

z/VM



TCP/IP Level 3A0 Programmer's Reference

Version 3 Release 1.0

z/VM



TCP/IP Level 3A0 Programmer's Reference

Version 3 Release 1.0

Note!

Before using this information and the product it supports, read the information under “Notices” on page 425.

First Edition (February 2001)

This edition applies to the IBM® Transmission Control Protocol/Internet Protocol Feature for z/VM (TCP/IP Level 3A0), program number 5654-A17 and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC24-5849-01.

© Copyright International Business Machines Corporation 1987, 2001. 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 (February 2001)	xv
IP Multicast	xv
Other Changes	xv
Second Edition for VM/ESA® (July 1999)	xv
First Edition for VM/ESA	xv

Chapter 1. General Programming Information	1
Porting Considerations	1
Accessing System Return Messages	1
Printing System Return Messages	1
Compiling and Linking C Applications	1
Textlib (TXTLIB) Files	2
TCPLOAD EXEC	2
Using TCPLOAD	3
SET LDRTBLS Command	4

Chapter 2. C Sockets Application Program Interface	5
Programming with C Sockets	5
Socket Programming Concepts	5
Guidelines for Using Socket Types	6
Addressing within Sockets	6
Main Socket Calls	8
C Socket Library	16
Porting	17
Environment Variables Used by the Sockets Library	17
C Socket Reference	19
C Socket Calls	21
accept()	22
bind()	23
close()	27
connect()	27
endhostent()	30
endnetent()	31
endprotoent()	31
endservent()	31
fcntl()	31
getclientid()	32
getdtablesize()	33
gethostbyaddr()	33

gethostbyname()	34
gethostent()	35
gethostid()	36
gethostname()	36
getibmssockopt()	36
getnetbyaddr()	38
getnetbyname()	39
getnetent()	40
getpeername()	40
getprotobyname()	41
getprotobynumber()	41
getprotoent()	42
getservbyname()	43
getservbyport()	43
getservent()	44
getsockname()	44
getsockopt()	45
givesocket()	49
htonl()	51
htons()	51
ibmsflush()	51
inet_addr()	52
inet_naof()	52
inet_makeaddr()	53
inet_netof()	53
inet_network()	53
inet_ntoa()	54
ioctl()	54
listen()	56
maxdesc()	57
ntohl()	58
ntohs()	59
read()	59
readv()	60
recv()	61
recvfrom()	62
recvmsg()	63
select()	64
selectex()	67
send()	68
sendmsg()	69
sendto()	70
sethostent()	72
setibmssockopt()	72
setnetent()	75
setprotoent()	75
setservent()	75
setsockopt()	76
sockdb_sock_debug()	79
sock_debug_bulk_perfo ()	80
sock_do_bulkmode()	80
sock_do_teststor()	81
shutdown()	81
socket()	82
takesocket()	85
tcperror()	86

write()	87
writew()	88
Sample C Socket Programs	89
C Socket TCP Client	89
C Socket TCP Server	90
C Socket UDP Server	92
C Socket UDP Client	93

Chapter 3. TCP/UDP/IP API (Pascal Language) 95

Software Requirements	95
Data Structures	96
Connection State.	96
Connection Information Record.	97
Socket Record	98
Notification Record.	98
File Specification Record.	105
Using Procedure Calls	105
Notifications.	105
TCP/UDP Initialization Procedures	106
TCP/UDP Termination Procedure	106
Handling External Interrupts	107
TCP Communication Procedures	107
Ping Interface	107
Monitor Procedures	108
UDP Communication Procedures.	108
Raw IP Interface	108
Timer Routines	109
Host Lookup Routines	109
AddUserNote	110
Other Routines	110
Procedure Calls.	110
BeginTcpIp	110
ClearTimer	111
CreateTimer	111
DestroyTimer	111
EndTcpIp	111
GetHostNumber	112
GetHostResol	112
GetHostString	113
GetIdentity	113
GetNextNote	113
GetSmsg	114
Handle	114
LocalAddress	115
IsLocalHost	115
MonCommand	116
MonQuery	117
NotifyIo	118
PingRequest	119
RawIpClose	119
RawIpOpen	120
RawIpReceive	120
RawIpSend	121
ReadXlateTable	122
RTcpExtRupt	123
RTcpVmcfRupt	123
SayCalRe	124
SayConSt	124
SayIntAd	124
SayIntNum	125

SayNotEn	125
SayPorTy	125
SayProTy	125
SetTimer	126
StartTcpNotice	126
TcpAbort	127
TcpClose	127
TcpExtRupt	128
TcpFReceive, TcpReceive, and TcpWaitReceive	128
TcpFSend, TcpSend, and TcpWaitSend	131
TcpNameChange	133
TcpOpen and TcpWaitOpen.	134
TcpOption	136
TcpStatus	137
TcpVmcfRupt	138
UdpClose	138
UdpNReceive	139
UdpOpen	139
UdpReceive	140
UdpSend	141
Unhandle.	142
UnNotifyIo	142
Sample Pascal Program	143

Chapter 4. Virtual Machine Communication Facility Interface . . . 147

General Information	147
Data Structures	147
VMCF Functions	149
VMCF TCPIP Communication CALLCODE Requests	150
VMCF TCPIP Communication CALLCODE Notifications.	151
TCP/UDP/IP Initialization and Termination Procedures	152
BEGINtcpIPservice	152
HANDLEnotice	153
ENDtcpIPservice	153
TCP CALLCODE Requests	154
OPENTcp	154
SENDDtcp and FSENDDtcp	155
FRECEIVtcp	156
RECEIVtcp.	157
CLOSEtcp	157
ABORTtcp	158
STATUStcp	158
OPTIONtcp	158
UDP CALLCODE Requests.	158
OPENudp	159
SENDudp	159
NRECEIVEudp.	159
CLOSEudp	160
IP CALLCODE Requests	160
OPENrawip	160
SENDrawip	160
RECEIVERawip.	161
CLOSErawip	161
CALLCODE System Queries	161
IShostLOCAL	161
MONITORcommand	162
MONITORquery	162

PINGreq	163
CALLCODE Notifications	163
BUFFERspaceAVAILABLE	164
CONNECTIONstateCHANGED	164
DATAdelivered.	164
URGENTpending	165
UDPdatagramDELIVERED	165
UDPdatagramSPACEavailable	166
RAWIPpacketsDELIVERED.	166
RAWIPspaceAVAILABLE	167
RESOURCESavailable	167
UDPResourcesAVAILABLE	167
PINGresponse	167
DUMMYprobe	168
ACTIVEprobe	168

Chapter 5. Inter-User Communication

Vehicle Sockets 169

Prerequisite Knowledge	169
Available Functions	169
Socket Programming with IUCV	170
Preparing to use the IUCV Socket API	171
Establishing an IUCV connection to TCP/IP	171
Initializing the IUCV Connection	171
Severing the IUCV Connection	173
Sever by the Application	173
Sever by TCP/IP	173
Issuing Socket Calls	174
Overlapping Socket Requests	174
TCP/IP Response to an IUCV Request	175
Cancelling a Socket Request	175
IUCV Socket Call Syntax	176
IUCV Socket Calls.	177
ACCEPT	177
BIND	178
CANCEL.	178
CLOSE	179
CONNECT	180
FCNTL	180
GETCLIENTID	181
GETHOSTID	182
GETHOSTNAME	182
GETPEERNAME	183
GETSOCKNAME	183
GETSOCKOPT	184
GIVESOCKET	185
IOCTL.	186
LISTEN	188
MAXDESC	189
READ, READV.	189
RECV, RECVFROM, RECVMSG	190
SELECT, SELECTEX	191
SEND	193
SENDMSG	194
SENDTO	194
SETSOCKOPT	195
SHUTDOWN	196
SOCKET	197
TAKESOCKET	198
WRITE, WRITEV	199
LASTERRNO	200

Chapter 6. Remote Procedure Calls 201

The RPC Interface	201
Portmapper	203
Contacting Portmapper	204
Target Assistance	204
RPCGEN Command	204
enum clnt_stat Structure.	205
Remote Procedure Call Library	206
Porting	206
Remapping C Identifiers with RPC.H	206
Accessing System Return Messages	206
Printing System Return Messages	207
Enumerations	207
RPC Global Variables.	207
rpc_createerr	207
svc_fds	207
Remote Procedure and eXternal Data Representation Calls	207
auth_destroy()	207
authnone_create()	208
authunix_create()	208
authunix_create_default()	208
callrpc()	209
clnt_broadcast()	209
clnt_call()	211
clnt_control()	211
clnt_create()	212
clnt_destroy()	213
clnt_freeres()	213
clnt_geterr()	213
clnt_pcreateerror()	214
clnt_perrno()	214
clnt_perror()	214
clnt_screateerror()	215
clnt_sperrno()	215
clnt_sperror()	216
clntraw_create()	216
clnttcp_create()	217
clntudp_create()	217
get_myaddress()	218
getrpcport()	219
pmap_getmaps()	219
pmap_getport()	219
pmap_rmtcall()	220
pmap_set()	221
pmap_unset()	221
registerrpc()	222
svc_destroy()	222
svc_freeargs()	223
svc_getargs()	223
svc_getcaller()	224
svc_getreq()	224
svc_register()	224
svc_run()	225
svc_sendreply()	225
svc_unregister()	226
svcerr_auth()	226
svcerr_decode()	226
svcerr_noproc()	226
svcerr_noprogram()	227
svcerr_progvers()	227

svcerr_systemerr()	227
svcerr_weakauth()	228
svccraw_create()	228
svctcp_create()	228
svcudp_create()	229
xdr_accepted_reply()	229
xdr_array()	230
xdr_authunix_parms()	230
xdr_bool()	231
xdr_bytes()	231
xdr_callhdr()	232
xdr_callmsg()	232
xdr_double()	232
xdr_enum()	233
xdr_float()	234
xdr_inline()	234
xdr_int()	235
xdr_long()	235
xdr_opaque()	235
xdr_opaque_auth()	236
xdr_pmap()	236
xdr_pmaplist()	237
xdr_pointer()	237
xdr_reference()	237
xdr_rejected_reply()	238
xdr_replymsg()	238
xdr_short()	239
xdr_string()	239
xdr_u_int()	239
xdr_u_long()	240
xdr_u_short()	240
xdr_union()	241
xdr_vector()	241
xdr_void()	242
xdr_wrapstring()	242
xdrmem_create()	243
xdrrec_create()	243
xdrrec_endofrecord()	244
xdrrec_eof()	244
xdrrec_skiprecord()	244
xdrstdio_create()	244
xprt_register()	245
xprt_unregister()	245
Sample RPC Programs	246
RPC Client	246
RPC Server	246
RPC Raw Data Stream	248

Chapter 7. X Window System Interface 251

What Is Provided	251
Software Requirements	251
Using the X Window System Interface in the VM Environment	251
Application Resource File	253
Identifying the Target Display	254
Creating an Application	254
Generating X-Window System Applications	254
X Window System Subroutines	256
Opening and Closing a Display	256
Creating and Destroying Windows	256
Manipulating Windows	257

Changing Window Attributes	257
Obtaining Window Information	258
Obtaining Properties and Atoms	258
Manipulating Window Properties	258
Setting Window Selections	258
Manipulating Colormaps	259
Manipulating Color Cells	259
Creating and Freeing Pixmaps	259
Manipulating Graphics Contexts	260
Clearing and Copying Areas	261
Drawing Lines	261
Filling Areas	261
Loading and Freeing Fonts	262
Querying Character String Sizes	262
Drawing Text	263
Transferring Images	263
Manipulating Cursors	263
Handling Window Manager Functions	264
Manipulating Keyboard Settings	264
Controlling the Screen Saver	265
Manipulating Hosts and Access Control	265
Handling Events	266
Enabling and Disabling Synchronization	266
Using Default Error Handling	267
Communicating with Window Managers	267
Manipulating Keyboard Event Functions	268
Manipulating Regions	269
Using Cut and Paste Buffers	270
Querying Visual Types	270
Manipulating Images	271
Manipulating Bitmaps	271
Using the Resource Manager	271
Manipulating Display Functions	272
Extension Routines	275
MIT Extensions to X	275
Associate Table Functions	276
Miscellaneous Utility Routines	277
X Authorization Routines	280
X Intrinsic Routines	280
Athena Widget Support	290
Extension Routines	292
MIT Extensions to X	293
Associate Table Functions	293
Miscellaneous Utility Routines	293
X Authorization Routines	293
X Window System Toolkit	293
Application Resources	295
Athena Widget Set	296
OSF/Motif-Based Widget Support	297
Sample X Window System Applications	299
Xlib Sample Program	299
Athena Widget Sample Program	300
OSF/Motif-Based Widget Sample Program	302

Chapter 8. Kerberos Authentication System 303

Authentication Server	303
Name Structures	303
Tickets and Authenticators	304
Communicating with the Authentication Server	304
Ticket-Granting Server	305

Accessing a Service	305
Kerberos Database	306
Administration Server	306
Kerberos C Language Applications Library	307
Kerberos Routines Reference	308
Client Commands	308
Applications	309
Kerberos Routines	309
krb_get_cred()	309
krb_kntoln()	309
krb_mk_err()	310
krb_mk_priv()	310
krb_mk_req()	311
krb_mk_safe()	311
krb_rd_err()	312
krb_rd_priv()	313
krb_rd_req()	314
krb_rd_safe()	315
krb_rcvauth()	315
krb_sendauth()	316
Sample Kerberos Programs	318
Kerberos Client	318
Kerberos Server	320

Chapter 9. SNMP Agent Distributed Program Interface 325

SNMP Agents and Subagents	325
Processing DPI Requests	326
Processing a GET Request	327
Processing a SET Request	327
Processing a GET_NEXT Request	327
Processing a REGISTER Request	328
Processing a TRAP Request	328
Compiling and Linking	328
SNMP DPI Reference	329
DPI Library Routines	329
DPIdebug()	329
fDPIparse()	330
mkDPIlist()	330
mkDPIregister()	331
mkDPIresponse()	331
mkDPIset()	332
mkDPItrap()	333
mkDPItrape()	334
Example of an Extended Trap	334
pDPIpacket()	335
query_DPI_port()	336
Sample SNMP DPI Client Program	337
The DPISAMPLE Program (Sample DPI Subagent)	337
DPISAMPLE TABLE	339
Client Sample Program	339
Compiling and Linking the DPISAMPLE.C	
Source Code	358

Chapter 10. SMTP Virtual Machine Interfaces 359

SMTP Transactions	359
SMTP Commands	360
HELO	360
EHLO	361

MAIL FROM	361
RCPT TO	362
DATA	363
RSET	364
QUIT	364
NOOP	364
HELP	364
QUEUE	365
VERFY	367
EXPN	368
VERB	368
TICK	369
SMTP Command Example	369
SMTP Command Responses	369
Path Address Modifications	370
Batch SMTP Command Files	371
Batch SMTP Examples	371
Sending Mail to a TCP Network Recipient	371
Querying SMTP Delivery Queues	372
SMTP Exit Routines	373
Client Verification Exit	373
Built-in Client Verification Function	373
Client Verification Exit Parameter Lists	374
Using the Mail Forwarding Exit	378
Mail Forwarding Exit Parameter Lists	379
Using the SMTP Command Exit	384
SMTP Command Exit Parameter Lists	385

Chapter 11. Telnet Exits 391

Telnet Session Connection Exit	391
Telnet Exit Parameter List	392
Sample Exit	392
Telnet Printer Management Exit	393
Telnet Printer Management Exit Parameter List	393
Sample Exit	393

Chapter 12. FTP Server Exit 395

The FTP Server Exit	395
Sample Exit	395
Audit Processing	395
Audit Processing Parameter List	396
Audit Processing Parameter Descriptions	396
Return Codes from Audit Processing	398
General Command Processing	398
General Command Processing Parameter List	398
General Command Processing Parameter	
Descriptions	399
Return Codes from General Command	
Processing	400
Change Directory Processing	401
Change Directory Processing Parameter List	401

Appendix A. Pascal Return Codes . . 405
Explanatory Notes 407

Appendix B. C API System Return Codes 409

Appendix C. Well-Known Port Assignments	413	Installation and Service	447
TCP Well-Known Port Assignments	413	Planning and Administration	447
UDP Well-Known Port Assignments	414	Customization	447
Appendix D. Related Protocol Specifications	417	Operation	447
Appendix E. Abbreviations and Acronyms	421	Application Programming	447
Notices	425	End Use	448
Trademarks	427	Diagnosis.	448
Glossary	429	Publications for Additional Facilities.	448
Bibliography.	447	DFSMS/VM [®]	448
z/VM Base Publications	447	OSA/SF	448
Evaluation	447	Language Environment [®]	448
		Publications for Optional Features	448
		CMS Utilities Feature.	448
		TCP/IP Feature for z/VM	448
		OpenEdition [®] DCE Feature for VM/ESA [®]	448
		LANRES/VM	449
		CD-ROM.	449
		Other TCP/IP Related Publications	449
		Index	451

Preface

TCP/IP for VM: Programmer's Reference describes the routines for application programming in IBM* Transmission Control Protocol/Internet Protocol Version 3 Release 1.0 for VM (TCP/IP Level 3A0 for VM).

This book contains reference 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 VM/ESA 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 3A0 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 3A0 for VM.

"Chapter 1. General Programming Information" on page 1, provides information for programmers who use the application program interfaces described in this book.

“Chapter 2. C Sockets Application Program Interface” on page 5, describes the C Socket application program interface provided with TCP/IP Level 3A0 for VM.

“Chapter 3. TCP/UDP/IP API (Pascal Language)” on page 95, 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 4. Virtual Machine Communication Facility Interface” on page 147, describes how to communicate directly with the TCPIP virtual machine using Virtual Machine Communication Facility calls.

“Chapter 5. Inter-User Communication Vehicle Sockets” on page 169, 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 6. Remote Procedure Calls” on page 201, describes the Remote Procedure Call protocol, which permits remote execution of subroutines across a TCP/IP network.

“Chapter 7. X Window System Interface” on page 251, describes the X Window System and subroutines for TCP/IP Level 3A0 for VM.

“Chapter 8. Kerberos Authentication System” on page 303, describes the Kerberos Authentication System and the routines for writing authentication programs.

“Chapter 9. SNMP Agent Distributed Program Interface” on page 325, describes the Simple Network Management Protocol (SNMP) agent distributed program interface (DPI).

“Chapter 10. SMTP Virtual Machine Interfaces” on page 359, describes the communication interfaces to the SMTP virtual machine.

“Chapter 11. Telnet Exits” on page 391, 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 12. FTP Server Exit” on page 395, describes the FTP Server Exit

“Appendix A. Pascal Return Codes” on page 405, lists the system return codes as they apply to Pascal calls and provides their numeric value and description.

“Appendix B. C API System Return Codes” on page 409, lists the system return codes and provides their numeric value and a description.

“Appendix C. Well-Known Port Assignments” on page 413, 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 D. Related Protocol Specifications” on page 417, lists the related protocol specifications concerning TCP/IP Level 3A0 for VM.

“Appendix E. Abbreviations and Acronyms” on page 421, lists and defined the abbreviations and acronyms used in this book.

This book also includes a glossary, a bibliography, and an index.

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 E. Abbreviations and Acronyms” on page 421, lists the abbreviations and acronyms that are used throughout this book.

The “Glossary” on page 429, defines terms used throughout this book that are associated with TCP/IP communication in an internet environment.

For more information about related publications, see “Bibliography” on page 447.

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

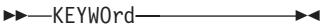





This section describes how to read the syntax diagrams in this book.

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:

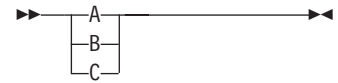
- Directly on the line (required)
- Above the line (default)
- Below the line (optional).

