifications that you no longer want to receive.

ReturnCode Indicates the success or failure of the call. Possible return values are:
  • OK
  • ABNORMALcondition
  • FATALerror
  • INVALIDrequest
  • NOTyetBEGUN
  • TCPipSHUTDOWN

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.

UnNotifyIo

The UnNotifyIo routine is used to indicate that you no longer wish to be sent a notification when an I/O interrupt occurs on the specified virtual address.

```pascal
procedure UnNotifyIo
  (DeviceAddress: integer;
   var ReturnCode: integer
  );
external;
```

Operand Description

DeviceAddress Specifies the address of the device for which IOinterrupt notifications are no longer to be generated.

ReturnCode Indicates the success or failure of the call. Possible return values are:
  • OK
  • NOsuchCONNECTION
  • SOFTWAREerror

For a description of Pascal ReturnCodes, see Appendix B, “Pascal Return Codes” on page 357.
Sample Pascal Program

```pascal
{**********************************************************************}
{ * Memory-to-memory Data Transfer Rate Measurement *
{ * Pseudocode: Establish access to TCP/IP Services *
{ * Prompt user for operation operands *
{ * Open a connection (Sender:passive, Receiver:active) *
{ * If Sender: *
{ * Send 5M of data using TcpFSend *
{ * Use GetNextNote to know when Send is complete *
{ * Print transfer rate after every 1M of data *
{ * else Receiver: *
{ * Receive 5M of data using TcpFReceive *
{ * Use GetNextNote to know when data is delivered *
{ * Print transfer rate after every 1M of data *
{ * Close connection *
{ * Use GetNextNote to wait until connection is closed *
{ *
{**********************************************************************}

program SAMPLE;

%include CMALLCL
%include CMINTER
%include CMRESGLB

const
  BUFFERlength = 8192;
  PORTnumber = 999;
  CLOCKunitsPERthousandth = '3E8000'x;

var
  Buffer : packed array (.1..BUFFERlength.) of char;
  BufferAddress : Address31Type;
  ConnectionInfo : StatusInfoType;
  Count : integer;
  DataRate : real;
  Difference : TimeStampType;
  Error : boolean;
  HostAddress : InternetAddressType;
  IbmSeconds : integer;
  Ignored : integer;
  Line : string(80);
  Note : NotificationInfoType;
  NumBytes : integer;
  RealRate : real;
  ReturnCode : integer;
  SendFlag : boolean;
  StartingTime : TimeStampType;
  Thousandths : integer;
  TotalBytes : integer;

{**********************************************************}
{* Print message, release resources and reset environment *}
{**********************************************************}
procedure Restore ( const Message: string;
  const ReturnCode: integer );
begin
  Write (Message);
  if ReturnCode <> OK then
    Write (SayCalRe(ReturnCode));
  WriteLn ('');
EndTcpIp;
```

Chapter 2. TCP/UDP/IP API (Pascal Language) 91
Sample Pascal Program

```pascal
begin
  TermOut (Output);
  TermIn (Input);

  { Enable program to run with ECMODE OFF. It has no effect if }
  { ECMODE ON. There is, however, additional overhead and }
  { possible performance impact when running with ECMODE OFF. }
  InitEmulation (Error);
  if Error then begin
    WriteLn ('InitEmulation failed');
    return;
  end;

  { Establish access to TCP/IP services }
  BeginTcpIp (ReturnCode);
  if ReturnCode <> OK then begin
    WriteLn ('BeginTcpip: ', SayCalRe(ReturnCode));
    return;
  end;

  { Inform TCPIP which notifications will be handled by the program }
  Handle ((.DATAdelivered, BUFFERspaceAVAILABLE, 
            CONNECTIONstateCHANGED.), ReturnCode);
  if ReturnCode <> OK then begin
    Restore ('Handle: ', ReturnCode);
    return;
  end;

  { Prompt user for operation operands }
  WriteLn ('Transfer mode: (Send or Receive)');
  ReadLn (Line);
  if (Substr(Line,1,1) = 's') or (Substr(Line,1,1) = 'S') then
    SendFlag := TRUE
  else
    SendFlag := FALSE;

  WriteLn ('Host Name or Internet Address :');
  ReadLn (Line);
  GetHostResol (Line, HostAddress);
  if HostAddress = NOhost then begin
    Restore ('GetHostResol failed', OK);
    return;
  end;

  { Open a TCP connection: active for Send and passive for Receive }
  with ConnectionInfo do begin
    Connection := UNSPECIFIEDconnection;
    OpenAttemptTimeout := WAITforever;
    Security := DEFAULTsecurity;
    Compartment := DEFAULTcompartment;
    Precedence := DEFAULTprecedence;
    if SendFlag then begin
      ConnectionState := TRYINGtoOPEN;
      LocalSocket.Address := UNSPECIFIEDaddress;
      LocalSocket.Port := UNSPECIFIEDport;
      ForeignSocket.Address := HostAddress;
      ForeignSocket.Port := PORTnumber;
    end
    else begin
      ConnectionState := LISTENING;
      LocalSocket.Address := HostAddress;
    end
  end
end;
```
Sample Pascal Program

LocalSocket.Port := PORTnumber;
ForeignSocket.Address := UNSPECIFIEDaddress;
ForeignSocket.Port := UNSPECIFIEDport;
end;
end;
TcpWaitOpen (ConnectionInfo, ReturnCode);
if ReturnCode <> OK then begin
  Restore ('TcpWaitOpen: ', ReturnCode);
  return;
end;

{ Initialization }
BufferAddress := AddressOfChar(Buffer(.1.));
NumBytes := BUFFERlength;
StartingTime := ClockTime;
TotalBytes := 0;
Count := 0;

{ Repeat until 5M bytes of data have been transferred }
while (Count < 5) do begin
  { Transfer data and wait until operation is completed }
  if SendFlag then
    TcpWaitSend (ConnectionInfo.Connection, BufferAddress,
      BUFFERlength, FALSE, FALSE, ReturnCode)
  else begin
    TcpWaitReceive (ConnectionInfo.Connection, BufferAddress,
      BUFFERlength, NumBytes);
    if NumBytes < 0 then
      ReturnCode := NumBytes;
  end;
  if ReturnCode <> OK then begin
    WriteLn ('TcpSend/Receive: ', SayCalRe(ReturnCode));
    leave;
  end;
  TotalBytes := TotalBytes + NumBytes;
  if TotalBytes < 1048576 then
    continue;

  { Print transfer rate after every 1M bytes of data transferred }
  DoubleSubtract (ClockTime, StartingTime, Difference);
  DoubleDivide (Difference, CLOCKunitsPERthousandth, Thousandths, Ignored);
  RealRate := (TotalBytes/Thousandths) * 1000.0;
  WriteLn ('Transfer rate ', RealRate:1:0, ' Bytes/sec. '
  StartingTime := ClockTime;
  TotalBytes := 0;
  Count := Count + 1;
end;

{ Close TCP connection and wait till partner also drops connection }
TcpClose (ConnectionInfo.Connection, ReturnCode);
if ReturnCode <> OK then begin
  Restore ('TcpClose: ', ReturnCode);
  return;
end;
repeat
  GetNextNote (Note, True, ReturnCode);
  if ReturnCode <> OK then begin
    Restore ('GetNextNote: ', ReturnCode);
    return;
  end;
until (Note.NotificationTag = CONNECTIONstateCHANGED) &
Sample Pascal Program

(Note.NewState = NONEXISTENT);

    Restore ('Program terminated successfully', OK);
    end.
The Virtual Machine Communication Facility (VMCF) is part of the Control Program (CP) of VM. VMCF enables virtual machines to send data to and receive data from any other virtual machine.

You can communicate directly with the TCPIP virtual machine using VMCF calls, rather than Pascal API or C socket calls. You can use VMCF calls when:
- You want to write your program in assembler.
- You add TCP/IP communication to an existing complex program, and it can be difficult or impossible for your program to monitor TCP/IP events through the Pascal GetNextNote interface.

If your program drives the VMCF interface directly, do not link any of the TCP interface library modules with your program. Consequently, you cannot use any of the auxiliary routines, such as the Say functions and timer routines. (You must use VM timer support, or support provided by your existing program). VMCF consists of data transfer functions, control functions, a special external interrupt for pending messages, and an external interrupt message header to pass control information and data to another virtual machine.

For more information about the VMCF interface, see the VM/ESA: CP Programming Services book.

General Information

The following section describes the data structure of the VMCF interrupt header used by TCP/IP for VM. The section also lists the VMCF functions available with TCP/IP for VM. Tables summarizing the CALLCODE for making VMCF requests and receiving notifications from TCPIP virtual machine are provided. The remainder of the chapter describes these CALLCODE calls in details.

Data Structures

VMCF is implemented with functions invoked using DIAGNOSE X’68’ and a special 40-byte parameter list. A VMCF function is requested by a particular function subcode in the FUNC field in the parameter list.

Your program uses the standard 40-byte VMCF parameter list to submit VMCF requests to the TCPIP virtual machine. The TCPIP virtual machine returns VMCF interrupts results in the similar 40-byte VMCF parameter list. The parameter list is the interrupt header being stored in your virtual machine. In this chapter, fields in the parameter list and interrupt header are referred to using the data structure header names in Figure 21 on page 94.
VMCF Interface

V1  DS  X
V2  DS  X
FUNC DS  H
MSGID DS  F
JOBNAME DS  CL8
VADA DS  A
LENA DS  F
VADB DS  A
LENB DS  F

* User-doubleword field is divided into the following fields:
ANTEGR DS  F
CONN DS  H
CALLCODE DS  X
RETCODE DS  X

Figure 21. Assembler Format of the VMCF Parameter List Fields

VMCF Parameter List Fields
The following describes the VMCF parameter list fields.

V1  Used for security and data integrity. You can enable your virtual machine for VMCF communication to the TCPIP virtual machine by executing the AUTHORIZE control function. The AUTHORIZE control function is set by issuing a DIAGNOSE Code X’68’ Subcode X’0000’ assembler call. If you do not set the AUTHORIZE function in V1, check the JOBNAME field when processing each interrupt to ensure that interrupts from other virtual machines are not misinterpreted as coming from TCPIP. V1 must be zero for all VMCF functions other than AUTHORIZE. To terminate VMCF activities for a virtual machine, issue the UNAUTHORIZE control function. The UNAUTHORIZE control function is set by issuing a DIAGNOSE Code X’68’ Subcode X’0001’ assembler call.

FUNC The IUCV operation.

V2  Reserved for IBM use, and should be X’00’ initially.

MSGID  Contains a unique message identifier associated with a transaction. You must use a unique, even number for each outstanding transaction. A simple method is to use consecutive, even numbers for each transaction.

JOBNAME  Specifies the user ID of the virtual machine making VMCF requests. You must set this field to the user ID of the TCPIP virtual machine.

VADA  Contains the address of the source or destination address depending on the VMCF function requested.

LENA  Contains the length of the data sent by a user, the length of a RECEIVE buffer, or the length of an external interrupt buffer, whichever is specified in the VADA field.

VADB  Contains the address of a source virtual machine’s REPLY buffer for VMCF request.

LENB  Specifies the length of the source virtual machine’s REPLY buffer.

The use of each field is described individually for each TCP/IP function.

VMCF Interrupt Header Fields
The following describes the VMCF parameter list fields for the interrupt header.
**VMCF Interface**

**V1**  
Sets the VMCMRESP flag, which is the interrupt in response to a transaction initiated by your virtual machine. If the TCPIP virtual machine responds using the REJECT function, the VMCMRJCT flag is also set. This flag by itself does not usually indicate that the transaction was unsuccessful. Your program should check the completion status code in the RETCODE field, as described for each function.

**ANINTEGR**  
Checks the status of VMCF transactions. It is a field, of fullword length (four bytes), used to check the status of VMCF transactions. The field is described for each function.

**CONN**  
Establishes a TCP connection. If a connection between your virtual machine and TCPIP virtual machine was established successfully and the RETCODE field indicates OK, the connection number of the new connection is stored in this field.

**CALLCODE**  
Calls instructions to be passed by your program when initiating a VMCF function to interface with TCPIP virtual machine. If the interrupt is in response to a transaction initiated by your virtual machine (VMCMRESP flag set in V1), the CALLCODE value is the same as the value set by your program when it initiated the transaction.

**RETCODE**  
Contains the completion status codes of a transaction. Return codes reported in this field are taken from the same set used by Pascal programs (see Appendix B, “Pascal Return Codes” on page 357). Further information is given in the description of each function.

**VMCF Functions**

Table 17 lists the available VMCF functions, with descriptions, to communicate with the TCPIP virtual machine.

**Table 17. Available VMCF Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTHORIZE</td>
<td>Control</td>
<td>Initializes VMCF for a given virtual machine. Once AUTHORIZE is executed, the virtual machine can execute other VMCF functions and receive messages or requests from other users.</td>
</tr>
<tr>
<td>UNAUTHORIZE</td>
<td>Control</td>
<td>Terminates VMCF activity.</td>
</tr>
<tr>
<td>SEND</td>
<td>Data</td>
<td>Directs a message or block of data to another virtual machine.</td>
</tr>
<tr>
<td>SEND/RECV</td>
<td>Data</td>
<td>Directs a message or block of data to another virtual machine, and requests a reply.</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>Data</td>
<td>Allows you to accept selective messages or data sent using the SEND or SEND/RECV functions.</td>
</tr>
<tr>
<td>REPLY</td>
<td>Data</td>
<td>Allows you to direct data back to the originator of a SEND/RECV function, simulating duplex communication.</td>
</tr>
<tr>
<td>REJECT</td>
<td>Data</td>
<td>Allows you to reject specific SEND or SEND/RECV requests pending for your virtual machine.</td>
</tr>
</tbody>
</table>
VMCF Interface

Table 17. Available VMCF Functions (continued)

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note: Data</td>
<td>Indicates a data transfer</td>
</tr>
<tr>
<td>Note: Control</td>
<td>Indicates a VMCF control function</td>
</tr>
</tbody>
</table>

VMCF TCP/IP Communication CALLCODE Requests

Table 18 lists the equate values and available calls for initiating a VMCF TCP/IP request; it also includes a description of each CALLCODE request.

Table 18. VMCF TCP/IP CALLCODE Requests

<table>
<thead>
<tr>
<th>Call Code</th>
<th>Equates</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECTIONclosing</td>
<td>00</td>
<td>Data may no longer be transmitted on this connection since the TCP/IP service is in the process of closing down the process.</td>
</tr>
<tr>
<td>LISTENING</td>
<td>01</td>
<td>Waiting for a foreign site to open a connection.</td>
</tr>
<tr>
<td>NONEXISTENT</td>
<td>02</td>
<td>The connection no longer exists.</td>
</tr>
<tr>
<td>OPEN</td>
<td>03</td>
<td>Data can go either way on the connection.</td>
</tr>
<tr>
<td>RECEIVINGonly</td>
<td>04</td>
<td>Data can be received but not sent on this connection, because the client has done a one-way close.</td>
</tr>
<tr>
<td>SENDINGonly</td>
<td>05</td>
<td>Data can be sent out but not received on this connection. This means that the foreign site has done a one-way close.</td>
</tr>
<tr>
<td>TRYINGtoOPEN</td>
<td>06</td>
<td>Trying to contact a foreign site to establish a connection.</td>
</tr>
<tr>
<td>ABORTtcp</td>
<td>100</td>
<td>Terminates a TCP connection.</td>
</tr>
<tr>
<td>BEGINtcpIPservice</td>
<td>101</td>
<td>Initializes a TCP/IP connection between your program and the TCPIP virtual machine.</td>
</tr>
<tr>
<td>CLOSEtcp</td>
<td>102</td>
<td>Initiates the closing of a TCP connection.</td>
</tr>
<tr>
<td>CLOSEudp</td>
<td>103</td>
<td>Initiates the closing of a UDP connection.</td>
</tr>
<tr>
<td>ENDTcpIPservice</td>
<td>104</td>
<td>Terminates the use of TCPIP services. All existing TCP connections are reset, all open UDP ports are canceled, and all IP protocols are released.</td>
</tr>
<tr>
<td>HANDLEnotice</td>
<td>105</td>
<td>Specifies the types of notifications to be received from TCPIP.</td>
</tr>
<tr>
<td>IShostLOCAL</td>
<td>106</td>
<td>Determines whether a given internet address is one of your host’s local addresses.</td>
</tr>
<tr>
<td>MONITORcommand</td>
<td>107</td>
<td>Instructs TCPIP to obey a file of commands.</td>
</tr>
<tr>
<td>MONITORquery</td>
<td>108</td>
<td>Obtains status information from the TCPIP virtual machine or requests that it performs certain functions.</td>
</tr>
<tr>
<td>OPENtcp</td>
<td>110</td>
<td>Initiates a TCP connection.</td>
</tr>
<tr>
<td>OPENudp</td>
<td>111</td>
<td>Initiates a UDP connection.</td>
</tr>
<tr>
<td>OPTIONtcp</td>
<td>112</td>
<td>Sets an option for a TCP connection.</td>
</tr>
<tr>
<td>RECEIVEtcp</td>
<td>113</td>
<td>Tells TCPIP that you are ready to receive data on a specified TCP connection.</td>
</tr>
</tbody>
</table>
Table 18. VMCF TCPIP CALLCODE Requests (continued)

<table>
<thead>
<tr>
<th>Call Code</th>
<th>Equates</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NRECEIVEudp</td>
<td>115</td>
<td>Tells TCPIP that your program is ready to receive a UDP datagram on a particular port.</td>
</tr>
<tr>
<td>SENDtcp</td>
<td>118</td>
<td>Sends data on a TCP connection. The SENDtcp transaction is unsuccessful if the receiving TCPIP virtual machine has insufficient buffer space to receive the data.</td>
</tr>
<tr>
<td>SENDudp</td>
<td>119</td>
<td>Sends a UDP datagram.</td>
</tr>
<tr>
<td>STATUStcp</td>
<td>120</td>
<td>Obtains a Connection Information Record giving the current status of a TCP connection.</td>
</tr>
<tr>
<td>FRECEIVEtcp</td>
<td>121</td>
<td>Tells TCPIP virtual machine that you are ready to receive data on a specified TCP connection. TCPIP does not respond or send a notification until the data has been placed in the receiving buffer or the connection has been closed.</td>
</tr>
<tr>
<td>FSENDtcp</td>
<td>122</td>
<td>Sends data to a TCP connection. FSENDtcp waits for available receiving buffer space in the TCPIP virtual machine before completing the VMCF transaction.</td>
</tr>
<tr>
<td>CLOSErawIP</td>
<td>123</td>
<td>Tells TCPIP that your program does not handle the protocol any longer. Any queued incoming packets are discarded.</td>
</tr>
<tr>
<td>OPENrawIP</td>
<td>124</td>
<td>Initiates a connection and tells TCPIP virtual machine that your program is ready to send and receive packets of a specified IP protocol.</td>
</tr>
<tr>
<td>RECEIVEmrawIP</td>
<td>125</td>
<td>Tells TCPIP that your program is ready to receive raw IP packets of a given protocol. Your program receives a RAWIPpacketsDELIVERED notification when a packet arrives.</td>
</tr>
<tr>
<td>SENDrawIP</td>
<td>126</td>
<td>Tells TCPIP virtual machine to send raw IP packets of a given protocol number.</td>
</tr>
<tr>
<td>PINGreq</td>
<td>127</td>
<td>Sends an ICMP echo request to a specified host and wait a specified time for a response.</td>
</tr>
</tbody>
</table>

VMCF TCPIP Communication CALLCODE Notifications

Table 19 lists the equate values for the CALLCODE field when VMCF TCPIP sends a notification to your program. The table includes a description of each CALLCODE response.

Table 19. VMCF TCPIP CALLCODE Notifications

<table>
<thead>
<tr>
<th>Notification Code</th>
<th>Equates</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFFERspaceAVAILABLE</td>
<td>10</td>
<td>Notification that there is space available to send data on this connection. The space is currently set to 8192 bytes of buffer space.</td>
</tr>
<tr>
<td>CONNECTIONstateCHANGED</td>
<td>11</td>
<td>Notification that the state of the connection between the TCPIP virtual machine and your program has changed.</td>
</tr>
<tr>
<td>DATAdelivered</td>
<td>12</td>
<td>Notification that the TCPIP virtual machine data was delivered to your program, after issuing a RECEIVEmtcp or FRECEIVEtcp call.</td>
</tr>
</tbody>
</table>
TCP/UDP/IP Initialization and Termination Procedures

This section contains information about procedures for initializing and terminating TCP/UDP/IP connections.

BEGINtcpIPservice

Your program performs the BEGINtcpIPservice call after doing a VMCF AUTHORIZE function, but before performing any other TCP/IP functions. The BEGINtcpIPservice call informs TCPIP that your virtual machine uses TCPIP services. An ENDtcpIPservice call is logically performed first, in the case where your virtual machine already has TCPIP resources allocated.
The TCPIP virtual machine responds using the VMCF REJECT function. The VMCF interrupt header, stored in your virtual machine by the response interrupt, contains a return code in the RETCODE field. The return code can be any of those listed for the \textit{BeginTcpIp} Pascal procedure (page 58).

**HANDLEnotice**

Your program performs the \textit{HANDLEnotice} call to specify the types of notifications to be received from TCPIP. The VADB field in the VMCF parameter list contains a notification mask, with 1 bit set for each notification you want to handle. The bit to be set for each notification type is shown in Figure 22.

Figure 22 shows the equates used for notification mask in the \textit{HANDLEnotice} call.

```plaintext
MaskBUFFERspaceAVAILABLE EQU X'00000001'
MaskCONNECTIONstateCHANGED EQU X'00000002'
MaskDATAdelivered EQU X'00000004'
MaskURGENTpending EQU X'00000020'
MaskUDPdgramDELIVERED EQU X'00000040'
MaskUDPdgramSPACEavailable EQU X'00000080'
MaskRAWIPpacketsDELIVERED EQU X'00004000'
MaskRAWIPspaceAVAILABLE EQU X'00008000'
MaskRESOURCESavailable EQU X'00040000'
MaskUDPResourcesAVAILABLE EQU X'00080000'
MaskPINGresponse EQU X'01000000'
```

Figure 22. Equates for Notification Mask in the HANDLEnotice Call

Each \textit{HANDLEnotice} call must specify all the notification types to be handled. Notification types specified in previous \textit{HANDLEnotice} calls are not stored.

The TCPIP virtual machine responds using the VMCF REJECT function. The VMCF interrupt header contains a return code in the RETCODE field. The return code can be any of those listed for the \textit{Handle} Pascal procedure (see “Handle” on page 62).

**ENDtcpIPservice**

Your program performs the \textit{ENDtcpIPservice} call when it has finished using TCPIP services. All existing TCP connections are reset (aborted), all open UDP port opens are canceled, and all IP protocols are released.
The TCPIP virtual machine responds using the VMCF REJECT function. The VMCF interrupt header indicates a return code of OK in the RETCODE field.

TCP CALLCODE Requests

The following sections describe the VMCF interrupt headers that are stored in your virtual machine for CALLCODE calls used to make TCP requests.

OPENtcp

The OPENtcp call initiates a TCP connection. Your program sends a Connection Information Record to TCPIP. Figure 23 gives the assembler format of the record. Figure 24 gives the equates for the assorted constants used to set up the record. For more information about the usage of the fields of the Connection Information Record, see "TcpOpen and TcpWaitOpen" on page 82.

Connection DS H
OpenAttemptTimeout DS F
Security DS H
Compartment DS H
Precedence DS X
BytesToRead DS F
UnackedBytes DS F
ConnectionState DS X
LocalSocket.Address DS F
LocalSocket.Port DS H
ForeignSocket.Address DS F
ForeignSocket.Port DS H

Figure 23. Assembler Format of the Connection Information Record for VM

UNSPECIFIEDconnection EQU -48
DEFAULTsecurity EQU 0
DEFAULTcompartment EQU 0
DEFAULTprecedence EQU 0
UNSPECIFIEDaddress EQU 0
UNSPECIFIEDport EQU X'FFFF'
ANintegerFLAGrequestERR EQU X'80000000'

Figure 24. Miscellaneous Assembler Constants

FUNC: SEND/RECV
VADA: Address of Connection Information Record initialized by your program
LENA: Length of Connection Information Record
VADB: Address of Connection Information Record to be filled in with TCPIP reply
LENB: Length of Connection Information Record
CONN: UNSPECIFIEDconnection
CALLCODE: OPENtcp
If the open attempt cannot be initiated, the TCPIP virtual machine responds using the VMCF REJECT function. The VMCF interrupt header contains a return code in the RETCODE field. The return code can be any of those listed for the TcpOpen Pascal procedure.

If the OPENtcp call was rejected because not enough TCPIP resources were available, a ZEROresources code is returned. When the TCPIP resources are available, a notice of RESOURCESavailable is sent to your program.

If the open attempt is not immediately rejected, the TCPIP virtual machine uses the VMCF RECEIVE function to receive the Connection Information Record describing the connection to be opened. If the connection then cannot be initiated, TCPIP responds using the VMCF REJECT function. The RETCODE field in the VMCF interrupt header is set as described in the previous paragraph.

If the open was successfully initiated, the TCPIP virtual machine responds using the VMCF REPLY function to send back the updated Connection Information Record. The Connection field of the Connection Information Record contains the connection number of the new connection. The RETCODE field in the VMCF interrupt header indicates OK, and the CONN field also contains the connection number of the new connection. The connection is not yet open; your program receives notification(s) during the opening sequence. For more information about NotificationInfoType, see the section on the Pascal under "Notification Record" on page 46 and see also "CALLCODE Notifications" on page 111.

**SENDtcp and FSENDtcp**

The SENDtcp or FSENDtcp calls send data on a TCP connection. For the advantages and disadvantages of using each function, and for information about sending TCP data, see "TcpFSend, TcpSend, and TcpWaitSend" on page 79.

<table>
<thead>
<tr>
<th>FUNC:</th>
<th>SEND</th>
</tr>
</thead>
<tbody>
<tr>
<td>VADA:</td>
<td>Address of data</td>
</tr>
<tr>
<td>LENA:</td>
<td>Length of data</td>
</tr>
<tr>
<td>VADB:</td>
<td>1 if push desired, else 0</td>
</tr>
<tr>
<td>LENB:</td>
<td>1 if urgent data, else 0</td>
</tr>
<tr>
<td>CONN:</td>
<td>Connection number from open</td>
</tr>
<tr>
<td>CALLCODE:</td>
<td>SENDtcp or FSENDtcp</td>
</tr>
</tbody>
</table>

If TCPIP can successfully queue the data for sending, it responds with the VMCF RECEIVE function. The VMCF interrupt header indicates a RETCODE of OK.

If TCPIP cannot queue the data for sending, it responds with the VMCF REJECT function. The RETCODE field indicates the type of error. The return code can be any of those listed for the TcpSend Pascal procedure. For a list of the return codes, see "TcpFSend, TcpSend, and TcpWaitSend" on page 79.

If the SENDtcp transaction is unsuccessful, because of insufficient space in the buffer of the receiving TCPIP virtual machine, a return code of NOBUFFERSPACE is placed in the RETCODE field. A notification of BUFFERspaceAVAILABLE is sent, on this connection, when the space is available to process data.

TcpFSend is the same as FSENDtcp. If TCPIP cannot accept the data, because of a buffer shortage, it does not respond immediately. Instead, it waits until space is available, and then uses VMCF RECEIVE to receive the data. While it is waiting, if the connection is reset, it responds with VMCF REJECT, with a return code as described previously. In summary, TCPIP may not respond immediately to FSENDtcp, and the response, after waiting, may indicate either success or failure.
TCPIP responds with REJECT, your program can check the ANintegerFLAGRequestERR bit in the ANINTEGR field to determine if the request was rejected during initial or retry processing (bit on) or because of connection closing (bit off).

Your program does not need to wait for a response from SENDtcp or FSENDtcp VMCF transaction. It can issue functions involving other connections, before receiving a response from making a SENDtcp or FSENDtcp VMCF transaction.

There is a limit of 50 outstanding VMCF transactions for each virtual machine; therefore, your program can have FSENDtcp functions outstanding on only 50 connections at a time. If your application needs more outstanding sends, use the SENDtcp function.

**FRECEIVEtcp**

The FRECEIVEtcp call tells TCPIP that you are ready to receive data on a specified TCP connection. TCPIP does not respond or send a notification notice until data is received or the connection is closed. Consequently, each outstanding FRECEIVEtcp function results in an outstanding VMCF transaction. There is a limit of 50 outstanding VMCF transactions for each virtual machine; you can therefore have FRECEIVEtcp functions outstanding on only 50 connections at one time. If your application needs more outstanding receives, use the RECEIVEtcp function.

Your program does not need to wait for a response from FRECEIVEtcp. It can issue functions involving other connections, before receiving a response from FRECEIVEtcp.

For general information about receiving TCP data, see the TcpFReceive Pascal procedure under "TcpFReceive, TcpReceive, and TcpWaitReceive" on page 76.

**FUNCTIONS**

SEND/RECV

VADA: 0
LENA: 1
VADB: Address of buffer to receive data
LENB: Length of buffer to receive data
CONN: Connection number from open
CALLCODE: FRECEIVEtcp

If TCPIP accepts the request, your program receives no response until TCPIP is ready to deliver data to your buffer, or until the request is canceled, because the connection has closed without delivering data.

When TCPIP is ready to deliver data for this connection, it issues a VMCF REPLY function. Significant fields in the VMCF interrupt header are:

**LENB** Indicates the residual count. Subtract this from the size of your buffer (LENB value in parameter list) to determine the number of bytes actually delivered.

**ANINTEGR**

Contains a value where the high-order byte is nonzero if data was pushed; otherwise, it is zero. The low-order three bytes are interpreted as a 24-bit integer, indicating the offset of the byte following the last byte of urgent data, measured from the first byte of data delivered to your buffer. If it is zero or a negative number, then there is no urgent data pending.

**CONN**

Specifies the connection number.
If TCPIP responds with the VMCF REJECT function (VMCFRJCT flag set in the VMCF interrupt header), then one of the following occurred:

- TCPIP did not accept the request, in which case the ANintegerFLAGrequestERR bit in ANINTEGR is on.
- TCPIP accepted the request initially, but the connection closed before data was delivered. ANintegerFLAGrequestERR bit in ANINTEGR is off. In this case, the RETCODE field indicates one of the reason codes listed for CONNECTIONstateCHANGED with the NewState field set to NONEXISTENT. For more information, see 2 on page 49.

**Note:** Your program does not have to take any special action in this case, because it receives one or more CONNECTIONstateCHANGED notifications indicating that the connection is closing.

**RECEIVEtcp**

The RECEIVEtcp call tells TCPIP that you are ready to receive data on a specified TCP connection. Unlike FRECEIVEtcp, TCPIP responds immediately to RECEIVEtcp. You can have more than 50 receive requests pending using RECEIVEtcp without exceeding the limit of 50 outstanding VMCF transactions.

For more information about receiving TCP data, see the TcpReceive Pascal procedure under "TcpFReceive, TcpReceive, and TcpWaitReceive" on page 76.

**CLOSEtcp**

The CLOSEtcp call initiates the closing of a TCP connection. For more information about the close connection call, see the Pascal procedure, "TcpClose" on page 75.
VMCF Interface

ABORTtcp

The ABORTtcp call terminates a TCP connection. For more information about the abort connection call, see the Pascal procedure, "TcpAbort" on page 73.

FUNC: SEND
VADA: 0
LENA: 1
VADB: 0
LENB: 0
CONN: Connection number from open
CALLCODE: ABORTtcp

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt buffer contains the return code. It is one of those listed for the TcpAbort Pascal procedure.

STATUStcp

The STATUStcp call obtains a Connection Information Record giving the current status of a TCP connection. For the assembler format of the Connection Information Record, see Figure 23 on page 103. For more information about the connection status call, see the Pascal procedure, "TcpStatus" on page 85.

FUNC: SEND/RECV
VADA: 0
LENA: 1
VADB: Address of Connection Information Record to fill in
LENB: Length of Connection Information Record to fill in
CONN: Connection number from open
CALLCODE: STATUStcp

TCPIP responds with the VMCF REPLY function, filling in the record whose address you supplied in LENB. The record is valid only if the return code, in the RETCODE field of the VMCF interrupt header, returns OK. Otherwise, the return code is one of those listed for the TcpStatus Pascal procedure.

OPTIONtcp

The OPTIONtcp call sets an option for a TCP connection. For more information about the connection options, see the Pascal procedure, "TcpOption" on page 84.

FUNC: SEND
VADA: 0
LENA: 1
VADB: Option name
LENB: Option value
CONN: Connection number from open
CALLCODE: OPTIONtcp

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt buffer contains the return code. The return code is one of those listed for the Pascal TcpOption procedure.

UDP CALLCODE Requests

The following sections describe the VMCF interrupt headers, which are stored in your virtual machine, for CALLCODE calls used to make UDP requests.
**OPENudp**

The OPENudp call opens a UDP port. For more information about the UDP open function, see the Pascal procedure, "UdpOpen" on page 87.

**TCPIP** responds with the VMCF REJECT function. The RETCODE field in the VMCF interrupt header can be any of the return codes listed for the UdpOpen Pascal procedure. If the OPENudp call was rejected, because not enough TCPIP resources were available, a UDPzeroRESOURCES code is returned. When the TCPIP resources are available, a notice of UDPresourcesAVAILABLE is sent to your program.

**SENDudp**

The SENDudp call sends a UDP datagram. For more information about the UDP send function, see the Pascal procedure, "UdpSend" on page 89.

If TCPIP can send the datagram, it responds with the VMCF RECEIVE function. The RETCODE field in the VMCF interrupt header contains a return code of OK. If TCPIP cannot send the datagram, it responds with the VMCF REJECT function. The RETCODE field contains one of the return codes listed for the UdpSend Pascal procedure. When the buffer space is not available to process the data, an error is returned. The notification message of UDPdatagramSPACEavailable is sent to your program when the buffer space is available to process data.

**NRECEIVEudp**

The NRECEIVEudp call tells TCPIP that your program is ready to receive a UDP datagram on a particular port. TCPIP responds immediately to inform you whether it accepted the request. If TCPIP has accepted your request, your program receives a UDPdatagramDELIVERED notification when a datagram arrives. For more information about receiving UDP datagrams, see the Pascal procedure, "UdpNReceive" on page 87.
TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt header contains one of the return codes listed for the UdpNReceive Pascal procedure.

CLOSEudp

The CLOSEudp call closes a UDP port. For more information about the CLOSEudp call, see the Pascal procedure, "UdpClose" on page 86.

```
FUNC: SEND
VADA: 0
LENA: 1
VADB: 0
LENB: 0
CONN: Connection number
CALLCODE: CLOSEudp
```

TCPIP responds with the VMCF REJECT function. The RETCODE field in the VMCF interrupt header can be any of the return codes listed for the UdpClose Pascal procedure. If the return code is OK, and your program specified UNSPECIFIEDport as the port number, the actual port number assigned is encoded in the CONN field of the interrupt header. Add the value of 32 768 to the value in the CONN field, using unsigned arithmetic, to compute the port number.

IP CALLCODE Requests

The following sections describe the VMCF interrupt headers, which are stored in your virtual machine, for CALLCODE calls used to make IP requests.

OPENrawip

The OPENrawip call tells TCPIP that your program is ready to send and receive packets of a specified IP protocol. For more information, see the Pascal procedure, "RawIpOpen" on page 68.

```
FUNC: SEND/RECV
VADA: 0
LENA: 1
VADB: 0
LENB: 0
CONN: Protocol number
CALLCODE: OPENrawip
```

TCPIP uses the VMCF REJECT function to respond to the request. The RETCODE field of the VMCF interrupt header contains one of the return codes listed for the RawIpOpen Pascal procedure.

SENDrawip

The SENDrawip call sends raw IP packets of a given protocol number. For more information, see the Pascal procedure, "RawIpSend" on page 69.

```
FUNC: SEND/RECV
VADA: Address of buffer containing packets to send
LENA: Length of buffer
VADB: 0
LENB: 0
CONN: (Number of packets shifted left 8 bits) + protocol number
CALLCODE: SENDrawip
```
If TCPIP immediately determines that the request cannot be fulfilled, it responds with the VMCF REJECT function. Otherwise, it uses the VMCF RECEIVE function to receive your data, followed by VMCF REJECT. The RETCODE field of the VMCF interrupt header contains one of the return codes listed for the RawIpSend Pascal procedure. If TCPIP virtual machine is out of buffers, the data is rejected and a return code of NObufferSPACE is returned. When buffer space is available, the notification of RAWIPspaceAVAILABLE is sent to your program.

**RECEIVErawip**

The RECEIVErawip call tells TCPIP that your program is ready to receive raw IP packets of a given protocol. Your program receives a RAWIIPacketsDELIVERED notification when a packet arrives. For information about the RAWIIPacketsDELIVERED notification record, see the Pascal procedure, “RawIpReceive” on page 68, and the section on the Pascal NotificationInfoType under “Notification Record” on page 46.

```
FUNC: SEND/RECV
VADA: 0
LENA: 1
VADB: 0
LENB: Length of your buffer
CONN: Protocol number
CALLCODE: RECEIVErawip
```

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt header contains one of the return codes listed for the RawIpReceive Pascal procedure.

**CLOSErawip**

The CLOSErawip call tells TCPIP that your program is ready to cease sending and receiving packets of a specified IP protocol. For more information, see the Pascal procedure, “RawIpClose” on page 67.

```
FUNC: SEND/RECV
VADA: 0
LENA: 1
VADB: 0
CONN: Protocol number
CALLCODE: CLOSErawip
```

TCPIP uses the VMCF REJECT function to respond to the request. The RETCODE field of the VMCF interrupt header contains one of the return codes listed for the RawIpClose Pascal procedure.

**CALLCODE System Queries**

The following sections describe the VMCF interrupt headers, which are stored in your virtual machine, for CALLCODE calls used to make system queries.

**ISHostLOCAL**

The ISHostLOCAL call determines whether a given internet address is one of your host’s local addresses. For more information about this procedure, see the Pascal procedure “LocalAddress” on page 63.
TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt header contains the return code, as described in the IsLocalAddress Pascal procedure section.

**MONITORcommand**

The MONITORcommand call instructs TCPIP to obey a file of commands. For more information, see the Pascal procedure, [“MonCommand” on page 64](#), and for more information about the OBEYFILE command, which uses the MonCommand procedure, see [TCP/IP Planning and Customization](#).

Owner DS CL8
DatasetPassword DS CL8
FullDatasetName DS CL44
MemberName DS CL8
DDName DS CL8

*Figure 25. Assembler Format of the SpecOfFile Type Record for VM*

If TCPIP cannot process the request, it responds immediately with the VMCF REJECT function. Otherwise, it uses the VMCF RECEIVE function to receive the SpecOfFile record provided by your program. It then attempts to process the file, and uses the VMCF REJECT function to report the return code. In either case, the return code is one of those specified for the MonCommand Pascal procedure.

**MONITORquery**

The MONITORquery call obtains status information from the TCPIP virtual machine or to request that it performs certain functions. For more information, see the Pascal procedure, [“MonQuery” on page 65](#). Assembler formats of constants and records used with MONITORquery are:

COMMANDcpCMD EQU 6
COMMANDdropCONNECTION EQU 8
QUERYhomeONLY EQU 9

*Figure 26. Equates for MonQueryRecordType used in the MONITORquery Call*
The Pascal type HomeOnlyListType is an array of 64 InternetAddressType elements found in the COMMMAC MACLIB file. The size of InternetAddressType is a fullword.

If TCPIP cannot process the request, it responds immediately with the VMCF REJECT function. Otherwise, it uses the VMCF RECEIVE function to receive the MonQueryRecord describing your request, followed by either a VMCF REPLY to send the response to your return buffer (not applicable to COMMAND dropCONNECTION), or a VMCF REJECT to send a return code but no return data. For information about the return codes and the data returned (if any), see the Pascal procedure, “MonQuery” on page 65.

PINGreq

The PINGreq call sends an ICMP echo request (PING request) to a specified host and wait a specified time for a response. For more information, see the Pascal procedure “PingRequest” on page 67.

TCPIP uses the VMCF REJECT function to respond to the request. The RETCODE field of the VMCF interrupt header contains one of the return codes listed for the PingRequest Pascal procedure. If the return code is OK, your program receives a PINGresponse notification later.

CALLCODE Notifications

The following sections describe the VMCF interrupt headers that are stored in your virtual machine for the various types of notifications. The action that your program should take is also indicated.

For more information about the various notification types, see the Pascal NotificationInfoType record under “Notification Record” on page 46.
The VMCF transaction for a notification must be completed before TCPIP sends your program another notification. You must ensure that your program takes the VMCF actions in the following sections, or TCPIP cannot communicate further with your program.

### BUFFERspaceAVAILABLE

This interrupt header notifies you that there is space available to send data on this connection. The space is currently set to 8192 bytes of buffer space. The notification is sent after making a SENDtcp call and receiving an unsuccessful return code of NObufferSPACE in the RETCODE field.

**FUNC:** SEND
**JOBNAME:** Name of the TCPIP virtual machine
**VADB:** Space available to send on this connection, in bytes. Currently always 8192
**CONN:** Connection number
**CALLCODE:** BUFFERspaceAVAILABLE
**RETCODE:** OK

Your program should issue the VMCF REJECT function, with VMCF parm list copied from the interrupt header, with the following fields changed:

- V1: 0
- V2: 0
- **FUNC:** REJECT

### CONNECTIONstateCHANGED

This interrupt header notifies you that the state of the connection between the TCPIP virtual machine and your program has changed.

**FUNC:** SEND
**JOBNAME:** Name of the TCPIP virtual machine
**VADB:** New connection state
**LENB:** Reason for state change, if new state is NONEXISTENT
**CONN:** Connection number
**CALLCODE:** CONNECTIONstateCHANGED
**RETCODE:** OK

Your program should issue the VMCF REJECT function, with VMCF parm list copied from the interrupt header, with the following fields changed:

- V1: 0
- V2: 0
- **FUNC:** REJECT

### DATAdelivered

This interrupt header notifies you that the TCPIP virtual machine data was delivered to your program, after issuing a RECEIVETcp or FRECEIVETcp call.
VMCF Interface

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNC: SEND</td>
<td>Name of the TCPIP virtual machine</td>
</tr>
<tr>
<td>LENA:</td>
<td>Length of data being delivered</td>
</tr>
<tr>
<td>VADB:</td>
<td>Non-zero if data was pushed, else zero.</td>
</tr>
<tr>
<td>LENB:</td>
<td>The offset of the byte following the last byte of urgent data, measured</td>
</tr>
<tr>
<td></td>
<td>from the first byte of data delivered to your buffer. If zero or negative</td>
</tr>
<tr>
<td></td>
<td>then there is no urgent data pending.</td>
</tr>
<tr>
<td>CONN:</td>
<td>Connection number</td>
</tr>
<tr>
<td>CALLCODE:</td>
<td>DATA delivered</td>
</tr>
<tr>
<td>RETCODE:</td>
<td>OK</td>
</tr>
</tbody>
</table>

Your program should issue the VMCF RECEIVE function, with VMCF parm list copied from the interrupt header, with the following fields changed:

- V1: 0
- V2: 0
- FUNC: RECEIVE
- VADA: Address of your buffer to receive data. Buffer should be at least as long as indicated by LENA. LENA is no larger than the buffer length you specified using the RECEIVEtcp function.

**URGENTpending**

This interrupt header notifies you that there is queued incoming data on a TCP connection not yet received by your program.

- FUNC: SEND
- JOBNAME: Name of the TCPIP virtual machine
- VADB: Number of bytes of queued incoming data not yet received by your program.
- LENB: Subtract 1 from LENB to get the offset of the byte following the last byte of urgent data, measured from the first byte not yet received by your program. If this quantity is zero or negative then there is no urgent data pending.
- CONN: Connection number
- CALLCODE: URGENTpending
- RETCODE: OK

Your program should issue the VMCF REJECT function, with VMCF parm list copied from the interrupt header, with the following fields changed:

- V1: 0
- V2: 0
- FUNC: REJECT

**UDPdatagramDELIVERED**

This interrupt header notifies you that the UDP datagram has been delivered to your program after issuing a NRECEIVEudp call to the TCPIP virtual machine.
VMCF Interface

FUNC: SEND
JOBNAME: Name of the TCPIP virtual machine
LENA: Length of data being delivered.
VADB: Source port
LENB: Source address
ANINTEGR: Length of entire datagram excluding UDP header. If larger than LENA then the datagram was too large to fit into the buffer size specified in NRECEIVEudp call, and has been truncated.
CONN: Connection number
CALLCODE: UDPdatagramDELIVERED
RETCODE: OK

Your program should issue the VMCF RECEIVE function, with VMCF parm list copied from the interrupt header, with the following fields changed:

V1: 0
V2: 0
FUNC: RECEIVE
VADA: Address of your buffer to receive data. Buffer should be at least as long as indicated by LENA.

UDPdatagramSPACEavailable

This interrupt header notifies you that buffer space is available to process the data, after an error occurred performing a SENDudp call.

FUNC: SEND
JOBNAME: Name of the TCPIP virtual machine
CONN: Connection number
CALLCODE: UDPdatagramSPACEavailable
RETCODE: OK

Your program should issue the VMCF REJECT function, with VMCF parm list copied from the interrupt header, with the following fields changed:

V1: 0
V2: 0
FUNC: REJECT

RAWIPpacketsDELIVERED

This interrupt header notifies you that your buffer has received the raw IP packets.

FUNC: SEND
JOBNAME: Name of the TCPIP virtual machine
ANINTEGR: Total length of datagram being delivered (including IP header)
LENA: Length of data (including IP header) that TCPIP delivers to you.
CONN: Low-order byte is protocol number, 3 high order bytes is number of packets, currently always 1.
CALLCODE: RAWIPpacketsDELIVERED
RETCODE: OK

Your program should issue the VMCF RECEIVE function, with VMCF parm list copied from the interrupt header, with the following fields changed:

V1: 0
V2: 0
FUNC: RECEIVE
VADA: Address of your buffer to receive data. Buffer should be at least as long as indicated by LENA.
RAWIPspaceAVAILABLE

This interrupt header notifies you that buffer space is available to process the data. This notification is sent after the SENDrawip call was rejected by TCPIP virtual machine because of insufficient buffer space.

FUNC: SEND
JOBNAME: Name of the TCPIP virtual machine
LENB: Space available. Always equals maximum IP datagram size.
CONN: Protocol number
CALLCODE: RAWIPspaceAVAILABLE
RETCODE: OK

Your program should issue the VMCF REJECT function, with VMCF parm list copied from the interrupt header, with the following fields changed:

V1: 0
V2: 0
FUNC: REJECT

RESOURCESavailable

This interrupt header notifies you that the resources needed to initiate a TCP connection are now available. This notification is sent only if a previous OPENtcp call received a ZEROresources return code.

FUNC: SEND
JOBNAME: Name of the TCPIP virtual machine
CALLCODE: RESOURCESavailable
RETCODE: OK

Your program should issue the VMCF REJECT function, with VMCF parm list copied from the interrupt header, with the following fields changed:

V1: 0
V2: 0
FUNC: REJECT

UDPresourcesAVAILABLE

This interrupt header notifies you that the resources needed to initiate a UDP connection are now available. This notification is sent only if a previous OPENudp call received a UDPrzeroRESOURCES return code.

FUNC: SEND
JOBNAME: Name of the TCPIP virtual machine
CALLCODE: UDPresourcesAVAILABLE
RETCODE: OK

Your program should issue the VMCF REJECT function, with VMCF parm list copied from the interrupt header, with the following fields changed:

V1: 0
V2: 0
FUNC: REJECT

PINGresponse

This interrupt header notifies you that your ping request from the PINGreq call has been received or that the time-out limit or your request has been reached.
VMCF Interface

FUNC: SEND
JOBNAME: Name of the TCPIP virtual machine
VADB: High order word of elapsed time, in TOD clock format
       Valid only if ANINTEGR is zero
LENB: Low order word of elapsed time, in TOD clock format
       Valid only if ANINTEGR is zero
ANINTEGR: Return code from ping operation
CALLCODE: PINGresponse
RETCODE: OK

Your program should issue the VMCF REJECT function, with VMCF parm list copied from the interrupt header, with the following fields changed:

V1: 0
V2: 0
FUNC: REJECT

DUMMYprobe

This interrupt header notifies you that the TCPIP virtual machine is monitoring your machine so it can determine if it logs off or resets unexpectedly.

FUNC: SEND
JOBNAME: Name of the TCPIP virtual machine
CALLCODE: DUMMYprobe
RETCODE: OK

Your program should leave this message pending.

ACTIVEprobe

This interrupt header notifies you that the TCPIP virtual machine is monitoring your machine so it can determine if it is still responsive.

FUNC: SEND
JOBNAME: Name of the TCPIP virtual machine
CALLCODE: ACTIVEprobe
RETCODE: OK

Your program should issue the VMCF REJECT function, with the VMCF parameter list copied from the interrupt header and with the following fields changed:

V1: 0
V2: 0
FUNC: REJECT

The response to this message must be made within one minute after the associated interrupt is received.
Chapter 4. Inter-User Communication Vehicle Sockets

The Inter-User Communication Vehicle (IUCV) socket API is an assembler language application programming interface that can be used with TCP/IP. While not every C socket library function is provided, all of the basic operations necessary to communicate with other socket programs are present.

Prerequisite Knowledge

This chapter assumes you have a working knowledge of IUCV, as documented in z/VM CP Programming Services.

You must also know how and when to use the CMS CMSIUCV macro or the GCS IUCVCOM macro, depending on the execution environment, as documented in z/VM CMS Application Development Reference for Assembler or z/VM Group Control System, respectively.

You should also have a working knowledge of TCP/IP sockets.

Available Functions

Only these functions are available when you use the IUCV socket interface:

Table 20. Socket functions available using IUCV

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT</td>
<td>READ</td>
</tr>
<tr>
<td>BIND</td>
<td>READV</td>
</tr>
<tr>
<td>CLOSE</td>
<td>RECV</td>
</tr>
<tr>
<td>CONNECT</td>
<td>RECVFROM</td>
</tr>
<tr>
<td>FCNTL</td>
<td>RECVMSG</td>
</tr>
<tr>
<td>GETCLIENTID</td>
<td>SELECT</td>
</tr>
<tr>
<td>GETHOSTID</td>
<td>SELECTEX</td>
</tr>
<tr>
<td>GETHOSTNAME</td>
<td>SEND</td>
</tr>
<tr>
<td>GETPEERNAME</td>
<td>Sendmsg</td>
</tr>
<tr>
<td>GETSOCKNAME</td>
<td>Sendto</td>
</tr>
<tr>
<td>GETSOCKOPT</td>
<td>Setsockopt</td>
</tr>
<tr>
<td>GIVESOCKET</td>
<td>SHUTDOWN</td>
</tr>
<tr>
<td>IOCTL</td>
<td>SOCKET (AF_INET sockets only)</td>
</tr>
<tr>
<td>LISTEN</td>
<td>TAKESOCKET</td>
</tr>
<tr>
<td>MAXDESC</td>
<td>WRITE</td>
</tr>
<tr>
<td></td>
<td>WRITEV</td>
</tr>
</tbody>
</table>
TCP/IP sockets are manipulated by using the following assembler macros:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IUCV</td>
<td>HCPGPI</td>
<td>Provides the mechanisms for setting values in the IUCV input parameter list and for executing the IUCV instruction.</td>
</tr>
<tr>
<td>IPARML</td>
<td>HCPGPI</td>
<td>Mapping macro for the IUCV parameter list and the external interrupt buffer.</td>
</tr>
<tr>
<td>HNDIUCV</td>
<td>DMSGPI</td>
<td>Informs CMS that your program wishes to handle IUCV or APPC/VM interrupts. Only those interrupts occurring on IUCV paths that your application created will be routed to your program.</td>
</tr>
<tr>
<td>CMSIUCV</td>
<td>DMSGPI</td>
<td>Used to perform IUCV CONNECT and SEVER functions. It enables multiple IUCV or APPC/VM applications to run at the same without interference.</td>
</tr>
<tr>
<td>IUCVINI</td>
<td>GCTGPI</td>
<td>Similar to HNDIUCV, but for the GCS execution environment.</td>
</tr>
<tr>
<td>IUCVCOM</td>
<td>GCTGPI</td>
<td>Similar to CMSIUCV, but for the GCS execution environment. In addition to providing multiple application support, it provides a way for GCS programs running in problem state to use IUCV services.</td>
</tr>
</tbody>
</table>

A typical socket application uses only four IUCV operations: CONNECT, SEND (with reply), PURGE, and SEVER. CONNECT establishes the IUCV connection with the TCP/IP virtual machine, SEND performs initialization and socket operations, PURGE cancels an outstanding socket operation, and SEVER deletes the IUCV connection.

If an IUCV operation completes with condition code 0, the requested operation was successfully started. An IUCV interrupt will be received when the operation completes. When your interrupt routine receives control, it receives a pointer to the external interrupt buffer which contains information about the IUCV function that completed. The IPTYPE field of the external interrupt buffer (mapped by IPARML) identifies the interrupt:

<table>
<thead>
<tr>
<th>IPTYPE</th>
<th>Interrupt Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'02'</td>
<td>Connection Complete</td>
<td>Acknowledgement that TCP/IP has accepted your request to establish an IUCV connection (IUCV CONNECT)</td>
</tr>
<tr>
<td>X'03'</td>
<td>Connection Severed</td>
<td>Your IUCV connection has been deleted by TCP/IP</td>
</tr>
<tr>
<td>X'07'</td>
<td>Message complete</td>
<td>The requested socket function has completed</td>
</tr>
</tbody>
</table>

**Note:** IPTYPE is byte 3 of the external interrupt buffer.

While there are other types of IUCV interrupts, they are not normally seen on TCP/IP IUCV socket paths. *z/VM CP Programming Services* has a complete description of each interrupt type.
If an IUCV operation completes with condition code 1, the requested function could not be performed. The exact cause of the error is stored in byte 3 of the IUCV parameter list (IPRCODE). See the description of each IUCV function in z/VM CP Programming Services for the possible return codes.

Note: CMSIUCV and IUCVCOM use return codes in general register 15 to indicate the success or failure of the operation. Refer to z/VM CMS Application Development Reference for Assembler or z/VM Group Control System for details on these system services.

If an IUCV PURGE operation completes with condition code 2, it means that TCP/IP has already finished processing the socket request.

Preparing to use the IUCV Socket API

Before the socket functions can be used, an IUCV socket API environment must be established. This is done in two steps:

1. Establish an IUCV connection to the TCP/IP service virtual machine.
2. Send an initialization message to TCP/IP, identifying your application and defining how the IUCV connection will be used.

Establishing an IUCV connection to TCP/IP

To create an IUCV connection to the TCP/IP service virtual machine, issue IUCV CONNECT with the following parameters:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>USERID</td>
<td>The user ID of the TCP/IP virtual machine.</td>
</tr>
<tr>
<td>PRTY</td>
<td>NO</td>
</tr>
<tr>
<td>PRMDATA</td>
<td>YES</td>
</tr>
<tr>
<td>QUIESCE</td>
<td>NO</td>
</tr>
<tr>
<td>MSGLIM</td>
<td>If this IUCV connection may have more than one outstanding socket function on it at the same time, set MSGLIM to the maximum number of socket calls that may be outstanding simultaneously on this path. Otherwise, set it to zero.</td>
</tr>
<tr>
<td>USERDTA</td>
<td>Binary zeros</td>
</tr>
<tr>
<td>CONTROL</td>
<td>NO</td>
</tr>
</tbody>
</table>

If IUCV CONNECT returns condition code 0, you subsequently receive either a Connection Complete external interrupt or a Connection Severed external interrupt. If you receive a Connection Severed interrupt now or later, see "Severing the IUCV Connection" on page 121 for more information.

To ensure that your program does not interfere with other IUCV or APPC/VM applications, your program should use the HNDIUCV and CMSIUCV macros in CMS, or the IUCVINI and IUCVCOM macros in GCS.

Initializing the IUCV Connection

If you receive a Connection Complete interrupt in response to IUCV CONNECT, then TCP/IP has accepted the connection request.
IUCV Sockets

Your program responds by sending an *initialization message* using IUCV SEND to TCP/IP, identifying your application and the way that it will use the IUCV socket interface.

When the IUCV SEND completes, then, if the IPAUDIT field shows no error, the reply buffer has been filled. The `maxsock` field indicates that maximum number of sockets you can open on this IUCV path at the same time.

Your program can now issue any supported socket call. See "Issuing Socket Calls" on page 122.

The initialization message is sent using an IUCV SEND with the following parameters:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRGCLS</td>
<td>0</td>
</tr>
<tr>
<td>DATA</td>
<td>BUFFER</td>
</tr>
<tr>
<td>BUFLEN</td>
<td>20</td>
</tr>
<tr>
<td>TYPE</td>
<td>2WAY</td>
</tr>
<tr>
<td>ANSLEN</td>
<td>8</td>
</tr>
<tr>
<td>PRTY</td>
<td>NO</td>
</tr>
</tbody>
</table>

**BUFFER** Points to a buffer in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>8</td>
<td>Constant ‘IUCVAPI ‘. The trailing blank is required.</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>2</td>
<td>Halfword integer. Maximum number of sockets that can be established on this IUCV connection. minimum: 50, Default: 50.</td>
</tr>
<tr>
<td>10</td>
<td>apitype</td>
<td>2</td>
<td>X'0002‘. Provided for compatibility with prior implementations of TCP/IP. Use X'0003‘ instead. X'0003‘- Any number of socket requests may be outstanding, on this IUCV connection at the same time. For more information, see &quot;Overlapping Socket Requests&quot; on page 122.</td>
</tr>
<tr>
<td>12</td>
<td>subtaskname</td>
<td>8</td>
<td>Eight printable characters. The combination of your user ID and subtaskname uniquely identifies the TCP/IP client using this path. This value is displayed by the NETSTAT CLIENT command.</td>
</tr>
</tbody>
</table>

**ANSBUF** Points to a buffer to contain the reply from TCP/IP:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>maxsock</td>
<td>4</td>
<td>The maximum socket number that your application can use on this path. The minimum socket number is always 0. Your application chooses a socket number for the accept, socket, and takesocket calls.</td>
</tr>
</tbody>
</table>
Note: A single virtual machine can establish more than one IUCV path to TCP/IP, but a different subtaskname must be specified on each IUCV path. If the same subtaskname is specified for more than one IUCV path, TCP/IP severs the existing path with that subtaskname.

Severing the IUCV Connection

An IUCV connection to TCP/IP can be severed (deleted) by your application or by TCP/IP at any time.

Sever by the Application

Your application can sever a socket API IUCV path at any time by calling IUCV SEVER with USERDTA specified as 16 bytes of binary zeros. TCP/IP cleans up all sockets associated with the IUCV path.

Clean-Up of Stream Sockets

The TCP connection corresponding to each stream socket associated with the IUCV path is reset. In the case of a listening socket, all connections in the process of opening, or already open and in the accept queue, are reset.

If your program closed a stream socket earlier, the corresponding TCP connection might still be in the process of closing. Such connections, which are no longer associated with any socket, are not reset when your program severs the IUCV path.

Sever by TCP/IP

TCP/IP severs a socket API IUCV path only in case of shutdown or an unexpected error. The 16-byte IPUSER field in the SEVER external interrupt indicates the reason for the sever. The reason is coded in EBCDIC. The following are possible reason codes and explanations:

<table>
<thead>
<tr>
<th>Reason Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IUCVCHECKRC</td>
<td>IUCV error detected. This code is used only before or during processing of the initialization message.</td>
</tr>
<tr>
<td>SHUTTINGDOWN</td>
<td>TCP/IP service is being shut down. This code is used only in response to the Pending Connection interrupt.</td>
</tr>
<tr>
<td>BAD PATH ID</td>
<td>An attempt was made to exceed the maximum number of IUCV connections support by the target TCPIP virtual machine.</td>
</tr>
<tr>
<td>NULL SAVED NAME</td>
<td>A software error occurred in TCP/IP. This code is used only before or during processing of the initialization message.</td>
</tr>
<tr>
<td>BAD INIT MSG LEN</td>
<td>Your program sent an initialization message that was not of the expected length.</td>
</tr>
<tr>
<td>REQUIREDCONSTANT</td>
<td>The first 8 bytes of your initialization message were not &quot;IUCVAPI &quot;.</td>
</tr>
<tr>
<td>BAD API TYPE</td>
<td>The apitype field in your initialization message contained an incorrect value.</td>
</tr>
<tr>
<td>RESTRICTED</td>
<td>Your virtual machine is not permitted to use TCP/IP.</td>
</tr>
</tbody>
</table>
IUCV Sockets

NO MORE CCBS
Your IUCV path cannot be accepted because there are no more client control blocks available in the TCPIP virtual machine.

NO CCB!!!!
A software error occurred in TCP/IP. Contact your system support personnel or the IBM Support Center.

Issuing Socket Calls

The following section describes how to issue an IUCV socket call.

All socket calls are invoked by issuing an IUCV SEND with the following parameters:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRGCLS</td>
<td>The high-order halfword specifies the socket call. For most calls, the low-order halfword specifies the socket descriptor.</td>
</tr>
<tr>
<td>DATA</td>
<td>BUFFER or PRMMSG, depending on call</td>
</tr>
<tr>
<td>BUFLIST</td>
<td>If DATA=BUFFER, then either YES or NO as desired. If DATA=PRMMSG, not applicable.</td>
</tr>
<tr>
<td>BUFFER</td>
<td>If DATA=BUFFER, points to the buffer (or buffer list) in the format required by the call. If DATA=PRMMSG, not applicable.</td>
</tr>
<tr>
<td>BUFLEN</td>
<td>If DATA=BUFFER, length of buffer. If DATA=PRMMSG, not applicable.</td>
</tr>
<tr>
<td>PRMMSG</td>
<td>If DATA=PRMMSG, data as required by the call. DATA=PRMMSG is not allowed when ANSLIST=YES. If DATA=BUFFER, not applicable.</td>
</tr>
<tr>
<td>TYPE</td>
<td>2WAY</td>
</tr>
<tr>
<td>ANSLIST</td>
<td>Either YES or NO as desired. DATA=PRMMSG is not allowed when ANSLIST=YES.</td>
</tr>
<tr>
<td>ANSBUF</td>
<td>Points to a buffer to contain the reply from TCP/IP.</td>
</tr>
<tr>
<td>ANSLEN</td>
<td>Length of the reply buffer</td>
</tr>
<tr>
<td>PRTY</td>
<td>NO</td>
</tr>
<tr>
<td>SYNC</td>
<td>YES or NO as desired. Applications that need to serve multiple clients at the same time should specify SYNC=NO. SYNC=YES will block the entire virtual machine from execution until the function is complete.</td>
</tr>
</tbody>
</table>

Overlapping Socket Requests

Your program may have more than one socket call outstanding on the same IUCV path. There are some restrictions on the types of calls that are queued simultaneously for the same socket descriptor.

The following list describes the restrictions for each type of socket call:

- Multiple read-type calls (READ, READV, RECV, RECVFROM, RECVMSG) and multiple write-type calls (WRITE, WRITEV, SEND, SENDTO, SENDMSG), for the same socket, can be queued simultaneously. The read-type calls are satisfied in order, independently of the write-type calls. Similarly, the write-type calls are satisfied in order, independently of the read-type calls.
Multiple ACCEPT calls, for the same listening stream socket, can be queued simultaneously. They are satisfied in order.

Multiple SELECT calls, referring to any combination of sockets, can be queued simultaneously on an IUCV path. TCP/IP checks all queued SELECT calls when an event occurs and responds to any that are satisfied.

**Note:** This applies only to programs that specified `apitype=3` in the initialization message.

Calls other than the read-type, write-type, ACCEPT, and SELECT calls, cannot be queued simultaneously for the same socket. For example, your program must wait for TCP/IP’s response to a write-type call before issuing a CLOSE call for the same socket.

### TCP/IP Response to an IUCV Request

TCP/IP’s response to your socket call is signaled by the Message Complete external interrupt. When the Message Complete external interrupt is received, if the IPAUDIT field shows no error, your program’s reply buffer has been filled. The IPBFLN2F field indicates how many bytes of the reply buffer were not used.

If the IPADDRJCT bit of the IPAUDIT field is set, then TCP/IP was unable to use IUCV REPLY to respond, and instead used IUCV REJECT. Your program issues the special LASTERRNO function (see “LASTERRNO” on page 148) to retrieve the return code and `errno` for the rejected call. TCP/IP’s use of IUCV REJECT does not necessarily mean the socket call failed.

The following `errno` values (shown in decimal) are seen only by a program using the IUCV socket interface.

<table>
<thead>
<tr>
<th><code>errno</code> Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>An unrecognized socket call constant was found in the high-order halfword of the Target Message Class.</td>
</tr>
<tr>
<td>1001</td>
<td>A request or reply length field is incorrect</td>
</tr>
<tr>
<td>1002</td>
<td>The socket number assigned by your program for ACCEPT, SOCKET, or TAKESOCKET is out of range.</td>
</tr>
<tr>
<td>1003</td>
<td>The socket number assigned by your program for ACCEPT, SOCKET, or TAKESOCKET is already in use.</td>
</tr>
<tr>
<td>1008</td>
<td>This request conflicts with a request already queued on the same socket (see “Overlapping Socket Requests” on page 122).</td>
</tr>
<tr>
<td>1009</td>
<td>The request was canceled by the CANCEL call (see “CANCEL” on page 127).</td>
</tr>
<tr>
<td>1010</td>
<td>Returned by the Offload function when a beginthread failure occurs.</td>
</tr>
</tbody>
</table>

### Cancelling a Socket Request

Your socket program can use the CANCEL call to cancel a previously issued socket call. Read-type calls, write-type calls, ACCEPT calls, and SELECT calls can be canceled using this function. See “CANCEL” on page 127 for more information about using the CANCEL call.

IUCV PURGE can also be used to cancel a call, but it does not stop TCP/IP processing the same way as the CANCEL call.
IUCV Sockets

Each IUCV SEND operation that completes with condition code zero is assigned a unique message identification number. This number is placed in the IUCV parameter list. To use the CANCEL or IUCV PURGE functions, your program must keep track of the message ID numbers assigned to each socket request.

IUCV Socket Call Syntax

Each of the IUCV Socket calls described includes the C language syntax for the call. IUCV SEND parameters and buffer contents are described using variable names from the C syntax. Call types are in capital letters. For example, the accept call is ACCEPT.

The parameter lists for some C language socket calls include a pointer to a data structure defined by a C structure. When using the IUCV socket interface, the contents of the data structure are passed in the send buffer, the reply buffer, or both. Table 21 shows the C structures used, and the corresponding assembler language syntax.

Table 21. C Structures in Assembler Language Format

<table>
<thead>
<tr>
<th>C Structure</th>
<th>Assembler Language Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct sockaddr_in {</td>
<td></td>
</tr>
<tr>
<td>short sin_family;</td>
<td>FAMILY DS H</td>
</tr>
<tr>
<td>ushort sin_port;</td>
<td>PORT DS H</td>
</tr>
<tr>
<td>struct in_addr sin_addr;</td>
<td>ADDR DS F</td>
</tr>
<tr>
<td>char sin_zero[8];</td>
<td>ZERO DC XLB‘00’</td>
</tr>
<tr>
<td>};</td>
<td></td>
</tr>
<tr>
<td>struct timeval {</td>
<td></td>
</tr>
<tr>
<td>long tv_sec;</td>
<td>TVSEC DS F</td>
</tr>
<tr>
<td>long tv_usec;</td>
<td>TVUSEC DS F</td>
</tr>
<tr>
<td>};</td>
<td></td>
</tr>
<tr>
<td>struct linger {</td>
<td></td>
</tr>
<tr>
<td>int l_onoff;</td>
<td>ONOFF DS F</td>
</tr>
<tr>
<td>int l_linger;</td>
<td>LINGER DS F</td>
</tr>
<tr>
<td>};</td>
<td></td>
</tr>
<tr>
<td>struct ifreq {</td>
<td></td>
</tr>
<tr>
<td>#define IFNAMSIZ 16</td>
<td>NAME DS CL16</td>
</tr>
<tr>
<td>char ifr_name[IFNAMSIZ];</td>
<td>ADDR.FAMILY DS H</td>
</tr>
<tr>
<td>union {</td>
<td>ADDR.PORT DS H</td>
</tr>
<tr>
<td>struct sockaddr ifru_addr;</td>
<td>ADDR.ADDR DS F</td>
</tr>
<tr>
<td>struct sockaddr ifru_dstaddr;</td>
<td>ADDR.ZERO DC XLB‘00’</td>
</tr>
<tr>
<td>struct sockaddr ifru_broadaddr;</td>
<td>ORG ADDR.FAMILY</td>
</tr>
<tr>
<td>short ifru_flags;</td>
<td>DST.FAMILY DS H</td>
</tr>
<tr>
<td>int ifru_metric;</td>
<td>DST.PORT DS H</td>
</tr>
<tr>
<td>caddr_t ifru_data;</td>
<td>DST.ADDR DS F</td>
</tr>
<tr>
<td>} ifr_ifru;</td>
<td>DST.ZERO DC XLB‘00’</td>
</tr>
<tr>
<td>}</td>
<td>ORG ADDR.FAMILY</td>
</tr>
<tr>
<td>}</td>
<td>BRD.FAMILY DS H</td>
</tr>
<tr>
<td>}</td>
<td>BRD.PORT DS H</td>
</tr>
<tr>
<td>}</td>
<td>BRD.ADDR DS F</td>
</tr>
<tr>
<td>}</td>
<td>BRD.ZERO DC XLB‘00’</td>
</tr>
<tr>
<td>}</td>
<td>ORG ADDR.FAMILY</td>
</tr>
<tr>
<td>}</td>
<td>FLAGS DS H</td>
</tr>
<tr>
<td>}</td>
<td>ORG ADDR.FAMILY</td>
</tr>
<tr>
<td>}</td>
<td>METRIC DS F</td>
</tr>
</tbody>
</table>

IUCV Sockets
### Table 21. C Structures in Assembler Language Format (continued)

<table>
<thead>
<tr>
<th>C Structure</th>
<th>Assembler Language Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct ifconf {</td>
<td></td>
</tr>
<tr>
<td>int ifc_len;</td>
<td>IFCLEN DS F</td>
</tr>
<tr>
<td>union {</td>
<td>IGNORED DS F</td>
</tr>
<tr>
<td>caddr_t ifcu_buf;</td>
<td></td>
</tr>
<tr>
<td>struct ifreq *ifcu_req;</td>
<td></td>
</tr>
<tr>
<td>} ifc_ifcu;</td>
<td></td>
</tr>
<tr>
<td>struct clientid {</td>
<td></td>
</tr>
<tr>
<td>int domain;</td>
<td>DOMAIN DS F</td>
</tr>
<tr>
<td>char name[8];</td>
<td>NAME DS CL8</td>
</tr>
<tr>
<td>char subtaskname[8];</td>
<td>SUBTASK DS CL8</td>
</tr>
<tr>
<td>char reserved[20];</td>
<td>RESERVED DC XL20'00'</td>
</tr>
<tr>
<td>};</td>
<td></td>
</tr>
</tbody>
</table>

### IUCV Socket Calls

This section provides the C language syntax, parameters, and other appropriate information for each IUCV socket call supported by TCP/IP. For information about C socket calls, see Chapter 1, “z/VM C Socket Application Programming Interface” on page 1.

**ACCEPT**

The ACCEPT call is issued when the server receives a connection request from a client. ACCEPT points to a socket that was created with a socket call and marked by a LISTEN call. ACCEPT can also be used as a blocking call. Concurrent server programs use the ACCEPT call to pass connection requests to child servers.

When issued, the ACCEPT call:

1. Accepts the first connection on a queue of pending connections
2. Creates a new socket with the same properties as the socket used in the call and returns the address of the client for use by subsequent server calls. The new socket cannot be used to accept new connections, but can be used by the calling program for its own connection. The original socket remains available to the calling program for more connection requests.
3. Returns the new socket descriptor to the calling program.

```c
ns = accept(s, addr, addrlen)
int ns, s;
struct sockaddr_in *addr;
int *addrlen;
```

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRGCLS</td>
<td>High-order halfword = 1</td>
</tr>
<tr>
<td></td>
<td>Low-order halfword = s</td>
</tr>
<tr>
<td>DATA</td>
<td>PRMMSG</td>
</tr>
<tr>
<td>PRMMSG</td>
<td>High-order fullword = 0</td>
</tr>
<tr>
<td></td>
<td>Low-order fullword = socket number for the new socket, chosen by your program, in the range 0 through maxsock. See Initializing the IUCV Connection on page 119.</td>
</tr>
</tbody>
</table>
**ACCEPT**

ANSLEN 24

ANSBUF Points to a buffer to contain the reply from TCP/IP:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ns</td>
<td>4</td>
<td>The new socket number assigned to this connection. A value of −1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When ns is −1, this field contains a reason code. <strong>Note:</strong> The rest of the reply buffer is filled only if the call was successful.</td>
</tr>
<tr>
<td>8</td>
<td>*addr</td>
<td>16</td>
<td>The remote address and port of the new socket. See Table 21 on page 124 for format.</td>
</tr>
</tbody>
</table>

**BIND**

In a typical server program, the BIND call follows a SOCKET call and completes the new socket creation process.

The BIND call can either specify the port or let the system choose the port. A listener program should always bind to the same well-known port so that clients know what socket address to use when issuing a CONNECT call.

```c
rc = bind(s, name, namelen)
int rc, s;
struct sockaddr_in *name;
int namelen;
```

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRGCLS</td>
<td>High-order halfword = 2</td>
</tr>
<tr>
<td></td>
<td>Low-order halfword = s</td>
</tr>
<tr>
<td>DATA</td>
<td>BUFFER</td>
</tr>
<tr>
<td>BUFLEN</td>
<td>16</td>
</tr>
<tr>
<td>BUFFER</td>
<td>Points to a buffer in the following format:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*name</td>
<td>16</td>
<td>The local address and port to which the socket is to be bound. See Table 21 on page 124 for format.</td>
</tr>
</tbody>
</table>

ANSLEN 8

ANSBUF Points to a buffer to contain the reply from TCP/IP:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rc</td>
<td>4</td>
<td>The return code from the BIND call. A return code of 0 indicates that the call was successful. A return code of −1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When the return code is −1, this field contains a reason code.</td>
</tr>
</tbody>
</table>
CANCEL

The CANCEL call is used to cancel a previously issued socket call. TCP/IP responds to the canceled call with a return code of −1 and an *errno* value of 1009.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
</table>
| TRGCLS  | High-order halfword = 42  
          | Low-order halfword = Low-order halfword of TRGCLS from call to be canceled. |
| DATA    | PRMMSG |
| PRMMSG  | High-order fullword = High-order halfword of TRGCLS from call to be canceled.  
          | Low-order fullword = IUCV message ID of call to be canceled. |
| ANSLEN  | 8 |
| ANSBUF  | Points to a buffer to contain the reply from TCP/IP: |

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rc</td>
<td>4</td>
<td>The return code from the CANCEL call. A return code of 0 indicates that the call was successful. A return code of −1 indicates that the function could not be completed and that <em>errno</em> contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When the return code is −1, this field contains a reason code. Possible <em>errno</em> values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>22</td>
</tr>
</tbody>
</table>

CLOSE

The CLOSE call shuts down the socket and frees the resources that are allocated to the socket.

\[
rc = close(s)  
\]

\[
\text{int } rc, s;  
\]

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
</table>
| TRGCLS  | High-order halfword = 3  
          | Low-order halfword = s |
| DATA    | PRMMSG |
| PRMMSG  | Binary zeros |
| ANSLEN  | 8 |
| ANSBUF  | Points to a buffer to contain the reply from TCP/IP: |

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rc</td>
<td>4</td>
<td>The return code from the CLOSE call. A return code of 0 indicates that the call was successful. A return code of −1 indicates that the function could not be completed and that <em>errno</em> contains a reason code.</td>
</tr>
</tbody>
</table>
**CONNECT**

The CONNECT call is used by a client to establish a connection between a local socket and a remote socket.