Syntax Diagram Description	Example
<p>Abbreviations:</p> <p>Uppercase letters denote the shortest acceptable abbreviation. If an item appears entirely in uppercase letters, it cannot be abbreviated.</p> <p>You can type the item in uppercase letters, lowercase letters, or any combination.</p> <p>In this example, you can enter KEYWO, KEYWOR, or KEYWORD in any combination of uppercase and lowercase letters.</p>	
<p>Symbols:</p> <p>You must code these symbols exactly as they appear in the syntax diagram.</p>	<ul style="list-style-type: none"> * Asterisk : , = Equal Sign - Hyphen () Parentheses .
<p>Variables:</p> <p>Highlighted lowercase items (<i>like this</i>) denote variables.</p> <p>In this example, <i>var_name</i> represents a variable you must specify when you code the KEYWORD command.</p>	 
<p>Repetition:</p> <p>An arrow returning to the left means that the item can be repeated.</p> <p>A character within the arrow means you must separate repeated items with that character.</p> <p>A footnote (1) by the arrow references a limit that tells how many times the item can be repeated.</p>	  
	<p>Notes:</p> <p>1 Specify <i>repeat</i> up to 5 times.</p>

Syntax Diagram Description

Example**Required Choices:**

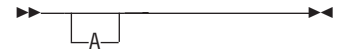
When two or more items are in a stack and one of them is on the line, you *must* specify one item.



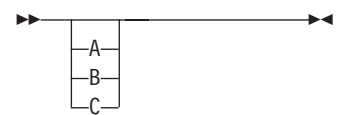
In this example, you must choose A, B, or C.

Optional Choice:

When an item is below the line, the item is optional. In this example, you can choose A or nothing at all.

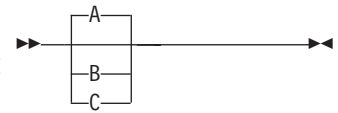


When two or more items are in a stack below the line, all of them are optional. In this example, you can choose A, B, C, or nothing at all.



Defaults:

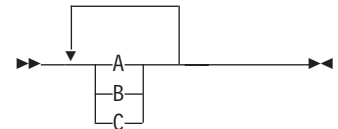
Defaults are above the line. The system uses the default unless you override it. You can override the default by coding an option from the stack below the line.



In this example, A is the default. You can override A by choosing B or C.

Repeatable Choices:

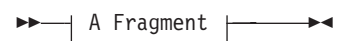
A stack of items followed by an arrow returning to the left means that you can select more than one item or, in some cases, repeat a single item.



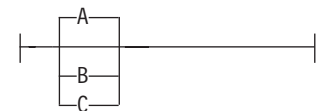
In this example, you can choose any combination of A, B, or C.

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.

**A Fragment:**

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 form number (including the suffix), and the page, section title, or topic you are commenting on.

- Visit the z/VM web site at:

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

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: pubrcf@vnet.ibm.com

IBMLink™: GDLVME(PUBRCF)

- Fill out the Readers' Comments form at the back of this book and return it using one of the following methods:
 - Mail it to the address printed on the form (no postage required in the USA).
 - Fax it to 1-607-752-2327.
 - Give it to an IBM representative.

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.

First Edition for z/VM (February 2001)

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

IP Multicast

Support for IP Multicast has been added for the C sockets application program interfaces `getsockopt` 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.

Second Edition for VM/ESA[®] (July 1999)

This edition contains updates for the General Availability of TCP/IP Function Level 320 Programmer's Reference.

- The chapter 'SMTP Virtual Machine Interfaces', which discusses the interfaces to the SMTP server, has been updated to include new support for Extended Services and the server exits.
- A new chapter with details of the telnet server session connection and printer management exits was added.
- A new chapter for the FTP Server Exit was added.

First Edition for VM/ESA

This edition contains updates for the General Availability of TCP/IP Function Level 310 Programmer's Reference.

- The DUMMYprobe and ACTIVEprobe VMCF interrupt headers were added to Chapter 4.

Chapter 1. General Programming Information

This chapter contains fundamental, technical information that programmers need to know before they attempt to work with the application program interfaces (API) provided with TCP/IP.

Porting Considerations

Important Note

In order to develop or port applications in the C programming language that interface directly to TCP, UDP, and IP, or which modify TCP/IP for VM C components, the following are required:

- IBM C for VM/ESA Compiler, Version 3 Release 1 (5654-033)
- Language Environment, which is included in the base release.

Accessing System Return Messages

To access system return values while running your C program, use only the `errno.h` include statement supplied with the compiler. To access TCP/IP network return values, add the following include statement to your C program:

```
#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 provided by IBM and included with TCP/IP Level 320 for VM.

Compiling and Linking C Applications

This section describes how to compile and link-edit C applications that use the TCP/IP C sockets

1. Access the TCP/IP client minidisk (TCPMAINT 592) ahead of the S-disk to avoid conflicts with OpenEdition[®] for VM/ESA.

Establish the C development environment:

- Access the C compiler
- SET LDRTBLS 25 (the number you need depends on your program)
- GLOBAL LOADLIB SCEERUN

Note: CMSLIB is required only when the program will run in a 370-mode virtual machine.

2. Compile your programs, ensuring that the preprocessor symbol `VM` is defined. For example:

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

With OpenEdition for VM/ESA you can also use the `c89` command and the `make` shell utility.

General Programming Information

3. Select the link libraries your applications need and put them on a GLOBAL TXTLIB command. SCEELKED and COMMTXT are the minimum required:

```
GLOBAL TXTLIB SCEELKED COMMTXT
```

Additional libraries, listed in Table 1 may be required depending on the TCP/IP functions your application uses. For example, programs that use RPC must issue:

```
GLOBAL TXTLIB SCEELKED COMMTXT RPCLIB
```

Textlib (TXTLIB) Files

Table 1. TXTLIB Files and Applications

TXTLIB File	Application
COMMTXT	C Sockets and Pascal API
RPCLIB	Remote Procedure Calls
BPLDBM	Kerberos
KDB	Kerberos
KRB	Kerberos
DES	Kerberos
X11LIB	Xlib, Xmu, Xext and Xau Routines
OLDXLIB	X Release 10 compatibility routines
XTLIB	X Intrinsic
XAWLIB	Athena Widget Set
XMLIB	OSF/Motif-based widget set
DPILIB	SNMP DPI

4. 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:

```
TCPLOAD sample@ c
```

or

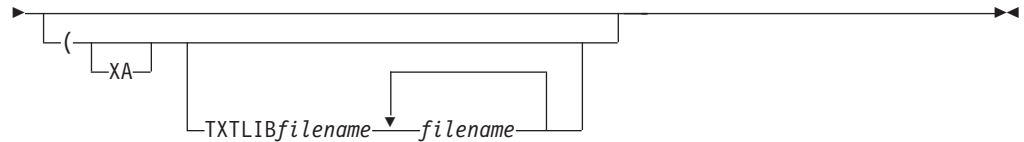
```
CMOD myprog1 myprog2 (AUTO
```

Complete information on compiling and link-editing C programs can be found in the *C VM/ESA User's Guide, SC09-2151*.

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—————▶▶
```



Parameter	Description
<i>load_list</i>	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 <i>load_list</i> specifies the name of the main object module. Subsequent lines specify additional object modules to be included in the load module.
<i>control_file</i>	Specifies the <i>control_file</i> , which determines the file types of text files according to the standard update identifier procedure.
<i>type</i>	Specifies the <i>type</i> parameter as one of the following: C Includes SCEELKED, CMSLIB, RPCLIB, TCPASCAL, TCPLANG, COMMTXT, and CLIB ttxlibs. C-ONLY Includes SCEELKED, RPCLIB, COMMTXT, CMSLIB, and CLIB ttxlib. PASCAL Includes TCPASCAL, TCPLANG, and COMMTXT ttxlibs. PASCAL is the default for the <i>type</i> 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 <i>filenames</i> that will be added to the GLOBAL TXTLIB command. See Table 1 on page 2 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.

1. Create a file with file type LOADLIST, which contains all the object (TEXT) files to be linked. For example, `l1istfn 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 l1istfn ctrlfn C (TXTLIB mylib1 mylib2
```

Where:

- `l1istfn` 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

General Programming Information

- 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.

SET LDRTBLS Command

The SET LDRTBLS command defines the number of pages of storage to be used for loader tables (LDRTBLS). By default, a virtual machine having up to 384 Kb of addressable storage has three pages of loader tables; a larger virtual machine has four pages. Each loader table page has a capacity of 169 external names. During LOAD and INCLUDE command processing, each unique, external name encountered in a TEXT deck is entered in the loader table.

▶▶—SET—LDRTBLS—*nn*—————▶▶

Parameter	Description
<i>nn</i>	Specifies the number of pages to be used for loader tables.

For more information about the SET LDRTBLS command, see the *CMS Command Reference*.

Chapter 2. C Sockets Application Program Interface

This chapter describes the C socket application program interface (API) provided with TCP/IP. Use the socket routines in your C program to interface with the TCP, UDP, and IP protocols. This allows you to communicate across networks with other programs. You can, for example, make use of socket routines when you write a client program that must communicate with a server program running on another computer.

To use the C socket API, you should know C language programming.

For more information about C sockets, see the *IBM AIX® Operating System Technical Reference: System Calls and Subroutines*.

Programming with C Sockets

The VM C socket API provides a standard interface to the transport and internetwork layer interfaces of TCP/IP. It supports three socket types: stream, datagram, and raw. Stream and datagram sockets interface to the transport layer protocols, and raw sockets interface to the network layer protocols. The programmer chooses the most appropriate interface for an application.

Socket Programming Concepts

Before programming with the C socket API, you should consider the following important concepts.

What is a Socket?

A socket is an endpoint for communication that can be named and addressed in a network. From an application program perspective, it is a resource allocated by the virtual machine. It is represented by an integer called a socket descriptor.

The socket interface was designed to provide applications with a network interface that hides the details of the physical network. The interface is differentiated by the different services that are provided. Stream, datagram, and raw sockets each define a different service available to applications.

The **stream** socket interface defines a reliable connection-oriented service. Data is sent without errors or duplication and is received in the same order as it is sent. Flow control is built in to avoid data overruns. No boundaries are imposed on the data; it is considered to be a stream of bytes. An example of an application that uses stream sockets is Telnet.

The **datagram** socket interface defines a connectionless service. Datagrams are sent as independent packets. The service provides no guarantees; data can be lost or duplicated, and datagrams can arrive out of order. The size of a datagram is limited to the size that can be sent in a single transaction (currently the default is 8192 and the maximum is 32,767). No disassembly and reassembly of packets is performed. An example of an application that uses datagram sockets is the Network File System (NFS).

C Sockets Application Program Interface

The *raw* socket interface allows direct access to lower layer protocols such as IP and Internet Control Message Protocol (ICMP). The interface is often used for testing new protocol implementations.

The socket interface can be extended; therefore, you can define new socket types to provide additional services. Because socket interfaces isolate you from the communication functions of the different protocol layers, the interfaces are largely independent of the underlying network. In the VM implementation of sockets, stream sockets interface to TCP, datagram sockets interface to UDP, and raw sockets interface to ICMP and IP. In the future, the underlying protocols can change; however, the socket interface remains the same. For example, stream sockets can eventually interface to the International Standards Organization (ISO) Open System Interconnection (OSI) transport class 4 protocol. This means that applications do not have to be rewritten as underlying protocols change. For more information about open System Interconnection, see *Open System Interconnection: Reference Summary*.

Guidelines for Using Socket Types

The following considerations help you choose the appropriate socket type for an application.

If you are communicating to an existing application, use the same protocols as the existing application. For example, if you interface to an application that uses TCP, use stream sockets. For other applications you should consider the following factors:

- **Reliability.** Stream sockets provide the most reliable connection. Datagram, or raw sockets, are unreliable, because packets can be discarded, corrupted, or duplicated during transmission. This can be acceptable if the application does not require reliability, or if the application implements the reliability on top of the sockets interface. The trade-off is the increased performance available over stream sockets.
- **Performance.** The overhead associated with reliability, flow control, packet reassembly, and connection maintenance degrade the performance of stream sockets so that they do not perform as well as datagram sockets.
- **The amount of data to be transferred.** 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, it makes more sense to use stream sockets.

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

Addressing within Sockets

The following sections describe the different ways to address within the C socket API.

Address Families

Address families define different styles of addressing. All hosts in the same addressing family understand and use the same scheme for addressing socket endpoints. TCP/IP supports two addressing families: AF_INET and AF_IUCV. The AF_INET domain defines addressing in the internet (INET) domain. The AF_IUCV domain defines addressing in the Inter-User Communication Vehicle (IUCV) domain. In the IUCV domain, virtual machines can use the socket interface to communicate with other virtual machines on the same host.

Socket Address

A socket address is defined by the *sockaddr* structure in the SOCKETS.H header file. It has two fields as shown in the following example:

```
struct sockaddr
{
    unsigned short sa_family;    /* address family */
    char    sa_data[14]; /* up to 14 bytes of direct address */
};
```

The *sa_family* field contains the addressing family. It is AF_INET for the internet 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. For more information about the internet domain, see “Addressing within an Internet Domain” and for more information about the IUCV domain, see “Addressing within the IUCV Domain” on page 8.

Internet Addresses

Internet addresses are 32-bit quantities that represent a network interface. Every internet address within an administered AF_INET domain must be unique. A common misunderstanding is that every host must have a unique internet address. In fact, a host has as many internet addresses as it has network interfaces.

Ports

A port is used to differentiate between different applications using the same network interface: it is an additional qualifier used by the system software to get data to the correct application. Physically, a port is a 16-bit integer. Some ports are reserved for particular applications and are called well-known ports. For a listing of well-known ports, see “Appendix C. Well-Known Port Assignments” on page 413.

Network Byte Order

Ports and addresses are usually specified to calls using the network byte ordering convention. Network byte order is also known as big endian byte ordering, where the high-order byte defines significance, (as compared with little endian byte ordering, where the low-order byte defines significance). Using network byte ordering for data exchanged between hosts allows hosts using different architectures to exchange address information. For examples of using the htons() call to put ports into network byte order, see Figure 2 on page 9, Figure 5 on page 10, and Figure 8 on page 12. For more information about network byte order, see “accept()” on page 22, “bind()” on page 23, “htonl()” on page 51, “htons()” on page 51, “ntohl()” on page 58, and “ntohs()” on page 59.

Note: The socket interface does not handle application data byte ordering differences. Application writers must handle byte order differences themselves or use higher-level interfaces, such as Remote Procedure Calls (RPC).

Addressing within an Internet Domain

A socket address in an internet addressing family comprises four fields: the address family (AF_INET), an internet address, a port, and a character array. The structure of an internet socket address is defined by the following *sockaddr_in* structure, which is found in the IN.H header file:

```
struct in_addr
{
    unsigned long s_addr;
};
struct sockaddr_in
{
```

C Sockets Application Program Interface

```
short    sin_family;           /* addressing family */
unsigned short sin_port;      /* port number */
struct  in_addr sin_addr;    /* internet address */
char    sin_zero[8];        /* zeros */
};
```

The *sin_family* field is set to AF_INET. The *sin_port* field is the port used by the application, in network byte order. The *sin_addr* field is the internet address of the network interface used by the application. It is also in network byte order. The *sin_zero* field should be set to all zeroes.

Addressing within the IUCV Domain

A socket address in the IUCV addressing family is comprised of six fields: the address family (AF_INET), three reserved fields, a VM user ID, and an application name. The structure of an IUCV socket address is defined by the following *sockaddr_iucv* structure, which is found in the SAIUCV.H header file.

```
struct sockaddr_iucv
{
    short        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_user_id[8]; /* user_id to connect to */
    unsigned char siucv_name[8];  /* iucvname for connect */
};
```

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_userid* field is the VM user ID and the *siucv_name* field is the application name by which the socket is to be known for the bind() or connect() calls.

Either IUCV ANY or IUCV ALLOW must be specified in the system directory for the virtual machine. This specification shows the approval of the client or server to do socket communication with another virtual machine.

Main Socket Calls

You can write a very powerful network application with less than a dozen socket calls.

1. First, an application must get a socket descriptor using the socket() call, as in the example shown in Figure 1. For a complete description, see “socket()” on page 82.

```
int socket(int domain, int type, int protocol);
:
:
int s;
:
:
s = socket(AF_INET, SOCK_STREAM, 0);
```

Figure 1. An Application Uses the socket() Call

The code fragment in Figure 1 allocates a socket descriptor *s* in the internet addressing family (AF_INET). The *domain* parameter is a constant that specifies the domain where the communication is taking place. A domain is the collection of applications using the same naming convention. VM supports two addressing families: AF_INET and AF_IUCV. The *type* parameter is a constant that specifies the type of socket, which can be SOCK_STREAM for stream sockets, SOCK_DGRAM for datagram sockets, or SOCK_RAW for raw

C Sockets Application Program Interface

sockets. In the AF_IUCV domain, the *type* must be SOCK_STREAM. The *protocol* parameter is a constant that specifies the protocol to use, and is ignored unless *type* is set to SOCK_RAW. Passing 0 chooses the default protocol. If successful, `socket()` returns a positive integer socket descriptor.

2. Once an application has a socket descriptor, it can explicitly `bind()` a unique name to the socket, as in the example shown in Figure 2. For a complete description, see “`bind()`” on page 23.

```
int bind(int s, struct sockaddr *name, int namelen);
:
int rc;
int s;
struct sockaddr_in myname;

/* clear the structure to be sure that the sin_zero field is clear */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET; /* internet addressing family */
myname.sin_addr.s_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = htons(1024); /* port number */
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

Figure 2. An Application Uses the `bind()` Call

This example binds `myname` to socket `s`. The name specifies that the application is in the internet domain (AF_INET) at internet address 129.5.24.1, and is bound to port 1024. Servers must bind a name to become accessible from the network. The example in Figure 2 shows two useful utility routines:

- `inet_addr()` takes an internet address in dotted-decimal form and returns it in network byte order. For a complete description, see “`inet_addr()`” on page 52.
- `htons()` takes a port number in host byte order and returns the port in network byte order. For a complete description, see “`htons()`” on page 51.

For another example of the `bind()` call, see Figure 3. It uses the utility routine `gethostbyname()` to find the internet address of the host, rather than using `inet_addr()` with a specific address.

```
int bind(int s, struct sockaddr_in name, int namelen);
:
int rc;
int s;
char *hostname = "myhost";
struct sockaddr_in myname;
struct hostent *hp;
hp = gethostbyname(hostname);
/*clear the structure to be sure that the sin_zero field is clear*/
memset(&myname,0,sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = *((unsignedlong *)hp->h_addr);
myname.sin_port = htons(1024);
:
rc = bind(s,(struct sockaddr *) &myname, sizeof(myname));
```

Figure 3. A `bind()` Call Uses the `gethostbyname()` Call

C Sockets Application Program Interface

3. After binding a name to a socket, a server using stream sockets must indicate its readiness to accept connections from clients. The server does this with the `listen()` call as illustrated in the example in Figure 4.

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

Figure 4. An Application Uses the listen() Call

The `listen()` call tells the TCPIP virtual machine 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. For a complete description, see “`listen()`” on page 56.

4. Clients using stream sockets initiate a connection request by calling `connect()`, as shown in the example in Figure 5.

```
int connect(int s, struct sockaddr *name, int namelen);
:
:
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(1024);
:
:
rc = connect(s, (struct sockaddr *) &servername, sizeof(servername));
```

Figure 5. An Application Uses the connect() Call

The `connect()` call attempts to connect socket `s` to the server with name `servername`. This could be the server that was used in the previous `bind()` example. The caller optionally blocks until the connection is accepted by the server. On successful return, the socket is associated with the connection to the server. For a complete description, see “`connect()`” on page 27.

5. Servers using stream sockets accept a connection request with the `accept()` call, as shown in the example shown in Figure 6.

```
int accept(int s, struct sockaddr *addr, int *addrlen);
:
:
int clientsocket;
int s;
struct sockaddr clientaddress;
int addrlen;
:
:
addrlen = sizeof(clientaddress);
:
:
clientsocket = accept(s, &clientaddress, &addrlen);
```

Figure 6. An Application Uses the accept() Call

If connection requests are not pending on socket `s`, the `accept()` call optionally blocks the server. When a connection request is accepted on socket `s`, the name

C Sockets Application Program Interface

of the client and length of the client name are returned, along with a new socket descriptor. The new socket descriptor is associated with the client that initiated the connection and `s` is again available to accept new connections. For a complete description, see “`accept()`” on page 22.

6. Clients and servers have many calls from which to choose for data transfer. The `read()` and `write()`, `readv()` and `writv()`, and `send()` and `recv()` 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. The example shown in Figure 7 illustrates the use of `send()` and `recv()`.

```
int send(int socket, char *buf, int buflen, int flags);
int recv(int socket, char *buf, int buflen, int flags);
:
:
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
int s;
:
:
bytes_sent = send(s, data_sent, sizeof(data_sent), 0);
:
:
bytes_received = recv(s, data_received, sizeof(data_received), 0);
```

Figure 7. An Application Uses the `send()` and `recv()` Calls

The example in Figure 7 shows an application 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. For more information about these routines, see “`read()`” on page 59, “`readv()`” on page 60, “`recv()`” on page 61, “`send()`” on page 68, “`write()`” on page 87, and “`writv()`” on page 88.

7. 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 of the use of the `sendto()` and `recvfrom()` calls is shown in Figure 8 on page 12.

C Sockets Application Program Interface

```
int sendto(int socket, char *buf, int buflen, int flags,
           struct sockaddr *addr, int addrlen);
int recvfrom(int socket, char *buf, int buflen, int flags,
            struct sockaddr *addr, int addrlen);
:
:
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
struct sockaddr_in to;
struct sockaddr from;
int addrlen;
int s;
:
:
to.sin_family = AF_INET;
to.sin_addr.s_addr = inet_addr("129.5.24.1");
to.sin_port = htons(1024);
:
:
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, &from, &addrlen);
```

Figure 8. An Application Uses the `sendto()` and `recvfrom()` Calls

The `sendto()` and `recvfrom()` 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. For more information about these additional parameters, see “`recvfrom()`” on page 62, “`recvmsg()`” on page 63, “`sendmsg()`” on page 69, and “`sendto()`” on page 70. Usually, `sendto()` and `recvfrom()` are used for datagram sockets, and `send()` and `recv()` are used for stream sockets.

8. The `writv()`, `readv()`, `sendmsg()`, calls provide the additional features of scatter and gather data. Scattered data can be located in multiple data buffers. The `writv()` and `sendmsg()` calls gather the scattered data and send it. The `readv()` and `recvmsg()` calls receive data and scatter it into multiple buffers.
9. Applications can handle multiple sockets. In such situations, use the `select()` call to determine the sockets that have data to be read, those that are ready for data to be written, and the sockets that have pending exceptional conditions. An example of how the `select()` call is used is shown in Figure 9 on page 13.

```

fd_set readsocks;
fd_set writesocks;
fd_set exceptsocks;
struct timeval timeout;
int number_of_sockets;
int number_found;
:
:
/* set bits in read write except bit masks. To set mask for a descriptor s use
*   fd_set(s,&readsocks);
*
* set number of sockets to be checked
* number_of_sockets = getdtablesize();
*/
:
:
number_found = select(number_of_sockets,
                      &readsocks, &writesocks, &exceptsocks, &timeout);

```

Figure 9. An Application Uses the `select()` Call

In this example, the application sets bit masks to indicate the sockets being tested for certain conditions and also indicates a time-out. If the time-out parameter is `NULL`, the call does not wait for any socket to become ready on these conditions. If the time-out parameter is nonzero, `select()` waits up to this amount of time for at least one socket to become ready on the indicated conditions. This is useful for applications servicing multiple connections that cannot afford to block, waiting for data on one connection. For a complete description, see “`select()`” on page 64.

10. In addition to `select()`, applications can use the `ioctl()` or `fcntl()` calls to help perform asynchronous (nonblocking) socket operations. An example of the use of the `ioctl()` call is shown in Figure 10.

```

int ioctl(int s, unsigned long command, char *command_data);
:
:
int s;
int dontblock;
char buf[256];
int rc;
dontblock = 1;
:
:
rc = ioctl(s, FIONBIO, (char *) &dontblock);
:
:
if (recv(s, buf, sizeof(buf), 0) == -1 && errno == EWOULDBLOCK)
    /* no data available */
else
    /* either got data or some other error occurred */

```

Figure 10. An Application Uses the `ioctl()` and `fcntl()` Calls

This example causes the socket `s` to be placed into nonblocking mode. When this socket 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`. Setting the mode of the socket to be nonblocking allows an application to continue processing without becoming blocked. For a complete description see: “`fcntl()`” on page 31 and “`ioctl()`” on page 54.

C Sockets Application Program Interface

11. A socket descriptor, `s`, is deallocated with the `close()` call. For a complete description see: “`close()`” on page 27. An example of the `close()` call is shown in Figure 11.

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

Figure 11. An Application Uses the close() Call

A Typical TCP Socket Session

You can use TCP sockets for both passive (server) and active (client) processes. While some commands are necessary for both types, some are role-specific. For sample C socket communication client and server programs, see “Sample C Socket Programs” on page 89.

Once 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.

For the general sequence of calls to be followed for most socket routines using TCP sockets, see Figure 12 on page 15.

C Sockets Application Program Interface

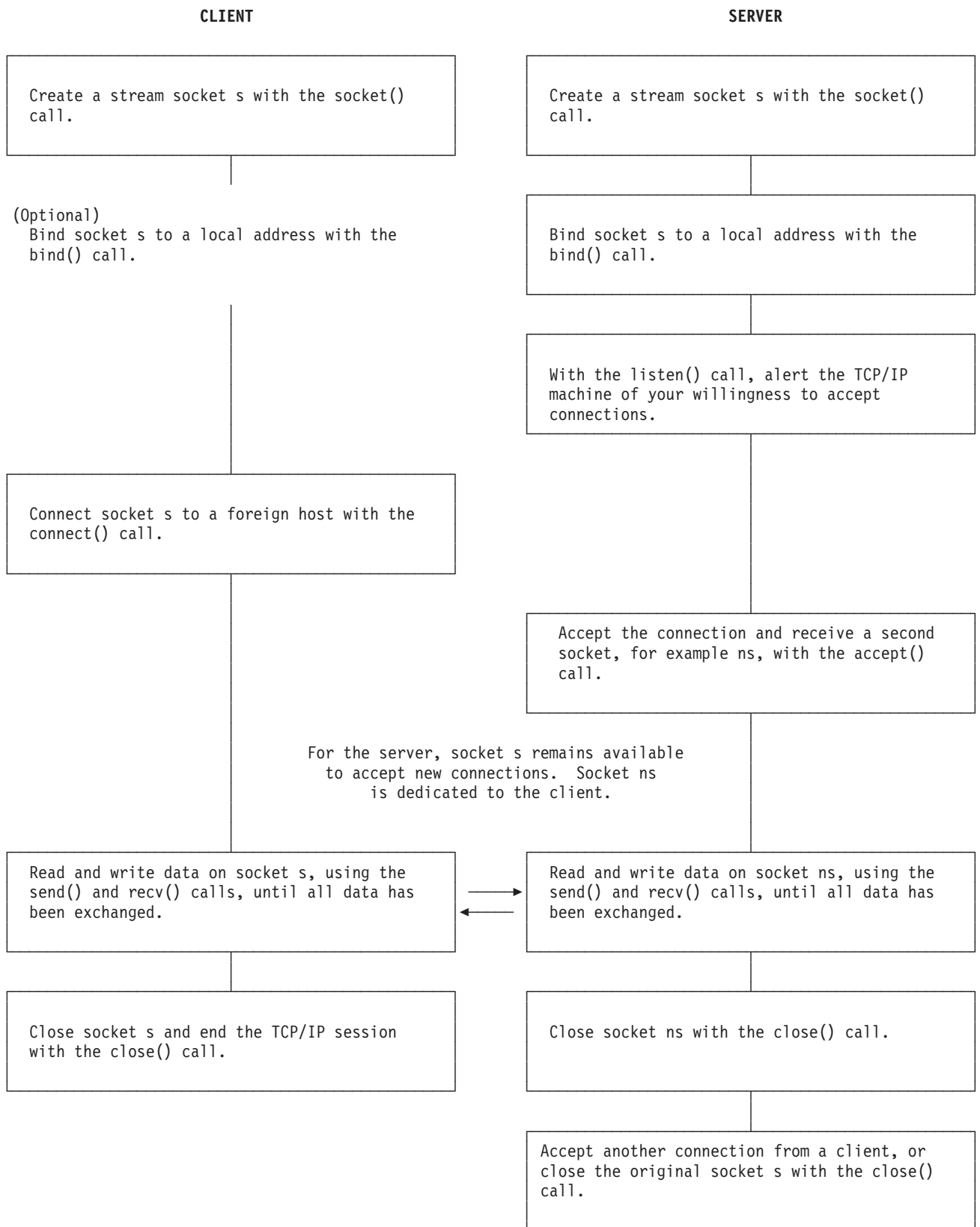


Figure 12. A Typical TCP Socket Session

C Sockets Application Program Interface

A Typical UDP Socket Session

UDP socket processes, unlike TCP socket processes, are not clearly distinguished by server and client roles. Instead, the distinction is between connected and unconnected sockets. An unconnected socket can be used to communicate with any host; however, 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, once a packet has been accepted by the UDP interface, the arrival of the packet and the integrity of the packet cannot be guaranteed.

For the general sequence of calls to be followed for most socket routines using UDP sockets, see Figure 13.

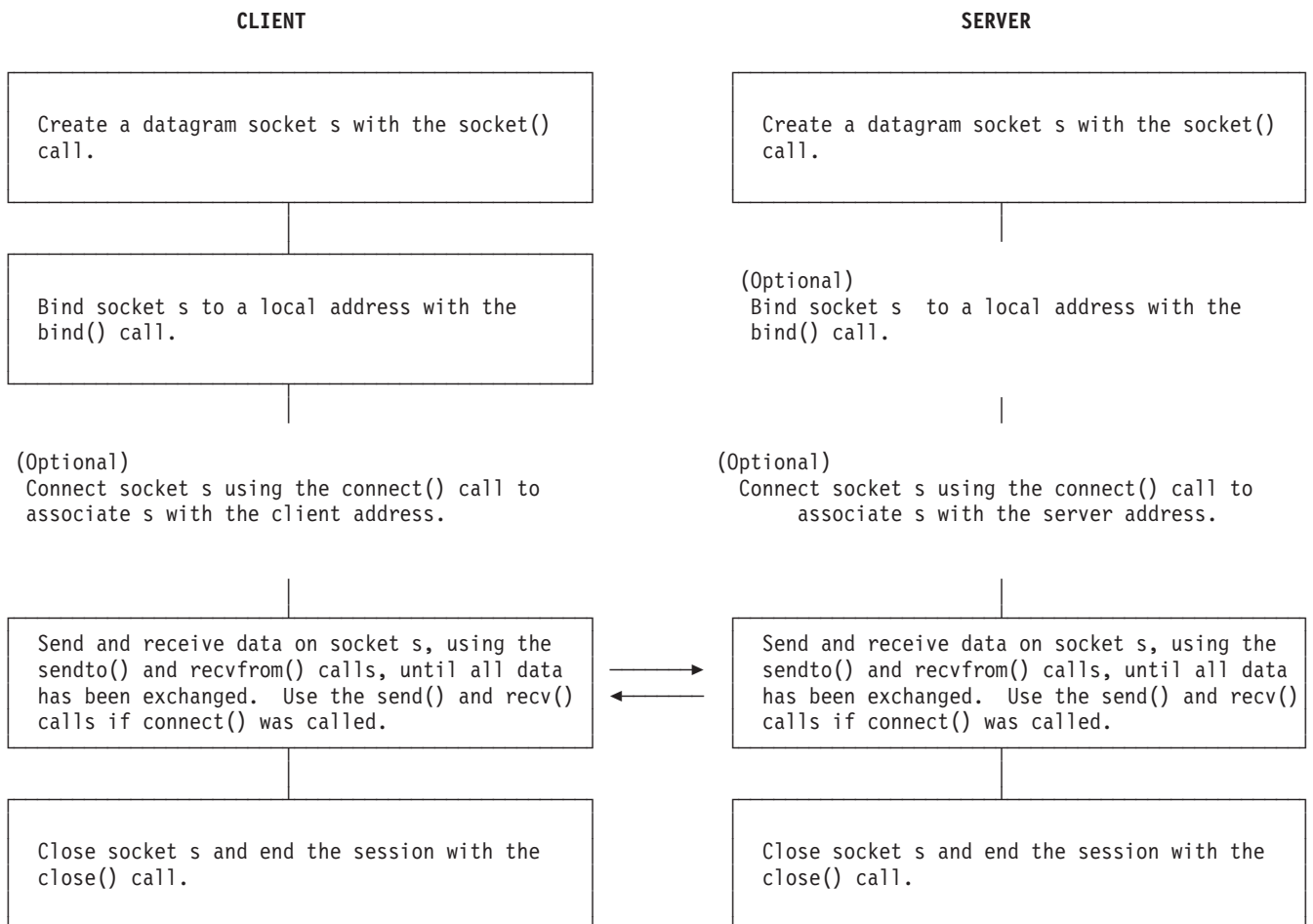


Figure 13. A Typical UDP Socket Session

C Socket Library

To use the socket routines described in this chapter, you must have the following header files available on your system:

- `bsdtypes.h`
- `fcntl.h`
- `if.h`
- `in.h`

- `inet.h`
- `ioctl.h`
- `manifest.h`
- `netdb.h`
- `saiucv.h`
- `socket.h`

The MANIFEST.H header file contains the prototypes for all the functions. To reference the prototypes, you should include the following statement at the beginning of each program:

```
#include <manifest.h>
```

The socket library routines are contained in the COMMTXT TXTLIB file.

Porting

The IBM socket implementation differs from the University of California at Berkeley socket implementation. The following list summarizes the differences between the IBM socket implementation and the Berkeley implementation.

- In the IBM implementation, you must make reference to the additional header file, TCPERRNO.H, if you want to reference the networking errors other than those described in the compiler-supplied ERRNO.H file.
- In the IBM implementation, you must use the `tcperror()` routine to print the networking `errno` messages. `tcperror()` should be used only after socket calls. `perror()` should be used only after C library calls.
- In the IBM implementation, you must include the MANIFEST.H header file to include the prototypes of the socket function calls.
- The IBM `ioctl()` implementation can be different from the current Berkeley `ioctl()` implementation. See “`ioctl()`” on page 54 for a description of the functions supported by the IBM implementation.
- The IBM `getsockopt()` and `setsockopt()` calls support only a subset of the options available. See “`getsockopt()`” on page 45 and “`setsockopt()`” on page 76 for details about the supported options.
- The IBM `fcntl()` call supports only a subset of the options available. See “`fcntl()`” on page 31 for details on the supported options.
- The IBM implementation supports an additional addressing family called `AF_IUCV`, which allows VM virtual machines on the same host to communicate using IUCV.
- The IBM implementation now allows the program to increase the maximum number of simultaneous sockets through the use of the `maxdesc()` call. Previously, the total number of sockets was limited to 253 (numbered 3 through 255) and the first 47 (numbered 3 through 49) could be `AF_INET` sockets. The default maximum number of sockets is 47, any or all of which can be `AF_INET` sockets.

Environment Variables Used by the Sockets Library

Environment variables can be used to affect certain aspects of the execution of a sockets program. If the sockets program is executed from a shell, the shell controls the contents of the program’s environment. For example, the following shell command could be used to set the `X-SITE` environment variable:

```
export X-SITE=//MY.SITEINFO
```

C Sockets Application Program Interface

If the 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 X-SITE environment variable:¹

```
GLOBALV SELECT CENV SET X-SITE MY.SITEINFO
```

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 sockets library:

Variable	Description
X-SITE	This environment variable tells the socket library resolver code to use the 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. This environment variable is used by the gethostbyname() function call, the gethostent() function call, and several others.
X-ADDR	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. This environment variable is used by the gethostbyaddr() function call, the getnetent() function call, and several others.
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. This environment variable is used by the gethostbyname() and gethostbyaddr() function calls.
HOSTALIASES	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

1. Be aware, however, that some of the environment variables described 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.

C Sockets Application Program Interface

tells the resolver to use the `/etc/aliases` file when needed. By default, no aliases files is used by the resolver.

SOCK_DEBUG

This environment variable is used to activate internal socket library debugging messages. If this environment variable is set to the value `ON` (case is not important), then the debugging messages are activated.

C Socket Reference

This section provides a C socket reference table.

Table 2. C Socket Reference

Socket() Call	Description	Page
<code>accept()</code>	Accepts a connection request from a foreign host.	22
<code>bind()</code>	Assigns a local address to the socket.	23
<code>close()</code>	Closes the socket associated with the socket descriptor <code>s</code> .	27
<code>connect()</code>	Requests a connection to a foreign host.	27
<code>endhostent()</code>	Closes the <code>HOSTS SITEINFO</code> and <code>HOSTS ADDRINFO</code> files.	30
<code>endnetent()</code>	Closes the <code>HOSTS SITEINFO</code> file.	31
<code>endprotoent()</code>	Closes the <code>ETC PROTO</code> file.	31
<code>endservent()</code>	Closes the <code>ETC SERVICES</code> file.	31
<code>fcntl()</code>	Controls socket operating characteristics.	31
<code>getclientid()</code>	Returns the identifier by which the calling application is known to the TCPIP virtual machine.	32
<code>gethostbyaddr()</code>	Returns information about a host specified by an address.	33
<code>gethostbyname()</code>	Returns information about a host specified by a name.	34
<code>gethostent()</code>	Returns the next entry in the <code>HOSTS SITEINFO</code> file.	35
<code>gethostid()</code>	Returns the unique identifier of the current host.	36
<code>gethostname()</code>	Returns the standard name of the current host.	36
<code>getnetbyaddr()</code>	Returns the network entry specified by address.	38
<code>getnetbyname()</code>	Returns the network entry specified by name.	39
<code>getnetent()</code>	Returns the next entry in the <code>HOSTS SITEINFO</code> file.	40
<code>getpeername()</code>	Returns the name of the peer connected to socket <code>s</code> .	40

C Socket Reference

Table 2. C Socket Reference (continued)

Socket() Call	Description	Page
getprotobyname()	Returns a protocol entry specified by name.	41
getprotobynumber()	Searches the ETC PROTO file for a specified protocol number.	41
getprotoent()	Returns the next entry in the ETC PROTO file.	42
getservbyname()	Returns a service entry specified by name.	43
getservbyport()	Returns a service entry specified by port number.	43
getservent()	Returns the next entry in the SERVICES file.	44
getsockname()	Obtains local socket name.	44
getsockopt()	Gets options associated with sockets in the AF_INET domain.	45
htonl()	Translates host byte order to network byte order for a long integer.	51
givesocket()	Tells TCPIP to make the specified socket available to a takesocket() call issued by another application.	49
htons()	Translates host byte order to network byte order for a short integer.	51
inet_addr()	Constructs an internet address from character strings set in standard dotted-decimal notation.	52
inet_lnaof()	Returns the local network portion of an internet address.	52
inet_makeaddr()	Constructs an internet address from a network number and a local address.	53
inet_netof()	Returns the network portion of the internet address in network byte order.	53
inet_network()	Constructs a network number from character strings set in standard dotted-decimal notation.	53
inet_ntoa()	Returns a pointer to a string in dotted-decimal notation.	54
ioctl()	Performs special operations on <i>s</i> socket descriptor.	54
listen()	Indicates that a stream socket is ready for a connection request from a foreign host.	56
maxdesc()	Allows socket numbers to extend beyond default range of 0 - 49.	57
ntohl()	Translates network byte order to host byte order for a long integer.	58
ntohs()	Translates network byte order to host byte order for a short integer.	59

Table 2. C Socket Reference (continued)

Socket() Call	Description	Page
read()	Reads a set number of bytes into a buffer.	59
readv()	Obtains data from a socket and reads this data into specified buffers.	60
recv()	Receives messages from a connected socket.	61
recvfrom()	Receives messages from a datagram socket, regardless of its connection status.	62
recvmsg()	Receives messages on a socket with the descriptor <i>s</i> .	63
select()	Detects whether read is possible on a group of sockets.	64
selectex()	Monitors activity on a set of different sockets.	67
send()	Transmits messages to a connected socket.	68
sendmsg()	Sends messages on a socket with descriptor <i>s</i> in an array of headers.	69
sendto()	Transmits messages to a datagram socket, regardless of its connection status.	70
sethostent()	Opens the HOSTS SITEINFO file at the beginning.	72
setnetent()	Opens the HOSTS SITEINFO file at the beginning.	75
setprotoent()	Opens the ETC PROTO file at the beginning.	75
setservent()	Opens the ETC SERVICES file at the beginning.	75
setsockopt()	Sets options associated with a socket in the AF_INET domain.	76
shutdown()	Shuts down all or part of a full-duplex connection.	81
socket()	Requests that a socket be created.	82
takesocket()	Acquires a socket from another application.	85
write()	Writes a set number of bytes from a buffer to a socket.	87
writev()	Writes data in the buffers specified by an array of iovec structures.	88

C Socket Calls

This section provides the syntax, parameters, and other appropriate information for each C socket call supported by TCP/IP.

accept()

accept()

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

int accept(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>name</i>	Specifies the socket address of the connecting client that is filled by <code>accept()</code> before it returns. The format of <i>name</i> is determined by the domain in which the client resides. This parameter can be NULL if the caller is not interested in the client address.
<i>namelen</i>	Initially points to an integer that contains the size in bytes of the storage pointed to by <i>name</i> . On return, that integer contains the size of the data returned in the storage pointed to by <i>name</i> . If <i>name</i> is NULL, then <i>namelen</i> is ignored and can be NULL.