For stream sockets, the CONNECT call:
- Completes the binding process for a stream socket if a BIND call has not been previously issued.
- Attempts a connection to a remote socket. This connection must be completed before data can be transferred.

For datagram sockets, a CONNECT call is not essential, but you can use it to send messages without including the destination.

```c
rc = connect(s, name, namelen)
int rc, s;
struct sockaddr_in *name;
int namelen;
```

**Keyword** | **Value**
--- | ---
TRGCLS | High-order halfword = 4  
          Low-order halfword = s
DATA | BUFFER
BUFLEN | 16
BUFFER | Points to a buffer in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*name</td>
<td>16</td>
<td>The remote address and port to which the socket is to be connected. See Table 21 on page 124 for format.</td>
</tr>
</tbody>
</table>

ANSLEN | 8  
ANSBUF | The pointer to the buffer that is filled with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rc</td>
<td>4</td>
<td>The return code from the CONNECT call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When the return code is -1, this field contains a reason code.</td>
</tr>
</tbody>
</table>

**FCNTL**

The blocking mode for a socket can be queried or set using the FNDELAY flag described in the FCNTL call.
See “IOCTL” on page 134 for another way to control blocking for a socket.

```c
retval = fcntl(s, cmd, arg)
int retval;
int s, cmd, arg;
```

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRGCLS</td>
<td>High-order halfword = 5</td>
</tr>
<tr>
<td></td>
<td>Low-order halfword = s</td>
</tr>
<tr>
<td>DATA</td>
<td>PRMMSG</td>
</tr>
<tr>
<td>PRMMSG</td>
<td>High-order fullword:</td>
</tr>
<tr>
<td></td>
<td>F_GETFL (X ’00000003’)</td>
</tr>
<tr>
<td></td>
<td>F_SETFL (X ’00000004’)</td>
</tr>
</tbody>
</table>

The low-order fullword is used only for the F_SETFL command:

- Zero (X ’00000000’) Socket will block
- FNDELAY (X ’00000004’) Socket is non-blocking

| ANSLEN  | 8 |
| ANSBUF  | Points to a buffer that is filled with a reply in the format described as follows: |

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>retval</td>
<td>4</td>
<td>For F_SETFL, the return code. A value of zero indicates FNDELAY flag was set. For F_GETFL, the value of the FNDELAY flag. Zero means the socket will block. A value of FNDELAY (4) means the socket is non-blocking. A return code of −1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When the return code is -1, this field contains a reason code.</td>
</tr>
</tbody>
</table>

**GETCLIENTID**

The GETCLIENTID call returns the identifier by which the calling application is known to the TCPIP address space. The client ID structure that is returned is used in the GIVESOCKET and TAKESOCKET calls.

```c
rc = getclientid(domain, clientid)
int rc, domain;
struct clientid *clientid;
```

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRGCLS</td>
<td>High-order halfword = 30</td>
</tr>
<tr>
<td></td>
<td>Low-order halfword = 0</td>
</tr>
<tr>
<td>DATA</td>
<td>PRMMSG</td>
</tr>
<tr>
<td>PRMMSG</td>
<td>Binary zeros</td>
</tr>
<tr>
<td>ANSLEN</td>
<td>48</td>
</tr>
</tbody>
</table>
GETCLIENTID

ANSBUF  Points to the buffer that is filled with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rc</td>
<td>4</td>
<td>The return code from the GETCLIENTID call. A return code of 0 indicates</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>that the call was successful. A return code of -1 indicates that the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When the return code is -1, this field contains a reason code. Note:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The rest of the reply buffer is filled only if the call was successful.</td>
</tr>
<tr>
<td>8</td>
<td>*clientid</td>
<td>40</td>
<td>See Table 21 on page 124 for format.</td>
</tr>
</tbody>
</table>

Note: domain is not passed to TCP/IP. It is implicitly AF_INET.

GETHOSTID

The GETHOSTID call gets the unique 32-bit identifier for the current host. This value is the default home internet address.

    hostid = gethostid
    unsigned long hostid;

Keyword    Value
TRGCLS     High-order halfword = 7
           Low-order halfword = 0
DATA       PRMMSG
PRMMSG      Binary zeros
ANSLEN      8
ANSBUF      Points to the buffer that is filled with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hostid</td>
<td>4</td>
<td>The default home internet address.</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
<td>Your program should ignore this field.</td>
</tr>
</tbody>
</table>

GETHOSTNAME

The GETHOSTNAME call returns the name of the host processor on which the program is running. Up to namelen characters are copied into the name field.

    rc = gethostname(name, namelen)
    int rc;
    char *name;
    int namelen;

Keyword    Value
TRGCLS     High-order halfword = 8
           Low-order halfword = 0
GETHOSTNAME

<table>
<thead>
<tr>
<th>DATA</th>
<th>PRMMSG</th>
<th>Binary zeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSLEN</td>
<td>name + 8</td>
<td></td>
</tr>
<tr>
<td>ANSBUF</td>
<td>Points to the buffer that is filled with a reply in the following format:</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rc</td>
<td>4</td>
<td>The return code from the GETHOSTNAME call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When the return code is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.</td>
</tr>
<tr>
<td>8</td>
<td>*name</td>
<td>namelen</td>
<td>The host name, not null-terminated.</td>
</tr>
</tbody>
</table>

GETPEERNAME

The GETPEERNAME call returns the name of the remote socket to which the local socket is connected.

```c
rc = getpeername(s, name, namelen)
int rc, s;
struct sockaddr_in *name;
int *namelen;
```

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRGCLS</td>
<td>High-order halfword = 9</td>
</tr>
<tr>
<td></td>
<td>Low-order halfword = s</td>
</tr>
<tr>
<td>DATA</td>
<td>PRMMSG</td>
</tr>
<tr>
<td>PRMMSG</td>
<td>Binary zeros</td>
</tr>
<tr>
<td>ANSLEN</td>
<td>24</td>
</tr>
<tr>
<td>ANSBUF</td>
<td>Points to the buffer that is filled with a reply in the following format:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rc</td>
<td>4</td>
<td>The return code from the GETPEERNAME call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When the return code is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.</td>
</tr>
<tr>
<td>8</td>
<td>*name</td>
<td>16</td>
<td>The remote address and port to which the socket is connected. See Table 21 on page 124 for format.</td>
</tr>
</tbody>
</table>
GETSOCKNAME

GETSOCKNAME

The GETSOCKNAME call stores the name of the socket into the structure pointed to by the name parameter and returns the address to the socket that has been bound. If the socket is not bound to an address, the call returns with the family field completed and the rest of the structure set to zeros.

```c
rc = getsockopt(s, name, namelen)
int rc, s;
struct sockaddr_in *name;
int *namelen;
```

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRGCLS</td>
<td>High-order halfword = 10</td>
</tr>
<tr>
<td></td>
<td>Low-order halfword = s</td>
</tr>
<tr>
<td>DATA</td>
<td>PRMMSG</td>
</tr>
<tr>
<td>PRMMSG</td>
<td>Binary zeros</td>
</tr>
<tr>
<td>ANSLEN</td>
<td>24</td>
</tr>
<tr>
<td>ANSBUF</td>
<td>Points to the buffer that is filled with a reply in the following format:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rc</td>
<td>4</td>
<td>The return code from the GETSOCKNAME call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When the return code is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.</td>
</tr>
<tr>
<td>8</td>
<td>*name</td>
<td>16</td>
<td>The local address and port to which the socket is bound. See Table 21 on page 124 for format.</td>
</tr>
</tbody>
</table>

GETSOCKOPT

The GETSOCKOPT call returns the current setting of an option for a specific socket. Some of these options are under program control and can be changed using the SETSOCKOPT call.

```c
rc = getsockopt(s, level, optname, optval, &optlen)
int rc, s, level, optname, optlen;
char *optval;
```

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRGCLS</td>
<td>High-order halfword = 11</td>
</tr>
<tr>
<td></td>
<td>Low-order halfword = s</td>
</tr>
<tr>
<td>DATA</td>
<td>PRMMSG</td>
</tr>
<tr>
<td>PRMMSG</td>
<td>High-order fullword = level. Possible values are:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>C Symbol</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'FFFF'</td>
<td>SOL_SOCKET</td>
<td>Socket option</td>
</tr>
<tr>
<td>Value</td>
<td>C Symbol</td>
<td>Comments</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>X'0006'</td>
<td>IPPROTO_TCP</td>
<td>TCP protocol option</td>
</tr>
</tbody>
</table>

Low-order fullword = `optname`. Possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Option Name</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0001'</td>
<td>SO_DEBUG</td>
<td>Returns current setting.</td>
</tr>
<tr>
<td>X'0004'</td>
<td>SO_REUSEADDR</td>
<td>Returns current setting.</td>
</tr>
<tr>
<td>X'0008'</td>
<td>SO_KEEPALIVE</td>
<td>Returns current setting.</td>
</tr>
<tr>
<td>X'0010'</td>
<td>SO_DONTROUTE</td>
<td>Returns current setting.</td>
</tr>
<tr>
<td>X'0020'</td>
<td>SO_BROADCAST</td>
<td>Returns current setting.</td>
</tr>
<tr>
<td>X'0080'</td>
<td>SO_LINGER</td>
<td>Returns current setting in a C language <code>struct linger</code>. See Table 21 on page 124 for the assembler language equivalent.</td>
</tr>
<tr>
<td>X'0100'</td>
<td>SO_OOBINLINE</td>
<td>Returns current setting.</td>
</tr>
<tr>
<td>X'1001'</td>
<td>SO_SNDBUF</td>
<td>Returns the size of the TCP/IP send buffer.</td>
</tr>
<tr>
<td>X'1007'</td>
<td>SO_ERROR</td>
<td>Returns any pending error code and clears any error status conditions.</td>
</tr>
<tr>
<td>X'1008'</td>
<td>SO_TYPE</td>
<td>Socket type is returned:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stream</td>
</tr>
<tr>
<td>2</td>
<td>Datagram</td>
</tr>
<tr>
<td>3</td>
<td>Raw</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>C Symbol</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0001'</td>
<td>TCP_NODELAY</td>
<td>Returns current setting.</td>
</tr>
</tbody>
</table>

Note: This option applies only to `level=IPPROTO_TCP`

**ANSLEN**

16 for option `SO_LINGER`, 12 for all other options

**ANSBUF**

Points to a buffer to contain the reply from TCP/IP:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>rc</code></td>
<td>4</td>
<td>The return code from the GETSOCKOPT call. A return code of 0 indicates that the call was successful. A return code of −1 indicates that the function could not be completed and that <code>errno</code> contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td><code>errno</code></td>
<td>4</td>
<td>When the return code is −1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.</td>
</tr>
<tr>
<td>8</td>
<td><code>optval</code></td>
<td>4 or 8</td>
<td>The value of the requested option. If the option <code>SO_LINGER</code> was requested, 8 bytes are returned. For all other options, 4 bytes are returned.</td>
</tr>
</tbody>
</table>

**GIVESOCKET**

The GIVESOCKET call makes the socket available for a TAKESOCKET call issued by another program. The GIVESOCKET call can specify any connected stream socket. Typically, the GIVESOCKET call is issued by a concurrent server program that creates sockets to be passed to a child server.
The GIVESOCKET sequence is:

- To pass a socket, the concurrent server first calls GIVESOCKET. If the optional parameters, name of the child server’s virtual machine and subtask ID are specified in the GIVESOCKET call, only a child with a matching virtual machine and subtask ID can take the socket.
- The concurrent server then starts the child server and passes it the socket descriptor and concurrent server’s ID that were obtained from earlier SOCKET and GETCLIENTID calls.
- The child server calls TAKESOCKET, with the concurrent server’s ID and socket descriptor.
- The concurrent server issues the select call to test the socket for the exception condition, TAKESOCKET completion.
- When the TAKESOCKET has successfully completed, the concurrent server issues the CLOSE call to free the socket.

```
rc = givesocket(s, clientid)
int rc, s;
struct clientid *clientid;
```

### Keyword Value

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRGCLS</td>
<td>High-order halfword = 31</td>
</tr>
<tr>
<td></td>
<td>Low-order halfword = s</td>
</tr>
<tr>
<td>DATA</td>
<td>BUFFER</td>
</tr>
<tr>
<td>BUFLEN</td>
<td>40</td>
</tr>
<tr>
<td>BUFFER</td>
<td>Points to the message in the following format:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*clientid</td>
<td>40</td>
<td>See Table 21 on page 124 for format.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSLEN</td>
<td>8</td>
</tr>
<tr>
<td>ANSBUF</td>
<td>The pointer to the buffer that is filled with a reply in the following format:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rc</td>
<td>4</td>
<td>The return code from the GIVESOCKET call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When the return code is -1, this field contains a reason code.</td>
</tr>
</tbody>
</table>

### IOCTL

The IOCTL call is used to control certain operating characteristics for a socket.

Before you issue an IOCTL call, you must load a value representing the characteristic that you want to control into the cmd field.
Keyword | Value
--- | ---
TRGCLS | High-order halfword = 12
Low-order halfword = s
DATA | BUFFER
BUFSIZE | Request arg length + 4
BUFFER | The pointer to the message in the format described in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*cmd</td>
<td>4</td>
<td>The type of request. See Table 22 for values.</td>
</tr>
<tr>
<td>4</td>
<td>*arg</td>
<td>See Table 22</td>
<td>The request data, if any.</td>
</tr>
</tbody>
</table>

ANSLEN | Reply arg length + 8
ANSBUF | The pointer to the buffer that is filled with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rc</td>
<td>4</td>
<td>A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When the return code is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.</td>
</tr>
<tr>
<td>8</td>
<td>*arg</td>
<td>See Table 22</td>
<td>The response data, if any.</td>
</tr>
</tbody>
</table>

Table 22. Values for cmd Argument in ioctl Call

<table>
<thead>
<tr>
<th>C Symbol</th>
<th>Value</th>
<th>Request arg Length</th>
<th>Request arg Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIONBIO</td>
<td>X'8004A77E'</td>
<td>4</td>
<td>0</td>
<td>Request arg data is a fullword integer.</td>
</tr>
<tr>
<td>FIONREAD</td>
<td>X'4004A77F'</td>
<td>0</td>
<td>4</td>
<td>Reply arg data is a fullword integer.</td>
</tr>
<tr>
<td>SIOCADDRT</td>
<td>X'8030A70A'</td>
<td>48</td>
<td>0</td>
<td>For IBM use only.</td>
</tr>
<tr>
<td>SIOCATMARK</td>
<td>X'4004A707'</td>
<td>0</td>
<td>4</td>
<td>Reply arg data is a fullword integer.</td>
</tr>
<tr>
<td>SIOCDELRT</td>
<td>X'8030A70B'</td>
<td>48</td>
<td>0</td>
<td>For IBM use only.</td>
</tr>
<tr>
<td>SIOCGFADDR</td>
<td>X'C020A70D'</td>
<td>32</td>
<td>32</td>
<td>arg data is the C language struct ifreq. See Table 21 on page 124 for the assembler language equivalent.</td>
</tr>
</tbody>
</table>

\[
\text{rc} = \text{ioctl(s, cmd, arg)} \\
\text{int rc, s;} \\
\text{unsigned long cmd;} \\
\text{char *arg;}
\]
IOCTL

Table 22. Values for cmd Argument in ioctl Call (continued)

<table>
<thead>
<tr>
<th>C Symbol</th>
<th>Value</th>
<th>Request arg Length</th>
<th>Reply arg Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIOCGIFBRDADDR</td>
<td>X'0C020A712'</td>
<td>32</td>
<td>32</td>
<td>arg data is the C language struct ifreq. See Table 21 on page 124 for the assembler language equivalent.</td>
</tr>
<tr>
<td>SIOCGIFCONF</td>
<td>X'0C008A714'</td>
<td>8</td>
<td>*</td>
<td>Request arg data is the C-language struct ifconf. See Table 21 on page 124 for the assembler language equivalent. Your program sets ifc_len to the reply length. The other field is ignored. Response arg data is an array of C language struct ifreq structures, one for each defined interface. Note: * = the maximum number of interfaces multiplied by 32.</td>
</tr>
<tr>
<td>SIOCGIFDSTADDR</td>
<td>X'0C020A70F'</td>
<td>32</td>
<td>32</td>
<td>arg data is the C language struct ifreq. See Table 21 on page 124 for the assembler language equivalent.</td>
</tr>
<tr>
<td>SIOCGIFFLAGS</td>
<td>X'0C020A711'</td>
<td>32</td>
<td>32</td>
<td>arg data is the C language struct ifreq. See Table 21 on page 124 for the assembler language equivalent.</td>
</tr>
<tr>
<td>SIOCGIFMETRIC</td>
<td>X'0C020A717'</td>
<td>32</td>
<td>32</td>
<td>For IBM use only.</td>
</tr>
<tr>
<td>SIOCGIFNETMASK</td>
<td>X'0C020A715'</td>
<td>32</td>
<td>32</td>
<td>arg data is the C language struct ifreq. See Table 21 on page 124 for the assembler language equivalent.</td>
</tr>
<tr>
<td>SIOCSIIFDSTADDR</td>
<td>X'8020A70E'</td>
<td>32</td>
<td>0</td>
<td>For IBM use only.</td>
</tr>
<tr>
<td>SIOCSIFFLAGS</td>
<td>X'8020A710'</td>
<td>32</td>
<td>0</td>
<td>For IBM use only.</td>
</tr>
<tr>
<td>SIOCSIIFMETRIC</td>
<td>X'8020A718'</td>
<td>32</td>
<td>0</td>
<td>For IBM use only.</td>
</tr>
<tr>
<td>SIOCSIBMOPT</td>
<td>X'0C048D900'</td>
<td>72</td>
<td>*</td>
<td>For IBM use only.</td>
</tr>
<tr>
<td>SIOCSIBMOPT</td>
<td>X'8048D900'</td>
<td>*</td>
<td>0</td>
<td>For IBM use only.</td>
</tr>
</tbody>
</table>

LISTEN

The LISTEN call:
- Completes the bind, if BIND has not already been called for the socket.
- Creates a connection-request queue of a specified length for incoming connection requests.

The LISTEN call is typically used by a concurrent server to receive connection requests from clients. When a connection request is received, a new socket is created by a later ACCEPT call. The original socket continues to listen for additional connection requests. The LISTEN call converts an active socket to a passive socket and configures it to accept connection requests from client programs. If a socket is passive it cannot initiate connection requests.
rc = listen(s, backlog)
int rc, s, backlog;

Keyword Value
TRGCLS	High-order halfword = 13
Low-order halfword = s
DATA	PRMMSG
PRMMSG	High-order fullword = 0
Low-order fullword = backlog
ANSLEN	8
ANSBUF	Points to the buffer that is filled with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rc</td>
<td>4</td>
<td>The return code from the LISTEN call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When the return code is -1, this field contains a reason code.</td>
</tr>
</tbody>
</table>

MAXDESC
Your program specifies the maximum number of AF_INET sockets in the initialization message. For more information about the initialization message, see “Initializing the IUCV Connection” on page 119.

READ, READV
From the point of view of TCP/IP, the READ and READV calls are identical. From the point of view of the application, they differ only in that the buffer for READ is contiguous in storage, while the buffer for READV might not be contiguous.

Your program, utilizing the direct IUCV socket interface, can use the ANSLIST=YES parameter on IUCV SEND to specify a noncontiguous READ buffer. You can choose to use ANSLIST=YES even if your READ buffer is contiguous, so that the reply area for cc and errno need not adjoin the READ buffer in storage.

This section does not distinguish between READ and READV. IUCV usage is described in terms of variable names from the C language syntax of READ.

cc = read(s, buf, len)
int cc, s;
char *buf;
int len;

Keyword Value
TRGCLS	High-order halfword = 14
Low-order halfword = s
DATA	PRMMSG
PRMMSG	Binary zeros
**ANSLEN**  
\( len + 24 \)

**ANSBUF**  
Points to the buffer that is filled with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( cc )</td>
<td>4</td>
<td>The number of bytes read. A value of zero means the partner has closed the connection. A value of (-1) indicates that the function could not be completed and that ( errno ) contains a reason code.</td>
</tr>
</tbody>
</table>
| 4      | \( errno \) | 4      | When \( cc \) is \(-1\), this field contains a reason code.  
**Note:** The rest of the reply buffer is filled only if the call was successful. |
| 8      |        | 16     | Your program should ignore this field.       |
| 24     | \(*buf\) | \( len \) | The received data.                           |

**RECV, RECVFROM, RECVMSG**

From the point of view of TCP/IP, the RECV, RECVFROM, and RECVMSG calls are identical.

From the point of view of the application, RECVFROM differs from RECV in that RECVFROM additionally provides the source address of the message. Your program, using the direct IUCV socket interface, must provide space to receive the source address of the message, even if the source address is not required.

From the point of view of the application, RECVMSG differs from RECVFROM in that RECVMSG additionally allows the buffer to be in noncontiguous storage. Your program, utilizing the direct IUCV socket interface, can use the ANSLIST=YES parameter on IUCV SEND to specify a noncontinuous read buffer. You can choose to use ANSLIST=YES even if your read buffer is contiguous, so that the reply area for \( cc \) and \( errno \), and the space to receive the source address of the message, need not adjoin the read buffer in storage.

\[
cc = \text{recvfrom}(s, buf, len, flags, from, fromlen)
\]

```c
int cc, s;
char *buf;
int len, flags;
struct sockaddr_in *from;
int *fromlen;
```

**Keyword**  
**Value**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRGCLS</td>
<td>High-order halfword = 16</td>
</tr>
<tr>
<td></td>
<td>Low-order halfword = ( s )</td>
</tr>
<tr>
<td>DATA</td>
<td>PRMMSG</td>
</tr>
<tr>
<td>PRMMSG</td>
<td>High-order fullword = 0.</td>
</tr>
<tr>
<td></td>
<td>Low-order fullword = ( flags ):</td>
</tr>
<tr>
<td></td>
<td>MSG_OOB (X'00000001')</td>
</tr>
<tr>
<td></td>
<td>MSG_PEEK (X'00000002')</td>
</tr>
<tr>
<td>ANSLEN</td>
<td>( len + 24 )</td>
</tr>
</tbody>
</table>
ANSBUF  Points to the buffer that is filled with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>cc</td>
<td>4</td>
<td>The number of bytes read. A value of zero indicates that communication is closed. A value of −1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When cc is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.</td>
</tr>
<tr>
<td>8</td>
<td>*from</td>
<td>16</td>
<td>The source address and port of the message. See Table 21 on page 124 for format.</td>
</tr>
<tr>
<td>24</td>
<td>*buf</td>
<td>len</td>
<td>The received data.</td>
</tr>
</tbody>
</table>

SELECT, SELECTEX

From the point of view of the TCP/IP, the SELECT and SELECTEX calls are identical. From the point of view of the application, they differ in that return from SELECTEX can be triggered by the posting of an ECB as well as the selection of a descriptor or a time-out.

Your program cannot initiate other activity on an IUCV path until TCP/IP responds to the SELECT call. However, your program can cancel a SELECT call before TCP/IP responds by issuing IUCV PURGE. A successful IUCV PURGE can be followed immediately by an IUCV SEND initiating another socket call.

Multiple SELECT calls, referring to any combination of sockets, can be queued simultaneously on an IUCV path.

Note: IUCV PURGE cannot be used by a multiple-request program to cancel a SELECT call. See “Issuing Socket Calls” on page 122 for more information about multiple-request socket programs.

```c
nfound = select(nfds, readfds, writefds, exceptfds, timeout)
int nfound, nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;
```

Descriptor Sets

A descriptor set is an array of fullwords. The following is the required array size in integer arithmetic:

```
number_of_fullwords = (nfds + 31) / 32
number_of_bytes = number_of_fullwords * 4
```

**DESCRIPTOR SET, FD_CLR, FD_ISSET Calls**

The following describes how to perform the function of these C language calls, which set, clear, and test the bit in the specified descriptor set corresponding to the specified descriptor number.

You can compute the offset of the fullword containing the bit (integer arithmetic) as follows:

```
offset = (descriptor_number / 32) * 4
```
Compute a mask to locate the bit within the fullword by:

\[
\text{bitmask} = X'00000001' \ll (\text{descriptor\_number} \mod 32)
\]

(“\ll” is the left-shift operator).

Then use the mask, or a complemented copy of the mask, to set, clear, or test the bit, as appropriate.

The IUCV SEND parameters particular to select are:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRGCLS</td>
<td>High-order halfword = 19 Low-order halfword = descriptor set size in bytes (fdsize). See “Descriptor Sets” on page 139.</td>
</tr>
<tr>
<td>DATA</td>
<td>BUFFER</td>
</tr>
<tr>
<td>BUFLEN</td>
<td>(3*fdsize)+28</td>
</tr>
<tr>
<td>BUFFER</td>
<td>The pointer to the message in the following format:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Number of file descriptors</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>nfds</td>
<td>4</td>
<td></td>
<td>To improve processing efficiency, nfds should be no greater than one plus the largest descriptor number actually in use.</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
<td></td>
<td>Set this field to zero if you want select to block. Otherwise set this field to any nonzero value and fill in *timeval.</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>4</td>
<td></td>
<td>If any descriptor bits are set in readfds, your program sets this field to a nonzero value. If no descriptor bits are set in readfds, your program can set this field to zero, to improve processing efficiency.</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>4</td>
<td></td>
<td>If any descriptor bits are set in writefds, your program sets this field to a nonzero value. If no descriptor bits are set in writefds, your program can set this field to zero, to improve processing efficiency.</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>4</td>
<td></td>
<td>If any descriptor bits are set in exceptfds, your program sets this field to a nonzero value. If no descriptor bits are set in exceptfds your program can set this field to zero, to improve processing efficiency.</td>
</tr>
<tr>
<td>20</td>
<td>*timeval</td>
<td>8</td>
<td></td>
<td>See Table 21 on page 124 for format. If field at offset 4 is zero, then set this field to binary zeros.</td>
</tr>
<tr>
<td>28</td>
<td>*readfds</td>
<td>fdsize</td>
<td>If field at offset 8 is zero, then this field is not used.</td>
<td></td>
</tr>
<tr>
<td>28 + fdsize</td>
<td>*writefds</td>
<td>fdsize</td>
<td>If field at offset 12 is zero, then this field is not used.</td>
<td></td>
</tr>
<tr>
<td>28 + (2*fdsize)</td>
<td>*exceptfds</td>
<td>fdsize</td>
<td>If field at offset 16 is zero, then this field is not used.</td>
<td></td>
</tr>
</tbody>
</table>

ANSLEN   | (3*fdsize)+16                        |
| ANSBUF   | The pointer to the buffer that is filled in with a reply in the following format: |
The SEND call sends datagrams on a specified connected socket.

The `flags` field allows you to:
- Send out-of-band data, for example, interrupts, aborts, and data marked urgent.
- Suppress use of local routing tables. This implies that the caller takes control of routing, writing network software.

For datagram sockets, the entire datagram is sent if the datagram fits into the buffer. Excess data is discarded.

For stream sockets, data is processed as streams of information with no boundaries separating data the data. For example, if a program is required to send 1000 bytes, each call to this function can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in `errno`. Therefore, programs using stream sockets should place this call in a loop, reissuing the call until all data has been sent.

```c
cc = send(s, msg, len, flags)
int cc, s;
char *msg;
int len, flags;
```

**Keyword** | **Value**
---|---
TRGCLS | High-order halfword = 20
| Low-order halfword = s
BUFLEN | len + 20
DATA | BUFFER
BUFFER | The pointer to the message in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
</table>
| 0 | flags | 4 | MSG_OOB (X'00000001')
| | | | MSG_DONTROUTE (X'00000004')
| 4 | 16 | Your program should set this field to binary zeros.
**SEND**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>*msg</td>
<td>len</td>
<td>The data to be sent.</td>
</tr>
</tbody>
</table>

**ANSLEN**  8  
**ANSBUF**  The pointer to the buffer that is filled in with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>cc</td>
<td>4</td>
<td>The number of bytes sent. A value of −1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When cc is -1, this field contains a reason code.</td>
</tr>
</tbody>
</table>

**SENMDSG**

From the point of view of TCP/IP, the SENDMSG call with a null msg->msg_name parameter is identical to the SEND call. Similarly, the SENDMSG call with a non-null msg->msg_name parameter is identical to the SENDTO call.

From the point of view of the application, SENDMSG differs from SEND and SENDTO in that SENDMSG additionally allows the write buffer to be in noncontiguous storage.

Your program, using the direct IUCV socket interface can use the BUFLIST=YES parameter on IUCV SEND to specify a noncontiguous write buffer. You can choose to use BUFLIST=YES even if your write buffer is contiguous, so that the fields preceding the write data in the request format need not adjoin the write data in storage.

See ["SEND" on page 141](#) and ["SENDTO"](##) for more information.

**SENDTO**

SENDTO is similar to SEND, except that it includes the destination address parameter. You can use the destination address on the SENDTO call to send datagrams on a UDP socket that is connected or not connected.

Use the flags parameter to:

- Send out-of-band data such as, interrupts, aborts, and data marked as urgent.
- Suppress the local routing tables. This implies that the caller takes control of routing, which requires writing network software.

For datagram sockets, the SENDTO call sends the entire datagram if the datagram fits into the buffer.

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if a program is required to send 1000 bytes, each SENDTO call can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in errno. Therefore, programs using stream sockets should place SENDTO in a loop that repeats the call until all data has been sent.
cc = sendto(s, msg, len, flags, to, tolen)
int cc, s;
char *msg;
int len, flags;
struct sockaddr_in *to;
int tolen;

**Keyword** | **Value**
--- | ---
TRGCLS | High-order halfword = 22
Low-order halfword = s
DATA | BUFFER
BUFLEN | len + 20.
BUFFER | The pointer to the message in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
</table>
| 0      | flags | 4      | MSG_OOB (X'00000001')
|        |       |        | MSG_DONTROUTE (X'00000004') |
| 4      | *to   | 16     | See Table 21 on page 124 for format. |
| 20     | *msg  | len    | The data to be sent. |

ANSLEN | 8
ANSBUF | The pointer to the buffer that is filled with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>cc</td>
<td>4</td>
<td>The number of bytes sent. A value of -1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When cc is -1, this field contains a reason code.</td>
</tr>
</tbody>
</table>

**SETSOCKOPT**

The SETSOCKOPT call sets the options associated with a socket.

rc = setsockopt(s, level, optname, optval, optlen)
int rc, s, level, optname;
char *optval;
int optlen;

**Keyword** | **Value**
--- | ---
TRGCLS | High-order halfword = 23
Low-order halfword = s
DATA | BUFFER
BUFLEN | 16 for option SO_LINGER, 12 for all other options
BUFFER | Points to a buffer in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
</table>
| 0      | level| 4      | X'FFFF' - SOL_SOCKET - Socket option
X'0006' - IPPROTO_TCP - TCP protocol option |
SETSOCKOPT

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>optname</td>
<td>4</td>
<td>Option to set. See Table 23 for values.</td>
</tr>
<tr>
<td>8</td>
<td>optval</td>
<td>4 or 8</td>
<td>The value of the specified option. If the option SO_LINGER is specified, 8 bytes are needed. For all other options, 4 bytes are needed.</td>
</tr>
</tbody>
</table>

Table 23. Option name values for SETSOCKOPT

<table>
<thead>
<tr>
<th>Value</th>
<th>Option Name</th>
<th>Option Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0001'</td>
<td>SO_DEBUG</td>
<td>On (1) or Off (0). Option may be set, but has no effect.</td>
</tr>
<tr>
<td>X'0004'</td>
<td>SO_REUSEADDR</td>
<td>Yes (1) or No (0).</td>
</tr>
<tr>
<td>X'0008'</td>
<td>SO_KEEPALIVE</td>
<td>Yes (1) or No (0).</td>
</tr>
<tr>
<td>X'0010'</td>
<td>SO_DONTROUTE</td>
<td>Yes (1) or No (0). Option may be set, but has no effect. Use MSG_DONTROUTE on write-type calls instead.</td>
</tr>
<tr>
<td>X'0020'</td>
<td>SO_BROADCAST</td>
<td>Yes (1) or No (0).</td>
</tr>
<tr>
<td>X'0080'</td>
<td>SO_LINGER</td>
<td>Value is a C language struct linger. See Table 21 on page 124 for the assembler language equivalent.</td>
</tr>
<tr>
<td>X'0100'</td>
<td>SO_OOBINLINE</td>
<td>Yes (1) or No (0). Note: The following option applies only to level=IPPROTO_TCP</td>
</tr>
<tr>
<td>X'0001'</td>
<td>TCP_NODELAY</td>
<td>Yes (1) or No (0).</td>
</tr>
</tbody>
</table>

ANSLEN  8

ANSBUF  Points to a buffer to contain the reply from TCP/IP:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rc</td>
<td>4</td>
<td>The return code from the SETSOCKOPT call. A return code of 0 indicates that the call was successful. A return code of −1 indicates that the function could not be completed and that errno contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>errno</td>
<td>4</td>
<td>When the return code is −1, this field contains a reason code.</td>
</tr>
</tbody>
</table>

SHUTDOWN

The normal way to terminate a network connection is to issue the CLOSE call which attempts to complete all outstanding data transmission requests prior to breaking the connection. The SHUTDOWN call can be used to close one-way traffic while completing data transfer in the other direction. The how parameter determines the direction of the traffic to shutdown.

A client program can use the SHUTDOWN call to reuse a given socket with a different connection.

rc = shutdown(s, how)
int rc, s, how;

Keyword     Value
TRGCLS      High-order halfword = 24
SHUTDOWN

Low-order halfword = \( s \)

**DATA**

PRMMSG

High-order fullword = 0

Low-order fullword = \( how \):

0 = receive
1 = send
2 = both

**ANSLEN** 8

**ANSSBUF** Points to the buffer that is filled with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( rc )</td>
<td>4</td>
<td>The return code from the SHUTDOWN call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that ( errno ) contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>( errno )</td>
<td>4</td>
<td>When the return code is -1, this field contains a reason code.</td>
</tr>
</tbody>
</table>

SOCKET

The SOCKET call creates an endpoint for communication and returns a socket descriptor representing the endpoint. Different types of sockets provide different communication services.

\[
s = \text{socket}(\text{domain}, \text{type}, \text{protocol})
\]

**Keyword** | **Value**
--- | ---
TRGCLS | High-order halfword = 25
 | Low-order halfword = 0

**DATA** BUFFER

**BUFLEN** 16

**BUFFER** The pointer to the message in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \text{domain} )</td>
<td>4</td>
<td>The only valid value is AF_INET (X'00000002')</td>
</tr>
<tr>
<td>4</td>
<td>( \text{type} )</td>
<td>4</td>
<td>Fullword integer:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SOCK_STREAM (X'00000001')</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SOCK_DGRAM (X'00000002')</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SOCK_RAW (X'00000003')</td>
</tr>
<tr>
<td>8</td>
<td>( \text{protocol} )</td>
<td>4</td>
<td>Fullword integer:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IPPROTO_ICMP (X'00000001')</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IPPROTO_TCP (X'00000006')</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IPPROTO_UDP (X'00000011')</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IPPROTO_RAW (X'000000FF')</td>
</tr>
</tbody>
</table>
Socket number for the new socket, chosen by your program, in the range 0 through `maxsock`. See "Initializing the IUCV Connection" on page 119.

Offset Name Length Comments
12 s 4 Socket number for the new socket, chosen by your program, in the range 0 through `maxsock`. See "Initializing the IUCV Connection" on page 119.

ANSLEN 8

ANSBUF Points to the buffer that is filled with a reply in the following format:

Offset Name Length Comments
0 s 4 The socket number assigned to this communications end point. A value of −1 indicates that the function could not be completed and that `errno` contains a reason code.
4 errno 4 When `s` is -1, this field contains a reason code.

Takesocket

The TAKESOCKET call acquires a socket from another program and creates a new socket. Typically, a child server issues this call using client ID and socket descriptor data which it obtained from the concurrent server. When TAKESOCKET is issued, a new socket descriptor is returned in `errno`. You should use this new socket descriptor in later calls such as GETSOCKOPT, which require the `s (socket descriptor)` parameter.

Note: Both concurrent servers and iterative servers are used by this interface. An iterative server handles one client at a time. A concurrent server receives connection requests from multiple clients and creates child servers that process the client requests. When a child server is created, the concurrent server gets a new socket, passes the new socket to the child server, and dissociates itself from the connection. The TCP/IP Listener program is an example of a concurrent server.

```
s = takesocket(clientid, hisdesc)
int s;
struct clientid *clientid;
int hisdesc;
```

Keyword Value
TRGCLS High-order halfword = 32
Low-order halfword = 0
DATA BUFFER
BUFLen 48
BUFFER The pointer to the message in the following format:

Offset Name Length Comments
0 *clientid 40 See Table 21 on page 124 for format.
40 hisdesc 4
44 s 4 Socket number for the new socket, chosen by your program, in the range 0 through `maxsock`. See "Initializing the IUCV Connection" on page 119.
ANSLEN 8
ANSBUF The pointer to the buffer that is filled with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$s$</td>
<td>4</td>
<td>The socket number assigned to this communications end point. A value of −1 indicates that the function could not be completed and that $errno$ contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>$errno$</td>
<td>4</td>
<td>When $s$ is -1, this field contains a reason code.</td>
</tr>
</tbody>
</table>

WRITE, WRITEV

From the point of view of TCP/IP, the WRITE and WRITEV calls are identical. From the point of view of the application, WRITEV differs from WRITE in that WRITEV additionally allows the write buffer to be in noncontiguous storage.

Your program, using the direct IUCV socket interface, can use the BUFLIST=YES parameter on IUCV SEND to specify a noncontiguous write buffer. You can choose to use BUFLIST=YES even if your write buffer is contiguous, so that the 20-byte prefix need not adjoin the write buffer in storage.

This section does not distinguish between WRITE and WRITEV. IUCV usage is described in terms of variable names from the C language syntax of WRITE.

```c
cc = write(s, buf, len)
int cc, s;
char *buf;
int len;
```

Keyword  Value
---  ----
TRGCLS  High-order halfword = 26
Low-order halfword = $s$
DATA BUFFER
BUFLEN $len + 20$
BUFFER The pointer to the message in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$*buf$</td>
<td>$len$</td>
<td>The data to be sent.</td>
</tr>
</tbody>
</table>

ANSLEN 8
ANSBUF Points to the buffer that is filled with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$cc$</td>
<td>4</td>
<td>The number of bytes sent. A value of −1 indicates that the function could not be completed and that $errno$ contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td>$errno$</td>
<td>4</td>
<td>When $cc$ is -1, this field contains a reason code.</td>
</tr>
</tbody>
</table>
As explained in [TCP/IP Response to an IUCV Request](#) on page 123, if TCP/IP uses IUCV REJECT to respond to a socket request, your program uses the LASTERRNO special request to retrieve the return code and `errno`.

**Keyword** | **Value**
--- | ---
TRGCLS | High-order halfword = 29  
| Low-order halfword = 0
DATA | PRMMSG
PRMMSG | Binary zeros
ANSLEN | 8
ANSBUF | Points to the buffer that is filled in with a reply in the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>rc</code></td>
<td>4</td>
<td>The return code from the last rejected call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <code>errno</code> contains a reason code.</td>
</tr>
<tr>
<td>4</td>
<td><code>errno</code></td>
<td>4</td>
<td>When the return code is -1, this field contains a reason code.</td>
</tr>
</tbody>
</table>
Chapter 5. Remote Procedure Calls

This chapter 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.

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.

For sample RPC client, server, and raw data stream programs, see “Sample RPC Programs” on page 195. Figure 28 on page 150 and Figure 29 on page 151 provide an overview of the high-level RPC client and server processes from initialization through cleanup.
RPCs

Figure 28. Remote Procedure Call (Client)
Portmapper

Portmapper is the software that supplies client programs with the port numbers of server programs.

You can communicate between different computer operating systems when messages are directed to port numbers rather than to targeted remote programs. Clients contact server programs by sending messages to the port numbers where receiving processes receive the message. Because you make requests to the port...
number of a server rather than directly to a server program, client programs need a way to find the port number of the server programs they wish to call. Portmapper standardizes the way clients locate the port number of the server programs supported on a network.

Portmapper resides on all hosts on well-known port 111.

The port-to-program information maintained by Portmapper is called the portmap. Clients ask Portmapper about entries for servers on the network. Servers contact Portmapper to add or update entries to the portmap.

Contacting Portmapper
To find the port of a remote program, the client sends an RPC to well-known port 111 of the server’s host. If Portmapper has a portmap entry for the remote program, Portmapper provides the port number in a return RPC. The client then requests the remote program by sending an RPC to the port number provided by Portmapper.

Clients can save port numbers of recently called remote programs to avoid having to contact Portmapper for each request to a server.

To see all the servers currently registered with Portmapper, use the RPCINFO command in the following manner:

```
RPCINFO -p host_name
```

For more information about Portmapper and RPCINFO, see [TCP/IP User's Guide](#) and [TCP/IP Planning and Customization](#).

Target Assistance
Portmapper offers a program to assist clients in contacting server programs. If the client sends Portmapper an RPC with the target program number, version number, procedure number, and arguments, Portmapper searches the portmap for an entry, and passes the client’s message to the server. When the target server returns the information to Portmapper, the information is passed to the client, along with the port number of the remote program. The client can then contact the server directly.

RPCGEN Command
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. For RPCGEN to work correctly you must have access to the CC EXEC that is a part of the C compiler and have accessed the TCPIP Client-code minidisk (usually the TCPMAINT 592).

RPCGEN `in file` is normally used when you want to generate all four of the following output files. For example, if the `in file` 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

**Note:** 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.
For additional information about the RPCGEN command, see the Sun Microsystems publication, *Network Programming*.

Operand Description

- **-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 call-back routines and for writing a main routine for initialization.

- **-s TCP|UDP**
  Compiles into server-side stubs using the given transport. The TCP option supports the TCP transport protocol. The UDP option supports the UDP transport protocol.

- **-o outfile**
  Specifies the name of the output file. If none is specified, standard output is used for -c, -h, -l, -m, and -s modes.

- **infile**
  Specifies the name of the input file written in the RPC language. *infile* should be a variable record format file (RECFM V).

**enum clnt_stat Structure**

The enumerated set clnt_stat structure is defined in the CLNT.H header file.

RPCs frequently return the enumerated set clnt_stat information. The following is the format and a description of the enumerated set clnt_stat structure:

```c
enum clnt_stat {
    RPC_SUCCESS=0,          /* call succeeded */
    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 */

    RPC_VERSMISMATCH=6,     /* RPC versions not compatible */
    RPC_AUTHERROR=7,        /* authentication error */
    RPC_PROGUNAVAIL=8,      /* program not available */
    RPC_PROGVERSDECLINCONS=9, /* program version mismatched */
    RPC_PROCUINAVAIL=10,    /* procedure unavailable */
    RPC_CANTDECODEARGS=11,  /* decode arguments error */
    RPC_SYSTEMERROR=12,     /* generic "other problem" */

    /*
    */
};
```
RPCs

    * callrpc errors
    */
    RPC_UNKNOWNHOST=13,  /* unknown host name */
    */
    * create errors
    */
    RPC_PMAPFAILOURE=14,  /* the pmapper failed in its call */
    RPC_PROGNOTREGISTERED=15,  /* remote program is not registered */
    */
    * unspecified error
    */
    RPC_FAILED=16,  /* call failed */
    RPC_UNKNOWNPROTO=17 /* unknown protocol */
};

Porting

This section contains information about porting RPC applications.

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

To account for varying length enumerations, use the xdr_enum() and xdr_union() macros. xdr_enum() cannot be referenced by callrpc(), svc_freeargs(), svc_getargs(), or svc_sendreply(). An XDR routine for the specific enumeration must be created. The xdr_union() is not eligible for reference by these calls in any RPC environment. For more information, see "xdr_enum() on page 182".

Compiling, Linking, and Running an RPC Program

Note: If your program uses z/VM C sockets, follow the instructions in this section. If your program uses VM TCP/IP C sockets, see "Recompiling with the TCP/IP C Sockets Library" on page 29.

Before you compile and link an RPC program, read the information under "Compiling and Linking a z/VM C Sockets Program" on page 27 and "Running a Sockets Program" on page 31.

To compile, link and run an RPC program:

1. Access the TCP/IP Client-code disk (usually TCPMAINT 592), which contains the header files for RPC and the VMRPC TXTLIB, after the disk that contains the Language Environment header files (usually the Y-disk).
2. Specify the _OE_SOCKETS and VM preprocessor symbols in your source code or on the c89 command.
3. Compile the program using `c89`. The following are examples of how to compile the sample RPC programs shown at the end of this chapter (see “Sample RPC Programs” on page 193):

```c

c89 //genesend.c -D_OE_SOCKETS -l//VMRPC
c89 //geneserv.c -D_OE_SOCKETS -l//VMRPC
c89 //rawex.c -D_OE_SOCKETS -l//VMRPC
```

Note the use of the `//` syntax before the name of the c part. This convention informs c89 that the c source part will be found in the CMS search order. The previous three c89 commands will produce the GENESEND MODULE, GENESERV MODULE, and RAWEX MODULE, respectively. Additionally, note that `−DVM` is not specified on these compiles because the define for VM is in the C source.

4. Make sure that the SCEERUN LOADLIB is GLOBALed by issuing the command: `GLOBAL LOADLIB SCEERUN`

5. Run your program from either the CMS command line or from a POSIX shell command line. For example, run the GENESERV MODULE from the CMS command line as follows:

```cm
openvm run GENESERV
```

or

```cm
GENESERV
```

---

### RPC Global Variables

This section describes the RPC global variables, `rpc_createerr`, `svc_fds`, and `svc_fdset`.

#### rpc_createerr

**Description:** A global variable that is set when any RPC client creation routine fails. Use `clnt_pcreateerror()` to print the message.

```c
#include <rpc.h>

struct rpc_createerr rpc_createerr;
```

**See Also:** `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

#### svc_fds

```c
#include <rpc.h>

int svc_fds;
```

**Description:** 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 VM 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 VM programs for coordinating concurrent activities.

**Attention:** Do not modify this variable.

**See Also:** `svc_getreq()`.
svc_fdset

svc_fdset

#include <rpc.h>

fd_set svc_fdset;

Description: 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 VM 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 VM programs for coordinating concurrent activities.

Attention: Do not modify this variable.

See Also: svc_getreqset().

Remote Procedure and eXternal Data Representation Calls

This section provides the syntax, operands, and other appropriate information for each remote procedure and external data representation call supported by TCP/IP.

auth_destroy()

#include <rpc.h>

void auth_destroy(auth)
AUTH *auth;

Operand Description
auth Points to authentication information.

Description: The auth_destroy() call deletes the authentication information for auth. Once this procedure is called, auth is undefined.

See Also: authnone_create(), authunix_create(), authunix_create_default().

authnone_create()

#include <rpc.h>

AUTH *
authnone_create()

The authnone_create() call has no operands.

Description: The authnone_create() call creates and returns an RPC authentication handle. The handle passes the NULL authentication on each call.

See Also: auth_destroy(), authunix_create(), authunix_create_default().

authunix_create()
authunix_create()

#include <rpc.h>

AUTH *
authunix_create(host, uid, gid, len, aup_gids)
char *host;
int uid;
int gid;
int len;
int *aup_gids;

Operand Description

host Specifies a pointer to the symbolic name of the host where the desired server is located.

uid Identifies the user’s user ID.

gid Identifies the user’s group ID.

len Specifies 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.

Description: The authunix_create() call creates and returns an authentication handle that contains UNIX-based authentication information.

See Also: auth_destroy(), authnone_create(), authunix_create_default().

authunix_create_default()

#include <rpc.h>

AUTH *
authunix_create_default()

The authunix_create_default() call has no operands.

Description: The authunix_create_default() call calls authunix_create() with default operands.

See Also: auth_destroy(), authnone_create(), authunix_create().

callrpc()

#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;

Operand Description

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.
**callrpc()**

- **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.

**Description:** The callrpc() call calls the remote procedure described by progunm, versnum, and procnum running on the host system. callrpc() encodes and decodes the operands 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 182 for more information.
3. This procedure uses UDP as its transport layer, see "clntudp_create()" on page 166 for more information.

**Return Values:** RPC_SUCCESS indicates success; otherwise, an error has occurred. The results of the remote procedure call are returned to out.

**See Also:** clnt_broadcast(), clnt_call(), clnt_perrno(), clntudp_create(), clnt_sperrno(), xdr_enum().

---

**clnt_broadcast()**

```c
#include <rpc.h>

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

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>progunm</td>
<td>Identifies the program number of the remote procedure.</td>
</tr>
<tr>
<td>versnum</td>
<td>Identifies the version number of the remote procedure.</td>
</tr>
<tr>
<td>procnum</td>
<td>Identifies the procedure number of the remote procedure.</td>
</tr>
<tr>
<td>inproc</td>
<td>Specifies the XDR procedure used to encode the arguments of the remote procedure.</td>
</tr>
<tr>
<td>in</td>
<td>Specifies a pointer to the arguments of the remote procedure.</td>
</tr>
<tr>
<td>outproc</td>
<td>Specifies the XDR procedure used to decode the results of the remote procedure.</td>
</tr>
<tr>
<td>out</td>
<td>Specifies a pointer to the results of the remote procedure.</td>
</tr>
</tbody>
</table>
Specifies the procedure called after each response.

Note: resultproc_t is a type definition:

```c
#include <rpc.h>
typedef bool_t (*resultproc_t)();
```

**Description:** 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:

```c
#include <rpc.h>

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

**Operands**

- **out**
  - Has the same function as it does for clnt_broadcast(), except that the output of the remote procedure is decoded.

- **addr**
  - Points to the address of the machine that sent the results.

**Return Values:** 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.

**See Also:** callrpc(), clnt_call().

```c
#include <rpc.h>
enum clnt_stat
cint_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;
```

**Operands**

- **clnt**
  - Points to a client handle that was previously obtained using clntraw_create(), clnttcp_create(), or clntudp_create().

- **procnum**
  - Identifies the remote procedure number.

- **inproc**
  - Identifies the XDR procedure used to encode *procnum’s* arguments.

- **in**
  - Points to the remote procedure’s arguments.

- **outproc**
  - Specifies the XDR procedure used to decode the remote procedure’s results.

- **out**
  - Points to the remote procedure’s results.
clnt_call()

tout Specifies the time allowed for the server to respond.

Description: The clnt_call() call calls the remote procedure (procnum) associated with the client handle (clnt).

Return Values: RPC_SUCCESS indicates success; otherwise, an error has occurred. The results of the remote procedure call are returned to out.

See Also: callrpc(), clnt_broadcast(), clnt_geterr(), clnt_perror(), clnt_sperror(), clntraw_create(), clnttcp_create(), clntudp_create().

clnt_control()

#include <rpc.h>

bool_t
clnt_control(clnt, request, info)
CLIENT *clnt;
int request;
void *info;

Operand Description
clnt Specifies 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 Points to information used by the request.

Description: The clnt_control() call performs one of the following control operations.
• Control operations that apply to both UDP and TCP transports:
   CLSET_TIMEOUT
   Sets time-out (info points to the timeval structure).
   CLGET_TIMEOUT
   Gets time-out (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 time-out (information points to the timeval structure).
   CLGET_RETRY_TIMEOUT
   Gets retry time-out (info points to the timeval structure). If you set the timeout using clnt_control(), the timeout operand to clnt_call() will be ignored in all future calls.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: clnt_create(), clnt_destroy(), clntraw_create(), clnttcp_create(), clntudp_create().
### clnt_create()

```c
#include <rpc.h>

CLIENT *
cln_create(host, prognum, versnum, protocol)
char *host;
ulong prognum;
ulong versnum;
char *protocol;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>host</code></td>
<td>Points to the name of the host at which the remote program resides.</td>
</tr>
<tr>
<td><code>prognum</code></td>
<td>Specifies the remote program number.</td>
</tr>
<tr>
<td><code>versnum</code></td>
<td>Specifies the version number of the remote program.</td>
</tr>
<tr>
<td><code>protocol</code></td>
<td>Points to the protocol, which can be either tcp or udp.</td>
</tr>
</tbody>
</table>

**Description:** The `clnt_create()` call creates a generic 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 can be modified using `clnt_control()`.

**Return Values:** NULL indicates failure.

**See Also:** `clnt_create()`, `clnt_destroy()`, `clnt_pcreateerror()`, `clnt_spcreateerror()`, `clnt_sperror()`, `clnttcp_create()`, `clntudp_create()`.

### clnt_destroy()

```c
#include <rpc.h>

void
clnt_destroy(clnt)
CLIENT *clnt;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clnt</code></td>
<td>Points to a client handle that was previously created using <code>clnt_create()</code>, <code>clntudp_create()</code>, <code>clnttcp_create()</code>, or <code>clntraw_create()</code>.</td>
</tr>
</tbody>
</table>

**Description:** 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 socket, it will close it also. Otherwise, the socket remains open.

**See Also:** `clnt_control()`, `clnt_create()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

### clnt_freeres()
clnt_freeres()

#include <rpc.h>

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

Operand  Description
clnt     Points 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       Points to the results of the remote procedure.

Description: The clnt_freeres() call deallocates any resources that were
             assigned by the system to decode the results of an RPC.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: clntraw_create(), clnttcp_create(), clntudp_create().

clint_geterr()

#include <rpc.h>

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

Operand  Description
clnt     Points to a client handle that was previously obtained using
         clnt_create(), clntraw_create(), clnttcp_create(), or clntudp_create().
errp      Points to the address into which the error structure is copied.

Description: The clnt_geterr() call copies the error structure from the client
             handle to the structure at address errp.

See Also: clnt_call(), clnt_pcreateerror(), clnt_perrno(), clnt_perror(),
         clnt_spcreateerror(), clnt_sperrno(), clnt_sperror(), clnt_create(), clntraw_create(),
         clnttcp_create(), clntudp_create().

clint_pcreateerror()

#include <rpc.h>

void
clnt_pcreateerror(s)
char *s;

Operand  Description
s        Specifies a NULL or NULL-terminated character string. If s is
         non-NULL, clnt_pcreateerror() prints the string s followed by a
         colon, followed by a space, followed by the error message, and
clnt_pcreateerror()

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 the clnt_create(), clntraw_create(), clnttcp_create(), or clntudp_create() calls fail.

See Also: clnt_create(), clnt_geterr(), clnt_perrno(), clnt_perror(),
clnt_spcreateerror(), clnt_sperrno(), clnt_sperror(), clntraw_create(), clnttcp_create(),
clntudp_create().

clnt_perrno()

#include <rpc.h>

void
clnth_perrno(stat)
enum clnt_stat stat;

Operand Description
stat Specifies the client status.

Description: 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.

See Also: callrpc(), clnt_geterr(), clnt_pcreateerror(), clnt_perror(),
clnt_spcreateerror(), clnt_sperrno(), clnt_sperror().

clnt_perror()

#include <rpc.h>

void
clnth_perror(clnt, s)
CLIENT *clnt;
char *s;

Operand Description
clnt Points to a client handle that was previously obtained using clnt_create(), clntudp_create(), clnttcp_create(), or clntraw_create().
s Specifies a NULL or NULL-terminated character string. If s is non-NULL, clnt_perror() prints the string s followed by a colon,
followed by a space, followed by the error message, and terminated with a new-line character. If s is NULL or points to a
NULL string, just the error message and the new-line character are output.

Description: 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.
See Also: clnt_call(), clnt_create(), clnt_geterr(), clnt_pcreateerror(), clnt_perror(),
clnt_spcreateerror(), clnt_sperrno(), clnt_sperror(), clntraw_create(), clnttcp_create(),
clntudp_create().

clnt_spcreateerror()

#include <rpc.h>

char *
clnt_spcreateerror(s)
char *s;

Operand Description
s Specifies a NULL or NULL-terminated character string. If s is
non-NULL, 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 character. If s is NULL or points to a
NULL string, just the error message and the new-line character are
output.

Description: The clnt_spcreateerror() call returns the address of a message
indicating why a client handle cannot be created. This procedure is used after the
clnt_create(), clntraw_create(), clnttcp_create(), or clntudp_create() calls fail.

Return Values: Returns a pointer to a character string in a static data area. This
data area is overwritten with each subsequent call. This function is not thread-safe.

See Also: clnt_create(), clnt_geterr(), clnt_perror(), clnt_perrno(), clnt_pcreateerror(),
clnt_sperrno(), clnt_sperror(), clntraw_create(), clnttcp_create(), clntudp_create().

clnt_sperrno()

#include <rpc.h>

char *
clnt_sperrno(stat)
enum clnt_stat stat;

Operand Description
stat Specifies the client status.

Description: 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 Values: Returns a pointer to a character string ending with a newline. This
data area is overwritten with each subsequent call. This function is not thread-safe.

See Also: callrpc(), clnt_geterr(), clnt_pcreateerror(), clnt_spcreateerror(),
clnt_sperror(), clnt_perror(), clnt_perrno().

clnt_sperror()
# clnt_sperror()

```c
#include <rpc.h>

char *
clnt_sperror(clnt, s)
CLIENT *clnt;
char *s;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clnt</code></td>
<td>Points to a client handle that was previously obtained using clnt_create(), clntudp_create(), clnttcp_create(), or clntraw_create().</td>
</tr>
<tr>
<td><code>s</code></td>
<td>Specifies a NULL or a NULL-terminated character string. If <code>s</code> is non-NULL, clnt_sperror() prints the string <code>s</code> followed by a colon, followed by a space, followed by the error message, and terminated with a newline character. If <code>s</code> is NULL or points to a NULL string, just the error message and the newline character are output.</td>
</tr>
</tbody>
</table>

**Description:** 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 Values:** Returns a pointer to a character string in a static data area. This data area is overwritten with each subsequent call. This function is not thread-safe.

**See Also:** clnt_call(), clnt_create(), clnt_geterr(), clnt_pcreateerror(), clnt_perror(), clnt_spcreateerror(), clnt_sperrno(), clnttcp_create(), clntudp_create().

# clntraw_create()

```c
#include <rpc.h>

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

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>prognum</code></td>
<td>Specifies the remote program number.</td>
</tr>
<tr>
<td><code>versnum</code></td>
<td>Specifies the version number of the remote program.</td>
</tr>
</tbody>
</table>

**Description:** The clntraw_create() call creates a dummy client for the remote double `(prognum, versnum)`. Because messages are passed using a buffer within the virtual machine of the local process, the server should also use the same virtual machine, which simulates RPC programs within one virtual machine. For more information, see `svcraw_create()` on page 178.

**Return Values:** NULL indicates failure.

**See Also:** clnt_call(), clnt_destroy(), clnt_freees(), clnt_geterr(), clnt_pcreateerror(), clnt_perror(), clnt_spcreateerror(), clnt_sperrno(), clntudp_create(), clnttcp_create(), svcraw_create().

# clnttcp_create()

```c
#include <rpc.h>
```
clnttcp_create()

```c
#include <rpc.h>

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

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>Points to the internet address of the remote program. If the <code>addr</code> port number is zero (<code>addr-&gt;sin_port</code>), <code>addr</code> is set to the port on which the remote program is receiving.</td>
</tr>
<tr>
<td>prognum</td>
<td>Specifies the remote program number.</td>
</tr>
<tr>
<td>versnum</td>
<td>Specifies the version number of the remote program.</td>
</tr>
<tr>
<td>sockp</td>
<td>Points to the socket. If <code>*sockp</code> is <code>RPC_ANYSOCK</code>, then this routine opens a new socket and sets <code>*sockp</code>.</td>
</tr>
<tr>
<td>sendsz</td>
<td>Specifies the size of the send buffer. Specify 0 to choose the default.</td>
</tr>
<tr>
<td>recvsz</td>
<td>Specifies the size of the receive buffer. Specify 0 to choose the default.</td>
</tr>
</tbody>
</table>

**Description:** The `clnttcp_create()` call creates an RPC client transport handle for the remote program specified by `(prognum, versnum)`. The client uses TCP as the transport layer.

**Return Values:** NULL indicates failure.

**See Also:** `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()

```c
#include <rpc.h>

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

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>Points to the internet address of the remote program. If the <code>addr</code> port number is zero (<code>addr-&gt;sin_port</code>), <code>addr</code> is set to the port on which the remote program is receiving. The remote portmap service is used for this.</td>
</tr>
<tr>
<td>prognum</td>
<td>Specifies the remote program number.</td>
</tr>
<tr>
<td>versnum</td>
<td>Specifies the version number of the remote program.</td>
</tr>
<tr>
<td>wait</td>
<td>Indicates that UDP resends the call request at intervals of <code>wait</code></td>
</tr>
</tbody>
</table>
clntudp_create()

...time, until either a response is received or the call times out. The...time-out length is set using the clnt_call() procedure.

sockp  Points to the socket. If *sockp is RPC_ANYSOCK, this routine opens...a new socket and sets *sockp.

Description: 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 eight...kilobytes.

Return Values: NULL indicates failure.

See Also: call_rpc(), clnt_call(), clnt_control(), clnt_create(), clnt_destroy(),...clnt_spcreateerror(), clntsperror(), clntraw_create(), clnttcp_create().

get_myaddress()

#include <rpc.h>

void
get_myaddress(addr)
struct sockaddr_in *addr;

Operand  Description
addr      Points to the location where the local internet address is placed.

Description: The get_myaddress() call puts the local host's internet address into...addr. The port number (addr->sin_port) is set to htons(PMAPPORT), which is 111.

See Also: getrpcport(), pmap_getmaps(), pmap_getport(), pmap_rmtcall(),...pmap_unset().

getrpcport()

#include <rpc.h>

u_short
getrpcport(host, prognum, versnum, protocol)
char *host;
u_long prognum;
u_long versnum;
int protocol;

Operand  Description
host      Points 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).
getrpcport()

Description: The getrpcport() call returns the port number associated with the remote program (prognum), the version (versnum), and the transport protocol (protocol).

Return Values: The value 0 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.

See Also: get_myaddress(), pmap_getmaps(), pmap_getport(), pmap_rmtcall(), pmap_set(), pmap_unset().

pmap_getmaps()

#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

struct pmaplist *
pmap_getmaps(addr)
struct sockaddr_in *addr;

Operand Description
addr Points to the internet address of the foreign host.

Description: The pmap_getmaps() call returns a list of current program-to-port mappings on the foreign host specified by addr.

Return Values: Returns a pointer to a pmaplist structure or NULL.

See Also: getrpcport(), pmap_getport(), pmap_rmtcall(), pmap_set(), pmap_unset().

pmap_getport()

#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

u_short
pmap_getport(addr, prognum, versnum, protocol)
struct sockaddr_in *addr;
#define long prognum;
define long versnum;
define int protocol;

Operand Description
addr Points to the internet address of the foreign host.
prognum Identifies the program number to be mapped.
versnum Identifies the version number of the program to be mapped.
protocol Specifies the transport protocol used by the program (IPPROTO_TCP or IPPROTO_UDP).

Description: The pmap_getport() call returns the port number associated with the remote program (prognum), the version (versnum), and the transport protocol (protocol).
pmap_getport()

Return Values: The value 0 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.

See Also: getrpcport(), pmap_getmaps(), pmap_rmtcall(), pmap_set(), pmap_unset().

pmap_rmtcall()

```c
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

elem clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum, inproc, in, outproc, out, tout, portp)
struct sockaddr_in *addr;
ulong *prognum;
ulong versnum;
ulong procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
ulong *portp;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>Points to the internet address of the foreign host.</td>
</tr>
<tr>
<td>prognum</td>
<td>Identifies the remote program number.</td>
</tr>
<tr>
<td>versnum</td>
<td>Identifies the version number of the remote program.</td>
</tr>
<tr>
<td>procnum</td>
<td>Identifies the procedure to be called.</td>
</tr>
<tr>
<td>inproc</td>
<td>Identifies the XDR procedure used to encode the arguments of the remote procedure.</td>
</tr>
<tr>
<td>in</td>
<td>Points to the arguments of the remote procedure.</td>
</tr>
<tr>
<td>outproc</td>
<td>Identifies the XDR procedure used to decode the results of the remote procedure.</td>
</tr>
<tr>
<td>out</td>
<td>Points to the results of the remote procedure.</td>
</tr>
<tr>
<td>tout</td>
<td>Identifies the time-out period for the remote request.</td>
</tr>
<tr>
<td>portp</td>
<td>If the call from the remote portmap service is successful, portp contains the port number of the triple (prognum, versnum, procnum).</td>
</tr>
</tbody>
</table>

Description: The pmap_rmtcall() call instructs Portmapper on the host at addr to make an RPC call to a procedure on that host, on your behalf. This procedure should be used only for ping type functions.

Return Values: Returns a clnt_stat enumerated type.

See Also: getrpcport(), pmap_getmaps(), pmap_getport(), pmap_set(), pmap_unset().

pmap_set()
**pmap_set()**

```c
#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;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prognum</td>
<td>Identifies the local program number.</td>
</tr>
<tr>
<td>versnum</td>
<td>Identifies the version number of the local program.</td>
</tr>
<tr>
<td>protocol</td>
<td>Specifies the transport protocol used by the local program.</td>
</tr>
<tr>
<td>port</td>
<td>Identifies the port to which the local program is mapped.</td>
</tr>
</tbody>
</table>

**Description:** The `pmap_set()` call sets the mapping of the program (specified by `prognum`, `versnum`, and `protocol`) to `port` on the local machine. This procedure is automatically called by the `svc_register()` procedure.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** `getrpcport()`, `pmap_getmaps()`, `pmap_getport()`, `pmap_rmtcall()`, `pmap_unset()`.

---

**pmap_unset()**

```c
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

bool_t
pmap_unset(prognum, versnum)
  u_long prognum;
  u_long versnum;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prognum</td>
<td>Identifies the local program number.</td>
</tr>
<tr>
<td>versnum</td>
<td>Identifies the version number of the local program.</td>
</tr>
</tbody>
</table>

**Description:** The `pmap_unset()` call removes the mappings associated with `prognum` and `versnum` on the local machine. All ports for each transport protocol currently mapping the `prognum` and `versnum` are removed from the portmap service.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** `getrpcport()`, `pmap_getmaps()`, `pmap_getport()`, `pmap_rmtcall()`, `pmap_set()`.

---

**registerrpc()**
#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;

Operand    Description
prognum    The program number to register.
versnum    Identifies the version number to register.
procnum    Specifies the procedure number to register.
procname   Specifies 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 used to encode the results.

Description: 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 183 for more information.

Return Values: The value 0 indicates success; the value −1 indicates an error.

See Also: svc_register(), svc_run().

svc_destroy()

#include <rpc.h>

void
svc_destroy(xprt)
SVCXPRT *xprt;

Operand    Description
xprt    Points to the service transport handle.

Description: The svc_destroy() call deletes the RPC service transport handle xprt,
which becomes undefined after this routine is called.

See Also: svcraw_create(), svtcp_create(), svcudp_create().

svc_freeargs()
svc_freeargs()

```
#include <rpc.h>

bool_t
svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xprt</td>
<td>Points to the service transport handle.</td>
</tr>
<tr>
<td>inproc</td>
<td>Specifies the XDR routine used to decode the arguments.</td>
</tr>
<tr>
<td>in</td>
<td>Points to the input arguments.</td>
</tr>
</tbody>
</table>

**Description:** The svc_freeargs() call frees storage allocated to decode the arguments to a service procedure using svc_getargs().

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** svc_getargs().

svc_getargs()

```
#include <rpc.h>

bool_t
svc_getargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xprt</td>
<td>Points to the service transport handle.</td>
</tr>
<tr>
<td>inproc</td>
<td>Specifies the XDR routine used to decode the arguments.</td>
</tr>
<tr>
<td>in</td>
<td>Points to the decoded arguments.</td>
</tr>
</tbody>
</table>

**Description:** 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 Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** svc_freeargs().

svc_getcaller()

```
#include <rpc.h>

struct sockaddr_in *
svc_getcaller(xprt)
SVCXPRT *xprt;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xprt</td>
<td>Points to the service transport handle.</td>
</tr>
</tbody>
</table>

**Description:** This macro obtains the network address of the client associated with the service transport handle xprt.

**Return Values:** Returns a pointer to a sockaddr_in structure.


See Also: get_myaddress().

**svc_getreq()**

```c
#include <rpc.h>

void
svc_getreq(rdfds)
int rdfds;
```

### Operand Description

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdfds</td>
<td>Specifies the read descriptor bit mask.</td>
</tr>
</tbody>
</table>

**Description:** 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.

**Note:** svc_getreq() limits you to 32 socket descriptors, of which 3 are reserved. Use svc_getreqset() if you have more than 29 socket descriptors.

See Also: svc_run().

**svc_getreqset()**

```c
#include <rpc.h>

void
svc_getreqset(rdfds)
fd_set rdfds;
```

### Operand Description

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdfds</td>
<td>Specifies the read descriptor bit set.</td>
</tr>
</tbody>
</table>

**Description:** 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().

See Also: svc_run().

**svc_register()**
svc_register()

#include <rpc.h>

bool_t
svc_register(xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
unsigned prognum;
unsigned versnum;
void (*dispatch) ()
int protocol;

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xprt</td>
<td>Points to the service transport handle.</td>
</tr>
<tr>
<td>prognum</td>
<td>Specifies the program number to be registered.</td>
</tr>
<tr>
<td>versnum</td>
<td>Specifies the version number of the program to be registered.</td>
</tr>
<tr>
<td>dispatch</td>
<td>Specifies the dispatch routine associated with prognum and versnum.</td>
</tr>
</tbody>
</table>

Specifies the structure of the dispatch routine is:
#include <rpc.h>

dispatch(request, xprt)
struct svc_req *request;
SVCXPRT *xprt;

protocol
Specifies the protocol used. The value is generally one of the following:
- 0 (zero)
- IPPROTO_UDP
- IPPROTO_TCP

When a value of 0 is used, the service is not registered with Portmapper.

Note: When using a dummy RPC service transport created with svcraw_create(), a call to xprt_register() must be made immediately after a call to svc_register().

Description: The svc_register() call associates the program described by (prognum, versnum) with the service dispatch routine dispatch.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: registerrpc(), svc_unregister(), xprt_register().

svc_run()

The svc_run() call has no operands.
#include <rpc.h>

void
svc_run()

Description: The svc_run() call does not return control. It accepts RPC requests and calls the appropriate service using svc_getreqset().

See Also: svc_getreqset().

svc_sendreply()
svc_sendreply()

```c
#include <rpc.h>

bool_t
svc_sendreply(xprt, outproc, out)
SVCXprt *xprt;
xdrproc_t outproc;
char *out;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xprt</td>
<td>Points to the caller’s transport handle.</td>
</tr>
<tr>
<td>outproc</td>
<td>Specifies the XDR procedure used to encode the results.</td>
</tr>
<tr>
<td>out</td>
<td>Points to the results.</td>
</tr>
</tbody>
</table>

**Description:** The `svc_sendreply()` call is called by the service dispatch routine to send the results of the call to the caller.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** `callrpc()`, `clnt_call()`.

svc_unregister()

```c
#include <rpc.h>

void
svc_unregister(prognum, versnum)
u_long prognum;
u_long versnum;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prognum</td>
<td>Specifies the program number that is removed.</td>
</tr>
<tr>
<td>versnum</td>
<td>Specifies the version number of the program that is removed.</td>
</tr>
</tbody>
</table>

**Description:** The `svc_unregister()` call removes all local mappings of `(prognum, versnum)` to dispatch routines and `(prognum, versnum, *)` to port numbers.

**See Also:** `svc_register()`.

svcerr_auth()

```c
#include <rpc.h>

void
svcerr_auth(xprt, why)
SVCXprt *xprt;
enum auth_stat why;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xprt</td>
<td>Points to the service transport handle.</td>
</tr>
<tr>
<td>why</td>
<td>Specifies the reason the call is refused.</td>
</tr>
</tbody>
</table>

**Description:** The `svcerr_auth()` call is called by a service dispatch routine that refuses to execute an RPC request because of authentication errors.

**See Also:** `svcerr_decode()`, `svcerr_noproc()`, `svcerr_noprog()`, `svcerr_progvers()`, `svcerr_systemerr()`, `svcerr_weakauth()`.
svcerr_decode()

svcerr_decode()

#include <rpc.h>

void
svcerr_decode(xprt)
SVCXPRT *xprt;

Operand    Description
xprt        Points to the service transport handle.

Description: The svcerr_decode() call is called by a service dispatch routine that cannot decode its operands.

See Also: svcerr_auth(), svcerr_noproc(), svcerr_noprog(), svcerr_progvers(), svcerr_systemerr(), svcerr_weakauth().

svcerr_noproc()

svcerr_noproc()

#include <rpc.h>

void
svcerr_noproc(xprt)
SVCXPRT *xprt;

Operand    Description
xprt        Points to the service transport handle.

Description: The svcerr_noproc() call is called by a service dispatch routine that does not implement the requested procedure.

See Also: svcerr_auth(), svcerr_decode(), svcerr_noprog(), svcerr_progvers(), svcerr_systemerr(), svcerr_weakauth().

svcerr_noprog()

svcerr_noprog()

#include <rpc.h>

void
svcerr_noprog(xprt)
SVCXPRT *xprt;

Operand    Description
xprt        Points to the service transport handle.

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

See Also: svcerr_auth(), svcerr_decode(), svcerr_noproc(), svcerr_progvers(), svcerr_systemerr(), svcerr_weakauth().

svcerr_progvers()
#include <rpc.h>

void
svcerr_progvers(xprt, low_vers, high_vers)
SVCXPRT *xprt;
ulong low_vers;
ulong high_vers;

**Operand** | **Description**
--- | ---
xprt | Points 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.

**Description:** 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.

**See Also:** `svcerr_auth()`, `svcerr_decode()`, `svcerr_noproc()`, `svcerr_noprog()`, `svcerr_progvers()`, `svcerr_systemerr()`, `svcerr_weakauth()`. 

---

svcerr_systemerr()

#include <rpc.h>

void
svcerr_systemerr(xprt)
SVCXPRT *xprt;

**Operand** | **Description**
xprt | Points to the service transport handle.

**Description:** The `svcerr_systemerr()` call is called by a service dispatch routine when it detects a system error that is not handled by the protocol.

**See Also:** `svcerr_auth()`, `svcerr_decode()`, `svcerr_noproc()`, `svcerr_noprog()`, `svcerr_progvers()`, `svcerr_weakauth()`. 

---

svcerr_weakauth()

#include <rpc.h>

void
svcerr_weakauth(xprt)
SVCXPRT *xprt;

**Operand** | **Description**
xprt | Points to the service transport handle.

**Note:** This is the equivalent of: `svcerr_auth(xprt, AUTH_TOOWEAK)`.

**Description:** The `svcerr_weakauth()` call is called by a service dispatch routine that cannot execute an RPC because of correct but weak authentication operands.

**See Also:** `svcerr_auth()`, `svcerr_decode()`, `svcerr_noproc()`, `svcerr_noprog()`, `svcerr_progvers()`, `svcerr_systemerr()`. 

---
svcraw_create()

The svcraw_create() call has no operands.

#include <rpc.h>

SVCXPRT *
svcraw_create()

Description: 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 virtual machine of the local process; so, the client process must also use the same virtual machine. This allows the simulation of RPC programs within one computer. See "clntraw_create()" on page 163 for more information.

Return Values: NULL indicates failure.

See Also: clntraw_create(), svc_destroy(), svctcp_create(), svcudp_create().

svctcp_create()

#include <rpc.h>

SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
int sock;
uint send_buf_size;
uint recv_buf_size;

Operand Description

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.

Description: The svctcp_create() call creates a TCP-based service transport to which it returns a pointer. xprt->xp_sock contains the transport’s socket descriptor. xprt->xp_port contains the transport’s port number.

Return Values: NULL indicates failure.

See Also: svc_destroy(), svcraw_create(), svcudp_create().

svcudp_create()

#include <rpc.h>

SVCXPRT *
svcudp_create(sock, sendsz, recvsz)
int sock;
uint sendsz;
uint recvsz;

Operand Description

sock Specifies the socket descriptor. If sock is RPC_ANYSOCK, a new socket is created. If the socket is not bound to a local UDP port, it is bound to an arbitrary port.

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.
sock
Specifies the socket descriptor. If sock is RPC_ANYSOCK, a new 
socket is created. If the socket is not bound to a local UDP port, it 
is bound to an arbitrary port.

sendsz
Specifies the size of the send buffer.

recvsz
Specifies the size of the receive buffer.

Description: The svcudp_create() call creates a UDP-based service transport to 
which it returns a pointer. xprt—>xp_sock contains the transport’s socket 
descriptor. xprt—>xp_port contains the transport’s port number.

Return Values: NULL indicates failure.

See Also: svc_destroy(), svcraw_create(), svctcp_create().

xdr_accepted_reply()

#include <rpc.h>

bool_t
xdr_accepted_reply(xdrs, ar)
XDR *xdrs;
struct accepted_reply *ar;

Operand Description
xdrs Points to an XDR stream.
ar Points to the reply to be represented.

Description: The xdr_accepted_reply() call translates RPC reply messages.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(),
registrerp(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_array()

#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;

Operand Description
xdrs Points to an XDR stream.
arrp Specifies the address of the pointer to the array.
sizep Points 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().
xdr_array()

Specifies the XDR routine that translates an individual array element.

Description: The xdr_array() call translates between an array and its external representation.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_authunix_parms()

#include <rpc.h>

bool_t
xdr_authunix_parms(xdrs, aupp)
XDR *xdrs;
struct authunix_parms *aupp;

Operand Description
xdrs Points to an XDR stream.
aupp Points to the authentication information.

Description: The xdr_authunix_parms() call translates UNIX-based authentication information.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_bool()

#include <rpc.h>

bool_t
xdr_bool(xdrs, bp)
XDR *xdrs;
bool_t *bp;

Operand Description
xdrs Points to an XDR stream.
bp Points to the Boolean.

Description: The xdr_bool() call translates between Booleans and their external representation.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply.

xdr_bytes()
# include <rpc.h>

bool_t
xdr_bytes(xdrs, sp, sizep, maxsize)
XDR *xdrs;
char **sp;
u_int *sizep;
u_int maxsize;

**Operand** | **Description**
--- | ---
xdrs | Points to an XDR stream.
sp | Points to a pointer to the byte string.
sizep | Points to the byte string size.
maxsize | Specifies the maximum size of the byte string.

**Description:** The xdr_bytes() call translates between byte strings and their external representations.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_callhdr()

#include <rpc.h>

bool_t
xdr_callhdr(xdrs, chdr)
XDR *xdrs;
struct rpc_msg *chdr;

**Operand** | **Description**
xdrs | Points to an XDR stream.
chdr | Points to the call header.

**Description:** The xdr_callhdr() call translates an RPC message header into XDR format.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_callmsg()

#include <rpc.h>

bool_t
xdr_callmsg(xdrs, cmsg)
XDR *xdrs;
struct rpc_msg *cmsg;

**Operand** | **Description**
xdrs | Points to an XDR stream.
cmsg | Points to the call message.
xdr_callmsg()

**Description:** The xdr_callmsg() call translates RPC messages (header and authentication, not argument data) to and from the xdr format.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_double()

```c
#include <rpc.h>

bool_t xdr_double(xdrs, dp)
XDR *xdrs;
double *dp;
```

**Operand** | **Description**
--- | ---
`xdrs` | Points to an XDR stream.
`dp` | Points to a double-precision number.

**Description:** The xdr_double() call translates between C double-precision numbers and their external representations.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_enum()

```c
#include <rpc.h>

bool_t xdr_enum(xdrs, ep)
XDR *xdrs;
enum_t *ep;
```

**Operand** | **Description**
--- | ---
`xdrs` | Points to an XDR stream.
`ep` | Points to the enumerated number. `enum_t` can be any enumeration type such as `enum colors`, with `colors` declared as `enum colors` (black, brown, red).

**Description:** 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(). The following is the format of the stub procedure.
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:

```c
error = callrpc(argv[1], ENUMRCVPROG, VERSION, ENUMRCVPROC,
    xdr_enum_t, &innumber, xdr_enum_t,
    &outnumber);
```

An RPC server would contain the following line:

```c
registerrpc(ENUMRCVPROG, VERSION, ENUMRCVPROC,
    xdr_enum_t, xdr_enum_t);
```

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

### xdr_float()

```c
#include <rpc.h>

bool_t xdr_float(xdrs, fp)
    XDR *xdrs;
    float *fp;
```

**Operand Description**

- **xdrs**: Points to an XDR stream.
- **fp**: Points to the floating-point number.

**Description:** The xdr_float() call translates between C floating-point numbers and their external representations.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.
xdr_float()

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_inline()

#include <rpc.h>

long *
xdr_inline(xdrs, len)
XDR *xdrs;
uint len;

Operand    Description
xdrs       Points to an XDR stream.
len         Specifies the byte length of the desired buffer.

Description: The xdr_inline() call returns a pointer to a continuous piece of the XDR stream’s 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 Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_int()

#include <rpc.h>

bool_t
xdr_int(xdrs, ip)
XDR *xdrs;
int *ip;

Operand    Description
xdrs       Points to an XDR stream.
ip         Points to the integer.

Description: The xdr_int() call translates between C integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_long()
xdr_long()

```c
#include <rpc.h>

bool_t
xdr_long(xdrs, lp)
XDR *xdrs;
long *lp;
```

**Operand** | **Description**
--- | ---
xdrs | Points to an XDR stream.
lp | Points to the long integer.

**Description:** The xdr_long() call translates between C long integers and their external representations.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

---

xdr_opaque()

```c
#include <rpc.h>

bool_t
xdr_opaque(xdrs, cp, cnt)
XDR *xdrs;
char *cp;
u_int cnt;
```

**Operand** | **Description**
xdrs | Points to an XDR stream.
cp | Points to the opaque object.
cnt | Specifies the size of the opaque object.

**Description:** The xdr_opaque() call translates between fixed-size opaque data and its external representation.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

---

xdr_opaque_auth()

```c
#include <rpc.h>

bool_t
xdr_opaque_auth(xdrs, ap)
XDR *xdrs;
struct opaque_auth *ap;
```

**Operand** | **Description**
xdrs | Points to an XDR stream.
ap | Points to the opaque authentication information.

**Description:** The xdr_opaque_auth() call translates RPC message authentications.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.
### xdr_pmap()

```c
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

bool_t
xdr_pmap(xdrs, regs)
XDR *xdrs;
struct pmap *regs;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xdrs</td>
<td>Points to an XDR stream.</td>
</tr>
<tr>
<td>regs</td>
<td>Points to the portmap operands.</td>
</tr>
</tbody>
</table>

**Description:** The `xdr_pmap()` call translates an RPC procedure identification, such as is used in calls to Portmapper.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

### xdr_pmaplist()

```c
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

bool_t
xdr_pmaplist(xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xdrs</td>
<td>Points to an XDR stream.</td>
</tr>
<tr>
<td>rp</td>
<td>Points to a pointer to the portmap data array.</td>
</tr>
</tbody>
</table>

**Description:** The `xdr_pmaplist()` call translates a variable number of RPC procedure identifications, such as Portmapper creates.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

### xdr_pointer()
xdr_pointer()

#include <rpc.h>

bool_t
xdr_pointer(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
ulong_t size;
xdrproc_t proc;

Operand          Description
xdrs             Points to an XDR stream.
pp               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.

Description:   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 Values: The value 1 indicates success; the value 0 indicates an error.

See Also:       callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(),
                 registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_reference()

#include <rpc.h>

bool_t
xdr_reference(xdrs, pp, size, proc)
XDR *xdrs;
ulong_t size;
xdrproc_t proc;

Operand          Description
xdrs             Points to an XDR stream.
pp               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.

Description:   The xdr_reference() call provides pointer-chasing within structures.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also:       callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(),
                 registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_rejected_reply()
xdr_rejected_reply()

```c
#include <rpc.h>

bool_t
xdr_rejected_reply(xdrs, rr)
    XDR *xdrs;
    struct rejected_reply *rr;
```

**Operand**  
**Description**
- `xdrs` Points to an XDR stream.
- `rr` Points to the rejected reply.

**Description:** The `xdr_rejected_reply()` call translates rejected RPC reply messages.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

---

xdr_replymsg()

```c
#include <rpc.h>

bool_t
xdr_replymsg(xdrs, rmsg)
    XDR *xdrs;
    struct rpc_msg *rmsg;
```

**Operand**  
**Description**
- `xdrs` Points to an XDR stream.
- `rmsg` Points to the reply message.

**Description:** The `xdr_replymsg()` call translates RPC reply messages.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

---

xdr_short()

```c
#include <rpc.h>

bool_t
xdr_short(xdrs, sp)
    XDR *xdrs;
    short *sp;
```

**Operand**  
**Description**
- `xdrs` Points to an XDR stream.
- `sp` Points to the short integer.

**Description:** The `xdr_short()` call translates between C short integers and their external representations.

**Return Values:** The value 1 indicates success; the value 0 indicates an error.

**See Also:** `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.
xdr_string()

#include <rpc.h>

bool_t
xdr_string(xdrs, sp, maxsize)
XDR *xdrs;
char **sp;
u_int maxsize;

Operand Description
xdrs Points to an XDR stream.
sp Points to a pointer to the string.
maxsize Specifies the maximum size of the string.

Description: The xdr_string() call translates between C strings and their external representations. The xdr_string() call is the only xdr routine to convert ASCII to EBCDIC.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_u_int()

#include <rpc.h>

bool_t
xdr_u_int(xdrs, up)
XDR *xdrs;
unsigned *up;

Operand Description
xdrs Points to an XDR stream.
up Points to the unsigned integer.

Description: The xdr_u_int() call translates between C unsigned integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_u_long()

#include <rpc.h>

bool_t
xdr_u_long(xdrs, ulp)
XDR *xdrs;
u_long *ulp;

Operand Description
xdrs Points to an XDR stream.
ulp Points to the unsigned long integer.
xdr_u_long()

Description: The xdr_u_long() call translates between C unsigned long integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_u_short()

#include <rpc.h>

bool_t
xdr_u_short(xdrs, usp)
XDR *xdrs;
u_short *usp;

Operand Description
xdrs Points to an XDR stream.
usp Points to the unsigned short integer.

Description: The xdr_u_short() call translates between C unsigned short integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_union()

#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;

Operand Description
xdrs Points to an XDR stream.
dscmp Points to the union’s discriminant. enum_t can be any enumeration type.
unp Points to the union.
choices Points to an array detailing the XDR procedure to use on each arm of the union.
default Specifies the default XDR procedure to use.

Description: The xdr_union() call translates between a discriminated C union and its external representation.

Return Values: The value 1 indicates success; the value 0 indicates an error.

The following is an example of this call:
en include <rpc.h>

enum colors (black, brown, red);

bool_t
xdr_union(xdrs, dscmp, unp, choices, default)
XDR *xdrs;
enum colors *dscmp;
char *unp;
struct xdr_discrim *choices;
xdrproc_t default;

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(),
registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_vector()

# 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;

Operand Description
xdrs Points to an XDR stream.
basep Specifies the base of the array.
nelem Specifies the element count of the array.
elemsize Specifies the size of each of the array’s elements, found using
sizeof().
xdr_elem Specifies the XDR routine that translates an individual array
element.

Description: 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 unfreeable.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(),
registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_void()

The xdr_void() call has no operands.
# include <rpc.h>

bool_t
xdr_void()

Description: The xdr_void() call is used like a command that does not require
any other xdr functions. This call can be placed in the inproc or outproc operand of
the clnt_call function when the user does not need to move data.
xdr_wrapstring()

Return Values: Always a value of 1.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply.

xdr_wrapstring()

#include <rpc.h>

bool_t
xdr_wrapstring(xdrs, sp)
XDR *xdrs;
char **sp;

Operand Description
xdrs Points to an XDR stream.
sp Points to a pointer to the string.

Description: 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-operand XDR routines, and xdr_string() is a three-operand routine.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply.

xdrmem_create()

#include <rpc.h>

void
xdrmem_create(xdrs, addr, size, op)
XDR *xdrs;
char *addr;
u_int size;
enum xdr_op op;

Operand Description
xdrs Points to an XDR stream.
addr Points 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).

Description: The xdrmem_create() call initializes the XDR stream pointed to by xdrs. Data is written to, or read from, addr.

xdrrec_create()
#include <rpc.h>

void
xdrrec_create(xdrs, sendsize, recvsize, handle, readit, writeit)
XDR *xdrs;
uint sendsize;
uint recvsize;
char *handle;
int (**readit)();
int (**writeit)();

**Operand** | **Description**
--- | ---
xdrs | Points to an XDR stream.
sendsize | Indicates the size of the send buffer. Specify 0 to choose the default.
recvsize | Indicates the size of the receive buffer. Specify 0 to choose the default.
handle | Specifies the first operand passed to readit() and writeit().
readit() | Called when a stream’s input buffer is empty.
writeit() | Called when a stream’s output buffer is full.

**Description:** The xdrrec_create() call creates a record-oriented stream and initializes the XDR stream pointed to by xdrs.

**Notes:**
1. The x_op field must be set by the caller.
2. This XDR procedure implements an intermediate record string.
3. Additional bytes in the XDR stream provide record boundary information.

# include <rpc.h>

bool_t
xdrrec_endofrecord(xdrs, sendnow)
XDR *xdrs;
int sendnow;

**Operand** | **Description**
xdrs | Points to an XDR stream.
sendnow | Specifies nonzero to write out data in the output buffer.

**Description:** 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 Values:** The value 1 indicates success; the value 0 indicates an error.

# include <rpc.h>

bool_t
xdrrec_eof(xdrs)
XDR *xdrs;

**Operand** | **Description**
**xdrrec_eof()**

```c
xdrs Points to an XDR stream.
```

**Description:** The `xdrrec_eof()` call can be invoked only on streams created by `xdrrec_create()`.

**Return Values:** The value 1 indicates the current record has been consumed; the value 0 indicates continued input on the stream.

---

**xdrrec_skiprecord()**

```c
#include <rpc.h>
bool_t xdrrec_skiprecord(xdrs)
    XDR *xdrs;
```

**Operand Description**

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xdrs</td>
<td>Points to an XDR stream.</td>
</tr>
</tbody>
</table>

**Description:** 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 Values:** The value 1 indicates success; the value 0 indicates an error.

---

**xdrstdio_create()**

```c
#include <rpc.h>
#include <stdio.h>
void xdrstdio_create(xdrs, file, op)
    XDR *xdrs;
    FILE *file;
    enum xdr_op op;
```

**Operand Description**

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xdrs</td>
<td>Points to an XDR stream.</td>
</tr>
<tr>
<td>file</td>
<td>Specifies the file name for the I/O stream.</td>
</tr>
<tr>
<td>op</td>
<td>Determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).</td>
</tr>
</tbody>
</table>

**Description:** The `xdrstdio_create()` call initializes the XDR stream pointed to by `xdrs`. Data is written to, or read from, `file`.

**Note:** `fflush()` is the destroy routine associated with this procedure. `fclose()` is not called.

---

**xprt_register()**

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

---

194  
---
xprt_register()

Operand Description
xprt Points to the service transport handle.

Description: The xprt_register() call registers service transport handles with the RPC service package. This routine also modifies the global variable svc_fds.

See Also: svc_register(), svc_fds.

xprt_unregister()

#include <rpc.h>

void
xprt_unregister(xprt)
SVCXPRT *xprt;

Operand Description
xprt Points to the service transport handle.

Description: 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 variable svc_fds and svc_fdset.

See also: svc_fds, svc_fdset.

Sample RPC Programs

This appendix provides examples of the following programs:
- RPC Genesend client (see page 196)
- RPC Geneserv server (see page 197)
- RPC Rawex raw data stream (see page 198)

Refer back to "Compiling, Linking, and Running an RPC Program" on page 153 for examples of how to compile, link, and run RPC programs.

Running the Geneserv server and Genesend client

The Geneserv server and Genesend client are a pair of client-server programs. Typically, the Geneserv server is run in one virtual machine and the Genesend client in another. If a POSIX shell command line is available, both can be run in the same virtual machine by starting the Geneserv server in the background and running the Genesend client in the foreground. The steps for running these programs are as follows:

1. Make sure the TCPIP Client-code disk is accessed (usually TCPMAINT 592).
2. Before the Geneserv server can be started, the Portmapper must be running. To determine if the Portmapper is running, contact the Portmapper with the command RPCINFO -p.
3. Start the Geneserv server. From the CMS command line issue:
   openvm run GENESERV

   To start in the foreground from a POSIX shell command line issue:
   geneserv

   To start in the background from a POSIX shell command line issue:
Sample RPC Programs

geneserv &

After starting the Geneserv server you should see output similar to the following:
```
openvm run GENESERV
Intrcv Registration with Port Mapper completed
Floatrcv Registration with Port Mapper completed
integer received: 10
integer being returned: 10
```

4. Start the Genesend client. From the CMS command line issue:
```
openvm run GENESEND hostname some_number
```

To start in the foreground from a POSIX shell command line issue:
```
genesis hostname some_number
```

To start in the background from a POSIX shell command line issue:
```
genesis hostname some_number &
```

The `hostname` argument is the host running the Geneserv server. The `some_number` argument is an integer value that will be sent to the Geneserv server and then returned.

The following is a sample run of the Genesend client:
```
openvm run GENESEND myvmhost 10
value sent: 10 value received: 10
Ready;
```

Running the Rawex program

The rawex program uses the raw RPC interfaces and is a client and server program in the same program. To start Rawex from a CMS command line issue:
```
openvm run RAWEX some_number
```

To start in the foreground from a POSIX shell command line issue:
```
rawex some_number
```

To start in the background from a POSIX shell command line issue:
```
rawex some_number &
```

The following is a sample run of Rawex:
```
openvm run RAWEX 5678
Argument: 5678
Received: 5678
Sent: 5678
Result: 5678
Ready;
```

RPC Genesend Client

The following is an example of an RPC client program.
```
/* GENESEND.C */
/* Send an integer to the remote host and receive the integer back */
/* PORTMAPPER AND REMOTE SERVER MUST BE RUNNING */

#define VM
#include <stdio.h>
#include <rpc.h>
```
RPC Geneserv 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 */

#define VM

#include <rpc.h>
#include <stdio.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();
    float *floatrcv();

    /*REGISTER PROG, VERS AND PROC WITH THE PORTMAPPER*/

    /*FIRST PROGRAM*/
    registerrpc(intrcvprog,intvers,intrcvproc,intrcv,xdr_int,xdr_int);
    registerrpc(fltrcvprog,fltvers,fltrcvproc,floatrcv,xdr_float,xdr_float);
}
RPC Server

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 *
inintrcv(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);
}

RPC Rawex 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 */
/* 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. */
#define VM
#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)
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", 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);

    /* 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);
    }
}
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
", 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);
    }
  }
}
Chapter 6. X Window System Interface

This chapter contains information specific to the VM implementation of the X Window System. The X Window System application program interface (API) allows you to write applications in the VM/CMS environment that can be displayed on X11 servers on a TCP/IP-based network. This API provides the application with graphics capabilities as defined by the X Window System protocol. For more information about the X protocol and application program interface, see the X Window System publications listed in "Bibliography" on page 399.

What Is Provided

The X Window System support provided with TCP/IP includes support for the following API from the X Window System Version 11, Release 4:

- X11LIB TXTLIB (Xlib, Xmu, Xext, and Xau routines)
- OLDDXLIB TXTLIB (X Release 10 compatibility routines)
- XTLIB TXTLIB (X Intrinsics)
- XAWLIB TXTLIB (Athena widget set)
- Header files needed for compiling X clients

In addition, it also includes support for the following API based on Release 1.1 of the OSF/Motif-based widget set:

- XMLIB TXTLIB (OSF/Motif-based widget set)
- Header files needed for compiling clients using the OSF/Motif-based widget set.

Software Requirements

Application programs using the X Window System API are written in C and require the following:

- IBM C for VM/ESA Compiler, Version 3 Release 1 Program (Program Number 5654-033).
- IBM Language Environment® for MVS & VM, Version 1 Release 5 Program (Program Number 5688-198).

Using the X Window System Interface in the VM 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 VM/CMS environment, X Window System support consists of a set of application calls that create the X protocol, as requested by the application. This application program 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 30 shows a high-level abstraction of how the X Window System works in a VM environment. As an application writer, you need to be concerned only with the client API in writing your application.

User Virtual Machine

The communication path from the VM 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 VM/CMS encoding for character strings is EBCDIC. The X Window System client code in the VM/CMS environment automatically transforms character strings from EBCDIC to ISO Latin-1 or from ISO Latin-1 to EBCDIC, as needed using internal translate tables.
When programming using the C/VM™ Compiler, CMS file identifiers are specified as `filename filetype filemode`. In the following sections a file specified as `filename.filetype` refers to the CMS file, file name file type.

In the VM/CMS environment, external names must be eight characters or less. Many of the X Window System application program interface names exceed this limit. To support the X API in CMS, all X names longer than eight characters are remapped to unique names using the C compiler preprocessor. This name remapping is found in a header file called `X11GLUE.H`, which is automatically included in your program when you include the standard X header file called `XLIB.H`. In debugging your application, it may be helpful to reference the `X11GLUE.H` header file to find the remapped names of the X API routines.

### 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 may 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 VM/CMS environment, this file is called `X DEFAULTS`. Each line of this file represents resource information for an application. Figure 31 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 31. 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, which control all applications that are displayed on an X server. You can use the `xrdb` program to control the X server resource database in the resource property.

The `xrdb` program 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.
Identifying the Target Display

A CMS global command is used by the X Window System to identify the internet address of the target display.

The following is the format of the CMS global command.

```
GLOBALV SELECT CENV SET DISPLAY internet_address : target_server
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>internet_address</td>
<td>Specifies the internet address of the host machine on which the X Window System server is running.</td>
</tr>
<tr>
<td>target_server</td>
<td>Specifies the number of the display server on the host machine.</td>
</tr>
<tr>
<td>target_screen</td>
<td>Specifies the screen to be used on the same target server.</td>
</tr>
</tbody>
</table>

Creating an Application

To create an application that uses the X Window System protocol, you should study the X Window System application program interface.

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 X Intrinsics, you should ensure that the INTRINSIC.H header file is the first X header file included in your program. This file defines a number of preprocessor symbols that allow your program to compile correctly. In addition, these header files include the CMS header files that remap the external names of the X Window System routines to shorter names that are unique within the X Window System support in TCP/IP.

Generating X-Window System Applications

The following steps should serve as a guideline for generating an X Window System application called myxprog. Your installation may have different names or techniques for performing these steps.

Before you begin to generate your application, make sure you have access to the C/VM Compiler and to the TCPMAINT 592 minidisk, where the X Window System support is installed. To compile your application, do the following:

1. Set the LOADLIB and TXTLIB search order by entering the following GLOBAL commands:
   ```
   SET LDRTBLS 25
   GLOBAL LOADLIB SCEERUN
   GLOBAL TXTLIB X11LIB COMMTXT CMSLIB
   ```

2. Compile the application with the IBM C for VM/ESA Compiler by entering the following command:
   ```
   CC myxprog (DEFINE(IBMCPP))
   ```
The option passed to the CC EXEC defines the preprocessor symbol IBMCPP, which is used in many of the X Window System header files by the C preprocessor to generate the correct code for use under CMS. To simplify compiling X Window System applications, you can create a version of the CC EXEC that automatically defines this symbol. If your application includes the XLIB.H header file as the first included X Window System header file, you do not have to define the IBMCPP preprocessor symbol, because it is automatically defined by this header file.

For applications written using the X Window System Toolkit, including the INTRINSIC.H header file as the first X Window System header file also defines the preprocessor symbol for you. You do not have to define it again when you invoke the compiler.

3. Generate the executable module by entering the CMOD command:

   CMOD myxprog

   You should now have the file, myxprog MODULE, on your A disk. Verify that you have set the LOADLIB and TXTLIB search order. To run this module, do the following:

4. Specify the IP address of the X server on which you wish to display the application output by setting the DISPLAY global variable in the CENV (C Environment) group of global variables:

   GLOBALV SELECT CENV SET DISPLAY charm.cambridge.ibm.com:0.0

   or

   GLOBALV SELECT CENV SET DISPLAY 129.42.3.105:0.0

   **Note:** charm.cambridge.ibm.com:0.0 and 129.42.3.105:0.0 are example addresses.

5. Allow the host application access to the X server.

   On the workstation where you wish to display the application output, you must grant permission for the VM host to access the X server. To do this enter the xhost command:

   xhost cambvm3

   **Note:** cambvm3 is an example host name.

6. Run the application by entering the following command:

   myxprog

   Typically, an X application uses the global variable DISPLAY in the CENV group as the X server address on which it is to display. Some applications also allow you to specify this information as a command line option. You should refer to the documentation for the application you are attempting to run for these and other options that are available to you.

   In the preceding example, the application is displayed on the X server at the internet address associated with the name charm.cambridge.ibm.com. The application is displayed on the X server on screen 0 on this system.

   In the preceding example, the GLOBAL TXTLIB command specified the X11LIB and COMMTXT TXTLIBs in the TXTLIB search order. Depending on the facilities of X that your application makes use of, additional libraries may have to be specified to generate your application. For example, to generate an application that is written to the OSF/Motif interface, specify the following command:
X Window System Interface

The following tables list the subroutines supported by TCP/IP for z/VM. The subroutines are grouped according to the type of function provided.

Opening and Closing a Display

This section provides information about X Window System in tabular form for quick reference.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCloseDisplay()</td>
<td>Closes a display.</td>
</tr>
<tr>
<td>XFree()</td>
<td>Frees in-memory data created by Xlib function.</td>
</tr>
<tr>
<td>XNoOp()</td>
<td>Executes a NoOperation protocol request.</td>
</tr>
<tr>
<td>XOpenDisplay()</td>
<td>Opens a display.</td>
</tr>
</tbody>
</table>

Creating and Destroying Windows

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XConfigureWindow()</td>
<td>Configures the specified window.</td>
</tr>
<tr>
<td>XCreateSimpleWindow()</td>
<td>Creates unmapped InputOutput subwindow.</td>
</tr>
<tr>
<td>XCreateWindow()</td>
<td>Creates unmapped subwindow.</td>
</tr>
<tr>
<td>XDestroySubwindows()</td>
<td>Destroys all subwindows of specified window.</td>
</tr>
<tr>
<td>XDestroyWindow()</td>
<td>Unmaps and destroys window and all subwindows.</td>
</tr>
</tbody>
</table>
Manipulating Windows

Table 27 provides the subroutines for manipulating windows.

Table 27. Manipulating Windows

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCirculateSubwindows()</td>
<td>Circulates a subwindow up or down.</td>
</tr>
<tr>
<td>XCirculateSubwindowsUp()</td>
<td>Raises the lowest mapped child of the window.</td>
</tr>
<tr>
<td>XCirculateSubwindowsDown()</td>
<td>Lowers the highest mapped child of the window.</td>
</tr>
<tr>
<td>XIconifyWindow()</td>
<td>Sends a WM_CHANGE_STATE ClientMessage to the root window of the specified screen.</td>
</tr>
<tr>
<td>XLowerWindow()</td>
<td>Lowers the specified window.</td>
</tr>
<tr>
<td>XMapRaised()</td>
<td>Maps and raises the specified window.</td>
</tr>
<tr>
<td>XMapSubwindows()</td>
<td>Maps all subwindows of the specified window.</td>
</tr>
<tr>
<td>XMapWindow()</td>
<td>Maps the specified window.</td>
</tr>
<tr>
<td>XMoveResizeWindow()</td>
<td>Changes the specified window’s size and location.</td>
</tr>
<tr>
<td>XMoveWindow()</td>
<td>Moves the specified window.</td>
</tr>
<tr>
<td>XRaiseWindow()</td>
<td>Raises the specified window.</td>
</tr>
<tr>
<td>XReconfigureWMWindow()</td>
<td>Issues a ConfigureWindow request on the specified top-level window.</td>
</tr>
<tr>
<td>XResizeWindow()</td>
<td>Changes the specified window’s size.</td>
</tr>
<tr>
<td>XRestackWindows()</td>
<td>Restacks a set of windows from top to bottom.</td>
</tr>
<tr>
<td>XSetWindowBorderWidth()</td>
<td>Changes the border width of the window.</td>
</tr>
<tr>
<td>XUnmapSubwindows()</td>
<td>Unmaps all subwindows of the specified window.</td>
</tr>
<tr>
<td>XUnmapWindow()</td>
<td>Unmaps the specified window.</td>
</tr>
<tr>
<td>XWithdrawWindow()</td>
<td>Unmaps the specified window and sends a synthetic UnmapNotify event to the root window of the specified screen.</td>
</tr>
</tbody>
</table>

Changing Window Attributes

Table 28 provides the subroutines for changing window attributes.

Table 28. Changing Window Attributes

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XChangeWindowAttributes()</td>
<td>Changes one or more window attributes.</td>
</tr>
<tr>
<td>XSetWindowBackground()</td>
<td>Sets the window’s background to a specified pixel.</td>
</tr>
<tr>
<td>XSetWindowBackgroundPixmap()</td>
<td>Sets the window’s background to a specified pixmap.</td>
</tr>
<tr>
<td>XSetWindowBorder()</td>
<td>Changes the window’s border to a specified pixel.</td>
</tr>
</tbody>
</table>
Table 28. Changing Window Attributes (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSetWindowBorderPixmap()</td>
<td>Changes the window’s border tile.</td>
</tr>
<tr>
<td>XTranslateCoordinates()</td>
<td>Transforms coordinates between windows.</td>
</tr>
</tbody>
</table>

### Obtaining Window Information

Table 29 provides the subroutines for obtaining window information.

Table 29. Obtaining Window Information

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XGetGeometry()</td>
<td>Gets the current geometry of the specified drawable.</td>
</tr>
<tr>
<td>XGetWindowAttributes()</td>
<td>Gets the current attributes for the specified window.</td>
</tr>
<tr>
<td>XQueryPointer()</td>
<td>Gets the pointer coordinates and the root window.</td>
</tr>
<tr>
<td>XQueryTree()</td>
<td>Obtains the IDs of the children and parent windows.</td>
</tr>
</tbody>
</table>

### Obtaining Properties and Atoms

Table 30 provides the subroutines for obtaining properties and atoms.

Table 30. Properties and Atoms

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XGetAtomName()</td>
<td>Gets a name for the specified atom ID.</td>
</tr>
<tr>
<td>XInternAtom()</td>
<td>Gets an atom for the specified name.</td>
</tr>
</tbody>
</table>

### Manipulating Window Properties

Table 31 provides the subroutines for manipulating the properties of windows.

Table 31. Manipulating Window Properties

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XChangeProperty()</td>
<td>Changes the property for the specified window.</td>
</tr>
<tr>
<td>XDeleteProperty()</td>
<td>Deletes a property for the specified window.</td>
</tr>
<tr>
<td>XGetWindowProperty()</td>
<td>Gets the atom type and property format for the window.</td>
</tr>
<tr>
<td>XListProperties()</td>
<td>Gets the specified window’s property list.</td>
</tr>
<tr>
<td>XRotateWindowProperties()</td>
<td>Rotates the properties in a property array.</td>
</tr>
</tbody>
</table>

### Setting Window Selections

Table 32 provides the subroutines for setting window selections.

Table 32. Setting Window Selections

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XConvertSelection()</td>
<td>Converts a selection.</td>
</tr>
</tbody>
</table>
Table 32. Setting Window Selections (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XGetSelectionOwner()</td>
<td>Gets the selection owner.</td>
</tr>
<tr>
<td>XSetSelectionOwner()</td>
<td>Sets the selection owner.</td>
</tr>
</tbody>
</table>

**Manipulating Colormaps**

Table 33 provides the subroutines for manipulating color maps.

Table 33. Manipulating Colormaps

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAllocStandardColormap()</td>
<td>Allocates an XStandardColormap structure.</td>
</tr>
<tr>
<td>XCopyColormapAndFree()</td>
<td>Creates a new colormap from a specified colormap.</td>
</tr>
<tr>
<td>XCreateColormap()</td>
<td>Creates a colormap.</td>
</tr>
<tr>
<td>XFreeColormap()</td>
<td>Frees the specified colormap.</td>
</tr>
<tr>
<td>XQueryColor()</td>
<td>Queries the RGB value for a specified pixel.</td>
</tr>
<tr>
<td>XQueryColors()</td>
<td>Queries the RGB values for an array of pixels.</td>
</tr>
<tr>
<td>XSetWindowColormap()</td>
<td>Sets the colormap of the specified window.</td>
</tr>
</tbody>
</table>

**Manipulating Color Cells**

Table 34 provides the subroutines for manipulating color cells.

Table 34. Manipulating Color Cells

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAllocColor()</td>
<td>Allocates a read-only color cell.</td>
</tr>
<tr>
<td>XAllocColorCells()</td>
<td>Allocates read/write color cells.</td>
</tr>
<tr>
<td>XAllocColorPlanes()</td>
<td>Allocates read/write color resources.</td>
</tr>
<tr>
<td>XAllocNamedColor()</td>
<td>Allocates a read-only color cell by name.</td>
</tr>
<tr>
<td>XFreeColors()</td>
<td>Frees colormap cells.</td>
</tr>
<tr>
<td>XLookupColor()</td>
<td>Looks up a colormame.</td>
</tr>
<tr>
<td>XStoreColor()</td>
<td>Stores an RGB value into a single colormap cell.</td>
</tr>
<tr>
<td>XStoreColors()</td>
<td>Stores RGB values into colormap cells.</td>
</tr>
<tr>
<td>XStoreNamedColor()</td>
<td>Sets a pixel color to the named color.</td>
</tr>
</tbody>
</table>

**Creating and Freeing Pixmaps**

Table 35 provides the subroutines for creating and freeing pixmaps.

Table 35. Creating and Freeing Pixmaps

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCreatePixmap()</td>
<td>Creates a pixmap of a specified size.</td>
</tr>
<tr>
<td>XFreePixmap()</td>
<td>Frees all storage associated with specified pixmap.</td>
</tr>
</tbody>
</table>
# Manipulating Graphics Contexts

Table 36 provides the subroutines for manipulating graphics contexts.

## Table 36. Manipulating Graphics Contexts

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XChangeGC()</td>
<td>Changes the components in the specified Graphics Context (GC).</td>
</tr>
<tr>
<td>XCopyGC()</td>
<td>Copies the components from a source GC to a destination GC.</td>
</tr>
<tr>
<td>XCreateGC()</td>
<td>Creates a new GC.</td>
</tr>
<tr>
<td>XFreeGC()</td>
<td>Frees the specified GC.</td>
</tr>
<tr>
<td>XGetGCValues()</td>
<td>Returns the GC values in the specified structure.</td>
</tr>
<tr>
<td>XGContextFromGC()</td>
<td>Obtains the GContext resource ID for GC.</td>
</tr>
<tr>
<td>XQueryBestTile()</td>
<td>Gets the best fill tile shape.</td>
</tr>
<tr>
<td>XQueryBestSize()</td>
<td>Gets the best size tile, stipple, or cursor.</td>
</tr>
<tr>
<td>XQueryBestStipple()</td>
<td>Gets the best stipple shape.</td>
</tr>
<tr>
<td>XSetArcMode()</td>
<td>Sets the arc mode of the specified GC.</td>
</tr>
<tr>
<td>XSetBackground()</td>
<td>Sets the background of the specified GC.</td>
</tr>
<tr>
<td>XSetClipmask()</td>
<td>Sets the clip_mask of the specified GC to a specified pixmap.</td>
</tr>
<tr>
<td>XSetClipOrigin()</td>
<td>Sets the clip origin of the specified GC.</td>
</tr>
<tr>
<td>XSetClipRectangles()</td>
<td>Sets the clip_mask of GC to a list of rectangles.</td>
</tr>
<tr>
<td>XSetDashes()</td>
<td>Sets the dashed line style components of a specified GC.</td>
</tr>
<tr>
<td>XSetFillRule()</td>
<td>Sets the fill rule of the specified GC.</td>
</tr>
<tr>
<td>XSetFillStyle()</td>
<td>Sets the fill style of the specified GC.</td>
</tr>
<tr>
<td>XSetFont()</td>
<td>Sets the current font of the specified GC.</td>
</tr>
<tr>
<td>XSetForeground()</td>
<td>Sets the foreground of the specified GC.</td>
</tr>
<tr>
<td>XSetFunction()</td>
<td>Sets display function in the specified GC.</td>
</tr>
<tr>
<td>XSetGraphicsExposures()</td>
<td>Sets the graphics-exposure flag of the specified GC.</td>
</tr>
<tr>
<td>XSetLineAttributes()</td>
<td>Sets the line-drawing components of the GC.</td>
</tr>
<tr>
<td>XSetPlaneMask()</td>
<td>Sets the plane mask of the specified GC.</td>
</tr>
<tr>
<td>XSetState()</td>
<td>Sets the foreground, background, plane mask, and function in GC.</td>
</tr>
<tr>
<td>XSetStipple()</td>
<td>Sets the stipple of the specified GC.</td>
</tr>
<tr>
<td>XSetSubwindowMode()</td>
<td>Sets the subwindow mode of the specified GC.</td>
</tr>
<tr>
<td>XSetTile()</td>
<td>Sets the fill tile of the specified GC.</td>
</tr>
<tr>
<td>XSetTSoOrigin()</td>
<td>Sets the tile or stipple origin of the specified GC.</td>
</tr>
</tbody>
</table>
Clearing and Copying Areas

Table 37 provides the subroutines for clearing and copying areas.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XClearArea()</td>
<td>Clears a rectangular area of window.</td>
</tr>
<tr>
<td>XClearWindow()</td>
<td>Clears the entire window.</td>
</tr>
<tr>
<td>XCopyArea()</td>
<td>Copies the drawable area between drawables of the same root and the same depth.</td>
</tr>
<tr>
<td>XCopyPlane()</td>
<td>Copies single bit-plane of the drawable.</td>
</tr>
</tbody>
</table>

Drawing Lines

Table 38 provides the subroutines for drawing lines.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDraw()</td>
<td>Draws an arbitrary polygon or curve that is defined by the specified list of Vertexes as specified in vlist.</td>
</tr>
<tr>
<td>XDrawArc()</td>
<td>Draws a single arc in the drawable.</td>
</tr>
<tr>
<td>XDrawArcs()</td>
<td>Draws multiple arcs in a specified drawable.</td>
</tr>
<tr>
<td>XDrawFilled()</td>
<td>Draws arbitrary polygons or curves and then fills them.</td>
</tr>
<tr>
<td>XDrawLine()</td>
<td>Draws a single line between two points in a drawable.</td>
</tr>
<tr>
<td>XDrawLines()</td>
<td>Draws multiple lines in the specified drawable.</td>
</tr>
<tr>
<td>XDrawPoint()</td>
<td>Draws a single point in the specified drawable.</td>
</tr>
<tr>
<td>XDrawPoints()</td>
<td>Draws multiple points in the specified drawable.</td>
</tr>
<tr>
<td>XDrawRectangle()</td>
<td>Draws an outline of a single rectangle in the drawable.</td>
</tr>
<tr>
<td>XDrawRectangles()</td>
<td>Draws an outline of multiple rectangles in the drawable.</td>
</tr>
<tr>
<td>XDrawSegments()</td>
<td>Draws multiple line segments in the specified drawable.</td>
</tr>
</tbody>
</table>

Filling Areas

Table 39 provides the subroutines for filling areas.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XFillArc()</td>
<td>Fills a single arc in drawable.</td>
</tr>
<tr>
<td>XFillArcs()</td>
<td>Fills multiple arcs in drawable.</td>
</tr>
<tr>
<td>XFillPolygon()</td>
<td>Fills a polygon area in the drawable.</td>
</tr>
</tbody>
</table>
Table 39. Filling Areas (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XFillRectangle()</td>
<td>Fills a single rectangular area in the drawable.</td>
</tr>
<tr>
<td>XFillRectangles()</td>
<td>Fills multiple rectangular areas in the drawable.</td>
</tr>
</tbody>
</table>

Loading and Freeing Fonts

Table 40 provides the subroutines for loading and freeing fonts.

Table 40. Loading and Freeing Fonts

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XFreeFont()</td>
<td>Unloads the font and frees the storage used by the font.</td>
</tr>
<tr>
<td>XFreeFontInfo()</td>
<td>Frees the font information array.</td>
</tr>
<tr>
<td>XFreeFontNames()</td>
<td>Frees a font name array.</td>
</tr>
<tr>
<td>XFreeFontPath()</td>
<td>Frees data returned by XGetFontPath.</td>
</tr>
<tr>
<td>XGetFontPath()</td>
<td>Gets the current font search path.</td>
</tr>
<tr>
<td>XGetFontProperty()</td>
<td>Gets the specified font property.</td>
</tr>
<tr>
<td>XListFontsWithInfo()</td>
<td>Gets names and information about loaded fonts.</td>
</tr>
<tr>
<td>XLoadFont()</td>
<td>Loads a font.</td>
</tr>
<tr>
<td>XLoadQueryFont()</td>
<td>Loads and queries font in one operation.</td>
</tr>
<tr>
<td>XListFonts()</td>
<td>Gets a list of available font names.</td>
</tr>
<tr>
<td>XQueryFont()</td>
<td>Gets information about a loaded font.</td>
</tr>
<tr>
<td>XSetFontPath()</td>
<td>Sets the font search path.</td>
</tr>
<tr>
<td>XUnloadFont()</td>
<td>Unloads the specified font.</td>
</tr>
</tbody>
</table>

Querying Character String Sizes

Table 41 provides the subroutines for querying the character size of a string.

Table 41. Querying Character String Sizes

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XFreeStringList()</td>
<td>Frees the in-memory data associated with the specified string list.</td>
</tr>
<tr>
<td>XQueryTextExtents()</td>
<td>Gets a 1-byte character string bounding box from the server.</td>
</tr>
<tr>
<td>XQueryTextExtents16()</td>
<td>Gets a 2-byte character string bounding box from the server.</td>
</tr>
<tr>
<td>XTextExtents()</td>
<td>Gets a bounding box of a 1-byte character string.</td>
</tr>
<tr>
<td>XTextExtents16()</td>
<td>Gets a bounding box of a 2-byte character string.</td>
</tr>
<tr>
<td>XTextPropertyToStringList()</td>
<td>Returns a list of strings representing the elements of the specified XTextProperty structure.</td>
</tr>
</tbody>
</table>
Table 41. Querying Character String Sizes (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XTextWidth()</td>
<td>Gets the width of an 8-bit character string.</td>
</tr>
<tr>
<td>XTextWidth16()</td>
<td>Gets the width of a 2-byte character string.</td>
</tr>
</tbody>
</table>

**Drawing Text**

Table 42 provides the subroutines for drawing text.

Table 42. Drawing Text

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDrawImageString()</td>
<td>Draws 8-bit image text in the specified drawable.</td>
</tr>
<tr>
<td>XDrawImageString16()</td>
<td>Draws 2-byte image text in the specified drawable.</td>
</tr>
<tr>
<td>XDrawString()</td>
<td>Draws 8-bit text in the specified drawable.</td>
</tr>
<tr>
<td>XDrawString16()</td>
<td>Draws 2-byte text in the specified drawable.</td>
</tr>
<tr>
<td>XDrawText()</td>
<td>Draws 8-bit complex text in the specified drawable.</td>
</tr>
<tr>
<td>XDrawText16()</td>
<td>Draws 2-byte complex text in the specified drawable.</td>
</tr>
</tbody>
</table>

**Transferring Images**

Table 43 provides the subroutines for transferring images.

Table 43. Transferring Images

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XGetImage()</td>
<td>Gets the image from the rectangle in the drawable.</td>
</tr>
<tr>
<td>XGetSubImage()</td>
<td>Copies the rectangle on the display to image.</td>
</tr>
<tr>
<td>XPutImage()</td>
<td>Puts the image from memory into the rectangle in the drawable.</td>
</tr>
</tbody>
</table>

**Manipulating Cursors**

Table 44 provides the subroutines for manipulating cursors.

Table 44. Manipulating Cursors

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCreateFontCursor()</td>
<td>Creates a cursor from a standard font.</td>
</tr>
<tr>
<td>XCreateGlyphCursor()</td>
<td>Creates a cursor from font glyphs.</td>
</tr>
<tr>
<td>XDefineCursor()</td>
<td>Defines a cursor for a window.</td>
</tr>
<tr>
<td>XFreeCursor()</td>
<td>Frees a cursor.</td>
</tr>
<tr>
<td>XQueryBestCursor()</td>
<td>Gets useful cursor sizes.</td>
</tr>
<tr>
<td>XRecolorCursor()</td>
<td>Changes the color of a cursor.</td>
</tr>
<tr>
<td>XUndefineCursor()</td>
<td>Undefines a cursor for a window.</td>
</tr>
</tbody>
</table>
### Handling Window Manager Functions

Table 45 provides the subroutines for handling the window manager functions.

**Table 45. Handling Window Manager Functions**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAddToSaveSet()</td>
<td>Adds a window to the client's save-set.</td>
</tr>
<tr>
<td>XAllowEvents()</td>
<td>Allows events to be processed after a device is frozen.</td>
</tr>
<tr>
<td>XChangeActivePointerGrab()</td>
<td>Changes the active pointer grab.</td>
</tr>
<tr>
<td>XChangePointerControl()</td>
<td>Changes the interactive feel of the pointer device.</td>
</tr>
<tr>
<td>XChangeSaveSet()</td>
<td>Adds or removes a window from the client's save-set.</td>
</tr>
<tr>
<td>XGetInputFocus()</td>
<td>Gets the current input focus.</td>
</tr>
<tr>
<td>XGetPointerControl()</td>
<td>Gets the current pointer parameters.</td>
</tr>
<tr>
<td>XGrabButton()</td>
<td>Grabs a mouse button.</td>
</tr>
<tr>
<td>XGrabKey()</td>
<td>Grabs a single key of the keyboard.</td>
</tr>
<tr>
<td>XGrabKeyboard()</td>
<td>Grabs the keyboard.</td>
</tr>
<tr>
<td>XGrabPointer()</td>
<td>Grabs the pointer.</td>
</tr>
<tr>
<td>XGrabServer()</td>
<td>Grabs the server.</td>
</tr>
<tr>
<td>XInstallColormap()</td>
<td>Installs a colormap.</td>
</tr>
<tr>
<td>XKillClient()</td>
<td>Removes a client.</td>
</tr>
<tr>
<td>XListInstalledColormaps()</td>
<td>Gets a list of currently installed colormaps.</td>
</tr>
<tr>
<td>XRemoveFromSaveSet()</td>
<td>Removes a window from the client's save-set.</td>
</tr>
<tr>
<td>XReparentWindow()</td>
<td>Changes the parent of a window.</td>
</tr>
<tr>
<td>XSetCloseDownMode()</td>
<td>Changes the close down mode.</td>
</tr>
<tr>
<td>XSetInputFocus()</td>
<td>Sets the input focus.</td>
</tr>
<tr>
<td>XUngrabButton()</td>
<td>Ungrabs a mouse button.</td>
</tr>
<tr>
<td>XUngrabKey()</td>
<td>Ungrabs a key.</td>
</tr>
<tr>
<td>XUngrabKeyboard()</td>
<td>Ungrabs the keyboard.</td>
</tr>
<tr>
<td>XUngrabPointer()</td>
<td>Ungrabs the pointer.</td>
</tr>
<tr>
<td>XUngrabServer()</td>
<td>Ungrabs the server.</td>
</tr>
<tr>
<td>XUninstallColormap()</td>
<td>Uninstalls a colormap.</td>
</tr>
<tr>
<td>XWarpPointer()</td>
<td>Moves the pointer to arbitrary point on the screen.</td>
</tr>
</tbody>
</table>

### Manipulating Keyboard Settings

Table 46 provides the subroutines for manipulating keyboard settings.

**Table 46. Manipulating Keyboard Settings**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAutoRepeatOff()</td>
<td>Turns off the keyboard auto-repeat.</td>
</tr>
<tr>
<td>XAutoRepeatOn()</td>
<td>Turns on the keyboard auto-repeat.</td>
</tr>
<tr>
<td>XBell()</td>
<td>Sets the volume of the bell.</td>
</tr>
</tbody>
</table>
Table 46. Manipulating Keyboard Settings (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XChangeKeyboardControl()</td>
<td>Changes the keyboard settings.</td>
</tr>
<tr>
<td>XChangeKeyboardMapping()</td>
<td>Changes the mapping of symbols to keycodes.</td>
</tr>
<tr>
<td>XDeleteModifiermapEntry()</td>
<td>Deletes an entry from the XModifierKeymap structure.</td>
</tr>
<tr>
<td>XFreeModifiermap()</td>
<td>Frees XModifierKeymap structure.</td>
</tr>
<tr>
<td>XGetKeyboardControl()</td>
<td>Gets the current keyboard settings.</td>
</tr>
<tr>
<td>XGetKeyboardMapping()</td>
<td>Gets the mapping of symbols to keycodes.</td>
</tr>
<tr>
<td>XGetModifierMapping()</td>
<td>Gets keycodes to be modifiers.</td>
</tr>
<tr>
<td>XGetPointerMapping()</td>
<td>Gets the mapping of buttons on the pointer.</td>
</tr>
<tr>
<td>XInsertModifiermapEntry()</td>
<td>Adds an entry to the XModifierKeymap structure.</td>
</tr>
<tr>
<td>XNewModifiermap()</td>
<td>Creates the XModifierKeymap structure.</td>
</tr>
<tr>
<td>XQueryKeymap()</td>
<td>Gets the state of the keyboard keys.</td>
</tr>
<tr>
<td>XSetPointerMapping()</td>
<td>Sets the mapping of buttons on the pointer.</td>
</tr>
<tr>
<td>XSetModifierMapping()</td>
<td>Sets keycodes to be modifiers.</td>
</tr>
</tbody>
</table>

Controlling the Screen Saver

Table 47 provides the subroutines for controlling the screen saver.

Table 47. Controlling the Screen Saver

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XActivateScreenSaver()</td>
<td>Activates the screen saver.</td>
</tr>
<tr>
<td>XForceScreenSaver()</td>
<td>Turns the screen saver on or off.</td>
</tr>
<tr>
<td>XGetScreenSaver()</td>
<td>Gets the current screen saver settings.</td>
</tr>
<tr>
<td>XResetScreenSaver()</td>
<td>Resets the screen saver.</td>
</tr>
<tr>
<td>XSetScreenSaver()</td>
<td>Sets the screen saver.</td>
</tr>
</tbody>
</table>

Manipulating Hosts and Access Control

Table 48 provides the subroutines for manipulating hosts and toggling the access control.

Table 48. Manipulating Hosts and Access Control

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDisableAccessControl()</td>
<td>Disables access control.</td>
</tr>
<tr>
<td>XEnableAccessControl()</td>
<td>Enables access control.</td>
</tr>
<tr>
<td>XListHosts()</td>
<td>Gets the list of hosts.</td>
</tr>
<tr>
<td>XSetAccessControl()</td>
<td>Changes access control.</td>
</tr>
</tbody>
</table>
**Handling Events**

Table 49 provides the subroutines for handling events.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCheckIfEvent()</td>
<td>Checks event queue for the specified event without blocking.</td>
</tr>
<tr>
<td>XCheckMaskEvent()</td>
<td>Removes the next event that matches a specified mask without blocking.</td>
</tr>
<tr>
<td>XCheckTypedEvent()</td>
<td>Gets the next event that matches event type.</td>
</tr>
<tr>
<td>XCheckTypedWindowEvent()</td>
<td>Gets the next event for the specified window.</td>
</tr>
<tr>
<td>XCheckWindowEvent()</td>
<td>Removes the next event that matches the specified window and mask without blocking.</td>
</tr>
<tr>
<td>XEventsQueued()</td>
<td>Checks the number of events in the event queue.</td>
</tr>
<tr>
<td>XFlush()</td>
<td>Flushes the output buffer.</td>
</tr>
<tr>
<td>XGetMotionEvents()</td>
<td>Gets the motion history for the specified window.</td>
</tr>
<tr>
<td>XIfEvent()</td>
<td>Checks the event queue for the specified event and removes it.</td>
</tr>
<tr>
<td>XMaskEvent()</td>
<td>Removes the next event that matches a specified mask.</td>
</tr>
<tr>
<td>XNextEvent()</td>
<td>Gets the next event and removes it from the queue.</td>
</tr>
<tr>
<td>XPeekEvent()</td>
<td>Peeks at the event queue.</td>
</tr>
<tr>
<td>XPeekIfEvent()</td>
<td>Checks the event queue for the specified event.</td>
</tr>
<tr>
<td>XPending()</td>
<td>Returns the number of events that are pending.</td>
</tr>
<tr>
<td>XPutBackEvent()</td>
<td>Pushes the event back to the top of the event queue.</td>
</tr>
<tr>
<td>XSelectInput()</td>
<td>Selects events to be reported to the client.</td>
</tr>
<tr>
<td>XSendEvent()</td>
<td>Sends an event to a specified window.</td>
</tr>
<tr>
<td>XSync()</td>
<td>Flushes the output buffer and waits until all requests are completed.</td>
</tr>
<tr>
<td>XWindowEvent()</td>
<td>Removes the next event that matches the specified window and mask.</td>
</tr>
</tbody>
</table>

**Enabling and Disabling Synchronization**

Table 50 provides the subroutines for toggling synchronization.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSetAfterFunction()</td>
<td>Sets the previous after function.</td>
</tr>
<tr>
<td>XSynchronize()</td>
<td>Enables or disables synchronization.</td>
</tr>
</tbody>
</table>
## Using Default Error Handling

Table 51 provides the subroutines for using the default error handling.

**Table 51. Using Default Error Handling**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDisplayName()</td>
<td>Gets the name of the display currently being used.</td>
</tr>
<tr>
<td>XGetErrorText()</td>
<td>Gets the error text for the specified error code.</td>
</tr>
<tr>
<td>XGetErrorDatabaseText()</td>
<td>Gets the error text from the error database.</td>
</tr>
<tr>
<td>XSetErrorHandler()</td>
<td>Sets the error handler.</td>
</tr>
<tr>
<td>XSetIOErrorHandler()</td>
<td>Sets the error handler for unrecoverable I/O errors.</td>
</tr>
</tbody>
</table>

## Communicating with Window Managers

Table 52 provides the subroutines for communicating with window managers.

**Table 52. Communicating with Window Managers**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAllocClassHints()</td>
<td>Allocates storage for an XClassHint structure.</td>
</tr>
<tr>
<td>XAllocIconSize()</td>
<td>Allocates storage for an XIconSize structure.</td>
</tr>
<tr>
<td>XAllocSizeHints()</td>
<td>Allocates storage for an XSizeHints structure.</td>
</tr>
<tr>
<td>XAllocWMHints()</td>
<td>Allocates storage for an XWMHints structure.</td>
</tr>
<tr>
<td>XGetClassHint()</td>
<td>Gets the class of a window.</td>
</tr>
<tr>
<td>XFetchName()</td>
<td>Gets the name of a window.</td>
</tr>
<tr>
<td>XGetCommand()</td>
<td>Gets a window’s WM_COMMAND property.</td>
</tr>
<tr>
<td>XGetIconName()</td>
<td>Gets the name of an icon window.</td>
</tr>
<tr>
<td>XGetIconSizes()</td>
<td>Gets the values of icon size atom.</td>
</tr>
<tr>
<td>XGetNormalHints()</td>
<td>Gets size hints for window in normal state.</td>
</tr>
<tr>
<td>XGetRGBColormaps()</td>
<td>Gets colormap associated with specified atom.</td>
</tr>
<tr>
<td>XGetSizeHints()</td>
<td>Gets the values of type WM_SIZE_HINTS properties.</td>
</tr>
<tr>
<td>XGetStandardColormap()</td>
<td>Gets colormap associated with specified atom.</td>
</tr>
<tr>
<td>XGetTextProperty()</td>
<td>Gets a window’s property of type TEXT.</td>
</tr>
<tr>
<td>XGetTransientForHint()</td>
<td>Gets WM_TRANSIENT_FOR property for window.</td>
</tr>
<tr>
<td>XGetWMClientMachine()</td>
<td>Gets the value of a window’s WM_CLIENT_MACHINE property.</td>
</tr>
<tr>
<td>XGetWMColormapWindows()</td>
<td>Gets the value of a window’s WM_COLORMAP_WINDOWS property.</td>
</tr>
<tr>
<td>XGetWMHints()</td>
<td>Gets the value of the window manager’s hints atom.</td>
</tr>
<tr>
<td>XGetWMName()</td>
<td>Gets the value of the WM_NAME property.</td>
</tr>
<tr>
<td>XGetWMIconName()</td>
<td>Gets the value of the WM_ICON_NAME property.</td>
</tr>
</tbody>
</table>
### Table 52. Communicating with Window Managers (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XGetWMNormalHints()</td>
<td>Gets the value of the window manager’s hints atom.</td>
</tr>
<tr>
<td>XGetWMProtocols()</td>
<td>Gets the value of a window’s WM_PROTOCOLS property.</td>
</tr>
<tr>
<td>XGetWMSIZEHints()</td>
<td>Gets the values of type WM_SIZE_HINTS properties.</td>
</tr>
<tr>
<td>XGetZoomHints()</td>
<td>Gets values of the zoom hints atom.</td>
</tr>
<tr>
<td>XSetCommand()</td>
<td>Sets the value of the command atom.</td>
</tr>
<tr>
<td>XSetClassHint()</td>
<td>Sets the class of a window.</td>
</tr>
<tr>
<td>XSetIconName()</td>
<td>Assigns a name to an icon window.</td>
</tr>
<tr>
<td>XSetIconSizes()</td>
<td>Sets the values of icon size atom.</td>
</tr>
<tr>
<td>XSetNormalHints()</td>
<td>Sets size hints for a window in normal state.</td>
</tr>
<tr>
<td>XSetRGBColormaps()</td>
<td>Sets the colormap associated with the specified atom.</td>
</tr>
<tr>
<td>XSetSizeHints()</td>
<td>Sets the values of the type WM_SIZE_HINTS properties.</td>
</tr>
<tr>
<td>XSetStandardColormap()</td>
<td>Sets the colormap associated with the specified atom.</td>
</tr>
<tr>
<td>XSetStandardProperties()</td>
<td>Specifies a minimum set of properties.</td>
</tr>
<tr>
<td>XSetTextProperty()</td>
<td>Sets a window’s properties of type TEXT.</td>
</tr>
<tr>
<td>XSetTransientForHint()</td>
<td>Sets WM_TRANSIENT_FOR property for window.</td>
</tr>
<tr>
<td>XSetWMClientMachine()</td>
<td>Sets a window’s WM_CLIENT_MACHINE property.</td>
</tr>
<tr>
<td>XSetWMColormapWindows()</td>
<td>Sets a window’s WM_COLOMAP_WINDOW property.</td>
</tr>
<tr>
<td>XSetWMHints()</td>
<td>Sets the value of the window manager’s hints atom.</td>
</tr>
<tr>
<td>XSetWMIconName()</td>
<td>Sets the value of the WM_ICON_NAME property.</td>
</tr>
<tr>
<td>XSetWMName()</td>
<td>Sets the value of the WM_NAME property.</td>
</tr>
<tr>
<td>XSetWMNormalHints()</td>
<td>Sets the value of the window manager’s hints atom.</td>
</tr>
<tr>
<td>XSetWMProperties()</td>
<td>Sets the values of properties for a window manager.</td>
</tr>
<tr>
<td>XSetWMProtocols()</td>
<td>Sets the value of the WM_PROTOCOLS property.</td>
</tr>
<tr>
<td>XSetWMSIZEHints()</td>
<td>Sets the values of type WM_SIZE_HINTS properties.</td>
</tr>
<tr>
<td>XSetZoomHints()</td>
<td>Sets the values of the zoom hints atom.</td>
</tr>
<tr>
<td>XStoreName()</td>
<td>Assigns a name to a window.</td>
</tr>
</tbody>
</table>

### Manipulating Keyboard Event Functions

Table 53 on page 219 provides the subroutines for manipulating the functions of keyboard events.
### Table 53. Keyboard Event Functions

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XKeycodeToKeysym()</td>
<td>Converts a keycode to a keysym value.</td>
</tr>
<tr>
<td>XKeysymToKeycode()</td>
<td>Converts a keysym value to keycode.</td>
</tr>
<tr>
<td>XKeysymToString()</td>
<td>Converts a keysym value to keysym name.</td>
</tr>
<tr>
<td>XLookupKeysym()</td>
<td>Translates a keyboard event into a keysym value.</td>
</tr>
<tr>
<td>XLookupMapping()</td>
<td>Gets the mapping of a keyboard event from a keymap file.</td>
</tr>
<tr>
<td>XLookupString()</td>
<td>Translates the keyboard event into a character string.</td>
</tr>
<tr>
<td>XRebindCode()</td>
<td>Changes the keyboard mapping in the keymap file.</td>
</tr>
<tr>
<td>XRebindKeysym()</td>
<td>Maps the character string to a specified keysym and modifiers.</td>
</tr>
<tr>
<td>XRefreshKeyboardMapping()</td>
<td>Refreshes the stored modifier and keymap information.</td>
</tr>
<tr>
<td>XStringToKeysym()</td>
<td>Converts the keysym name to the keysym value.</td>
</tr>
<tr>
<td>XUseKeymap()</td>
<td>Changes the keymap files.</td>
</tr>
<tr>
<td>XGeometry()</td>
<td>Parses window geometry given padding and font values.</td>
</tr>
<tr>
<td>XGetDefault()</td>
<td>Gets the default window options.</td>
</tr>
<tr>
<td>XParseColor()</td>
<td>Obtains RGB values from color name.</td>
</tr>
<tr>
<td>XParseGeometry()</td>
<td>Parses standard window geometry options.</td>
</tr>
<tr>
<td>XWMGeometry()</td>
<td>Obtains a window’s geometry information.</td>
</tr>
</tbody>
</table>

### Manipulating Regions

Table 54 provides the subroutines for manipulating regions.

### Table 54. Manipulating Regions

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XClipBox()</td>
<td>Generates the smallest enclosing rectangle in the region.</td>
</tr>
<tr>
<td>XCreateRegion()</td>
<td>Creates a new empty region.</td>
</tr>
<tr>
<td>XEmptyRegion()</td>
<td>Determines whether a specified region is empty.</td>
</tr>
<tr>
<td>XEqualRegion()</td>
<td>Determines whether two regions are the same.</td>
</tr>
<tr>
<td>XIntersectRegion()</td>
<td>Computes the intersection of two regions.</td>
</tr>
<tr>
<td>XDestroyRegion()</td>
<td>Frees storage associated with the specified region.</td>
</tr>
<tr>
<td>XOffsetRegion()</td>
<td>Moves the specified region by the specified amount.</td>
</tr>
<tr>
<td>XPointInRegion()</td>
<td>Determines if a point lies in the specified region.</td>
</tr>
<tr>
<td>XPolygonRegion()</td>
<td>Generates a region from points.</td>
</tr>
</tbody>
</table>
### Table 54. Manipulating Regions (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRectInRegion()</td>
<td>Determines if a rectangle lies in the specified region.</td>
</tr>
<tr>
<td>XSetRegion()</td>
<td>Sets the GC to the specified region.</td>
</tr>
<tr>
<td>XShrinkRegion()</td>
<td>Reduces the specified region by a specified amount.</td>
</tr>
<tr>
<td>XSubtractRegion()</td>
<td>Subtracts two regions.</td>
</tr>
<tr>
<td>XUnionRegion()</td>
<td>Computes the union of two regions.</td>
</tr>
<tr>
<td>XUnionRectWithRegion()</td>
<td>Creates a union of source region and rectangle.</td>
</tr>
<tr>
<td>XXorRegion()</td>
<td>Gets the difference between the union and intersection of regions.</td>
</tr>
</tbody>
</table>

### Using Cut and Paste Buffers

Table 55 provides the subroutines for using cut and paste buffers.

**Table 55. Using Cut and Paste Buffers**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XFetchBuffer()</td>
<td>Gets data from a specified cut buffer.</td>
</tr>
<tr>
<td>XFetchBytes()</td>
<td>Gets data from the first cut buffer.</td>
</tr>
<tr>
<td>XRotateBuffers()</td>
<td>Rotates the cut buffers.</td>
</tr>
<tr>
<td>XStoreBuffer()</td>
<td>Stores data in a specified cut buffer.</td>
</tr>
<tr>
<td>XStoreBytes()</td>
<td>Stores data in first cut buffer.</td>
</tr>
</tbody>
</table>

### Querying Visual Types

Table 56 provides the subroutines for querying visual types.

**Table 56. Querying Visual Types**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XGetVisualInfo()</td>
<td>Gets a list of visual information structures.</td>
</tr>
<tr>
<td>XListDepths()</td>
<td>Determines the number of depths that are available on a given screen.</td>
</tr>
<tr>
<td>XListPixmapFormats()</td>
<td>Gets the pixmap format information for a given display.</td>
</tr>
<tr>
<td>XMatchVisualInfo()</td>
<td>Gets visual information matching screen depth and class.</td>
</tr>
<tr>
<td>XPixmapFormatValues()</td>
<td>Gets the pixmap format information for a given display.</td>
</tr>
</tbody>
</table>
Manipulating Images

Table 57 provides the subroutines for manipulating images.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAddPixel()</td>
<td>Increases each pixel in a pixmap by a constant value.</td>
</tr>
<tr>
<td>XCreateImage()</td>
<td>Allocates memory for the XImage structure.</td>
</tr>
<tr>
<td>XDestroyImage()</td>
<td>Frees memory for the XImage structure.</td>
</tr>
<tr>
<td>XGetPixel()</td>
<td>Gets a pixel value in an image.</td>
</tr>
<tr>
<td>XPutPixel()</td>
<td>Sets a pixel value in an image.</td>
</tr>
<tr>
<td>XSubImage()</td>
<td>Creates an image that is a subsection of a specified image.</td>
</tr>
</tbody>
</table>

Manipulating Bitmaps

Table 58 provides the subroutines for manipulating bitmaps.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCreateBitmapFromData()</td>
<td>Includes a bitmap in the C program.</td>
</tr>
<tr>
<td>XCreatePixmapFromBitmapData()</td>
<td>Creates a pixmap using bitmap data.</td>
</tr>
<tr>
<td>XDeleteContext()</td>
<td>Deletes data associated with the window and context type.</td>
</tr>
<tr>
<td>XFindContext()</td>
<td>Gets data associated with the window and context type.</td>
</tr>
<tr>
<td>XReadBitmapFile()</td>
<td>Reads in a bitmap from a file.</td>
</tr>
<tr>
<td>XSaveContext()</td>
<td>Stores data associated with the window and context type.</td>
</tr>
<tr>
<td>XUniqueContext()</td>
<td>Allocates a new context.</td>
</tr>
<tr>
<td>XWriteBitmapFile()</td>
<td>Writes out a bitmap to a file.</td>
</tr>
</tbody>
</table>

Using the Resource Manager

Table 59 provides the subroutines for using the resource manager.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xpermalloc()</td>
<td>Allocates memory that is never freed.</td>
</tr>
<tr>
<td>XrmDestroyDatabase()</td>
<td>Destroys a resource database and frees its allocated memory.</td>
</tr>
<tr>
<td>XrmGetFileDatabase()</td>
<td>Creates a database from a specified file.</td>
</tr>
<tr>
<td>XrmGetResource()</td>
<td>Retrieves a resource from a database.</td>
</tr>
<tr>
<td>XrmGetStringDatabase()</td>
<td>Creates a database from a specified string.</td>
</tr>
<tr>
<td>XrmInitialize()</td>
<td>Initializes the resource manager.</td>
</tr>
<tr>
<td>XrmMergeDatabases()</td>
<td>Merges two databases.</td>
</tr>
<tr>
<td>XrmParseCommand()</td>
<td>Stores command options in a database.</td>
</tr>
<tr>
<td>XrmPutFileDatabase()</td>
<td>Copies the database into a specified file.</td>
</tr>
</tbody>
</table>
Table 59. Using the Resource Manager (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XrmPutLineResource()</td>
<td>Stores a single resource entry in a database.</td>
</tr>
<tr>
<td>XrmPutResource()</td>
<td>Stores a resource in a database.</td>
</tr>
<tr>
<td>XrmPutStringResource()</td>
<td>Stores string resource in a database.</td>
</tr>
<tr>
<td>XrmQGetResource()</td>
<td>Retrieves a quark from a database.</td>
</tr>
<tr>
<td>XrmQGetSearchList()</td>
<td>Gets a resource search list of database levels.</td>
</tr>
<tr>
<td>XrmQGetSearchResource()</td>
<td>Gets a quark search list of database levels.</td>
</tr>
<tr>
<td>XrmQPutResource()</td>
<td>Stores binding and quarks in a database.</td>
</tr>
<tr>
<td>XrmQPutStringResource()</td>
<td>Stores string binding and quarks in a database.</td>
</tr>
<tr>
<td>XrmQuarkToString()</td>
<td>Converts a quark to a character string.</td>
</tr>
<tr>
<td>XrmStringToQuark()</td>
<td>Converts a character string to a quark.</td>
</tr>
<tr>
<td>XrmStringToQuarkList()</td>
<td>Converts character strings to a quark list.</td>
</tr>
<tr>
<td>XrmStringToBindingQuarkList()</td>
<td>Converts strings to bindings and quarks.</td>
</tr>
<tr>
<td>XrmUniqueQuark()</td>
<td>Allocates a new quark.</td>
</tr>
</tbody>
</table>

Manipulating Display Functions

Table 60 provides the subroutines for manipulating the display functions.