Description: The `accept()` call is used by a server to accept a connection request from a client. The call accepts the first connection on its queue of pending connections. The `accept()` call creates a new socket descriptor with the same properties as *s* and returns it to the caller. If the queue has no pending connection requests, `accept()` blocks the caller unless *s* is in nonblocking mode. If no connection requests are queued and *s* is in nonblocking mode, `accept()` returns `-1` and sets `errno` to `EWOULDBLOCK`. The new socket descriptor cannot be used to accept new connections. The original socket, *s*, remains available to accept more connection requests.

The *s* parameter is a stream socket descriptor created with the `socket()` call. It is usually bound to an address with the `bind()` call, and is made capable of accepting connections with the `listen()` call. The `listen()` call marks the socket as one that accepts connections and allocates a queue to hold pending connection requests. The `listen()` call allows the caller to place an upper boundary on the size of the queue.

The *name* parameter is a pointer to a buffer into which the connection requester's address is placed. The *name* parameter is optional and can be set to be the NULL pointer. If set to NULL, the requester's address is not copied into the buffer. The exact format of *name* depends on the addressing domain from which the communication request originated. For example, if the connection request originated in the `AF_INET` domain, *name* points to a `sockaddr_in` structure as defined in the header file `IN.H`. The *namelen* parameter is used only if *name* is not NULL. Before calling `accept()`, you must set the integer pointed to by *namelen* to the size of the buffer, in bytes, pointed to by *name*. On successful return, the integer pointed to by *namelen* contains the actual number of bytes copied into the buffer. If the buffer is not large enough to hold the address, up to *namelen* bytes of the requester's address are copied.

This call is used only with `SOCK_STREAM` sockets. You cannot screen requesters without calling `accept()`. The application cannot tell the system from which

accept()

requesters it accepts connections from. The caller can, however, choose to close a connection immediately after discovering the identity of the requester.

A socket can be checked for incoming connection requests using the `select()` call.

Return Values: A non-negative socket descriptor indicates success; the value `-1` indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
EBADF	Indicates that the <i>s</i> parameter is not a valid socket descriptor.
EINVAL	Indicates that the <code>listen()</code> was not called for socket <i>s</i> .
EMFILE	Indicates that the socket descriptor table is already full.
ENOBUFS	Indicates that there is insufficient buffer space available to create the new socket.
EOPNOTSUPP	Indicates that the <i>s</i> parameter is not of type <code>SOCK_STREAM</code> .
EFAULT	Indicates that the use of <i>name</i> and <i>namelen</i> would result in an attempt to copy the address into a portion of the caller's virtual storage into which information cannot be written.
EWouldBlock	Indicates that the <i>s</i> parameter is in nonblocking mode and no connections are on the queue.
EIBMIUCVERR	Indicates that an IUCV error occurred.

Examples: The following are two examples of the `accept()` call. In the first, the caller wishes to have the requester's address returned. In the second, the caller does not wish to have the requester's address returned.

```
int clientsocket;
int s;
struct sockaddr clientaddress;
int addrlen;
int accept(int s, struct sockaddr *addr, int *addrlen);
/* socket(), bind(), and listen() have been called */
/* EXAMPLE 1: I want the address now */

addrlen = sizeof(clientaddress);
clientsocket = accept(s, &clientaddress, &addrlen);
/* EXAMPLE 2: I can get the address later using getpeername() */
addrlen = 0;
clientsocket = accept(s, (struct sockaddr *) 0, (int *) 0);
```

See Also: `bind()`, `connect()`, `getpeername()`, `listen()`, `socket()`.

bind()

bind()

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

int bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor returned by a previous <code>socket()</code> call.
<i>name</i>	Points to a <i>sockaddr</i> structure containing the name that is to be bound to <i>s</i> .
<i>namelen</i>	Specifies the size of <i>name</i> in bytes.

Description: The `bind()` call binds a unique local name to the socket with descriptor *s*. After calling `socket()`, a descriptor does not have a name associated with it. However, it does belong to a particular addressing family as specified when `socket()` is called. The exact format of a name depends on the addressing family. The `bind()` procedure also allows servers to specify from which network interfaces they wish to receive UDP packets and TCP connection requests.

The *s* parameter is a socket descriptor of any type created by calling `socket()`.

The *name* parameter is a pointer to a buffer containing the name to be bound to *s*. The *namelen* parameter is the size, in bytes, of the buffer pointed to by *name*.

Socket Descriptor Created in the AF_INET Domain: If the socket descriptor *s* was created in the AF_INET domain, then the format of the name buffer is expected to be *sockaddr_in* as defined in the header file IN.H:

```
struct in_addr
{
    unsigned long s_addr;
};
struct sockaddr_in
{
    short  sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    char  sin_zero[8];
};
```

The *sin_family* field must be set to AF_INET.

The *sin_port* field is set to the port to which the application must bind. It must be specified in network byte order. If *sin_port* is set to 0, the caller leaves it to the system to assign an available port. When the `bind()` call returns, this field is set to the port chosen by the system. Alternatively, the application can call `getsockname()` to discover the port number assigned.

The *sin_addr.s_addr* field is set to the internet address and must be specified in network byte order. On hosts with more than one network interface (called multi-homed hosts), a caller can select the interface with which it is to bind. Subsequently, only UDP packets and TCP connection requests from this interface (which match the bound name) are routed to the application. If this field is set to the constant INADDR_ANY, as defined in the header file IN.H, the caller is leaving the address unspecified and requesting that the socket be bound to all network interfaces on the host. Subsequently, UDP packets and TCP connections from all

interfaces (which match the bound name) are routed to the application. This becomes important when a server offers a service to multiple networks. By leaving the address unspecified, the server can accept all UDP packets and TCP connection requests made for its port, regardless of the network interface on which the requests arrived.

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

Socket Descriptor Created in the AF_IUCV Domain: If the socket descriptor *s* is created in the AF_IUCV domain, the format of the name buffer is expected to be *sockaddr_iucv*, as defined in the header file SAIUCV.H.

```
struct sockaddr_iucv
{
    short          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_family* field must be 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 making the bind call. This field must be eight characters long, padded with blanks to the right. It cannot contain the NULL character.

The *siucv_name* field is set to the application name by which the socket is to be known. It must be unique, because only one socket can be bound to a given name. The recommended form of the name contains eight characters, padded with blanks to the right. The eight characters for a connect() call executed by a client must exactly match the eight characters passed in the bind() call executed by the server.

Note: Internally, dynamic names are built using hexadecimal character strings representing the internal memory address of the socket. An example of this is when an application calls connect and specifies an unbound socket. You should choose names that contain at least one non-hexadecimal character to prevent potential conflicts. Hexadecimal characters include 0—9, and a—f. Uppercase A—F are considered nondecimal and can be used by the user in building dynamic names.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of errno indicates the specific error.

Errno Value	Description
EBADF	Indicates that the <i>s</i> parameter is not a valid socket descriptor.
EADDRNOTAVAIL	Indicates that the address specified is not valid on this host. For example, if the internet address does not specify a valid network interface.
EFAULT	Using <i>name</i> and <i>namelen</i> would result in an attempt

bind()

	to copy the address into a non-writable portion of the caller's virtual storage.
EAFNOSUPPORT	Indicates that the address family is not supported (it is not AF_IUCV or AF_INET).
EADDRINUSE	Indicates that the address is already in use. See the SO_REUSEADDR option described under "getsockopt()" on page 45 and the SO_REUSEADDR described under the "setsockopt()" on page 76.
EINVAL	Indicates that the socket is already bound to an address. For example, trying to bind a name to a socket that is in the connected state. If you are using sockets in the AF_IUCV domain, this error also occurs if NULL characters appear in the <i>siucv_nodeid</i> or <i>siucv_user_id</i> fields. This value is also returned if <i>namelen</i> is not the expected length.

Examples: The following are examples of the bind() call. The internet address and port must be in network byte order. To put the port into network byte order, a utility routine, htons(), is called to convert a short integer from host byte order to network byte order. The *address* field is set using another utility routine, inet_addr(), which takes a character string representing the dotted-decimal address of an interface and returns the binary internet address representation in network byte order. Finally, you should zero the structure before using it to ensure that the name requested does not set any reserved fields. For examples of how a client might connect to servers, see "connect()" on page 27.

AF_INET Domain Example: The following example illustrates the bind() call binding to interfaces in the AF_INET domain.

```
int rc;
int s;
struct sockaddr_in myname;
struct sockaddr_iucv myvmname;
int bind(int s, struct sockaddr *name, int namelen);

/* Bind to a specific interface in the internet domain */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = htons(1024);
:
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
/* Bind to all network interfaces in the internet domain */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = INADDR_ANY; /* specific interface */
myname.sin_port = htons(1024);
:
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
/* Bind to a specific interface in the internet domain.
   Let the system choose a port */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = 0;
```

```

:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));

```

AF_IUCV Domain Example: The following example illustrates the bind() call binding to interfaces in the AF_IUCV domain.

```

/* Bind to a name in the IUCV domain */
/* make sure the siucv_addr, siucv_port fields are zeroed and the
   siucv_nodeid fields is set to blanks */
memset(&myvmname, 0, sizeof(myvmname));
strncpy(myvmname.siucv_nodeid, "          ", 8);
strncpy(myvmname.siucv_userid, "          ", 8);
strncpy(myvmname.siucv_name, "          ", 8);
myvmname.siucv_family = AF_IUCV;
strncpy(myvmname.siucv_userid, "VMUSER1", 7);
strncpy(myvmname.siucv_name, "APPL", 4);
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));

```

The binding of a stream socket is not complete until a successful call to bind(), listen(), or connect() is made. Applications using stream sockets should check the return values of bind(), listen(), and connect() before using any function that requires a bound stream socket.

See Also: connect(), gethostbyname(), getsockname(), htons(), inet_addr(), listen(), socket().

close()

```

int close(s)
int s;

```

Parameter	Description
s	Specifies the descriptor of the socket to be closed.

Description: The close() call shuts down the socket associated with the socket descriptor s, and frees resources allocated to the socket. If s refers to an open TCP connection, the connection is closed. If a stream socket is closed when there is input data queued, the TCP connection is reset rather than being cleanly closed.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of errno indicates the specific error.

Errno Value	Description
EBADF	Indicates that the s parameter is not a valid socket descriptor.

See Also: accept(), getsockopt(), setsockopt(), socket().

connect()

connect()

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

int connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>name</i>	Points to a <i>socket address</i> structure containing the address of the socket to which a connection is attempted.
<i>namelen</i>	Specifies the size of the <i>socket address</i> pointed to by <i>name</i> in bytes.

Description: For stream sockets, the connect() call attempts to establish a connection between two sockets. For UDP sockets, the connect() call specifies the peer for a socket. The *s* parameter is the socket used to originate the connection request. The connect() call performs two tasks when called for a stream socket. First, it completes the binding necessary for a stream socket (in case it has not been previously bound using the bind() call). Second, it attempts to make a connection to another socket.

The connect() call on a stream socket is used by the client application to establish a connection to a server. The server must have a passive open pending. If the server is using sockets, this means the server must successfully call bind() and listen() before a connection can be accepted by the server with accept(). Otherwise, connect() returns -1 and errno is set to ECONNREFUSED.

If *s* is in blocking mode, the connect() call blocks the caller until the connection is set up, or until an error is received. If the socket is in nonblocking mode then connect() returns -1 with errno set to EINPROGRESS if the connection can be initiated (no other errors occurred). The caller can test the completion of the connection setup by calling select() and testing for the ability to write to the socket.

When called for a datagram or raw socket, connect() specifies the peer with which this socket is associated. This gives the application the ability to use data transfer calls reserved for sockets that are in the connected state. In this case, read(), write(), readv(), writev(), send(), recv() are then available in addition to sendto(), recvfrom(), sendmsg(), recvmsg(). Stream sockets can call connect() only once, however, datagram sockets can call connect() multiple times to change their association. Datagram sockets can dissolve their association by connecting to an invalid address such as the NULL address (all fields zeroed).

The *name* parameter is a pointer to a buffer containing the name of the peer to which the application needs to connect. The *namelen* parameter is the size, in bytes, of the buffer pointed to by *name*.

Servers in the AF_INET Domain: If the server is in the AF_INET domain, the format of the name buffer is expected to be *sockaddr_in*, as defined in the header file IN.H.

```
struct in_addr
{
    unsigned long s_addr;
};
struct sockaddr_in
{
```

```

    short    sin_family;
    unsigned short sin_port;
    struct   in_addr sin_addr;
    char     sin_zero[8];
};

```

The *sin_family* field must be set to AF_INET. The *sin_port* field is set to the port to which the server is bound. It must be specified in network byte order. The *sin_zero* field is not used and must be set to all zeroes.

Servers in the AF_IUCV Domain: If the server is in the AF_IUCV domain, the format of the name buffer is expected to be *sockaddr_iucv*, as defined in the header file SAIUCV.H.

```

struct sockaddr_iucv
{
    short          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_family* field must be 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 8 blank characters. The *siucv_userid* field is set to the VM user ID of the application to which the application is requesting a connection. This field must be 8 characters long, padded with blanks to the right. It cannot contain the NULL character. The *siucv_name* field is set to the application name by which the server socket is known. The name must exactly match the 8 characters passed in the bind() call executed by the server.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of errno indicates the specific error.

Errno Value	Description
EADDRNOTAVAIL	Indicates that the calling host cannot reach the specified destination.
EAFNOSUPPORT	Indicates that the address family is not supported.
EALREADY	Indicates that the socket <i>s</i> is marked nonblocking, and a previous connection attempt has not completed.
EBADF	Indicates that the <i>s</i> parameter is not a valid socket descriptor.
ECONNREFUSED	Indicates that the connection request was rejected by the destination host.
EFAULT	Indicates that the use of <i>name</i> and <i>namelen</i> would result in an attempt to copy the address into a portion of the caller's virtual storage to which data cannot be written.
EINPROGRESS	Indicates that the socket <i>s</i> is marked nonblocking, and the connection cannot be completed immediately. The EINPROGRESS value does not indicate an error condition.

connect()

EINVAL	Indicates that the <i>namelen</i> parameter is not a valid length.
EISCONN	Indicates that the socket <i>s</i> is already connected.
ENETUNREACH	Indicates that the network cannot be reached from this host.
ETIMEDOUT	Indicates that the connection establishment timed out before a connection was made.

Examples: The following are examples of the `connect()` call. The internet address and port must be in network byte order. To put the port into network byte order a utility routine, `htons()`, is called to convert a short integer from host byte order to network byte order. The *address* field is set using another utility routine, `inet_addr()`, which takes a character string representing the dotted-decimal address of an interface and returns the binary internet address representation in network byte order. Finally, note that it is a good idea to zero the structure before using it to ensure that the name requested does not set any reserved fields. These examples could be used to connect to the servers shown in the examples listed with the call, “`bind()`” on page 23.

```
int s;
struct sockaddr_in servername;
struct sockaddr_iucv vmservername;
int rc;
int connect(int s, struct sockaddr *name, int namelen);

/* Connect to server bound to a specific interface in the internet domain */
/* make sure the sin_zero field is cleared */
memset(&servername, 0, sizeof(servername));
servername.sin_family = AF_INET;
servername.sin_addr.s_addr = inet_addr("129.5.24.1"); /* specific interface */
servername.sin_port = htons(1024);
:
:
rc = connect(s, (struct sockaddr *) &servername, sizeof(servername));

/* Connect to a server bound to a name in the IUCV domain */
/* make sure the siucv_addr, siucv_port, siucv_nodeid fields are cleared */
memset(&vmservername, 0, sizeof(vmservername));
vmservername.siucv_family = AF_IUCV;
strncpy(vmservername.siucv_nodeid, "          ",8);
/* The field is eight positions padded to the right with blanks */
strncpy(vmservername.siucv_userid, "VMUSER1 ", 8);
strncpy(vmservername.siucv_name, "APPL    ", 8);
:
:
rc = connect(s, (struct sockaddr *) &vmservername, sizeof(vmservername));
```

See Also: `accept()`, `bind()`, `htons()`, `inet_addr()`, `listen()`, `select()`, `selectex()`, `socket()`.

endhostent()

```
void endhostent()
```

The `endhostent()` call has no parameters.

Description: The `endhostent()` call closes the `HOSTS SITEINFO` and `HOSTS ADDRINFO` files. The `HOSTS SITEINFO` and `HOSTS ADDRINFO` files contain information about known hosts.

endhostent()

The `endhostent()` call is available only if `RESOLVE_VIA_LOOKUP` is defined before the `MANIFEST.H` header file is included.

See Also: `gethostbyaddr()`, `gethostbyname()`, `gethostent()`, `sethostent()`.

endnetent()

```
void endnetent()
```

The `endnetent()` call has no parameters.

Description: The `endnetent()` call closes the `HOSTS SITEINFO` file. The `HOSTS SITEINFO` file contains information about known networks.

See Also: `getnetbyaddr()`, `getnetbyname()`, `getnetent()`, `setnetent()`.

endprotoent()

```
void endprotoent()
```

The `endprotoent()` call has no parameters.

Description: The `endprotoent()` call closes the `ETC PROTO` file.

See Also: `getprotobyname()`, `getprotoent()`, `setprotoent()`.

endservent()

```
void endservent()
```

The `endservent()` call has no parameters.

Description: The `endservent()` call closes the `ETC SERVICES` file.

See Also: `getservbyname()`, `getservbyport()`, `getservent()`, `setservent()`.

fcntl()

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

int fcntl(s, cmd, data)
int s;
int cmd;
int data;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>cmd</i>	Specifies the command to perform.
<i>data</i>	Specifies the data associated with the <i>cmd</i> parameter.

fcntl()

Description: The operating characteristics of sockets can be controlled with `fcntl()` requests. The operations to be controlled are determined by the `cmd` parameter. The `data` parameter is a variable with a meaning that depends on the value of the `cmd` parameter.

The following are valid `fcntl()` commands:

Command	Description
F_SETFL	Sets the status flags of socket <i>s</i> . One flag, <code>FNDELAY</code> , can be set. <ul style="list-style-type: none">Setting the <code>FNDELAY</code> flag marks <i>s</i> as being in nonblocking mode. If data is not present on calls that can block, such as <code>read()</code>, <code>readv()</code>, <code>recv()</code>, the call returns <code>-1</code> and <code>errno</code> is set to <code>EWOULDBLOCK</code>.
F_GETFL	Gets the status flags of socket <i>s</i> . One flag, <code>FNDELAY</code> , can be queried. <ul style="list-style-type: none">The <code>FNDELAY</code> flag marks <i>s</i> as being in nonblocking mode. If data is not present on calls that can block, such as <code>read()</code>, <code>readv()</code>, <code>recv()</code>, the call returns with <code>-1</code> and <code>errno</code> is set to <code>EWOULDBLOCK</code>.

Return Values: For the `F_GETFL` command, the return value is the flags, set as a bit mask. For the `F_SETFL` command, the value `0` indicates success; the value `-1` indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
EBADF	Indicates that the <i>s</i> parameter is not a valid socket descriptor.
EINVAL	Indicates that the <i>data</i> parameter is not a valid flag.

Examples: The following are examples of the `fcntl()` call.

```
int s;
int rc;
int flags;
:
:
/* Place the socket into nonblocking mode */
rc = fcntl(s, F_SETFL, FNDELAY);

/* See if asynchronous notification is set */
flags = fcntl(s, F_GETFL, 0);
if (flags & FNDELAY)
    /* it is set */
else
    /* it is not */
```

See Also: `fcntl()`, `ioctl()`, `getsockopt()`, `setsockopt()`, `socket()`.

getclientid()

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

int getclientid(domain, clientid)
int domain,
struct clientid *clientid;
```

Parameter	Description
<i>domain</i>	Specifies the address family <i>domain</i> and must be <code>AF_INET</code> .

clientid Points to a *clientid* structure to be filled.

Description: The `getclientid()` call returns the identifier by which the calling program is known to the TCPIP virtual machine. The *clientid* is used in the `givesocket()` and `takesocket()` calls.

Return Values: The value 0 indicates success. The value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
-------------	-------------

EFAULT	Indicates that using the <i>clientid</i> parameter as specified would result in an attempt to access storage outside the caller's virtual storage, or storage not modifiable by the caller.
---------------	---

EPFNOSUPPORT	Indicates that the domain is not AF_INET.
---------------------	---

See Also: `givesocket()`, `takesocket()`.

getdtablesize()

```
int getdtablesize()
```

The `getdtablesize()` call has no parameters.

Description: The TCPIP virtual machine reserves a fixed-size table for each machine using sockets. The size of this table is the number of sockets a machine can allocate simultaneously. The `getdtablesize()` function returns the size of this table.

To increase the table size, use `maxdesc()`. After calling `maxdesc()`, always use `getdtablesize()` to verify that the table was changed.

See Also: `maxdesc()`.

gethostbyaddr()

```
#include <netdb.h>

struct hostent *gethostbyaddr(addr, addrlen, domain)
char *addr;
int addrlen;
int domain;
```

Parameter	Description
-----------	-------------

<i>addr</i>	Points to a structure containing the address of the socket. For AF_INET, <i>addr</i> is a pointer to an <i>in_addr</i> structure.
-------------	---

<i>addrlen</i>	Specifies the size of <i>addr</i> in bytes.
----------------	---

<i>domain</i>	Specifies the address domain supported (AF_INET).
---------------	---

Description: The `gethostbyaddr()` call tries to resolve the host name through a name server, if one is present. If a name server is not present, `gethostbyaddr()`

gethostbyaddr()

searches the HOSTS ADDRINFO file until a matching host address is found or an EOF marker is reached. These files are described in *TCP/IP Planning and Customization*.

The `gethostbyaddr()` call returns a pointer to a *hostent* structure for the host address specified on the call.

If `RESOLVE_VIA_LOOKUP` is defined before including the MANIFEST.H header file, `gethostbyaddr()` uses only the HOSTS ADDRINFO file for resolution. Otherwise, the name server is required.

Note: The HOSTS ADDRINFO file is created when `MAKESITE` is run against the HOSTS LOCAL file. These files are described in the *TCP/IP for VM: Planning and Customization*.

The NETDB.H header file defines the *hostent* structure, and contains the following elements:

Element	Description
<i>h_name</i>	Indicates the official name of the host.
<i>h_aliases</i>	Indicates a zero-terminated array of alternative names for the host.
<i>h_addrtype</i>	Indicates the type of address returned; currently, always set to <code>AF_INET</code> .
<i>h_length</i>	Indicates the length of the address in bytes.
<i>h_addr</i>	Indicates a pointer to a pointer to a string of bytes of length <i>h_length</i> , which is the network address number of the host.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: `gethostbyname()`, `gethostent()`, `sethostent()`.

gethostbyname()

```
#include <netdb.h>

struct hostent *gethostbyname(name)
char *name;
```

Parameter	Description
<i>name</i>	Specifies the name of the host being queried.

Description: The `gethostbyname()` call tries to resolve the host name through a name server, if one is present. If a name server is not present, `gethostbyname()` searches the HOSTS SITEINFO file until a matching host name is found or an EOF marker is reached.

The `gethostbyname()` call returns a pointer to a *hostent* structure for the host name specified on the call.

If `RESOLVE_VIA_LOOKUP` is defined before including the MANIFEST.H header file, `gethostbyname()` uses only the HOSTS ADDRINFO file for resolution.

gethostbyname()

Note: The HOSTS ADDRINFO file is created when MAKESITE is run against the HOSTS LOCAL file. These files are described in *TCP/IP Planning and Customization*.

The NETDB.H header file defines the *hostent* structure and contains the following elements:

Element	Description
<i>h_name</i>	Indicates the official name of the host.
<i>h_aliases</i>	Indicates a zero-terminated array of alternative names for the host.
<i>h_addrtype</i>	Indicates the type of address returned; currently, it is always set to AF_INET.
<i>h_length</i>	Indicates the length of the address in bytes.
<i>h_addr</i>	Indicates a pointer to a pointer to a string of bytes of length <i>h</i> length, which is the network address number of the host.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: gethostbyaddr(), gethostent(), sethostent().

gethostent()

```
#include <netdb.h>
struct hostent *gethostent()
```

The gethostent() call has no parameters.

Description: The gethostent() call reads the next line of the HOSTS SITEINFO file.

The gethostent() call returns a pointer to the next entry in the HOSTS SITEINFO file. gethostent() uses HOSTS ADDRINFO to get aliases.

gethostent() is available only if RESOLVE_VIA_LOOKUP is defined before the MANIFEST.H header file is included. The NETDB.H header file defines the *hostent* structure, and contains the following elements:

Element	Description
<i>h_name</i>	Indicates the official name of the host.
<i>h_aliases</i>	Indicates a zero-terminated array of alternative names for host.
<i>h_addrtype</i>	Indicates the type of address returned; currently, always set to AF_INET.
<i>h_length</i>	Indicates the length of the address in bytes.
<i>h_addr</i>	Indicates a pointer to a pointer to a string of bytes of length <i>h</i> length, which is the network address number of the host.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or end-of-file.

gethostid()

See Also: gethostbyaddr(), gethostbyname(), sethostent().

gethostid()

```
#include <bsdtypes.h>
unsigned long gethostid()
```

The `gethostid()` call has no parameters.

Description: The `gethostid()` call gets the unique 32-bit identifier for the current host. This value is the default home internet address.

Return Values: The `gethostid()` call returns the 32-bit identifier of the current host, which should be unique across all hosts.

gethostname()

```
int gethostname(name, namelen)
char *name;
int namelen;
```

Parameter	Description
<i>name</i>	Specifies the character array to be filled with the host name.
<i>namelen</i>	Specifies the length of <i>name</i> .

Description: 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 array. The returned name is NULL terminated unless there is insufficient room in the name array.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of `errno` indicates the specific error.

See Also: `gethostbyname()`.

getibmsockopt()

Like `getsockopt()`, the `getibmsockopt()` gets the options associated with a socket in the `AF_INET` domain. This call is for options specific to the IBM implementation of sockets. Currently, only the `SOL_SOCKET` level and the socket options `SO_BULKMODE` and `SO_IGNOREINCOMINGPUSH` are supported.

Use `getibmsockopt()` with the socket option `SO_BULKMODE` to test whether the UDP socket *s* is in bulk mode. The `bulkmode` socket option enables an application to queue multiple datagrams, sending all of the datagrams in one send. This reduces the CPU consumption for each datagram.

This call can be used only in the `AF_INET` domain.

```
#include <manifest.h>
#include <socket.h>
```

```
int getibmssockopt(int s, int level, int optname, char *optval, int *optlen)
```

Parameter	Description
<i>s</i>	The socket descriptor.
<i>level</i>	The level for which the option is set.
<i>optname</i>	The name of a specified socket option.
<i>optval</i>	The pointer to option data.
<i>optlen</i>	The pointer to the length of the option data.

For SO_BULKMODE, *optval* should point to an `ibm_bulkmode_struct`, which is defined in SOCKET.H. The `ibm_bulkmode_struct` contains the following fields:

Element	Description
b_onoff	1 means bulk mode is on; 0 means bulk mode is off.
b_max_receive_queue_size	The maximum receiving queue size in bytes.
b_max_send_queue_size	The maximum sending queue size in bytes.
b_teststor	If this element is nonzero, the address of the message buffer and the message buffer itself is checked for addressability during each socket call. <code>errno</code> is set to EFAULT if there is an addressing exception. If this element is zero, no checking is performed.
b_max_send_queue_size_avail	The maximum send queue size in bytes that can be set with <code>setibmssockopt()</code> .
b_num_IUCVs_sent	The number of actual IUCVs issued in sending datagrams to TCPIP.
b_num_IUCVs_received	The number of actual IUCVs issued in receiving datagrams from TCPIP.

The fields `b_num_IUCVs_sent` and `b_num_IUCVs_received` represent cumulative totals for this socket since the time the application was started.

For SO_IGNOREINCOMINGPUSH, *optval* should point to an integer. `getibmssockopt()` returns 0 in *optval* if the option is not set, and returns 1 in *optval* if the option is set.

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

getibmssockopt()

```
#include <manifest.h>
#include <socket.h>
#include <tcperror.h>

{ struct ibm_bulkmode_struct bulkstr;
  int optlen, rc;

  optlen = sizeof(bulkstr);
  rc = getibmssockopt(s, SOL_SOCKET, SO_BULKMODE, (char *), &bulkstr, &optlen);
  if (rc < 0)
  { tcperror("on getibmssockopt()");
    exit(1);
  }
  fprintf(stream, "%d byte buffer available for outbound queue.\n",
          bulkstr.b_max_send_queue_size_avail);
}
```

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
EBADF	The <code>s</code> parameter is not a valid socket descriptor.
EFAULT	Using <code>optval</code> and <code>optlen</code> parameters would result in an attempt to access storage outside the caller's address space.
EIBMIUCVERR	An IUCV error occurred.
ENOPROTOOPT	The <code>optname</code> parameter is unrecognized, or the level parameter is not <code>SOL_SOCKET</code> .

Related Calls: `ibmsflush()`, `setibmssockopt()`, `getsockopt()`.

getnetbyaddr()

```
#include <netdb.h>

struct netent *getnetbyaddr(net, type)
unsigned long net;
int type;
```

Parameter	Description
<i>net</i>	Specifies the network address.
<i>type</i>	Specifies the address domain supported (<code>AF_INET</code>).

Description: The `getnetbyaddr()` call searches the `HOSTS ADDRINFO` file for the specified network address.

Note: `HOSTS LOCAL`, `HOSTS ADDRINFO`, and `HOSTS SITEINFO` are described in *TCP/IP Planning and Customization*.

The `netent` structure is defined in the `NETDB.H` header file and contains the following elements:

Element	Description
<i>n_name</i>	Indicates the official name of the network.
<i>n_aliases</i>	Indicates an array, terminated with a NULL pointer, of alternative names for the network.

getnetbyaddr()

<i>n_addrtype</i>	Indicates the type of network address returned. The call always sets this value to AF_INET.
<i>n_net</i>	Indicates the network number, returned in host byte order.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: `endnetent()`, `getnetbyname()`, `getnetent()`, `setnetent()`.

getnetbyname()

```
#include <netdb.h>

struct netent *getnetbyname(name)
char *name;
```

Parameter	Description
<i>name</i>	Points to a network name.

Description: The `getnetbyname()` call searches the HOSTS ADDRINFO file for the specified network name.

Note: HOSTS LOCAL, HOSTS ADDRINFO, and HOSTS SITEINFO are described in *TCP/IP Planning and Customization*.

The `getnetbyname()` call returns a pointer to a *netent* structure for the network name specified on the call.

The *netent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
<i>n_name</i>	Indicates the official name of the network.
<i>n_aliases</i>	Indicates an array, terminated with a NULL pointer, of alternative names for the network.
<i>n_addrtype</i>	Indicates the type of network address returned. The call always sets this value to AF_INET.
<i>n_net</i>	Indicates the network number, returned in host byte order.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: `endnetent()`, `getnetbyaddr()`, `getnetent()`, `setnetent()`.

getnetent()

getnetent()

```
#include <netdb.h>

struct netent *getnetent()
```

The `getnetent()` call has no parameters.

Description: The `getnetent()` call reads the next entry of the HOSTS SITEINFO file.

Note: HOSTS LOCAL, HOSTS ADDRINFO, and HOSTS SITEINFO are described in *TCP/IP Planning and Customization*.

The *netent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
<i>n_name</i>	Indicates the official name of the network.
<i>n_aliases</i>	Indicates an array, terminated with a NULL pointer, of alternative names for the network.
<i>n_addrtype</i>	Indicates the type of network address being returned. The call always sets this value to AF_INET.
<i>n_net</i>	Indicates the network number, returned in host byte order.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: `endnetent()`, `getnetbyaddr()`, `getnetbyname()`, `setnetent()`.

getpeername()

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

int getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>name</i>	Specifies the internet address of the connected socket that is filled by <code>getpeername()</code> before it returns. The exact format of the <i>name</i> parameter is determined by the domain in which communication occurs.
<i>namelen</i>	Specifies the size of the address structure pointed to by the <i>name</i> parameter in bytes.

Description: The `getpeername()` call returns the name of the peer connected to socket *s*. The *namelen* pointer must be initialized to indicate the size of the space pointed to by *name* and is set to the number of bytes copied into the space before

getpeername()

the call returns. The size of the peer name is returned in bytes. If the buffer of the local host is too small, the peer name is truncated.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
EBADF	Indicates that the <i>s</i> parameter is not a valid socket descriptor.
EFAULT	Indicates that using the <i>name</i> and <i>namelen</i> parameters as specified would result in an attempt to access storage outside of the caller's virtual storage.
ENOTCONN	Indicates that the socket is not in the connected state.

See Also: `accept()`, `connect()`, `getsockname()`, `socket()`.

getprotobyname()

```
#include <netdb.h>

struct protoent *getprotobyname(name)
char *name;
```

Parameter	Description
<i>name</i>	Points to the specified protocol.

Description: The `getprotobyname()` call searches the ETC PROTO file for the specified protocol name.

The `getprotobyname()` call returns a pointer to a *protoent* structure for the network protocol specified on the call.

The *protoent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
<i>p_name</i>	Indicates the official name of the protocol.
<i>p_aliases</i>	Indicates an array, terminated with a NULL pointer, of alternative names for the protocol.
<i>p_proto</i>	Indicates the protocol number.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: `endprotoent()`, `getprotobynumber()`, `getprotoent()`, `setprotoent()`.

getprotobynumber()

getprotobynumber()

```
#include <netdb.h>

struct protoent * getprotobynumber(proto)
int proto;
```

Parameter	Description
<i>proto</i>	Specifies the protocol number.

Description: The `getprotobynumber()` call searches the ETC PROTO file for the specified protocol number.

The `getprotobynumber()` call returns a pointer to a *protoent* structure for the network protocol specified on the call.

The *protoent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
<i>p_name</i>	Indicates the official name of the protocol.
<i>p_aliases</i>	Indicates an array, terminated with a NULL pointer, of alternative names for the protocol.
<i>p_proto</i>	Indicates the protocol number.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: `endprotoent()`, `getprotobyname()`, `getprotoent()`, `setprotoent()`.

getprotoent()

```
#include <netdb.h>

struct protoent *getprotoent()
```

The `getprotoent()` call has no parameters.

Description: The `getprotoent()` call reads the ETC PROTO file.

The `getprotoent()` call returns a pointer to the next entry in the ETC PROTO file.

The *protoent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
<i>p_name</i>	Indicates the official name of the protocol.
<i>p_aliases</i>	Indicates an array, terminated with a NULL pointer, of alternative names for the protocol.
<i>p_proto</i>	Indicates the protocol number.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endprotoent(), getprotobyname(), getprotobynumber(), setprotoent().

getservbyname()

```
#include <netdb.h>

struct servent *getservbyname(name, proto)
char *name;
char *proto;
```

Parameter	Description
<i>name</i>	Points to the specified service name.
<i>proto</i>	Points to the specified protocol.

Description: The getservbyname() call searches the ETC SERVICES file for the specified service name. Searches for a service name must match the protocol, if a protocol is stated.

The getservbyname() call returns a pointer to a *servent* structure for the network service specified on the call.

The *servent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
<i>s_name</i>	Indicates the official name of the service.
<i>s_aliases</i>	Indicates an array, terminated with a NULL pointer, of alternative names for the service.
<i>s_port</i>	Indicates the port number of the service.
<i>s_proto</i>	Indicates the required protocol to contact the service.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *servent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endservent(), getservbyport(), getservent(), setservent().

getservbyport()

```
#include <netdb.h>

struct servent *getservbyport(port, proto)
int port;
char *proto;
```

Parameter	Description
<i>port</i>	Specifies the port.
<i>proto</i>	Points to the specified protocol.

Description: The getservbyport() call searches the ETC SERVICES file for the specified port number. Searches for a port number must match the protocol if a protocol is stated.

getservbyport()

The `getservbyport()` call returns a pointer to a *servent* structure for the port number specified on the call.

The *servent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
<i>s_name</i>	Specifies the official name of the service.
<i>s_aliases</i>	Indicates an array, terminated with a NULL pointer, of alternative names for the service.
<i>s_port</i>	Specifies the port number of the service.
<i>s_proto</i>	Specifies the protocol required to contact the service.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *servent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: `endservent()`, `getservbyname()`, `getservent()`, `setservent()`.

getservent()

```
#include <netdb.h>
struct servent *getservent()
```

The `getservent()` call has no parameters.

Description: The `getservent()` call reads the next line of the ETC SERVICES file.

The `getservent()` call returns a pointer to the next entry in the ETC SERVICES file.

The *servent* structure is defined in the NETDB.H header file and contains the following elements:

Element	Description
<i>s_name</i>	Specifies the official name of the service.
<i>s_aliases</i>	Indicates an array, terminated with a NULL pointer, of alternative names for the service.
<i>s_port</i>	Specifies the port number of the service.
<i>s_proto</i>	Specifies the required protocol to contact the service.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *servent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: `endservent()`, `getservbyname()`, `getservbyport()`, `setservent()`.

getsockname()

getsockname()

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

int getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>name</i>	Specifies the address of the buffer into which <code>getsockname()</code> copies the name of <i>s</i> .
<i>namelen</i>	Initially points to an integer that contains the size in bytes of the storage pointed to by <i>name</i> . Upon return, this integer contains the size of the data returned in the storage pointed to by <i>name</i> .

Description: The `getsockname()` call stores the current name for the socket specified by the *s* parameter into the structure pointed to by the *name* parameter. It returns the address to the socket that has been bound. If the socket is not bound to an address, the call returns with the family set and the rest of the structure set to zero. For example, an inbound socket in the internet domain would cause the name to point to a `sockaddr_in` structure with the `sin_family` field set to `AF_INET` and all other fields zeroed.

Stream sockets are not assigned a name, until after a successful call to either `bind()`, `connect()`, or `accept()`.

The `getsockname()` call is often used to discover the port assigned to a socket after the socket has been implicitly bound to a port. For example, an application can call `connect()` without previously calling `bind()`. In this case, the `connect()` call completes the binding necessary by assigning a port to the socket. This assignment can be discovered with a call to `getsockname()`.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
<code>EBADF</code>	Indicates that the <i>s</i> parameter is not a valid socket descriptor.
<code>EFAULT</code>	Using the <i>name</i> and <i>namelen</i> parameters as specified would result in an attempt to access storage outside of the caller's virtual storage.

See Also: `accept()`, `bind()`, `connect()`, `getpeername()`, `socket()`.

getsockopt()

getsockopt()

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

int getsockopt(s, level, optname, optval, optlen)
int s;
int level;
int optname;
char *optval;
int *optlen;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>level</i>	Specifies the level for which the option is set. Only SOL_SOCKET and IPPROTO_IP are supported.
<i>optname</i>	Specifies the name of a specified socket option.
<i>optval</i>	Points to option data.
<i>optlen</i>	Points to the length of the option data.

Description: The `getsockopt()` call gets options associated with a socket. It can be called only for sockets in the AF_INET domain. This call is not supported in the AF_IUCV domain. Options can exist at multiple protocol levels; they are always present at the highest socket level.

When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level or IP level, the *level* parameter must be set to SOL_SOCKET or IPPROTO_IP, as defined in SOCKET.H. To manipulate options at any other level, such as the TCP or IP level, supply the appropriate protocol number for the protocol controlling the option. Currently, only the SOL_SOCKET and IPPROTO_IP levels are supported. The `getprotobyname()` call can be used to return the protocol number for a named protocol.

The *optval* and *optlen* parameters are used to return data used by the particular `get` command. The *optval* parameter points to a buffer that is to receive the data requested by the `get` command. The *optlen* parameter points to the size of the buffer pointed to by the *optval* parameter. It must be initially set to the size of the buffer before calling `getsockopt()`. On return it is set to the actual size of the data returned.

All of the socket level options except SO_LINGER expect *optval* to point to an integer and *optlen* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is zero, the option is disabled. The SO_LINGER option expects *optval* to point to a *linger* structure as defined in SOCKET.H. This structure is defined in the following example:

```
struct linger
{
    int    l_onoff;           /* option on/off */
    int    l_linger;        /* linger time */
};
```

The `l_onoff` field is set to zero if the SO_LINGER option is being disabled. A nonzero value enables the option. The `l_linger` field specifies the amount of time to linger on close.

The following options are recognized at the IP level (IPPROTO_IP):

Option	Description
IP_MULTICAST_TTL	Returns the IP time-to-live of outgoing multicast datagrams. The TTL value is passed back as <code>u_char</code> . This option is only supported for sockets with an address family of <code>AF_INET</code> and type of <code>SOCK_DGRAM</code> or <code>SOCK_RAW</code> .
IP_MULTICAST_LOOP	Determines whether loopback of outgoing multicast datagrams is enabled or disabled. When loopback is enabled, datagrams sent by this system should also be delivered to this system as long as it is a member of the multicast group. The loopback indicator is passed back as <code>u_char</code> . A value of 0 means loopback is disabled; a value of 1 means it is enabled. This option is only supported for sockets with an address family of <code>AF_INET</code> and type of <code>SOCK_DGRAM</code> or <code>SOCK_RAW</code> .
IP_MULTICAST_IF	Returns the interface IP address used for sending outbound multicast datagrams from socket applications. The IP Address is passed back using structure <code>in_addr</code> . This option is only supported for sockets with an address family of <code>AF_INET</code> and type of <code>SOCK_DGRAM</code> or <code>SOCK_RAW</code> .

The following options are recognized at the socket level:

Option	Description
SO_BROADCAST	Indicates the ability to broadcast messages. If this option is enabled, it allows the application to send broadcast messages over <code>s</code> , if the interface specified in the destination supports broadcasting of packets. This option has no meaning for stream sockets.
SO_ERROR	Returns any pending error on the socket and clears the error status. It can be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors (errors that are not returned explicitly by one of the socket calls).
SO_KEEPALIVE	Toggles the TCP keep-alive mechanism for a stream socket. When activated, the keep-alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error <code>ETIMEDOUT</code> .
SO_LINGER	Lingers on close if data is present. When this option is enabled and there is unsent data present when <code>close()</code> is called, the calling application is blocked during the <code>close()</code> call until the data is transmitted or the connection has timed out. If this option is disabled, the TCPIP virtual machine waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed, because the TCPIP virtual machine waits only a

getsockopt()

finite amount of time trying to send the data. The `close()` call returns without blocking the caller. This option has meaning only for stream sockets.

SO_OOBINLINE

Indicates reception of out-of-band data. When this option is enabled, it causes out-of-band data to be placed in the normal data input queue as it is received, making it available to `recv()`, `recvfrom()`, and `recvmsg()` without having to specify the `MSG_OOB` flag in those calls. When this option is disabled, it causes out-of-band data to be placed in the priority data input queue as it is received, making it available to `recv()`, `recvfrom()`, and `recvmsg()` only by specifying the `MSG_OOB` flag in those calls. This option has meaning only for stream sockets.

SO_REUSEADDR

Toggles local address reuse. When enabled, this option allows local addresses that are already in use to be bound. This alters the normal algorithm used in the `bind()` call. The system checks at connect time to be sure that no local address and port have the same foreign address and port. The error `EADDRINUSE` is returned if the association already exists.

SO_SNDBUF

Returns the size of the TCP/IP send buffer in *optval*.

SO_TYPE

Returns the type of the socket. On return, the integer pointed to by *optval* is set to one of `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value

Description

EBADF

Indicates that the *s* parameter is not a valid socket descriptor.

EFAULT

Using *optval* and *optlen* parameters would result in an attempt to access memory outside the caller's virtual storage.

EIBMIUCVERR

Indicates that an IUCV error occurred.

ENOPROTOOPT

Indicates that the *optname* parameter is unrecognized, or the *level* parameter is not `SOL_SOCKET`.

EOPNOTSUPP

Indicates the *s* parameter is not a socket descriptor that supports the *optname* parameter.

Examples: The following are examples of the `getsockopt()` call. See “`setsockopt()`” on page 76 for examples of how the `setsockopt()` call options are set.

```
int rc;
int s;
int optval;
int optlen;
struct linger l;
int getsockopt(int s, int level, int optname, char *optval, int *optlen);
```

```

:
:
/* Is out of band data in the normal input queue? */
optlen = sizeof(int);
rc = getsockopt(
    s, SOL_SOCKET, SO_OOBINLINE, (char *) &optval, &optlen);
if (rc == 0)
{
    if (optlen == sizeof(int))
    {
        if (optval)
            /* yes it is in the normal queue */
        else
            /* no it is not */
    }
}
:
:
/* Do I linger on close? */
optlen = sizeof(l);
rc = getsockopt(
    s, SOL_SOCKET, SO_LINGER, (char *) &l, &optlen);
if (rc == 0)
{
    if (optlen == sizeof(l))
    {
        if (l.l_onoff)
            /* yes I linger */
        else
            /* no I do not */
    }
}

```

See Also: getprotobyname(), setsockopt(), socket().

givesocket()

```

#include <bsdtypes.h>
#include <socket.h>

int givesocket(s, clientid)
int s,
struct clientid *clientid;

```

Parameter	Description				
<i>s</i>	Specifies the descriptor of a socket to be given to another application.				
<i>clientid</i>	Points to a <i>struct clientid</i> structure specifying the target program to whom the socket is to be given. Your program sets the fields of the structure as follows: <table border="1" data-bbox="646 1705 1455 1881"> <tbody> <tr> <td>domain</td> <td>Specifies AF_INET</td> </tr> <tr> <td>name</td> <td>Specifies the virtual storage name or virtual machine name of the target program, left-justified and padded with blanks. The target program can run in the same virtual storage or virtual machine</td> </tr> </tbody> </table>	domain	Specifies AF_INET	name	Specifies the virtual storage name or virtual machine name of the target program, left-justified and padded with blanks. The target program can run in the same virtual storage or virtual machine
domain	Specifies AF_INET				
name	Specifies the virtual storage name or virtual machine name of the target program, left-justified and padded with blanks. The target program can run in the same virtual storage or virtual machine				

givesocket()

as your program, in which case your program sets this field to its own virtual storage name or virtual machine name.

subtaskname Contains blanks.
reserved Contains binary zeros.

Any program running in the specified virtual storage or virtual machine can use `takesocket()` to take control of the socket if it knows the client ID and the socket descriptor of your program.

For backward compatibility, *clientid* may point to the *struct clientid* structure when the target program calls `getclientid()`. In this case, only the target program can take the socket.

Description: The `givesocket()` call tells TCPIP to make the specified socket available to a `takesocket()` call issued by another application running on the same host. Any connected stream socket can be given. Typically, `givesocket()` is used by a master program that obtains sockets by means of `accept()` and gives them to agent programs that handle one socket at a time.

After calling `givesocket()`, your program passes its client ID (obtained by using the `getclientid()` call) and the socket descriptor to the target program by passing it in the target program's startup parameter list. Your program then uses the `select()` call to check for a pending exception condition on the given socket, indicating that the target program has successfully called `takesocket()`. When the `select()` call indicates a pending exception condition, your program calls `close()` to close the given socket.

If your program closes the socket before a pending exception condition is indicated, the TCP connection is immediately reset, and the target program's call to `takesocket()` is unsuccessful. Calls other than `close()` issued on a given socket return a value of `-1`, with `errno` set to `EBADF`.

Return Values: The value `0` indicates success. The value `-1` indicates an error. The value of `errno` indicates a specific error.

Errno Value	Description
EBADF	Indicates that the <i>d</i> parameter is not a valid socket descriptor. The socket has already been given. The socket domain is not <code>AF_INET</code> .
EBUSY	Indicates that <code>listen()</code> has been called for the socket.
EFAULT	Using the <i>clientid</i> parameter as specified would result in an attempt to access storage outside the caller's virtual storage.
EINVAL	Indicates that the <i>clientid</i> parameter does not specify a valid client identifier.
ENOTCONN	Indicates that the socket is not connected.
EOPNOTSUPP	Indicates that the socket type is not <code>SOCK_STREAM</code> .

See Also: `getclientid()`, `takesocket()`.

htonl()

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

```
unsigned long htonl(a)
unsigned long a;
```

Parameter	Description
<i>a</i>	Specifies the unsigned long integer to be put into network byte order.

Description: The `htonl()` call translates a long integer from host byte order to network byte order.

Return Values: Returns the translated long integer.

See Also: `htons()`, `ntohs()`, `ntohl()`.

htons()

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

```
unsigned short htons(a)
unsigned short a;
```

Parameter	Description
<i>a</i>	Specifies the unsigned short integer to be put into network byte order.

Description: The `htons()` call translates a short integer from host byte order to network byte order.

Return Values: Returns the translated short integer.

See Also: `ntohs()`, `htonl()`, `ntohl()`.

ibmsflush()

For outbound sockets, the application-side datagram queue is flushed (transferred to the TCPIP address space) if any one of the following occur:

- An `ibmsflush()` is issued on the socket.
- The queue is full and another send-type socket call is issued.
- The socket is closed.
- Another `setibmssockopt()` is issued.

```
#include <manifest.h>
#include <socket.h>
```

```
int ibmsflush(int s)
```

Parameter	Description
<i>s</i>	The socket descriptor.

Related Calls: `getibmssockopt()`, `setibmssockopt()`.

inet_addr()

inet_addr()

```
#include <bsdtypes.h>

unsigned long inet_addr(cp)
char *cp;
```

Parameter	Description
<i>cp</i>	Specifies a character string in standard dotted-decimal (.) notation.

Description: The `inet_addr()` call interprets character strings representing numbers expressed in standard dotted-decimal notation and returns numbers suitable for use as an internet address.

Values specified in standard dotted-decimal notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When a four-part address is specified, each part is interpreted as a byte of data and assigned, from left to right, to one of the four bytes of an internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the two rightmost bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as `128.net.host`.

When a two-part address is specified, the last part is interpreted as a 24-bit quantity and placed in the three rightmost bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as `net.host`.

When a one-part address is specified, the value is stored directly in the network virtual storage without any rearrangement of its bytes.

Numbers supplied as address parts in standard dotted-decimal notation can be decimal, hexadecimal, or octal. Numbers are interpreted in C language syntax. A leading `0x` implies hexadecimal; a leading `0` implies octal. A number without a leading `0` implies decimal.

Return Values: The internet address is returned in network byte order.

See Also: `inet_lnaof()`, `inet_makeaddr()`, `inet_netof()`, `inet_network()`, `inet_ntoa()`.

inet_lnaof()

```
#include <bsdtypes.h>
#include <socket.h>
#include <in.h>

unsigned long inet_lnaof(in)
struct in_addr in;
```

Parameter	Description
<i>in</i>	Specifies the host internet address.

Description: The `inet_lnaof()` call breaks apart the internet host address and returns the local network address portion.

Return Values: The local network address is returned in host byte order.

See Also: `inet_addr()`, `inet_makeaddr()`, `inet_netof()`, `inet_network()`, `inet_ntoa()`.

inet_makeaddr()

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

struct in_addr
inet_makeaddr(net, lna)
unsigned long net;
unsigned long lna;
```

Parameter	Description
<i>net</i>	Specifies the network number.
<i>lna</i>	Specifies the local network address.

Description: The `inet_makeaddr()` call takes a network number and a local network address and constructs an internet address.

Return Values: The internet address is returned in network byte order.

See Also: `inet_addr()`, `inet_lnaof()`, `inet_netof()`, `inet_network()`, `inet_ntoa()`.

inet_netof()

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

unsigned long inet_netof(in)
struct in_addr in;
```

Parameter	Description
<i>in</i>	Specifies the internet address in network byte order.

Description: The `inet_netof()` call breaks apart the internet host address and returns the network number portion.

Return Values: The network number is returned in host byte order.

See Also: `inet_addr()`, `inet_lnaof()`, `inet_makeaddr()`, `inet_ntoa()`.

inet_network()

inet_network()

```
#include <bsdtypes.h>

unsigned long inet_network(cp)
char *cp;
```

Parameter	Description
<i>cp</i>	Specifies a character string in standard dotted-decimal (.) notation.

Description: The `inet_network()` call interprets character strings representing numbers expressed in standard dotted-decimal notation and returns numbers suitable for use as a network number.

Numbers supplied as address parts in standard dotted-decimal notation can be decimal, hexadecimal, or octal. Numbers are interpreted in C language syntax. A leading 0x implies hexadecimal; a leading 0 implies octal. A number without a leading 0 implies decimal.

Return Values: The network number is returned in host byte order.

See Also: `inet_addr()`, `inet_lnaof()`, `inet_makeaddr()`, `inet_ntoa()`.

inet_ntoa()

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

char *inet_ntoa(in)
struct in_addr in;
```

Parameter	Description
<i>in</i>	Specifies the host internet address.

Description: The `inet_ntoa()` call returns a pointer to a string expressed in the dotted-decimal notation. `inet_ntoa()` accepts an internet address expressed as a 32-bit quantity in network byte order and returns a string expressed in dotted-decimal notation.

Return Values: Returns a pointer to the internet address expressed in dotted-decimal notation.

See Also: `inet_addr()`, `inet_lnaof()`, `inet_makeaddr()`, `inet_network()`, `inet_ntoa()`.

ioctl()

```
#include <bsdtypes.h>
#include <ioctl.h>
#include <rtrouteh.h>
#include <if.h>

int ioctl(s, cmd, data)
int s;
unsigned long cmd;
char *data;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.

cmd Specifies the command to perform.
data Points to the data associated with *cmd*.

Description: The operating characteristics of sockets can be controlled with `ioctl()` requests. The operations to be controlled are determined by *cmd*. The *data* parameter is a pointer to data associated with the particular command, and its format depends on the command that is requested. The following are valid `ioctl()` commands:

Option	Description
FIONBIO	Sets or clears nonblocking input-output for a socket. <i>data</i> is a pointer to an integer. If the integer is 0, nonblocking input-output on the socket is cleared. Otherwise, the socket is set for nonblocking input-output.
FIONREAD	Gets the number of immediately readable bytes for the socket. <i>data</i> is a pointer to an integer. Sets the value of the integer to the number of immediately readable characters for the socket.
SIOCADDRT	Adds a routing table entry. The <i>data</i> parameter is a pointer to a <i>rtenry</i> structure, as defined in the <code>RTRROUTE.H</code> header file. The routing table entry, passed as an argument, is added to the routing tables.
SIOCATMARK	Queries whether the current location in the data input is pointing to out-of-band data. The <i>data</i> parameter is a pointer to an integer. Sets the argument to 1 if the socket points to a mark in the data stream for out-of-band data. Otherwise, sets the argument to 0.
SIOCDELRT	Deletes a routing table entry. The <i>data</i> parameter is a pointer to a <i>rtenry</i> structure, as defined in the <code>RTRROUTE.H</code> header file. If it exists, the routing table entry passed as an argument is deleted from the routing tables.
SIOCGIFADDR	Gets the network interface address. The <i>data</i> parameter is a pointer to an <i>ifreq</i> structure, as defined in the <code>IF.H</code> header file. The interface address is returned in the argument.
SIOCGIFBRDADDR	Gets the network interface broadcast address. The <i>data</i> parameter is a pointer to an <i>ifreq</i> structure, as defined in the <code>IF.H</code> header file. The interface broadcast address is returned in the argument.
SIOCGIFCONF	Gets the network interface configuration. The <i>data</i> parameter is a pointer to an <i>ifconf</i> structure, as defined in the <code>IF.H</code> header file. The interface configuration is returned in the argument.
SIOCGIFDSTADDR	Gets the network interface destination address. The <i>data</i> parameter is a pointer to an <i>ifreq</i> structure, as defined in the <code>IF.H</code> header file. The interface destination (point-to-point) address is returned in the argument.
SIOCGIFFLAGS	Gets the network interface flags. The <i>data</i> parameter is a pointer to an <i>ifreq</i> structure, as defined in the <code>IF.H</code> header file. The interface flags are returned in the argument.

ioctl()

SIOCGIFMETRIC

Gets the network interface routing metric. The *data* parameter is a pointer to an *ifreq* structure, as defined in the IF.H header file. The interface routing metric is returned in the argument.

SIOCGIFNETMASK

Gets the network interface network mask. The *data* parameter is a pointer to an *ifreq* structure, as defined in the IF.H header file. The interface network mask is returned in the argument.

SIOCSIFMETRIC

Sets the network interface routing metric. The *data* parameter is a pointer to an *ifreq* structure, as defined in the IF.H header file. Set the interface routing metric to the value passed in the argument.

SIOCSARP Sets an address translation entry. The variable *arg* is a pointer to an *arpreq* structure, as defined in IF ARP.H.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of *errno* indicates the specific error.

Errno Value	Description
-------------	-------------

EBADF	Indicates that the <i>s</i> parameter is not a valid socket descriptor.
--------------	---

EINVAL	Indicates that the request is invalid or not supported.
---------------	---

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

```
int s;
int dontblock;
int rc;
:
:
/* Place the socket into nonblocking mode */
dontblock = 1;
rc = ioctl(s, FIONBIO, (char *) &dontblock);
:
:
```

listen()

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

```
int listen(s, backlog)
int s;
int backlog;
```

Parameter	Description
-----------	-------------

<i>s</i>	Specifies the socket descriptor.
----------	----------------------------------

<i>backlog</i>	Defines the maximum length for the queue of pending connections.
----------------	--

Description: The `listen()` call applies only to stream sockets. It performs two tasks: it completes the binding necessary for a socket *s*, if `bind()` has not been called for *s*, and it creates a connection request queue of length *backlog* to queue incoming connection requests. Once full, additional connection requests are ignored.

The `listen()` call indicates a readiness to accept client connection requests. It transforms an active socket into a passive socket. Once called, *s* can never be used

listen()

as an active socket to initiate connection requests. Calling `listen()` is the third of four steps that a server performs to accept a connection. It is called after allocating a stream socket with `socket()`, and after binding a name to `s` with `bind()`. It must be called before calling `accept()`.

If the backlog is less than 0, *backlog* is set to 0. If the backlog is greater than `SOMAXCONN`, as defined in the `SOCKET.H` header file, *backlog* is set to `SOMAXCONN`.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
<code>EBADF</code>	Indicates that the <i>s</i> parameter is not a valid socket descriptor.
<code>EOPNOTSUPP</code>	Indicates that the <i>s</i> parameter is not a socket descriptor that supports the <code>listen()</code> call.

See Also: `accept()`, `bind()`, `connect()`, `socket()`.

maxdesc()

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

int maxdesc(totdesc, inetdesc)
int *totdesc;
int *inetdesc;
```

Parameter	Description
<i>totdesc</i>	Points to an integer containing a value one greater than the largest desired socket number.
<i>inetdesc</i>	Points to an integer containing a value one greater than the largest desired socket number usable for <code>AF_INET</code> sockets.

Description: The `maxdesc()` call reserves additional space in the TCPIP virtual machine to allow socket numbers to extend beyond the default range of 0 through 49. Socket numbers 0 through 2 are never assigned, so the default maximum number of sockets is 47.

Set the integer pointed to by *totdesc*, to one more than the desired maximum socket number. If your program does not use `AF_INET` sockets, set the integer pointed to by *inetdesc* to 0. If your program uses `AF_INET` sockets, set the integer pointed to by *inetdesc*, to the same value as *totdesc*. `maxdesc()` must be called before your program creates its first socket or after all sockets have been closed. Your program should use `getdtablesize()` to verify that the number of sockets was changed.

Return Values: The value 0 indicates success. Your application should check the integer pointed to by *inetdesc*. It can contain less than the original value, if there was insufficient storage available in the TCPIP virtual machine. In this case the desired number of `AF_INET` sockets are not available. The value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
-------------	-------------

maxdesc()

EFAULT	Using the <i>totdesc</i> or <i>inetdesc</i> parameters as specified would result in an attempt to access storage outside of the caller's virtual storage, or storage not modifiable by the caller.
EALREADY	Indicates that your program called <code>maxdesc()</code> to allocate a new socket descriptor set but was using one or more sockets in the current set.
EINVAL	Indicates that <i>*totdesc</i> is less than <i>*inetdesc</i> ; <i>*totdesc</i> is less than or equal to 0; or <i>*inetdesc</i> is less than 0.
ENOMEM	Indicates that your virtual machine has insufficient memory.
EIBMIUCVERR	Indicates that an IUCV error occurred.