Table 60. Display Functions

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllPlanes() XAllPlanes()</td>
<td>Returns all bits suitable for use in plane argument.</td>
</tr>
<tr>
<td>BitMapBitOrder() XBitMapOrder()</td>
<td>Returns either the most or least significant bit in each bitmap unit.</td>
</tr>
<tr>
<td>BitMapPad() XBitMapPad()</td>
<td>Returns the multiple of bits padding each scanline.</td>
</tr>
<tr>
<td>BitMapUnit() XBitMapUnit()</td>
<td>Returns the size of a bitmap’s unit in bits.</td>
</tr>
<tr>
<td>BlackPixel() XBlackPixel()</td>
<td>Returns the black pixel value of the screen specified.</td>
</tr>
<tr>
<td>BlackPixelOfScreen() XBlackPixelOfScreen()</td>
<td>Returns the black pixel value of the screen specified.</td>
</tr>
<tr>
<td>CellsOfScreen() XCellsOfScreen()</td>
<td>Returns the number of colormap cells.</td>
</tr>
<tr>
<td>ConnectionNumber() XConnectionNumber()</td>
<td>Returns the file descriptor of the connection.</td>
</tr>
<tr>
<td>CreatePixmapCursor() XCreatePixmapCursor()</td>
<td>Creates a pixmap of a specified size.</td>
</tr>
<tr>
<td>CreateWindow() XCreateWindow()</td>
<td>Creates an unmapped subwindow for a specified parent window.</td>
</tr>
<tr>
<td>DefaultColormap() XDefaultColormap()</td>
<td>Returns a default colormap ID for allocation on the screen specified.</td>
</tr>
<tr>
<td>DefaultColormapOfScreen() XDefaultColormapOfScreen</td>
<td>Returns the default colormap ID of the screen specified.</td>
</tr>
<tr>
<td>DefaultDepth() XDefaultDepth()</td>
<td>Returns the depth of the default root window.</td>
</tr>
<tr>
<td>Subroutine</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DefaultDepthOfScreen()</td>
<td>Returns the default depth of the screen specified.</td>
</tr>
<tr>
<td>XDefaultDepthOfScreen()</td>
<td></td>
</tr>
<tr>
<td>DefaultGC()</td>
<td>Returns the default GC of the default root window.</td>
</tr>
<tr>
<td>XDefaultGC()</td>
<td></td>
</tr>
<tr>
<td>DefaultGCOfScreen()</td>
<td>Returns the default GC of the screen specified.</td>
</tr>
<tr>
<td>XDefaultGCOfScreen()</td>
<td></td>
</tr>
<tr>
<td>DefaultScreen()</td>
<td>Obtains the default screen referenced in the XOpenDisplay routine.</td>
</tr>
<tr>
<td>XDefaultScreen()</td>
<td></td>
</tr>
<tr>
<td>DefaultScreenofDisplay()</td>
<td>Returns the default screen of the display specified.</td>
</tr>
<tr>
<td>XDefaultScreenofDisplay()</td>
<td></td>
</tr>
<tr>
<td>DefaultRootWindow()</td>
<td>Obtains the root window for the default screen specified.</td>
</tr>
<tr>
<td>XDefaultRootWindow()</td>
<td></td>
</tr>
<tr>
<td>DefaultVisual()</td>
<td>Returns the default visual type of the screen specified.</td>
</tr>
<tr>
<td>XDefaultVisual()</td>
<td></td>
</tr>
<tr>
<td>DefaultVisualOfScreen()</td>
<td>Returns the default visual type of the screen specified.</td>
</tr>
<tr>
<td>XDefaultVisualOfScreen()</td>
<td></td>
</tr>
<tr>
<td>DisplayCells()</td>
<td>Displays the number of entries in the default colormap.</td>
</tr>
<tr>
<td>XDisplayCells()</td>
<td></td>
</tr>
<tr>
<td>DisplayHeight()</td>
<td>Displays the height of the screen in pixels.</td>
</tr>
<tr>
<td>XDisplayHeight()</td>
<td></td>
</tr>
<tr>
<td>DisplayHeightMM()</td>
<td>Displays the height of the screen in millimeters.</td>
</tr>
<tr>
<td>XDisplayHeightMM()</td>
<td></td>
</tr>
<tr>
<td>DisplayOfScreen()</td>
<td>Displays the type of screen specified.</td>
</tr>
<tr>
<td>XDisplayOfScreen()</td>
<td></td>
</tr>
<tr>
<td>DisplayPlanes()</td>
<td>Displays the depth (number of planes) of the root window of the screen specified.</td>
</tr>
<tr>
<td>XDisplayPlanes()</td>
<td></td>
</tr>
<tr>
<td>DisplayString()</td>
<td>Displays the string passed to XOpenDisplay when the current display was opened.</td>
</tr>
<tr>
<td>XDisplayString()</td>
<td></td>
</tr>
<tr>
<td>DisplayWidth()</td>
<td>Displays the width of the specified screen in pixels.</td>
</tr>
<tr>
<td>XDisplayWidth()</td>
<td></td>
</tr>
<tr>
<td>DisplayWidthMM()</td>
<td>Displays the width of the specified screen in millimeters.</td>
</tr>
<tr>
<td>XDisplayWidthMM()</td>
<td></td>
</tr>
<tr>
<td>DoesBackingStore()</td>
<td>Indicates whether the specified screen supports backing stores.</td>
</tr>
<tr>
<td>XDoesBackingStore()</td>
<td></td>
</tr>
<tr>
<td>DoesSaveUnders()</td>
<td>Indicates whether the specified screen supports save unders.</td>
</tr>
<tr>
<td>XDoesSaveUnders()</td>
<td></td>
</tr>
<tr>
<td>EventMaskOfScreen()</td>
<td>Returns the initial root event mask for a specified screen.</td>
</tr>
<tr>
<td>XEventMaskOfScreen()</td>
<td></td>
</tr>
<tr>
<td>HeightMMOfScreen()</td>
<td>Returns the height of a specified screen in millimeters.</td>
</tr>
<tr>
<td>XHeightMMOfScreen()</td>
<td></td>
</tr>
<tr>
<td>HeightOfScreen()</td>
<td>Returns the height of a specified screen in pixels.</td>
</tr>
<tr>
<td>XHeightOfScreen()</td>
<td></td>
</tr>
<tr>
<td>ImageByteOrder()</td>
<td>Specifies the required byte order for each scanline unit of an image.</td>
</tr>
<tr>
<td>XImageByteOrder()</td>
<td></td>
</tr>
<tr>
<td>IsCursorKey()</td>
<td>Returns TRUE if keysym is on cursor key.</td>
</tr>
<tr>
<td>XIsCursorKey()</td>
<td></td>
</tr>
<tr>
<td>IsFunctionKey()</td>
<td>Returns TRUE if keysym is on function keys.</td>
</tr>
<tr>
<td>XIsFunctionKey()</td>
<td></td>
</tr>
<tr>
<td>IsKeypadKey()</td>
<td>Returns TRUE if keysym is on keypad.</td>
</tr>
<tr>
<td>XIsKeypadKey()</td>
<td></td>
</tr>
</tbody>
</table>
### Table 60. Display Functions (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsMiscFunctionKey()</td>
<td>Returns TRUE if keysym is on miscellaneous function keys.</td>
</tr>
<tr>
<td>IsModifierKey()</td>
<td>Returns TRUE if keysym is on modifier keys.</td>
</tr>
<tr>
<td>IsPFKey()</td>
<td>Returns TRUE if keysym is on PF keys.</td>
</tr>
<tr>
<td>LastKnownRequestProcessed()</td>
<td>Extracts the full serial number of the last known request processed by the X server.</td>
</tr>
<tr>
<td>XLastKnownRequestProcessed()</td>
<td></td>
</tr>
<tr>
<td>MaxCmapsOfScreen() XMaxCmapsOfScreen()</td>
<td>Returns the maximum number of colormaps supported by the specified screen.</td>
</tr>
<tr>
<td>MinCmapsOfScreen() XMinCmapsOfScreen()</td>
<td>Returns the minimum number of colormaps supported by the specified screen.</td>
</tr>
<tr>
<td>NextRequest() XNextRequest()</td>
<td>Extracts the full serial number to be used for the next request to be processed by the X Server.</td>
</tr>
<tr>
<td>PlanesOfScreen() XPlanesOfScreen()</td>
<td>Returns the depth (number of planes) in a specified screen.</td>
</tr>
<tr>
<td>ProtocolRevision() XProtocolRevision()</td>
<td>Returns the minor protocol revision number (zero) of the X server associated with the display.</td>
</tr>
<tr>
<td>ProtocolVersion() XProtocolVersion()</td>
<td>Returns the major version number (11) of the protocol associated with the display.</td>
</tr>
<tr>
<td>QLength() XQLength()</td>
<td>Returns the length of the event queue for the display.</td>
</tr>
<tr>
<td>RootWindow() XRootWindow()</td>
<td>Returns the root window of the current screen.</td>
</tr>
<tr>
<td>RootWindowOfScreen() XRootWindowOfScreen()</td>
<td>Returns the root window of the specified screen.</td>
</tr>
<tr>
<td>ScreenCount() XScreenCount()</td>
<td>Returns the number of screens available.</td>
</tr>
<tr>
<td>XScreenNumberOfScreen()</td>
<td>Returns the screen index number of the specified screen.</td>
</tr>
<tr>
<td>ScreenOfDisplay() XScreenOfDisplay()</td>
<td>Returns the pointer to the screen of the display specified.</td>
</tr>
<tr>
<td>ServerVendor() XServerVendor()</td>
<td>Returns the pointer to a null-determined string that identifies the owner of the X server implementation.</td>
</tr>
<tr>
<td>VendorRelease() XVendorRelease()</td>
<td>Returns the number related to the vendor's release of the X server.</td>
</tr>
<tr>
<td>WhitePixel() XWhitePixel()</td>
<td>Returns the white pixel value for the current screen.</td>
</tr>
<tr>
<td>WhitePixelOfScreen() XWhitePixelOfScreen()</td>
<td>Returns the white pixel value of the specified screen.</td>
</tr>
<tr>
<td>WidthMMOfScreen() XWidthMMOfScreen()</td>
<td>Returns the width of the specified screen in millimeters.</td>
</tr>
<tr>
<td>WidthOfScreen() XWidthOfScreen()</td>
<td>Returns the width of the specified screen in pixels.</td>
</tr>
</tbody>
</table>
## Extension Routines

Table 61 lists the X Window System Extension Subroutines.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAllocID()</td>
<td>Returns a resource ID that can be used when creating new resources.</td>
</tr>
<tr>
<td>XSetCloseDisplay()</td>
<td>Defines a procedure to call when XCloseDisplay is called.</td>
</tr>
<tr>
<td>XSetCopyGC()</td>
<td>Defines a procedure to call when a GC is copied.</td>
</tr>
<tr>
<td>XSetCreateFont()</td>
<td>Defines a procedure to call when XLoadQueryFont is called.</td>
</tr>
<tr>
<td>XSetCreateGC()</td>
<td>Defines a procedure to call when a new GC is created.</td>
</tr>
<tr>
<td>XSetError()</td>
<td>Suppresses the call to an external error handling routine and defines an alternative routine for error handling.</td>
</tr>
<tr>
<td>XSetErrorString()</td>
<td>Defines a procedure to call when an I/O error is detected.</td>
</tr>
<tr>
<td>XSetEventToWire()</td>
<td>Defines a procedure to call when an event must be converted from the host to wire format.</td>
</tr>
<tr>
<td>XSetFreeFont()</td>
<td>Defines a procedure to call when XFreeFont is called.</td>
</tr>
<tr>
<td>XSetFreeGC()</td>
<td>Defines a procedure to call when a GC is freed.</td>
</tr>
<tr>
<td>XSetWireToEvent()</td>
<td>Defines a procedure to call when an event is converted from the wire to the host format.</td>
</tr>
<tr>
<td>XFreeExtensionList()</td>
<td>Frees memory allocated by XListExtensions.</td>
</tr>
<tr>
<td>XListExtensions()</td>
<td>Returns a list of all extensions supported by the server.</td>
</tr>
<tr>
<td>XQueryExtension()</td>
<td>Indicates whether named extension is present.</td>
</tr>
</tbody>
</table>

## MIT Extensions to X

Table 62 lists the routines that allow an application to use these extensions:

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XShapeQueryExtension</td>
<td>Queries to see if the server supports the SHAPE extension.</td>
</tr>
<tr>
<td>XShapeQueryVersion</td>
<td>Checks the version number of the server SHAPE extension.</td>
</tr>
<tr>
<td>XShapeCombineRegion</td>
<td>Converts the specified region into a list of rectangles and calls XShapeRectangles.</td>
</tr>
<tr>
<td>XShapeCombineRectangles</td>
<td>Performs a CombineRectangles operation.</td>
</tr>
</tbody>
</table>
Table 62. MIT Extensions to X (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XShapeCombineMask</td>
<td>Performs a CombineMask operation.</td>
</tr>
<tr>
<td>XShapeCombineShape</td>
<td>Performs a CombineShape operation.</td>
</tr>
<tr>
<td>XShapeOffsetShape</td>
<td>Performs an OffsetShape operation.</td>
</tr>
<tr>
<td>XShapeQueryExtents</td>
<td>Sets the extents of the bounding and clip shapes.</td>
</tr>
<tr>
<td>XShapeSelectInput</td>
<td>Selects Input Events.</td>
</tr>
<tr>
<td>XShapeInputSelected</td>
<td>Returns the current input mask for extension events on the specified window.</td>
</tr>
<tr>
<td>XShapeGetRectangles</td>
<td>Gets a list of rectangles describing the region specified.</td>
</tr>
<tr>
<td>XMITMiscQueryExtension</td>
<td>Queries to see if the server supports the MITMISC extension.</td>
</tr>
<tr>
<td>XMITMiscSetBugMode</td>
<td>Sets the compatibility mode switch.</td>
</tr>
<tr>
<td>XMITMiscGetBugMode</td>
<td>Queries the compatibility mode switch.</td>
</tr>
<tr>
<td>XmbufQueryExtension</td>
<td>Queries to see if the server supports the MULTIBUF extension.</td>
</tr>
<tr>
<td>XmbufGetVersion</td>
<td>Gets the version number of the extension.</td>
</tr>
<tr>
<td>XmbufCreateBuffers</td>
<td>Requests that multiple buffers be created.</td>
</tr>
<tr>
<td>XmbufDestroyBuffers</td>
<td>Requests that the buffers be destroyed.</td>
</tr>
<tr>
<td>XmbufDisplayBuffers</td>
<td>Displays the indicated buffers.</td>
</tr>
<tr>
<td>XmbufGetWindowAttributes</td>
<td>Gets the multibuffering attributes.</td>
</tr>
<tr>
<td>XmbufChangeWindowAttributes</td>
<td>Sets the multibuffering attributes.</td>
</tr>
<tr>
<td>XmbufGetBufferAttributes</td>
<td>Gets the attributes for the indicated buffer.</td>
</tr>
<tr>
<td>XmbufChangeBufferAttributes</td>
<td>Sets the attributes for the indicated buffer.</td>
</tr>
<tr>
<td>XmbufGetScreenInfo</td>
<td>Gets the parameters controlling how mono and stereo windows may be created on the indicated screen.</td>
</tr>
<tr>
<td>XmbufCreateStereoWindow</td>
<td>Creates a stereo window.</td>
</tr>
</tbody>
</table>

Associate Table Functions

[Table 63] lists the Associate Table functions.

Table 63. Associate Table Functions

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCreateAssocTable()</td>
<td>Returns a pointer to the newly created associate table.</td>
</tr>
<tr>
<td>XDeleteAssoc()</td>
<td>Deletes an entry from the specified associate table.</td>
</tr>
<tr>
<td>XDestroyAssocTable()</td>
<td>Frees memory allocated to the specified associate table.</td>
</tr>
<tr>
<td>XLookUpAssoc()</td>
<td>Obtains data from the specified associate table.</td>
</tr>
<tr>
<td>XMakeAssoc()</td>
<td>Creates an entry in the specified associate table.</td>
</tr>
</tbody>
</table>
# Miscellaneous Utility Routines

Table 64 lists the Miscellaneous Utility routines.

## Table 64. Miscellaneous Utility Routines

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XctCreate()</td>
<td>Creates an XctData structure for parsing a Compound Text string.</td>
</tr>
<tr>
<td>XctFree()</td>
<td>Frees all data associated with the XctData structure.</td>
</tr>
<tr>
<td>XctNextItem()</td>
<td>Parses the next item from the Compound Text string.</td>
</tr>
<tr>
<td>XctReset()</td>
<td>Resets the XctData structure to reparse the Compound Text string.</td>
</tr>
<tr>
<td>XmuAddCloseDisplayHook()</td>
<td>Adds a callback for the given display.</td>
</tr>
<tr>
<td>XmuAddInitializer()</td>
<td>Registers a procedure, to be invoked the first time XmuCallInitializers is called on a given application context.</td>
</tr>
<tr>
<td>XmuAllStandardColormaps()</td>
<td>Creates all of the appropriate standard colormaps.</td>
</tr>
<tr>
<td>XmuCallInitializers()</td>
<td>Calls each of the procedures that have been registered with XmuAddInitializer.</td>
</tr>
<tr>
<td>XmuClientWindow()</td>
<td>Finds a window, at or below the specified window.</td>
</tr>
<tr>
<td>XmuCompareISOLatin1()</td>
<td>Compares two strings, ignoring case differences.</td>
</tr>
<tr>
<td>XmuConvertStandardSelection()</td>
<td>Converts many standard selections.</td>
</tr>
<tr>
<td>XmuCopyISOLatin1Lowered()</td>
<td>Copies a string, changing all Latin-1 uppercase letters to lowercase.</td>
</tr>
<tr>
<td>XmuCopyISOLatin1Uppered()</td>
<td>Copies a string, changing all Latin-1 lowercase letters to uppercase.</td>
</tr>
<tr>
<td>XmuCreateColormap()</td>
<td>Creates a colormap.</td>
</tr>
<tr>
<td>XmuCreatePixmapFromBitmap()</td>
<td>Creates a pixmap of the specified width, height, and depth.</td>
</tr>
<tr>
<td>XmuCreateStippledPixmap()</td>
<td>Creates a two pixel by one pixel stippled pixmap of specified depth on the specified screen.</td>
</tr>
<tr>
<td>XmuCursorNameToIndex()</td>
<td>Returns the index in the standard cursor font for the name of a standard cursor.</td>
</tr>
<tr>
<td>XmuCvtFunctionToCallback()</td>
<td>Converts a callback procedure to a callback list containing that procedure.</td>
</tr>
<tr>
<td>XmuCvtStringToBackingStore()</td>
<td>Converts a string to a backing-store integer.</td>
</tr>
<tr>
<td>XmuCvtStringToBitmap()</td>
<td>Creates a bitmap suitable for window manager icons.</td>
</tr>
<tr>
<td>XmuCvtStringToCursor()</td>
<td>Converts a string to a Cursor.</td>
</tr>
<tr>
<td>XmuCvtStringToJustify()</td>
<td>Converts a string to an XtJustify enumeration value.</td>
</tr>
<tr>
<td>XmuCvtStringToLong()</td>
<td>Converts a string to an integer of type long.</td>
</tr>
</tbody>
</table>
### Table 64. Miscellaneous Utility Routines (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XmuCvtStringToOrientation()</td>
<td>Converts a string to an XtOrientation enumeration value.</td>
</tr>
<tr>
<td>XmuCvtStringToShapeStyle()</td>
<td>Converts a string to an integer shape style.</td>
</tr>
<tr>
<td>XmuCvtStringToWidget()</td>
<td>Converts a string to an immediate child widget of the parent widget passed as an argument.</td>
</tr>
<tr>
<td>XmuDeleteStandardColormap()</td>
<td>Removes the specified property from the specified screen.</td>
</tr>
<tr>
<td>XmuDQAddDisplay()</td>
<td>Adds the specified display to the queue.</td>
</tr>
<tr>
<td>XmuDQCreate()</td>
<td>Creates and returns an empty XmuDisplayQueue.</td>
</tr>
<tr>
<td>XmuDQDestroy()</td>
<td>Releases all memory associated with the specified queue.</td>
</tr>
<tr>
<td>XmuDQLookupDisplay()</td>
<td>Returns the queue entry for the specified display.</td>
</tr>
<tr>
<td>XmuDQNDDisplays()</td>
<td>Returns the number of displays in the specified queue.</td>
</tr>
<tr>
<td>XmuDQRemoveDisplay()</td>
<td>Removes the specified display from the specified queue.</td>
</tr>
<tr>
<td>XmuDrawLogo()</td>
<td>Draws the official X Window System logo.</td>
</tr>
<tr>
<td>XmuDraw RoundedRectangle()</td>
<td>Draws a rounded rectangle.</td>
</tr>
<tr>
<td>XmuFill RoundedRectangle()</td>
<td>Draws a filled rounded rectangle.</td>
</tr>
<tr>
<td>XmuGetAtomName()</td>
<td>Returns the name of an Atom.</td>
</tr>
<tr>
<td>XmuGetColormapAllocation()</td>
<td>Determines the best allocation of reds, greens, and blues in a standard colormap.</td>
</tr>
<tr>
<td>XmuGetHostname()</td>
<td>Returns the host name.</td>
</tr>
<tr>
<td>XmuInternAtom()</td>
<td>Caches the Atom value for one or more displays.</td>
</tr>
<tr>
<td>XmuInternStrings()</td>
<td>Converts a list of atom names into Atom values.</td>
</tr>
<tr>
<td>XmuLocateBitmapFile()</td>
<td>Reads a file in standard bitmap file format.</td>
</tr>
<tr>
<td>XmuLookupAPL()</td>
<td>Maps a key event to an APL string. This function is similar to XLookupString.</td>
</tr>
<tr>
<td>XmuLookupArabic()</td>
<td>Maps a key event to a Latin/Arabic (ISO 8859-6) string. This function is similar to XLookupString.</td>
</tr>
<tr>
<td>XmuLookupCyrillic()</td>
<td>Maps a key event to a Latin/Cyrillic (ISO 8859-5) string. This function is similar to XLookupString, except that it maps a</td>
</tr>
<tr>
<td>XmuLookupGreek()</td>
<td>Maps a key event to a Latin/Greek (ISO 8859-7) string. This function is similar to XLookupString.</td>
</tr>
<tr>
<td>XmuLookupHebrew()</td>
<td>Maps a key event to a Latin/Hebrew (ISO 8859-8) string. This function is similar to XLookupString.</td>
</tr>
</tbody>
</table>
### Table 64. Miscellaneous Utility Routines (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XmuLookupJISX0201()</td>
<td>Maps a key event to a string in the JIS X0201-1976 encoding. This function is similar to XLookupString.</td>
</tr>
<tr>
<td>XmuLookupKana()</td>
<td>Maps a key event to a string in the JIS X0201-1976 encoding. This function is similar to XLookupString.</td>
</tr>
<tr>
<td>XmuLookupLatin1()</td>
<td>This function is identical to XLookupString.</td>
</tr>
<tr>
<td>XmuLookupLatin2()</td>
<td>Maps a key event to a Latin-2 (ISO 8859-2) string. This function is similar to XLookupString.</td>
</tr>
<tr>
<td>XmuLookupLatin3()</td>
<td>Maps a key event to a Latin-3 (ISO 8859-3) string. This function is similar to XLookupString.</td>
</tr>
<tr>
<td>XmuLookupLatin4()</td>
<td>Maps a key event to a Latin-4 (ISO 8859-4) string. This function is similar to XLookupString.</td>
</tr>
<tr>
<td>XmuLookupStandardColormap()</td>
<td>Creates or replaces a standard colormap if one does not currently exist.</td>
</tr>
<tr>
<td>XmuLookupString()</td>
<td>Maps a key event into a specific key symbol set.</td>
</tr>
<tr>
<td>XmuMakeAtom()</td>
<td>Creates and initializes an opaque object.</td>
</tr>
<tr>
<td>XmuNameOfAtom()</td>
<td>Returns the name of an AtomPtr.</td>
</tr>
<tr>
<td>XmuPrintDefaultErrorMessage()</td>
<td>Prints an error message, equivalent to Xlib's default error message.</td>
</tr>
<tr>
<td>XmuReadBitmapData()</td>
<td>Reads a standard bitmap file description.</td>
</tr>
<tr>
<td>XmuReadBitmapDataFromFile()</td>
<td>Reads a standard bitmap file description from the specified file.</td>
</tr>
<tr>
<td>XmuReleaseStippledPixmap()</td>
<td>Frees a pixmap created with XmuCreateStippledPixmap.</td>
</tr>
<tr>
<td>XmuRemoveCloseDisplayHook()</td>
<td>Deletes a callback that has been added with XmuAddCloseDisplayHook.</td>
</tr>
<tr>
<td>XmuReshapeWidget()</td>
<td>Reshapes the specified widget, using the Shape extension.</td>
</tr>
<tr>
<td>XmuScreenOfWindow()</td>
<td>Returns the screen on which the specified window was created.</td>
</tr>
<tr>
<td>XmuSimpleErrorHandler()</td>
<td>A simple error handler for Xlib error conditions.</td>
</tr>
<tr>
<td>XmuStandardColormap()</td>
<td>Creates a standard colormap for the given screen.</td>
</tr>
<tr>
<td>XmuUpdateMapHints()</td>
<td>Clears the PPosition and PSize flags and sets the USPosition and USSize flags.</td>
</tr>
<tr>
<td>XmuVisualStandardColormaps()</td>
<td>Creates all of the appropriate standard colormaps for a given visual.</td>
</tr>
</tbody>
</table>
X Window System Interface

### X Authorization Routines

Table 65 lists the X Authorization routines.

**Table 65. Authorization Data Routines**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XauFileName()</td>
<td>Generates the default authorization file name.</td>
</tr>
<tr>
<td>XauReadAuth()</td>
<td>Reads the next entry from the authfile.</td>
</tr>
<tr>
<td>XuWriteAuth()</td>
<td>Writes an authorization entry to the authfile.</td>
</tr>
<tr>
<td>XauGetAuthByAddr()</td>
<td>Searches for an authorization entry.</td>
</tr>
<tr>
<td>XauLockAuth()</td>
<td>Does the work necessary to synchronously update an authorization file.</td>
</tr>
<tr>
<td>XauUnlockAuth()</td>
<td>Undoes the work of XauLockAuth.</td>
</tr>
<tr>
<td>XauDisposeAuth()</td>
<td>Frees storage allocated to hold an authorization entry.</td>
</tr>
</tbody>
</table>

### X Intrinsics Routines

Table 66 provides the X Intrinsics Routines.

**Table 66. X Intrinsics Routines**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompositeClassPartInitialize</td>
<td>Initializes the CompositeClassPart of a Composite Widget.</td>
</tr>
<tr>
<td>CompositeDeleteChild</td>
<td>Deletes a child widget from a Composite Widget.</td>
</tr>
<tr>
<td>CompositeDestroy</td>
<td>Destroys a composite widget.</td>
</tr>
<tr>
<td>CompositeInitialize</td>
<td>Initializes a Composite Widget structure.</td>
</tr>
<tr>
<td>CompositeInsertChild</td>
<td>Inserts a child widget in a Composite Widget.</td>
</tr>
<tr>
<td>RemoveCallback</td>
<td>Removes a callback procedure from a callback list.</td>
</tr>
<tr>
<td>XrmCompileResourceList</td>
<td>Compiles an XtResourceList into an XrmResourceList.</td>
</tr>
<tr>
<td>XtAddActions</td>
<td>Declares an action table and registers it with the translation manager</td>
</tr>
<tr>
<td>XtAddCallback</td>
<td>Adds a callback procedure to the callback list of the specified widget.</td>
</tr>
<tr>
<td>XtAddCallbacks</td>
<td>Adds a list of callback procedures to the callback list of specified widget.</td>
</tr>
<tr>
<td>XtAddConverter</td>
<td>Adds a new converter.</td>
</tr>
<tr>
<td>XtAddEventHandler</td>
<td>Registers an event handler procedure with the dispatch mechanism when an event matching the mask occurs on the specified widget.</td>
</tr>
<tr>
<td>XtAddExposureToRegion</td>
<td>Computes the union of the rectangle defined by the specified exposure event and region.</td>
</tr>
<tr>
<td>XtAddGrab</td>
<td>Redirects user input to a model widget.</td>
</tr>
<tr>
<td>Routine</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>XtAddInput</td>
<td>Registers a new source of events.</td>
</tr>
<tr>
<td>XtAddRawEventHandler</td>
<td>Registers an event handler procedure with the dispatch mechanism without causing the server to select for that event.</td>
</tr>
<tr>
<td>XtAddTimeOut</td>
<td>Creates a timeout value in the default application context and returns an identifier for it.</td>
</tr>
<tr>
<td>XtAddWorkProc</td>
<td>Registers a work procedure in the default application context.</td>
</tr>
<tr>
<td>XtAppAddActionHook</td>
<td>Adds an actionhook procedure to an application context.</td>
</tr>
<tr>
<td>XtAppAddActions</td>
<td>Declares an action table and registers it with the translation manager.</td>
</tr>
<tr>
<td>XtAppAddConverter</td>
<td>Registers a new converter.</td>
</tr>
<tr>
<td>XtAppAddInput</td>
<td>Registers a new file as an input source for a specified application.</td>
</tr>
<tr>
<td>XtAppAddTimeOut</td>
<td>Creates a timeout value and returns an identifier for it.</td>
</tr>
<tr>
<td>XtAppAddWorkProc</td>
<td>Registers a work procedure for a specified procedure.</td>
</tr>
<tr>
<td>XtAppCreateShell</td>
<td>Creates a top-level widget that is the root of a widget tree.</td>
</tr>
<tr>
<td>XtAppError</td>
<td>Calls the installed fatal error procedure.</td>
</tr>
<tr>
<td>XtAppErrorMsg</td>
<td>Calls the high-level error handler.</td>
</tr>
<tr>
<td>XtAppGetErrorDatabase</td>
<td>Obtains the error database and merges it with an application or widget-specified database.</td>
</tr>
<tr>
<td>XtAppGetErrorDatabaseText</td>
<td>Obtains the error database text for an error or warning for an error message handler.</td>
</tr>
<tr>
<td>XtAppGetSelectionTimeout</td>
<td>Gets and returns the current selection timeout (ms) value.</td>
</tr>
<tr>
<td>XtAppInitialize</td>
<td>A convenience routine for initializing the toolkit.</td>
</tr>
<tr>
<td>XtAppMainLoop</td>
<td>Processes input by calling XtAppNextEvent and XtDispatchEvent.</td>
</tr>
<tr>
<td>XtAppNextEvent</td>
<td>Returns the value from the top of a specified application input queue.</td>
</tr>
<tr>
<td>XtAppPeekEvent</td>
<td>Returns the value from the top of a specified application input queue without removing input from queue.</td>
</tr>
<tr>
<td>XtAppPending</td>
<td>Determines if the input queue has any events for a specified application.</td>
</tr>
<tr>
<td>XtAppProcessEvent</td>
<td>Processes applications that require direct control of the processing for different types of input.</td>
</tr>
<tr>
<td>XtAppReleaseCacheRefs</td>
<td>Decrements the reference count for the conversion entries identified by the refs argument.</td>
</tr>
<tr>
<td>Routine</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>XtAppSetErrorHandler</td>
<td>Registers a procedure to call on fatal error conditions. The default error handler prints the message to standard error.</td>
</tr>
<tr>
<td>XtAppSetErrorMsgHandler</td>
<td>Registers a procedure to call on fatal error conditions. The default error handler constructs a string from the error resource database.</td>
</tr>
<tr>
<td>XtAppSetFallbackResources</td>
<td>Sets the fallback resource list that will be loaded at display initialization time.</td>
</tr>
<tr>
<td>XtAppSetSelectionTimeout</td>
<td>Sets the Intrinsics selection time-out value.</td>
</tr>
<tr>
<td>XtAppSetTypeConverter</td>
<td>Registers the specified type converter and destructor in all application contexts created by the calling process.</td>
</tr>
<tr>
<td>XtAppSetWarningHandler</td>
<td>Registers a procedure to call on nonfatal error conditions. The default warning handler prints the message to standard error.</td>
</tr>
<tr>
<td>XtAppSetWarningMsgHandler</td>
<td>Registers a procedure to call on nonfatal error conditions. The default warning handler constructs a string from error resource database.</td>
</tr>
<tr>
<td>XtAppWarning</td>
<td>Calls the installed nonfatal error procedure.</td>
</tr>
<tr>
<td>XtAppWarningMsg</td>
<td>Calls the installed high-level warning handler.</td>
</tr>
<tr>
<td>XtAugmentTranslations</td>
<td>Merges new translations into an existing widget translation table.</td>
</tr>
<tr>
<td>XtBuildEventMask</td>
<td>Retrieves the event mask for a specified widget.</td>
</tr>
<tr>
<td>XtCallAcceptFocus</td>
<td>Calls the accept_focus procedure for the specified widget.</td>
</tr>
<tr>
<td>XtCallActionProc</td>
<td>Searches for the named action routine and, if found, calls it.</td>
</tr>
<tr>
<td>XtCallbackExclusive</td>
<td>Calls customized code for callbacks to create pop-up shell.</td>
</tr>
<tr>
<td>XtCallbackNone</td>
<td>Calls customized code for callbacks to create pop-up shell.</td>
</tr>
<tr>
<td>XtCallbackNonexclusive</td>
<td>Calls customized code for callbacks to create pop-up shell.</td>
</tr>
<tr>
<td>XtCallbackPopdown</td>
<td>Pops down a shell that was mapped by callback functions.</td>
</tr>
<tr>
<td>XtCallbackReleaseCacheRef</td>
<td>A callback that can be added to a callback list to release a previously returned XtCacheRef value.</td>
</tr>
<tr>
<td>XtCallbackReleaseCacheRefList</td>
<td>A callback that can be added to a callback list to release a list of previously returned XtCacheRef value.</td>
</tr>
<tr>
<td>XtCallbackList</td>
<td>Calls all callbacks on a callback list.</td>
</tr>
<tr>
<td>XtCallbackCalls</td>
<td>Executes the callback procedures in a widget callback list.</td>
</tr>
</tbody>
</table>
## X Intrinsics Routines (continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XtCallConverter</td>
<td>Looks up the specified type converter in the application context and invokes the conversion routine.</td>
</tr>
<tr>
<td>XtCalloc</td>
<td>Allocates and initializes an array.</td>
</tr>
<tr>
<td>XtClass</td>
<td>Obtains the class of a widget and returns a pointer to the widget class structure.</td>
</tr>
<tr>
<td>XtCloseDisplay</td>
<td>Closes a display and removes it from an application context.</td>
</tr>
<tr>
<td>XtConfigureWidget</td>
<td>Moves and resizes the sibling widget of the child making the geometry request.</td>
</tr>
<tr>
<td>XtConvert</td>
<td>Invokes resource conversions.</td>
</tr>
<tr>
<td>XtConvertAndStore</td>
<td>Looks up the type converter registered to convert from_type to to_type and then calls XtCallConverter.</td>
</tr>
<tr>
<td>XtConvertCase</td>
<td>Determines upper and lowercase equivalents for a KeySym.</td>
</tr>
<tr>
<td>XtCopyAncestorSensitive</td>
<td>Copies the sensitive value from a widget record.</td>
</tr>
<tr>
<td>XtCopyDefaultColormap</td>
<td>Copies the default colormap from a widget record.</td>
</tr>
<tr>
<td>XtCopyDefaultDepth</td>
<td>Copies the default depth from a widget record.</td>
</tr>
<tr>
<td>XtCopyFromParent</td>
<td>Copies the parent from a widget record.</td>
</tr>
<tr>
<td>XtCopyScreen</td>
<td>Copies the screen from a widget record.</td>
</tr>
<tr>
<td>XtCreateApplicationContext</td>
<td>Creates an opaque type application context.</td>
</tr>
<tr>
<td>XtCreateApplicationShell</td>
<td>Creates an application shell widget by calling XtAppCreateShell.</td>
</tr>
<tr>
<td>XtCreateManagedWidget</td>
<td>Creates and manages a child widget in a single procedure.</td>
</tr>
<tr>
<td>XtCreatePopupShell</td>
<td>Creates a pop-up shell.</td>
</tr>
<tr>
<td>XtCreateWindow</td>
<td>Creates an instance of a widget.</td>
</tr>
<tr>
<td>XtCreateWindow</td>
<td>Calls XcreateWindow with the widget structure and parameter.</td>
</tr>
<tr>
<td>XtDatabase</td>
<td>Obtains the resource database for a particular display.</td>
</tr>
<tr>
<td>XtDestroyApplicationContext</td>
<td>Destroys an application context.</td>
</tr>
<tr>
<td>XtDestroyGC</td>
<td>Deallocates graphics context when it is no longer needed.</td>
</tr>
<tr>
<td>XtDestroyWidget</td>
<td>Destroys a widget instance.</td>
</tr>
<tr>
<td>XtDirectConvert</td>
<td>Invokes resource conversion.</td>
</tr>
<tr>
<td>XtDisownSelection</td>
<td>Informs the Intrinsics selection mechanism that the specified widget is to lose ownership of the selection.</td>
</tr>
<tr>
<td>XtDispatchEvent</td>
<td>Receives X events and calls appropriate event handlers.</td>
</tr>
<tr>
<td>Routine</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>XtDisplay</td>
<td>Returns the display pointer for the specified widget.</td>
</tr>
<tr>
<td>XtDisplayInitialize</td>
<td>Initializes a display and adds it to an application context.</td>
</tr>
<tr>
<td>XtDisplayOfObject</td>
<td>Returns the display pointer for the specified widget.</td>
</tr>
<tr>
<td>XtDisplayStringConversionWarning</td>
<td>Issues a warning message for conversion routines.</td>
</tr>
<tr>
<td>XtDisplayToApplicationContext</td>
<td>Retrieves the application context associated with a Display.</td>
</tr>
<tr>
<td>XtError</td>
<td>Calls the installed fatal error procedure.</td>
</tr>
<tr>
<td>XtErrorMsg</td>
<td>A low-level error and warning handler procedure type.</td>
</tr>
<tr>
<td>XtFindFile</td>
<td>Searches for a file using substitutions in a path list.</td>
</tr>
<tr>
<td>XtFree</td>
<td>Frees an allocated block of storage.</td>
</tr>
<tr>
<td>XtGetActionKeysym</td>
<td>Retrieves the KeySym and modifiers that matched the final event specification in a translation table entry.</td>
</tr>
<tr>
<td>XtGetApplicationNameAndClass</td>
<td>Returns the application name and class as passed to XtDisplayInitialize.</td>
</tr>
<tr>
<td>XtGetApplicationResources</td>
<td>Retrieves resources that are not specific to a widget, but apply to the overall application.</td>
</tr>
<tr>
<td>XtGetConstraintResourceList</td>
<td>Returns the constraint resource list for a particular widget.</td>
</tr>
<tr>
<td>XtGetErrorDatabase</td>
<td>Obtains the error database and returns the address of the error database.</td>
</tr>
<tr>
<td>XtGetErrorDatabaseText</td>
<td>Obtains the error database text for an error or warning.</td>
</tr>
<tr>
<td>XtGetGC</td>
<td>Returns a read-only sharable GC.</td>
</tr>
<tr>
<td>XtGetKeysymTable</td>
<td>Returns a pointer to the KeySym to KeyCode mapping table for a particular display.</td>
</tr>
<tr>
<td>XtGetMultiClickTime</td>
<td>Returns the multi-click time setting.</td>
</tr>
<tr>
<td>XtGetResourceList</td>
<td>Obtains the resource list structure for a particular class.</td>
</tr>
<tr>
<td>XtGetSelectionRequest</td>
<td>Retrieves the SelectionRequest event which triggered the convert_selection procedure.</td>
</tr>
<tr>
<td>XtGetSelectionTimeout</td>
<td>Obtains the current selection timeout.</td>
</tr>
<tr>
<td>XtGetSelectionValue</td>
<td>Obtains the selection value in a single, logical unit.</td>
</tr>
<tr>
<td>XtGetSelectionValueIncremental</td>
<td>Obtains the selection value using incremental transfers.</td>
</tr>
<tr>
<td>XtGetSelectionValues</td>
<td>Takes a list of target types and client data and obtains the current value of the selection converted to each of the targets.</td>
</tr>
</tbody>
</table>
### Table 66. X Intrinsics Routines (continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XtGetSelectionValuesIncremental</td>
<td>A function similar to XtGetSelectionValueIncremental except that it takes a list of targets and client_data.</td>
</tr>
<tr>
<td>XtGetSubresources</td>
<td>Obtains resources other than widgets.</td>
</tr>
<tr>
<td>XtGetSubvalues</td>
<td>Retrieves the current value of a non-widget resource data associated with a widget instance.</td>
</tr>
<tr>
<td>XtGetValues</td>
<td>Retrieves the current value of a resource associated with a widget instance.</td>
</tr>
<tr>
<td>XtGrabButton</td>
<td>Passively grabs a single pointer button.</td>
</tr>
<tr>
<td>XtGrabKey</td>
<td>Passively grabs a single key of the keyboard.</td>
</tr>
<tr>
<td>XtGrabKeyboard</td>
<td>Actively grabs the keyboard.</td>
</tr>
<tr>
<td>XtGrabPointer</td>
<td>Actively grabs the pointer.</td>
</tr>
<tr>
<td>XtHasCallbacks</td>
<td>Finds the status of a specified widget callback list.</td>
</tr>
<tr>
<td>XtInitialize</td>
<td>Initializes the toolkit, application, and shell.</td>
</tr>
<tr>
<td>XtInitializeWidgetClass</td>
<td>Initializes a widget class without creating any widgets.</td>
</tr>
<tr>
<td>XtInsertEventHandler</td>
<td>Registers an event handler procedure that receives events before or after all previously registered event handler.</td>
</tr>
<tr>
<td>XtInsertRawEventHandler</td>
<td>Registers an event handler procedure that receives events before or after all previously registered event handler without selecting for the events.</td>
</tr>
<tr>
<td>XtInstallAccelerators</td>
<td>Installs accelerators from a source widget to destination widget.</td>
</tr>
<tr>
<td>XtInstallAllAccelerators</td>
<td>Installs all the accelerators from a widget and all the descendants of the widget onto one destination widget.</td>
</tr>
<tr>
<td>XtIsApplicationShell</td>
<td>Determines whether a specified widget is a subclass of an Application Shell widget.</td>
</tr>
<tr>
<td>XtIsComposite</td>
<td>Determines whether a specified widget is a subclass of a Composite widget.</td>
</tr>
<tr>
<td>XtIsConstraint</td>
<td>Determines whether a specified widget is a subclass of a Constraint widget.</td>
</tr>
<tr>
<td>XtIsManaged</td>
<td>Determines the managed state of a specified child widget.</td>
</tr>
<tr>
<td>XtIsObject</td>
<td>Determines whether a specified widget is a subclass of an Object widget.</td>
</tr>
<tr>
<td>XtIsOverrideShell</td>
<td>Determines whether a specified widget is a subclass of an Override Shell widget.</td>
</tr>
<tr>
<td>XtIsRealized</td>
<td>Determines if a widget has been realized.</td>
</tr>
<tr>
<td>XtIsRectObj</td>
<td>Determines whether a specified widget is a subclass of a RectObj widget.</td>
</tr>
<tr>
<td>XtIsSensitive</td>
<td>Determines the current sensitivity state of a widget.</td>
</tr>
<tr>
<td>Routine</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>XtIsShell</td>
<td>Determines whether a specified widget is a subclass of a Shell widget.</td>
</tr>
<tr>
<td>XtIsSubclass</td>
<td>Determines whether a specified widget is in a specific subclass.</td>
</tr>
<tr>
<td>XtIsTopLevelShell</td>
<td>Determines whether a specified widget is a subclass of a TopLevelShell widget.</td>
</tr>
<tr>
<td>XtIsTransientShell</td>
<td>Determines whether a specified widget is a subclass of a TransientShell widget.</td>
</tr>
<tr>
<td>XtIsVendorShell</td>
<td>Determines whether a specified widget is a subclass of a VendorShell widget.</td>
</tr>
<tr>
<td>XtIsWidget</td>
<td>Determines whether a specified widget is a subclass of a Widget widget.</td>
</tr>
<tr>
<td>XtIsWMShell</td>
<td>Determines whether a specified widget is a subclass of a WMShell widget.</td>
</tr>
<tr>
<td>XtKeysymToKeycodeList</td>
<td>Returns the list of KeyCodes that map to a particular KeySym.</td>
</tr>
<tr>
<td>XtLastTimestampProcessed</td>
<td>Retrieves the timestamp from the most recent call to XtDispatchEvent.</td>
</tr>
<tr>
<td>XtMainLoop</td>
<td>An infinite loop which processes input.</td>
</tr>
<tr>
<td>XtMakeGeometryRequest</td>
<td>A request from the child widget to a parent widget for a geometry change.</td>
</tr>
<tr>
<td>XtMakeResizeRequest</td>
<td>Makes a resize request from a widget.</td>
</tr>
<tr>
<td>XtMalloc</td>
<td>Allocates storage.</td>
</tr>
<tr>
<td>XtManageChild</td>
<td>Adds a single child to a parent widget list of managed children.</td>
</tr>
<tr>
<td>XtManageChildren</td>
<td>Adds a list of widgets to the geometry-managed, displayable, subset of its composite parent widget.</td>
</tr>
<tr>
<td>XtMapWidget</td>
<td>Maps a widget explicitly.</td>
</tr>
<tr>
<td>XtMenuPopupAction</td>
<td>Pops up a menu when a pointer button is pressed or when the pointer is moved into the widget.</td>
</tr>
<tr>
<td>XtMergeArgLists</td>
<td>Merges two ArgList structures.</td>
</tr>
<tr>
<td>XtMoveWidget</td>
<td>Moves a sibling widget of the child making the geometry request.</td>
</tr>
<tr>
<td>XtName</td>
<td>Returns a pointer to the instance name of the specified object.</td>
</tr>
<tr>
<td>XtNameToWidget</td>
<td>Translates a widget name to a widget instance.</td>
</tr>
<tr>
<td>XtNewString</td>
<td>Copies an instance of a string.</td>
</tr>
<tr>
<td>XtNextEvent</td>
<td>Returns the value from the header of the input queue.</td>
</tr>
<tr>
<td>XtOpenDisplay</td>
<td>Opens, initializes, and adds a display to an application context.</td>
</tr>
<tr>
<td>XtOverrideTranslations</td>
<td>Overwrites existing translations with new translations.</td>
</tr>
<tr>
<td>Routine</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>XtOwnSelection</td>
<td>Sets the selection owner when using atomic transfer.</td>
</tr>
<tr>
<td>XtOwnSelectionIncremental</td>
<td>Sets the selection owner when using incremental transfers.</td>
</tr>
<tr>
<td>XtParent</td>
<td>Returns the parent widget for the specified widget.</td>
</tr>
<tr>
<td>XtParseAcceleratorTable</td>
<td>Parses an accelerator table into the opaque internal representation.</td>
</tr>
<tr>
<td>XtParseTranslationTable</td>
<td>Compiles a translation table into the opaque internal representation of type XtTranslations.</td>
</tr>
<tr>
<td>XtPeekEvent</td>
<td>Returns the value from the front of the input queue without removing it from the queue.</td>
</tr>
<tr>
<td>XtPending</td>
<td>Determines if the input queue has events pending.</td>
</tr>
<tr>
<td>XtPopdown</td>
<td>Unmaps a pop-up from within an application.</td>
</tr>
<tr>
<td>XtPopup</td>
<td>Maps a pop-up from within an application.</td>
</tr>
<tr>
<td>XtPopupSpringLoaded</td>
<td>Maps a spring-loaded pop-up from within an application.</td>
</tr>
<tr>
<td>XtProcessEvent</td>
<td>Processes one input event, timeout, or alternate input source.</td>
</tr>
<tr>
<td>XtQueryGeometry</td>
<td>Queries the preferred geometry of a child widget.</td>
</tr>
<tr>
<td>XtRealizeWidget</td>
<td>Realizes a widget instances.</td>
</tr>
<tr>
<td>XtRealloc</td>
<td>Changes the size of an allocated block of storage, sometimes moving it.</td>
</tr>
<tr>
<td>XtRegisterCaseConverter</td>
<td>Registers a specified case converter.</td>
</tr>
<tr>
<td>XtRegisterGrabAction</td>
<td>Registers button and key grabs for a widget's window according to the event bindings in the widget's translation table.</td>
</tr>
<tr>
<td>XtReleaseGC</td>
<td>Deallocates a shared GC when it is no longer needed.</td>
</tr>
<tr>
<td>XtRemoveActionHook</td>
<td>Removes an action hook procedure without destroying the application context.</td>
</tr>
<tr>
<td>XtRemoveAllCallbacks</td>
<td>Deletes all callback procedures from a specified widget callback list.</td>
</tr>
<tr>
<td>XtRemoveCallback</td>
<td>Deletes a callback procedure from a specified widget callback list only if both the procedure and the client data match.</td>
</tr>
<tr>
<td>XtRemoveCallbacks</td>
<td>Deletes a list of callback procedures from a specified widget callback list.</td>
</tr>
<tr>
<td>XtRemoveEventHandler</td>
<td>Removes a previously registered event handler.</td>
</tr>
<tr>
<td>XtRemoveGrab</td>
<td>Removes the redirection of user input to a modal widget.</td>
</tr>
</tbody>
</table>
## Table 66. X Intrinsics Routines (continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XtRemoveInput</td>
<td>Discontinues a source of input by causing the Intrinsics read routine to stop watching for input from the input source.</td>
</tr>
<tr>
<td>XtRemoveRawEventHandler</td>
<td>Removes previously registered raw event handler.</td>
</tr>
<tr>
<td>XtRemoveTimeOut</td>
<td>Clears a timeout value by removing the timeout.</td>
</tr>
<tr>
<td>XtRemoveWorkProc</td>
<td>Removes the specified background work procedure.</td>
</tr>
<tr>
<td>XtResizeWidget</td>
<td>Resizes a sibling widget of the child making the geometry request.</td>
</tr>
<tr>
<td>XtResizeWindow</td>
<td>Resizes a child widget that already has the values for its width, height, and border width.</td>
</tr>
<tr>
<td>XtResolvePathname</td>
<td>Searches for a file using standard substitutions in a path list.</td>
</tr>
<tr>
<td>XtScreen</td>
<td>Returns the screen pointer for the specified widget.</td>
</tr>
<tr>
<td>XtScreenOfObject</td>
<td>Returns the screen pointer for the nearest ancestor of object that is of class Widget.</td>
</tr>
<tr>
<td>XtSetErrorHandler</td>
<td>Registers a procedure to call under fatal error conditions.</td>
</tr>
<tr>
<td>XtSetErrorMsgHandler</td>
<td>Registers a procedure to call under fatal error conditions.</td>
</tr>
<tr>
<td>XtSetKeyboardFocus</td>
<td>Redirects keyboard input to a child of a composite widget without calling XSetInputFocus.</td>
</tr>
<tr>
<td>XtSetKeyTranslator</td>
<td>Registers a key translator.</td>
</tr>
<tr>
<td>XtSetMappedWhenManaged</td>
<td>Changes the widget map_when_managed field.</td>
</tr>
<tr>
<td>XtSetMultiClickTime</td>
<td>Sets the multi-click time for an application.</td>
</tr>
<tr>
<td>XtSetSelectionTimeout</td>
<td>Sets the Intrinsics selection timeout.</td>
</tr>
<tr>
<td>XtSetSensitive</td>
<td>Sets the sensitivity state of a widget.</td>
</tr>
<tr>
<td>XtSetSubvalues</td>
<td>Sets the current value of a non-widget resource associated with an instance.</td>
</tr>
<tr>
<td>XtSetTypeConverter</td>
<td>Registers a type converter for all application contexts in a process.</td>
</tr>
<tr>
<td>XtSetValues</td>
<td>Modifies the current value of a resource associated with widget instance.</td>
</tr>
<tr>
<td>XtSetWarningHandler</td>
<td>Registers a procedure to be called on non-fatal error conditions.</td>
</tr>
<tr>
<td>XtSetWarningMsgHandler</td>
<td>Registers a procedure to be called on non-fatal error conditions.</td>
</tr>
<tr>
<td>XtSetWMColormapWindows</td>
<td>Sets the value of the WM_COLOMAP_WINDOWS property on a widget's window.</td>
</tr>
<tr>
<td>Routine</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>--------------------------------------------------------------------</td>
</tr>
<tr>
<td>XtStringConversionWarning</td>
<td>A convenience routine for old-format resource converters that convert from strings.</td>
</tr>
<tr>
<td>XtSuperclass</td>
<td>Obtains the superclass of a widget by returning a pointer to the superclass structure of the widget.</td>
</tr>
<tr>
<td>XtToolkitInitialize</td>
<td>Initializes the X Toolkit internals.</td>
</tr>
<tr>
<td>XtTranslateCoords</td>
<td>Translates an ([x,y]) coordinate pair from widget coordinates to root coordinates.</td>
</tr>
<tr>
<td>XtTranslateKey</td>
<td>The default key translator routine.</td>
</tr>
<tr>
<td>XtTranslateKeycode</td>
<td>Registers a key translator.</td>
</tr>
<tr>
<td>XtUngrabButton</td>
<td>Cancels a passive button grab.</td>
</tr>
<tr>
<td>XtUngrabKey</td>
<td>Cancels a passive key grab.</td>
</tr>
<tr>
<td>XtUngrabKeyboard</td>
<td>Cancels an active keyboard grab.</td>
</tr>
<tr>
<td>XtUngrabPointer</td>
<td>Cancels an active pointer grab.</td>
</tr>
<tr>
<td>XtUninstallTranslations</td>
<td>Causes the entire translation table for widget to be removed.</td>
</tr>
<tr>
<td>XtUnmanageChild</td>
<td>Removes a single child from the managed set of its parent.</td>
</tr>
<tr>
<td>XtUnmanageChildren</td>
<td>Removes a list of children from the managed list of the parent, but does not destroy the children widgets.</td>
</tr>
<tr>
<td>XtUnmapWidget</td>
<td>Unmaps a widget explicitly.</td>
</tr>
<tr>
<td>XtUnrealizeWidget</td>
<td>Destroys the associated with a widget and its descendants.</td>
</tr>
<tr>
<td>XtVaAppCreateShell</td>
<td>Creates a top-level widget that is the root of a widget tree using varargs lists.</td>
</tr>
<tr>
<td>XtVaAppInitialize</td>
<td>Initializes the Xtk internals, creates an application context, opens and initializes a display and creates the initial application shell instance using varargs lists.</td>
</tr>
<tr>
<td>XtVaCreateArgsList</td>
<td>Dynamically allocates a varargs list for use with XtVaNestedList in multiple calls.</td>
</tr>
<tr>
<td>XtVaCreateManagedWidget</td>
<td>Creates and manages a child widget in a single procedure using varargs lists.</td>
</tr>
<tr>
<td>XtVaCreatePopupShell</td>
<td>Creates a pop-up shell using varargs lists.</td>
</tr>
<tr>
<td>XtVaCreateWidget</td>
<td>Creates an instance of a widget using varargs lists.</td>
</tr>
<tr>
<td>XtVaGetApplicationResources</td>
<td>Retrieves resources for the overall application using varargs list.</td>
</tr>
<tr>
<td>XtVaGetSubresources</td>
<td>Fetches resources for widget sub-parts using varargs list.</td>
</tr>
<tr>
<td>XtVaGetSubvalues</td>
<td>Retrieves the current values of non-widget resources associated with a widget instance using varargs lists.</td>
</tr>
<tr>
<td>XtVaGetValues</td>
<td>Retrieves the current values of resources associated with a widget instance using varargs lists.</td>
</tr>
</tbody>
</table>
### Table 66: X Intrinsics Routines (continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XtVaSetSubvalues</td>
<td>Sets the current values of non-widget resources associated with a widget instance using varargs lists.</td>
</tr>
<tr>
<td>XtVaSetValues</td>
<td>Modifies the current values of resources associated with a widget instance using varargs lists.</td>
</tr>
<tr>
<td>XtWarning</td>
<td>Calls the installed non-fatal error procedure.</td>
</tr>
<tr>
<td>XtWarningMsg</td>
<td>Calls the installed high-level warning handler.</td>
</tr>
<tr>
<td>XtWidgetToApplicationContext</td>
<td>Gets the application context for given widget.</td>
</tr>
<tr>
<td>XtWindow</td>
<td>Returns the window of the specified widget.</td>
</tr>
<tr>
<td>XtWindowOfObject</td>
<td>Returns the window for the nearest ancestor of object that is of class Widget.</td>
</tr>
<tr>
<td>XtWindowToWidget</td>
<td>Translates a window and display pointer into a widget instance.</td>
</tr>
</tbody>
</table>

### Athena Widget Support

Table 67 provides the Athena widget routines.

### Table 67: Athena Widget Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XawAsciiSave</td>
<td>Saves the changes made in the current text source into a file.</td>
</tr>
<tr>
<td>XawAsciiSaveAsFile</td>
<td>Saves the contents of the current text buffer into a named file.</td>
</tr>
<tr>
<td>XawAsciiSourceChanged</td>
<td>Determines if the text buffer in an AsciiSrc object has changed.</td>
</tr>
<tr>
<td>XawAsciiSourceFreeString</td>
<td>Frees the storage associated with the string from an AsciiSrc widget requested with a call to XtGetValues.</td>
</tr>
<tr>
<td>XawDialogAddButton</td>
<td>Adds a new button to a Dialog widget.</td>
</tr>
<tr>
<td>XawDialogGetValueString</td>
<td>Returns the character string in the text field of a Dialog Widget.</td>
</tr>
<tr>
<td>XawDiskSourceCreate</td>
<td>Creates a disk source.</td>
</tr>
<tr>
<td>XawFormDoLayout</td>
<td>Forces or defers a re-layout of the form.</td>
</tr>
<tr>
<td>XawInitializeWidgetSet</td>
<td>Forces a reference to vendor shell so that the one in this widget is installed.</td>
</tr>
<tr>
<td>XawListChange</td>
<td>Changes the list that is displayed.</td>
</tr>
<tr>
<td>XawListHighlight</td>
<td>Highlights an item in the list.</td>
</tr>
<tr>
<td>XawListShowCurrent</td>
<td>Retrieves the list element that is currently set.</td>
</tr>
<tr>
<td>XawListUnhighlight</td>
<td>Unhighlights an item in the list.</td>
</tr>
<tr>
<td>XawPanedAllowResize</td>
<td>Enables or disables a child’s request for pane resizing.</td>
</tr>
<tr>
<td>XawPanedGetMinMax</td>
<td>Retrieves the minimum and maximum height settings for a pane.</td>
</tr>
</tbody>
</table>

X Window System Interface

240  
z/VM: TCP/IP Programmer’s Reference
Table 67. Athena Widget Routines (continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XawPanedGetNumSub</td>
<td>Retrieves the number of panes in a paned widget.</td>
</tr>
<tr>
<td>XawPanedSetMinMax</td>
<td>Sets the minimum and maximum height settings for a pane.</td>
</tr>
<tr>
<td>XawPanedSetRefigureMode</td>
<td>Enables or disables automatic recalculation of pane sizes and positions.</td>
</tr>
<tr>
<td>XawScrollbarSetThumb</td>
<td>Sets the position and length of a Scrollbar thumb.</td>
</tr>
<tr>
<td>XawSimpleMenuAddGlobalActions</td>
<td>Registers an XawPositionSimpleMenu global action routine.</td>
</tr>
<tr>
<td>XawSimpleMenuClearActiveEntry</td>
<td>Clears the SimpleMenu widget’s internal information about the currently highlighted menu entry.</td>
</tr>
<tr>
<td>XawSimpleMenuGetActiveEntry</td>
<td>Gets the currently highlighted menu entry.</td>
</tr>
<tr>
<td>XawStringSourceCreate</td>
<td>Creates a string source.</td>
</tr>
<tr>
<td>XawTextDisableRedisplay</td>
<td>Disables redisplay while making several changes to a Text Widget.</td>
</tr>
<tr>
<td>XawTextDisplay</td>
<td>Displays batched updates.</td>
</tr>
<tr>
<td>XawTextDisplayCaret</td>
<td>Enables and disables the insert point.</td>
</tr>
<tr>
<td>XawTextEnableRedisplay</td>
<td>Enables redisplay.</td>
</tr>
<tr>
<td>XawTextGetInsertionPoint</td>
<td>Returns the current position of the insert point.</td>
</tr>
<tr>
<td>XawTextGetSelectionPos</td>
<td>Retrieves the text that has been selected by this text widget.</td>
</tr>
<tr>
<td>XawTextGetSource</td>
<td>Retrieves the current text source for the specified widget.</td>
</tr>
<tr>
<td>XawTextInvalidate</td>
<td>Redisplays a range of characters.</td>
</tr>
<tr>
<td>XawTextReplace</td>
<td>Modifies the text in an editable Text widget.</td>
</tr>
<tr>
<td>XawTextSearch</td>
<td>Searches for a string in a Text widget.</td>
</tr>
<tr>
<td>XawTextSetInsertionPoint</td>
<td>Moves the insert point to the specified source position.</td>
</tr>
<tr>
<td>XawTextSetLastPos</td>
<td>Sets the last position data in an AsciiSource Object.</td>
</tr>
<tr>
<td>XawTextSetSelection</td>
<td>Selects a piece of text.</td>
</tr>
<tr>
<td>XawTextSetSelectionArray</td>
<td>Assigns a new selection array to a text widget.</td>
</tr>
<tr>
<td>XawTextSetSource</td>
<td>Replaces the text source in the specified widget.</td>
</tr>
<tr>
<td>XawTextSinkClearToBackground</td>
<td>Clears a region of the sink to the background color</td>
</tr>
<tr>
<td>XawTextSinkDisplayText</td>
<td>Stub function that in subclasses will display text.</td>
</tr>
<tr>
<td>XawTextSinkFindDistance</td>
<td>Finds the Pixel Distance between two text Positions.</td>
</tr>
<tr>
<td>XawTextSinkFindPosition</td>
<td>Finds a position in the text.</td>
</tr>
</tbody>
</table>
### Athena Widget Routines (continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XawTextSinkGetCursorBounds</td>
<td>Finds the bounding box for the insert cursor.</td>
</tr>
<tr>
<td>XawTextSinkInsertCursor</td>
<td>Places the InsertCursor.</td>
</tr>
<tr>
<td>XawTextSinkMaxHeight</td>
<td>Finds the Minimum height that will contain a given number of lines.</td>
</tr>
<tr>
<td>XawTextSinkMaxLines</td>
<td>Finds the Maximum number of lines that will fit in a given height.</td>
</tr>
<tr>
<td>XawTextSinkResolve</td>
<td>Resolves a location to a position.</td>
</tr>
<tr>
<td>XawTextSinkSetTabs</td>
<td>Sets the Tab stops.</td>
</tr>
<tr>
<td>XawTextSourceConvertSelection</td>
<td>Dummy selection converter.</td>
</tr>
<tr>
<td>XawTextSourceRead</td>
<td>Reads the source into a buffer.</td>
</tr>
<tr>
<td>XawTextSourceReplace</td>
<td>Replaces a block of text with new text.</td>
</tr>
<tr>
<td>XawTextSourceScan</td>
<td>Scans the text source for the number and type of item specified.</td>
</tr>
<tr>
<td>XawTextSourceSearch</td>
<td>Searches the text source for the text block passed.</td>
</tr>
<tr>
<td>XawTextSourceSetSelection</td>
<td>Allows special setting of the selection.</td>
</tr>
<tr>
<td>XawTextTopPosition</td>
<td>Returns the character position of the left-most character on the first line displayed in the widget.</td>
</tr>
<tr>
<td>XawTextUnsetSelection</td>
<td>Unhighlights previously highlighted text in a widget.</td>
</tr>
<tr>
<td>XawToggleChangeRadioGroup</td>
<td>Allows a toggle widget to change radio groups.</td>
</tr>
<tr>
<td>XawToggleGetCurrent</td>
<td>Returns the RadioData associated with the toggle widget that is currently active in a toggle group.</td>
</tr>
<tr>
<td>XawToggleSetCurrent</td>
<td>Sets the Toggle widget associated with the radio_data specified.</td>
</tr>
<tr>
<td>XawToggleUnsetCurrent</td>
<td>Unsets all Toggles in the radio_group specified.</td>
</tr>
</tbody>
</table>

### 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

For a table that lists these extension routines and provides a description of each extension routine, see Table 61 on page 225.
MIT Extensions to X

The AIX extensions described in the IBM AIX X-Windows 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 TCP/IP for z/VM:

- SHAPE
- MITMISC
- MULTIBUF

See [Table 62 on page 225]

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 bitmaps, 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. For a table that lists these associate table functions and provides a description of each function, see [Table 63 on page 226]

Miscellaneous Utility Routines

Included in the X11LIB TXTLIB are the MIT X Miscellaneous Utility routines. These routines are a set of common utility functions that have been useful to application writers. For a table that lists these utility routines and provides a description of each utility routine, see [Table 64 on page 227]

X Authorization Routines

Included in the X11LIB TXTLIB are the MIT X Authorization routines. These routines are used to deal with X authorization data in X clients. For a table that lists these subroutines and provides a description of each authorization routine, see [Table 65 on page 228]

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, 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, file, and timer events
- Input focus
- Selections
- Resources and resource conversion
- Translation of events
- Graphics contexts
- Pixmaps
- Errors and warnings.

In the VM/CMS environment, you must remap many of the X Widget and X Intrinsics routine names. This remapping is done in a file called XT_REMAP.H. This file is automatically included by the INTRINSIC.H header file. In debugging your application, it may be helpful to reference the XT_REMAP.H file to find the remapped names of the X Toolkit routines.

Some of the X Window System header files have been renamed from their original distribution names, because of the file-naming conventions in the VM/CMS 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 VM/CMS environment.

In porting applications to the VM/CMS environment, you may have to make file name changes as shown in Table 68 on page 243.
Table 68. X Intrinsic Header File Names

<table>
<thead>
<tr>
<th>MIT Distribution Name</th>
<th>TCP/IP Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompositeI.h</td>
<td>ComposiI.h</td>
</tr>
<tr>
<td>CompositeP.h</td>
<td>ComposiP.h</td>
</tr>
<tr>
<td>ConstrainP.h</td>
<td>ConstraP.h</td>
</tr>
<tr>
<td>IntrinsicI.h</td>
<td>IntriniI.h</td>
</tr>
<tr>
<td>IntrinsicP.h</td>
<td>IntriniP.h</td>
</tr>
<tr>
<td>PassivGraI.h</td>
<td>PassivGr.h</td>
</tr>
<tr>
<td>ProtocolsP.h</td>
<td>ProtocoP.h</td>
</tr>
<tr>
<td>SelectionI.h</td>
<td>SelectiI.h</td>
</tr>
<tr>
<td>WindowObjP.h</td>
<td>WindowOP.h</td>
</tr>
</tbody>
</table>

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 CMS environment in descending order of preference is:

1. Command Line
   Standard arguments include:
   a. Command switches (-display, -fg, -foreground, +rv, and so forth)
   b. Resource manager directives (-name, -xrm)
   c. Natural language directive (-xnllanguage)

2. User Environment File
   Use the first source found from:
   a. The file named by the XENVIRONMENT environment variable, which can be set with the CMS command:
      `GLOBALV SELECT CENV SET XENVIRONMENT filename.filetype`
   b. XDEFAULT. host file
      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 (screen())
   b. X.DEFaulTS file

4. Application User Resources
   Use the first source found from:
   a. The file named by the XUSERFILESEARCHPATH environment variable that can be set with the CMS command:
      `GLOBALV SELECT CENV SET XUSERFILESEARCHPATH filename.filetype`
   b. The file, which is called XAPDF.classname.xapplresdir, if the XAPPLRESDIR environment variable has been set. In this case, XAPDF is the file name; and classname is the file type. The environment variable, xapplresdir, contains the value of the file mode.
X Window System Interface

The XAPPLRESDIR environment variable can be set with the CMS command:
GLOBALV SELECT CENV SET XAPPLRESDIR filemode

The CMS file name XAPDF is modified if a natural language directive is specified to be XAPDF xnllanguage, where xnllanguage is the string specified by the natural language directive.

5. Application Class Resources
Use the first source found from:

a. The file named by the XFILESEARCHPATH environment variable, which can be set with the CMS command:
GLOBALV SELECT CENV SET XFILESEARCHPATH filename.filename

b. The default application resource file named XAPDF:classname, where classname is the application-specified class name.

The CMS file name XAPDF is modified if a natural language directive is specified as xnllanguageXAPDF, where xnllanguage is the string specified by the natural language directive.

c. Fallback resources defined by XtAppSetFallbackResources within the application.

Athena Widget Set

The X Window System Support with TCP/IP includes the widget set developed at MIT, which is generally known as the Athena widget set.

The Athena widget set supports the following widgets:

| AsciiSink  | Pane
|---|---|
| AsciiSrc  | scrollbar
| AsciiText | Simple
| Box  | SimpleMenu
| Clock | Sme (Simple Menu Entry)
| Command | SmeBSB (BSB Menu Entry)
| Dialog | Smeline
| Form  | StripChart
| Grip  | Text
| Label | TextSink
| List  | TextSrc
| Logo  | Toggle
| Mailbox | VPaned
| MenuButton | Viewport

For a complete list of the widgets supported by the Athena widget set, see "Athena Widget Support" on page 240

Some of the header files have been renamed from their original distribution names, because of the file-naming conventions in the VM/CMS environment. In addition, some of the header file names were changed to eliminate duplicate file names with the OSF/Motif-based Widget support. If your application uses these header files, it
will have to be modified to use the new header file name, see Table 69.

<table>
<thead>
<tr>
<th>MIT Distribution Name</th>
<th>TCP/IP Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsciiSinkP.h</td>
<td>AscSinkP.h</td>
</tr>
<tr>
<td>AsciiSrcP.h</td>
<td>AscSrcP.h</td>
</tr>
<tr>
<td>AsciiTextP.h</td>
<td>AscTextP.h</td>
</tr>
<tr>
<td>Command.h</td>
<td>ACommand.h</td>
</tr>
<tr>
<td>CommandP.h</td>
<td>ACommanP.h</td>
</tr>
<tr>
<td>Form.h</td>
<td>AForm.h</td>
</tr>
<tr>
<td>FormP.h</td>
<td>AFormP.h</td>
</tr>
<tr>
<td>Label.h</td>
<td>ALabel.h</td>
</tr>
<tr>
<td>LabelP.h</td>
<td>ALabelP.h</td>
</tr>
<tr>
<td>List.h</td>
<td>AList.h</td>
</tr>
<tr>
<td>ListP.h</td>
<td>AListP.h</td>
</tr>
<tr>
<td>MenuButtoP.h</td>
<td>MenuButP.h</td>
</tr>
<tr>
<td>Scrollbar.h</td>
<td>AScrollb.h</td>
</tr>
<tr>
<td>ScrollbarP.h</td>
<td>AScrollP.h</td>
</tr>
<tr>
<td>SimpleMenP.h</td>
<td>SimpleMP.h</td>
</tr>
<tr>
<td>StripCharP.h</td>
<td>StripChP.h</td>
</tr>
<tr>
<td>TemplateP.h</td>
<td>TemplatP.h</td>
</tr>
<tr>
<td>Text.h</td>
<td>AText.h</td>
</tr>
<tr>
<td>TextSinkP.h</td>
<td>TextSinP.h</td>
</tr>
<tr>
<td>TextP.h</td>
<td>ATextP.h</td>
</tr>
<tr>
<td>TextSrcP.h</td>
<td>ATextSrP.h</td>
</tr>
<tr>
<td>ViewportP.h</td>
<td>ViewporP.h</td>
</tr>
</tbody>
</table>

**OSF/Motif-Based Widget Support**

The X Window System support with TCP/IP includes the OSF/Motif-based widget set (Release 1.1).

The OSF/Motif-based Widget set supports the
### X Window System Interface

Following gadgets and widgets:

<table>
<thead>
<tr>
<th>ArrowButton</th>
<th>MenuShell</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrowGadget</td>
<td>MessageBox</td>
</tr>
<tr>
<td>ArrowButtonDown</td>
<td>PanedWindow</td>
</tr>
<tr>
<td>BulletinBoard</td>
<td>PushButton</td>
</tr>
<tr>
<td>CascadeButton</td>
<td>PushButtonGadget</td>
</tr>
<tr>
<td>CascadeButtonDown</td>
<td>RowColumn</td>
</tr>
<tr>
<td>Command</td>
<td>Sash</td>
</tr>
<tr>
<td>DialogShell</td>
<td>Scale</td>
</tr>
<tr>
<td>DrawingArea</td>
<td>ScrolledWindow</td>
</tr>
<tr>
<td>DrawnButton</td>
<td>SelectionBox</td>
</tr>
<tr>
<td>FileSelectionBox</td>
<td>SelectionDialog</td>
</tr>
<tr>
<td>FileSelectionDialog</td>
<td>SelectionDialog</td>
</tr>
<tr>
<td>Form</td>
<td>Separator</td>
</tr>
<tr>
<td>Frame</td>
<td>SeparatorGadget</td>
</tr>
<tr>
<td>Label</td>
<td>Text</td>
</tr>
<tr>
<td>LabelGadget</td>
<td>ToggleButton</td>
</tr>
<tr>
<td>List</td>
<td>ToggleButtonGadget</td>
</tr>
<tr>
<td>MainWindow</td>
<td></td>
</tr>
</tbody>
</table>

Some of the header files have been renamed from their original distribution names, because of the file-naming conventions in the CMS 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 VM/CMS environment.

In porting applications to the CMS environment, you may have to make header file name changes as shown in Table 68 on page 243. In porting applications to the CMS environment, the header file name changes shown in Table 70 on page 249 may have to be made.
### Sample X Window System Applications

This section contains the following sample programs:

- A simple program that uses Xlib calls (see page "Xlib Sample Program")
- A simple program that uses the Athena widget set (see page "Athena Widget Sample Program" on page 250)
- A simple program that uses the OSF/Motif-based widget set (see page "OSF/Motif-Based Widget Sample Program" on page 252).

#### Xlib Sample Program

The following is an X Window System program that uses basic Xlib functions to create a window, map the window to the screen, wait 60 seconds, destroy the window, and end.

```c
/*
 * This is an X window program using X11 API,
 * that opens the display and creates a window, waits
 * 60 seconds, then destroys the window before ending.
 */
#include <X11.h>
#include <types.h>
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
    Display *dp;
    Window w;
    /*
    * X will lookup the value of the DISPLAY global variable in
    */
```
Xlib Sample Program

* the CENV group when passed a NULL pointer in XOpenDisplay.
*/

dp = XOpenDisplay(NULL);

/*
* Create a 200X200 window at xy(40,40) with black border and name.
*/
w = XCreateSimpleWindow(dp, RootWindow(dp, 0), 40, 40, 200, 200, 2, BlackPixel(dp, 0), WhitePixel(dp, 0));
XStoreName(dp, w, "VM/CMS X Sample");
XSetIconName(dp, w, "X Sample");

/*
* Map the window to the display.
* This will cause the window to become visible on the screen.
*/
XMapWindow(dp, w);

/*
* Force X to write buffered requests.
*/
XFlush(dp);

fprintf(stderr, "Going to sleep now.... 60 seconds...
");
system("CP SLEEP 60 SEC");
fprintf(stderr, "Okay, back!
");

/*
* Destroy the window and end the connection to the X Server.
*/
XDestroyWindow(dp, w);
XCloseDisplay(dp);
}

Athena Widget Sample Program

The following is a simple X Window System program that uses the Athena Label widget to create a window with the string Hello, World centered in the middle of a window.

/*
* This an example of how "Hello, World" could be written using
* The X Toolkit and the Athena widget set.
* November 14, 1989 - Chris D. Peterson
*/

/*
* $XConsortium: xhw.c,v 1.7 89/12/11 15:31:33 kit Exp $
* Copyright 1989 Massachusetts Institute of Technology
* Permission to use, copy, modify, distribute, and sell this
* software and its documentation for any purpose is hereby
* granted without fee, 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. 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.
*
* M.I.T. DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
* INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS,
* IN NO EVENT SHALL M.I.T. BE LIABLE FOR ANY SPECIAL, INDIRECT
* OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING
* FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION
* OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
* OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*/

#include <stdio.h>
#include <X11/Intrinsic.h> /* Include standard Toolkit Header file.
We do not need "StringDefs.h" */
#ifdef IBMCPP
#include <X11/Xaw/ALabel.h> /* Include the Label widget's header file. */
#else
#include <X11/Xaw/Label.h> /* Include the Label widget's header file. */
#endif
#include <X11/Xaw/Cardinals.h> /* Definition of ZERO. */

/*
* These resources will be loaded only if there is no app-defaults
* file for this application. Since this is such a simple application
* I am just loading the resources here. For more complex applications
* It is best to install an app-defaults file.
*/
String fallback_resources[] = { "*Label.Label: Hello, World", NULL };

main(argc, argv)
int argc;
char **argv;
{
    XtAppContext app_con;
    Widget toplevel;
    
    /*
    * Initialize the Toolkit, set the fallback resources, and get
    * the application context associated with this application.
    */
    toplevel = XtAppInitialize(&app_con, "Xhw", NULL, ZERO, &argc, argv,
                                fallback_resources, NULL, ZERO);

    /*
    * Create a Widget to display the string. The label is picked up
    * from the resource database.
    */
    (void) XtCreateManagedWidget("label", labelWidgetClass, toplevel,
                                NULL, ZERO);

    /*
    * Create the windows, and set their attributes according
    * to the Widget data.
    */
    XtRealizeWidget(toplevel);
    /*
    * Now process the events.
    */
OSF/Motif-Based Widget Sample Program

The following is a simple X Window System program that uses the OSF/Motif-based PushButton widget to pop up a window with the string Press here in it. It exits when you press the button.

```c
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <Xm/Shell.h>
#include <Xm/PushB.h>
static void CloseApp();

int main(argc, argv) 
{
    Arg args[10];
    int n;
    XmString xst;
    Display *display;
    XtAppContext app_context;

    XtToolkitInitialize();
    app_context = XtCreateApplicationContext();
    display = XtOpenDisplay(app_context, NULL, argv[0], "Xsamp3", NULL, 0, &argc, argv);
    if (display == NULL) {
        fprintf(stderr, "%s: Can't open display\n", argv[0]);
        exit(1);
    }
    n = 0;
    XtSetArg(args[n], XmNwidth, 100); n++;
    XtSetArg(args[n], XmNheight, 75); n++;
    XtSetArg(args[n], XmNallowShellResize, True); n++;
    Shell = XtAppCreateShell(argv[0], NULL, applicationShellWidgetClass, display, args, n);
    XtRealizeWidget(Shell);
    n = 0;
    Button = XmCreatePushButton(Shell, "Press here", args, n);
    XtManageChild(Button);
    XtAddCallback (Button, XmNactivateCallback, CloseApp, NULL);
    XtAppMainLoop(app_context);
}

/********************************************************************/
/* CloseApp() */
/********************************************************************/
static void CloseApp(w, client_data, call_data)
{
    exit(0);
}
```
Chapter 7. Kerberos Authentication System

This chapter describes the Kerberos Authentication system and the routines that you can use to write applications that make use of the ticket-granting system.

Kerberos is an authentication system that can be used within or across a TCP/IP network to identify clients and authenticate connection requests.

Most conventional time-sharing systems require prospective users to identify themselves to the system during the logon process. For example, in an VM environment a CMS user must enter a CMS user ID and password to access the applications running on the system. In other environments that contain workstations, you cannot rely on the operating system to provide authentication. Because of this limitation, a third party must authenticate the prospective user. In a TCP/IP environment, Kerberos provides this authentication service. You must supply a password only when first contacting Kerberos. You do not have to enter a password for each remote service that you request.

The Kerberos system in TCP/IP consists of the following protocols and functions:

- Authentication server
- Ticket-granting server
- Kerberos database
- Administration server
- Kerberos applications library
- Applications
- User programs.

Authentication Server

When you log on to most computer systems, you must identify yourself with a password. Initiating the Kerberos session is similar to logging on to any other time-sharing system, except that Kerberos requires additional checks. The authentication server provides a way for authenticated users to prove their identity to other servers across a network. The authentication server reads the Kerberos database to verify that the client making the request is the client named in the request.

Name Structures

For Kerberos to authenticate a client, that client must first be assigned a Kerberos name. A Kerberos name consists of three parts:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>principal name</td>
<td>Specifies the unique name of a user (client) or service.</td>
</tr>
<tr>
<td>instance</td>
<td>Specifies a label that is used to distinguish among variations of the principal name. An instance allows for the possibility that the same client or service can exist in several forms, which require distinct authentication. For users, an instance can provide different identifiers for different privileges. For example, the admin instance provides special privileges to the users assigned to it.</td>
</tr>
</tbody>
</table>
Kerberos Authentication System

For services, an instance usually specifies the host name of the machine that provides the service.