Examples: The following are examples of the `maxdesc()` call.

```
int totdesc, inetdesc;
totdesc = 100;
inetdesc = 0;
rc = maxdesc(&totdesc, &inetdesc)
```

If successful, your application can create 97 sockets, all of type `AF_IUCV`. The socket numbers run from 3 through 99.

```
int totdesc, inetdesc;
totdesc = 100;
inetdesc = 100;
rc = maxdesc(&totdesc, &inetdesc)
```

If successful, your application can create 97 sockets, each of which can be of type `AF_INET` or `AF_IUCV`. The socket numbers run from 3 through 99.

If your application calls `maxdesc()` to define a socket descriptor set with more than 255 sockets, then the socket descriptor bit set (`fd_set`) it uses on subsequent calls to `select()` should be defined with the same *totdesc* size used on the call to `maxdesc()`. The default size of an `fd_set` (the `FD_SETSIZE`) can accommodate up to 255 sockets. You can define an `fd_set` larger than `FD_SETSIZE` at either compile time or runtime.

To define an `fd_set` at compile time, set the `FD_SETSIZE` macro in your source code to the desired value before including the `BSDTYPES.H` header file. Recompile your source code to generate `fd_sets` that can handle up to the specified number of sockets. Use the macros `FD_ZERO`, `FD_SET`, `FD_CLR`, and `FD_ISSET` in your code to manipulate your `fd_sets`.

To define an `fd_set` dynamically at runtime, include the `TYPES.H` header file in place of `BSDTYPES.H`. Use the `_GET_FDSET`, `_FREE_FDSET`, `_GET_FDSETSIZE`, `_FDSET_ZERO`, and `FD_ISSET` macros defined in `TYPES.H` to manipulate your `fd_sets`.

See Also: `select()`, `socket()`, `getdtablesize()`.

ntohl()

```
#include <bsdtypes.h>

unsigned long ntohl(a)
unsigned long a;
```

Parameter	Description
<i>a</i>	Specifies the unsigned long integer to be put into host byte order.

Description: The `ntohl()` call translates a long integer from network byte order to host byte order.

Return Values: Returns the translated long integer.

See Also: `htonl()`, `htons()`, `ntohs()`.

ntohs()

```
#include <bsdtypes.h>

unsigned short ntohs(a)
unsigned short a;
```

Parameter	Description
<i>a</i>	Specifies the unsigned short integer to be put into host byte order.

Description: The `ntohs()` call translates a short integer from network byte order to host byte order.

Return Values: Returns the translated short integer.

See Also: `ntohl()`, `htons()`, `htonl()`.

read()

```
int read(s, buf, len)
int s;
char *buf;
int len;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>buf</i>	Points to the buffer that receives the data.
<i>len</i>	Indicates the length in bytes of the buffer pointed to by the <i>buf</i> parameter.

Description: The `read()` call reads data on a socket with descriptor *s* and stores it in a buffer. The `read()` call applies only to connected sockets.

This call returns up to *len* bytes of data. If less than the number of bytes requested is available, the call returns the number currently available. If data is not available at the socket with descriptor *s*, the `read()` call waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode. See “`ioctl()`” on page 54 or “`fcntl()`” on page 31 for a description of how to set nonblocking mode.

read()

Return Values: If successful, the number of bytes copied into the buffer is returned. The value 0 indicates that the connection was closed to the remote host. The value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
EBADF	Indicates that <i>s</i> is not a valid socket descriptor.
EFAULT	Using the <i>buf</i> and <i>len</i> parameters would result in an attempt to access memory outside the caller's virtual storage.
EWOULDBLOCK	Indicates that <i>s</i> is in nonblocking mode, and no data is available to read.

See Also: `connect()`, `fcntl()`, `getsockopt()`, `ioctl()`, `readv()`, `recv()`, `recvmsg()`, `recvfrom()`, `select()`, `selectex()`, `send()`, `sendmsg()`, `sendto()`, `setsockopt()`, `socket()`, `write()`, `writv()`.

readv()

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

int readv(s, iov, iovcnt)
int s;
struct iovec *iov;
int iovcnt;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>iov</i>	Points to an <i>iovec</i> structure.
<i>iovcnt</i>	Specifies the number of buffers pointed to by the <i>iov</i> parameter.

Description: The `readv()` call reads data on a socket with descriptor *s* and stores it in a set of buffers. The data is scattered into the buffers specified by `iov[0]...iov[iovcnt-1]`. The *iovec* structure is defined in the `UIO.H` header file and contains the following fields:

Element	Description
<i>iov_base</i>	Points to the buffer.
<i>iov_len</i>	Defines the length of the buffer.

The `readv()` call applies only to connected sockets.

This call returns up to *len* bytes of data. If less than the number of bytes requested is available, the call returns the number currently available. If data is not available at the socket with descriptor *s*, the `readv()` call waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode. See “`fcntl()`” on page 31 or “`ioctl()`” on page 54 for a description of how to set nonblocking mode.

Return Values: If successful, the number of bytes read into the buffer(s) is returned. The value 0 indicates that the connection was closed to the remote host. The value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
EBADF	Indicates that <i>s</i> is not a valid socket descriptor.

EFAULT	Using <i>iov</i> and <i>iovcnt</i> would result in an attempt to access memory outside the caller's virtual storage.
EINVAL	Indicates that <i>iovcnt</i> was not valid, or one of the fields in the <i>iov</i> array was not valid.
EWOULDBLOCK	Indicates that <i>s</i> is in nonblocking mode, and no data is available to read.

See Also: connect(), fcntl(), getsockopt(), ioctl(), read(), recv(), recvmsg(), recvfrom(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), write(), writev().

recv()

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

int recv(s, buf, len, flags)
int s;
char *buf;
int len;
int flags;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>buf</i>	Points to the buffer that receives the data.
<i>len</i>	Indicates the length in bytes of the buffer pointed to by the <i>buf</i> parameter.
<i>flags</i>	Set by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator () must be used to separate them. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported in the AF_IUCV domain.
MSG_OOB	Reads any out-of-band data on the socket.
MSG_PEEK	Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.

Description: The recv() call receives data on a socket with descriptor *s* and stores it in a buffer. The recv() call applies only to connected sockets.

This call returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard excess bytes. If data is not available at the socket with descriptor *s*, the recv() call waits for a message to arrive and blocks the caller, unless the socket is in nonblocking mode. See “fcntl()” on page 31 or “ioctl()” on page 54 for a description of how to set nonblocking mode.

Return Values: If successful, the length of the message or datagram in bytes is returned. The value 0 indicates that the connection was closed to the remote host. The value -1 indicates an error. The value of errno indicates the specific error.

Errno Value	Description
--------------------	--------------------

recv()

EBADF	Indicates that <i>s</i> is not a valid socket descriptor.
EFAULT	Using the <i>buf</i> and <i>len</i> parameters would result in an attempt to access memory outside the caller's virtual storage.
EWOULDBLOCK	Indicates that <i>s</i> is in nonblocking mode, and no data is available to read.

See Also: connect(), fcntl(), getsockopt(), ioctl(), read(), readv(), recvfrom(), recvmsg(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), write(), writev().

recvfrom()

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

int recvfrom(s, buf, len, flags, name, namelen)
int s;
char *buf;
int len;
int flags;
struct sockaddr *name;
int *namelen;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>buf</i>	Points to the buffer that receives the data.
<i>len</i>	Indicates the length in bytes of the buffer pointed to by the <i>buf</i> parameter.
<i>flags</i>	Set to 0 or MSG_PEEK. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported in the AF_IUCV domain. MSG_OOB Reads any out-of-band data on the socket. MSG_PEEK Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.
<i>name</i>	Points to a <i>socket address</i> structure from which data is received. If <i>name</i> is a nonzero value, the source address is returned.
<i>namelen</i>	Indicates the size of <i>name</i> in bytes.

Description: The recvfrom() call receives data on a socket with descriptor *s* and stores it in a buffer. The recvfrom() call applies to any datagram socket, whether connected or unconnected.

If the *name* parameter is nonzero, the source address of the message is filled. The *namelen* parameter must first be initialized to the size of the buffer associated with the *name* parameter, and is then modified on return to indicate the actual size of the address stored there.

This call returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard excess bytes. If datagram packets are not available at the socket with descriptor *s*, the recvfrom()

recvfrom()

call waits for a message to arrive and blocks the caller, unless the socket is in nonblocking mode. See “fcntl()” on page 31 or “ioctl()” on page 54 for a description of how to set nonblocking mode.

Return Values: If successful, the length of the message or datagram in bytes is returned. The value 0 indicates that the connection was closed to the remote host. The value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
EBADF	Indicates that <i>s</i> is not a valid socket descriptor.
EFAULT	Using the <i>buf</i> and <i>len</i> parameters would result in an attempt to access memory outside the caller’s virtual storage.
EWOULDBLOCK	Indicates that <i>s</i> is in nonblocking mode, and no data is available to read.

See Also: `fcntl()`, `getsockopt()`, `ioctl()`, `read()`, `readv()`, `recv()`, `recvmsg()`, `select()`, `selectex()`, `send()`, `sendmsg()`, `sendto()`, `setsockopt()`, `socket()`, `write()`, `writv()`.

recvmsg()

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

int recvmsg(s, msg, flags)
int s;
struct msghdr msg[];
int flags;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>msg</i>	Specifies an array of message headers into which messages are received.
<i>flags</i>	Set by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator () must be used to separate them. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported in the AF_IUCV domain. MSG_OOB Reads any out-of-band data on the socket. MSG_PEEK Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation will see the same data.

Description: The `recvmsg()` call receives messages on a socket with descriptor *s* and stores them in an array of message headers. A message header is defined by a `msghdr`. The definition of this structure can be found in the `SOCKET.H` header file and contains the following elements:

Element	Description
<i>msg_name</i>	Specifies an optional pointer to a buffer where the sender’s address is stored.
<i>msg_namelen</i>	Indicates the size of the address buffer.

recvmsg()

<i>msg_iov</i>	Specifies an array of <i>iovec</i> buffers into which the message is placed.
<i>msg_iovlen</i>	Specifies the number of elements in the <i>msg_iov</i> array.
<i>msg_accrights</i>	Indicates the access rights received. This field is ignored.
<i>msg_accrightslen</i>	Indicates the length of access rights received. This field is ignored.

The `recvmsg()` call applies to sockets, regardless of whether they are in the connected state.

This call returns the length of the data received. If data is not available at the socket with descriptor *s*, the `recvmsg()` call waits for a message to arrive and blocks the caller, unless the socket is in nonblocking mode. See “`fcntl()`” on page 31 or “`ioctl()`” on page 54 for a description of how to set nonblocking mode.

Return Values: If successful, the length of the message in bytes is returned. The value 0 indicates that the connection was closed to the remote host. The value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
EBADF	Indicates that <i>s</i> is not a valid socket descriptor.
EFAULT	Using <i>msg</i> would result in an attempt to access memory outside the caller’s virtual storage.
EWOULDBLOCK	Indicates that <i>s</i> is in nonblocking mode, and no data is available to read.

See Also: `connect()`, `fcntl()`, `getsockopt()`, `ioctl()`, `read()`, `readv()`, `recv()`, `recvfrom()`, `select()`, `selectex()`, `send()`, `sendmsg()`, `sendto()`, `setsockopt()`, `socket()`, `write()`, `writv()`.

select()

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

int select(nfds, readfds, writefds, exceptfds, timeout)
int nfds;
fd_set *readfds;
fd_set *writefds;
fd_set *exceptfds;
struct timeval *timeout;
```

Parameter	Description
<i>nfds</i>	Specifies the greatest socket descriptor to check plus 1.
<i>readfds</i>	Points to a bit set of descriptors to check for reading.
<i>writefds</i>	Points to a bit set of descriptors to check for writing.
<i>exceptfds</i>	Points to a bit set of descriptors to be checked for exceptional pending conditions.
<i>timeout</i>	Points to the time to wait for the <code>select()</code> call to complete.

Description: The select() call monitors activity on a set of sockets to see if any of the sockets are ready for reading, writing, or have an exceptional condition pending.

If *timeout* is not a NULL pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a NULL pointer, the select call blocks until a socket becomes ready. To poll the sockets and return immediately, *timeout* should be a non-NULL pointer to a zero-valued *timeval* structure.

To completely understand the implementation of the select call, you must recognize the difference between a socket and a port. TCP/IP defines ports to represent a certain process on a certain machine. A port represents the location of one process; it does not represent a connection between processes. In the VM programming interface for TCP/IP, a socket describes an endpoint of communication. Therefore, a socket describes both a port and a machine. Like file descriptors, a socket is a small integer representing an index into a table of communication endpoints in a TCPIP virtual machine.

If your program specified *apitype=2*, only one SELECT may be outstanding on the IUCV path, and will be cancelled with a return value of zero when any subsequent function is performed on the same path, without regard to the specific socket descriptors involved.

To allow you to test more than one socket at a time, the sockets to test are placed into a bit set of type FD_SET. A bit set is a string of bits such that if X is an element of the set, the bit representing X is set to 1. If X is not an element of the set, the bit representing X is set to 0. For example, if socket 33 is an element of a bit set, then bit 33 is set to 1. If socket 33 is not an element of a bit set, then bit 33 is set to 0.

Because the bit sets contain a bit for every socket that a process can allocate, the bit sets are of constant size. The function getdtablesize() returns the number of sockets that your program can allocate. If your program needs to allocate a large number of sockets, use getdtablesize() and maxdesc() to increase the number of sockets that can be allocated. Increasing the size of the bit sets must be done at compile time. To increase the size of the bit sets before including the BSDTYPES.H header file, define FD_SETSIZE to be the largest value of any socket. The default size of FD_SETSIZE is 255 sockets.

The following macros are provided to manipulate bit sets.

Macro	Description
FD_ZERO(&fdset)	Sets all bits in the bit set <i>fdset</i> to zero. After this operation, the bit set has no sockets as elements. This macro should be called to initialize the bit set before calling FD_SET() to set a socket as a member.
FD_SET(sock, &fdset)	Sets the bit for the socket <i>sock</i> to a 1, making <i>sock</i> a member of the bit set <i>fdset</i> .
FD_CLR(sock, &fdset)	Clears the bit for the socket <i>sock</i> in bit set <i>fdset</i> . This operation sets the appropriate bit to a zero.
FD_ISSET(sock, &fdset)	Returns > 0 if <i>sock</i> is a member of the bit set <i>fdset</i> . Returns zero if <i>sock</i> is not a member of <i>fdset</i> . (This operation returns the bit representing <i>sock</i> .)

select()

A socket is ready for reading when incoming data is buffered for it or when a connection request is pending. A call to `accept()`, `read()`, `recv()`, or `recvfrom()` does not block. To test whether any sockets are ready for reading, use `FD_ZERO()` to initialize the *readfds* bit set, and invoke `FD_SET()` for each socket to test.

A socket is ready for writing if there is buffer space for outgoing data. A nonblocking stream socket in the process of connecting (`connect()` returned `EINPROGRESS`) is selected for write when the `connect()` completes. A call to `write()`, `send()`, or `sendto()` does not block providing that the amount of data is less than the amount of buffer space. If a socket is selected for write, the amount of available buffer space is guaranteed to be at least as large as the size returned from using `SO_SNDBUF` with `getsockopt()`. To test whether any sockets are ready for writing, initialize *writfds* with `FD_ZERO()`, and use `FD_SET()` for each socket to test.

The `select()` call checks for a pending exception condition on the given socket, indicating that the target program has successfully called `takesocket()`. When `select()` indicates a pending exception condition, your program calls `close()` to close the given socket. A socket has exceptional conditions pending if it has received out-of-band data. A stream socket that was given using `givesocket()` is selected for exception when another application successfully takes the socket using `takesocket()`.

The programmer can pass `NULL` for any bit sets that do not have any sockets to test. For example, if a program need only check a socket for reading, it can pass `NULL` for both *writfds* and *exceptfds*.

Because the sets of sockets passed to `select()` are bit sets, the `select()` call must test each bit in each bit set before polling the socket for its status. For efficiency, the *nfsd* parameter specifies the largest socket that is passed in any of the bit sets. The `select` call then tests only sockets in the range 0 to *nfsd*-1. *nfsd* can be the result of `getdtablesize()`; but if the application only has two sockets and *nfsd* is the result of `getdtablesize()`, `select()` is going to test every bit in each bit set.

Return Values: The total number of ready sockets in all bit sets is returned. The value `-1` indicates `errno` should be checked for an error. The value zero indicates an expired time limit. If the return value is greater than zero, the sockets that are ready in each bit set are set 1. Sockets in each bit set that are not ready are set to zero. Use the macro `FD_ISSET()` with each socket to test its status.

Errno Value	Description
EBADF	Indicates that one of the bit sets specified an invalid socket. <code>FD_ZERO()</code> was probably not called to clear the bit set before the sockets were set.
EFAULT	Indicates that one of the bit sets pointed to a value outside the caller's virtual storage.
EINVAL	Indicates that one of the fields in the <code>timeval</code> structure is invalid.