`realm` Specifies the domain name of an administrative entity. The `realm` identifies each independent Kerberos site. The principal name and instance are qualified by the `realm` to which they belong, and are unique only within that `realm`. The `realm` is commonly the domain name.

When writing a Kerberos name, the principal name is separated from the instance (if not NULL) by a period. The `realm` follows, preceded by an `@` sign. The following are examples of valid Kerberos names:

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>billb</td>
<td>Principal name</td>
</tr>
<tr>
<td>jis.admin</td>
<td>Principal name and instance</td>
</tr>
<tr>
<td><a href="mailto:srz@inorg.chem.edu">srz@inorg.chem.edu</a></td>
<td>Principal name, null instance, realm</td>
</tr>
<tr>
<td><a href="mailto:trees.root@org.chem.edu">trees.root@org.chem.edu</a></td>
<td>Principal name, instance, realm</td>
</tr>
</tbody>
</table>

**Tickets and Authenticators**

Kerberos uses the combination of a ticket and an authenticator to provide authentication.

A ticket includes the following information:

- The client’s identity
- A session key
- A time stamp
- A lifetime for the ticket
- A service name

This information is encrypted in a private key, which is known only to Kerberos and the end server. A ticket can be used multiple times by the named client to gain access to the named server for the lifetime of the ticket.

An authenticator contains the name of the client, the client’s fully qualified domain name, and the current time. The authenticator maintains this initial information to keep other users from capturing and using tickets not granted to them and impersonating another user. This initial information, when compared against the information contained in the ticket, verifies that the client presenting the ticket is the same client to whom the ticket was issued. Unlike a ticket, the authenticator can be used only once. A new authenticator must be obtained each time a client program needs access to a service.

**Note:** The design of Kerberos assumes that system clocks are synchronized to within a few minutes on all machines that run Kerberos-authenticated services.

**Communicating with the Authentication Server**

The following four steps describe the authentication process. You must:

1. Establish your identity with the authentication server.
2. Obtain the initial ticket to access the ticket-granting server.
3. Request a ticket for a specific service from the ticket-granting server.
4. Present your ticket to the end server.
When you contact Kerberos, you are prompted for your user name. A request is then sent to the authentication server containing your name and the name of a special service called the ticket-granting server.

The authentication server searches the Kerberos database for your user name. If your user name appears in the database, the authentication server generates a random session key and the initial ticket.

The information contained in the ticket is encrypted in a key known only to the ticket-granting server and to the authentication server. The encrypted ticket and the session key are further encrypted, using a key known only to the Kerberos authentication server and the requester, and derived from the user’s password.

In the TCP/IP implementation of Kerberos, the network service that supplies the tickets is called the KERBEROS server. The KERBEROS server is comprised of the authentication server and the ticket-granting server. For information about how to set up the KERBEROS server, see TCP/IP Planning and Customization.

When you receive the initial ticket, you are prompted for your password, which is converted to a data encryption standard key and used to decrypt the response from the authentication server. Your password is not passed to Kerberos; it is only used locally to decrypt the initial ticket. The ticket and session key, along with the other information provided by the authentication server, are kept for future use.

**Ticket-Granting Server**

A ticket, granted by the ticket-granting server, is valid only for a single, specific service. You must obtain a ticket for each service you wish to access. The ticket can be used to access the service over the lifetime of the ticket.

The ticket granting server generates tickets to be used by client applications with different servers. To obtain a ticket for a new service, the program must provide the ticket-granting server with the name of the target service, as well as the initial ticket and the authenticator. The ticket-granting server again compares information, builds a ticket for the new service, and generates a new random session key. This information is encrypted and returned to the client program to authorize access to the new service.

**Accessing a Service**

Once you have been authenticated and obtained a ticket to a service, the client application then builds an authenticator, encrypting your name and fully qualified domain name, as well as the current time, with the session key that was originally received. With this information you can prove your identity for the lifetime of the ticket-granting ticket.

The target service decrypts the ticket, and uses the session key included in the ticket to decrypt the authenticator. The target service compares the information contained in the ticket and the authenticator. If the information matches, the request for the service is authorized. If the information does not match, the request for service is denied.

A client can request that a server prove its identity. To do so, the server returns an authenticator time stamp, incremented by one, back to the client.
Kerberos Authentication System

Figure 32 summarizes the ticket-granting process for accessing a service.

<table>
<thead>
<tr>
<th></th>
<th>Kerberos Authentication Server</th>
<th></th>
<th>User</th>
<th></th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 32. Protocol for Accessing a Service

1. Client asks the authentication server for a ticket to the ticket-granting server.
2. The authentication server provides the client with a ticket to the ticket-granting server.
3. Client asks the ticket-granting server for a ticket to a service.
4. Ticket-granting server provides the client with a ticket to a service.
5. Client accesses the service.
6. Service returns incremented time stamp.

Note: The authentication server and the ticket-granting server are implemented in a single program (KERBEROS authentication server or KERBEROS server).

Kerberos Database

Kerberos requires that each realm maintain a database of Kerberos user names (principal names), their private session keys, their expiration dates, and other administrative information. The authentication server reads this database to authenticate clients, but cannot change or update the information residing on the database. The administration server has both read and write authority. Only one Kerberos database can be maintained in each realm. TCP/IP provides this database with its Kerberos support.

Administration Server

The Administration server, known as ADMSERV, provides a read-write interface to the Kerberos database. You can request to change a password by using the KPASSWD program. Database administrators use the KADMIN program to add or update information from the database. All transactions performed with the ADMSERV server are logged. Both KADMIN and KPASSWD client applications use the authentication server rather than the ticket-granting server to get a ticket for the ADMSERV server. You must enter the password.

The Figure 33 on page 257 summarizes the steps involving the Kerberos database and the ADMSERV.
The Kerberos applications library provides an interface for client and server application programs. Usually these applications are used to write application programs in the C language. The applications library also contains routines for creating and reading authentication requests, and routines for creating and passing safe or private messages.

krb_mk_req() is the most commonly used client-side routine. krb_rd_req() is the most commonly used server-side routine.

The following is an example of a typical client-server exchange:

1. The client supplies krb_mk_req() with the service principal name, service instance, and realm of the target service.
2. The client sends the message returned by the krb_mk_req() routine over the network to the server-side of the application.
3. When the server receives this message, it calls krb_rd_req().
4. krb_rd_req() authenticates the identity of the requester and returns either permission or denial to access the application program.

Note: If the application requires that the messages exchanged between client and server be secret, the krb_mk_priv() and krb_rd_priv() routines are used to encrypt and decrypt the exchanges.

Figure 33. Protocol for Changing the Kerberos Database
This section provides a reference for Kerberos routines. Table 71 provides the names, descriptions, and page numbers of the routines, located in the KRB TXTLIB, which are needed to interface with Kerberos.

**Table 71. Kerberos krb_ Routines Reference**

<table>
<thead>
<tr>
<th>Kerberos krb_ Routine</th>
<th>Description</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>krb_get_cred()</td>
<td>Searches the caller’s ticket file for a ticket containing the specified principal name, instance, and realm.</td>
<td>259</td>
</tr>
<tr>
<td>krb_kntoln()</td>
<td>Converts a Kerberos name to a local name.</td>
<td>259</td>
</tr>
<tr>
<td>krb_mk_err()</td>
<td>Constructs an application level error message that can be used in conjunction with the krb_mk_priv() and krb_mk_safe() routines.</td>
<td>260</td>
</tr>
<tr>
<td>krb_mk_priv()</td>
<td>Creates an encrypted, authenticated message from any arbitrary application data pointed to by in.</td>
<td>260</td>
</tr>
<tr>
<td>krb_mk_req()</td>
<td>Takes a pointer to a text structure in which an authenticator is to be built. It also takes the principal name, instance, and realm of the service to be used and an optional checksum.</td>
<td>261</td>
</tr>
<tr>
<td>krb_mk_safe()</td>
<td>Creates an authenticated, but unencrypted message from any arbitrary application data pointed to by in.</td>
<td>261</td>
</tr>
<tr>
<td>krb_rd_err()</td>
<td>Unpacks a message received from krb_mk_err().</td>
<td>262</td>
</tr>
<tr>
<td>krb_rd_priv()</td>
<td>Decrypts and authenticates a message received from krb_mk_priv().</td>
<td>262</td>
</tr>
<tr>
<td>krb_rd_req()</td>
<td>Finds out information about the principal name when a request has been made to a service.</td>
<td>264</td>
</tr>
<tr>
<td>krb_rd_safe()</td>
<td>Authenticates a message received from krb_mk_safe().</td>
<td>265</td>
</tr>
<tr>
<td>krb_recvauth()</td>
<td>Called by the server to verify an authentication message received from a client.</td>
<td>265</td>
</tr>
<tr>
<td>krb_sendauth()</td>
<td>Prepares and transmits a ticket over a file descriptor.</td>
<td>266</td>
</tr>
</tbody>
</table>

**Client Commands**

Kerberos provides the following end-user commands:

- KINIT, to log on to Kerberos
- KLIST, to display Kerberos tickets
- KDESTROY, to destroy Kerberos tickets
- KPASSWD, to change a Kerberos password

For information about how to use these commands, see [TCP/IP User’s Guide](#).
Applications

You are responsible for securing your particular application through Kerberos. Programmers can write routines to call the applications library as an interface to their applications programs.

For a sample of a typical client program that establishes a connection on a remote server and a typical service program that authenticates a client’s service request, see "Sample Kerberos Programs" on page 268.

Kerberos Routines

This section provides the syntax, operands, and other appropriate information for routines, located in the KRB_TXTLIB, which are needed to interface with Kerberos.

**krb_get_cred()**

```c
int krb_get_cred(char *service, char *instance, char *realm, CREDENTIALS *c);
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service</td>
<td>Specifies the first part of the Kerberos name (principal name) of the target service.</td>
</tr>
<tr>
<td>instance</td>
<td>Specifies the second part of the Kerberos name of the target service.</td>
</tr>
<tr>
<td>realm</td>
<td>Specifies the third part of the Kerberos name of the target service.</td>
</tr>
<tr>
<td>c</td>
<td>Points to the structure, which is filled with credentials information.</td>
</tr>
</tbody>
</table>

**Description:** The krb_get_cred() routine searches the caller’s ticket file (tickets are maintained in the TMP_TKT0 file) for a ticket containing the specified principal name, instance, and realm. If a matching ticket is found, krb_get_cred() fills the specified CREDENTIALS structure with the ticket information. See the KRB.H header file for a definition of the CREDENTIALS structure.

**Return Values:** If successful, krb_get_cred() returns KSUCCESS. The error GC_TKFIL is returned when any of the following occur:
- The ticket cannot be read.
- The ticket file does not belong to the user.
- The ticket file is not a regular file.

If the ticket file cannot be found, krb_get_cred() returns GC_NOTKT.

See the KRB.H header file for a definition of the GC_TKFIL and GC_NOTKT return codes.

**krb_kntoln()**

```c
int krb_kntoln(AUTH_DAT *, char *iname);
```

Chapter 7. Kerberos Authentication System
### krb_kntoln()

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ad</td>
<td>Specifies an authentication structure containing a Kerberos name.</td>
</tr>
<tr>
<td>lname</td>
<td>Specifies a local name.</td>
</tr>
</tbody>
</table>

**Description:** The krb_kntoln() routine takes a Kerberos name in an AUTHDAT structure and checks that the `instance` is NULL and that the `realm` is the same as the local realm.

**Return Values:** KSUCCESS indicates success. The principal name is returned in `lname`. KFAILURE indicates an error.

### krb_mk_err()

```c
long krb_mk_err(out, code, string)
unsigned char *out;
long code;
char *string;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>Points to the output buffer area.</td>
</tr>
<tr>
<td>code</td>
<td>Specifies an application-specific error code.</td>
</tr>
<tr>
<td>string</td>
<td>Specifies an application-specific error string.</td>
</tr>
</tbody>
</table>

**Description:** The krb_mk_err() routine constructs an application-level error message consisting of the protocol version number, the message type, the host byte order, the specified code, and the text string. The krb_mk_err() routine returns a packet pointed to by `out`. The returned packet can be used in conjunction with the krb_mk_priv() and krb_mk_safe() routines.

The counterpart of the krb_mk_err() routine is the krb_rd_err() routine, which reads the message that is returned.

**Return Values:** krb_mk_error() returns an application-level error message pointed to by `out`. The long integer that is returned specifies the length of the message pointed to by `out`.

### krb_mk_priv()

```c
long krb_mk_priv(in, out, in_length, schedule, key, sender, receiver)
unsigned char *in;
unsigned char *out;
unsigned long in_length;
des_key_schedule schedule;
C_Block key;
struct sockaddr_in *sender;
struct sockaddr_in *receiver;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Points to an input structure containing application data.</td>
</tr>
<tr>
<td>out</td>
<td>Points to the output structure containing the encrypted data.</td>
</tr>
<tr>
<td>in_length</td>
<td>Specifies the length of the application data pointed to by <code>in</code>.</td>
</tr>
<tr>
<td>schedule</td>
<td>Specifies the session key schedule.</td>
</tr>
<tr>
<td>key</td>
<td>Points to a private session key.</td>
</tr>
<tr>
<td>sender</td>
<td>Specifies the fully qualified domain name of the sender.</td>
</tr>
<tr>
<td>receiver</td>
<td>Specifies the fully qualified domain name of the receiver.</td>
</tr>
</tbody>
</table>
**krb_mk_priv()**

**Description:** The krb_mk_priv() routine constructs an AUTH_MSG_PRIV message. The routine takes user data pointed to by `in`, of length specified by `in_length`, and creates a packet in `out`. This packet consists of the message type, the host byte order, user data, a time stamp, and the network address of the sender and receiver.

The packet is encrypted using the supplied `key` and `schedule`. The returned packet is decoded by the krb_rd_priv() routine in the receiver. In addition to providing privacy, this protocol message protects against modifications, insertions, or replays.

**Return Values:** krb_mk_priv() places the encrypted and authenticated message and header information in the area pointed to by `out`. The length of the output is returned upon success; the value −1 indicates an error.

**krb_mk_req()**

```c
extern char *krb_err_txt[];
int krb_mk_req(authent, service, instance, realm, checksum)
    KTEXT authent;
    char *service;
    char *instance;
    char *realm;
    unsigned long checksum;
```

**Operand Description**

- **authent**: Points to the text structure in which an authenticator (including a service ticket) is to be built.
- **service**: Specifies the first part of the Kerberos name (principal name) of the service.
- **instance**: Specifies the second part of the Kerberos name of the service.
- **realm**: Specifies the third part of the Kerberos name of the service.
- **checksum**: Specifies any long integer supplied by the calling routine for verification.

**Description:** The krb_mk_req() routine generates an authenticator by taking the principal name, instance, and realm of the service and an optional checksum. The application decides how to generate the checksum.

krb_mk_req() then retrieves a ticket for the desired service and creates an authenticator. If the ticket is not in the ticket file, krb_mk_req() obtains the desired ticket from the KERBEROS server. The calling routine passes the returned authenticator to the service, where it is read by krb_rd_req().

The authenticator cannot be modified without the session key contained in the ticket. The checksum can be used to verify the authenticity of the returned data.

**Return Values:** krb_mk_req() returns an authenticator, which is built in the authent structure and is accessible to the calling procedure. The return code is an index into an array of error messages called krb_err_txt. A return code of KSUCCESS indicates success; otherwise, an error.

**krb_mk_safe()**
**krb_mk_safe()**

```c
long krb_mk_safe(in, out, in_length, key, sender, receiver)
unsigned char *in;
unsigned char *out;
unsigned long in_length;
C_Block *key;
struct sockaddr_in *sender;
struct sockaddr_in *receiver;
```

**Operand Description**

- **in**: Points to an input structure containing application data.
- **out**: Points to the output structure containing the encrypted data.
- **in_length**: Indicates the length of the application data pointed to by `in`.
- **key**: Points to a private session key.
- **sender**: Specifies the fully qualified domain name of the sender.
- **receiver**: Specifies the fully qualified domain name of the receiver.

**Description**: The `krb_mk_safe()` routine constructs an AUTH_MSG_SAFE message. The routine takes user data pointed to by `in` of length `in_length`. The `krb_mk_safe()` routine then creates a packet in `out` consisting of the user data, a time stamp, the Kerberos protocol version, the host byte order, and the network addresses of the sender and receiver. A checksum is derived from this information using the specified private session key. This protocol message does not provide privacy (the data is not encrypted), but it does protect against modifications, insertions, or replays. The message is received and verified using the `krb_rd_safe()` function.

The authentication provided by this routine is not as stringent as that provided by `krb_mk_priv()`.

**Return Values**: `krb_mk_safe()` places the encapsulated message and header information in the area pointed to by `out`. The length of the output is returned upon success; the value −1 indicates an error.

**krb_rd_err()**

```c
int krb_rd_err(in, in_length, code, msg_data)
unsigned char *in;
unsigned long in_length;
long *code;
MSG_DAT *msg_data;
```

**Operand Description**

- **in**: Points to the beginning of the received message.
- **in_length**: Indicates the length of the received message pointed to by `in`.
- **code**: Points to a value filled with the error value provided by the application.
- **msg_data**: Points to the MSG_DAT structure, defined in KRB.H, which is filled by `krb_rd_err()`.

**Description**: The `krb_rd_err()` routine unpacks a message received from `krb_mk_err()`, and fills the following MSG_DAT fields:

**Operand Description**

- **app_data**: Points to the application error text.
- **app_length**: Indicates the `in_length` specified by the calling routine.

`krb_rd_err()` detects host byte order differences and swaps bytes accordingly.
Return Values: krb_rd_err() places the decrypted message and header information in the area pointed to by msg_data. The value 0 (RD_AP_OK) indicates success. Other return codes that indicate failure are:

- RD_AP_VERSION
- RD_AP_MSG_TYPE

See the KRB.H header file for a description of these return codes. See the PROT.H header file for the definition, current protocol version, and possible Kerberos message types.

**krb_rd_priv()**

```c
long krb_rd_priv(in, in_length, schedule, key, sender, receiver, msg_data)
unsigned char *in;
unsigned long in_length;
des_key_schedule schedule;
C_BLock key;
struct sockaddr_in *sender;
struct sockaddr_in *receiver;
MSG_DAT *msg_data;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Points to the beginning of the received message.</td>
</tr>
<tr>
<td>in_length</td>
<td>Indicates the length of the received message pointed to by in.</td>
</tr>
<tr>
<td>schedule</td>
<td>Specifies the session key schedule.</td>
</tr>
<tr>
<td>key</td>
<td>Points to a private session key.</td>
</tr>
<tr>
<td>sender</td>
<td>Specifies the fully qualified domain name of the sender to be checked against the message pointed to by in.</td>
</tr>
<tr>
<td>receiver</td>
<td>Specifies the fully qualified domain name of the receiver to be checked against the message pointed to by in.</td>
</tr>
<tr>
<td>msg_data</td>
<td>Points to the MSG_DAT structure, defined in KRB.H, which is filled by krb_rd_priv().</td>
</tr>
</tbody>
</table>

Description: The krb_rd_priv() routine decrypts and authenticates a message received from krb_mk_priv(), and, if successful, fills the following MSG_DAT fields:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_data</td>
<td>Points to the decrypted application data.</td>
</tr>
<tr>
<td>app_length</td>
<td>Indicates the length of the app_data field.</td>
</tr>
<tr>
<td>time_sec</td>
<td>Specifies the time stamps in the message.</td>
</tr>
<tr>
<td>time_5ms</td>
<td>Specifies the time stamps in the message.</td>
</tr>
</tbody>
</table>

krb_rd_priv() detects host byte order differences and swaps bytes accordingly. krb_rd_priv() checks for additional errors (see Return Values).

Return Values: krb_rd_priv() places the decrypted message and header information in the area pointed to by msg_data. The value 0 (RD_AP_OK) indicates success; a return code indicates an error. Valid error codes are:

- RD_AP_VERSION
- RD_AP_MSG_TYPE
- RD_AP_MODIFIED
- RD_AP_TIME
krb_rd_prv()

See the KRB.H header file for a description of these return codes. See the PROTH.H header file for the definition, current protocol version, and possible Kerberos message types.

krb_rd_req()

int krb_rd_req(authent, service, instance, from_addr, ad, filename)
KTEXT authent;
char *service;
char *instance;
unsigned long from_addr;
AUTH_DAT *ad;
char *filename;

Operand                Description
authent               Specifies the authenticator of type KTEXT.
service               Specifies the first part of the Kerberos name (principal name).
instance              Specifies the second part of the Kerberos name.
from_addr             Specifies the address of the host originating the request, obtained from the incoming packet, to check against the client’s host address in the authenticator. This is ignored in the current version.
ad                    Points to the structure AUTH_DAT, which is filled with information obtained from the authenticator.
filename              Specifies an optional file name containing the secret keys for the service.

Description: The krb_rd_req() routine reads an authentication request and returns information about the identity of the requestor or an indication that the identity information was not authentic.

The service and instance operands name the desired service and are used to get the service’s key from a key file to decrypt the ticket in the received message, and compare it against the service name contained in the ticket.

The krb_rd_req() routine is used by a service to obtain information about the principal name when a request has been made to a service. The application protocol passes the authenticator from the client to the service. The authenticator is then passed to krb_rd_req() to extract the desired information.

If the value of filename is a null string, the ETC SRVTAB file (the default key file) is searched to find the secret keys. If the value of filename is NULL, the routine assumes that the keys have been set and does not search for them. For information on ETC SRVTAB, see TCP/IP Planning and Customization.

Return Values: The value 0 (RD_AP_OK) indicates success. If a packet was forged, modified, or replayed, authentication fails. If the authentication fails, a nonzero value is returned indicating the particular problem encountered. Valid error codes are:
• RD_AP_VERSION
• RD_AP_MSG_TYPE
• RD_AP_MODIFIED
• RD_AP_UNDEC
• RD_AP_INCON
• RD_AP_BADD
krb_rd_safe()

```c
long krb_rd_safe(in, in_length, key, sender, receiver, msg_data)
unsigned char *in;
unsigned long in_length;
C_Block *key;
struct sockaddr_in *sender;
struct sockaddr_in *receiver;
MSG_DAT *msg_data;
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Points to the beginning of the received message.</td>
</tr>
<tr>
<td>in_length</td>
<td>Indicates the length of the received message pointed to by in.</td>
</tr>
<tr>
<td>key</td>
<td>Points to a private session key.</td>
</tr>
<tr>
<td>sender</td>
<td>Specifies the fully qualified domain name of the sender to be checked against the message pointed to be in.</td>
</tr>
<tr>
<td>receiver</td>
<td>Specifies the fully qualified domain name of the receiver to be checked against the message pointed to be in.</td>
</tr>
<tr>
<td>msg_data</td>
<td>Points to the MSG_DAT structure, defined in KRB.H, which is filled by krb_rd_safe().</td>
</tr>
</tbody>
</table>

**Description:** The krb_rd_safe() routine authenticates a message received from krb_mk_safe(), and, if successful, fills the following MSG_DAT fields:

**Operand**  | **Description** |
-------------|-----------------|
app data     | Points to the decrypted application data.       |
app_length    | Indicates the length of the app_data field.     |
time_sec      | Specifies the time stamps in the message.       |
time_5ms      | Specifies the time stamps in the message.       |

krb_rd_safe() detects host byte order differences and swaps bytes accordingly. krb_rd_safe() checks for additional errors (see Return Values).

**Return Values:** The authenticated message is placed in the area pointed to by msg_data. The value 0 (RD_AP_OK) indicates success; otherwise, a return code indicates an error. Valid error codes are:

- RD_AP_VERSION
- RD_AP_MSG_TYPE
- RD_AP_MODIFIED
- RD_AP_TIME

See the KRB.H header file for a description of these return codes. See the PROT.H header file for the definition, current protocol version, and possible Kerberos message types.

krb_recvauth()
**krb_recvauth()**

```c
int krb_recvauth(options, fd, ticket, service, instance, faddr, laddr, kdata, filename, schedule, version)
```

- `options`: Indicates a bit-field of selected options. The only option valid for `krb_recvauth()` is KOPT_DO_MUTUAL.
- `fd`: Specifies the socket descriptor from which to read the authentication message (and write to, if mutual authentication is requested).
- `ticket`: Specifies a Kerberos ticket, which is part of the received message sent by the client.
- `service`: Specifies the first part of the Kerberos name (principal name) of the target service.
- `instance`: Specifies the second part of the Kerberos name of the target service.
- `faddr`: Specifies the network address of the sending host (client).
- `laddr`: Specifies the network address of the local server. Can be NULL, unless mutual authentication is requested.
- `kdata`: Specifies the authentication information extracted from the message.
- `filename`: Specifies the name of the file containing the server’s keys. `filename` is passed to `krb_rd_req()`. If `filename` is NULL, the ETC SRVTAB file is used.
- `schedule`: Specifies the session key schedule.
- `version`: Specifies the version string, which should be large enough to hold a KRB_SENDAUTH_VLEN character string (defined in KRB.H).

**Description:** The `krb_recvauth()` routine is called by the server to verify an authentication message received from a client. The client must use the corresponding routine, `krb_sendauth()`, to prepare and transmit this authentication message. For an example of the usage of `krb_recvauth()`, see “Sample Kerberos Programs” on page 268.

**Return Values:** The integer KSUCCESS indicates that the authentication was successful. KFAILURE indicates that the authentication has failed.

**krb_sendauth()**
The krb_sendauth() routine takes the supplied information and prepares and transmits a ticket over a file descriptor for a desired service, instance, and realm, and performs mutual authentication if requested.

The following options can be specified:
- KOPT_DO_MUTUAL
- KOPT_DONT_CANON
- KOPT_DONT_MK_REQ
krb_sendauth()

The KOPT_DO_MUTUAL option requests mutual authentication. If you select KOPT_DO_MUTUAL, you must supply the checksum, msg_data, cred, schedule, laddr, and faddr variables. The krb_mk_priv() routine performs the mutual authentication on the remote side. The krb_rd_priv() routine performs the mutual authentication on the local side.

The KOPT_DONT_CANON option requests that instance not be used as a host name.

The KOPT_DONT_MK_REQ option requests that a server ticket not be supplied by the KERBEROS server. You must supply the ticket variable.

For an example of the usage of krb_sendauth(), see “Sample Kerberos Programs”.

Return Values: KSUCCESS indicates that a ticket was successfully transmitted. KFAILURE indicates an error.

Sample Kerberos Programs

This section provides examples of the following programs:
- Kerberos client (SAMPLE@C C).
- Kerberos server (SAMPLE@S C).

Kerberos Client

The following is an example of a Kerberos client program.

```c
/*
 * $Source: /mit/kerberos/src/appl/sample/RCS/sample_client.c,v $
 * $Author: jtkohl $
 * Copyright 1987, 1988 by the Massachusetts Institute of Technology.
 * For copying and distribution information,
 * please see the file <mit-copyright.h>.
 * sample_client:
 * A sample Kerberos client, which connects to a server on a remote host,
 * at port "sample" (be sure to define it in /etc/services)
 * and authenticates itself to the server. The server then writes back
 * (in ASCII) the authenticated name.
 * Usage:
 * sample_client <hostname> <checksum>
 * <hostname> is the name of the foreign host to contact.
 * <checksum> is an integer checksum to be used for the call to krb_mk_req()
 * and mutual authentication
 */
#define VM
#include <mit-copy.h>
#include <stdio.h>
#include <types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <des_ext.h>
#include <krb_ext.h>
#include <manifest.h>
```
#include <krb.h>
#include <errno.h>
#include <tcperrno.h>

#define SAMPLE_SERVICE "sample"

extern char *malloc();

main(argc, argv)
int argc;
char **argv;
{
    struct servent *sp;
    struct hostent *hp;
    struct sockaddr_in sin, lsin;
    char *remote_host;
    int status;
    int sock, namelen;
    KTEXT_ST ticket;
    char buf[512];
    long authopts;
    MSG_DAT msg_data;
    CREDENTIALS cred;
    Key_schedule sched;
    long cksum;

    if (argc != 3) {
        fprintf(stderr, "usage: %s <hostname> <checksum>
argv[0]);
        exit(1);
    }

    /* convert cksum to internal rep */
    cksum = atol(argv[2]);

    (void) printf("Setting checksum to %ld\n",cksum);

    /* clear out the structure first */
    (void) memset((char *)&sin, 0, sizeof(sin));

    /* find the port number for knetd */
    sp = getservbyname(SAMPLE_SERVICE, "tcp");
    if (!sp) {
        fprintf(stderr, "unknown service %s/tcp; check etc services file\nhostname: %s
argv[1]);
        exit(1);
    }

    /* copy the port number */
    sin.sin_port = sp->s_port;
    sin.sin_family = AF_INET;

    /* look up the server host */
    hp = gethostbyname(argv[1]);
    if (!hp) {
        fprintf(stderr, "unknown host %s\nargv[1]);
        exit(1);
    }

    /* copy the hostname into dynamic storage */
    remote_host = malloc(strlen(hp->h_name) + 1);
    (void) strncpy(remote_host, hp->h_name, strlen(hp->h_name) + 1);

    /* set up the address of the foreign socket for connect() */
    sin.sin_family = hp->h_addrtype;
    (void) memcpy((char *)sin_addr,
                   hp->h_addr,
                   hp->h_length);

    /* stuff... */
Kerberos Client

```c
sizeof(hp->h_addr));
/* open a TCP socket */
sock = socket(PF_INET, SOCK_STREAM, 0);
if (sock < 0) {
    tcperror("socket");
    exit(1);
}
/* connect to the server */
if (connect(sock, &sin, sizeof(sin)) < 0) {
    tcperror("connect");
    close(sock);
    exit(1);
}
/* find out who I am, now that we are connected and therefore bound */
namelen = sizeof(lsin);
if (getsockname(sock, (struct sockaddr *) &lsin, &namelen) < 0) {
    tcperror("getsockname");
    close(sock);
    exit(1);
}
/* call Kerberos library routine to obtain an authenticator,
   pass it over the socket to the server, and obtain mutual
   authentication. */
authopts = KOPT_DO_MUTUAL;
status = krb_sendauth(authopts, sock, &ticket,
    SAMPLE_SERVICE, remote_host,
    NULL, cksum, &msg_data, &cred,
    sched, &lsin, &sin,
    "VERSION9");
if (status != KSUCCESS) {
    fprintf(stderr,
        "%s: cannot authenticate to server: %s\n",
        argv[0], krb_err_txt[status]);
    exit(1);
}
/* After we send the authenticator to the server, it will write
   back the name we authenticated to. Recv what it has to say. */
status = recv(sock, buf, 512,0);
if (status < 0) {
    printf("error: recv\n");
    exit(1);
}
/* make sure it's null terminated before printing */
if (status < 512)
    buf[status] = '\0';
printf("The server says:\n%hs\n", buf);
close(sock);
exit(0);
```

Kerberos Server

The following is an example of a Kerberos server program.

```c
"Source: /mit/kerberos/src/appl/sample/RCS/sample_server.c,v $" "Author: jtkohi $" "Copyright 1987, 1988 by the Massachusetts Institute of Technology."
"For copying and distribution information,"
* sample_server:
  * A sample Kerberos server, which reads a ticket from a TCP socket,
  * decodes it, and writes back the results (in ASCII) to the client.
  *
  * Usage:
  * sample_server
  *
  * file descriptor 0 (zero) should be a socket connected to the requesting
  * client (this will be correct if this server is started by inetd).
  */
#define VM
#include <mit-copy.h>
#include <stdio.h>
#include <types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <krb.h>
#include <krb_ext.h>
#include <des_ext.h>
#include <manifest.h>
#include <netdb.h>
#include <syslog.h>
#include <errno.h>
#include <tcperrno.h>
#define SAMPLE_SERVICE "sample"
#define SAMPLE_SERVER "sample"
#define SRVTAB ""

main()
{
  struct sockaddr_in peername, myname;
  int name_len = sizeof(peername);
  int status, count, len;
  long authopts;
  AUTH_DAT auth_data;
  KTEXT_ST clt_ticket;
  Key_schedule sched;
  char instance[INST_SZ];
  char version[9];
  char retbuf[512];
  char lname[ANAME_SZ];
  int s, ns;
  struct servent *sp;

  openlog("sample_server", 0);

  sp = getservbyname(SAMPLE_SERVICE, "tcp");
  if (!sp) {
    fprintf(stderr,
      "unknown service %s/tcp; check etc services file\n",
      SAMPLE_SERVICE);
    exit(1);
  }
  /* copy the port number */
  myname.sin_port = sp->s_port;
  myname.sin_family = AF_INET;

  s = socket(AF_INET, SOCK_STREAM, 0);
  if (s < 0) {
    printf("sample_s: socket\n");
    exit(1);
  }

  if (bind(s, &myname, sizeof myname) < 0) {
    printf("sample_s: bind\n");
    exit(1);
if (listen(s, 10) < 0) {
    tcperror("sample_s: listen");
    exit(1);
}

again:
    namelen = sizeof(peername);
    ns = accept(s, &peername, &namelen);
    /*
     * To verify authenticity, we need to know the address of the
     * client.
     */
    if (getpeername(ns, (struct sockaddr *)&peername, &namelen) < 0) {
        syslog(LOG_ERR, "getpeername: %m");
        exit(1);
    }
    /* for mutual authentication, we need to know our address */
    namelen = sizeof(myname);
    if (getsockname(ns, (struct sockaddr *)&myname, &namelen) < 0) {
        syslog(LOG_ERR, "getsockname: %m");
        exit(1);
    }
    /* read the authenticator and decode it. Since we
don't care what the instance is, we use "*" so that krb_rd_req
will fill it in from the authenticator */
    (void) strcpy(instance, "*");
    /* we want mutual authentication */
    authopts = KOPT_DO_MUTUAL;
    status = krb_recvauth(authopts, ns, &clt_ticket, SAMPLE_SERVER, instance, &peername, &myname, &auth_data, SRVTAB, sched, version);
    if (status != KSUCCESS) {
        syslog(LOG_ERR, "Kerberos error: %s
", krb_err_txt[status]);
        (void) sprintf(retbuf, "Kerberos error: %s
", krb_err_txt[status]);
    } else {
        /* Check the version string (8 chars) */
        if (strcmp(version, "VERSION9") != 0) {
            /* could do something different, but we just log an error
            and continue */
            version[8] = '\0'; /* make sure null term */
            syslog(LOG_ERR, "Version mismatch: '%s' isn't 'VERSION9'",
                   version);
        }
        /* now that we have decoded the authenticator, translate
         the kerberos principal.instance@realm into a local name */
        if (krb_kntoln(&auth_data, lname) != KSUCCESS)
            strcpy(lname, "*No local name returned by krb_kntoln*");
        /* compose the reply */
        sprintf(retbuf,
            "You are %s.%s@%s (local name %s),
  at address %s,
  version %s,
  cksum %ld
",
            auth_data.pname,
            auth_data.pinst,
            auth_data.prealm,
            lname,
            inet_ntoa(peername.sin_addr),
            version,
            auth_data.checksum);
    }

{A Kerberos Server

Kerberos Server

}
/* write back the response */
if ((count = send(ns, retbuf, (len = strlen(retbuf) + 1), 0)) < 0) {
    syslog(LOG_ERR, "write: %m");
    exit(1);
} else if (count != len) {
    syslog(LOG_ERR, "write count incorrect: %d != %d
", count, len);
    exit(1);
}

/* close up and exit */
close(ns);
goto again;
exit(0);
Kerberos Server
Chapter 8. SNMP Agent Distributed Program Interface

The Simple Network Management Protocol (SNMP) agent distributed program interface (DPI) permits end users to dynamically add, delete, or replace management variables in the local Management Information Base (MIB) without requiring you to recompile the SNMP agent.

SNMP Agents and Subagents

SNMP defines an architecture that consists of network management stations (SNMP clients), network elements (hosts and gateways), and network management agents and subagents. The network management agents perform information management functions, such as gathering and maintaining network performance information and formatting and passing this data to clients when requested. This information is collectively called the Management Information Base (MIB). For more information about clients, agents, and the MIB, see the TCP/IP User’s Guide.

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. The subagent then 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.

To allow the subagents to perform these functions, the SNMP 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 potentially results in a different TCP port being used.

A subagent of the SNMP agent determines the port number by sending a GET request for the MIB variable, which represents the value of the TCP port. The subagent is not required to create and parse SNMP packets, because the DPI C language application program interface (API) has a library routine query_DPI_port(). This routine handles the GET request and response called Protocol Data Units (PDUs) necessary to obtain the port number of the TCP port used by the agent for DPI requests. After the subagent obtains the value of the DPI TCP port, it should make a TCP connection to the appropriate port. After a successful connect(), the subagent registers 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.

Note: Although TCP/IP for VM V2R4 supports SNMP DPI 1.0 subagents, you should recompile and link-edit the SNMP DPI subagents when upgrading to DPI 1.1.
**SNMP Agent Distributed Program Interface**

**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 initiate only two requests: REGISTER and TRAP. For an overview of the SNMP DPI, see Figure 34.

---

**Notes:**

1. The SNMP agent communicates with the SNMP manager by the standard SNMP protocol.
2. The SNMP agent communicates with the TCP/IP layers and kernel (operating system) in an implementation-dependent manner. It implements the standard MIB II view.
3. An SNMP Subagent, running as a separate process (potentially even on another machine), can register objects with the SNMP agent.
4. The SNMP agent decodes SNMP Packets. If such a packet contains a Get, GetNext or Set request for an object registered by a subagent, it sends the request to the subagent by a query packet.
5. The SNMP subagent sends responses back by a reply packet.
6. The SNMP agent then encodes the reply into an SNMP packet and sends it back to the requesting SNMP manager.
7. If the subagent wants to report an important state change, it sends a trap packet to the SNMP agent, which encodes it into an SNMP trap packet and sends it to the manager(s).

---

**Figure 34. SNMP DPI overview**

---

---

---
Processing a GET Request

The DPI packet is parsed, using the pDPIpacket() routine, 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:

```c
unsigned char *cp;

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

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

```c
char *obj_id;
unsigned char *cp;
struct dpi_set_packet *ret_value;
char *data;
/* obj_id = object ID of variable, like 1.3.6.1.2.1.1.1 */
/* should be identical to object ID sent in GET request */
data = a string to be returned;
ret_value = mkDPIset(obj_id, SNMP_TYPE_STRING,
                     strlen(data) + 1, data);
```

Processing a SET Request

Processing a SET request is similar to processing a GET request, but you must pass 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 operands: 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 first determine the reason for the request to then determine the path to traverse in the MIB tree. The second operand 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 will call on the subagent again and pass 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 is 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:

```c
unsigned char *cp;

cp = mkDPIregister(’1.3.6.1.2.1.1.2.’);
```

**Note:** Object IDs are registered with a trailing dot ("."). Although DPI 1.0 level did accept an Object ID without a trailing dot, the new level (DPI 1.1) does not.

**Processing a TRAP Request**

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

**Compiling and Linking**

To compile your program, you must include the SNMP_DPI.H header file.

To compile and link your applications, use the following procedures:

1. To set up the C environment, enter the following commands:
   ```
   SET LDRTBLS nn
   GLOBAL LOADLIB SCEERUN
   GLOBAL TXTLIB SCEELKED
   ```

2. To compile your program, enter one of the following commands:
   - Place compile options on the CC command:
     ```
     CC filename (def(VM)
     ```
   - Place #define VM in the first line of all user’s C source files:
     ```
     CC filename
     ```

3. To generate an executable module, enter the following command:
SNMP Agent Distributed Program Interface

TCPLOAD load_list control_file c (TXTLIB DPILIB)

Notes:
1. It is necessary to global CMSLIB TXTLIB only when running in 370 mode.
2. Make sure you have access to the IBM C for VM/ESA Compiler and to the TCPMAINT 592 minidisk.
3. For the syntax of the TCPLOAD EXEC, see Appendix A, “TCPLOAD EXEC” on page 355 and for the syntax of the SET LDRTBLS command, see the z/VM: CMS Command and Utility Reference.

SNMP DPI Reference

The following table provides a reference for SNMP DPI. Table 72 describes each SNMP DPI routine supported by TCP/IP, and identifies the page in the book where you can find more information.

Table 72. SNMP DPI Reference

<table>
<thead>
<tr>
<th>SNMP DPI Routine</th>
<th>Description</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPIdebug()</td>
<td>Used to turn some DPI internal tracing on or off.</td>
<td>279</td>
</tr>
<tr>
<td>fDPIparse()</td>
<td>Frees a parse tree previously created by a call to pDPIpacket().</td>
<td>280</td>
</tr>
<tr>
<td>mkDPIlist()</td>
<td>Creates the portion of the parse tree that represents a list of name and</td>
<td>280</td>
</tr>
<tr>
<td></td>
<td>value pairs.</td>
<td></td>
</tr>
<tr>
<td>mkDPIregister()</td>
<td>Creates a register request packet and returns a pointer to a static buffer.</td>
<td>281</td>
</tr>
<tr>
<td>mkDPIresponse()</td>
<td>Creates a response packet.</td>
<td>281</td>
</tr>
<tr>
<td>mkDPIset()</td>
<td>Creates a representation of a parse tree name and value pair.</td>
<td>282</td>
</tr>
<tr>
<td>mkDPItrap()</td>
<td>Creates a trap request packet.</td>
<td>283</td>
</tr>
<tr>
<td>mkDPItrape()</td>
<td>Creates an extended trap. Basically the same as the mkDPItrap() routine</td>
<td>284</td>
</tr>
<tr>
<td></td>
<td>but allows you to pass a list of variables and an enterprise object ID.</td>
<td></td>
</tr>
<tr>
<td>pDPIpacket()</td>
<td>Parses a DPI packet and returns a parse tree representation.</td>
<td>285</td>
</tr>
<tr>
<td>query_DPI_port()</td>
<td>Determines what TCP port is associated with DPI.</td>
<td>286</td>
</tr>
</tbody>
</table>

DPI Library Routines

This section provides the syntax, operands, and other appropriate information for each DPI routine supported by TCP/IP Level 310 for VM.

DPIdebug()

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

void DPIdebug(onoff)
int *onoff;
```
### DPIdebug()

**Operand**

<table>
<thead>
<tr>
<th>onoff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies an integer. A value of 0 turns tracing off and a value of 1 (or nonzero) turns tracing on.</td>
</tr>
</tbody>
</table>

**Description:** The DPIdebug() routine can be used to turn DPI internal tracing on or off.

### fDPIparse()

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

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

**Operand**

| 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(), no further references to the parse tree can be made.

### mkDPIlist()

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

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

**Operand**

| packet |
| Specifies a pointer to a structure dpi_set_packet. |
| oid_name |
| Specifies the object identifier of the variable. |
| type |
| Specifies the type of the value. |
| len |
| Specifies the length of the value. |
| value |
| Specifies 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, then a new `dpi_set_packet` structure is dynamically allocated and the pointer to that structure is returned. The structure contains the new name and value pair. If the pointer `packet` is not NULL, then a new `dpi_set_packet` structure is dynamically allocated and chained to the list. The new structure contains the new name and value pair. The pointer `packet` is 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 values are defined in the `snmp_dpi.h` header file.
As a result, the structure dpi_set_packet has changed and now has a next pointer (zero in case of a mkDPIset() call and also zero upon the first mkDPIlist() call). The following is the format of dpi_set_packet:

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

A subagent writer would normally look only at the dpi_set_packet structure when receiving a SNMP_DPI_SET request and after having issued a pDPIpacket() call.

### mkDPIregister()

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

unsigned char *mkDPIregister(oid_name)
char *oid_name;

Operand Description
oid_name Specifies the object identifier of the variable to be registered. Object identifiers are registered with a trailing dot (".").

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 two bytes of the packet.

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

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

```c
unsigned char *packet;
int len;

/* register sysDescr variable */
packet = mkDPIregister("1.3.6.1.2.1.1.1.");

len = *packet * 256 + *(packet + 1);
len += 2; /* include length bytes */
```

### mkDPIresponse()

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

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

Operand Description
ret_code Determines the error code to be returned.
value_list Points to a parse tree containing the name, type, and value information to be returned.
```
**mkDPIresponse()**

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

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

See the SNMP_DPI.H header file for a description of these messages.

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

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

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

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

**Example:** The following is an example of the mkDPIresponse() routine.

```c
unsigned char *packet;
int error_code;
struct dpi_set_packet *ret_value;
packet = mkDPIresponse(error_code, ret_value);
len = *packet * 256 + *(packet + 1);
len += 2; /* include length bytes */
```

**mkDPIset()**

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

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

**Operand Description**

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid_name</td>
<td>Specifies the object identifier of the variable.</td>
</tr>
<tr>
<td>type</td>
<td>Specifies the type of the object identifier.</td>
</tr>
<tr>
<td>len</td>
<td>Indicates the length of the value.</td>
</tr>
<tr>
<td>value</td>
<td>Points to the first byte of the value of the object identifier.</td>
</tr>
</tbody>
</table>

**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 there is an error detected while creating the parse tree, a NULL pointer is returned.
The value of **type** can be one of the following (which are defined in the SNMP_DPI.H header file):
- SNMP_TYPE_NUMBER
- SNMP_TYPE_STRING
- SNMP_TYPE_OBJECT
- SNMP_TYPE INTERNET
- SNMP_TYPE_COUNTER
- SNMP_TYPE_GAUGE
- SNMP_TYPE_TICKS

The **value** operand is always a pointer to the first byte of the object ID’s value.

**Note:** The parse tree is dynamically allocated, and copies are made of the passed operands. After a successful call to **mkDPIset()**, the application can dispose of the passed operands 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()**

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

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

**Operand**  
**Description**
- **generic**: Specifies the generic field in the SNMP TRAP packet.
- **specific**: Identifies the specific field in the SNMP TRAP packet.
- **value_list**: Passes 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 two 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 is an example of the **mkDPItrap()** routine.

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

data = 3; /* interface number = 3 */
if_index_value = mkDPIset("1.3.6.1.2.1.2.2.1.1", SNMP_TYPE_NUMBER,
```
`mkDPItrap()`

```c
sizeof(unsigned long), &data);
packet = mkDPItrap(2, 0, if_index_value);
len = *packet * 256 + *(packet + 1);
len += 2; /* include length bytes */
write(fd, packet, len);
```

`mkDPItrape()`

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

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

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generic</td>
<td>Specifies the generic field for the SNMP TRAP packet.</td>
</tr>
<tr>
<td>specific</td>
<td>Specifies the specific field for the SNMP TRAP packet.</td>
</tr>
<tr>
<td>value_list</td>
<td>Specifies 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 send.</td>
</tr>
<tr>
<td>enterprise_oid</td>
<td>Specifies 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). Specifies NULL if you want the SNMP agent to use its own enterprise object ID.</td>
</tr>
</tbody>
</table>

**Description:** The `mkDPItrape()` routine can be used to create an extended trap. An extended trap resembles the `mkDPItrap()` routine, but it allows you to pass a list of variables and an enterprise-object ID.

The structure for dpi_trap_packet has changed, but this structure is not exposed to subagent writers.

**Example of an Extended Trap**

The following is a piece of sample code to send an extended trap. No error checking is done.

```c
struct dpi_set_packet *set;
int len;
long int num = 15; /* 4 octet integer */
unsigned long int ctr = 1234;
char *str[] = "a string";
unsigned char *packet;

set = 0;
set = mkDPIlist(set,"1.3.6.1.4.1.2.1.4.1",SNMP_TYPE_NUMBER,sizeof(num),&num);
set = mkDPIlist(set,"1.3.6.1.4.1.2.1.4.2",SNMP_TYPE_STRING,strlen(str),str);
set = mkDPIlist(set,"1.3.6.1.4.1.2.1.4.6",SNMP_TYPE_COUNTER,sizeof(ctr),&ctr);

packet = mkDPItrape(6L, 37L, set, "1.3.6.1.4.1.2.1.4");
len = *packet * 256 + *(packet+1);
len += 2;
write(fd, packet, len) /* use send on OS/2 */
```
You can use a `mkDPIset()` call to create an initial dpi_set_packet for the first name and value pair. So the following sample is equivalent to the one above.

```c
struct dpi_set_packet *set;
int len;
long int num = 15; /* 4 octet integer */
unsigned long int ctr = 1234;
char str[] = "a string";
unsigned char *packet;
set = mkDPIset("1.3.6.1.4.1.2.2.1.4.1", SNMP_TYPE_NUMBER, sizeof(num), &num);
set = mkDPIlist(set,"1.3.6.1.4.1.2.2.1.4.2", SNMP_TYPE_STRING, strlen(str), str);
set = mkDPIlist(set,"1.3.6.1.4.1.2.2.1.4.6", SNMP_TYPE_COUNTER, sizeof(ctr), &ctr);
packet = mkDPItrape(6L, 37L, set, "1.3.6.1.4.1.2.2.1.4");
len = *packet * 256 + *(packet+1);
len += 2;
write(fd, packet, len) /* use send on OS/2 */
```

If the high order bit must be on for the specific trap type, then a negative integer must be passed.

**pDPIpacket()**

```c
#include <snmp_dpi.h>
#include <types.h>
struct snmp_dpi_hdr *pDPIpacket(packet)
unsigned char *packet;

Operand  Description
---      ------
packet   Specifies the DPI packet to be parsed.

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

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

Note: The parse tree structures are defined in the SNMP_DPI.H header file.

Example: The following is an example of the mkDPItrape() routine. 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.

```c
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.

```c
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.

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

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_NUMBER
- SNMP_TYPE_STRING
- SNMP_TYPE_OBJECT
- SNMP_TYPE_EMPTY
- SNMP_TYPE_INTERNET
- SNMP_TYPE_COUNTER
- SNMP_TYPE_GAUGE
- 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()

The `query_DPI_port()` function retrieves the DPI port number. It takes an argument of type `c_int` and returns a `c_int` value.
query_DPI_port()

```c
#include <snmp_dpi.h>

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

**Operands**

**host_name**
Points to the SNMP agent’s host name or internet address.

**community_name**
Points to the community name to be used when making a request.

**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. `community_name` and `host_name` are the arguments that are passed to the `query_DPI_port()` routine.

**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**

This section provides an example of an SNMP DPI agent program. You can run the dpisample program against the SNMP agents that support the SNMP-DPI interface, as described in RFC 1228.

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

The DPISAMPLE program implements a set of variables in the DPISAMPLE table which consists of a set of objects in the IBM Research tree (1.4.1.2.2.1.4). See Figure 35 on page 283 for the object type and objectID.

---

**The DPISAMPLE Program (Sample DPI Subagent)**

The DPISAMPLE program accepts the following arguments:
The DPISAMPLE Program (Sample DPI Subagent)

Operand Description

?  Invokes output with an explanation about how the dpisample command is used. This option should be used in a C-shell environment.

-d n  Sets the debug level. The level, n, has a range from 0 - 4, 0 is silent and 4 is most verbose. The default level is 0.

-trap  Generates a trap with the following options:

   gtype  Specifies the type as generic. The available ranges are 0 - 6.
   stype  Specifies the type as specific.
   data  Passes data as an additional value for the variable dpiSample.stype.0. Data is interpreted depending on stype. The following list describes the available values for the stype operand and their data descriptions:

1  number
2  octet string
3  object id
4  empty (ignored)
5  internet address
6  counter
7  gauge
8  time ticks
9  display string
other  octet string

-trape  Generates an extended trap (available with DPI 1.1 level) with the following defined options:

   gtype  Specifies the trap as generic. The available ranges are 0 - 6.
   stype  Specifies the type as specific.
   enterprise  Provides the object ID for the extended trap.
   data  Passes data values for additional variables. Data is passed as octet strings. Instances of data can be 1-n.

-ent_traps  Generates nine enterprise-specific traps with stype values of 1 - 9, using the internal dpiSample variables as data.
The DPISAMPLE Program (Sample DPI Subagent)

- **ent_traps** Generates nine enterprise-specific traps with stype values of 11 - 19, using the internal dpiSample variables as data.
- **std_traps** Generates and simulates the standard five SNMP traps (generic types 1 - 5) including the link-down trap.
- **all_traps** Generates both the standard traps (-std_traps) and the enterprise-specific traps with stype of 1 - 9 (-ent_traps).
- **iucv** Specifies that an AF_IUCV socket is to be used to connect to the SNMP agent. The -iucv operand is the default.
- **-u agent_userid** Specifies the user ID where the SNMP agent (SNMPD) is running. The default is SNMPD.
- **-inet** Specifies that an AF_INET socket is to be used 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, if -inet is specified, is LOOPBACK.
- **community_name** Specifies the community name to get the dpiPort. The default is PUBLIC.

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
```

Figure 35. DPISAMPLE Table MIB descriptions

Client Sample Program

The following is an example of a SNMP-DPI subagent program.
```
/***********************************************************************************/
/*/ * 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 *
/*/ * Note: dpisample = dpisampl on OS/2 *
/*/ */
```
/* Copyright None */
/* */
/* dpisample.c - a sample SNMP-DPI subagent */
/* can be used to test agent DPI implementations. */
/* */
/* 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, MIB DESC DATA for SQESERV on VM and */
/* MIBDESC.DATA for SQESERV on MVS, MIB2TBL on OS/2). */
/* */
/*************************************************************************/
#define OID "1.3.6.1.4.1.2.2.1.4."
#define ENTERPRISE_OID "1.3.6.1.4.1.2.2.1.4" /* dpiSample */
#define ifIndex "1.3.6.1.2.1.2.2.1.1.0"
#define egpNeighAddr "1.3.6.1.2.8.5.1.2.0"
#define PUBLIC_COMMUNITY_NAME "public"
#if defined(VM) || defined(MVS)
    #define SNMPAGENTUSERID "SNMPD"
    #define SNMPIUCVNAME "SNMP_DPI"
    #pragma csect(CODE, "$DPISAMP")
    #pragma csect(STATIC,"$DPISAMP")
    #include <manifest.h> /* VM specific things */
    #include "snmpnms.h" /* short external names for VM/MVS */
    #include "snmp_vm.h" /* more of those short names */
    #include <saiucv.h>
    #include <bsdtime.h>
    #include <bsdtypes.h>
    #include <socket.h>
    #include <in.h>
    #include <netinet.h>
    #include <netdb.h>
    #include <inet.h>
    extern char ebcdicto[], asciiito[];
    #pragma linkage(cmxlrate,OS)
    #define DO_ETOA(a) cmxlrate((a),ebcdictoascii,strlen((a)))
    #define DO_ATOE(a) cmxlrate((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 */
#endif
#ifdef OS2
    #define INCL_DOSPROCESS
    #include <stdlib.h>
    #include <types.h>
    #include <doscalls.h> /* GKS */
    #include <os2.h> /* GKS */
#endif
#define sleep(a) DosSleep(1000L * (a)) /*GKS*/
#define close soclose
/*char * malloc(); */
/*unsigned long strtoul(); */
#endif
#include <sys/time.h> /* GKS */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
Client Sample Program

```c
#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
#if defined(AIX221)
  #define isdigit(c) (((c) >= '0') && ((c) <= '9'))
#else
  #include <sys/select.h>
#endif /* AIX221 */
#endif /* defined(VM) || defined(MVS) */

#include <stdio.h>
#define WAIT_FOR_AGENT 3 /* time to wait before closing agent fd */
#endif /* defined(TRUE) */
#define TRUE 1
#define FALSE 0
#endif /* _NO_PROTO */ /* for classic K&R C */

#ifndef NO_PROTO
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);
#endif /* _NO_PROTO */
```

Chapter 8. SNMP Agent Distributed Program Interface 291
Client Sample Program

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);
static 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 */
    SNMP_TYPE_EMPTY/* 4 do not check type */
    SNMP_TYPE_INET, /* 5 internet address */
    SNMP_TYPE_COUNTER, /* 6 counter */
    SNMP_TYPE_GAUGE, /* 7 gauge */
    SNMP_TYPE_TICKS, /* 8 time ticks */
    SNMP_TYPE_STRING, /* 9 display string */
    SNMP_TYPE_STRING /* 10 command (display string) */
};
#define OID_COUNT_FOR_TRAPS 9
#define OID_COUNT 10

static char *packet = NULL; /* ptr to send packet. */
static char inbuf[BUFSIZE]; /* buffer for receive packets */
static int dpi_fd; /* fd for socket to DPI agent */
static short int dpi_port; /* DPI port at agent */
static unsigned long dpi_ipaddress; /* IP address of DPI agent */
static char *dpi_hostname; /* hostname of DPI agent */
static char *dpi_userid; /* userid of DPI agent VM/MVS */
static char *var_gid; /* groupID received */
static char *var_oid; /* objectID received */
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
static void issue_traps()
#else /* _NO_PROTO */
static void issue_traps(void)
#endif /* _NO_PROTO */
{
    switch (do_trap) {
        case ONE_TRAP: /* only need to issue one trap */
            issue_one_trap();
            break;
    }
}

#ifdef _NO_PROTO
main(argc, argv) /* main line */
#else /* _NO_PROTO */
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 */
    if (do_trap) { /* we just need to do traps */
        issue_traps(); /* issue the trap(s) */
        sleep(WAIT_FOR_AGENT); /* sleep a bit, so agent can */
        close(dpi_fd); /* read data before we close */
        exit(0); /* and that's it */
    }
    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 */
    }
    printf("Quitting, %s set to: quit\n", DPI_var10[]);
    exit(2); /* sampleDisplayString == quit */
}

#endif /* _NO_PROTO */

Client Sample Program

Chapter 8. SNMP Agent Distributed Program Interface 293
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 */
{
    len, rc, bytes_to_read, bytes_read = 0;

    #ifdef OS2
    int socks)5[; 
    #else
    fd_set read_mask;
    #endif

    struct timeval timeout;

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

    if (rc != 1) { /* exit on error */
        DO_ERROR("await_and_read_packet: select");
        close(dpi_fd);
        exit(1);
    }

    #ifdef OS2
    len = recv(dpi_fd, inbuf, 2, 0); /* read 2 bytes first */
    #else
    len = read(dpi_fd, inbuf, 2); /* read 2 bytes first */
    #endif

    if (len <= 0) { /* exit on error or EOF */
        if (len < 0) DO_ERROR("await_and_read_packet: read");
        else printf("Quitting, EOF received from DPI-agent\n");
        close(dpi_fd);
        exit(1);
    }

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

#endif /* _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;
    }
Client Sample Program

```c
var_type = hdr->packet_body.dpi_set->type;
break;
default: /* Return a GEN_ERROR */
    if (debug_lvl > 0) printf("Unexpected packet_type %d, genErr\n",
        hdr->packet_type);
    packet = mkDPIresponse(SNMP_GEN_ERR, NULL);
    fDPIparse(hdr); /* return storage allocated by pDPIpacket() */
    send_packet(packet);
    return;
break;
} /* end switch(hdr->packet_type) */
if (debug_lvl > 0) printf("objectID: %s \n",var_oid);
if (strlen(var_oid) <= strlen(OID)) { /* not in our tree */
    if (hdr->packet_type == SNMP_DPI_GET_NEXT) var_index = 0; /* OK */
    else { /* cannot handle */
        if (debug_lvl>0) printf("...Ignored %s, noSuchName\n",var_oid);
        packet = mkDPIresponse(SNMP_NO_SUCH_NAME, NULL);
        fDPIparse(hdr); /* return storage allocated by pDPIpacket() */
        send_packet(packet);
        return;
    } else { /* Extract our variable index (from OID.index.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;
}
} /* end switch(hdr->packet_type) */
#endif /* _NO_PROTO */ /* for classic K&R C */
static void do_get() /* handle SNMP_GET request */
#else /* _NO_PROTO */ /* for ANSI-C compiler */
#endif /* _NO_PROTO */
{
    struct dpi_set_packet *data = NULL;

    switch (var_index) {
    case 0: /* table, cannot be queried by itself */
        printf("...Should not issue GET for table %s.0\n", OID);
        break;
    case 1: /* a number */
        data = mkDPIset(var_oid,SNMP_TYPE_NUMBER,sizeof(number),&number);
        break;
    case 2: /* an octet_string (can have binary data) */
        ...
    }
```
data = mkDPIset(var_oid, SNMP_TYPE_STRING, ostring_len, ostring);
break;
case 3: /* object id */
data = mkDPIset(var_oid, SNMP_TYPE_OBJECT, objectID_len, objectID);
break;
case 4: /* some empty variable */
data = mkDPIset(var_oid, SNMP_TYPE_EMPTY, 0, NULL);
break;
case 5: /* internet address */
data = mkDPIset(var_oid, SNMP_TYPE_INTERNET, sizeof(ipaddr), &ipaddr);
break;
case 6: /* counter (unsigned) */
data = mkDPIset(var_oid, SNMP_TYPE_COUNTER, sizeof(counter), &counter);
break;
case 7: /* gauge (unsigned) */
data = mkDPIset(var_oid, SNMP_TYPE_GAUGE, sizeof(gauge), &gauge);
break;
case 8: /* time ticks (unsigned) */
data = mkDPIset(var_oid, SNMP_TYPE_TICKS, sizeof(ticks), &ticks);
break;
case 9: /* a display_string (printable ascii only) */
    DO_ETOA(dstring);
data = mkDPIset(var_oid, SNMP_TYPE_STRING, strlen(dstring), dstring);
    DO_ATOE(dstring);
break;
case 10: /* a command request (command is a display string) */
    DO_ETOA(command);
data = mkDPIset(var_oid, SNMP_TYPE_STRING, strlen(command), command);
    DO_ATOE(command);
break;
default: /* Return a NoSuchName */
    if (debug_lvl > 1)
        printf("...GET\]NEXT\[ for %s, not found\n", var_oid);
break;
} /* end switch (var_index) */

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

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

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

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

```c
trap_data = dstring;
break;
} /* end switch (trap_stype) */
issue_one_trap();
}

/* issue a set of extended traps, pass enterprise ID and multiple */
* variable (assume octet string) as passed by caller */
#ifdef _NO_PROTO /* for classic K&R C */
static void issue_ent_trapse()
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void issue_ent_trapse(void)
#endif /* _NO_PROTO */
{
int i, n;
struct dpi_set_packet *data = NULL;
unsigned char *packet = NULL;
unsigned long ipaddr, ulnum;
char oid[256];
char *cp;
trape_gtype = 6;
trape_eprise = ENTERPRISE_OID;
for (n=11; n < (11+OID_COUNT_FOR_TRAPS); n++) {
data = 0;
trap_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, trap_stype, data, trape_eprise);
if ((debug_lvl > 0) && (packet)) {
printf("sending trape packet: %lu %lu enterprise=%s\n",
trape_gtype, trap_stype, trape_eprise);
}
if (packet) send_packet(packet);
else printf("Could not make trape packet\n");
}

/* issue one extended trap, pass enterprise ID and multiple */
* variable (assume octet string) as passed by caller */
#ifdef _NO_PROTO /* for classic K&R C */
static void issue_one_trape()
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void issue_one_trape(void)
#endif /* _NO_PROTO */
{
struct dpi_set_packet *data = NULL;
unsigned char *packet = NULL;
char oid[256];
char *cp;
int i;
for (i=0; i<trape_datacnt; i++) {
sprintf(oid,"%s2.%d",OID,i);
/* assume an octet_string (could have hex data) */
data = mkDPIlist(data, oid, SNMP_TYPE_STRING,
strlen(trape_data[i]), trape_data[i]);
if (data == 0) {
printf("Could not make dpiset_packet\n");
} else if (debug_lvl > 0) {
```
Client Sample Program

```c
printf("Preparing: oid=%s[ value: ", oid);
printf("\n");
for (cp = trape_data[i]; *cp; cp++) /* loop through data */
  printf("%2.2x",*cp); /* hex print one byte */
printf("\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");
#endif /* _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 so far. */
  /* could put specific handling here if needed/wanted. */
  case 0: /* simulate cold start */
    break;
  case 1: /* simulate warm start */
    break;
  case 4: /* simulate authentication failure */
    strcpy(oid,"none");
    break;
  case 2: /* simulate link down */
    break;
  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),\n                       trap_data);
        break;
      case 3: /* object id */
        data = mkDPIset(oid, SNMP_TYPE_OBJECT,strlen(trap_data) + 1,\n                        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);
        break;
    }
  ```
Client Sample Program

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_TIMETICKS, sizeof(ulnum), &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) */

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("%s\n", trap_data);
    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");

#endif /* _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));
    }
#endif OS2
rc = send(dpi_fd,packet,PACKET_LEN(packet),0);
#else
rc = write(dpi_fd,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) */
```c
#ifdef _NO_PROTO    /* for classic K&R C */
    static void do_register()    /* register our objectID with agent */
#else    /* _NO_PROTO */    /* for ANSI-C compiler */
    static void do_register(void)    /* register our objectID with agent */
#endif    /* _NO_PROTO */
{
    int i, rc;
    char toid[256][;]
    if (debug_lvl > 0) printf("Registering variables:\n");
    for (i=1; i<=OID_COUNT; i++) {
        sprintf(toid,"%s%d.",OID,i);
        packet = mkDPIRegister(toid);
        #ifdef OS2
            rc = send(dpi_fd, packet, PACKET_LEN(packet),0);
        #else
            rc = write(dpi_fd, packet, PACKET_LEN(packet));
        #endif
        if (rc <= 0) {
            DO_ERROR("do_register: write");
            printf("Quitting, unsuccessful register for %s\n",toid);
            close(dpi_fd);
            exit(1);  
        }
        if (debug_lvl > 0) {
            printf("...Registered: %-25s oid: %s\n",DPI_var[i],toid);
            printf("......Initial value: ");
            print_val(i); /* prints \n at end */
        }
    }
}

/* add specified variable to list of variable in the dpi_set_packet */
#ifdef _NO_PROTO    /* for classic K&R C */
    struct dpi_set_packet *addtoset(data, stype)
    struct dpi_set_packet *data;
    int stype;
#else    /* _NO_PROTO */    /* for ANSI-C compiler */
    struct dpi_set_packet *addtoset(struct dpi_set_packet *data, int stype)
#endif    /* _NO_PROTO */
{
    char var_oid[256][]
    sprintf(var_oid,"%s%d.0",OID, stype);
    switch (stype) {
    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;
    }
```
Client Sample Program

case 6: /* counter (unsigned) */
data = mkDPIlist(data, var_oid, SNMP_TYPE_COUNTER,
    sizeof(counter), &counter);
    break;
case 7: /* gauge (unsigned) */
data = mkDPIlist(data, var_oid, SNMP_TYPE_GAUGE,
    sizeof(gauge), &gauge);
    break;
case 8: /* time ticks (unsigned) */
data = mkDPIlist(data, var_oid, SNMP_TYPE_TICKS,
    sizeof(ticks), &ticks);
    break;
case 9: /* a display_string (printable ascii only) */
    DO_ETOA(dstring);
data = mkDPIlist(data, var_oid, SNMP_TYPE_STRING,
    strlen(dstring), dstring);
    DO_ATOE(dstring);
    break;
} /* end switch (stype) */
return(data);

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

    switch (index) {
    case 1 :
        printf("%ld\n",number);
        break;
    case 2 :
        printf("\n");
        for (cp = ostring; cp < ostring + ostring_len; cp++)
            printf("%2.2x",*cp);
        printf("\n");
        break;
    case 3 :
        printf("%s\n",objectID_len, objectID);
        break;
    case 4 :
        printf("no value (EMPTY)\n");
        break;
    case 5 :
        display_ipaddr.s_addr = (u_long) ipaddr;
        printf("%s\n",inet_ntoa(display_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);

304  z/VM: TCP/IP Programmer's Reference
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 */
    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;
        } 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;
    } else if (strcmp(argv[i],"-inet") == 0) {
        use_iucv = 0;
    } else if (strcmp(argv[i],"-iucv") == 0) {
        use_iucv = TRUE;
    } else if (strcmp(argv[i],"-u") == 0) {
        use_iucv = TRUE; /* -u implies -iucv */
        i++;
        if (argc > i) {
Client Sample Program

```c

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 */
#endif
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;
}
#define _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]-trap g_type s_type enterprise data1 data2 .. datan[\n", strlen(pname)+8,"\n");
    printf("%s]-std_traps[ ]-ent_traps[ ]-ent_trapse[\n", strlen(pname)+8,"\n");
#if defined(VM) || defined(MVS)
    printf("%s]-iucv[ ]-u agent_userid[\n",strlen(pname)+8,"\n");
    printf("%s", strlen(pname)+8,"\n");
    printf("%s]agent_hostname ]community_name[\n"];
    printf("default: -d 0 -iucv -u %s, SNMPAGENTUSERID];
    printf(" -inet %s %s", LOOPBACK, PUBLIC_COMMUNITY_NAME);
#else
    printf("%s]agent_hostname ]community_name[\n",strlen(pname)+8,"\n");
    printf("default: -d 0 %s %s", LOOPBACK, PUBLIC_COMMUNITY_NAME);
#endif
    exit(exit_rc);
}
#define _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));
command = (char *)malloc(strlen(COMMAND)+1);
bcopy(COMMAND,command,strlen(COMMAND));
ipaddr = dpi_ipaddress;

#endif /* _NO_PROTO */
static void init_connection() /* connect to the DPI agent */
#else /* _NO_PROTO */
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 */
    #else
#endif
if (use_iucv == TRUE) {
    printf("Connecting to %s DPI_port %d userid %s (TCP, AF_IUCV)\n", 
dpi_hostname,dpi_port,dpi_userid);
bzero(&siu,sizeof(siu));
siu.siucv_family = AF_IUCV;
siu.siucv_addr = dpi_ipaddress;
siu.siucv_port = dpi_port;
memset(siu.siucv_nodeid, ' ', sizeof(siu.siucv_nodeid));
memset(siu.siucv_userid, ' ', sizeof(siu.siucv_userid));
memset(siu.siucv_name, ' ', sizeof(siu.siucv_name));
bcopy(dpi_userid, siu.siucv_userid, min(8,strlen(dpi_userid)));
bcopy(SNMP_IUCVNAME, siu.siucv_name, min(8,strlen(SNMP_IUCVNAME)));
dpi_fd = socket(AF_IUCV, SOCK_STREAM, 0);
sa = (struct sockaddr *) &siu;
sasize = sizeof(struct sockaddr_iucv);
} else {
    #endif
    printf("Connecting to %s DPI_port %d (TCP, AF_INET)\n", 
dpi_hostname,dpi_port);
bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
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);
#else defined(VM) || defined(MVS)
    }
#endif
if (dpi_fd < 0) { /* exit on error */
    DO_ERROR("init_connection: socket");
    exit(1);
}
Client Sample Program

```c
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 */
}

unsigned long lookup_host(const char *hostname)
{
   register unsigned long ret_addr;

   if ((hostname >= '0') && (hostname <= '9'))
      ret_addr = inet_addr(hostname);
   else {
      struct hostent *host;
      struct in_addr *addr;
      host = gethostbyname(hostname);
      if (host == NULL) return(0);
      addr = (struct in_addr *) (host->h_addr_list[0]);
      ret_addr = addr->s_addr;
   }
   return(ret_addr);
}
```

Compiling and Linking the DPISAMPLE.C Source Code

When compiling the Sample DPI Subagent program you may specify the following compile time flags:

**NO_PROTO**

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

**VM**

Indicates that compilation id for VM and uses VM-specific includes. Some VM/MVS specific code is compiled.
Chapter 9. SMTP Virtual Machine Interfaces

Electronic mail (e-mail) is prepared using local mail preparation facilities (or, user agents) such as the CMS NOTE and SENDFILE commands; these facilities are not discussed here. This chapter describes the interfaces to the SMTP virtual machine itself, and may be of interest to users who implement electronic mail programs that communicate with the IBM z/VM implementation of SMTP.

The interfaces to the SMTP virtual machine are:

- The TCP/IP network
  
  SMTP commands and replies can be sent and received interactively over a TCP network connection. Mail from TCP network sites destined for local VM users (or users on an RSCS network attached to the local z/VM system) arrives over this interface. All commands and data received and transmitted through this interface must be composed of ASCII characters.

- The local z/VM system (and systems attached to the local z/VM system by an RSCS network)
  
  SMTP commands can be written to a batch file and then spooled to the virtual reader of the SMTP virtual machine. SMTP processes each of the commands in this file, in order, as if they had been transmitted over a TCP connection. This is how mail is sent from local z/VM users (or users on an RSCS network attached to the local z/VM system) to recipients on the TCP network. Batch SMTP (or, BSMTP) files must contain commands and data composed of EBCDIC characters.

SMTP Transactions

Electronic mail is sent by a series of request/response transactions between a client, the sender-SMTP, and a server, the receiver-SMTP. These transactions pass (1) the message proper, which is composed of a header and a body (which by definition, are separated by the first blank line present in this information), and (2) SMTP commands, which are referred to as (and comprise) the mail envelope. These commands contain additional information about the mail, such as the host sending the mail and its source and destination addresses. Envelope addresses may be derived from information in the message header, supplied by the user interface, or derived from local configuration information.

The SMTP envelope is constructed at the sender-SMTP site. If this is the originating site, the information is typically provided by the user agent when the message is first queued for the sender-SMTP program. Each intermediate site receives the piece of mail and resends it on to the next site using an envelope that it creates. The content of the new envelope may be different from that of the one it received.

The envelope contains, at a minimum, the HELO or EHLO, MAIL FROM:, RCPT TO:, DATA, and QUIT commands. These, and other commands that can optionally appear in the envelope, are described in the next section. Some of these commands can appear more than once in an envelope. Also, more than one piece of mail can be sent using a given envelope.
SMTP Commands

This section describes SMTP commands that are recognized by the z/VM SMTP implementation. These commands are used to interface with user agent mail facilities (such as the CMS NOTE and SENDFILE commands) as well as with other SMTP servers.

For more complete information about SMTP and the commands that can be used with this protocol, it is suggested that you review the following RFCs:

- RFC 821, Simple Mail Transfer Protocol
- RFC 822, Standard for the Format of ARPA Internet Text Messages
- RFC 1869, SMTP Service Extensions
- RFC 1870, SMTP Service Extension for Message Size Declaration
- RFC 1652, SMTP Service Extension for 8bit-MIME transport

These RFCs are the basis for modern naming specifications associated with the SMTP protocol.

Note: The SMTP commands SEND, SOML, SAML, and TURN are not supported by the z/VM SMTP implementation, so are not described here.

HELO

The HELO command is used to identify the domain name of the sending host to SMTP. This command is used to initiate a mail transaction, and must be sent (once) before a MAIL FROM: command is used.

```
HELO domain_name
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain_name</td>
<td>Specifies the domain name of the sending host. The domain_name may be specified as either:</td>
</tr>
<tr>
<td></td>
<td>• a domain name</td>
</tr>
<tr>
<td></td>
<td>• an IP address in decimal integer form that is prefixed by the number or (US) pound sign (“#” or X’7B’)</td>
</tr>
<tr>
<td></td>
<td>• an IP address in dotted-decimal form, enclosed in brackets.</td>
</tr>
</tbody>
</table>

When HELO commands are received over a TCP connection, SMTP replies with the message 250 SMTP_server_domain is my domain name. The SMTP server client verification exit or built-in client verification function can be used to determine if the provided domain_name matches the client IP address and to include the result of that determination in the mail headers. See [TCP/IP Planning and Customization](#) for detailed information about configuring SMTP to use this support.

When HELO commands are received over a batch SMTP connection, SMTP replies with the message 250 SMTP_server_domain is my domain name. Additional text is included with this message that indicates whether the provided domain_name does or does not match the host name of the spool file origination point. The 250 reply code indicates the HELO command is accepted and that SMTP commands can continue to be sent and received.
The EHLO command operates and can be used in the same way as the HELO command. However, it additionally requests that the returned reply should identify specific SMTP service extensions that are supported by the SMTP server.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain_name</td>
<td>Specifies the domain name of the sending host. The domain_name may be specified as either:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• a domain name</td>
</tr>
<tr>
<td></td>
<td>• an IP address in decimal integer form that is prefixed by the number or (US) pound sign (“#” or X’7B’)</td>
</tr>
<tr>
<td></td>
<td>• an IP address in dotted-decimal form, enclosed in brackets.</td>
</tr>
</tbody>
</table>

If a server does not support SMTP service extensions, the client receives a negative reply to its EHLO command. When this occurs, the client should either supply a HELO command, if the mail being delivered can be processed without the use of SMTP service extensions, or it should end the current mail transaction.

If a client receives a positive response to an EHLO command, the server is then known to support one or more SMTP service extensions. This reply then can be further used by the client to determine whether certain kinds of mail can be effectively processed by that server.

For example, if the positive response includes the SIZE keyword, the server supports the SMTP service extension for Message Size Declaration. Whereas, if this response includes the 8BITMIME keyword, the server supports the SMTP service extension for 8-bit MIME transport.

SMTP supports the following service extensions:

**EXPN HELP SIZE 8BITMIME**

Following is an example of a positive reply to a client (c) EHLO command from an SMTP server (s) that supports these service extensions:

```
s: (wait for connection on TCP port 25)
c: (open connection to server)
s: 220 HOSTA.IBM.COM running IBM VM SMTP Level 320 on Sat, 1 May 99 ...
c: EHLO HOSTB.IBM.COM
s: 250-EHOSTA.IBM.COM is my domain name.
s: 250-EXPN
s: 250-HELP
s: 250-8BITMIME
s: 250 SIZE 524288
...
```

The hyphen (-), when present as the fourth character of a response, indicates the response is continued on the next line.
MAIL FROM

The MAIL FROM: command is used (once), after a HELO or EHLO command, to identify the sender of a piece of mail.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sender_path_address</td>
<td>Specifies the full path address of the sender of the mail. Definitions for valid sender_path_address specifications can be obtained from the RFCs that define the naming conventions used throughout the Internet. For detailed information, consult the RFCs listed in the section &quot;SMTP Commands&quot; on page 310.</td>
</tr>
<tr>
<td>SIZE=number_of_bytes</td>
<td>Specifies the size of the mail, in bytes, including carriage return/line feed (CRLF, X'0D0A') pairs. The SIZE parameter has a range from 0 to 2,147,483,647.</td>
</tr>
<tr>
<td>BODY=7BIT</td>
<td>Specifies that the message is encoded using seven significant bits per 8-bit octet (byte). In practice, however, the body is typically encoded using all eight bits.</td>
</tr>
<tr>
<td>BODY=8BITMIME</td>
<td>Specifies that the message is encoded using all eight bits of each octet (byte) and may contain MIME headers.</td>
</tr>
</tbody>
</table>

Note: The SIZE, BODY=7BIT, and BODY=8BITMIME options of the MAIL FROM: command should be used only if an EHLO command was used to initiate a mail transaction. If an EHLO command was not used for this purpose, SMTP ignores these parameters if they are present.

If the SMTP server is known to support the SMTP service extension for Message Size Declaration, the client sending the mail can specify the optional SIZE= parameter with its MAIL FROM: commands. The client then can use the responses to these commands to determine whether the receiving SMTP server has sufficient resources available to process its mail before any data is transmitted to that server.

When a MAIL FROM: command is received that includes the optional SIZE= parameter, the SMTP server compares the supplied number_of_bytes value to its allowed maximum message size (defined by the MAXMAILBYTES statement in the SMTP CONFIG file) to determine if the mail should be accepted. If number_of_bytes exceeds the MAXMAILBYTES value, a reply code 552 is returned to the client.

The SIZE= parameter is evaluated only for MAIL FROM: commands received over a TCP connection; this parameter and its value are ignored when they are received over a batch connection.
**RCPT TO**

The RCPT TO: command specifies the recipient(s) of a piece of mail. This command can be repeated any number of times.

```
 RCPT TO: <recipient_path_address>
```

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recipient_path_address</td>
<td>Specifies the full path address of a mail recipient. Definitions for valid recipient_path_address specifications can be obtained from the RFCs that define the naming conventions used throughout the Internet. For detailed information, consult the RFCs listed in the section &quot;SMTP Commands&quot; on page 310.</td>
</tr>
</tbody>
</table>

A RCPT TO: command must be used after a MAIL FROM: command. If the host system is not aware of the recipient’s host, a negative reply is returned in response to the RCPT TO: command.

**DATA**

The DATA command indicates that the next information provided by the client should be construed as the text of the mail being delivered (that is, the header and body of the mail message).

```
 DATA
```

The DATA command has no parameters.

The DATA command is used after a HELO or EHLO command, a MAIL FROM: command, and at least one RCPT TO: command have been accepted. When the DATA command has been accepted, the following response (reply code 354) is returned to indicate that the body of the mail can be transmitted:

```
  354 Enter mail body. End new line with just a `'.'`
```

The body of the mail is terminated by transmitting a single ASCII period (.) on a line by itself. When SMTP detects this “end of data” indicator, it returns the following reply:

```
  250 Mail Delivered
```

When mail is received over a TCP connection, this ASCII period should be followed by the ASCII CR-LF sequence (carriage return/line feed sequence, `X'0D0A'`). If any record in the body of the mail begins with a period, the sending SMTP program must convert the period into a pair of periods (..). Then, when the receiving SMTP encounters a record in the body of the mail that begins with two periods, it discards the leading period. This convention permits the mail body to contain records that would otherwise be incorrectly interpreted as the “end of data” indicator. These rules must be followed over both TCP and batch SMTP connections. The CMS NOTE and SENDFILE execs perform this period doubling
SMTP Virtual Machine Interfaces

on all mail spooled to SMTP. If the body of the mail in a batch SMTP command file is not explicitly terminated by a record with a single period, SMTP supplies one.

After the “end of data” indicator has been received, the SMTP connection is reset to its initial state (that is, the state before any sender or recipients have been specified). Additional MAIL FROM:, RCPT TO:, DATA, and other commands can again be sent. If no further mail is to be delivered through this connection, the connection should then be terminated with a QUIT command. If the QUIT command is omitted from the end of a batch SMTP command file, the QUIT is implicit — SMTP will proceed as if it had been provided.

If SMTP runs out of local mail storage space, it returns a 451 reply code to the sender-SMTP client. Local mail storage space is constrained by the size of the SMTP server A-disk (191 minidisk). For a large batch SMTP file, disk storage equivalent to four times the size of that file may be required for it to be processed by SMTP.

If the body of the mail being delivered is found to exceed the MAXMAILBYTES value established in the SMTP CONFIG file, a reply code 552 is returned to the client. See the TCP/IP Planning and Customization for more information about the MAXMAILBYTES configuration statement.

When mail arrives over a batch SMTP connection from an RSCS network host, and the REWRITE822HEADER configuration option was specified in the SMTP configuration file, then header fields are modified to ensure that all addresses are fully qualified domain names. See the TCP/IP Planning and Customization for more information about the header rewriting.

RSET

The RSET command resets an SMTP connection to an initial state. That is, all information about the current mail transaction is discarded, and the connection is ready to process a new mail transaction.

```
>>> RSET
```

The RSET command has no parameters.

QUIT

The QUIT command terminates an SMTP connection.

```
>>> QUIT
```

The QUIT command has no parameters.

NOOP

The NOOP command has no intrinsic function. However, it will cause the receiver-SMTP to return an “OK” response (reply code 250).
The NOOP command has no parameters.

**HELP**

The HELP command returns brief information about one or more SMTP commands.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command_name</td>
<td>Identifies a specific SMTP command.</td>
</tr>
</tbody>
</table>

The HELP command returns a multiple-line reply with brief help information about the SMTP commands supported by a host. If an SMTP command is specified for command_name, information about that specific command is returned.

**QUEU**

The QUEU command returns a multiple-line reply with information about the content of the mail processing queues maintained within the SMTP server.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>Causes information about the age of any queued mail to be included in the QUEU command response. By default, age information is not included in such responses.</td>
</tr>
</tbody>
</table>

The z/VM SMTP server maintains various internal queues for handling mail, that can be generalized to two categories — the mail (delivery) queues, and the mail resolution (or, resolver) queues. The QUEU command returns a multiple-line reply with information about the content of these queues, which are described in more detail here.

**Mail Delivery Queues:**

<table>
<thead>
<tr>
<th>Queue Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spool</td>
<td>Contains mail that is destined for recipients on the local z/VM system, or for recipients on an RSCS system attached to the local z/VM system. This queue is generally empty, because SMTP can...</td>
</tr>
</tbody>
</table>
SMTP Virtual Machine Interfaces

deliver this mail quickly by spooling it directly to the local recipient, or to the RSCS virtual machine for delivery to an RSCS network recipient.

Active
Identifies mail that is currently being transmitted by SMTP to a TCP network destination. All mail queued for that same destination is shown to be Active.

Queued
Identifies mail that has arrived over either a TCP or batch SMTP connection that is to be forwarded to a TCP network destination (possibly because of source routing). When SMTP obtains sufficient resources from the TCPIP virtual machine to process this mail, it is transferred to the Active queue.

Retry
Identifies mail for which SMTP has made one or more previous delivery attempts that were not successful. Delivery attempts may fail for a variety of reasons; two common reasons are:

• The SMTP server could not open a connection to deliver the mail.

• Delivery of the mail was interrupted for some reason, such as a broken connection or a temporary error condition at the target host.

After the RETRYINT interval (defined in the SMTP CONFIG file) has passed, mail in the Retry queue is promoted to the Queued queue (or state) for another delivery attempt. For more information about the RETRYINT configuration parameter, see the TCP/IP Planning and Customization.

Undeliverable
Identifies mail that SMTP cannot deliver to a local z/VM recipient, or to a recipient on the RSCS network attached to the local VM system, due to insufficient spooling resources on the local z/VM system. After spool space has been increased and SMTP has been reinitialized, delivery of this mail is again attempted.

Mail Resolution Queues:

The mail resolution (or, resolver) queues are used to maintain the status of host resolution queries — performed through DNS services — for mail host domains, originators, and recipients, when such resolution is necessary. If the SMTP server is configured to not use a name server, but only local host tables, these queues are not used.

Several notes regarding the response information associated with mail resolution queues follow:

• If a queue is empty the word Empty appears in the response, to the right of the name of that queue.

• If a queue contains active queries, a line that identifies that queue will be present; information about the mail in that queue, and its associated query (or queries) is provided immediately after this identification line.

• Because of timing situations that can occur within the SMTP server, a queue identification line may at times show that a queue is active (that is, Empty is not indicated), but no mail entries will be present.

<table>
<thead>
<tr>
<th>Queue Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>This queue is generally empty, as it contains queries that have not</td>
</tr>
</tbody>
</table>
yet been acted upon by the SMTP server. Once a query has been initially processed, it is placed in the *Resolver Send* queue.

**Send** Identifies queries that are awaiting SMTP resolver processing. SMTP staggers the number of queries it submits to a name server, to prevent overloading the network and the name server.

**Wait** Identifies queries for which the SMTP server is waiting a response from a name server. Queries remain in this queue for a specific amount of time, within which a reply should be received from the name server.

If a query is successful, that query is then placed in the *Resolver Completed* queue.

If a reply is not received for a query within the allotted time (that is, the resolver time-out has expired), that query is removed from this queue and placed in the *Resolver Retry* queue.

**Note:** The duration of the resolver time-out period can be controlled using the TCPIP DATA file RESOLVERTIMEOUT configuration statement. See the [TCP/IP Planning and Customization](https://www.ibm.com/support/docview.wss?uid=swg27046602) for more information about this statement.

**Retry** Identifies queries that have previously failed, possibly because:

- a name server response was not received for a query (within the designated time-out period), or
- the name server returned a temporary error that has forced the SMTP server to retry a query. A temporary error occurs if, for example, the name server truncates a packet, or if the name server detects a processing error.

**Note:** Mail for which queries are present in this queue can be significantly affected by the values defined for the RESOLVERRETRYINT and RETRYAGE configuration statements in the SMTP CONFIG file. See the [TCP/IP Planning and Customization](https://www.ibm.com/support/docview.wss?uid=swg27046602) for more information about these statements.

**Completed** Identifies queries that have been resolved and are waiting to be recorded by SMTP (and, possibly incorporated within a piece of mail). After the resolved information has been recorded, SMTP attempts to deliver the mail.

**Error** Identifies queries for which a name server response was obtained, but for which no answer was obtained. Mail that corresponds to such queries is returned to the originator as undeliverable, with an unknown recipient error indicated.

**VRFY**

The VRFY (“verify”) command determines if a given mailbox or user ID exists on the host where SMTP is running.

```
VRFY verify_string
```

Operands Description
The z/VM implementation of SMTP responds to the VRFY command and the EXPN command (see the EXPN command below) in the same manner. Thus, the VRFY command can be used with z/VM systems to expand a mailing list defined on such system; when this is done, a multiple-line reply may be returned in response to the VRFY command.

The VRFY command can also be used to verify the existence of the POSTMASTER mailbox or mailboxes defined for a system.

On z/VM systems, mailing lists are defined by the site administrator and are stored in the SMTP NAMES file; POSTMASTER mailboxes are defined by the POSTMASTER configuration statement in the SMTP CONFIG file. See the TCP/IP Planning and Customization for more information about defining mailing lists and specifying POSTMASTER mailboxes.

Some example VRFY commands (issued against an SMTP server running on host TESTVM1 at “somewhere.com”) and their corresponding responses follow:

```
vrfy tcpmaint  250 <tcpmaint@abcvm1.somewhere.com>
vrfy tcpadmin-list 250-<tcpmaint@abcvm1.somewhere.com>
                250-<tcpadmin@abcvm1.somewhere.com>
                250-<tcpadmin@adminpc.somewhere.com>
                250 <maint@abcvm1.somewhere.com>
vrfy postmaster 250-<TCPMAINT@TESTVM1.somewhere.com>
                250-<TCPADMIN@TESTVM1.SOMEWHERE.COM>
                250 <TCPADMIN@ADMINPC.SOMEWHERE.COM>
```

The hyphen (-), when present as the fourth character of a response, indicates the response is continued on the next line.

**EXPN**

The EXPN (“expand”) command expands a mailing list defined on the host where SMTP is running.

```
>EXPNexpand_string
```

**Operand**

<table>
<thead>
<tr>
<th>expand_string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Specifies the name of the mailing list to be expanded.</td>
</tr>
</tbody>
</table>

The EXPN command operates and can be used in the same way as the VRFY command.

**VERB**

The VERB command is used to enable or disable “verbose” mode for batch SMTP connections.
Operand | Description
---|---
ON | Specifies that verbose mode is to be enabled (turned on). When verbose mode is enabled for a batch SMTP connection, SMTP commands and their associated replies are recorded in a batch SMTP response file; this file is sent back to the origination point of the batch SMTP command file when the batch transaction is complete.

OFF | Specifies that verbose mode is to be disabled (turned off); this is the default. When verbose mode is disabled for a batch SMTP connection, only SMTP replies are recorded in the batch SMTP response file; this file is not returned to the origination point of the batch SMTP command file.

See [SMTP Command Responses](#) for more information about the batch SMTP response file, how this file is handled, and how origination points are determined.

**Note:** The VERB command has no effect when issued over a TCP connection.

**TICK**

The TICK command can be used (in conjunction with the VERB ON command) to cause an identifier string to be inserted into a batch SMTP response file.

---

Operand | Description
---|---
identifier | Specifies an identification string to be included in a batch SMTP response file.

This command can be useful for some mail systems that keep track of batch SMTP response files and their content.

**Note:** The TICK command has no effect when it is issued over a TCP connection.

**SMTP Command Example**

The following is an example of an SMTP envelope and its contained piece of mail. The SMTP commands that comprise the envelope are in upper case boldface text. The information after the DATA command, and before the single ASCII period (the “end of data” indicator) is the message header and body. The body is distinguished from the header by the blank line that follows the “Subject: Update” line of text.

```
HELO yourhost.yourdomain.edu
MAIL FROM: <carol@yourhost.yourdomain.edu>
RCPT TO: <msgs@host1.somewhere.com>
RCPT TO: <alice@host2.somewhere.com>
DATA
Date: Sun, 30 Nov 98 nn:nn:nn EST
From: Carol <carol@yourhost.yourdomain.edu>
To: <msgs@host1.somewhere.com>

```
SMTP Command Responses

The z/VM SMTP server can accept SMTP commands that arrive over a TCP connection or over a batch SMTP (BSMTP) connection. With either type of connection, a response (or, reply) is generated for each command received by SMTP. Each reply is prefixed with a three-digit number, or code. The nature of each response can be determined by inspecting the first digit of this reply code; possible values for this digit are:

<table>
<thead>
<tr>
<th>Digit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Echo reply; used only in batch SMTP response files. Received commands are “echoed” in these files to provide contextual information for other reply codes.</td>
</tr>
<tr>
<td>1</td>
<td>Positive Preliminary reply. SMTP does not use a 1 as the first digit of a reply code, because there are no SMTP commands for which such a reply is applicable.</td>
</tr>
<tr>
<td>2</td>
<td>Positive Completion reply; command accepted.</td>
</tr>
<tr>
<td>3</td>
<td>Positive Intermediate reply; data associated with the command should now be provided.</td>
</tr>
<tr>
<td>4</td>
<td>Temporary Negative Completion reply; try the command again, but at a later time.</td>
</tr>
<tr>
<td>5</td>
<td>Permanent Negative Completion reply; the command has been rejected.</td>
</tr>
</tbody>
</table>

For SMTP commands that arrive over a TCP connection, all responses (positive or negative) are returned over that TCP connection.

Similarly, for SMTP commands that arrive over a batch SMTP connection, all responses are written to a batch SMTP response file. If verbose mode is enabled for a batch SMTP connection (through use of the VERB ON command), SMTP returns this response file to the origination point of the spool file. The origination point is determined either from the ORIGINID field of the spool file (if the spool file was generated on the same z/VM system as the SMTP virtual machine) or from the spool file TAG field (if the spool file arrived from a remote system through the RSCS network). If the batch SMTP connection is not in verbose mode, the batch SMTP response file is not returned to the point of origin.

If an error occurs during the processing of commands over a batch SMTP connection, such as reception of a negative response (with a first digit of 4 or 5), an error report is mailed back to the sender of the mail. The sender is determined from the last valid MAIL FROM: command that was received by SMTP. If the sender cannot be determined from a MAIL FROM: command, the sender is assumed to be the origination point of the batch SMTP command file. The error report mailed to the sender includes the batch SMTP response file and the text of the undeliverable mail.
Note: All SMTP commands and data that arrive over TCP or batch SMTP connections are subject to the restrictions imposed by both SMTP conventions and constants defined in either the SMTPGLOB COPY file, or within other SMTP source files. Any changes made to these files to overcome a restriction will require the affected source files to be recompiled, and the SMTP module to be rebuilt. Several significant restrictions, the relevant constants, and their default values are:

- Command lines must not exceed MaxCommandLine (552 characters).
- Data lines longer than MaxDataLine (32767 characters) are wrapped.
- Path addresses must not exceed MaxPathLength (256 characters).
- Domain names must not exceed MaxDomainName (256 characters).
- User names, the local part of a mailbox specification, must not exceed MaxUserName (256 characters).

Path Address Modifications

When SMTP processes MAIL FROM: and RCPT TO: commands, the path addresses specified with these commands may be modified by SMTP due to use of the SOURCEROUTES configuration statement, or based on the content of the path addresses themselves. See the [TCP/IP Planning and Customization] for more information about the SOURCEROUTES statement and its affect on path addresses. For content-based changes, certain path addresses will be rewritten by SMTP as follows:

1. If the local part of a mailbox name includes a percent sign (%) and the domain of the mailbox is that of the host system where SMTP is running, the given domain name is eliminated, and the portion of the “local part” to the right of the percent sign (%) is used as the destination domain. For example, the path address:

   john%yourvm@ourvm.our.edu

is rewritten by SMTP running at “ourvm.our.edu” as:

   john@yourvm

2. Path addresses with source routes are accepted and rewritten to remove the domain name of the host system where SMTP is running. For example, the path address:

   @ourvm.our.edu,@next.host.edu:john@yourvm

is rewritten by SMTP running at “ourvm.our.edu” as:

   @next.host.edu:john@yourvm

Definitions for “valid path format” specifications can be obtained from the RFCs that define the naming conventions used throughout the Internet. For detailed information, consult the RFCs listed in the section [SMTP Commands] on page 311.

Batch SMTP Command Files

Batch SMTP command files are files that contain an SMTP envelope (as described in [SMTP Transactions] on page 309) which are sent to the virtual reader of the SMTP virtual machine using the CMS PUNCH, DISK DUMP, SENDFILE, or NETDATA SEND commands. For a description of these commands, see the z/VM: CMS Command Reference. These files are encoded using EBCDIC.
SMTP Virtual Machine Interfaces

Batch SMTP command files can be sent by users on the same z/VM system, or any system connected through an RSCS network. For information about RSCS networks, see the RSCS General Information.

Batch SMTP files may be modified when they are processed by the SMTP server, as follows:

- All trailing blanks are removed from each record of a file sent in PUNCH format. Trailing blanks are preserved for files sent in NETDATA or DISK DUMP format.
- A record that is entirely blank will be treated as a record with a single blank.

Batch SMTP Examples

The following sections contain examples that demonstrate batch SMTP capabilities.

Sending Mail to a TCP Network Recipient

The example that follows shows the content of a batch SMTP file used to send mail from a CMS user (CAROL at YOURHOST) to two TCP network recipients. The VERB ON command will cause a batch SMTP response file to be returned to the CMS user CAROL. The text included with the TICK command will appear in this file as well, so that the nature of the response file will be evident when it is returned.

```
VERB ON
TICK Carol's Batch Test File
HELO yourhost.yourdomain.edu
MAIL FROM: <carol@yourhost.yourdomain.edu>
RCPT TO: <msgs@host1.somewhere.com>
RCPT TO: <alice@host2.somewhere.com>
DATA
Date: Sun, 30 Nov 98 nn:nn:nn EST
From: Carol <carol@yourhost.yourdomain.edu>
To: <msgs@host1.somewhere.com>
Cc: <alice@host2.somewhere.com>
Subject: Update
Mike: Cindy stubbed her toe. Bobby went to
baseball camp. Marsha made the cheerleading team.
Jan got glasses. Peter has an identity crisis.
Greg made dates with 3 girls and couldn't
remember their names.
QUIT
```

With the exception of the VERB and TICK commands, this sample batch SMTP file contains commands that are identical to those shown in "SMTP Command Example" on page 319.

Following is the batch SMTP response file (BSMTP REPLY) produced for the previous command file:

```
220-YOURHOST.YOURDOMAIN.EDU running IBM VM SMTP Level 320 on Sun, 30 Nov 1998 nn
220 :nn:nn EST
050 VERB ON
250 Verbose Mode On
050 TICK Carol's Batch Test File
250 OK
050 HELO yourhost.yourdomain.edu
250 YOURHOST.YOURDOMAIN.EDU is my domain name. Yours too, I see!
050 MAIL FROM: <carol@yourhost.yourdomain.edu>
250 OK
050 RCPT TO: <msgs@host1.somewhere.com>
250 OK
```
Querying SMTP Delivery Queues

The SMTP delivery queues can be queried by sending a file that contains VERB ON and QUEU commands to the SMTP virtual machine. A batch SMTP response file that contains the QUEU command results is then returned to the originating user ID.

The SMTPQUEU EXEC (supplied with TCP/IP for z/VM on the TCPMAINT 592 “Client-code” minidisk) generates such a file and sends it to the SMTP virtual machine.

Sample content for a BSMPTRP REPLY file returned in response to an SMTPQUEU command follows:

```
220-YOURHOST.YOURDOMAIN.EDU running IBM VM SMTP Level 320 on Sun, 30 Nov 1998 nn
220 :nn:nn EST
050-VERB ON
050
250 Verbos Mode On
050-QUEU
050
250-Queues on YOURHOST.YOURDOMAIN.EDU at nn:nn:nn EST on 11/30/98
250-Spool Queue: Empty
250-Queue for Site: 123.45.67.89 RETRY QUEUE Last Tried: nn:nn:nn
250-Note 00000005 to <MSGS@HOST1.SOMEWHERE.COM>
250-Queue for Site: 98.76.54.32 RETRY QUEUE Last Tried: nn:nn:nn
250-Placeholder...no files queued for this site
250-Undeliverable Queue: Empty
250-Resolution Queues:
250-Resolver Process Queue: Empty
250-Resolver Send Queue: Empty
250-Resolver Wait Queue:
250-00000013 <userx@somehost.nowhereville.com>
250-Resolver Retry Queue: Empty
250-Resolver Completed Queue: Empty
250-Resolver Error Pending Queue: Empty
250 OK
```

SMTP Exit Routines

The SMTP user exits described in the next sections allow you greater control over each piece of mail that is processed by the SMTP server. To effectively use these exits and their parameters, it is necessary to understand SMTP transactions. Refer to the previous sections in this chapter for information about the commands, messages, and replies that are used to facilitate e-mail transactions between the sender and receiver of a piece of mail.

Prior to customizing the server exits described in this section, ensure that you have reviewed the exit limitations and customization recommendations presented in the "Customizing Server-specific Exits" section of the TCP/IP Planning and Customization.
Client Verification Exit

When a client connects to SMTP, the originating mail domain must be provided. The client verification exit can be used to verify that the domain name provided by a client matches that client’s IP address. Thus, this exit allows flexibility on actions you can take to deal with spoofing problems. In spoofing, the client provides a falsified domain in order to cause mail to appear to have come from someone (or somewhere) else. When client verification is performed, you might choose to include the verification results in mail headers, or possibly reject future communications on a connection.

With the client verification exit, you can perform any or all of the following:

- Reject mail from a particular host.
- Mark certain trusted sites as verified, but perform verification on all others.
- Control which users can use a particular SMTP server.

The exit can be further customized to perform additional actions that are unique or required for your environment.

**Note:** The client verification exit is called for each HELO or EHLO command processed for each mail item received from the network. Client verification is not performed for mail items received from the SMTP virtual reader.

**Built-in Client Verification Function**

In addition to the exit, SMTP can be configured to perform client verification through internal processing. When this support is enabled, this “built-in” client verification function will be called for each HELO or EHLO processed for each mail item. See the [TCP/IP Planning and Customization](#) for detailed information about configuring SMTP to use this support.

The built-in client verification function of the SMTP server can be used to determine if a client host name and client IP address match, and to include the result of that determination in the mail headers. This function will perform a DNS lookup against the HELO or EHLO command data provided by a client, and will then insert a message into the mail header that reflects the result of this lookup.

Client verification performed using the built-in function has three possible outcomes:

**Success**
The data the client provided in the HELO or EHLO command corresponds to the client address. The following line is inserted into the mail header:

```
X-Comment: localhost: Mail was sent by host
```

**Failure**
The data the client provided in the HELO or EHLO command is not associated with the client IP address. In this case, a reverse name lookup is done against the client IP address to determine the actual host name. The following line is inserted into the mail header:

```
X-Comment: localhost: Host host claimed to be helodata
```

**Unknown**
The validation could not be performed. This situation could occur if the name server is not responding, or the verification could not be performed in the allotted time (as controlled by the VERIFYCLIENTDELAY statement). The following line is inserted into the mail header:

```
X-Comment: localhost: Unknown client verification
```
The terms used in the previously listed mail header messages are described in more detail here:

- **localhost**: the local VM host name
- **helodata**: the data the client provided with the HELO or EHLO command
- **host**: the host name determined by the reverse DNS lookup; if a host name is not found, "unknown host" will be used
- **ipaddr**: the client IP address.

### Client Verification Exit Parameter Lists

The parameter lists passed to the REXX and the assembler exit routines follow. When you customize either of these exits, keep in mind the following:

- Because an identical exit parameter list definition is used for all of the SMTP user exits, not all parameters may be meaningful for this exit. Parameters that are not used by this exit are indicated in the exit parameter lists; their values should be ignored.
- For the REXX exit, the value of an unused parameter will be such that any parsing will not be affected.

Parameter descriptions that pertain to both the REXX and assembler exits are provided on page 326.

### REXX Parameter List

#### Inputs:

**Table 73. Client Verification REXX Exit Parameter List**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| ARG(1)   | Parameter list defined as follows:  
|          | • Exit type  
|          | • Version number  
|          | • Mail record ID  
|          | • Port number of SMTP server  
|          | • IP address of SMTP server  
|          | • Port number of client  
|          | • IP address of client  
|          | • Filename of note on disk  
|          | • Verify Client status  
|          | • Maximum length of Return String  
| ARG(2)   | SMTP command string |
| ARG(3)   | HELO/EHLO name |
| ARG(4-6) | Not used |

#### Outputs:

The following are returned to the caller in the RESULT variable via a REXX RETURN statement:
SMTP Virtual Machine Interfaces

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC</td>
<td>The exit return code; this must be a 4-byte binary value.</td>
</tr>
<tr>
<td>Return String</td>
<td>An exit-specified string; the returned value must have a length less than or equal to the maximum length passed to the exit.</td>
</tr>
</tbody>
</table>

**Assembler Parameter List**
Following is the parameter list that is passed to the assembler exit routine. General Register 1 points to the parameter list.

*Table 74. Client Verification ASSEMBLER Exit Parameter List*

<table>
<thead>
<tr>
<th>Offset in Decimal</th>
<th>Len</th>
<th>In/Out</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>4</td>
<td>Input</td>
<td>Char</td>
<td>Exit type</td>
</tr>
<tr>
<td>+4</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Version number</td>
</tr>
<tr>
<td>+8</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Mail record ID</td>
</tr>
<tr>
<td>+12</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Port number of SMTP server</td>
</tr>
<tr>
<td>+16</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>IP address of SMTP server</td>
</tr>
<tr>
<td>+20</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Port number of client</td>
</tr>
<tr>
<td>+24</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>IP address of client</td>
</tr>
<tr>
<td>+28</td>
<td>4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of SMTP command string</td>
</tr>
<tr>
<td>+32</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Length of SMTP command string</td>
</tr>
<tr>
<td>+36</td>
<td>4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of HELO/EHLO name</td>
</tr>
<tr>
<td>+40</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Length of HELO/EHLO name</td>
</tr>
<tr>
<td>+44</td>
<td>24</td>
<td></td>
<td></td>
<td>Not used</td>
</tr>
<tr>
<td>+68</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Verify Client status</td>
</tr>
<tr>
<td>+72</td>
<td>4</td>
<td>Output</td>
<td>Ptr</td>
<td>Address of Return String</td>
</tr>
<tr>
<td>+76</td>
<td>4</td>
<td>Output</td>
<td>Int</td>
<td>Length of Return String</td>
</tr>
<tr>
<td>+80</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Maximum length of Return String</td>
</tr>
<tr>
<td>+84</td>
<td>8</td>
<td></td>
<td></td>
<td>Not used</td>
</tr>
<tr>
<td>+92</td>
<td>4</td>
<td>Input/ Output</td>
<td>Char</td>
<td>User Word 1</td>
</tr>
<tr>
<td>+96</td>
<td>4</td>
<td>Input/ Output</td>
<td>Char</td>
<td>User Word 2</td>
</tr>
<tr>
<td>+100</td>
<td>4</td>
<td>Output</td>
<td>Int</td>
<td>Return code from exit</td>
</tr>
</tbody>
</table>

**Parameter Descriptions**

**Exit type**
A four-character field that indicates the type of exit called. For the client verification exit, this is VERX.

**Version number**
The parameter list version number; if the parameter list format is changed, the version number will change. Your exit should verify it has received the expected version number. The current version number is 1.
Mail record ID
A number that uniquely identifies a piece of mail so that multiple exit calls can be correlated to the same piece of mail. A value of 0 indicates a mail record ID is not available.

Port number of SMTP server
The port number used by the SMTP server for this connection.

IP address of SMTP server
For the REXX exit, a dotted-decimal format IP address is provided; for the assembler exit, this is an IP address in decimal integer form. For multi-homed hosts, this address can be compared with the client IP address to determine in which part of the network the client host resides.

Port number of client
If the connection no longer exists, -1 is supplied. Otherwise, this is the port number used by the foreign host for this connection.

IP address of client
For the REXX exit, a dotted-decimal format IP address is provided; for the assembler exit, this is an IP address in decimal integer form.

SMTP command string
Contains the HELO/EHLO command and the domain specified for this command. The string has been converted to uppercase (for example, “HELO DOMAIN1”).

HELO/EHLO name
A string that contains the name specified on the HELO or EHLO command; this string may be either:
• a domain name
• an IP address in decimal integer form that is prefixed by the number or (US) pound sign (“#” or X’7B’)
• an IP address in dotted-decimal form, enclosed in brackets.

For example, if the command HELO #123456 is provided by an SMTP client, this parameter would contain #123456.

The name has already been verified to have the correct syntax.

Verify Client status
A number that indicates client verification results. For this exit, client verification results are unknown when the exit receives control; thus, this field will contain a 3. Possible values and their meanings are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Client verification passed.</td>
</tr>
<tr>
<td>1</td>
<td>Client verification failed.</td>
</tr>
<tr>
<td>2</td>
<td>Client verification was not performed.</td>
</tr>
<tr>
<td>3</td>
<td>Client verification results are unknown.</td>
</tr>
</tbody>
</table>

Return String
When the exit returns a return code of 3, this value is appended to the X-Comment that is inserted in the mail header. When the exit returns a return code of 5, the Return String value is appended to the 550 reply code.

Maximum length of Return String
The current maximum is 512 bytes; ensure the Return String length is less than this value. If the returned string is longer than the indicated maximum, the return string is truncated and the following message is displayed on the SMTP sever console:

Return data from exit exitname exittype too long, data truncated
Normal processing continues.

**User Word 1**
Provided for use by the assembler exit only. The user word specified upon return from this exit will be passed back in this field for any future calls; 0 is the initial value. The SMTP server does not use this value in any way.

**User Word 2**
Provided for use by the assembler exit only. The user word specified upon return from this exit will be passed back in this field for any future calls; 0 is the initial value. The SMTP server does not use this value in any way.

**Return Codes from the Client Verification Exit Routine**
Following are the return codes recognized by SMTP for this exit.

*Table 75. Client Verification Exit Return Codes*

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Do not verify client. A comment will not be inserted in the mail header.</td>
</tr>
<tr>
<td>1</td>
<td>Perform verification using the built-in client verification function.</td>
</tr>
<tr>
<td>2</td>
<td>Mark as verified. The following comment will be inserted in the mail header:</td>
</tr>
<tr>
<td></td>
<td>X-Comment: localhost: Mail was sent by host</td>
</tr>
<tr>
<td>3</td>
<td>The following comment will be inserted in the mail header:</td>
</tr>
<tr>
<td></td>
<td>X-Comment: Return String</td>
</tr>
<tr>
<td></td>
<td>where the value for Return String can be specified by the exit.</td>
</tr>
<tr>
<td>4</td>
<td>Disable the exit. The following message will be displayed on the SMTP console:</td>
</tr>
<tr>
<td></td>
<td>VERIFYCLIENT EXIT function disabled</td>
</tr>
<tr>
<td></td>
<td>The exit will no longer be called. The client will not be verified and no comment will be inserted in the mail header.</td>
</tr>
<tr>
<td>5</td>
<td>Reject this command with: 550 Return String</td>
</tr>
<tr>
<td></td>
<td>If a return string is not provided by the exit, then the default message will be displayed:</td>
</tr>
<tr>
<td></td>
<td>550 Access denied</td>
</tr>
<tr>
<td></td>
<td>All future communications on this connection will be rejected with this 550 message.</td>
</tr>
<tr>
<td>Other</td>
<td>Any return code other than the above causes SMTP to issue this message:</td>
</tr>
<tr>
<td></td>
<td>Unexpected return from user exit exitname exittype, RC = rc</td>
</tr>
<tr>
<td></td>
<td>SMTP treats this return code as if it were a return code of 0.</td>
</tr>
</tbody>
</table>

**Client Verification Sample Exits**
Sample Client Verification exit routines are supplied with TCP/IP on the TCPMAINT 591 disk. The supplied samples are:

**SMTPVERX SSEXEC**
REXX exit routine that contains a sample framework for performing client verification actions. Your customized exit should be stored on the TCPMAINT 198 disk as SMTPVERX EXEC.

**SMTPVERX SAMPASM**
The assembler exit routine called by the SMTP server; the exit is used to
call SMTPVERX EXEC and to pass results back to the SMTP server. Your customized exit should be stored on the TCPMAINT 198 disk as file SMTPVERX ASSEMBLE. The customized ASSEMBLE file must be assembled (by using the VMFHLASM SMTPVERX DMSVM command), and the resulting text deck placed on the TCPMAINT 198 disk.

These samples are for illustrative purposes only. They should be modified to meet the needs of your installation before placing them in a production environment. The assembler exit will have better performance characteristics than the REXX exit. For best performance, EXECLOAD any REXX exits.

## Using the Mail Forwarding Exit

When SMTP clients use the VM SMTP server to send mail to hosts that their workstations cannot reach directly, this is an instance of mail forwarding. The mail forwarding exit provides a mechanism to control this activity. When SMTP determines the addressee specified on a RCPT TO: command is not “defined on” the local system, it has detected mail forwarding, and it will call this exit routine.

The phrase “defined on” in the previous paragraph is meant to convey that SMTP considers a user to be a local user, in addition to any other criteria, if that user is defined in the SMTP NAMES file — regardless of whether mail delivery to that user is performed via spooling (RSCS services) or through a network TCP connection. Also, keep in mind that the determination of whether mail forwarding is occurring is made on a recipient-by-recipient basis, not on other aspects of a given piece of mail. A piece of mail with multiple recipients can contain occurrences of both mail forwarding and local delivery.

With the mail forwarding exit, you can perform any or all of the following:

- Allow mail forwarding and mail delivery to proceed without interruption.
- Disallow mail forwarding from a known sender of “junk” mail, and possibly reject future communications on a connection used for this purpose.
- Intercept mail from specific clients and forward that mail to a local VM user ID for further analysis.
- Restrict the ability to forward mail to a particular set of hosts.

The exit can be further customized to perform additional actions that are required for your environment.

**Note:** The mail forwarding exit is only called for mail items received from the network; it is not called for mail items generated on the VM system or received via RSCS.

This exit can also be used to control spamming. Spamming is the act of sending mail to a large number of e-mail addressees and is often compared to the term “junk mail”, used to describe similar activities performed via postal services. Spam is a piece of mail that is perceived by the recipients to be unsolicited and unwanted. There are two aspects to consider when trying to control spamming problems:

- Is your system being used to relay spam messages to recipients throughout the internet?
- Are incoming spam messages to your local users seriously taxing or overloading your system?
The relaying of spam messages may be treated like any other type of mail forwarding. The exit can prevent delivery of all forwarded mail, prevent delivery of mail from particular sites known for spamming, or only allow delivery of mail from particular trusted sites. Handling spam messages directed to your local users will require the use of the SMTP command exit. When you address spamming problems, it’s important to realize that one person may consider a piece of mail to be a spam, while the same piece of mail may be valuable to someone else. There are no explicit rules that determine what is and is not spam.

In addition to the exit, SMTP can be configured to enable or disable mail forwarding for all mail. If mail forwarding is disabled in this manner and SMTP determines the recipient specified on a RCPT TO: record is not defined on the local system, it has detected mail forwarding, and it will reject the delivery of the mail to that recipient. See the TCP/IP Planning and Customization for more information about configuring SMTP to accept or reject all forwarded mail.

Mail Forwarding Exit Parameter Lists

The parameter lists passed to the REXX and the assembler exit routines follow. When you customize either of these exits, keep in mind the following:

- Because an identical exit parameter list definition is used for all of the SMTP user exits, not all parameters may be meaningful for this exit. Parameters that are not used by this exit are indicated in the exit parameter lists; their values should be ignored.
- For the REXX exit, the value of an unused parameter will be such that any parsing will not be affected.

Parameter descriptions that pertain to both the REXX and assembler exits are provided on page 331.

REXX Parameter List

Inputs:

Table 76. Mail Forwarding REXX Exit Parameter List

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARG(1)</td>
<td>Parameter list defined as follows:</td>
</tr>
<tr>
<td></td>
<td>• Exit type</td>
</tr>
<tr>
<td></td>
<td>• Version number</td>
</tr>
<tr>
<td></td>
<td>• Mail record ID</td>
</tr>
<tr>
<td></td>
<td>• Port number of SMTP server</td>
</tr>
<tr>
<td></td>
<td>• IP address of SMTP server</td>
</tr>
<tr>
<td></td>
<td>• Port number of client</td>
</tr>
<tr>
<td></td>
<td>• IP address of client</td>
</tr>
<tr>
<td></td>
<td>• Filename of note on disk</td>
</tr>
<tr>
<td></td>
<td>• Verify Client status</td>
</tr>
<tr>
<td></td>
<td>• Maximum length of Return String</td>
</tr>
<tr>
<td>ARG(2)</td>
<td>SMTP command string</td>
</tr>
<tr>
<td>ARG(3)</td>
<td>HELO/EHLO name</td>
</tr>
<tr>
<td>ARG(4)</td>
<td>MAIL FROM: string</td>
</tr>
<tr>
<td>ARG(5)</td>
<td>Client domain name</td>
</tr>
<tr>
<td>ARG(6)</td>
<td>Not used</td>
</tr>
</tbody>
</table>

Outputs:
The following are returned to the caller in the RESULT variable via a REXX RETURN statement:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC</td>
<td>The exit return code; this must be a 4-byte binary value.</td>
</tr>
<tr>
<td>Return String</td>
<td>An exit-specified string; the returned value must have a length less than or equal to the maximum length passed to the exit.</td>
</tr>
</tbody>
</table>

**Assembler Parameter List**
Following is the parameter list that is passed to the assembler exit routine. General Register 1 points to the parameter list.

<table>
<thead>
<tr>
<th>Offset in Decimal</th>
<th>Len</th>
<th>In/Out</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>4</td>
<td>Input</td>
<td>Char</td>
<td>Exit type</td>
</tr>
<tr>
<td>+4</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Version number</td>
</tr>
<tr>
<td>+8</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Mail record ID</td>
</tr>
<tr>
<td>+12</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Port number of SMTP server</td>
</tr>
<tr>
<td>+16</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>IP address of SMTP server</td>
</tr>
<tr>
<td>+20</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Port number of client</td>
</tr>
<tr>
<td>+24</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>IP address of client</td>
</tr>
<tr>
<td>+28</td>
<td>4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of SMTP command string</td>
</tr>
<tr>
<td>+32</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Length of SMTP command string</td>
</tr>
<tr>
<td>+36</td>
<td>4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of HELO/EHLO name</td>
</tr>
<tr>
<td>+40</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Length of HELO/EHLO name</td>
</tr>
<tr>
<td>+44</td>
<td>4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of client domain name</td>
</tr>
<tr>
<td>+48</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Length of client domain name</td>
</tr>
<tr>
<td>+52</td>
<td>4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of MAIL FROM: string</td>
</tr>
<tr>
<td>+56</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Length of MAIL FROM: string</td>
</tr>
<tr>
<td>+60</td>
<td>8</td>
<td>Input</td>
<td>Char</td>
<td>File name of note on disk</td>
</tr>
<tr>
<td>+68</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Verify Client status</td>
</tr>
<tr>
<td>+72</td>
<td>4</td>
<td>Output</td>
<td>Ptr</td>
<td>Address of Return String</td>
</tr>
<tr>
<td>+76</td>
<td>4</td>
<td>Output</td>
<td>Int</td>
<td>Length of Return String</td>
</tr>
<tr>
<td>+80</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Maximum length of Return String</td>
</tr>
<tr>
<td>+84</td>
<td>8</td>
<td></td>
<td></td>
<td>Not used</td>
</tr>
<tr>
<td>+92</td>
<td>4</td>
<td>Input/ Output</td>
<td>Char</td>
<td>User Word 1</td>
</tr>
<tr>
<td>+96</td>
<td>4</td>
<td>Input/ Output</td>
<td>Char</td>
<td>User Word 2</td>
</tr>
<tr>
<td>+100</td>
<td>4</td>
<td>Output</td>
<td>Int</td>
<td>Return code from exit</td>
</tr>
</tbody>
</table>

**Parameter Descriptions:**

**Exit type**
A four-character field that indicates the type of exit called. For the mail forwarding exit, this is FWDX.

**Version number**
The parameter list version number; if the parameter list format is changed, the version number will change. Your exit should verify it has received the expected version number. The current version number is 1.
Mail record ID
A number that uniquely identifies a piece of mail so that multiple exit calls can be correlated to the same piece of mail. A value of 0 indicates a mail record ID is not available.

Port number of SMTP server
The port number used by the SMTP server for this connection.

IP address of SMTP server
For the REXX exit, a dotted-decimal format IP address is provided; for the assembler exit, this is an IP address in decimal integer form. For multi-homed hosts, this address can be compared with the client IP address to determine in which part of the network the client host resides.

Port number of client
If the connection no longer exists, -1 is supplied. Otherwise, this is the port number used by the foreign host for this connection.

IP address of client
For the REXX exit, a dotted-decimal format IP address is provided; for the assembler exit, this is an IP address in decimal integer form.

SMTP command string
Contains the name specified on the RCPT TO: command. The recipient path, enclosed in angle brackets (< and >), is included. The recipient path may be in any valid path format; it has already been verified to have the correct syntax. Because the recipient address has been resolved, this string may not exactly match the data provided with the RCPT TO: command.

For example, if the following has been specified by the SMTP client:

```
RCPT TO: <usera@host1>
```

the SMTP command string might contain: `<usera@host1.com>`

HELO/EHLO name
A string that contains the name specified on the HELO or EHLO command; this string may be either:

- a domain name
- an IP address in decimal integer form that is prefixed by the number or (US) pound sign (“#” or X’7B’)
- an IP address in dotted-decimal form, enclosed in brackets.

For example, if the command `HELO #123456` is provided by an SMTP client, this parameter would contain: `#123456`.

The name has already been verified to have the correct syntax.

Client domain name
The domain name that corresponds to the client IP address. The length of this field will be zero if:

- client verification was not performed
- the results of client verification are unknown
- a reverse lookup failed

In all other cases, this will be a domain name.

MAIL FROM: string
Contains the name specified on the MAIL FROM: command. The sender path, enclosed in angle brackets (< and >), is included. The sender path may be in any valid path format; it has already been verified to have the correct syntax.
Because the sender address has been resolved, this string may not exactly match the data provided with the MAIL FROM: command.

For example, if the following has been specified by the SMTP client:

```
MAIL FROM: <userb@host2>
```

the SMTP command string might contain: `<userb@host2.com>`

**File name of note on disk**

Name of the file created after the “end of data” (EOD) condition, a period (.), is received. Prior to either of these conditions, the file name is not defined; in this case, an asterisk (*) will be supplied.

**Verify Client status**

A number that indicates client verification results. Possible values and their meanings are:

- 0: Client verification passed.
- 1: Client verification failed.
- 2: Client verification was not performed.
- 3: Client verification results are unknown.

**Return String**

When the exit returns a return code of 1 or 5, this value is appended to the 551 or 550 reply code. When the exit returns a return code of 2, the Return String value should contain a VM user ID to which mail should be transferred.

**Maximum length of Return String**

The current maximum is 512 bytes; ensure the Return String length is less than this value. If the returned string is longer than the indicated maximum, the return string is truncated and the following message is displayed on the SMTP sever console:

```
Return data from exit exitname exittype too long, data truncated
```

Normal processing continues.

**User Word 1**

Provided for use by the assembler exit only. The user word specified upon return from this exit will be passed back in this field for any future calls; 0 is the initial value. The SMTP server does not use this value in any way.

**User Word 2**

Provided for use by the assembler exit only. The user word specified upon return from this exit will be passed back in this field for any future calls; 0 is the initial value. The SMTP server does not use this value in any way.

**Return Codes from the Mail Forwarding Exit Routine**

Following are the return codes recognized by SMTP for this exit.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Accept and attempt mail delivery.</td>
</tr>
<tr>
<td>1</td>
<td>Reject mail with: 551 <em>Return String</em></td>
</tr>
</tbody>
</table>

If a return string is not provided by the exit, the following default message will be used:

```
551 User not local; please try user@otherhost
```

If the server has already responded to the command, this return code will result in error mail being sent back to the sender.
Table 78. Mail Forwarding Exit Return Codes (continued)

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Accept and forward to the local VM user ID specified by Return String. If the VM user ID is null or is not valid, the mail will be delivered to the local postmaster; the mail will not be delivered to the addressee.</td>
</tr>
<tr>
<td>4</td>
<td>Disable the exit. The following message will be displayed on the SMTP console: FORWARD MAIL EXIT function disabled. The exit will no longer be called. SMTP will attempt to deliver this mail.</td>
</tr>
<tr>
<td>5</td>
<td>Reject this command with: 550 Return String. If a return string is not provided by the exit, then the default message will be displayed: 550 Access denied. All future communications on this connection will be rejected with this 550 message.</td>
</tr>
<tr>
<td>Other</td>
<td>Any return code other than the above causes SMTP to issue this message: Unexpected return from user exit exitname exittype, RC = rc. SMTP treats this return code as if it were a return code of 0.</td>
</tr>
</tbody>
</table>

Mail Forwarding Sample Exits
Sample Mail Forwarding exit routines are supplied with TCP/IP on the TCPMAINT 591 disk. The supplied samples are:

**SMTPFWDX SEXEC**
REXX exit routine that contains a sample framework for handling forwarded mail items. Your customized exit should be stored on the TCPMAINT 198 disk as file SMTPFWDX EXEC.

**SMTPFWDX SAMPASM**
The assembler exit routine called by the SMTP server; the exit is used to call SMTPFWDX EXEC and to pass results back to the SMTP server. Your customized exit should be stored on the TCPMAINT 198 disk as file SMTPFWDX ASSEMBLE. The customized ASSEMBLE file must be assembled (by using the VMFHLASM SMTPFWDX DMSVM command), and the resulting text deck placed on the TCPMAINT 198 disk.

These samples are for illustrative purposes only. They should be modified to meet the needs of your installation before placing them in a production environment. The assembler exit will have better performance characteristics than the REXX exit. For best performance, EXECLOAD any REXX exits.

Using the SMTP Command Exit
The SMTP server can be configured to call an exit routine whenever certain SMTP commands are received, through use of the SMTP command exit. This exit can be defined such that it is invoked for any or all the commands that follow:

**HELO**
The SMTP 'HELO' command.

**EHLO**
The SMTP 'EHLO' command.
MAIL
The SMTP 'MAIL FROM:' command.

RCPT
The SMTP 'RCPT TO:' command.

DATA
The SMTP 'DATA' command.

EOD
The “end of data” condition. This occurs when a period (.) is received by the server, usually after all data has been transmitted.

VRFY
The SMTP 'VRFY' command.

EXPN
The SMTP 'EXPN' command.

RSET
The SMTP 'RSET' command.

PUNCH
The point in time when the server is about to deliver mail to a local destination on the same node or RSCS network; this command is unique to the VM TCP/IP SMTP server.

Notes:
1. The person responsible for creating or maintaining programs that exploit this capability should be knowledgeable of the protocol(s) related to the SMTP commands that are processed using this exit.
2. Only one SMTP command exit can be active at a time.

The SMTP command exit could be used for a wide variety of purposes; several possible uses are included here:
• Reject particular SMTP commands. For example, you may not want your server to support the VRFY and EXPN commands.
• Handle the delivery of local mail in a specific manner.
• Screen and reject mail that contains offensive language, or fails to meet other criteria defined by your installation.

Note: Scanning the content of a message will severely degrade server performance.

SMTP Command Exit Parameter Lists
The parameter lists passed to the REXX and the assembler exit routines follow.
When you customize either of these exits, keep in mind the in mind the following:
• Because an identical exit parameter list definition is used for all of the SMTP user exits, not all parameters may be meaningful for this exit. Parameters that are not used by this exit are indicated in the exit parameter lists; their values should be ignored.
• For the REXX exit, the value of an unused parameter will be such that any parsing will not be affected.

Parameter descriptions that pertain to both the REXX and assembler exits are provided on page 337.
SMTP Virtual Machine Interfaces

REXX Parameter List

Inputs:

Table 79. SMTP Commands REXX Exit Parameter List

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARG(1)</td>
<td>Parameter list defined as follows:</td>
</tr>
<tr>
<td></td>
<td>- Exit type</td>
</tr>
<tr>
<td></td>
<td>- Version number</td>
</tr>
<tr>
<td></td>
<td>- Mail record ID</td>
</tr>
<tr>
<td></td>
<td>- Port number of SMTP server</td>
</tr>
<tr>
<td></td>
<td>- IP address of SMTP server</td>
</tr>
<tr>
<td></td>
<td>- Port number of client</td>
</tr>
<tr>
<td></td>
<td>- IP address of client</td>
</tr>
<tr>
<td></td>
<td>- Filename of note on disk</td>
</tr>
<tr>
<td></td>
<td>- Verify Client status</td>
</tr>
<tr>
<td></td>
<td>- Maximum length of Return String</td>
</tr>
<tr>
<td>ARG(2)</td>
<td>SMTP command string</td>
</tr>
<tr>
<td>ARG(3)</td>
<td>HELO/EHLO name</td>
</tr>
<tr>
<td>ARG(4)</td>
<td>MAIL FROM: string</td>
</tr>
<tr>
<td>ARG(5)</td>
<td>Client domain name</td>
</tr>
<tr>
<td>ARG(6)</td>
<td>Batch VM user ID</td>
</tr>
</tbody>
</table>

Outputs: The following are returned to the caller in the RESULT variable via a REXX RETURN statement:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC</td>
<td>The exit return code; this must be a 4-byte numeric value.</td>
</tr>
<tr>
<td>Return String</td>
<td>An exit-specified string; the returned value must have a length less than or equal to the maximum length passed to the exit.</td>
</tr>
</tbody>
</table>

Assembler Parameter List

Following is the parameter list that is passed to the assembler exit routine. General Register 1 points to the parameter list.

Table 80. SMTP Commands ASSEMBLER Exit Parameter List

<table>
<thead>
<tr>
<th>Offset in Decimal</th>
<th>Len</th>
<th>In/Out</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>4</td>
<td>Input</td>
<td>Char</td>
<td>Exit type</td>
</tr>
<tr>
<td>+4</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Version number</td>
</tr>
<tr>
<td>+8</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Mail record ID</td>
</tr>
<tr>
<td>+12</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Port number of SMTP server</td>
</tr>
<tr>
<td>+16</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>IP address of SMTP server</td>
</tr>
<tr>
<td>+20</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Port number of client</td>
</tr>
<tr>
<td>+24</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>IP address of client</td>
</tr>
<tr>
<td>+28</td>
<td>4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of SMTP command string</td>
</tr>
<tr>
<td>+32</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Length of SMTP command string</td>
</tr>
<tr>
<td>+36</td>
<td>4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of HELO/EHLO name</td>
</tr>
</tbody>
</table>
Parameter Descriptions:

Exit type
A four-character field that indicates the type of exit called. For the SMTP command exit, this is CMDX.

Version number
The parameter list version number; if the parameter list format is changed, the version number will change. Your exit should verify it has received the expected version number. The current version number is 1.

Mail record ID
A number that uniquely identifies a piece of mail so that multiple exit calls can be correlated to the same piece of mail. A value of 0 indicates a mail record ID is not available.

Port number of SMTP server
The port number used by the SMTP server for this connection.

IP address of SMTP server
For the REXX exit, a dotted-decimal format IP address is provided; for the assembler exit, this is an IP address in decimal integer form. For multi-homed hosts, this address can be compared with the client IP address to determine in which part of the network the client host resides.

Port number of client
If the connection no longer exists, or if the command was received over a batch (BSMTP) connection, -1 is supplied. Otherwise, this is the port number used by the foreign host for this connection.

IP address of client
For the REXX exit, a dotted-decimal format IP address is provided; for the assembler exit, this is an IP address in decimal integer form. If the relevant SMTP command was received over a batch SMTP (BSMTP) connection, this field is 0.

File name of note on disk
Name of the file created after the “end of data” (EOD) condition, a period (.),
SMTP Virtual Machine Interfaces

is received. Prior to either of these conditions, the file name is not defined; in this case, an asterisk (*) will be supplied.

Verify Client status
A number that indicates client verification results. Possible values and their meanings are:
0 Client verification passed.
1 Client verification failed.
2 Client verification was not performed.
3 Client verification results are unknown.

SMTP command string
Contains the current command and parameters; the string has been converted to uppercase. For example, if this exit was called for the MAIL FROM: command, this string might contain: MAIL FROM: <USERA@MYDOMAIN>.

HELO/EHLO name
A string that contains the name specified on the HELO or EHLO command; this string may be either:
• a domain name
• an IP address in decimal integer form that is prefixed by the number or (US) pound sign (“#” or X’7B’)
• an IP address in dotted-decimal form, enclosed in brackets.
For example, if the command HELO #123456 is provided by an SMTP client, this parameter would contain: #123456.

The name has already been verified to have the correct syntax.

MAIL FROM: string
Contains the name specified on the MAIL FROM: command. The sender path, enclosed in angle brackets (< and >), is included. The sender path may be in any valid path format; it has already been verified to have the correct syntax. Because the sender address has been resolved, this string may not exactly match the data provided with the MAIL FROM: command.

For example, if the following has been specified by the SMTP client:
MAIL FROM: <userb@host2>

the SMTP command string might contain: <userb@host2.com>

Client domain name
The domain name that corresponds to the client IP address. This field will be a null string if:
• client verification was not performed
• the results of client verification are unknown
• a reverse lookup failed
In all other cases, this will be a domain name.

Batch VM user ID
This field is only used when SMTP commands arrive over a batch SMTP (BSMTP) connection. If this exit is called for batch SMTP connections, this field will contain the VM User ID that originated the mail. Otherwise, this field is not defined and will contain nulls.

User Word 1
Provided for use by the assembler exit only. The user word specified upon return from this exit will be passed back in this field for any future calls; 0 is the initial value. The SMTP server does not use this value in any way.
User Word 2
Provided for use by the assembler exit only. The user word specified upon return from this exit will be passed back in this field for any future calls; 0 is the initial value. The SMTP server does not use this value in any way.

Return String
When the exit returns a return code of 1 or 5, this value is appended to the 550 reply code.

Maximum length of Return String
The current maximum is 512 bytes; ensure the Return String length is less than this value. If the returned string is longer than the indicated maximum, the return string is truncated and the following message is displayed on the SMTP server console:

Return data from exit exitname exittype too long, data truncated

Normal processing continues.

Return Codes from the SMTP Command Exit Routine
Following are the return codes recognized by SMTP for this exit.

Table 81. SMTP Command Exit Return Codes

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Accept command, and continue normal processing.</td>
</tr>
<tr>
<td>1</td>
<td>Reject mail with: 551 Return String</td>
</tr>
<tr>
<td></td>
<td>If a return string is not provided by the exit, the following default message will be used: 550 Command Rejected</td>
</tr>
<tr>
<td></td>
<td>This return code is valid for only the PUNCH command; if 1 is returned for a PUNCH command exit call, it will be handled as an invalid exit return code.</td>
</tr>
<tr>
<td>2</td>
<td>The PUNCH command has been handled by the exit routine; therefore, bypass file delivery. This return code is valid for only the PUNCH command; if 1 is returned for a PUNCH command exit call, it will be handled as an invalid exit return code.</td>
</tr>
<tr>
<td>4</td>
<td>Disable the exit. The following message will be displayed on the SMTP console: SMTPCMDS EXIT function disabled</td>
</tr>
<tr>
<td></td>
<td>The exit will no longer be called. The command will be attempted, and processing will continue.</td>
</tr>
<tr>
<td>5</td>
<td>Reject this command with: 550 Return String</td>
</tr>
<tr>
<td></td>
<td>If a return string is not provided by the exit, then the default message will be displayed: 550 Access denied</td>
</tr>
<tr>
<td></td>
<td>All future communications on this connection will be rejected with this 550 message. This return code is valid for only the PUNCH command; if 1 is returned for a PUNCH command exit call, it will be handled as an invalid exit return code.</td>
</tr>
</tbody>
</table>
Table 81. SMTP Command Exit Return Codes (continued)

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other</td>
<td>Any return code other than the above causes SMTP to issue this message: Unexpected return from user exit exitname exittype, RC = rc</td>
</tr>
</tbody>
</table>

SMTP treats this return code as if it were a return code of 0.

Sample SMTP Command Exits

Sample SMTP Command exit routines are supplied with TCP/IP on the TCPMAINT 591 disk. The supplied samples are:

**SMTPCMDX SEXEC**
REXX exit routine that contains a sample framework for SMTP command processing. Your customized exit should be stored on the TCPMAINT 198 disk as file SMTPCMDX EXEC.

**SMTPCMDX SAMPASM**
The assembler exit routine called by the SMTP server; the exit is used to call SMTPCMDX EXEC and to pass results back to the SMTP server. Your customized exit should be stored on the TCPMAINT 198 disk as file SMTPCMDX ASSEMBLE. The customized ASSEMBLE file must be assembled (by using the VMFHLASM SMTPCMDX DMSVM command), and the resulting text deck placed on the TCPMAINT 198 disk.

These samples are for illustrative purposes only. They should be modified to meet the needs of your installation before placing them in a production environment. The assembler exit will have better performance characteristics than the REXX exit. For best performance, EXECLOAD any REXX exits.
Chapter 10. Telnet Exits

The Telnet server exits described in the sections that follow provide CP command simulation, TN3270E printer management, and system access control when Telnet connections are established with your host.

Prior to customizing the server exits described in this section, ensure that you have reviewed the exit limitations and customization recommendations presented in the "Customizing Server-specific Exits" section of the TCP/IP Planning and Customization.

While the SCEXIT or PMEXIT is running, the TCP/IP service machine cannot service any other requests. Therefore, it is advised that processing performed within these exits should be minimized.

Also, in environments with a high rate of TN3270 and/or TN3270E session creation and termination, the use of a REXX exec could adversely affect performance. While calling such an exec may be useful for designing and testing a prototype, a production-level exit should be written in assembler. For such environments, the supplied sample Telnet session connection exit (SCEXIT SAMPASM) and printer management exit (PMEXIT SAMPASM) should be used as a basis for assemble files which directly perform any actions appropriate for your environment. It is recommended that execs be used only for designing and testing an exit prototype; for best performance, such execs should be EXECLOADed.

Telnet Session Connection Exit

When a Telnet client establishes a session with TCP/IP for VM and InternalClientParms ConnectExit has been specified, the exit routine receives control using standard OS linkage conventions. Register 1 points to a parameter list to be used by the exit.
### Telnet Exit Parameter List

**Table 82. Telnet Session Connection Exit Parameter List**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Len</th>
<th>Name</th>
<th>In/Out</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>1</td>
<td>SCREASON</td>
<td>Input</td>
<td>Reason Exit was called&lt;br&gt;X’01’ - Client connect</td>
</tr>
<tr>
<td>+1</td>
<td>1</td>
<td>SCFLAG1</td>
<td>Output</td>
<td>Flags&lt;br&gt;1... .... - Hide VM logo from client&lt;br&gt;1... .... - Hide command simulation&lt;br&gt;xx xxxx - Reserved</td>
</tr>
<tr>
<td>+2</td>
<td>2</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+4</td>
<td>4</td>
<td>SCIPADDR</td>
<td>Input</td>
<td>IP address of client</td>
</tr>
<tr>
<td>+8</td>
<td>2</td>
<td>SCPART</td>
<td>Input</td>
<td>Telnet server port number</td>
</tr>
<tr>
<td>+10</td>
<td>2</td>
<td>SCCMDL</td>
<td>Input</td>
<td>Length of command buffer&lt;br&gt;Length of command placed in buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>+12</td>
<td>4</td>
<td>SCCPCMD</td>
<td>Input</td>
<td>Address of command buffer</td>
</tr>
<tr>
<td>+16</td>
<td>4</td>
<td>SCRC</td>
<td>Output</td>
<td>Return code&lt;br&gt;0 = Give client VM logo&lt;br&gt;4 = Reject client, no message&lt;br&gt;8 = Perform command in SCCPCMD;&lt;br&gt;S MCCMDL must be non-zero&lt;br&gt;12 = Same as 0, and disable exit&lt;br&gt;16 = Same as 4, and disable exit&lt;br&gt;20 = Same as 8, and disable exit&lt;br&gt;All others will reject the client, and a message is displayed on the TCPI virtual machine console.</td>
</tr>
<tr>
<td>+20</td>
<td>2</td>
<td>SCFPORT</td>
<td>Input</td>
<td>Client foreign port number</td>
</tr>
<tr>
<td>+22</td>
<td>2</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+24</td>
<td>16</td>
<td>SCLUNAME</td>
<td>Input</td>
<td>Client-provided LU name</td>
</tr>
<tr>
<td>+40</td>
<td>4</td>
<td>SCLIPADD</td>
<td>Input</td>
<td>Local IP address to which client connected</td>
</tr>
</tbody>
</table>

### Sample Exit

Sample Telnet connection exit routines are supplied with TCP/IP on the TCPMAINT 591 minidisk. The supplied samples are:

**SCEXIT SEXEC**

REXX exit routine that contains the logic for allowing or denying access by Telnet clients. Your customized exit should be stored on the TCPMAINT 198 disk as file SCEXIT EXEC.

**SCEXIT SAMPASM**

The assembler exit routine called by the Telnet server; the exit is used to call SCEXIT EXEC and to pass results back to the Telnet server. Your customized exit should be stored on the TCPMAINT 198 disk as file SCEXIT ASSEMBLE. The customized ASSEMBLE file must be assembled (by using the VMFHLSM SCEXIT DMSVM command), and the resulting text deck placed on the TCPMAINT 198 disk.

The sample exit is enabled by including the following in PROFILE TCPIP:

```plaintext
InternalClientParms
  ConnectExit SCEXIT
EndInternalClientParms
```
Telnet Printer Management Exit

When a client establishes a TN3270E printer session with TCP/IP for VM and InternalClientParms TN3270EExit has been specified, the exit routine receives control when a printer session is established or terminated. The exit is called using standard CMS linkage conventions. General Register 1 points to a parameter list that the exit may use.

Telnet Printer Management Exit Parameter List

Table 83. Telnet Exit Parameter List

<table>
<thead>
<tr>
<th>Offset</th>
<th>Len</th>
<th>Name</th>
<th>In/Out</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>1</td>
<td>PMXVERS N</td>
<td>Input</td>
<td>Parameter list version number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X‘01’ - Version 1</td>
</tr>
<tr>
<td>+1</td>
<td>1</td>
<td>PMXREASN</td>
<td>Input</td>
<td>Reason exit called</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X‘00’ - Printer connected</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X‘01’ - Printer disconnected</td>
</tr>
<tr>
<td>+2</td>
<td>2</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+4</td>
<td>4</td>
<td>PMXIPADD</td>
<td>Input</td>
<td>Client IP address</td>
</tr>
<tr>
<td>+8</td>
<td>2</td>
<td>PMXFPORT</td>
<td>Input</td>
<td>Client port number</td>
</tr>
<tr>
<td>+A</td>
<td>2</td>
<td>PMXLPRT</td>
<td>Input</td>
<td>Telnet server port number</td>
</tr>
<tr>
<td>+C</td>
<td>4</td>
<td>PMXLDEV</td>
<td>Input</td>
<td>Logical device number</td>
</tr>
<tr>
<td>+10</td>
<td>8</td>
<td>PMXLUNAM</td>
<td>Input</td>
<td>Logical unit name specified by client</td>
</tr>
<tr>
<td>+18</td>
<td>8</td>
<td>PMXUSER</td>
<td>Input</td>
<td>Associated user identifier. If no matching</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TN3270E configuration statement exists, contains “?”</td>
</tr>
<tr>
<td>+20</td>
<td>4</td>
<td>PMXVDEV</td>
<td>Input</td>
<td>Associated virtual device address. If no matching</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TN3270E configuration statement exists, contains “?”</td>
</tr>
<tr>
<td>+24</td>
<td>4</td>
<td>PMXRC</td>
<td>Output</td>
<td>Return code</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 = Accept client</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 = Reject client</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8 = Same as 0, and disable exit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12 = Same as 4, and disable exit</td>
</tr>
</tbody>
</table>

All others will reject client, and a message is displayed on the TCPIP virtual machine console.

Sample Exit

Sample printer management exit routines are supplied with TCP/IP on the TCPCMAINT 591 minidisk. The supplied samples are:

PMEXIT SEXEC

REXX exit routine that contains the logic for allowing or denying access by TN3270 clients. Your customized exit should be stored on the TCPCMAINT 198 disk as file PMEXIT EXEC.

PMEXIT SAMPASM

The assembler exit routine called by the Telnet server; the exit is used to call PMEXIT EXEC and to pass results back to the Telnet server. Your customized exit should be stored on the TCPCMAINT 198 disk as file PMEXIT ASSEMBLE. The customized ASSEMBLE file must be assembled (by using the VMFHLASM PMEXIT DMSVM command, and the resulting text deck placed on the TCPCMAINT 198 disk.

Enable the sample exit by including the following in PROFILE TCPIP:
Telnet Printer Management Exit

InternalClientParms
TN3270EExit PMEXIT
EndInternalClientParms
Chapter 11. FTP Server Exit

The FTP server user exits described in the next sections allow you greater control over commands received by the FTP server and allows for auditing of FTP logins, logouts and file transfers.

Prior to customizing the server exit described in this section, ensure that you have reviewed the exit limitations and customization recommendations presented in the "Customizing Server-specific Exits" section of the TCP/IP Planning and Customization.

The exit is enabled using the FTAUDIT, FTCHKCMD, and FTCHKDIR startup parameters on the SRVRFTP command or by using the FTP SMSG commands. The startup parameters and SMSG commands are documented in TCP/IP Planning and Customization.

Since the use of the FTP exits adversely affects performance, it is advised that processing performed within the exit should be minimized. While calling a REXX exec is useful for designing an exit prototype, a production-level exit should be written entirely in assembler. For best performance, any REXX execs should be EXECLOADed.

The FTP Server Exit

Sample Exit

Sample FTP server exit routines are supplied with TCP/IP on the TCPMAINT 591 minidisk. The supplied samples are:

**FTPEXIT SEXEC**

REXX exit routine that contains sample logic for login and directory control, and FTP command processing. Your customized exit should be stored on the TCPMAINT 198 disk as file FTPEXIT EXEC.

**FTPEXIT SAMPASM**

The assembler exit routine called by the FTP server; the exit is used to call FTPEXIT EXEC and to pass results back to the FTP server. Your customized exit should be stored on the TCPMAINT 198 disk as file FTPEXIT ASSEMBLE. The customized ASSEMBLE file must be assembled (by using the VMFHLASM FTPEXIT DMSVM command), and the resulting text deck placed on the TCPMAINT 198 disk.

These samples are for illustrative purposes only. They should be modified to meet the needs of your installation before placing them in a production environment.

Audit Processing

With the FTP server exit enabled for audit processing, the FTP Exit will be called for each of the following events:

- **LOGIN**
  Auditing occurs following FTP user login validation
- **LOGOUT**
  Logout occurs when a:
FTP Server Exit

- user enters a QUIT command
- user enters a new USER command while already logged in
- connection is dropped by an SMSG DROP command
- client aborts the connection
- connection is closed because the server is shutting down
- connection times out

- DATA TRANSFER
  Data transfers include the following commands:
  - APPE (client append command)
  - STOR, STOU (client put command)
  - RETR (client get command)
  - LIST, NLST (client dir, ls commands)

Note: Audit exit processing is enabled with the FTAUDIT startup parameter or with the SMSG command to enable exits.

Audit Processing Parameter List

Table 84. FTP Exit Audit Parameter List

<table>
<thead>
<tr>
<th>Offset in Decimal</th>
<th>Len</th>
<th>In/Out</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>4</td>
<td>Input</td>
<td>Char</td>
<td>Exit type (AUDX)</td>
</tr>
<tr>
<td>+4</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Version number</td>
</tr>
<tr>
<td>+8</td>
<td>8</td>
<td>Input</td>
<td>Char</td>
<td>FTP server command</td>
</tr>
<tr>
<td>+16</td>
<td>4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of command argument string; the first halfword contains the length.</td>
</tr>
<tr>
<td>+20</td>
<td>8</td>
<td>Input</td>
<td>Char</td>
<td>Login user ID</td>
</tr>
<tr>
<td>+28</td>
<td>8</td>
<td>Input</td>
<td>Char</td>
<td>LOGONBY user ID</td>
</tr>
<tr>
<td>+36</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>IP address of client</td>
</tr>
<tr>
<td>+40</td>
<td>4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of current working directory name; the first halfword contains the length.</td>
</tr>
<tr>
<td>+44</td>
<td>4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of target directory or file; the first halfword contains the length.</td>
</tr>
<tr>
<td>+48</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Number of bytes transferred.</td>
</tr>
<tr>
<td>+52</td>
<td>2</td>
<td>Input</td>
<td>Int</td>
<td>Port number of FTP server.</td>
</tr>
<tr>
<td>+54</td>
<td>2</td>
<td>Input</td>
<td>Int</td>
<td>Port number of FTP client.</td>
</tr>
<tr>
<td>+56</td>
<td>8</td>
<td>Input</td>
<td>Char</td>
<td>Event date (yyyyymmdd)</td>
</tr>
<tr>
<td>+64</td>
<td>8</td>
<td>Input</td>
<td>Char</td>
<td>Event time (hh:mm:ss)</td>
</tr>
<tr>
<td>+72</td>
<td>8</td>
<td></td>
<td></td>
<td>Not used</td>
</tr>
<tr>
<td>+80</td>
<td>4</td>
<td>Output</td>
<td>Int</td>
<td>Return code from exit</td>
</tr>
</tbody>
</table>

Audit Processing Parameter Descriptions

Exit Type
A 4 character field that indicates the type of exit processing to be performed. For audit processing, this is AUDX.
Version number
If the parameter list format is changed, then the version number will change. Your exit should verify it has received the expected version number. The current version number is 1.

FTP server command
This field contains one of the following commands: LOGIN, LOGOUT, XFER.

FTP command argument string
- For data transfer (XFER) commands, this string indicates the transfer direction. SENDING indicates data is being transferred to a client; RECEIVING indicates data is being received from a client.
- For login commands, this string indicates the command that initiated login validation processing (USER for anonymous logins or PASS).
- For logout commands, this string indicates the command or function which initiated the logout (QUIT, USER, DROPPED, TIMEOUT, SHUTDOWN, ABORTED).

Login user ID
The VM user identifier associated with this FTP session. All FTP client authorization checks are made using the login user ID.

LOGONBY user ID
The alternate logon name whose password is used for login authorization checking. A user ID will be present in this field only when the client has issued a USER subcommand that includes the userid/BY/byuserid operands; otherwise, a hyphen (-) will be present.

IP address of client
The IP address in decimal integer form.

Current working directory name
This field is not used for login or logout processing and will contain a hyphen (-). For data transfers, this field contains the type of directory in use, followed by the working directory name. For example:

<table>
<thead>
<tr>
<th>Directory Type</th>
<th>Working Directory passed to FTPEXIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minidisk</td>
<td>DSK TERI.191</td>
</tr>
<tr>
<td>Shared File System</td>
<td>SFS SERVK1:TERI.</td>
</tr>
<tr>
<td>Byte File System</td>
<td>BFS /../VMBFS:BFS:TERI/</td>
</tr>
<tr>
<td>Virtual Reader</td>
<td>RDR TERI.RDR</td>
</tr>
</tbody>
</table>

Target file
Target file for data transfer. Minidisk, SFS, or RDR files are identified in upper case using the filename.filetype format. BFS files are identified using a mixed case filename, that can be up to 255 characters long. This field is not used for login and logout processing and will contain a hyphen (-).

Number of bytes transferred
- For data transfer (XFER) commands, this field contains the number of bytes transferred on the data connection.
- For login commands, this field contains a zero.
- For logout commands, this field contains a zero.

Port number of FTP server
The port number used by the FTP server for this control connection.
FTP Server Exit

Port number of client
The port number used by the foreign host for this control connection.

Event date
The date format for this parameter is yyyymmdd.

Event time
The time format for this parameter is hh:mm:ss.

Return code from exit
An integer return code. For a list of return codes recognized by the FTP server see “Return Codes from Audit Processing”.

Return Codes from Audit Processing

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Use / Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continue processing.</td>
</tr>
</tbody>
</table>
| 8           | Continue processing, but disable audit exit. The following message is displayed on the FTP server console: “FTP AUDX exit has been disabled”.
| Other       | Any return code other than the above causes FTP to issue the message: “Unexpected return from user exit FTPEXIT AUDX, RC = rc”. The server will treat this return code as if it were a return code of 0.

General Command Processing

With the FTP server exit enabled for general command exit processing, the FTP exit will be called to perform command validation for every received FTP command.

Commands that may be passed to this exit follow:

<table>
<thead>
<tr>
<th>ABOR</th>
<th>ACCT</th>
<th>ALLO</th>
<th>APPE</th>
<th>CDUP</th>
<th>CWD</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELE</td>
<td>HELP</td>
<td>LIST</td>
<td>MKD</td>
<td>MODE</td>
<td>NLST</td>
</tr>
<tr>
<td>NOOP</td>
<td>PASS</td>
<td>PASV</td>
<td>PORT</td>
<td>PWD</td>
<td>QUIT</td>
</tr>
<tr>
<td>REIN</td>
<td>REST</td>
<td>RETR</td>
<td>RMD</td>
<td>RNFR</td>
<td>RNTO</td>
</tr>
<tr>
<td>SITE</td>
<td>SYST</td>
<td>STAT</td>
<td>STOR</td>
<td>STOU</td>
<td>STRU</td>
</tr>
<tr>
<td>TYPE</td>
<td>USER</td>
<td>XCWD</td>
<td>XMKD</td>
<td>XPWD</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

**Note:** See RFC 959 for details about each command.

The general command exit can be used to perform additional security checking and then take an appropriate action, such as the following:
- Reject commands from a particular IP address, user ID or LOGONBY user ID
- Reject a subset of commands for anonymous users
- Reject transfer requests for specific files
- Reject all users from issuing store (APPE, STOR, STOU) commands

**Note:** General command exit processing is enabled with the FTCHKCMD startup parameter or with the SMSG command to enable exits.
General Command Processing Parameter List

Table 85. FTP Exit Parameter List

<table>
<thead>
<tr>
<th>Offset in Decimal</th>
<th>Len</th>
<th>In/Out</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0 4</td>
<td>Input</td>
<td>Char</td>
<td>Exit type (CMDX)</td>
<td></td>
</tr>
<tr>
<td>+4 4</td>
<td>Input</td>
<td>Int</td>
<td>Version number</td>
<td></td>
</tr>
<tr>
<td>+8 8</td>
<td>Input</td>
<td>Char</td>
<td>FTP server command</td>
<td></td>
</tr>
<tr>
<td>+16 4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of command argument string; the first halfword contains the length.</td>
<td></td>
</tr>
<tr>
<td>+20 8</td>
<td>Input</td>
<td>Char</td>
<td>Login user ID</td>
<td></td>
</tr>
<tr>
<td>+28 8</td>
<td>Input</td>
<td>Char</td>
<td>LOGONBY user ID</td>
<td></td>
</tr>
<tr>
<td>+36 4</td>
<td>Input</td>
<td>Int</td>
<td>IP address of client</td>
<td></td>
</tr>
<tr>
<td>+40 4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of current working directory name; The first halfword contains the length.</td>
<td></td>
</tr>
<tr>
<td>+44 8</td>
<td>-</td>
<td>-</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td>+52 2</td>
<td>Input</td>
<td>Int</td>
<td>Port number of FTP server</td>
<td></td>
</tr>
<tr>
<td>+54 2</td>
<td>Input</td>
<td>Int</td>
<td>Port number of FTP client</td>
<td></td>
</tr>
<tr>
<td>+56 16</td>
<td>-</td>
<td>-</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td>+72 4</td>
<td>Input</td>
<td>Int</td>
<td>Maximum length of return string</td>
<td></td>
</tr>
<tr>
<td>+76 4</td>
<td>Output</td>
<td>Int</td>
<td>Address of return string (message text)</td>
<td></td>
</tr>
<tr>
<td>+80 4</td>
<td>Output</td>
<td>Int</td>
<td>Return code from exit</td>
<td></td>
</tr>
</tbody>
</table>

General Command Processing Parameter Descriptions

Exit Type
A 4 character field that indicates the type of exit processing to be performed. For command exit processing, this is CMDX.

Version number
If the parameter list format is changed, then the version number will change. Your exit should verify it has received the expected version number. The current version number is 1.

FTP server command
Commands received by the server such as USER, STOR, and DELE.

FTP command argument string
The argument string provided by the client. For ACCT and PASS commands, the argument string will contain all asterisks (******).

Login user ID
The VM user identifier associated with this FTP session. All FTP client authorization checks are made using the login user ID.

LOGONBY user ID
The alternate logon name (userid) whose password is used for login authorization checking. A user ID will be present in this field only when the client has issued a USER subcommand that includes the userid/BY/byuserid operands; otherwise, a hyphen (-) will be present.

IP address of client
The IP address in decimal integer form.
FTP Server Exit

**Current working directory name**
This field contains the type of directory, followed by the working directory name. For example:

<table>
<thead>
<tr>
<th>Directory Type</th>
<th>Working Directory passed to FTPEXIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minidisk</td>
<td>DSK TERI.191</td>
</tr>
<tr>
<td>Shared File System</td>
<td>SFS SERVK1:TERI.</td>
</tr>
<tr>
<td>Byte File System</td>
<td>BFS /../VMBFS:BFS:TERI/</td>
</tr>
<tr>
<td>Virtual reader</td>
<td>RDR TERI.RDR</td>
</tr>
<tr>
<td>No directory defined</td>
<td>-</td>
</tr>
</tbody>
</table>

**Port number of FTP server**
The port number used by the FTP server for this control connection.

**Port number of client**
The port number used by the foreign host for this control connection.

**Maximum length of return string**
The current maximum is 1000 bytes. If the returned string is longer than the maximum, the return string is truncated.

**Return string**
A return string is to be included as part of the server reply to an FTP client. This string is used only when a return code of 4 or 12 is returned by the exit.

**Return code from exit**
An integer return code. For a list of return codes recognized by the FTP server see "Return Codes from General Command Processing".

**Example**
If the FTP client provides the command “PUT PROFILE.EXEC”, the parameter values provided to the FTPEXIT might be:

<table>
<thead>
<tr>
<th>Exit Type</th>
<th>: AUDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTP command</td>
<td>: XFER</td>
</tr>
<tr>
<td>Client IP Address</td>
<td>: 9.111.32.29</td>
</tr>
<tr>
<td>UserID</td>
<td>: TERI</td>
</tr>
<tr>
<td>ByUserID</td>
<td>: -</td>
</tr>
<tr>
<td>Bytes transferred</td>
<td>: 2141</td>
</tr>
<tr>
<td>Server Port</td>
<td>: 1021</td>
</tr>
<tr>
<td>Client Port</td>
<td>: 21400</td>
</tr>
<tr>
<td>Event Date</td>
<td>: 19990309</td>
</tr>
<tr>
<td>Event Time</td>
<td>: 14:44:34</td>
</tr>
<tr>
<td>Working Directory</td>
<td>: SFS SERVK1:TERI.</td>
</tr>
<tr>
<td>Command args</td>
<td>: RECEIVING</td>
</tr>
<tr>
<td>Target File</td>
<td>: PROFILE.EXEC</td>
</tr>
</tbody>
</table>

**Return Codes from General Command Processing**

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Use / Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Accept command and continue processing</td>
</tr>
</tbody>
</table>
### Change Directory Processing

With the FTP server the exit will be called to validate FTP directory changes and provide greater control over access to system resources by selectively honoring or refusing a client change directory request. The exit is called when an FTP client provides one of the following commands:

- **CWD** or **CD** to change the working directory
- **CDUP** to change the working directory to the parent directory
- **PASS** with a default directory defined in CHKIPADR EXEC
- **USER** for an anonymous login with a default directory defined in CHKIPADR EXEC
- **APPE, DELE, LIST, NLST, RETR, SIZE, STOR, and STOU** commands that involve an explicit change in directory.

**Notes:**

1. CD command exit processing is enabled with the **FTCHKDIR** startup parameter or with the **SMSG** command to enable exits.
2. The CD command exit cannot be used to alter the directory name provided by the client.

### Change Directory Processing Parameter List

**Table 86. FTP Exit Parameter List**

<table>
<thead>
<tr>
<th>Offset in Decimal</th>
<th>Len</th>
<th>In/Out</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0 4</td>
<td>Input</td>
<td>Char</td>
<td>Exit type (DIRX)</td>
<td></td>
</tr>
<tr>
<td>+4 4</td>
<td>Input</td>
<td>Int</td>
<td>Version number</td>
<td></td>
</tr>
<tr>
<td>+8 8</td>
<td>Input</td>
<td>Char</td>
<td>FTP server command</td>
<td></td>
</tr>
<tr>
<td>+16 4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of command argument string; the first halfword contains the length.</td>
<td></td>
</tr>
<tr>
<td>+20 8</td>
<td>Input</td>
<td>Char</td>
<td>Login user ID</td>
<td></td>
</tr>
<tr>
<td>+28 8</td>
<td>Input</td>
<td>Char</td>
<td>LOGONBY user ID</td>
<td></td>
</tr>
<tr>
<td>+36 4</td>
<td>Input</td>
<td>Int</td>
<td>IP address of client</td>
<td></td>
</tr>
<tr>
<td>+40 4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of current working directory name; the first halfword contains the length.</td>
<td></td>
</tr>
<tr>
<td>+44 4</td>
<td>Input</td>
<td>Ptr</td>
<td>Address of target directory or file; the first halfword contains the length.</td>
<td></td>
</tr>
<tr>
<td>+48 4</td>
<td>-</td>
<td>-</td>
<td>Not used</td>
<td></td>
</tr>
</tbody>
</table>
Table 86. FTP Exit Parameter List (continued)

<table>
<thead>
<tr>
<th>Offset in Decimal</th>
<th>Len</th>
<th>In/Out</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+52</td>
<td>2</td>
<td>Input</td>
<td>Int</td>
<td>Port number of FTP server</td>
</tr>
<tr>
<td>+54</td>
<td>2</td>
<td>Input</td>
<td>Int</td>
<td>Port number of FTP client</td>
</tr>
<tr>
<td>+56</td>
<td>16</td>
<td>-</td>
<td>-</td>
<td>Not used</td>
</tr>
<tr>
<td>+72</td>
<td>4</td>
<td>Input</td>
<td>Int</td>
<td>Maximum length of return string</td>
</tr>
<tr>
<td>+76</td>
<td>4</td>
<td>Output</td>
<td>Int</td>
<td>Address of return string (message text)</td>
</tr>
<tr>
<td>+80</td>
<td>4</td>
<td>Output</td>
<td>Int</td>
<td>Return code from exit</td>
</tr>
</tbody>
</table>

Change Directory Processing Parameter Descriptions

Exit Type
A 4 character field that indicates the type of exit processing to be performed. For CD command exit processing, this is DIRX.

Version number
If the parameter list format is changed, then the version number will change. Your exit should verify it has received the expected version number. The current version number is 1.

FTP server command
This field will contain APPE, CWD, CDUP, DELE, LIST, NLST, PASS, RETR, SIZE, STOR, STOU, or USER.

FTP command argument string
The argument string entered by the client. For PASS commands, the argument string will contain asterisks (********).

Login user ID
The VM user identifier associated with this FTP session. All FTP client authorization checks are made using the login user ID.

LOGONBY user ID
The alternate logon name (userid) whose password is used for login authorization checking. A user ID will be present in this field only when the client has issued a USER subcommand that includes the userid/BY/byuserid operands; otherwise, a hyphen (-) will be present.

IP address of client
The IP address in decimal integer form.

Current working directory name
This field contains the type of directory, followed by the working directory name. For example:

<table>
<thead>
<tr>
<th>Directory Type</th>
<th>Working Directory Passed to FTPEXIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minidisk</td>
<td>DSK TERI.191</td>
</tr>
<tr>
<td>Shared File System</td>
<td>SFS SERVK1:TERI.</td>
</tr>
<tr>
<td>Byte File System</td>
<td>BFS /../VMBFS:BFS:TERI/</td>
</tr>
<tr>
<td>Virtual reader</td>
<td>RDR TERI.RDR</td>
</tr>
<tr>
<td>No directory defined</td>
<td>-</td>
</tr>
</tbody>
</table>
Target directory
This field contains the fully-qualified target directory for the command. Format of the target directory is similar to the current working directory name format. The following examples show representative values that would be passed to the exit for certain actions or requests made by a client user.

For user login:

FTP command : PASS
Working Directory : -
Command args : ********
Target Directory : DSK TERI.191

For a CD to a BFS directory request:

FTP command : CWD
Working Directory : DSK TERI.191
Command args : /../VMBFS:BFS:SCOTT/
Target Directory : BFS /../VMBFS:BFS:SCOTT/

For a CD to a BFS subdirectory request:

FTP command : CWD
Command args : SUBDIR
Target Directory : BFS /../VMBFS:BFS:SCOTT/SUBDIR/

Port number of FTP server
The port number used by the FTP server for this control connection.

Port number of client
The port number used by the foreign host for this control connection.

Maximum length of return string
The current maximum is 1000 bytes. If the returned string is longer than the maximum, the return string is truncated.

Return string
A return string is to be included as part of the server reply to an FTP client. This string is used only when a return code of 4 or 12 is returned by the exit.

Return code from exit
An integer return code. For a list of return codes recognized by the FTP server see [Return Codes from the FTPEXIT Routine for CD Command Processing].

Return Codes from the FTPEXIT Routine for CD Command Processing

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Use / Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Accept command and continue normal processing</td>
</tr>
<tr>
<td>4</td>
<td>Reject client command with “502 return_string”. If return string is not provided, display the default message “502 command rejected”.</td>
</tr>
<tr>
<td>8</td>
<td>Accept command, continue normal processing, but disable CD command exit processing. The following message is displayed on the FTP server console: “FTP DIRX exit has been disabled”.</td>
</tr>
</tbody>
</table>
## FTP Server Exit

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Use / Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Same as 4 and disable CD command exit processing. The following message is displayed on the FTP server console: “FTP DIRX exit has been disabled”.</td>
</tr>
<tr>
<td>Other</td>
<td>Any return code other than the above causes FTP to issue the message: “Unexpected return from user exit FTPEXIT DIRX, RC = rc”. The server will treat this return code as if it were a return code of 0.</td>
</tr>
</tbody>
</table>
Appendix A. TCPLOAD EXEC

The TCPLOAD EXEC is provided to generate an executable module from your compiled program. When running TCPLOAD, all disks containing object files must be accessed as extensions of the A-disk. The TCPLOAD EXEC generates a module when given a list of text file names and a control file.

```
TCPLOAD load_list control_file type XA TXTLIB filename filename
```

Parameter | Description
---|---
load_list | Specifies the file name of a file with the file type LOADLIST that contains file names to be included in the load module. The first line in the load_list specifies the name of the main object module. Subsequent lines specify additional object modules to be included in the load module.

control_file | Specifies the control_file, which determines the file types of text files according to the standard update identifier procedure.

type | Specifies the type parameter as one of the following:

- **C** Includes SCEELKED, CMSLIB, RPCLIB, TCPASCAL, TCPLANG, COMMTXT, and CLIB txtlibs.
- **C-ONLY** Includes SCEELKED, RPCLIB, COMMTXT, CMSLIB, and CLIB txtlib.
- **PASCAL** Includes TCPASCAL, TCPLANG, and COMMTXT txtlibs. PASCAL is the default for the type parameter.
- **XA** Specify this option if the application requires storage above the 16Mb line. The application will be generated RMODE ANY and AMODE 31. Pascal applications will require the GLOBAL LOADLIB TCPRTLIB command be issued before being run.
- **TXTLIB** Specifies the TXTLIB option, which allows you to specify up to 50 filenames that will be added to the GLOBAL TXTLIB command. See [Table 2 on page 30](#) for a list of the files necessary for each application.

If TCPLOAD is not used, you must global the appropriate TXTLIB files.

Using TCPLOAD

The following example describes how to use TCPLOAD to generate an executable module from object files.
TCPLOAD

1. Create a file with file type LOADLIST, which contains all the object (TEXT) files to be linked. For example, llistfn loadlist.
2. Create a control file with file type CNTRL, which contains the list of TEXT file types. For example, ctrlfn cntrl.
3. Invoke the TCPLOAD command, as shown in the following example.
   TCPLOAD llistfn ctrlfn C (TXTLIB mylib1 mylib2

   Where:
   • llistfn is the LOADLIST file name
   • ctrlfn is the control file name
   • C is the language of the main program
   • TXTLIB is the keyword that specifies the libraries to link
   • mylib1 and mylib2 are the libraries to link.

The following is an example of how to create an executable module from a list of object files. In the example, OBJ1, OBJ2, OBJ3, OBJ4, and OBJ5 are TEXT files created by compiling C programs, and MYLIB1 and MYLIB2 are libraries.

1. Create the file SAMPLE LOADLIST that lists the object files:
   OBJ1
   OBJ2
   OBJ3
   OBJ4
   OBJ5

2. Create the file TEST CNTRL with the following:
   TEXT

3. Invoke TCPLOAD with the following command:
   TCPLOAD SAMPLE TEST C (TXTLIB MYLIB1 MYLIB2

   This creates the executable file SAMPLE MODULE.
Appendix B. Pascal Return Codes

When using Pascal procedure calls, check to determine whether the call has been completed successfully. Use the SayCalRe function (see “SayCalRe” on page 72) to convert the ReturnCode parameter to a printable form.

The SayCalRe function converts a return code value into a descriptive message. For example, if SayCalRe is invoked with the integer constant BADlengthARGUMENT, it returns the message buffer length specified. For a description of Pascal return codes and their equivalent message text from SayCalRe, see Table 87.

Most return codes are self-explanatory in the context where they occur. The return codes you see as a result of issuing a TCP/UDP/IP request are in the range −128 to 0. For more information, see the Explanatory Notes at the end of Table 87.

Table 87. Pascal Language Return Codes

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Numeric Value</th>
<th>Message Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>0</td>
<td>OK.</td>
</tr>
<tr>
<td>ABNORMALcondition¹</td>
<td>−1</td>
<td>Abnormal condition during inter-address communication. (VMCF. RC=nn User=xxxxxxxx)</td>
</tr>
<tr>
<td>ALREADYclosing</td>
<td>−2</td>
<td>Connection already closing.</td>
</tr>
<tr>
<td>BADlengthARGUMENT</td>
<td>−3</td>
<td>Invalid length specified.</td>
</tr>
<tr>
<td>CANNOTSendDATA²</td>
<td>−4</td>
<td>Cannot send data.</td>
</tr>
<tr>
<td>CLIENTRestart</td>
<td>−5</td>
<td>Client reinitialized TCP/IP service.</td>
</tr>
<tr>
<td>CONNECTIONalreadyEXISTS</td>
<td>−6</td>
<td>Connection already exists.</td>
</tr>
<tr>
<td>DESTINATIONunreachable</td>
<td>−7</td>
<td>Destination address is unreachable.</td>
</tr>
<tr>
<td>ERRORinPROFILE</td>
<td>−8</td>
<td>Error in profile file; details are in PROFILE.TCPERROR.</td>
</tr>
<tr>
<td>FATALerror³</td>
<td>−9</td>
<td>Fatal inter-address communications error. (VMCF. RC=nn User=xxxxxxxx)</td>
</tr>
<tr>
<td>HASnoPASSWORD</td>
<td>−10</td>
<td>No password in RACF® directory.</td>
</tr>
<tr>
<td>INCORRECTpassword</td>
<td>−11</td>
<td>TCPIP not authorized to access file.</td>
</tr>
<tr>
<td>INVALIDrequest</td>
<td>−12</td>
<td>Invalid request.</td>
</tr>
<tr>
<td>INVALIDUserID</td>
<td>−13</td>
<td>Invalid user ID.</td>
</tr>
<tr>
<td>INVALIDvirtualADDRESS</td>
<td>−14</td>
<td>Invalid virtual address.</td>
</tr>
<tr>
<td>KILLEDbyCLIENT</td>
<td>−15</td>
<td>You aborted the connection.</td>
</tr>
<tr>
<td>LOCALportNOTavailable</td>
<td>−16</td>
<td>The requested local port is not available.</td>
</tr>
<tr>
<td>MINIDISKinUSE</td>
<td>−17</td>
<td>File is in use by someone else and cannot be accessed.</td>
</tr>
<tr>
<td>MINIDISKnotAVAILABLE</td>
<td>−18</td>
<td>File not available.</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1987, 2002
### Pascal Return Codes

**Table 87. Pascal Language Return Codes (continued)**

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Numeric Value</th>
<th>Message Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>NObufferSPACE⁴</td>
<td>−19</td>
<td>No more space for data currently available.</td>
</tr>
<tr>
<td>NOmoreINCOMINGdata</td>
<td>−20</td>
<td>The foreign host has closed this connection.</td>
</tr>
<tr>
<td>NONlocalADDRESS</td>
<td>−21</td>
<td>The internet address is not local to this host.</td>
</tr>
<tr>
<td>NOoutstandingNOTIFICATIONS</td>
<td>−22</td>
<td>No outstanding notifications.</td>
</tr>
<tr>
<td>NOSuchCONNECTION</td>
<td>−23</td>
<td>No such connection.</td>
</tr>
<tr>
<td>NOTcpIPservice</td>
<td>−24</td>
<td>No TCP/IP service available.</td>
</tr>
<tr>
<td>NOTyetBEGUN</td>
<td>−25</td>
<td>Not yet begun TCP/IP service.</td>
</tr>
<tr>
<td>NOTyetOPEN</td>
<td>−26</td>
<td>The connection is not yet open.</td>
</tr>
<tr>
<td>OPENrejected</td>
<td>−27</td>
<td>Foreign host rejected the open attempt.</td>
</tr>
<tr>
<td>PARAMlocalADDRESS</td>
<td>−28</td>
<td>TcpOpen error: invalid local address.</td>
</tr>
<tr>
<td>PARAMstate</td>
<td>−29</td>
<td>TcpOpen error: invalid initial state.</td>
</tr>
<tr>
<td>PARAMtimeout</td>
<td>−30</td>
<td>Invalid time−out parameter.</td>
</tr>
<tr>
<td>PARAMunspecADDRESS</td>
<td>−31</td>
<td>TcpOpen error: unspecified foreign address in active open.</td>
</tr>
<tr>
<td>PARAMunspecPORT</td>
<td>−32</td>
<td>TcpOpen error: unspecified foreign port in active open.</td>
</tr>
<tr>
<td>PROFINotFOUND</td>
<td>−33</td>
<td>TCPIP cannot read profile file.</td>
</tr>
<tr>
<td>RECEIVEstILLPENDING</td>
<td>−34</td>
<td>Receive still pending on this connection.</td>
</tr>
<tr>
<td>REMOTEclose</td>
<td>−35</td>
<td>Foreign host unexpectedly closed the connection.</td>
</tr>
<tr>
<td>REMOTEreset</td>
<td>−36</td>
<td>Foreign host aborted the connection.</td>
</tr>
<tr>
<td>SOFTWAREerror</td>
<td>−37</td>
<td>Software error in TCP/IP!</td>
</tr>
<tr>
<td>TCPIPshUTDOWN</td>
<td>−38</td>
<td>TCP/IP service is being shut down.</td>
</tr>
<tr>
<td>TIMEOUTconnection</td>
<td>−39</td>
<td>Foreign host is no longer responding.</td>
</tr>
<tr>
<td>TIMEOUTopen</td>
<td>−40</td>
<td>Foreign host did not respond within OPEN time−out</td>
</tr>
<tr>
<td>TOOmanyOPENS</td>
<td>−41</td>
<td>Too many open connections already exist.</td>
</tr>
<tr>
<td>UNAUTHORIZEDuser</td>
<td>−43</td>
<td>You are not authorized to issue this command.</td>
</tr>
<tr>
<td>UNEXPECTEDsyn</td>
<td>−44</td>
<td>Foreign host violated the connection protocol.</td>
</tr>
<tr>
<td>UNIMPLEMENTEDrequest</td>
<td>−45</td>
<td>Unimplemented TCP/IP request.</td>
</tr>
<tr>
<td>UNKNOWNhost</td>
<td>−46</td>
<td>Destination host is not known.</td>
</tr>
</tbody>
</table>
### Explanatory Notes

1. **ABNORMALcondition**
   The actual VMCF return code is available in the external integer variable LastVmcfCode, and is included in the output of SayCalRe if called immediately after the error is detected.

2. **CANNOTsendDATA**
   Cannot send data on this connection because the connection state is invalid for sending data.

3. **FATALerror**
   The actual VMCF return code is available in the external integer variable LastVmcfCode, and is included in the output of SayCalRe if called immediately after the error is detected.

4. **NObufferSPACE**
   Applies to this connection only. Space may still be available for other connections.
Appendix C. C API System Return Codes

This appendix provides a reference for system calls. Table 88 provides the system-wide message numbers set by the system calls. These message numbers are contained in the compiler file ERRNO.H and in the TCP file TCPERRNO.H.

Table 88. System Return Codes

<table>
<thead>
<tr>
<th>Message</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPERM</td>
<td>1</td>
<td>Permission denied.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>2</td>
<td>No such file or directory.</td>
</tr>
<tr>
<td>ESRCH</td>
<td>3</td>
<td>No such process.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>4</td>
<td>Interrupted system call.</td>
</tr>
<tr>
<td>EIO</td>
<td>5</td>
<td>I/O error.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>6</td>
<td>No such device or address.</td>
</tr>
<tr>
<td>E2BIG</td>
<td>7</td>
<td>Argument list too long.</td>
</tr>
<tr>
<td>ENOEXEC</td>
<td>8</td>
<td>Exec format error.</td>
</tr>
<tr>
<td>EBADF</td>
<td>9</td>
<td>Bad file number.</td>
</tr>
<tr>
<td>ECHILD</td>
<td>10</td>
<td>No children.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>11</td>
<td>No more processes.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>12</td>
<td>Not enough memory.</td>
</tr>
<tr>
<td>EACCES</td>
<td>13</td>
<td>Permission denied.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>14</td>
<td>Bad address.</td>
</tr>
<tr>
<td>ENOTBLK</td>
<td>15</td>
<td>Block device required.</td>
</tr>
<tr>
<td>EBUSY</td>
<td>16</td>
<td>Device busy.</td>
</tr>
<tr>
<td>EEXIST</td>
<td>17</td>
<td>File exists.</td>
</tr>
<tr>
<td>EDEV</td>
<td>18</td>
<td>Cross device link.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>19</td>
<td>No such device.</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>20</td>
<td>Not a directory.</td>
</tr>
<tr>
<td>EISDIR</td>
<td>21</td>
<td>Is a directory.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>22</td>
<td>Invalid argument.</td>
</tr>
<tr>
<td>ENFILE</td>
<td>23</td>
<td>File table overflow.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>24</td>
<td>Too many open files.</td>
</tr>
<tr>
<td>ENOTTY</td>
<td>25</td>
<td>Inappropriate device call.</td>
</tr>
<tr>
<td>ETEXTBSY</td>
<td>26</td>
<td>Text file busy.</td>
</tr>
<tr>
<td>EFBIG</td>
<td>27</td>
<td>File too large.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>28</td>
<td>No space left on device.</td>
</tr>
<tr>
<td>ESPIPE</td>
<td>29</td>
<td>Illegal seek.</td>
</tr>
<tr>
<td>EROFS</td>
<td>30</td>
<td>Read only file system.</td>
</tr>
<tr>
<td>EMLINK</td>
<td>31</td>
<td>Too many links.</td>
</tr>
<tr>
<td>EPIPE</td>
<td>32</td>
<td>Broken pipe.</td>
</tr>
<tr>
<td>EDOM</td>
<td>33</td>
<td>Argument too large.</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1987, 2002
### Table 88. System Return Codes (continued)

<table>
<thead>
<tr>
<th>Message</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERANGE</td>
<td>34</td>
<td>Result too large.</td>
</tr>
<tr>
<td>EWOULDBLOCK</td>
<td>35</td>
<td>Operation would block.</td>
</tr>
<tr>
<td>EINPROGRESS</td>
<td>36</td>
<td>Operation now in progress.</td>
</tr>
<tr>
<td>EALREADY</td>
<td>37</td>
<td>Operation already in progress.</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>38</td>
<td>Socket operation on non-socket.</td>
</tr>
<tr>
<td>EDESTADDRREQ</td>
<td>39</td>
<td>Destination address required.</td>
</tr>
<tr>
<td>EMSGSIZE</td>
<td>40</td>
<td>Message too long.</td>
</tr>
<tr>
<td>EPROTOTYPE</td>
<td>41</td>
<td>Protocol wrong type for socket.</td>
</tr>
<tr>
<td>ENOPROTOOPT</td>
<td>42</td>
<td>Protocol not available.</td>
</tr>
<tr>
<td>EPROTONOSUPPORT</td>
<td>43</td>
<td>Protocol not supported.</td>
</tr>
<tr>
<td>ESOCKTNOSUPPORT</td>
<td>44</td>
<td>Socket type not supported.</td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td>45</td>
<td>Operation not supported on socket.</td>
</tr>
<tr>
<td>EPFNOSUPPORT</td>
<td>46</td>
<td>Protocol family not supported.</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>47</td>
<td>Address family not supported by protocol family.</td>
</tr>
<tr>
<td>EADDRINUSE</td>
<td>48</td>
<td>Address already in use.</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>49</td>
<td>Cannot assign requested address.</td>
</tr>
<tr>
<td>ENETDOWN</td>
<td>50</td>
<td>Network is down.</td>
</tr>
<tr>
<td>ENETUNREACH</td>
<td>51</td>
<td>Network is unreachable.</td>
</tr>
<tr>
<td>ENETRESET</td>
<td>52</td>
<td>Network dropped connection on reset.</td>
</tr>
<tr>
<td>ECONNABORTED</td>
<td>53</td>
<td>Software caused connection abort.</td>
</tr>
<tr>
<td>ECONNRESET</td>
<td>54</td>
<td>Connection reset by peer.</td>
</tr>
<tr>
<td>ENOBUFS</td>
<td>55</td>
<td>No buffer space available.</td>
</tr>
<tr>
<td>EISCONN</td>
<td>56</td>
<td>Socket is already connected.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>57</td>
<td>Socket is not connected.</td>
</tr>
<tr>
<td>ESHUTDOWN</td>
<td>58</td>
<td>Cannot send after socket shutdown.</td>
</tr>
<tr>
<td>ETOOMANYREFS</td>
<td>59</td>
<td>Too many references: cannot splice.</td>
</tr>
<tr>
<td>ETIMEDOUT</td>
<td>60</td>
<td>Connection timed out.</td>
</tr>
<tr>
<td>ECONNREFUSED</td>
<td>61</td>
<td>Connection refused.</td>
</tr>
<tr>
<td>ELOOP</td>
<td>62</td>
<td>Too many levels of symbolic loops.</td>
</tr>
<tr>
<td>ENAMETOOLONG</td>
<td>63</td>
<td>File name too long.</td>
</tr>
<tr>
<td>EHOSTDOWN</td>
<td>64</td>
<td>Host is down.</td>
</tr>
<tr>
<td>EHOSTUNREACH</td>
<td>65</td>
<td>No route to host.</td>
</tr>
<tr>
<td>ENOTEMPTY</td>
<td>66</td>
<td>Directory not empty.</td>
</tr>
<tr>
<td>EPROCLIM</td>
<td>67</td>
<td>Too many processes.</td>
</tr>
<tr>
<td>EUSERS</td>
<td>68</td>
<td>Too many users.</td>
</tr>
<tr>
<td>EDQUOT</td>
<td>69</td>
<td>Disc quota exceeded.</td>
</tr>
<tr>
<td>Message</td>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>ESTALE</td>
<td>70</td>
<td>Stale NFS file handle.</td>
</tr>
<tr>
<td>EREMOTE</td>
<td>71</td>
<td>Too many levels of remote in path.</td>
</tr>
<tr>
<td>ENOSTR</td>
<td>72</td>
<td>Device is not a stream.</td>
</tr>
<tr>
<td>ETIME</td>
<td>73</td>
<td>Timer expired.</td>
</tr>
<tr>
<td>ENOSR</td>
<td>74</td>
<td>Out of streams resources.</td>
</tr>
<tr>
<td>ENOMSG</td>
<td>75</td>
<td>No message of desired type.</td>
</tr>
<tr>
<td>EBADMSG</td>
<td>76</td>
<td>Trying to read unreadable message.</td>
</tr>
<tr>
<td>EIDRM</td>
<td>77</td>
<td>Identifier removed.</td>
</tr>
<tr>
<td>EDEADLK</td>
<td>78</td>
<td>Deadlock condition.</td>
</tr>
<tr>
<td>ENOLCK</td>
<td>79</td>
<td>No record locks available.</td>
</tr>
<tr>
<td>ENONET</td>
<td>80</td>
<td>Machine is not on the network.</td>
</tr>
<tr>
<td>ERREMOTE</td>
<td>81</td>
<td>Object is remote.</td>
</tr>
<tr>
<td>ENOLINK</td>
<td>82</td>
<td>Link has been severed.</td>
</tr>
<tr>
<td>EADV</td>
<td>83</td>
<td>Advertise error.</td>
</tr>
<tr>
<td>ESRMNT</td>
<td>84</td>
<td>Srmount error.</td>
</tr>
<tr>
<td>ECOMM</td>
<td>85</td>
<td>Communication error on send.</td>
</tr>
<tr>
<td>EPROTO</td>
<td>86</td>
<td>Protocol error.</td>
</tr>
<tr>
<td>EMULTIHOP</td>
<td>87</td>
<td>Multihop attempted.</td>
</tr>
<tr>
<td>EDTDOT</td>
<td>88</td>
<td>Cross mount point.</td>
</tr>
<tr>
<td>EREMCHG</td>
<td>89</td>
<td>Remote address changed.</td>
</tr>
</tbody>
</table>
Appendix D. Well-Known Port Assignments

This appendix 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 ETC SERVICES file.

TCP Well-Known Port Assignments

Table 89 lists the well-known port assignments for TCP.

Table 89. TCP Well-Known Port Assignments

<table>
<thead>
<tr>
<th>Port Number</th>
<th>Keyword</th>
<th>Reserved for</th>
<th>Services Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>rje</td>
<td>remote job entry</td>
<td>remote job entry</td>
</tr>
<tr>
<td>7</td>
<td>echo</td>
<td>echo</td>
<td>echo</td>
</tr>
<tr>
<td>9</td>
<td>discard</td>
<td>discard</td>
<td>sink null</td>
</tr>
<tr>
<td>11</td>
<td>systat</td>
<td>active users</td>
<td>active users</td>
</tr>
<tr>
<td>13</td>
<td>daytime</td>
<td>daytime</td>
<td>daytime</td>
</tr>
<tr>
<td>15</td>
<td>netstat</td>
<td>Netstat</td>
<td>who is up or Netstat</td>
</tr>
<tr>
<td>19</td>
<td>chargen</td>
<td>ttytst source</td>
<td>character generator</td>
</tr>
<tr>
<td>21</td>
<td>ftp</td>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>23</td>
<td>telnet</td>
<td>Telnet</td>
<td>Telnet</td>
</tr>
<tr>
<td>25</td>
<td>smtp</td>
<td>mail</td>
<td>Simple Mail Transfer Protocol</td>
</tr>
<tr>
<td>37</td>
<td>time</td>
<td>timeserver</td>
<td>timeserver</td>
</tr>
<tr>
<td>39</td>
<td>rlp</td>
<td>resource</td>
<td>Resource Location Protocol</td>
</tr>
<tr>
<td>42</td>
<td>nameserver</td>
<td>name</td>
<td>host name server</td>
</tr>
<tr>
<td>43</td>
<td>nicname</td>
<td>who is</td>
<td>who is</td>
</tr>
<tr>
<td>53</td>
<td>domain</td>
<td>name server</td>
<td>domain name server</td>
</tr>
<tr>
<td>57</td>
<td>mtp</td>
<td>private terminal access</td>
<td>private terminal access</td>
</tr>
<tr>
<td>69</td>
<td>tftp</td>
<td>TFTP</td>
<td>Trivial File Transfer Protocol</td>
</tr>
<tr>
<td>77</td>
<td>rje</td>
<td>netrjs</td>
<td>any private RJE service</td>
</tr>
<tr>
<td>79</td>
<td>finger</td>
<td>finger</td>
<td>finger</td>
</tr>
<tr>
<td>87</td>
<td>link</td>
<td>ttylink</td>
<td>any private terminal link</td>
</tr>
<tr>
<td>95</td>
<td>supdup</td>
<td>supdup</td>
<td>SUPDUP Protocol</td>
</tr>
<tr>
<td>101</td>
<td>hostname</td>
<td>hostname</td>
<td>nic hostname server, usually from SRI-NIC</td>
</tr>
<tr>
<td>109</td>
<td>pop</td>
<td>postoffice</td>
<td>Post Office Protocol</td>
</tr>
<tr>
<td>111</td>
<td>sunrpc</td>
<td>sunrpc</td>
<td>Sun remote procedure call</td>
</tr>
<tr>
<td>113</td>
<td>auth</td>
<td>authentication</td>
<td>authentication service</td>
</tr>
<tr>
<td>115</td>
<td>sftp</td>
<td>sftp</td>
<td>Simple File Transfer Protocol</td>
</tr>
<tr>
<td>117</td>
<td>uucp-path</td>
<td>UUCP path service</td>
<td>UUCP path service</td>
</tr>
<tr>
<td>119</td>
<td>untp</td>
<td>readnews untp</td>
<td>USENET News Transfer Protocol</td>
</tr>
</tbody>
</table>
### TCP Well-Known Port Assignments

Table 89. TCP Well-Known Port Assignments  (continued)

<table>
<thead>
<tr>
<th>Port Number</th>
<th>Keyword</th>
<th>Reserved for</th>
<th>Services Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>ntp</td>
<td>NTP</td>
<td>Network Time Protocol</td>
</tr>
<tr>
<td>160–223</td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>REXEC</td>
<td>REXEC</td>
<td>Remote Execution Protocol</td>
</tr>
<tr>
<td>514</td>
<td>RSH</td>
<td>RSHELL</td>
<td>Remote Shell Service</td>
</tr>
<tr>
<td>712</td>
<td>vice-exec</td>
<td>vice-exec</td>
<td>Andrew File System authenticated service</td>
</tr>
<tr>
<td>713</td>
<td>vlogin</td>
<td>vice-login</td>
<td>Andrew File System authenticated service</td>
</tr>
<tr>
<td>714</td>
<td>vshell</td>
<td>vice-shell</td>
<td>Andrew File System authenticated service</td>
</tr>
<tr>
<td>2001</td>
<td>filesrv</td>
<td></td>
<td>Andrew File System service</td>
</tr>
<tr>
<td>2106</td>
<td>venus.ite</td>
<td></td>
<td>Andrew File System service, for the Venus process</td>
</tr>
</tbody>
</table>

### UDP Well-Known Port Assignments

Table 90 lists the well-known port assignments for UDP.

Table 90. UDP Well-Known Port Assignments

<table>
<thead>
<tr>
<th>Port Number</th>
<th>Keyword</th>
<th>Reserved for</th>
<th>Services Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>rje</td>
<td>remote job entry</td>
<td>remote job entry</td>
</tr>
<tr>
<td>7</td>
<td>echo</td>
<td>echo</td>
<td>echo</td>
</tr>
<tr>
<td>9</td>
<td>discard</td>
<td>discard</td>
<td>sink null</td>
</tr>
<tr>
<td>11</td>
<td>users</td>
<td>active users</td>
<td>active users</td>
</tr>
<tr>
<td>13</td>
<td>daytime</td>
<td>daytime</td>
<td>daytime</td>
</tr>
<tr>
<td>15</td>
<td>netstat</td>
<td>Netstat</td>
<td>Netstat</td>
</tr>
<tr>
<td>19</td>
<td>chargen</td>
<td>ttytst source</td>
<td>character generator</td>
</tr>
<tr>
<td>37</td>
<td>time</td>
<td>timeserver</td>
<td>timeserver</td>
</tr>
<tr>
<td>39</td>
<td>rlp</td>
<td>resource</td>
<td>Resource Location Protocol</td>
</tr>
<tr>
<td>42</td>
<td>nameserver</td>
<td>name</td>
<td>host name server</td>
</tr>
<tr>
<td>43</td>
<td>nicname</td>
<td>who is</td>
<td>who is</td>
</tr>
<tr>
<td>53</td>
<td>domain</td>
<td>nameserver</td>
<td>domain name server</td>
</tr>
<tr>
<td>67</td>
<td>bootpd</td>
<td>BOOTP</td>
<td>BOOTP Daemon</td>
</tr>
<tr>
<td>69</td>
<td>tftp</td>
<td>TFTP</td>
<td>Trivial File Transfer Protocol</td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
<td>any private dial out service</td>
</tr>
<tr>
<td>77</td>
<td>rje</td>
<td>netrjs</td>
<td>any private RJE service</td>
</tr>
<tr>
<td>79</td>
<td>finger</td>
<td>finger</td>
<td>finger</td>
</tr>
<tr>
<td>111</td>
<td>sunrpc</td>
<td>sunrpc</td>
<td>Sun remote procedure call</td>
</tr>
<tr>
<td>123</td>
<td>ntp</td>
<td>NTP</td>
<td>Network Time Protocol</td>
</tr>
<tr>
<td>160–223</td>
<td></td>
<td>reserved</td>
<td>RouteD and MROUTE using Rip</td>
</tr>
<tr>
<td>520</td>
<td></td>
<td></td>
<td>rvd-control</td>
</tr>
<tr>
<td>531</td>
<td></td>
<td></td>
<td>rvd control port</td>
</tr>
<tr>
<td>Port Number</td>
<td>Keyword</td>
<td>Reserved for</td>
<td>Services Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>---------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>2001</td>
<td>rauth2</td>
<td></td>
<td>Andrew File System service, for the Venus process</td>
</tr>
<tr>
<td>2002</td>
<td>rfilebulk</td>
<td></td>
<td>Andrew File System service, for the Venus process</td>
</tr>
<tr>
<td>2003</td>
<td>rfilesrv</td>
<td></td>
<td>Andrew File System service, for the Venus process</td>
</tr>
<tr>
<td>2018</td>
<td>console</td>
<td></td>
<td>Andrew File System service</td>
</tr>
<tr>
<td>2115</td>
<td>ropcons</td>
<td></td>
<td>Andrew File System service, for the Venus process</td>
</tr>
<tr>
<td>2131</td>
<td>rupdsrv</td>
<td></td>
<td>assigned in pairs; bulk must be srv +1</td>
</tr>
<tr>
<td>2132</td>
<td>rupdbulk;</td>
<td></td>
<td>assigned in pairs; bulk must be srv +1</td>
</tr>
<tr>
<td>2133</td>
<td>rupdsrv1</td>
<td></td>
<td>assigned in pairs; bulk must be srv +1</td>
</tr>
<tr>
<td>2134</td>
<td>rupdbulk1;</td>
<td></td>
<td>assigned in pairs; bulk must be srv +1</td>
</tr>
</tbody>
</table>
Well-Known Port Assignments
Appendix E. Related Protocol Specifications

IBM is committed to industry standards. 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 are so useful that they become a recommended protocol. That is, all future implementations for TCP/IP are recommended to implement this particular function or protocol. These become the de facto standards, on which the TCP/IP protocol suite is built.

Many features of TCP/IP for z/VM are based on the following RFCs:

<table>
<thead>
<tr>
<th>RFC</th>
<th>Title</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>768</td>
<td>User Datagram Protocol</td>
<td>J.B. Postel</td>
</tr>
<tr>
<td>791</td>
<td>Internet Protocol</td>
<td>J.B. Postel</td>
</tr>
<tr>
<td>792</td>
<td>Internet Control Message Protocol</td>
<td>J.B. Postel</td>
</tr>
<tr>
<td>793</td>
<td>Transmission Control Protocol</td>
<td>J.B. Postel</td>
</tr>
<tr>
<td>821</td>
<td>Simple Mail Transfer Protocol</td>
<td>J.B. Postel</td>
</tr>
<tr>
<td>822</td>
<td>Standard for the Format of ARPA Internet Text Messages</td>
<td>D. Crocker</td>
</tr>
<tr>
<td>823</td>
<td>DARPA Internet Gateway</td>
<td>R.M. Hinden, A. Sheltzer</td>
</tr>
<tr>
<td>826</td>
<td>Ethernet Address Resolution Protocol: or Converting Network Protocol Addresses to 48.Bit Ethernet Address for Transmission on Ethernet Hardware</td>
<td>D.C. Plummer</td>
</tr>
<tr>
<td>854</td>
<td>Telnet Protocol Specification</td>
<td>J.B. Postel, J.K. Reynolds</td>
</tr>
<tr>
<td>856</td>
<td>Telnet Binary Transmission</td>
<td>J.B. Postel, J.K. Reynolds</td>
</tr>
<tr>
<td>857</td>
<td>Telnet Echo Option</td>
<td>J.B. Postel, J.K. Reynolds</td>
</tr>
<tr>
<td>877</td>
<td>Standard for the Transmission of IP Datagrams over Public Data Networks</td>
<td>J.T. Korb</td>
</tr>
<tr>
<td>885</td>
<td>Telnet End of Record Option</td>
<td>J.B. Postel</td>
</tr>
<tr>
<td>903</td>
<td>Reverse Address Resolution Protocol</td>
<td>R. Finlayson, T. Mann, J.C. Mogul, M. Theimer</td>
</tr>
<tr>
<td>904</td>
<td>Exterior Gateway Protocol Formal Specification</td>
<td>D.L. Mills</td>
</tr>
<tr>
<td>919</td>
<td>Broadcasting Internet Datagrams</td>
<td>J.C. Mogul</td>
</tr>
<tr>
<td>922</td>
<td>Broadcasting Internet Datagrams in the Presence of Subnets</td>
<td>J.C. Mogul</td>
</tr>
<tr>
<td>950</td>
<td>Internet Standard Subnetting Procedure</td>
<td>J.C. Mogul, J.B. Postel</td>
</tr>
<tr>
<td>952</td>
<td>DoD Internet Host Table Specification</td>
<td>K. Harrenstien, M.K. Stahl, E.J. Feinler</td>
</tr>
<tr>
<td>959</td>
<td>File Transfer Protocol</td>
<td>J.B. Postel, J.K. Reynolds</td>
</tr>
<tr>
<td>974</td>
<td>Mail Routing and the Domain Name System</td>
<td>C. Partridge</td>
</tr>
<tr>
<td>1009</td>
<td>Requirements for Internet Gateways</td>
<td>R.T. Braden, J.B. Postel</td>
</tr>
<tr>
<td>1013</td>
<td>X Window System Protocol, Version II: Alpha Update</td>
<td>R.W. Scheifler</td>
</tr>
<tr>
<td>1014</td>
<td>XDR: External Data Representation Standard</td>
<td>Sun Microsystems Incorporated</td>
</tr>
<tr>
<td>1027</td>
<td>Using ARP to Implement Transparent Subnet Gateways</td>
<td>S. Carl-Mitchell, J.S. Quaterman</td>
</tr>
<tr>
<td>1032</td>
<td>Domain Administrators Guide</td>
<td>M.K. Stahl</td>
</tr>
<tr>
<td>RFC</td>
<td>Title</td>
<td>Author</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>1033</td>
<td>Domain Administrators Operations Guide</td>
<td>M. Lottor</td>
</tr>
<tr>
<td>1034</td>
<td>Domain Names—Concepts and Facilities</td>
<td>P.V. Mockapetris</td>
</tr>
<tr>
<td>1035</td>
<td>Domain Names—Implementation and Specification</td>
<td>P.V. Mockapetris</td>
</tr>
<tr>
<td>1042</td>
<td>Standard for the Transmission of IP Datagrams over IEEE 802 Networks</td>
<td>J.B. Postel, J.K. Reynolds</td>
</tr>
<tr>
<td>1055</td>
<td>Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP</td>
<td>J.L. Romkey</td>
</tr>
<tr>
<td>1058</td>
<td>Routing Information Protocol</td>
<td>C.L. Hedrick</td>
</tr>
<tr>
<td>1094</td>
<td>NFS: Network File System Protocol Specification</td>
<td>Sun Microsystems Incorporated</td>
</tr>
<tr>
<td>1112</td>
<td>Host Extensions for IP Multicasting</td>
<td>S. Deering</td>
</tr>
<tr>
<td>1118</td>
<td>Hitchhiker’s Guide to the Internet</td>
<td>E. Krol</td>
</tr>
<tr>
<td>1122</td>
<td>Requirements for Internet Hosts-Communication Layers</td>
<td>R.T. Braden</td>
</tr>
<tr>
<td>1123</td>
<td>Requirements for Internet Hosts-Application and Support</td>
<td>R.T. Braden</td>
</tr>
<tr>
<td>1155</td>
<td>Structure and Identification of Management Information for TCP/IP-Based Internets</td>
<td>M.T. Rose, K. McCloghrie</td>
</tr>
<tr>
<td>1156</td>
<td>Management Information Base for Network Management of TCP/IP-based Internets</td>
<td>K. McCloghrie, M.T. Rose</td>
</tr>
<tr>
<td>1157</td>
<td>Simple Network Management Protocol (SNMP),</td>
<td>J.D. Case, M. Fedor, M.L. Schoffstall, C. Davin</td>
</tr>
<tr>
<td>1179</td>
<td>Line Printer Daemon Protocol</td>
<td>The Wollongong Group, L. McLaughlin III</td>
</tr>
<tr>
<td>1180</td>
<td>TCP/IP Tutorial,</td>
<td>T. J. Socolofsky, C.J. Kale</td>
</tr>
<tr>
<td>1183</td>
<td>New DNS RR Definitions (Updates RFC 1034, RFC 1035)</td>
<td>C.F. Everhart, L.A. Mamakos, R. Ullmann, P.V. Mockapetris,</td>
</tr>
<tr>
<td>1187</td>
<td>Bulk Table Retrieval with the SNMP</td>
<td>M.T. Rose, K. McCloghrie, J.R. Davin</td>
</tr>
<tr>
<td>1188</td>
<td>Proposed Standard for the Transmission of IP Datagrams over FDDI Networks</td>
<td>D. Katz</td>
</tr>
<tr>
<td>1198</td>
<td>FYI on the X Window System</td>
<td>R.W. Scheifler</td>
</tr>
<tr>
<td>1207</td>
<td>FYI on Questions and Answers: Answers to Commonly Asked Experienced Internet User Questions</td>
<td>G.S. Malkin, A.N. Marine, J.K. Reynolds</td>
</tr>
<tr>
<td>1208</td>
<td>Glossary of Networking Terms</td>
<td>O.J. Jacobsen, D.C. Lynch</td>
</tr>
<tr>
<td>1213</td>
<td>Management Information Base for Network Management of TCP/IP-Based Internets: MIB-II,</td>
<td>K. McCloghrie, M.T. Rose</td>
</tr>
<tr>
<td>1215</td>
<td>Convention for Defining Traps for Use with the SNMP</td>
<td>M.T. Rose</td>
</tr>
<tr>
<td>1228</td>
<td>SNMP-DPI Simple Network Management Protocol Distributed Program Interface</td>
<td>G.C. Carpenter, B. Wijnen</td>
</tr>
<tr>
<td>1229</td>
<td>Extensions to the Generic-Interface MIB</td>
<td>K. McCloghrie</td>
</tr>
<tr>
<td>1230</td>
<td>IEEE 802.4 Token Bus MIB IEEE 802 4 Token Bus MIB</td>
<td>K. McCloghrie, R. Fox</td>
</tr>
<tr>
<td>1231</td>
<td>IEEE 802.5 Token Ring MIB IEEE 802.5 Token Ring MIB</td>
<td>K. McCloghrie, R. Fox, E. Decker</td>
</tr>
<tr>
<td>RFC</td>
<td>Title</td>
<td>Author</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>1267</td>
<td>A Border Gateway Protocol 3 (BGP-3)</td>
<td>K. Lougheed, Y. Rekhter</td>
</tr>
<tr>
<td>1268</td>
<td>Application of the Border Gateway Protocol in the Internet</td>
<td>Y. Rekhter, P. Gross</td>
</tr>
<tr>
<td>1269</td>
<td>Definitions of Managed Objects for the Border Gateway Protocol (Version 3)</td>
<td>S. Willis, J. Burruss</td>
</tr>
<tr>
<td>1293</td>
<td>Inverse Address Resolution Protocol</td>
<td>T. Bradley, C. Brown</td>
</tr>
<tr>
<td>1270</td>
<td>SNMP Communications Services</td>
<td>F. Kastenholz, ed.</td>
</tr>
<tr>
<td>1323</td>
<td>TCP Extensions for High Performance</td>
<td>V. Jacobson, R. Braden, D. Borman</td>
</tr>
<tr>
<td>1325</td>
<td>FYI on Questions and Answers: Answers to Commonly Asked New Internet User Questions</td>
<td>G.S. Malkin, A.N. Marine</td>
</tr>
<tr>
<td>1350</td>
<td>TFTP Protocol</td>
<td>K.R. Sollins</td>
</tr>
<tr>
<td>1351</td>
<td>SNMP Administrative Model</td>
<td>J. Davin, J. Galvin, K. McCloghrie</td>
</tr>
<tr>
<td>1352</td>
<td>SNMP Security Protocols</td>
<td>J. Galvin, K. McCloghrie, J. Davin</td>
</tr>
<tr>
<td>1353</td>
<td>Definitions of Managed Objects for Administration of SNMP Parties</td>
<td>K. McCloghrie, J. Davin, J. Galvin</td>
</tr>
<tr>
<td>1354</td>
<td>IP Forwarding Table MIB</td>
<td>F. Baker</td>
</tr>
<tr>
<td>1356</td>
<td>Multiprotocol Interconnect on X.25 and ISDN in the Packet Mode</td>
<td>A. Malis, D. Robinson, R. Ullmann</td>
</tr>
<tr>
<td>1374</td>
<td>IP and ARP on HIPPI</td>
<td>J. Renwick, A. Nicholson</td>
</tr>
<tr>
<td>1381</td>
<td>SNMP MIB Extension for X.25 LAPB</td>
<td>D. Throop, F. Baker</td>
</tr>
<tr>
<td>1382</td>
<td>SNMP MIB Extension for the X.25 Packet Layer</td>
<td>D. Throop</td>
</tr>
<tr>
<td>1387</td>
<td>RIP Version 2 Protocol Analysis</td>
<td>G. Malkin</td>
</tr>
<tr>
<td>1389</td>
<td>RIP Version 2 MIB Extension</td>
<td>G. Malkin</td>
</tr>
<tr>
<td>1390</td>
<td>Transmission of IP and ARP over FDDI Networks</td>
<td>D. Katz</td>
</tr>
<tr>
<td>1393</td>
<td>Traceroute Using an IP Option</td>
<td>G. Malkin</td>
</tr>
<tr>
<td>1397</td>
<td>Default Route Advertisement In BGP2 And BGP3 Versions of the Border Gateway Protocol</td>
<td>D. Haskin</td>
</tr>
<tr>
<td>1398</td>
<td>Definitions of Managed Objects for the Ethernet-like Interface Types</td>
<td>F. Kastenholz</td>
</tr>
<tr>
<td>1440</td>
<td>SIFT/UFT:Sender-Initiated/Unsolicited File Transfer</td>
<td>R. Troth</td>
</tr>
<tr>
<td>1483</td>
<td>Multiprotocol Encapsulation over ATM Adaptation Layer 5</td>
<td>J. Heinanen</td>
</tr>
<tr>
<td>1540</td>
<td>IAB Official Protocol Standards</td>
<td>J.B. Postel</td>
</tr>
<tr>
<td>1583</td>
<td>OSPF Version 2</td>
<td>J.Moy</td>
</tr>
<tr>
<td>1647</td>
<td>TN3270 Enhancements</td>
<td>B. Kelly</td>
</tr>
<tr>
<td>1700</td>
<td>Assigned Numbers</td>
<td>J.K. Reynolds, J.B. Postel</td>
</tr>
<tr>
<td>1723</td>
<td>RIP Version 2 — Carrying Additional Information</td>
<td>G. Malkin</td>
</tr>
<tr>
<td>1813</td>
<td>NFS Version 3 Protocol Specification</td>
<td>B. Callaghan, B. Pawlowski, P. Stauback, Sun Microsystems Incorporated</td>
</tr>
<tr>
<td>2060</td>
<td>IMAP Version 4 Protocol Specification</td>
<td>M. Crispin</td>
</tr>
<tr>
<td>2225</td>
<td>Classical IP and ARP over ATM</td>
<td>M. Laubach, J. Halper</td>
</tr>
</tbody>
</table>

These documents can be obtained from:
Many RFCs are available online. Hard copies of all RFCs are available from the NIC, either individually or on a subscription basis. Online copies are available using FTP from the NIC at nic.ddn.mil. Use FTP to download the files, using the following format:

RFC:RFC-INDEX.TXT
RFC:RFCnnnn.TXT
RFC:RFCnnnn.PS

Where:
nnnn   Is the RFC number.
TXT    Is the text format.
PS     Is the PostScript format.

You can also request RFCs through electronic mail, from the automated 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. Information is also available through http://www.internic.net.
Appendix F. Abbreviations and Acronyms

The following abbreviations and acronyms are used throughout this book.

AIX  Advanced Interactive Executive
ANSI  American National Standards Institute
API   Application Program Interface
APPC  Advanced Program-to-Program Communications
APPN® Advanced Peer-to-Peer Networking®
ARP   Address Resolution Protocol
ASCII American National Standard Code for Information Interchange
ASN.1 Abstract Syntax Notation One
ATM   Asynchronous Transfer Mode
AUI   Attachment Unit Interface
BFS   Byte File System
BIOS  Basic Input/Output System
BNC   Bayonet Neill-Concelman
CCITT Comite Consultatif International Telegraphique et Telephonique. The International Telegraph and Telephone Consultative Committee
CETI  Continuously Executing Transfer Interface
CLAW  Common Link Access to Workstation
CLIST Command List
CMS   Conversational Monitor System
CP    Control Program
CPI   Common Programming Interface
CREN  Corporation for Research and Education Networking
CSD   Corrective Service Diskette
CTC   Channel-to-Channel
CU    Control Unit
CUA® Common User Access®
DASD  Direct Access Storage Device
DBCS  Double Byte Character Set
DLL   Dynamic Link Library
DNS   Domain Name System
DOS   Disk Operating System
DPI   Distributed Program Interface
EBCDIC Extended Binary-Coded Decimal Interchange Code
ELANS IBM Ethernet LAN Subsystem
EISA  Enhanced Industry Standard Adapter
ESCON® Enterprise Systems Connection Architecture®
FAT   File Allocation Table
FDDI  Fiber Distributed Data Interface
FTAM  File Transfer Access Management
FTP   File Transfer Protocol
FTP API File Transfer Protocol Applications Programming Interface
GCS   Group Control System
GDDM® Graphical Data Display Manager
GDDMXD Graphics Data Display Manager Interface for X Window System
GDF   Graphics Data File
HCH   HYPERchannel device
## Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIPPI</td>
<td>High Performance Parallel Interface</td>
</tr>
<tr>
<td>HPFS</td>
<td>High Performance File System</td>
</tr>
<tr>
<td>ICMP</td>
<td>Internet Control Message Protocol</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronic Engineers</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IGMP</td>
<td>Internet Group Management Protocol</td>
</tr>
<tr>
<td>ILANS</td>
<td>IBM Token-Ring LAN Subsystem</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IPL</td>
<td>Initial Program Load</td>
</tr>
<tr>
<td>ISA</td>
<td>Industry Standard Adapter</td>
</tr>
<tr>
<td>ISDN</td>
<td>Integrated Services Digital Network</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>IUCV</td>
<td>Inter-User Communication Vehicle</td>
</tr>
<tr>
<td>JES</td>
<td>Job Entry Subsystem</td>
</tr>
<tr>
<td>JIS</td>
<td>Japanese Institute of Standards</td>
</tr>
<tr>
<td>JCL</td>
<td>Job Control Language</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LAPS</td>
<td>LAN Adapter Protocol Support</td>
</tr>
<tr>
<td>LCS</td>
<td>IBM LAN Channel Station</td>
</tr>
<tr>
<td>LPD</td>
<td>Line Printer Daemon</td>
</tr>
<tr>
<td>LPQ</td>
<td>Line Printer Query</td>
</tr>
<tr>
<td>LPR</td>
<td>Line Printer Client</td>
</tr>
<tr>
<td>LPRM</td>
<td>Line Printer Remove</td>
</tr>
<tr>
<td>LPRMON</td>
<td>Line Printer Monitor</td>
</tr>
<tr>
<td>LU</td>
<td>Logical Unit</td>
</tr>
<tr>
<td>MAC</td>
<td>Media Access Control</td>
</tr>
<tr>
<td>Mbps</td>
<td>Megabits per second</td>
</tr>
<tr>
<td>MBps</td>
<td>Megabytes per second</td>
</tr>
<tr>
<td>MCA</td>
<td>Micro Channel® Adapter</td>
</tr>
<tr>
<td>MIB</td>
<td>Management Information Base</td>
</tr>
<tr>
<td>MIH</td>
<td>Missing Interrupt Handler</td>
</tr>
<tr>
<td>MILNET</td>
<td>Military Network</td>
</tr>
<tr>
<td>MHS</td>
<td>Message Handling System</td>
</tr>
<tr>
<td>MTU</td>
<td>Maximum Transmission Unit</td>
</tr>
<tr>
<td>MVS</td>
<td>Multiple Virtual Storage</td>
</tr>
<tr>
<td>MX</td>
<td>Mail Exchange</td>
</tr>
<tr>
<td>NCP</td>
<td>Network Control Program</td>
</tr>
<tr>
<td>NDIS</td>
<td>Network Driver Interface Specification</td>
</tr>
<tr>
<td>NFS</td>
<td>Network File System</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Information Center</td>
</tr>
<tr>
<td>NLS</td>
<td>National Language Support</td>
</tr>
<tr>
<td>NSFNET</td>
<td>National Science Foundation Network</td>
</tr>
<tr>
<td>OS/2®</td>
<td>Operating System/2®</td>
</tr>
<tr>
<td>OSA</td>
<td>Open Systems Adapter</td>
</tr>
<tr>
<td>OSF</td>
<td>Open Software Foundation, Inc.</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>OSIMF/6000</td>
<td>Open Systems Interconnection Messaging and Filing/6000</td>
</tr>
<tr>
<td>OV/MVS</td>
<td>OfficeVision/MVS™</td>
</tr>
<tr>
<td>OV/VM</td>
<td>OfficeVision/VM™</td>
</tr>
<tr>
<td>PAD</td>
<td>Packet Assembly/Disassembly</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>PCA</td>
<td>Parallel Channel Adapter</td>
</tr>
<tr>
<td>PDN</td>
<td>Public Data Network</td>
</tr>
<tr>
<td>PDU</td>
<td>Protocol Data Units</td>
</tr>
<tr>
<td>PING</td>
<td>Packet Internet Groper</td>
</tr>
</tbody>
</table>
### Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIOAM</td>
<td>Parallel I/O Access Method</td>
</tr>
<tr>
<td>POP</td>
<td>Post Office Protocol</td>
</tr>
<tr>
<td>PROFS®</td>
<td>Professional Office Systems</td>
</tr>
<tr>
<td>PSCA</td>
<td>Personal System Channel Attach</td>
</tr>
<tr>
<td>PSDN</td>
<td>Packet Switching Data Network</td>
</tr>
<tr>
<td>PU</td>
<td>Physical Unit</td>
</tr>
<tr>
<td>PVM</td>
<td>Passthrough Virtual Machine</td>
</tr>
<tr>
<td>RACF</td>
<td>Resource Access Control Facility</td>
</tr>
<tr>
<td>RARP</td>
<td>Reverse Address Resolution Protocol</td>
</tr>
<tr>
<td>REXEC</td>
<td>Remote Execution</td>
</tr>
<tr>
<td>REXX</td>
<td>Restructured Extended Executor Language</td>
</tr>
<tr>
<td>RFC</td>
<td>Request For Comments</td>
</tr>
<tr>
<td>RIP</td>
<td>Routing Information Protocol</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>RSCS</td>
<td>Remote Spooling Communications Subsystem</td>
</tr>
<tr>
<td>SAA</td>
<td>System Application Architecture</td>
</tr>
<tr>
<td>SBCS</td>
<td>Single Byte Character Set</td>
</tr>
<tr>
<td>SDLC</td>
<td>Synchronous Data Link Control</td>
</tr>
<tr>
<td>SFS</td>
<td>Shared File System</td>
</tr>
<tr>
<td>SLIP</td>
<td>Serial Line Internet Protocol</td>
</tr>
<tr>
<td>SMIL</td>
<td>Structure for Management Information</td>
</tr>
<tr>
<td>SMTP</td>
<td>Simple Mail Transfer Protocol</td>
</tr>
<tr>
<td>SNA</td>
<td>Systems Network Architecture</td>
</tr>
<tr>
<td>SNMP</td>
<td>Simple Network Management Protocol</td>
</tr>
<tr>
<td>SOA</td>
<td>Start of Authority</td>
</tr>
<tr>
<td>SPOOL</td>
<td>Simultaneous Peripheral Operations Online</td>
</tr>
<tr>
<td>SQL</td>
<td>IBM Structured Query Language</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>TFTP</td>
<td>Trivial File Transfer Protocol</td>
</tr>
<tr>
<td>TSO</td>
<td>Time Sharing Option</td>
</tr>
<tr>
<td>TTL</td>
<td>Time-to-Live</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>VGA</td>
<td>Video Graphic Array</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VMCF</td>
<td>Virtual Machine Communication Facility</td>
</tr>
<tr>
<td>VM/ESA</td>
<td>Virtual Machine/Enterprise System Architecture</td>
</tr>
<tr>
<td>VMSES/E</td>
<td>Virtual Machine Serviceability Enhancements Staged/Extended</td>
</tr>
<tr>
<td>VTAM®</td>
<td>Virtual Telecommunications Access Method</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide Area Network</td>
</tr>
<tr>
<td>XDR</td>
<td>eXternal Data Representation</td>
</tr>
</tbody>
</table>
Abbreviations and Acronyms
Notices

IBM may not offer the products, services, or features discussed in this document in all 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 10594-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
U.S.A.
Attention: Information Request

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 may contain 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 may contain 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 IBM’s application programming interfaces. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.
Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

Advanced Peer-to-Peer Networking  AIX
BookManager  C/370
CICS  Common User Access
CUA  DATABASE 2
DB2  DFSMS/VM
DirMaint  GDDM
IBM  IBMLink
IMS  Language Environment
Micro Channel  MVS
MVS/ESA  MVS/XA
OfficeVision  OfficeVision/MVS
OpenEdition  OpenExtensions
Operating System/2  OS/2
OS/390  PROFS
RACF  S/390
System/390  VM/ESA
zSeries  z/VM

UNIX is a registered trademark in the United States and/or other countries.

NetView is a registered trademark in the United States and other countries licensed exclusively through Tivoli.

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

This glossary describes the most common terms associated with TCP/IP communication in an internet environment, as used in this book.

If you do not find the term you are looking for, see the IBM Dictionary of Computing, New York: McGraw-Hill, 1994.

For abbreviations, the definition usually consists only of the words represented by the letters; for complete definitions, see the entries for the words.

Numerics

3172. IBM Interconnect Controller.

3174. IBM Establishment Controller.

3270. Refers to a series of IBM display devices; for example, the IBM 3275, 3276 Controller Display Station, 3277, 3278, and 3279 Display Stations, the 3290 Information Panel, and the 3287 and 3286 printers. A specific device type is used only when a distinction is required between device types. Information about display terminal usage also refers to the IBM 3138, 3148, and 3158 Display Consoles when used in display mode, unless otherwise noted.

37xx Communication Controller. A network interface used to connect a TCP/IP or z/VM or z/OS network that supports X.25 connections. NCP with X.25 NPSI must be running in the controller, and VTAM must be running on the host.

6611. IBM Network Processor.

8232. IBM LAN Station.

9370. Refers to a series of processors, namely the IBM 9373 Model 20, the IBM 9375 Models 40 and 60, and the IBM 9377 Model 90 and other models.

A

abend. The abnormal termination of a program or task.

abstract syntax. A description of a data structure that is independent of machine-oriented structures and encodings.

Abstract Syntax Notation One (ASN.1). The OSI language for describing abstract syntax.

active gateway. A gateway that is treated like a network interface in that it is expected to exchange routing information, and if it does not do so for a period of time, the route associated with the gateway is deleted.

active open. The state of a connection that is actively seeking a service. Contrast with passive open.

adapter. A piece of hardware that connects a computer and an external device. An auxiliary device or unit used to extend the operation of another system.

address. The unique code assigned to each device or workstation connected to a network. A standard internet address uses a two-part, 32-bit address field. The first part of the address field contains the network address; the second part contains the local address.

address mask. A bit mask used to select bits from an Internet address for subnet addressing. The mask is 32 bits long and selects the network portion of the Internet address and one or more bits of the local portion. It is sometimes called a subnet mask.

address resolution. A means for mapping network layer addresses onto media-specific addresses. See ARP.

Address Resolution Protocol (ARP). A protocol used to dynamically bind an internet address to a hardware address. ARP is implemented on a single physical network and is limited to networks that support broadcast addressing.

address space. A collection of bytes that are allocated, and in many ways managed, as a single entity by CP. Each byte within an address space is identified by a unique address. An address space represents an extent of storage available to a program. Address spaces allocated by VM range in size from 64KB to 2GB.

Advanced Interactive Executive (AIX). IBM’s licensed version of the UNIX operating system.

Advanced Program-to-Program Communications (APPC). The interprogram communication service within SNA LU 6.2 on which the APPC/VM interface is based.

Advanced Research Projects Agency (ARPA). Now called DARPA, its the U.S. Government agency that funded the ARPANET.

Advanced Research Projects Agency Network (ARPANET). A packet switched network developed in the early 1970’s that is the forerunner of today’s Internet. It was decommissioned in June 1990.
agent. As defined in the SNMP architecture, an agent, or an SNMP server is responsible for performing the network management functions requested by the network management stations.

AIX. Advanced Interactive Executive.

American National Standard Code for Information Interchange (ASCII). The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. The default file transfer type for FTP, used to transfer files that contain ASCII text characters.

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. ANSI is sponsored by the Computer and Business Equipment Manufacturer Association and is responsible for establishing voluntary industry standards.

ANSI. American National Standards Institute.

API. Application Program Interface.

APPC. Advanced Program-to-Program Communications.

application. The use to which an information processing system is put, for example, a payroll application, an airline reservation application, a network application.

application layer. The seventh layer of the OSI (Open Systems Interconnection) model for data communication. It defines protocols for user or application programs.

Application Program Interface (API). The formally defined programming-language interface between an IBM system control program or licensed program and its user. APIs allow programmers to write application programs that use the TCP, UDP, and IP layers of the TCP/IP protocol suite.

argument. A parameter passed between a calling program and a called program.

ARP. Address Resolution Protocol.

ARPA. Advanced Research Projects Agency.

ARPANET. Advanced Research Projects Agency Network.

ASCII. American National Standard Code for Information Interchange. The default file transfer type for FTP, used to transfer files that contain ASCII text characters.

ASN.1. Abstract Syntax Notation One.

ASYNC. Asynchronous.

asynchronous (ASYNC). Without regular time relationship; unexpected or unpredictable with respect to the execution of program instruction. See synchronous.

asynchronous communication. A method of communication supported by the operating system that allows an exchange of data with remote device, using either a start-stop line or an X.25 line. Asynchronous communications include the file transfer and the interactive terminal facility support.

Athena Widgets. The X Window widget set developed by MIT for Project Athena.

Attachment Unit Interface (AUI). Connector used with thick Ethernet that often includes a drop cable.

AUI. Attachment Unit Interface.

attention key. A function key on terminals that, when pressed, causes an I/O interruption in the processing unit.

authentication server. The service that reads a Kerberos database to verify that a client making a request for access to an end-service is the client named in the request. The authentication server provides an authenticated client ticket as permission to access the ticket-granting server.

authenticator. Information encrypted by a Kerberos authentication server that a client presents along with a ticket to an end-server as permission to access the service.

authorization. The right granted to a user to communicate with, or to make use of, a computer system or service.

B

backbone. In a local area network multiple-bridge ring configuration, a high-speed link to which rings are connected by means of bridges. A backbone can be configured as a bus or as a ring. In a wide area network, a high-speed link to which nodes or data switching exchanges (DSES) are connected.

background task. A task with which the user is not currently interacting, but continues to run.

baseband. Characteristic of any network technology that uses a single carrier frequency and requires all stations attached to the network to participate in every transmission. See broadband.

Basic Encoding Rules (BER). Standard rules for encoding data units described in ASN.1. Sometimes
incorrectly grouped under the term ASN.1, which correctly refers only to the abstract description language, not the encoding technique.

**Basic Input/Output System (BIOS).** A set of routines that permanently resides in read-only memory (ROM) in a PC. The BIOS performs the most basic tasks, such as sending a character to the printer, booting the computer, and reading the keyboard.

**batch.** An accumulation of data to be processed. A group of records or data processing jobs brought together for processing or transmission. Pertaining to activity involving little or no user action. See interactive

**Bayonet Neill-Concelman (BNC).** A standardized connector used with Thinnet and coaxial cable.

**Because It’s Time NETwork (BITNET).** A network of hosts that use the Network Job Entry (NJE) protocol to communicate. The network is primarily composed of universities, nonprofit organizations, and research centers. BITNET has recently merged with the Computer and Science Network (CSNET) to form the Corporation for Research and Educational Networking (CSNET). See CSNET.

**BER.** Basic Encoding Rules.

**Berkeley Software Distribution (BSD).** Term used when describing different versions of the Berkeley UNIX software, as in “4.3BSD UNIX”.

**BFS.** Byte File System.

**big-endian.** A format for storage or transmission of binary data in which the most significant bit (or byte) comes first. The reverse convention is little-endian.

**BIOS.** Basic Input/Output System.

**BITNET.** Because It’s Time NETwork.

**Blat.** A denial-of-service attack in which the TCP/IP stack is flooded with SYN packets that have spoofed source IP addresses and port numbers that match the destination IP addresses and port numbers. The Blat attack also has the URG flag turned on in the TCP header and has the ability to incrementally spoof the source IP address. Blat is a version of the Land attack.

**block.** A string of data elements recorded, processed, or transmitted as a unit. The elements can be characters, words, or physical records.

**blocking mode.** If the execution of the program cannot continue until some event occurs, the operating system suspends the program until that event occurs.

**BNC.** Bayonet Neill-Concelman.

**BOOTPD.** Bootstrap Protocol Daemon.

**Bootstrap Protocol Daemon (BOOTPD).** The BOOTP daemon responds to client requests for boot information using information contained in a BOOTP machine file.

**bridge.** A router that connects two or more networks and forwards packets among them. The operations carried out by a bridge are done at the physical layer and are transparent to TCP/IP and TCP/IP routing. A functional unit that connects two local area networks (LANs) that use the same logical link control (LLC) procedures but may use different medium access control (MAC) procedures.

**broadband.** Characteristic of any network that multiplexes multiple, independent network carriers onto a single cable. This is usually done using frequency division multiplexing. Broadband technology allows several networks to coexist on one single cable; traffic from one network does not interfere with traffic from another, because the “conversations” happen on different frequencies in the ether, similar to a commercial radio system.

**broadcast.** The simultaneous transmission of data packets to all nodes on a network or subnetwork.

**broadcast address.** An address that is common to all nodes on a network.

**BSD.** Berkeley Software Distribution.

**bus topology.** A network configuration in which only one path is maintained between stations. Any data transmitted by a station is concurrently available to all other stations on the link.

**byte-ordering.** The method of sorting bytes under specific machine architectures. Of the two common methods, little endian byte ordering places the least significant byte first. This method is used in Intel** microprocessors. In the second method, big endian byte ordering, the most significant byte is placed first. This method is used in Motorola microprocessors.

**Byte File System (BFS).** A file system in which a file consists of an ordered sequence of bytes rather than records. BFS files can be organized into hierarchical directories. Byte file systems are enrolled as file spaces in CMS file pools.

**C**

**Carrier Sense Multiple Access with Collision Detection (CSMA/CD).** The access method used by local area networking technologies such as Ethernet.

**case-sensitive.** A condition in which entries for an entry field must conform to a specific lowercase, uppercase, or mixed-case format to be valid.
CCITT.  Comite Consultatif International Telegrafique et Telephonique.

CEC.  Central Electronics Complex.

channel.  A path in a system that connects a processor and main storage with an I/O device.

channel-attached.  Pertaining to attachment of devices directly by data channels (I/O channels) to a computer. Pertaining to devices attached to a controlling unit by cables, rather than by telecommunication lines. Synonymous with local, locally attached.

checksum.  The sum of a group of data associated with the group and used for checking purposes.

CICS.  Customer Information Control System.

Class A network.  An internet network in which the high-order bit of the address is 0. The host number occupies the three, low-order octets.

Class B network.  An internet network in which the high-order bit of the address is 1, and the next high-order bit is 0. The host number occupies the two low-order octets.

Class C network.  An internet network in which the two high-order bits of the address are 1 and the next high-order bit is 0. The host number occupies the low-order octet.

CLAW.  Common Link Access to Workstation.

client.  A function that requests services from a server, and makes them available to the user. In z/OS, an address space that is using TCP/IP services.

client-server model.  A common way to describe network services and the model user processes (programs) of those services. Examples include the name server and resolver paradigm of the DNS and file server/file client relationships such as NFS and diskless hosts.

client-server relationship.  Any device that provides resources or services to other devices on a network is a server. Any device that employs the resources provided by a server is a client. A machine can run client and server processes at the same time.

CLIST.  Command List.

CLPA.  Create Link Pack Area.

CMS.  Conversational Monitor System.


command.  The name and any parameters associated with an action that can be performed by a program. The command is entered by the user; the computer performs the action requested by the command name.

Command List (CLIST).  A list of commands and statements designed to perform a specific function for the user.

command prompt.  A displayed symbol, such as [C:] that requests input from a user.

Common Link Access to Workstation (CLAW).  A continuously executing duplex channel program designed to minimize host interrupts while maximizing channel utilization.

communications adapter.  A hardware feature that enables a computer or device to become a part of a data network.

community name.  A password used by hosts running Simple Network Management Protocol (SNMP) agents to access remote network management stations.

compile.  To translate a program written in a high-level language into a machine language program. The computer actions required to transform a source file into an executable object file.

compiler.  A program that translates a source program into an executable program (an object program).

Computer and Science Network (CSNET).  A large computer network, mostly in the U.S. but with international connections. CSNET sites include universities, research labs, and some commercial companies. It is now merged with BITNET to form CREN. See BITNET.

connection.  An association established between functional units for conveying information. The path between two protocol modules that provides reliable stream delivery service. In an internet, a connection extends from a TCP module on one machine to a TCP module on the other.

Control Program (CP).  The z/VM operating system that manages the real processor’s resources and is responsible for simulating select operating systems, known as virtual machines for individual users. Each virtual machine is the functional equivalent of a real machine.

conversational monitor system (CMS).  A virtual machine operating system that provides general interactive time sharing, problem solving, and program development capabilities, and operates only under control of the z/VM control program.
Corporation for Research and Educational Networking (CREN). A large computer network formed from the merging of BITNET and CSNET. See BITNET and CSNET.

**CP.** Control Program.

**Create Link Pack Area (CLPA).** A parameter specified at startup, which says to create the link pack area.

**CREN.** Corporation for Research and Educational Networking.

**CSMA/CD.** Carrier Sense Multiple Access with Collision Detection.

**CSNET.** Computer and Science Network.

**Customer Information Control System (CICS).** An IBM-licensed program that enables transactions entered at remote terminals to be processed concurrently by user written application programs. It includes facilities for building, using, and maintaining databases.

**D**

daemon. A background process usually started at system initialization that runs continuously and performs a function required by other processes. Some daemons are triggered automatically to perform their task; others operate periodically.

**DASD.** Direct Access Storage Device.

**DARPA.** Defense Advanced Research Projects Agency.

**DATABASE 2 (DB2).** An IBM relational database management system for the z/OS operating system.

**database administrator (DBA).** An individual or group responsible for the rules by which data is accessed and stored. The DBA is usually responsible for database integrity, security, performance and recovery.

**datagram.** A basic unit of information that is passed across the internet, it consists of one or more data packets.

**data link layer.** Layer 2 of the OSI (Open Systems Interconnection) model; it defines protocols governing data packetizing and transmission into and out of each node.

**data set.** The major unit of data storage and retrieval in z/OS, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access. Synonymous with file in z/VM.

**DB2.** DATABASE 2.

**DBA.** Database administrator.

**DBCS.** Double Byte Character Set.

**DDN.** Defense Data Network.

**decryption.** The unscrambling of data using an algorithm that works under the control of a key. The key allows data to be protected even when the algorithm is unknown. Data is unscrambled after transmission.

**default.** A value, attribute, or option that is assumed when none is explicitly specified.

**Defense Advanced Research Projects Agency (DARPA).** The U.S. government agency that funded the ARPANET.

**Defense Data Network (DDN).** Comprises the MILNET and several other Department of Defense networks.

**destination node.** The node to which a request or data is sent.

**DHCPD.** Dynamic Host Configuration Protocol Daemon.

**Direct Access Storage Device (DASD).** A device in which access to data is independent of where data resides on the device.

**directory.** A named grouping of files in a file system.

**Disk Operating System (DOS).** An operating system for computer systems that use disks and diskettes for auxiliary storage of programs and data.

**display terminal.** An input/output unit by which a user communicates with a data-processing system or sub-system. Usually includes a keyboard and always provides a visual presentation of data; for example, an IBM 3179 display.

**Distributed Program Interface (DPI).** A programming interface that provides an extension to the function provided by the SNMP agents.

**DLL.** Dynamic Link Library.

**DNS.** Domain Name System.

**domain.** In an internet, a part of the naming hierarchy. Syntactically, a domain name consists of a sequence of names (labels) separated by periods (dots).

**Domain Name System (DNS).** A system in which a resolver queries name servers for resource records about a host.

**domain naming.** A hierarchical system for naming network resources.

**DoS.** Denial-of-Service.

**DOS.** Disk Operating System.
**dotted-decimal notation.** The syntactic representation for a 32-bit integer that consists of four 8-bit numbers, written in base 10 and separated by periods (dots). Many internet application programs accept dotted decimal notations in place of destination machine names.

**double-byte character set (DBCS).** A set of characters in which each character is represented by two bytes. Languages such as Japanese, Chinese, Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS.

**doubleword.** A contiguous sequence of bits or characters that comprises two computer words and is capable of being addressed as a unit.

**DPI.** Distributed Program Interface.

**Dynamic Host Configuration Protocol Daemon (DHCPD).** The DHCP daemon (DHCPD server) responds to client requests for boot information using information contained in a DHCP machine file. This information includes the IP address of the client, the IP address of the TFTP daemon, and information about the files to request from the TFTP daemon.

**dynamic resource allocation.** An allocation technique in which the resources assigned for execution of computer programs are determined by criteria applied at the moment of need.

**dynamic link library (DLL).** A module containing dynamic link routines that is linked at load or run time.

**E**

**EBCDIC.** Extended binary-coded decimal interchange code.

**EGP.** Exterior Gateway Protocol.

**encapsulation.** A process used by layered protocols in which a lower-level protocol accepts a message from a higher-level protocol and places it in the data portion of the low-level frame. As an example, in Internet terminology, a packet would contain a header from the physical layer, followed by a header from the network layer (IP), followed by a header from the transport layer (TCP), followed by the application protocol data.

**encryption.** The scrambling or encoding of data using an algorithm that works under the control of a key. The key allows data to be protected even when the algorithm is unknown. Data is scrambled prior to transmission.

**ES/9370 Integrated Adapters.** An adapter you can use in TCP/IP for VM to connect into Token-Ring networks and Ethernet networks, as well as TCP/IP networks that support X.25 connections.

**Ethernet.** The name given to a local area packet-switched network technology invented in the early 1970s by Xerox**, Incorporated. Ethernet uses a Carrier Sense Multiple Access/Collision Detection (CSMA/CD) mechanism to send packets.

**EXEC.** In a VM operating system, a user-written command file that contains CMS commands, other user-written commands, and execution control statements, such as branches.

**extended binary-coded decimal interchange code (EBCDIC).** A coded character set consisting of 8-bit coded characters.

**extended character.** A character other than a 7-bit ASCII character. An extended character can be a 1-bit code point with the 8th bit set (ordinal 128-255) or a 2-bit code point (ordinal 256 and greater).

**Exterior Gateway Protocol (EGP).** A reachability routing protocol used by gateways in a two-level internet.

**eXternal Data Representation (XDR).** A standard developed by Sun Microsystems, Incorporated for representing data in machine-independent format.

**F**

**FAT.** File Allocation Table.

**FDDI.** Fiber Distributed Data Interface. Also used to abbreviate Fiber Optic Distributed Data Interface.

**Fiber Distributed Data Interface (FDDI).** The ANSI standard for high-speed transmission over fiber optic cable.

**Fiber Optic Network.** A network based on the technology and standards that define data transmission using cables of glass or plastic fibers carrying visible light. Fiber optic network advantages are: higher transmission speeds, greater carrying capacity, and lighter, more compact cable.

**file.** In z/VM, a named set of records stored or processed as a unit. Synonymous with data set in z/OS.

**File Allocation Table (FAT).** A table used to allocate space on a disk for a file.

**File Transfer Access and Management (FTAM).** An application service element that enables user application processes to manage and access a file system, which may be distributed.

**File Transfer Protocol (FTP).** A TCP/IP protocol used for transferring files to and from foreign hosts. FTP also
provides the capability to access directories. Password protection is provided as part of the protocol.

**foreign host.** Any machine on a network that can be interconnected.

**foreign network.** In an internet, any other network interconnected to the local network by one or more intermediate gateways or routers.

**foreign node.** See **foreign host.**

**Fraggle.** A denial-of-service attack in which a UDP Echo Request is sent to a broadcast or multicast address.

**frame.** The portion of a tape on a line perpendicular to the reference edge, on which binary characters can be written or read simultaneously.

**FTAM.** File Transfer Access and Management.

**FTP.** File Transfer Protocol.

**fullword.** A computer word. In System/370, 32 bits or 4 bytes.

**G**

**gadget.** A windowless graphical object that looks like its equivalent like-named widget but does not support the translations, actions, or pop-up widget children supplied by that widget.

**gateway.** A functional unit that interconnects a local data network with another network having different protocols. A host that connects a TCP/IP network to a non-TCP/IP network at the application layer. See also **router.**

**gather and scatter data.** Two related operations. During the gather operation, data is taken from multiple buffers and transmitted. In the scatter operation, data is received and stored in multiple buffers.

**GC.** Graphics Context.

**GContext.** See **Graphics Context.**

**GCS.** Group Control System.

**GDDM.** Graphical Data Display Manager.

**GDDMXD.** Graphical Data Display Manager interface for X Window System. A graphical interface that formats and displays alphanumeric, data, graphics, and images on workstation display devices that support the X Window System.

**GDF.** Graphics data file.

**Graphical Display Data Manager (GDDM).** A group of routines that allows pictures to be defined and displayed procedurally through function routines that correspond to graphic primitives.

**Graphics Context (GC).** The storage area for graphics output. Also known as **GC** and **GContext.** Used only with graphics that have the same root and depth as the graphics content.

**Group Control System (GCS).** A component of VM/ESA, consisting of a shared segment that you can Initial Program Load (IPL) and run in a virtual machine. It provides simulated z/OS or OS/390 services and unique supervisor services to help support a native SNA network.

**H**

**handle.** A temporary data representation that identifies a file.

**halfword.** A contiguous sequence of bits or characters that constitutes half a fullword and can be addressed as a unit.

**HASP.** Houston automatic spooling priority system.

**HDLC.** High-level Data Link Control.

**header file.** A file that contains constant declarations, type declarations, and variable declarations and assignments. Header files are supplied with all programming interfaces.

**High-level Data Link Control (HDLC).** An ISO protocol for X.25 international communication.

**High Performance File System (HPFS).** An OS/2 file management system that supports high-speed buffer storage, long file names, and extended attributes.

**HiperSockets.** A hardware feature that provides high performance internal communications between LPARs within the same CEC.

**hop count.** The number of gateways or routers through which a packet passes on its way to its destination.

**host.** A computer connected to a network, which provides an access method to that network. A host provides end-user services and can be a client, a server, or a client and server simultaneously.

**Houston automatic spooling priority system (HASP).** A computer program that provides supplementary job management, data management, and task management functions such as control of job flow, ordering of tasks, and spooling.

**HPFS.** High Performance File System.

**HYPERchannel Adapter.** A network interface used to connect a TCP/IP for z/VM or z/OS host into an
existing TCP/IP HYPERchannel network, or to connect
TCP/IP hosts together to create a TCP/IP
HYPERchannel network.

IAB. Internet Activities Board.

ICMP. Internet Control Message Protocol.

IEEE. Institute of Electrical and Electronic Engineers.

IETF. Internet Engineering Task Force.

IGMP. Internet Group Management Protocol (IGMP).

IGP. Interior Gateway Protocol.

include file. A file that contains preprocessor text,
which is called by a program, using a standard
programming call. Synonymous with header file.

IMAP. Internet Message Access Protocol.

IMS. Information Management System.

Information Management System (IMS). A
database/data communication (DB/DC) system that
can manage complex databases and networks.

initial program load (IPL). The initialization
procedure that causes an operating system to
commence operation.

instance. Indicates a label that is used to distinguish
among the variations of the principal name. An instance
allows for the possibility that the same client or service
can exist in several forms that require distinct
authentication.

Institute of Electrical and Electronic Engineers
(IEEE). An electronics industry organization.

Integrated Services Digital Network (ISDN). A
digital, end-to-end telecommunication network that
supports multiple services including, but not limited to,
voice and data.

interactive. Pertaining to a program or a system that
alternately accepts input and then responds. An
interactive system is conversational; that is, a
continuous dialog exists between user and system. See
batch.

Interior Gateway Protocol (IGP). The protocol used to
exchange routing information between collaborating
routers in the Internet. RIP is an example of an IGP.

Internet. The largest internet in the world consisting
of large national backbone nets (such as MILNET,
NSFNET, and CREN) and a myriad of regional and
local campus networks all over the world. The Internet
uses the Internet protocol suite. To be on the Internet,
you must have IP connectivity (be able to TELNET to,
or PING, other systems). Networks with only electronic
mail connectivity are not actually classified as being on
the Internet.

Internet Activities Board (IAB). The technical body
that oversees the development of the Internet suite of
protocols (commonly referred to as TCP/IP). It has two
task forces (the IRTF and the IETF) each charged with
investigating a particular area.

Internet address. A 32-bit address assigned to hosts
using TCP/IP. An internet address consists of a
network number and a local address. Internet addresses
are represented in a dotted-decimal notation and are
used to route packets through the network.

Internet Engineering Task Force (IETF). One of the
task forces of the IAB. The IETF is responsible for
solving short-term engineering needs of the Internet.

International Organization for Standardization (ISO).
An organization of national standards bodies from
various countries established to promote development
of standards to facilitate international exchange of
goods and services, and develop cooperation in
intellectual, scientific, technological, and economic
activity.

internet or internetwork. A collection of packet
switching networks interconnected by gateways,
routers, bridges, and hosts to function as a single,
coordinated, virtual network.

internet address. The unique 32-bit address
identifying each node in an internet. See also address.

Internet Control Message Protocol (ICMP). The part
of the Internet Protocol layer that handles error
messages and control messages.

Internet Group Management Protocol (IGMP). IGMP
is used by IP hosts to report their host group
memberships to multicast routers.

Internet Protocol (IP). The TCP/IP layer between the
higher level host-to-host protocol and the local network
protocols. IP uses local area network protocols to carry
packets, in the form of datagrams, to the next gateway,
router, or destination host.

interoperability. The capability of different hardware
and software by different vendors to effectively
communicate together.

Inter-user communication vehicle (IUCV). A VM
facility for passing data between virtual machines and
VM components.

intrinsics X-Toolkit. A set management mechanism
that provides for constructing and interfacing between
composite X Window widgets, their children, and other
clients. Also, intrinsics provide the ability to organize a
collection of widgets into an application.

**IP.** Internet Protocol.

**IP datagram.** The fundamental unit of information
passed across the Internet. An IP datagram contains
source and destination addresses along with data and a
number of fields that define such things as the length
of the datagram, the header checksum, and flags to say
whether the datagram can be (or has been) fragmented.

**IPL.** Initial Program Load.

**ISDN.** Integrated Services Digital Network.

**ISO.** International Organization for Standardization.

**IUCV.** Inter-User Communication Vehicle.

**J**

**JCL.** Job Control Language.

**JES.** Job Entry Subsystem.

**JIS.** Japanese Institute of Standards.

**Job Control Language (JCL).** A problem-oriented
language designed to express statements in a job that
are used to identify the job or describe its requirements
to an operating system.

**Job Entry Subsystem (JES).** An IBM System/370
licensed program that receives jobs into the system and
processes all output data produced by the jobs.

**JUNET.** The Japanese Academic and Research
Network that connects various UNIX operating
systems.

**K**

**Kanji.** A graphic character set consisting of symbols
used in Japanese ideographic alphabets. Each character
is represented by 2 bytes.

**katakana.** A character set of symbols used on one of
the two common Japanese phonetic alphabets, which is
used primarily to write foreign words phonetically. See
also kanji.

**Kerberos.** A system that provides authentication
service to users in a network environment.

**Kerberos Authentication System.** An authentication
mechanism used to check authorization at the user
level.

**Kiss-of-Death (KOD).** An IGMP based
denial-of-service attack that depletes the stack’s large
envelopes. See KOX.

**KOD.** Kiss-of-Death.

**KOX.** An IGMP based denial-of-service attack that
depletes the stack’s large envelopes and also has source
IP address spoofing. KOX is a version of the
Kiss-of-Death (KOD) attack.

**L**

**LaMail.** The client that communicates with the OS/2
Presentation Manager to manage mail on the network.

**LAN.** Local area network.

**Land.** A denial-of-service attack in which the TCP/IP
stack is flooded with SYN packets that have spoofed
source IP addresses and port numbers that match the
destination IP addresses and port numbers. See Blat.

**Line Printer Client (LPR).** A client command that
allows the local host to submit a file to be printed on a
remote print server.

**Line Printer Daemon (LPD).** The remote printer
server that allows other hosts to print on a printer local
to your host.

**little-endian.** A format for storage or transmission of
binary data in which the least significant bit (or byte)
comes first. The reverse convention is big-endian.

**local area network (LAN).** A data network located on
the user’s premises in which serial transmission is used
for direct data communication among data stations.

**local host.** In an internet, the computer to which a
user’s terminal is directly connected without using the
internet.

**local network.** The portion of a network that is
physically connected to the host without intermediate
gateways or routers.

**logical character delete symbol.** A special editing
symbol, usually the at (@) sign, which causes CP to
delete it and the immediately preceding character from
the input line. If many delete symbols are consecutively
entered, the same number of preceding characters are
deleted from the input line.

**Logical Unit (LU).** An entity addressable within an
SNA-defined network. LUs are categorized by the types
of communication they support.

**LPD.** Line Printer Daemon.

**LPR.** Line Printer Client.

**LU.** Logical Unit.
LU-LU session. In SNA, a session between two logical units (LUs). It provides communication between two end users, or between an end user and an LU services component.

LU type. In SNA, the classification of an LU-LU session in terms of the specific subset of SNA protocols and options supported by the logical units (LUs) for that session.

M

MAC. Media Access Control.

mail gateway. A machine that connects two or more electronic mail systems (often different mail systems on different networks) and transfers messages between them.

Management Information Base (MIB). A standard used to define SNMP objects, such as packet counts and routing tables, that are in a TCP/IP environment.

mapping. The process of relating internet addresses to physical addresses in the network.

mask. A pattern of characters used to control retention or elimination of portions of another pattern of characters. To use a pattern of characters to control retention or elimination of another pattern of characters. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters.

Maximum Transmission Unit (MTU). The largest possible unit of data that can be sent on a given physical medium.

media access control (MAC). The method used by network adapters to determine which adapter has access to the physical network at a given time.

Message Handling System (MHS). The system of message user agents, message transfer agents, message stores, and access units that together provide OSI electronic mail.

MHS. Message Handling System.

MIB. Management Information Base.

microcode. A code, representing the instructions of an instruction set, which is implemented in a part of storage that is not program-addressable.

MILNET. Military Network.

Military Network (MILNET). Originally part of the ARPANET, MILNET was partitioned in 1984 to make it possible for military installations to have reliable network service, while the ARPANET continued to be used for research. See DDN.

minidisk. Logical divisions of a physical direct access storage device.

modem (modulator/demodulator). A device that converts digital data from a computer to an analog signal that can be transmitted on a telecommunication line, and converts the analog signal received to data for the computer.

Motif. see OSF/Motif.

mouse. An input device that is used to move a pointer on the screen and select items.

MPROUTE. Multiple Protocol Routing. Implements the OSPF protocol described in RFC 1583, 1058, and 1723.

MTU. Maximum Transmission Unit.

multicast. The simultaneous transmission of data packets to a group of selected nodes on a network or subnetwork.

multiconnection server. A server that is capable of accepting simultaneous, multiple connections.

Multiple Virtual Storage (MVS). Implies the MVS/ESA, and follow-on OS/390 and z/OS products.

multitasking. A mode of operation that provides for the concurrent performance execution of two or more tasks.

MVS. Multiple Virtual Storage.

N

name server. The server that stores resource records about hosts.

National Science Foundation (NSF). Sponsor of the NSFNET.

National Science Foundation Network (NSFNET). A collection of local, regional, and mid-level networks in the U.S. tied together by a high-speed backbone. NSFNET provides scientists access to a number of supercomputers across the country.

NCP. Network Control Program.

NDB. Network Database.

NDIS. Network Driver Interface Specification.

Netman. This device keyword specifies that this device is a 3172 LAN Channel Station that supports IBM Enterprise-Specific SNMP Management Information Base (MIB) variables for 3172. TCP/IP for VM supports SNMP GET and SNMP GETNEXT operations to request and retrieve 3172 Enterprise-Specific MIB variables. These requests are
answered only by those 3172 devices with the NETMAN option in the PROFILE TCPIP file.

**NetView.** A system 390-based, IBM-licensed program used to monitor, manage, and diagnose the problems of a network.

**network.** An arrangement of nodes and connecting branches. Connections are made between data stations. Physical network refers to the hardware that makes up a network. Logical network refers to the abstract organization overlaid on one or more physical networks. An internet is an example of a logical network.

**network adapter.** A physical device, and its associated software, that enables a processor or controller to be connected to a network.

**network administrator.** The person responsible for the installation, management, control, and configuration of a network.

**Network Control Program (NCP).** An IBM-licensed program that provides communication controller support for single-domain, multiple-domain, and interconnected network capability.

**network database (NDB).** An IBM-licensed program that provides communication controller support for single-domain, multiple-domain, and interconnected network capability. NDB allows interoperability among different database systems, and uses RPC protocol with a client/server type of relationship. NDB is used for data conversion, security, I/O buffer management, and transaction management.

**Network Driver Interface Specification (NDIS).** An industry-standard specification used by applications as an interface with network adapter device drivers.

**network elements.** As defined in the SNMP architecture, network elements are gateways, routers, and hosts that contain management agents responsible for performing the network management functions requested by the network management stations.

**network file system (NFS).** The NFS protocol, which was developed by Sun Microsystems, Incorporated, allows computers in a network to access each other’s file systems. Once accessed, the file system appears to reside on the local host.

**Network Information Center (NIC).** Originally there was only one, located at SRI International and tasked to serve the ARPANET (and later DDN) community. Today, there are many NICs operated by local, regional, and national networks all over the world. Such centers provide user assistance, document service, training, and more.

**Network Job Entry (NJE).** In object distribution, an entry in the network job table that specifies the system action required for incoming network jobs sent by a particular user or group of users. Each entry is identified by the user ID of the originating user or group.

**network layer.** Layer 3 of the Open Systems Interconnection (OSI) model; it defines protocols governing data routing.

**network management stations.** As defined in the SNMP architecture, network management stations, or SNMP clients, execute management applications that monitor and control network elements.

**NFS.** Network file system.

**NIC.** Network Information Center.

**NJE.** Network Job Entry.

**node.** In a network, a point at which one or more functional units connect channels or data circuits. In a network topology, the point at an end of a branch.

**nonblocking mode.** If the execution of the program cannot continue until some event occurs, the operating system does not suspend the program until that event occurs. Instead, the operating system returns an error message to the program.

**NPSI.** X.25 NCP Packet Switching Interface.

**NSF.** National Science Foundation.

**NSFNET.** National Science Foundation Network.

**O**

**octet.** A byte composed of eight binary elements.

**Offload host.** Any device that is handling the TCP/IP processing for the z/OS host where TCP/IP for MVS is installed. Currently, the only supported Offload host is the 3172-3.

**Offload system.** Represents both the z/OS host where TCP/IP for z/OS is installed and the Offload host that is handling the TCP/IP Offload processing.

**open system.** A system with specified standards and that therefore can be readily connected to other systems that comply with the same standards.

**Open Systems Interconnection (OSI).** The interconnection of open systems in accordance with specific ISO standards. The use of standardized procedures to enable the interconnection of data processing systems.

**Operating System/2 (OS/2).** Pertaining to the IBM licensed program that can be used as the operating system for personal computers. The OS/2 licensed program can perform multiple tasks at the same time.
OS/2. Operating System/2.

OSF/Motif. OSF/Motif is an X Window System toolkit defined by Open Software Foundation, Inc. (OSF), which enables the application programmer to include standard graphic elements that have a 3-D appearance. Performance of the graphic elements is increased with gadgets and windowless widgets.

OSI. Open Systems Interconnection.

OSPF. Open Shortest Path First. An Interior Gateway Protocol that distributes routing information within a single Autonomous System.

out-of-band data. Data that is placed in a secondary channel for transmission. Primary and secondary communication channels are created physically by modulation on a different frequency, or logically by specifying a different logical channel. A primary channel can have a greater capacity than a secondary one.

OV. OfficeVision.

P

packet. A sequence of binary digits, including data and control signals, that is transmitted and switched as a composite whole.

Packet Switching Data Network (PSDN). A network that uses packet switching as a means of transmitting data.

parameter. A variable that is given a constant value for a specified application.

parse. To analyze the operands entered with a command.

passive open. The state of a connection that is prepared to provide a service on demand. Contrast with active open.

Partitioned data set (PDS). A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

PC. Personal computer.

PCA. Personal Channel Attach.

PC Network. A low-cost, broadband network that allows attached IBM personal computers, such as IBM 5150 Personal Computers, IBM Computer ATs, IBM PC/XTs, and IBM Portable Personal Computers to communicate and to share resources.

PDS. Partitioned data set.

PDU. Protocol data unit.

peer-to-peer. In network architecture, any functional unit that resides in the same layer as another entity.

Personal Channel Attach (PCA). see Personal System Channel Attach.

Personal Computer (PC). A microcomputer primarily intended for stand-alone use by an individual.

Personal System Channel Attach (PSCA). An adapter card to connect a micro-channel based personal computer (or processor) to a System/370 parallel channel.

physical layer. Layer 1 of the Open Systems Interconnection (OSI) model; it details protocols governing transmission media and signals.

physical unit (PU). In SNA, the component that manages and monitors the resources, such as attached links and adjacent link stations, associated with a node, as requested by an SSPC via an SSPC-PU session. An SSPC activates a session with the physical unit in order to indirectly manage, through the PU, resources of the node such as attached links.

PING. The command that sends an ICMP Echo Request packet to a host, gateway, or router with the expectation of receiving a reply.

Ping-o-Death (POD). A denial-of-service attack in which huge, fragmented ICMP packets are sent.

PM. Presentation Manager.

PMANT. In OS/2, the 3270 client terminal emulation program that is invoked by the PMANT command.

polling. On a multipoint connection or a point-to-point connection, the process whereby data stations are invited one at a time to transmit. Interrogation of devices for such purposes as to avoid contention, to determine operational status, or to determine readiness to send or receive data.

POP. Post Office Protocol.

port. An endpoint for communication between devices, generally referring to a logical connection. A 16-bit number identifying a particular Transmission Control Protocol or User Datagram Protocol resource within a given TCP/IP node.

PORTMAP. Synonymous with Portmapper.

Portmapper. A program that maps client programs to the port numbers of server programs. Portmapper is used with Remote Procedure Call (RPC) programs.

Post Office Protocol (POP). A protocol used for exchanging network mail.
presentation layer. Layer 6 of the Open Systems Interconnections (OSI) model; it defines protocols governing data formats and conversions.

Presentation Manager (PM). A component of OS/2 that provides a complete graphics-based user interface, with pull-down windows, action bars, and layered menus.

principal name. Specifies the unique name of a user (client) or service.

PostScript. A standard that defines how text and graphics are presented on printers and display devices.

process. A unique, finite course of events defined by its purpose or by its effect, achieved under defined conditions. Any operation or combination of operations on data. A function being performed or waiting to be performed. A program in operation; for example, a daemon is a system process that is always running on the system.

Professional Office Systems (PROFS). IBM’s proprietary, integrated office management system used for sending, receiving, and filing electronic mail, and a variety of other office tasks. PROFS has been replaced by OfficeVision. See OfficeVision.

PROFS. Professional Office Systems.

protocol. A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. Protocols can determine low-level details of machine-to-machine interfaces, such as the order in which bits from a byte are sent; they can also determine high-level exchanges between application programs, such as file transfer.

Protocol data unit (PDU). A set of commands used by the SNMP agent to request management station data.

protocol suite. A set of protocols that cooperate to handle the transmission tasks for a data communication system.

PSCA. Personal System Channel Attach.

PSDN. Packet Switching Data Network.

PU. Physical unit.

Public Data Network (PDN). A network established and operated by a telecommunication administration or by a Recognized Private Operating Agency (RPOA) for the specific purpose of providing circuit-switched, packet-switched, and leased-circuit services to the public.

queue. A line or list formed by items in a system waiting for service; for example, tasks to be performed or messages to be transmitted. To arrange in, or form, a queue.

R

R4P3D. A denial-of-service attack in which TCP packets are sent to the stack with no header flags set. R4P3D is an augmented version of the Stream attack.

RACEF. Resource access control facility.

RARP. Reverse Address Resolution Protocol.

read-only access. An access mode associated with a virtual disk directory that lets a user read, but not write or update, any file on the disk directory.

read/write access. An access mode associated with a virtual disk directory that lets a user read and write any file on the disk directory (if write authorized).

realm. One of the three parts of a Kerberos name. The realm specifies the network address of the principal name or instance. This address must be expressed as a fully qualified domain name, not as a “dot numeric” internet address.

recursion. A process involving numerous steps, in which the output of each step is used for the successive step.

reduced instruction-set computer (RISC). A computer that uses a small, simplified set of frequently used instructions for rapid execution.

reentrant. The attribute of a program or routine that allows the same copy of a program or routine to be used concurrently by two or more tasks.

Remote Execution Protocol (REXEC). A protocol that allows the execution of a command or program on a foreign host. The local host receives the results of the command execution. This protocol uses the REXEC command.

remote host. A machine on a network that requires a physical link to interconnect with the network.

remote logon. The process by which a terminal user establishes a terminal session with a remote host.

Remote Procedure Call (RPC). A facility that a client uses to request the execution of a procedure call from a server. This facility includes a library of procedures and an eXternal data representation.

Remote Spooling Communications Subsystem (RSCS). An IBM-licensed program that transfers spool files, commands, and messages between VM users, remote stations, and remote and local batch systems, through HASP-compatible telecommunication facilities.
**Request For Comments (RFC).** A series of documents that covers a broad range of topics affecting internetwork communication. Some RFCs are established as internet standards.

**resolver.** A program or subroutine that obtains information from a name server or local table for use by the calling program.

**resource access control facility (RACF).** An IBM-licensed program that provides for access control by identifying and by verifying the users to the system, authorizing access to protected resources, logging the detected unauthorized attempts to enter the system, and logging the detected accesses to protected resources.

**resource records.** Individual records of data used by the Domain Name System. Examples of resource records include the following: a host’s Internet Protocol addresses, preferred mail addresses, and aliases.

**response unit (RU).** In SNA, a message unit that acknowledges a request unit. It may contain prefix information received in a request unit. If positive, the response unit may contain additional information such as session parameters in response to BIND SESSION. If negative, it contains sense data defining the exception condition.

**Restructured Extended Executor (REXX) language.** A general purpose programming language, particularly suitable for EXEC procedures, XEDIT macros, or programs for personal computing. Procedures, XEDIT macros, and programs written in this language can be interpreted by the Procedures Language VM/REXX interpreter.

**return code.** A code used to influence the execution of succeeding instructions. A value returned to a program to indicate the results of an operation requested by that program.

**Reverse Address Resolution Protocol (RARP).** A protocol that maintains a database of mappings between physical hardware addresses and IP addresses.

**RExec.** Remote Execution Protocol.

**REXX.** Restructured Extended Executor language.

**RFC.** Request For Comments.

**RIP.** Routing Information Protocol.

**RISC.** Reduced instruction-set computer.

**ROUTED.** Routing Daemon.

**router.** A device that connects networks at the ISO Network Layer. A router is protocol-dependent and connects only networks operating the same protocol. Routers do more than transmit data; they also select the best transmission paths and optimum sizes for packets. In TCP/IP, routers operate at the Internetwork layer. See also gateway.

**Routing Information Protocol (RIP).** The protocol that maintains routing table entries for gateways, routers, and hosts.

**routing table.** A list of network numbers and the information needed to route packets to each.

**RPC.** Remote Procedure Call.

**RSCS.** Remote Spooling Communications Subsystem.

**RU.** Response unit.

**S**

**SAA.** Systems Application Architecture.

**SBCS.** Single Byte Character Set.

**SDLC.** Synchronous data link control.

**Sendmail.** The OS/2 mail server that uses Simple Mail Transfer Protocol to route mail from one host to another host on the network.

**serial line.** A network media that is a de facto standard, not an international standard, commonly used for point-to-point TCP/IP connections. Generally, a serial line consists of an RS-232 connection into a modem and over a telephone line.

**semantics.** The relationships of characters or groups of characters to their meanings, independent of the manner of their interpretation and use. The relationships between symbols and their meanings.

**server.** A function that provides services for users. A machine can run client and server processes at the same time.

**SFS.** Shared File System.

**Shared File System (SFS).** A part of CMS that lets users organize their files into groups known as directories and selectively share those files and directories with other users.

**Simple Mail Transfer Protocol (SMTP).** A TCP/IP application protocol used to transfer mail between users on different systems. SMTP specifies how mail systems interact and the format of control messages they use to transfer mail.

**Simple Network Management Protocol (SNMP).** A protocol that allows network management by elements, such as gateways, routers, and hosts. This protocol provides a means of communication between network elements regarding network resources.
Simultaneous Peripheral Operations Online (SPOOL). (Noun) An area of auxiliary storage defined to temporarily hold data during its transfer between peripheral equipment and the processor. (Verb) To use auxiliary storage as a buffer storage to reduce processing delays when transferring data between peripheral equipment and the processing storage of a computer.

Single-byte Character Set (SBCS). A character set in which each character is represented by a one-byte code. Contrast with double-byte character set.

SMI. Structure for Management Information.

SMTP. Simple Mail Transfer Protocol.

Smurf. A denial-of-service attack in which an ICMP Echo Request is sent to a broadcast or multicast address. There are three variants of the Smurf attack. See Smurf-IC, Smurf-OB, and Smurf-RP.

Smurf-IC. A denial-of-service attack in which an ICMP Echo Request is sent to a broadcast or multicast address. "IC" denotes that incoming packets are using the TCP/IP stack to launch an attack. See Smurf-OB and Smurf-RP.

Smurf-OB. A denial-of-service attack in which an ICMP Echo Request is sent to a broadcast or multicast address. "OB" denotes that an outbound ICMP Echo Request matched the description of a Smurf attack. See Smurf-IC and Smurf-RP.

Smurf-RP. A denial-of-service attack in which an ICMP Echo Request is sent to a broadcast or multicast address. "RP" denotes that the ICMP Echo Reply packets being received by the stack do not match any Echo Requests that were sent. See Smurf-IC and Smurf-OB.

SNA. Systems Network Architecture.

SNALINK. SNA Network Link.

SNA Network Link. An SNA network link function of TCP/IP for z/VM and OS/390 hosts running TCP/IP to communicate through an existing SNA backbone.


SOA. Start of authority record.

Socket. An endpoint for communication between processes or applications. A pair consisting of TCP port and IP address, or UDP port and IP address.

Socket Address. An address that results when the port identification number is combined with an internet address.

Socket Interface. An application interface that allows users to write their own applications to supplement those supplied by TCP/IP.

Spoofing. An act of forging and inserting data that is incorrect or not valid. It is most commonly used in reference to IP source spoofing, where the source address in an IP packet header is replaced with a false one, effectively masking the source of the packet (making it difficult to trace back to the originator).

SPOOL. Simultaneous Peripheral Operations Online.

Spoofing. The processing of files created by or intended for virtual readers, punches, and printers. The spool files can be sent from one virtual device to another, from one virtual machine to another, and to read devices.

SQL. Structured Query Language.

SQL/DS. Structured Query Language/Data System.

SSL. Secure Sockets Layer. Provides the secure (encrypted) communication between a remote client and a TCP/IP server.

Start of Authority Record (SOA). In the Domain Name System, the resource record that defines a zone.

Stream. A continuous sequence of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format.

Structured Query Language (SQL). Fourth generation English-like programming language used to perform queries on relational databases.

Structured Query Language/Data System (SQL/DS). An IBM relational database management system for the VM and VSE operating systems.

Structure for Management Information (SMI). The rules used to define the objects that can be accessed through a network management protocol. See also MIB.

Subagent. In the SNMP architecture, a subagent provides an extension to the utility provided by the SNMP agent.

Subdirectory. A directory contained within another directory in a file system hierarchy.

Subnet. A networking scheme that divides a single logical network into smaller physical networks to simplify routing.

Subnet Address. The portion of the host address that identifies a subnet.

Subnet Mask. A mask used in the IP protocol layer to separate the subnet address from the host portion of the address.
**subnet**. Synonymous with *subnet*.

**subsystem**. A secondary or subordinate system, usually capable of operating independent of, or asynchronously with, a controlling system.

**SYNC**. Synchronous.

**synchronous (SYNC)**. Pertaining to two or more processes that depend on the occurrences of a specific event such as common timing signal. Occurring with a regular or predictable time relationship. See *asynchronous*.

**synchronous data link control (SDLC)**. A data link over which communication is conducted using the synchronous data protocol.

**SynFlood**. A denial-of-service attack in which the initiator floods the TCP/IP stack with SYN packets that have spoofed source IP addresses, resulting in the server never receiving the final ACKs needed to complete the three-way handshake in the connection process.

**Systems Application Architecture (SAA)**. A formal set of rules that enables applications to be run without modification in different computer environments.

**Systems Network Architecture (SNA)**. The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks.

**TALK**. An interactive messaging system that sends messages between the local host and a foreign host.

**TCP**. Transmission Control Protocol.


**Telnet**. The Terminal Emulation Protocol, a TCP/IP application protocol for remote connection service. Telnet allows a user at one site to gain access to a foreign host as if the user’s terminal were connected directly to that foreign host.

**terminal emulator**. A program that imitates the function of a particular kind of terminal.

**Terminate and Stay Resident (TSR) program**. A TSR is a program that installs part of itself as an extension of DOS when it is executed.

**TFTP**. Trivial File Transfer Protocol Daemon.

**ticket**. Encrypted information obtained from a Kerberos authentication server or a ticket-granting server. A ticket authenticates a user and, in conjunction with an authenticator, serves as permission to access a service when presented by the authenticated user.

**ticket-granting server**. Grants Kerberos tickets to authenticated users as permission to access an end-service.

**Time Sharing Option (TSO)**. An operating system option; for System/370 system, the option provides interactive time sharing from remote terminals.

**time stamp**. To apply the current system time. The value on an object that is an indication of the system time at some critical point in the history of the object. In query, the identification of the day and time when a query report was created that query automatically provides on each report.

**TN3270**. An informally defined protocol for transmitting 3270 data streams over Telnet.

**token**. In a local network, the symbol of authority passed among data stations to indicate the station temporarily in control of the transmission medium.

**token-bus**. See *bus topology*.

**token ring**. As defined in IEEE 802.5, a communication method that uses a token to control access to the LAN. The difference between a token bus and a token ring is that a token-ring LAN does not use a master controller to control the token. Instead, each computer knows the address of the computer that should receive the token next. When a computer with the token has nothing to transmit, it passes the token to the next computer in line.

**token-ring network**. A ring network that allows unidirectional data transmission between data stations by a token-passing procedure over one transmission medium, so that the transmitted data returns to the transmitting station.

**Transmission Control Protocol (TCP)**. The TCP/IP layer that provides reliable, process-to-process data stream delivery between nodes in interconnected computer networks. TCP assumes that IP (Internet Protocol) is the underlying protocol.

**Transmission Control Protocol/Internet Protocol (TCP/IP)**. A suite of protocols designed to allow communication between networks regardless of the technologies implemented in each network.

**transport layer**. Layer 4 of the Open Systems Interconnection (OSI) model; it defines protocols governing message structure and some error checking.

**TRAP**. An unsolicited message that is sent by an SNMP agent to an SNMP network management station.

**Trivial File Transfer Protocol Daemon (TFTP)**. The TFTP daemon (TFTPD server) transfers files between
the Byte File System (BFS) and TFTP clients. TFTPD supports access to files maintained in a BFS directory structure that is mounted.

**TSO.** Time Sharing Option.

**TSR.** Terminate and stay resident. TSR usually refers to a terminate-and-stay-resident program.

**U**

**UDP.** User Datagram Protocol.

**user.** A function that uses the services provided by a server. A host can be a user and a server at the same time. See client.

**User Datagram Protocol (UDP).** A datagram level protocol built directly on the IP layer. UDP is used for application-to-application programs between TCP/IP hosts.

**user exit.** A point in an IBM-supplied program at which a user routine may be given control.

**user profile.** A description of a user, including user ID, user name, defaults, password, access authorization, and attributes.

**V**

**virtual address.** The address of a location in virtual storage. A virtual address must be translated into a real address to process the data in processor storage.

**Virtual Machine (VM).** Licensed software whose full name is Virtual Machine/Enterprise Systems Architecture (VM/ESA) It is a software operating system that manages the resources of a real processor to provide virtual machines to end users. It includes time-sharing system control program (CP), the conversational monitor system (CMS), the group control system (GCS), and the dump viewing facility (DVF).

**Virtual Machine Communication Facility (VMCF).** A connectionless mechanism for communication between address spaces.

**VM.** Virtual machine.

**virtual storage.** Storage space that can be regarded as addressable main storage by the user of a computer system in which virtual addresses are mapped into real addresses. The size of virtual storage is limited by the addressing scheme of the computing system and by the amount of auxiliary storage available, not by the actual number of main storage locations.

**Virtual Telecommunications Access Method (VTAM).** An IBM-licensed program that controls communication and the flow of data in an SNA network. It provides single-domain, multiple-domain, and interconnected network capability.

**VM.** Virtual Machine.

**VMCF.** Virtual Machine Communication Facility.

**VM/ESA.** Virtual Machine/Enterprise System Architecture

**VMSES/E.** Virtual Machine Serviceability Enhancements Staged/Extended.

**VTAM.** Virtual Telecommunications Access Method.

**W**

**WAN.** Wide area network.

**well-known port.** A port number that has been preassigned for specific use by a specific protocol or application. Clients and servers using the same protocol communicate over the same well-known port.

**wide area network (WAN).** A network that provides communication services to a geographic area larger than that served by a local area network.

**widget.** The basic data type of the X Window System Toolkit. Every widget belongs to a widget class that contains the allowed operations for that corresponding class.

**window.** An area of the screen with visible boundaries through which a panel or portion of a panel is displayed.

**working directory.** The directory in which an application program is found. The working directory becomes the current directory when the application is started.

**X**

**X Client.** An application program which uses the X protocol to communicate windowing and graphics requests to an X Server.

**XDR.** eXternal Data Representation.

**XEDIT.** The CMS facility, containing the XEDIT command and XEDIT subcommands and macros, that lets a user create, change, and manipulate CMS files.

**X Server.** A program which interprets the X protocol and controls one or more screens, a pointing device, a keyboard, and various resources associated with the X Window System, such as Graphics Contexts, Pixmaps, and color tables.
**X Window System.** The X Window System is a protocol designed to support network transparent windowing and graphics. TCP/IP for z/VM and OS/390 provides client support for the X Window System application program interface.

**X Window System API.** An application program interface designed as a distributed, network-transparent, device-independent, windowing and graphics system.

**X Window System Toolkit.** Functions for developing application environments.

**X.25.** A CCITT communication protocol that defines the interface between data terminal equipment and packet switching networks.

**X.25 NCP Packet Switching Interface (X.25 NPSI).** An IBM-licensed program that allows users to communicate over packet switched data networks that have interfaces complying with Recommendation X.25 (Geneva 1980) of the CCITT. It allows SNA programs to communicate with SNA equipment or with non-SNA equipment over such networks.

**Z**

**ZAP.** To modify or dump an individual text file/data set using the ZAP command or the ZAPTEXT EXEC.

**zone.** In the Domain Name System, a zone is a logical grouping of domain names that is assigned to a particular organization. Once an organization controls its own zone, it can change the data in the zone, add new tree sections connected to the zone, delete existing nodes, or delegate new subzones under its zone.
Bibliography

This bibliography lists the IBM publications that provide information about your z/VM system. The z/VM library includes z/VM base publications, publications for additional facilities included with z/VM, and publications for z/VM optional features. For abstracts of z/VM publications and information about current editions and available publication formats, see z/VM: General Information.

z/VM Internet Library

The latest editions of most z/VM publications are available in Adobe Portable Document Format (PDF) and IBM BookManager® format from the z/VM Internet Library:


The z/VM Internet Library also provides other information about z/VM, such as:
- Program directories
- Data areas and control blocks
- Monitor records

VM Collection CD-ROM

The Online Library Omnibus Edition: VM Collection, SK2T-2067, contains libraries in BookManager format for current IBM VM system products and IBM licensed programs that run on VM. It also contains PDF versions of many of these books.

Note: Only unlicensed publications are included.

z/VM Base Publications

Evaluation

- z/VM: General Information, GC24-5991
- z/VM License Information, GC24-6033
- z/VM: Migration Guide, GC24-5996

Installation and Service

- z/VM: Installation Guide, GC24-5992
- z/VM: Service Guide, GC24-5993
- z/VM: VMSES/E Introduction and Reference, GC24-5994

Planning and Administration

- z/VM: CMS File Pool Planning, Administration, and Operation, SC24-5949
- z/VM: CMS Planning and Administration, SC24-6042
- VM/ESA: Connectivity Planning, Administration, and Operation, SC24-5756
- z/VM: CP Planning and Administration, SC24-6043
- z/VM: Dynamic I/O Configuration Planning and Administration, SC24-6044
- z/VM: Group Control System, SC24-5998
- z/VM: Performance, SC24-5999
- z/VM: Running Guest Operating Systems, SC24-5997
- z/VM: Saved Segments Planning and Administration, SC24-6056
- z/VM: System Administration Facility, SC24-6034

Customization

- z/VM: CP Exit Customization, SC24-5953

Operation

- z/VM: System Operation, SC24-6000
- z/VM: Virtual Machine Operation, SC24-6036

Application Programming

- z/VM: CMS Application Development Guide, SC24-6002
- z/VM: CMS Application Development Guide for Assembler, SC24-6003
- z/VM: CMS Application Multitasking, SC24-5961
- z/VM: CMS Callable Services Reference, SC24-6004
- z/VM: CMS Macros and Functions Reference, SC24-6005
- z/VM: CP Programming Services, SC24-6001
- VM/ESA: CPI Communications User’s Guide, SC24-5595
- z/VM: Enterprise Systems Architecture/Extended Configuration Principles of Operation, SC24-5965
- z/VM: OpenExtensions Advanced Application Programming Tools, SC24-5979
### Publications for z/VM Additional Facilities

<table>
<thead>
<tr>
<th>DFSMS/VM®</th>
<th>z/VM: DFSMS/VM Function Level 221 Customization</th>
<th>SC24-6047</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>z/VM: DFSMS/VM Function Level 221 Diagnosis Guide</td>
<td>GC24-6046</td>
</tr>
<tr>
<td></td>
<td>z/VM: DFSMS/VM Function Level 221 Messages and Codes</td>
<td>GC24-6048</td>
</tr>
<tr>
<td></td>
<td>z/VM: DFSMS/VM Function Level 221 Planning Guide</td>
<td>SC24-6049</td>
</tr>
<tr>
<td></td>
<td>z/VM: DFSMS/VM Function Level 221 Removable Media Services</td>
<td>SC24-6050</td>
</tr>
<tr>
<td></td>
<td>z/VM: DFSMS/VM Function Level 221 Storage Administration</td>
<td>SC24-6051</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Language Environment®</th>
<th>z/VM: Language Environment 1.8 C Run-Time Library Reference</th>
<th>SC24-6038</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Language Environment for OS/390 &amp; VM Concepts Guide</td>
<td>GC28-1945</td>
</tr>
<tr>
<td></td>
<td>Language Environment for OS/390 &amp; VM Debugging Guide and Run-Time Messages</td>
<td>SC28-1942</td>
</tr>
<tr>
<td></td>
<td>Language Environment for OS/390 &amp; VM Programming Guide</td>
<td>SC28-1939</td>
</tr>
<tr>
<td></td>
<td>Language Environment for OS/390 &amp; VM Programming Reference</td>
<td>SC28-1940</td>
</tr>
<tr>
<td></td>
<td>Language Environment for OS/390 &amp; VM Run-Time Migration Guide</td>
<td>SC28-1944</td>
</tr>
<tr>
<td></td>
<td>Language Environment for OS/390 &amp; VM Writing Interlanguage Communication Applications</td>
<td>SC28-1943</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>zSeries 900: Open Systems Adapter-Express Customer’s Guide and Reference</td>
<td>SA22-7476</td>
</tr>
</tbody>
</table>
TCP/IP for z/VM

z/VM: TCP/IP Level 430 Diagnosis Guide, GC24-6023
z/VM: TCP/IP Level 430 Messages and Codes, GC24-6022
z/VM: TCP/IP Level 430 Planning and Customization, SC24-6019
z/VM: TCP/IP Level 430 Programmer's Reference, SC24-6021

Publications for z/VM Optional Features

DirMaint™

z/VM: Directory Maintenance Facility Function Level 410 Command Reference, SC24-6025
z/VM: Directory Maintenance Facility Function Level 410 Messages, GC24-6026
z/VM: Directory Maintenance Facility Function Level 410 Tailoring and Administration Guide, SC24-6024

PRF

z/VM: Performance Reporting Facility Function Level 410, SC24-6027

RTM

z/VM: RealTime Monitor Function Level 410, SC24-6028

RACF® for VM

RACF: Command Language Reference, SC28-0733
RACF: Command Language Reference Summary, SX22-0014
RACF: Diagnosis Guide, GY28-1016
RACF: General Information, GC28-0722
RACF: Macros and Interfaces, SC28-1345
RACF: Messages and Codes, SC38-1014
RACF: Migration and Planning, GC23-3054

Other TCP/IP Related Publications

This section lists other publications, outside the z/VM 4.3.0 library, that you may find helpful.

- TCP/IP Tutorial and Technical Overview, GG24-3376
- TCP/IP Illustrated, Volume 1: The Protocols, SR28-5586
- Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture, SC31-6144
- Internetworking With TCP/IP Volume II: Implementation and Internals, SC31-6145
- Internetworking With TCP/IP Volume III: Client-Server Programming and Applications, SC31-6146
- DNS and BIND in a Nutshell, SR28-4970
Index

A
abbreviations and acronyms 373
ACCEPT (IUCV) 125
address families, socket 4
address information file, specifying 32
address, socket 5
AddUserNote 58
administration server 256
AF_INET socket domain
definition 4
AF_IUCV socket domain
definition 4
AF_UNIX socket domain
definition 4
aliases information file, specifying 32
APITYPE=3 (multiple request) 122
applications program interface (API)
IUCV sockets API 119
APITYPE environment variable 32
ASCII to EBCDIC translation tables,
specifying 33
associate table functions 243
asynchronous communication, sequence
(Pascal API) 43
auth_destroy() 156
authenticators 254
communicating 254
name structures 253
tickets 254
authentication server 253, 255
authnone_create() 156
authunix_create_default() 157
authunix_create() 156
callrpc() 157
clnt_broadcast() 158
clnt_call() 159
clnt_destroy() 161
clnt_freeres() 161
clnt_geterr() 162
clnt_pcreateerror() 162
clnt_pererror() 163
clnt_pererror() 163
clnt_spcreateerror() 164
clnt_spererror() 164
clnt_spererror() 164
cmntp_create() 161
clntraw_create() 165
clnudp_create() 166
clntcp_create() 165
clnt_create() 166
command exit, SMTP 334
compiling and linking
SNMP DPI 278
X Windows 204
CONNECT (IUCV) 128
connection information record
(Pascal) 45
connection states (Pascal) 44
CONNECTclosing (Pascal) 44
CONNECTIONstateCHANGED
(VMCF) 112
CreateTimer 59
CREDENTIALS structure 259
data structures
Pascal 44
VMCF 95
DATAdelivered (VMCF) 112
datagram sockets 5
DestroyTimer 59
directories
Kerberos 257
DPI client program 287, 289
dating
HELO 310
HELP 315
host byte order 6
host information file, specifying 33
Host lokkup routines 57
HOSTALIASES environment variable 32
HOSTS ADDRINFO file, replacing 32
HOSTS SITEINFO file, replacing 33
E
EBCDIC to ASCII translation tables,
specifying 33
EHLO 311
EndTcpIp (Pascal) 59
envelope, SMTP
description 309
example 319
environment variables
APITYPE 32
HOSTALIASES 32
X_ADDR 32
X_SITE 32
X_XLATE 33
ETC SERVICES file 365
exit routines, SMTP 323, 340
Exits, Server
Telnet 341
EXPN 318
extension routines (X window
system) 242
eXternal Data Representation protocol,
general information 149
F
FCNTL (IUCV) 128
fDPIparse() 280
file specification record (Pascal) 53
ForeignSocket 46
G
get_myaddress() 167
GET-NEXT, SNMP DPI request 277
GET, SNMP DPI request 277
GETCLIENTID (IUCV) 129
GETHOSTID (IUCV) 130
GEThostname (REXX) 130
GetHostNumber 60
GetHostResol 60
GetHostString 61
GetIdentity 61
GetNextNote 61
GETPEERNAME (IUCV) 131
GetSmss 62
GETSOCKNAME (IUCV) 132
GETSOCKOPT (IUCV) 132
GIVESOCKET (IUCV) 133
H
Handle (Pascal) 62
handling external interrupts 55
HELO 310
HELP 315
host byte order 6
host information file, specifying 33
Host lokkup routines 57
HOSTALIASES environment variable 32
HOSTS ADDRINFO file, replacing 32
HOSTS SITEINFO file, replacing 33
I
initialization procedures, TCP/UDP
(Pascal) 54
inter-communication vehicle sockets 117
IOCTL (IUCV) 134
IPUSER variable, returned by socket
call 121
IsLocalAddress 63
IsLocalHost 63
IUCV socket API 119

© Copyright IBM Corp. 1987, 2002
IUCV socket call, buffer formats
ACCEPT 126
BIND 126
CANCEL 127
CLOSE 127
CONNECT 128
FCNTL 129
GETCLIENTID 130
GETHOSTID 130
GETHOSTNAME 131
GETPEERNAME 131
GETSOCKNAME 132
GETSOCKOPT 133
GIVESOCKET 134
IOCTL 135
LISTEN 137
READ 138
READV 138
RECV 139
RECVMSG 139
RRCVFROM 139
SELECT 140
SELECT and SELECTEX descriptor sets 139
DESCRIPTOR_SET macro 139
FD_CLR macro 139
FD_ISSET macro 139
SEND 141
SENDTO 143
SHUTDOWN 145
SOCKET 145
TAKESOCKET 146
WRITE 147
WRITEV 147

IUCV socket calls (continued)
WRITEV 147
IUCV sockets, general
connect parameters 119
general information 117
issuing socket calls 122
lasterrno special request 148
multiple-req socket program (apitype=3) 119, 122
path severance 121
response from initial message 120
response from TCPIP 123
restrictions 117
send parameters, initial message 119
sever, application initiated 121
sever, clean_up of stream sockets 121
sever, TCPIP initiated 121
socket API 119
socket call syntax 124
waiting for response from TCPIP 123
IUCV Sockets, prerequisite knowledge 117
IUCV, subsystem communication macros
IUCV CONNECT 119
IUCV PURGE 123
IUCV REJECT 123, 148
IUCV REPLY 123
IUCV SEND 119
IUCVMCOM SEVER 122

Kerberos
administration server 256
applications library 257
authentication server 254
authoritizers 254
communicating name structures 253
tickets 254
authentication system database 256
cryptography 255
Kerberos routines
krb_get_cred() 259
krb_kntoln() 259
krb_mk_err() 260
krb_mk_priv() 260
krb_mk_reply() 261
krb_mk_safe() 261
krb_rd_err() 262
krb_rd_reply() 263
krb_rd_reply() 264
krb_rd_safe() 265
krb_recvauth() 265
krb_sendauth() 266
Quick Reference routines 258
sample programs
client 268
server 270
ticket granting service 255
user programs 258

L
libraries
Kerberos 256, 257
remote procedure calls 27, 30
SNMP DPI 279
sockets 1
LISTEN (IUCV) 136
LISTENING (Pascal) 44
little-endian byte ordering convention 6

M
mail forwarding exit, SMTP 329
MAILFROM 312
Management Information Base (MIB) 275, 277
MAXDESC (IUCV) 137
messages
Pascal 52
mkDPRegister() 281
mkDPResponse() 281
mkDPSet() 282
mkDPTrap() 283
MonCommand 64
Monitor procedures 56
monitor query 65
MonQuery 65
MSG_DAT fields 262, 263, 265, 267

N
names
Kerberos 253, 254
network byte order 6
NONEXISTENT (Pascal) 44
NOOP 314
notification record (Pascal) 46
notifications
notifications (Pascal) 53
notifications (VMCF) 111
notifications, specifying those to receive (VMCF) 101
NotifyIo 66

O
OPEN (Pascal) 44
OpenAttemptTimeout 45
OSF/Motif 201, 246, 247

P
parse 280
Pascal
API, description 43
assembler calls
RTcpExtRupt 71
RTcpVmcRupt 71
asynchronous communication, general sequence 43
Compiler, IBM VS Pascal & Library 43
connection state type
CONNECTIONClosing 44
LISTENING 44
remote procedure calls (RPCs) (continued)
  xdr_callhdr() 181
  xdr_callmsg() 181
  xdr_double() 182
  xdr_enum() 182
  xdr_float() 183
  xdr_inline() 184
  xdr_long() 184
  xdr_opaque_auth() 185
  xdr_opaque() 185
  xdr_pmap() 186
  xdr_pmaplist() 186
  xdr_pointer() 186
  xdr_reference() 187
  xdr_rejected_reply() 187
  xdr_replymsg() 188
  xdr_short() 188
  xdr_string() 189
  xdr_u_int() 189
  xdr_u_long() 189
  xdr_u_short() 190
  xdr_vector() 191
  xdr_void() 191
  xdr_wrapstring() 192
  xdrmem_create() 192
  xdrrec_create() 192
  xdrrec_endofrecord() 193
  xdrrec_eof() 193
  xdrrec_skiprecord() 194
  xdrstdio_create() 194
  xprt_register() 194
  xprt_unregister() 195
RESOURCES available (VMCF) 115

return codes
  Pascal 357, 361
  RPC sample programs
    client 196
    raw data stream 198
    server 197
  rpc_createerr 155
  RPCGEN command 152
  RSET 314

S

SayCalRe  72
SayConSt  72
SayIntAd  46, 72
SayIntNum  73
SayNotEn  73
SayPorTy  73
SayProTy  73
SELECT (IUCV)  139
SELECTEX (IUCV)  139
SEND (IUCV)  141
SENDINGonly  45
SENDMSG (IUCV)  142
SENDTO (IUCV)  142
server
  Kerberos  253, 270, 275
  NCS  256
  remote procedure calls  151, 196, 197, 201
  sockets  38
SERVICES file  365
  SET, SNMP DPI request  277
  SETSOCKOPT (IUCV)  143
  SetTimer  74
  SHUTDOWN (IUCV)  144
  SMMSG command (VMCF)  62
  SMTP exit routines  323, 340
  SMTP interface
    batch command files, format  321
    batch examples
      converting to batch format  321
      querying delivery queues  323
      sending mail  322
    envelope, description of  319
    path addresses  321
    responses  320
  SMTP commands
    DATA  313
    EHSO  311
    EXPN  318
    HELO  310
    HELP  315
    MAILFROM  312
    NOOP  314
    QUEU  315
    QUIT  314
    RCPT TO  313
    RSET  314
    TICK  319
    VERB  318
    VRFY  317
  SMTP transactions  309
  SNMP agent distributed program interface (DPI)  275, 287
  SNMP DPI
    agents  275
    compiling and linking  278
    requests
      GET  277
      GET-NEXT  277
      REGISTER  278
      SET  277
      TRAP  278
  routines
    DPIdebug()  279
    fDPIparse()  280
    mkDPIlist()  280
    mkDPIregister()  281
    mkDPIresponse()  281
    mkDPIset()  282
    mkDPItrap()  283
    mkDPItrape()  284
    pDPIpacket()  285
    query_DPI_port()  286
    Quick Reference  279
  software requirements  278
subagents  275
  SOCKET (IUCV)  145
  socket record  46
  sockets, C
    address  5
    address families  4
    addressing
      AF_INET domain  6
      AF_IUCV domain  8
      AF_UNIX domain  7
  sockets, C (continued)
    AF_INET domain
      addressing  6
      client perspective  11
      definition  4
      server perspective  9
    TCP client program example  36
    TCP server program example  38
    UDP client program example  40
    UDP server program example  41
  AF_IUCV domain
    addressing  8
    definition  4
  AF_UNIX domain
    addressing  7
    definition  4
  API
  compiling and linking a sockets program
    VM TCP/IP C sockets program  29
    z/VM C sockets program  27
  conversation, client/server
    client perspective for AF_INET  11
    server perspective for AF_INET  9
  TCP socket session, typical  11
  UDP socket session, typical  12
  definition  3
  environment variables  31
  example programs
    TCP client  36
    TCP server  38
    UDP client  40
    UDP server  41
  header files  19
  incompatibilities
    with Berkeley socket implementation  26
    with OS/390 C sockets implementation  26
    with VM TCP/IP C sockets implementation  23
  internetworking overview  2
  multithreading  20
  network application example  13
  POSIX signals  21
  quick reference  35
  running a sockets program
    BFS, residing in  33
    environment variables, using  31
    minidisk or SFS directory, residing on  33
    preparing for  31
  transport protocols  2
  types
    datagram  5
    guidelines for using  5
    raw  5
    stream  4
  z/VM implementation, details of
    header files  19
    incompatibilities with Berkeley sockets  26
    incompatibilities with OS/390 C sockets  26
    incompatibilities with VM TCP/IP C sockets  23
TCP/IP initialization and termination procedures (VMCF) (continued)

obtain current status of TCP connection 106
obtain status information from
TCP/IP 110
open a UDP port 107
open TCP connection 102
receive raw IP packets of a given protocol 109
receive TCP data with FRECEIVEtcp function 104
receive TCP data with RECEIVEtcp function 105
receive UDP data 107
send an ICMP echo request 111
send raw IP packets 108
send TCP data 103
send UDP data 107
specifying the notifications to receive 101
tell TCP/IP that your program will no longer use a particular IP protocol 109
tell TCP/IP that your program will use a particular IP protocol 108
TCP/UDP initialization procedures (Pascal) 54
TCP/UDP termination procedure (Pascal) 54
TCP/UDP/IP API (Pascal) 43
connection information record 45
connection state 44
data structures 44
file specification record 53
handling external interrupts 55
notification record 46
notifications 53
socket record 46
software requirements 43
using procedure calls 53
TcpAbort (Pascal) 75
TcpClose (Pascal) 75
TcpExtRupt 76
TcpFReceive (Pascal) 76
TcpFSend (Pascal) 79
TCP/IP ATCPPSRC file (Pascal) 43
TCPLOAD
EXEC 355
using 355
tcpNameChange 81
tcpOpen (Pascal) 52, 82
tcpOption (Pascal) 84
tcpReceive (Pascal) 76
tcpSend (Pascal) 79
tcpStatus (Pascal) 85
tcpVmcrRupt 86
tcpWaitOpen (Pascal) 52, 82
tcpWaitReceive 76
tcpWaitSend 79
Textlib (TXTLIB) Files
CLIB 355
CMSLIB 355
COMMTXT 355
GLOBAL 355
IBMLIB 355
PASCAL 355

Textlib (TXTLIB) Files (continued)
RPCLIB 355
SCEELKED 355
TCPASCAL 355
TCPLANG 355
TICK 319
ticket-granting server 255
tickets 254, 255
timer routines 57
transactions, SMTP 309
translation information file, specifying 33
TRAP, SNMP DPI request 278
TRYINGtoOPEN (Pascal) 45

UDP communication procedure 56, 87
UDP socket session, typical 12
UdpClose (Pascal) 86
UDPDatagramDELIVERED (VMCF) 52, 113
UDPDatagramSPACEavailable (VMCF) 114
UdpNReceive 87
UdpReceive (Pascal) 52, 88
UDPPresourcesAVAILABLE (VMCF) 115
UdpSend (Pascal) 89
Unhandle (Pascal) 90
UnNotifyIo 90
UnpackedBytes 46
URGENTpending (VMCF) 113
user exit routines, SMTP 323, 340

variables, environment 31
VERB 318
Virtual Machine Communication Facility (VMCF) Interface
CALLCODE notifications
ACTIVEprobe 116
BUFFERspaceAVAILABLE 112
CONNECTIONstateCHANGED 112
DATAdelivered 112
DUMMYProbe 116
PINCResponse 115
RAWIPpacketsDELIVERED 114
RAWIPspaceAVAILABLE 115
RESOURCESavailable 115
UDPDatagramDELIVERED 113
UDPDatagramSPACEavailable 114
UDPPresourcesAVAILABLE 115
URGENTpending 113
CALLCODE system queries
IHostLOCAL 109
MONITORcommand 110
MONITORquery 110
PINCreq 111
functions 97
general information
data structures 95, 102
use of VMCF interrupt header fields 96
use of VMCF parameter list fields 96

Index 407
Virtual Machine Communication Facility (VMCF) Interface (continued)

IP CALLCODE requests
- CLOSEerawip 109
- OPENerawip 108
- RECEIVEerawip 109
- SENDerawip 108

TCP CALLCODE requests
- ABORTtcp 106
- CLOSEtcp 105
- FRECEIVEtcp 104
- FSENDtcp 103
- OPENtcp 102
- OPTIONtcp 106
- RECEIVEtcp 105
- SENDtcp 103
- STATUStcp 106

TCP/IP initialization and termination procedures
- abort a TCP connection 106
- begin TCP/IP service 100
- close a TCP connection 105
- close a UDP port 108
- determine whether an address is local 109
- end TCP/IP service 101
- instruct TCPIP to obey a file of commands 110
- obtain current status of TCP connection 106
- obtain status information from TCPIP 110
- open a UDP port 107
- open TCP connection 102
- receive raw IP packets of a given protocol 109
- receive TCP data with FRECEIVEtcp function 104
- receive TCP data with RECEIVEtcp function 105
- receive UDP data 107
- send an ICMP echo request 111
- send raw IP packets 108
- send TCP data 103
- send UDP data 107
- specifying the notifications to receive 101
- tell TCPIP that your program will no longer use a particular IP protocol 109
- tell TCPIP that your program will use a particular IP protocol 108
- TCP/UDP/IP initialization and termination procedures
- BEGINtcpIPservice 100
- ENDtcpIPservice 101
- HANDLEnotice 101

TCPIP communication CALLCODE requests
- 99

UDP CALLCODE requests
- CLOSEudp 108
- NRECEIVEudp 107
- OPENudp 107
- SENDudp 107
- when to use 95

TCP/IP initialization and termination
- TCP 365
- UDP 366

calls
- changing attributes 207
- communicating with window managers 217
- controlling the screen saver 215
- creating and destroying 206
- cut and paste buffers 220
- default error handling 217
- display functions 222
- enabling and disabling synchronization 216
- handling events 216
- hosts and access control 215
- keyboard event functions 218
- keyboard settings 214
- manipulating bitmaps 221
- manipulating images 221
- manipulating properties 208
- manipulating regions 219
- manipulating windows 207
- opening and closing 206
- obtaining information 208
- query visual types 220
- resource manager 221
- setting selections 208
- window manager functions 214
- WRITE (IUCV) 147
- WRITEV (IUCV) 147

X Window Quick Reference tables
- associated table functions 226
- Athena widget support 240
- authorization routines 230
- character string sizes, querying 212
- clearing and copying areas 211
- communicating with window managers 217
- controlling the screen saver 215
- default error handling 217
- drawing lines 211
- drawing text 213
- extension routines 225
- filling areas 211
- fonts, loading and freeing 212
- handling events 216
- handling window manager functions 214
- manipulating of bitmaps 221
- color cells 209
- colormaps 209
- cursors 213
- display functions 222
- graphics contents 210
- hosts and access control 215
- images 221
- keyboard event functions 218
- manipulating of (continued)
  - keyboard settings 214
  - regions 219
- miscellaneous utility routines 227
- MIT extensions to X 225
- Pixmaps, creating and freeing 209
- querying visual types 220
- synchronization, enabling and disabling 216
- transferring images 213
- using cut and paste buffers 220
- using the resource manager windows
  - changing attributes 207
  - creating and destroying 206
  - display, opening and closing 206
  - manipulating 207
  - manipulating properties 208
  - obtaining information 208
  - properties and atoms 208
  - selections, setting 208
- X intrinsics routines 230

X window system
- application resource file 203
- application resources 245
- associate table functions 243
- authorization routines 243
- compiling and linking 204
- creating an application 204
- defining widgets 244
- EBCDIC-ASCII translation 202
- extension routines 242
- how the interface works 201
- interface 201
- MIT extensions 243
- running an application 205
- sample programs
  - Athena widget set, use of 250
  - OSF/Motif-based widget set, use of 252
  - Xlib calls, use of 249
- software requirements 201

subroutines
- changing window attributes 207
- clearing and copying areas 211
- communicating with window managers 217
- controlling the screen saver 201, 215
- creating and destroying windows 206
- creating and freeing pixmaps 209
- drawing lines 211
- drawing text 213
- enabling and disabling synchronization 216
- filling areas 211
- handling events 216
- handling window manager functions 214
- manipulating fonts 212
- manipulating bitmaps 221
- manipulating color cells 209
- manipulating colormaps 209
- manipulating cursors 213

408 z/VM: TCP/IP Programmer’s Reference
X window system (continued)
subroutines (continued)
  manipulating display
    functions 222
  manipulating graphics
    contexts 210
  manipulating hosts and access
    control 215
  manipulating images 221
  manipulating keyboard event
    functions 218
  manipulating keyboard
    settings 214
  manipulating regions 219
  manipulating window
    properties 208
  manipulating windows 207
  obtaining window
    information 208
  opening and closing a
    display 206
  properties and atoms 208
  querying character string
    sizes 212
  querying visual types 220
  setting window selections 208
  transferring images 213
  using cut and paste buffers 220
  using default error handling 217
  using the resource manager 221
  target display, identifying 204
  utility routines 243
  widget support
    Athena 246
    OSF/MOTIF based 247
    X Defaults 203
    X Window System Toolkit 243
X_ADDR environment variable 32
X_SITE environment variable 32
X_XLATE environment variable 33
xdr_accepted_reply() 179
xdr_array() 179
xdr_authunix_parms() 180
xdr_bool() 180
xdr_bytes() 180
xdr_callhdr() 181
xdr_callmsg() 181
xdr_double() 182
xdr_enum() 182
xdr_float() 183
xdr_inline() 184
xdr_int() 184
xdr_long() 184
xdrOpaque_auth() 185
xdrOpaque_auth() 185
xdr_pmap() 186
xdr_pmaplist() 186
xdr_pointer() 186
xdr_reference() 187
xdr_rejected_reply() 187
xdr_replymsg() 188
xdr_short() 188
xdr_string() 189
xdr_u_int() 189
xdr_u_long() 189
xdr_u_short() 190
xdr_union() 190
xdr_vector() 191
xdr_void() 191
xdr_wrapstring() 192
xdrmem_create() 192
xdrrec_create() 192
xdrrec_endofrecord() 193
xdrrec_eof() 193
xdrrec_skiprecord() 194
xdrstdio_create() 194
xpport_register() 194
xpport_unregister() 195
Readers’ Comments — We’d Like to Hear from You

z/VM™
TCP/IP Level 430
Programmer’s Reference
Version 4 Release 3.0

Publication No. SC24-6021-01

Overall, how satisfied are you with the information in this book?

<table>
<thead>
<tr>
<th>Very Satisfied</th>
<th>Satisfied</th>
<th>Neutral</th>
<th>Dissatisfied</th>
<th>Very Dissatisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall satisfaction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How satisfied are you that the information in this book is:

<table>
<thead>
<tr>
<th>Very Satisfied</th>
<th>Satisfied</th>
<th>Neutral</th>
<th>Dissatisfied</th>
<th>Very Dissatisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accurate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy to find</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy to understand</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Well organized</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicable to your tasks</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please tell us how we can improve this book:

Thank you for your responses. May we contact you?  □ Yes  □ No

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.

Name

Address

Company or Organization

Phone No.