Examples: In the following example, `select()` is used to poll three sockets: one for reading, one for writing, and one for exceptional conditions.

```
/* sock_stats(r, w, e) - Print the status of sockets r, w, and e. */
int sock_stats(r, w, e)
int r, w, e;
{
    fd_set reading, writing, except;
    struct timeval timeout;
```

select()

```
int rc, max_sock;

/* initialize the bit sets */
FD_ZERO( &reading );
FD_ZERO( &writing );
FD_ZERO( &except );

/* add r, w, and e to the appropriate bit set */
FD_SET( r, &reading );
FD_SET( w, &writing );
FD_SET( e, &except );

/* for efficiency, what's the maximum socket number? */
max_sock = MAX( r, w );
max_sock = MAX( max_sock, e );

/* make select poll by sending a 0 timeval */
memset( &timeout, 0, sizeof(timeout) );

/* poll */
rc = select( max_sock, &reading, &writing, &except, &timeout );

if ( rc < 0 ) {
    /* an error occurred during the select() */
    tcperror( "select" );
}
else if ( rc == 0 ) {
    /* none of the sockets were ready in our little poll */
    printf( "nobody is home.\n" );
} else {
    /* at least one of the sockets is ready */
    printf("r is %s\n", FD_ISSET(r,&reading) ? "READY" : "NOT READY");
    printf("w is %s\n", FD_ISSET(w,&writing) ? "READY" : "NOT READY");
    printf("e is %s\n", FD_ISSET(e,&except) ? "READY" : "NOT READY");
}
}
```

See Also: getdtablesize(), maxdesc(), selectex().

selectex()

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

int selectex(nfds, readfds, writefds, exceptfds, timeout, ecbptr)
int nfds;
fd_set *readfds;
fd_set *writefds;
fd_set *exceptfds;
struct timeval *timeout;
int *ecbptr;
```

Parameter	Description
<i>nfds</i>	Specifies the greatest socket descriptor to check plus 1.
<i>readfds</i>	Points to a bit set of descriptors to check for reading.
<i>writefds</i>	Points to a bit set of descriptors to check for writing.
<i>exceptfds</i>	Points to a bit set of descriptors to be checked for exceptional pending conditions.
<i>timeout</i>	Points to the time to wait for the selectex() call to complete.

selectex()

ecbptr Points to the event control block (ECB).

Description: The `selectex()` call monitors activity on a set of different sockets until a time-out expires, to see if any sockets are ready for reading or writing, or if any exceptional conditions are pending. Bit mask is made up of an array of integers. Macros are provided to manipulate the bit masks. Refer to the `select()` call for a complete description of the macros.

Return Values: The total number of ready sockets (in all bit masks) is returned. The value `-1` indicates an error. The value `0` indicates an expired time limit. If the return value is greater than `0`, the socket descriptors in each bit mask that are ready are set to `1`. All others are set to `0`.

Errno Value Description

EBADF Indicates that one of the descriptor sets specified an invalid descriptor.

EFAULT Indicates that one of the parameters pointed to a value outside the caller's virtual storage.

EINVAL Indicates that one of the fields in the *timeval* structure is invalid.

See Also: `accept()`, `connect()`, `getdtablesize()`, `recv()`, `send()`, `select()`.

send()

```
#include <bsdtypes.h>
#include <socket.h>
int send(s, msg, len, flags)
int s;
char *msg;
int len;
int flags;
```

Parameter Description

s Specifies the socket descriptor.

msg Points to the buffer containing the message to transmit.

len Specifies the length of the message pointed to by the *buf* parameter.

flags Set by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (`|`) must be used to separate them. Setting this parameter is supported only for sockets in the `AF_INET` domain. Setting these flags is not supported in the `AF_IUCV` domain.

MSG_OOB

Sends out-of-band data on sockets that support this notion. Only `SOCK_STREAM` sockets created in the `AF_INET` address family support out-of-band data.

MSG_DONTRROUTE

Indicates that the `SO_DONTRROUTE` option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.

Description: The `send()` call sends packets on the socket with descriptor `s`. The `send()` call applies to all connected sockets.

If buffer space is not available at the socket to hold the message to be transmitted, the `send()` call normally blocks, unless the socket is placed in nonblocking Input/Output (I/O) mode. See “`ioctl()`” on page 54 or “`fcntl()`” on page 31 for a description of how to set nonblocking mode. The `select()` call can be used to determine when it is possible to send more data.

Return Values: No indication of failure to deliver is implicit in a `send()` routine. The value `-1` indicates locally detected errors. The value of `errno` indicates the specific error.

Errno Value	Description
EBADF	Indicates that <code>s</code> is not a valid socket descriptor.
EFAULT	Using the <code>buf</code> and <code>len</code> parameters would result in an attempt to access memory outside the caller’s virtual storage.
ENOBUFS	Indicates that no buffer space is available to send the message.
EWOULDBLOCK	Indicates that <code>s</code> is in nonblocking mode, and no data is available to read.

See Also: `connect()`, `fcntl()`, `getsockopt()`, `ioctl()`, `read()`, `readv()`, `recv()`, `recvfrom()`, `recvmsg()`, `select()`, `selectex()`, `sendmsg()`, `sendto()`, `socket()`, `write()`, `writenv()`.

sendmsg()

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

int sendmsg(s, msg, flags)
int s;
struct msghdr msg[];
int flags;
```

Parameter	Description
<code>s</code>	Specifies the socket descriptor.
<code>msg</code>	Specifies an array of message headers from which messages are sent.
<code>flags</code>	Set by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (<code> </code>) must be used to separate them. Setting this parameter is supported only for sockets in the <code>AF_INET</code> domain. Setting these flags is not supported in the <code>AF_IUCV</code> domain.
MSG_OOB	Sends out-of-band data on the socket.
MSG_DONTROUTE	Indicates that the <code>SO_DONTROUTE</code> option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.

sendmsg()

Description: The `sendmsg()` call sends messages on a socket with descriptor `s` passed in an array of message headers. A message header is defined by a `msg_hdr`. The definition of this structure can be found in the `SOCKET.H` header file and contains the following elements:

Element	Description
<code>msg_name</code>	Specifies the optional pointer to the buffer containing the recipient's address.
<code>msg_namelen</code>	Indicates the size of the address buffer.
<code>msg_iov</code>	Specifies an array of <code>iovec</code> buffers containing the message.
<code>msg_iovlen</code>	Specifies the number of elements in the <code>msg_iov</code> array.
<code>msg_accrights</code>	Indicates the access rights sent. This field is ignored.
<code>msg_accrightslen</code>	Indicates the length of the access rights sent. This field is ignored.

The `sendmsg()` call applies to sockets regardless of whether they are in the connected state.

This call returns the length of the data sent. If the socket with descriptor `s` is not ready for sending data, the `sendmsg()` call waits for the ability to send data and blocks the caller. `s` is in nonblocking mode unless the socket is in nonblocking mode. See “`fcntl()`” on page 31 or “`ioctl()`” on page 54 for a description of how to set nonblocking mode.

Return Values: If successful, the length of the message in bytes is returned. The value `-1` indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
EBADF	Indicates that <code>s</code> is not a valid socket descriptor.
EFAULT	Using <code>msg</code> would result in an attempt to access memory outside the caller's virtual storage.
EINVAL	Indicates that <code>tolen</code> is not the size of a valid address for the specified address family.
EMSGSIZE	Indicates that the message was too big to be sent as a single datagram. The default is 8192, and the maximum is 32 767.
ENOBUFS	Indicates that no buffer space is available to send the message.
EWOULDBLOCK	Indicates that <code>s</code> is in nonblocking mode, and no data is available to read.

See Also: `connect()`, `fcntl()`, `getsockopt()`, `ioctl()`, `read()`, `readv()`, `recv()`, `recvfrom()`, `recvmsg()`, `select()`, `selectex()`, `send()`, `sendto()`, `setsockopt()`, `socket()`, `write()`, `writv()`.

sendto()


```
#include <bsdtypes.h>
#include <socket.h>
int sendto(s, msg, len, flags, to, tolen)
int s;
char *msg;
int len;
int flags;
struct sockaddr *to;
int tolen;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>msg</i>	Points to the buffer containing the message to transmit.
<i>len</i>	Specifies the length of the message in the buffer pointed to by the <i>msg</i> parameter.
<i>flags</i>	Set to 0 or MSG_DONTROUTE. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported in the AF_IUCV domain.
	MSG_DONTROUTE The SO_DONTROUTE option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.
<i>to</i>	Specifies the address of the target.
<i>tolen</i>	Indicates the size of the address pointed to by the <i>to</i> pointer.

Description: The sendto() call sends packets on the socket with descriptor *s*. The sendto() call applies to any datagram socket, whether connected or unconnected.

Return Values: If successful, the number of characters sent is returned. The value -1 indicates an error. The value of errno indicates the specific error.

No indication of failure to deliver is implied in the return value of this call when used with datagram sockets.

Errno Value	Description
EBADF	Indicates that <i>s</i> is not a valid socket descriptor.
EFAULT	Using the <i>buf</i> and <i>len</i> parameters would result in an attempt to access memory outside the caller's virtual storage.
EINVAL	Indicates that <i>tolen</i> is not the size of a valid address for the specified address family.
EMSGSIZE	Indicates that the message was too big to be sent as a single datagram. The default is 8192, and the maximum is 32,767.
ENOBUFS	Indicates that no buffer space is available to send the message.
EWOULDBLOCK	Indicates that <i>s</i> is in nonblocking mode, and no data is available to read.

See Also: read(), readv(), recv(), recvfrom(), recvmsg(), send(), select(), selectex(), sendmsg(), socket() write(), writev().

sethostent()

sethostent()

```
int sethostent(stayopen)
int stayopen;
```

The `sethostent()` call has no parameters.

Description: The `sethostent()` call opens and rewinds the HOSTS SITEINFO file. The HOSTS file contains information about known hosts. If the *stayopen* flag is nonzero, the HOSTS file remains open after each call.

The `sethostent()` call is available only if `RESOLVE_VIA_LOOKUP` is defined before the `MANIFEST.H` header file is included.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of `errno` indicates the specific error.

See Also: `endhostent()`, `gethostbyaddr()`, `gethostbyname()`, `gethostent()`.

setibmssockopt()

`Setibmssockopt()` controls socket options specific to the IBM TCP/IP implementation.

```
#include <manifest.h>
#include <socket.h>
```

```
int setibmssockopt(int s, int level, int optname, char *optval, int optlen)
```

Parameter	Description
<i>s</i>	The socket descriptor.
<i>level</i>	The level for which the option is being set. Only <code>SOL_SOCKET</code> is supported.
<i>optname</i>	The name of a specified socket option: <ul style="list-style-type: none">• <code>SO_BULKMODE</code>• <code>SO_NONBLOCKLOCAL</code>• <code>SO_IGNOREINCOMINGPUSH</code>
<i>optval</i>	The pointer to option data.
<i>optlen</i>	The length of the option data.

SO_BULKMODE

Use `setibmssockopt()` with the *optname* `SO_BULKMODE` to place the UDP socket *s* in bulk mode. The bulk mode socket option enables an application to queue multiple datagrams, sending all of the datagrams in one large buffer. This reduces the CPU consumption for each datagram.

For inbound datagrams, a queue of pending datagrams is maintained on the application side of the interface. As the application performs receive calls, it draws from that queue. When the queue is empty, `recv()` call requests that TCPIP pass over whatever datagrams are pending in one transaction. The oldest datagram is returned to the caller, and the application-side queue is replenished.

For outbound datagrams, a separate queue is kept on the application side of the interface. When the application performs send calls, the datagram is queued. When there are no more datagrams to send out, `ibmsflush()` is called to send the queued datagrams to the TCPIP address space in one transaction.

Note: Bulk mode is valid for all send calls--`send()`, `sendmsg()`, `sendto()`, and `write()`--except for the `writev()` call. Bulk mode is valid for all read calls--`read()`, `recv()`, `recvmsg()`, and `recvfrom()`--except for the `readv()` call. However, for the `sendmsg()` and `recvmsg()` calls, only one datagram is processed per call.

Use of bulk mode can improve program performance. Performance improvement depends on the system load and the arrival pattern of the datagram messages at the socket. As system load increases, the reduction in CPU use because of bulk mode should also increase. When datagrams for the socket are processed, there should be an even greater reduction in CPU usage.

When *optname* is `SO_BULKMODE`, *optval* must point to an `ibm_bulkmode_struct` with values set as follows:

Element	Description
b_onoff	1 means bulk mode is on; 0 means bulk mode is off.
b_max_receive_queue_size	The maximum receiving queue size in bytes. Specifying a value of 0 prevents queuing for inbound datagrams.
b_max_send_queue_size	The maximum sending queue size in bytes. Specifying a value of 0 prevents queuing for outbound datagrams.
b_teststor	If this element is nonzero, the message buffer address and the message buffer are checked for addressability during each socket call. <code>errno</code> is set to <code>EFAULT</code> if there is an addressing exception. If this element is zero, checking is not performed.
b_move_data	Must be set to 1.

The socket calls that receive datagrams in your program (`recvfrom()`, `recv()`, `read()`, or `recvmsg()`) do not need to be changed. If you are using a queue for sending datagrams (by specifying a nonzero value for `b_max_send_queue_size` above), you should code `ibmsflush()` to flush the socket at appropriate points, such as after you send a burst of datagrams that is normally followed by a pause.

SO_NONBLOCKLOCAL

The option is meaningful only for sockets that have been enabled for bulk mode using the `setibmssockopt()` call with `SO_BULKMODE`. In nonblocking mode, when the application-side queue is empty, the socket library returns `-1` on a receive and sets `errno` to `EWOULDBLOCK`. *optval* should point to an integer. If *optval* points to 1, the socket is placed in nonblocking mode. If *optval* points to 0, the socket is placed in blocking mode.

Before the application calls `SO_NONBLOCKLOCAL`, the socket is in blocking mode.

setibmssockopt()

SO_IGNOREINCOMINGPUSH

This option is meaningful only for stream sockets connections established through an offload box. *optval* must point to an integer. If *optval* points to 1, the option is set. If *optval* points to 0, the option is off.

The SO_IGNOREINCOMINGPUSH option causes a receive call to return when:

- The requested length is reached.
- The internal TCPIP length is reached.
- The peer application closes the connection.

The amount of data returned for each call is maximized and the amount of CPU time consumed by your program and TCPIP can be reduced.

This option is not appropriate for your operation if your program 9 For example, this option is appropriate for an FTP data connection, but not for a Telnet connection.

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

```
#include <manifest.h>
#include <socket.h>
#include <tcperror.h>

{ struct ibm_bulkmode_struct bulkstr;
  int optlen, rc;

  optlen = sizeof(bulkstr);
  rc = getibmssockopt(s, SOL_SOCKET, SO_BULKMODE, (char *), &bulkstr, &optlen);
  if (rc < 0) {
    tcperror("on getibmssockopt()");
    exit(1);
  }
  fprintf(stream,"%d byte buffer available for outbound queue.\n",
    bulkstr.b_max_send_queue_size_avail);

  bulkstr.b_max_send_queue_size=bulkstr.b_max_send_queue_size_avail;
  bulkstr.b_onoff = 1;
  bulkstr.b_teststor = 0;
  bulkstr.b_move_data = 1;
  bulkstr.b_max_receive_queue_size = 65536;
  rc = setibmssockopt(s, SOL_SOCKET, SO_BULKMODE, (char *), &bulkstr, optlen);
  if (rc < 0) {
    tcperror("on setibmssockopt()");
    exit(1);
  }
}
```

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of *errno* indicates the specific error.

Errno Value	Description
EBADF	The <i>s</i> parameter is not a valid socket descriptor.
EFAULT	Using <i>optval</i> and <i>optlen</i> parameters would result in an attempt to access storage outside the caller's address space.
EIBMIUCVERR	An IUCV error occurred.
ENOPROTOOPT	The <i>optname</i> parameter is unrecognized, or the level parameter is not SOL_SOCKET.

Related Calls: getibmssockopt(), getsockopt(), ibmsflush(), setsockopt().

setnetent()

```
int setnetent(stayopen)
int stayopen;
```

The setnetent() call has no parameters.

Description: The setnetent() call opens and rewinds the HOSTS SITEINFO file. The HOSTS SITEINFO file contains information about known networks. If the *stayopen* flag is nonzero, the HOSTS SITEINFO file remains open after each call to setnetent().

Note: HOSTS LOCAL, HOSTS ADDRINFO, and HOSTS SITEINFO are shown in *TCP/IP Planning and Customization*.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of errno indicates the specific error.

See Also: endnetent(), getnetbyaddr(), getnetbyname(), getnetent().

setprotoent()

```
int setprotoent(stayopen)
int stayopen;
```

The setprotoent() call has no parameters.

Description: The setprotoent() call opens and rewinds the ETC PROTO file. If the *stayopen* flag is nonzero, the ETC PROTO file remains open after each call.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of errno indicates the specific error.

See Also: endprotoent(), getprotobyname(), getprotobynumber(), getprotoent().

setservent()

```
int setservent(stayopen)
int stayopen;
```

The setservent() call has no parameters.

Description: The setservent() call opens and rewinds the ETC SERVICES file. For more information about the ETC SERVICES file, see “Appendix C. Well-Known Port Assignments” on page 413. If the *stayopen* flag is nonzero, the ETC SERVICES file remains open after each call.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of errno indicates the specific error.

See Also: endservent(), getservbyname(), getservent().

setsockopt()

setsockopt()

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

int setsockopt(s, level, optname, optval, optlen)
int s;
int level;
int optname;
char *optval;
int optlen;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>level</i>	Specifies the level for which the option is set. Only SOL_SOCKET and IPPROTO_IP are supported.
<i>optname</i>	Specifies the name of a specified socket option.
<i>optval</i>	Points to option data.
<i>optlen</i>	Indicates the length of the option data.

Description: The `setsockopt()` call sets options associated with a socket. It can be called only for sockets in the AF_INET domain. This call is not supported in the AF_IUCV domain. Options can exist at multiple protocol levels; they are always present at the highest socket level.

When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, the *level* parameter must be set to SOL_SOCKET or IPPROTO_IP, as defined in SOCKET.H. To manipulate options at any other level, such as the TCP or IP level, supply the appropriate protocol number for the protocol controlling the option. Currently, only the SOL_SOCKET and IPPROTO_IP levels are supported. The `getprotobyname()` call can be used to return the protocol number for a named protocol.

The *optval* and *optlen* parameters are used to pass data used by the particular set command. The *optval* parameter points to a buffer containing the data needed by the set command. The *optval* is optional and can be set to the NULL pointer, if data is not needed by the command. The *optlen* parameter must be set to the size of the data pointed to by *optval*.

All of the socket level options except SO_LINGER expect *optval* to point to an integer and *optlen* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is zero, the option is disabled. The SO_LINGER option expects *optval* to point to a *linger* structure, as defined in SOCKET.H. This structure is defined in the following example:

```
struct linger
{
    int    l_onoff;           /* option on/off */
    int    l_linger;         /* linger time */
};
```

The `l_onoff` field is set to zero if the SO_LINGER option is being disabled. A nonzero value enables the option. The `l_linger` field specifies the amount of time to linger on close. The units of `l_linger` are seconds.

The following options are recognized at the IP level (IPPROTO_IP):

Option	Description
--------	-------------

IP_MULTICAST_TTL

Sets the IP time-to-live of outgoing multicast datagrams. The default value is 1 (multicast only to directly attached network). The TTL value is passed in as a `u_char`. This option is only supported for sockets with an address family of `AF_INET` and type of `SOCK_DGRAM` or `SOCK_RAW`.

IP_MULTICAST_LOOP

Enables/disables loopback of outgoing multicast datagrams. The default is enable. When loopback is enabled, multicast applications that have joined the outgoing multicast group can receive a copy of the multicast datagram destined for that address/port pair. The loopback indicator is passed as `u_char`. Specify a value of 0 to disable loopback; specify a value of 1 to enable loopback. This option is only supported for sockets with an address family of `AF_INET` and type of `SOCK_DGRAM` or `SOCK_RAW`.

IP_MULTICAST_IF

Sets the interface for sending outbound multicast datagrams from this socket application. Multicast datagrams will be transmitted on only the specified interface. The default is `INADDR_ANY` which means all interfaces. The IP Address of the interface is passed using structure `in_addr`. An address of `INADDR_ANY` removes the previous selection. This option is only supported for sockets with an address family of `AF_INET` and type of `SOCK_DGRAM` or `SOCK_RAW`.

IP_ADD_MEMBERSHIP

Joins a multicast group on a specific interface (an interface has to be specified with this option). Only applications that want to receive multicast datagrams need to join multicast groups. Applications that only transmit multicast datagrams do not need to join multicast groups. A single socket can join up to 20 groups. The multicast IP address and the interface IP address will be passed in `ip_mreq` structure defined in `IN.H`.

```
struct ip_mreq
{
    struct in_addr imr_multiaddr; /* IP multicast addr of group */
    struct in_addr imr_interface; /* local IP addr of interface */
};
```

This option is only supported for sockets with an address family of `AF_INET` and type of `SOCK_DGRAM` or `SOCK_RAW`.

IP_DROP_MEMBERSHIP

Leaves a multicast group on a specific interface. The multicast IP address and the interface IP address will be passed in the `ip_mreq` structure defined in `IN.H`.

```
struct ip_mreq
{
    struct in_addr imr_multiaddr; /* IP multicast addr of group */
    struct in_addr imr_interface; /* local IP addr of interface */
};
```

This option is only supported for sockets with an address family of `AF_INET` and type of `SOCK_DGRAM` or `SOCK_RAW`.

setsockopt()

The following options are recognized at the socket level:

Option	Description
SO_KEEPAIVE	Toggles the TCP keep-alive mechanism for a stream socket. When activated, the keep-alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT.
SO_BROADCAST	Toggles the ability to broadcast messages. If this option is enabled, it allows the application to send broadcast messages over <i>s</i> , if the interface specified in the destination supports broadcasting of packets. This option has no meaning for stream sockets.
SO_LINGER	Lingers on close if data is present. When this option is enabled and there is unsent data present when <code>close()</code> is called, the calling application is blocked during the <code>close()</code> call until the data is transmitted or the connection has timed out. If this option is disabled, the TCPIP virtual machine waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed, because the TCPIP virtual machine waits only a finite amount of time trying to send the data. The <code>close()</code> call returns without blocking the caller. This option has meaning only for stream sockets.
SO_OOINLINE	Toggles the reception of out-of-band data. When this option is enabled, it causes out-of-band data to be placed in the normal data input queue as it is received, making it available to <code>recv()</code> , <code>recvfrom()</code> , and <code>recvmsg()</code> without having to specify the <code>MSG_OOB</code> flag in those calls. When this option is disabled, it causes out-of-band data to be placed in the priority data input queue as it is received, making it available to <code>recv()</code> , <code>recvfrom()</code> , and <code>recvmsg()</code> only by specifying the <code>MSG_OOB</code> flag in those calls. This option has meaning only for stream sockets.
SO_REUSEADDR	Toggles local address reuse. When enabled, this option allows local addresses that are already in use to be bound. This alters the normal algorithm used in the <code>bind()</code> call. The system checks at connect time to be sure that no local address and port have the same foreign address and port. The error <code>EADDRINUSE</code> is returned if the association already exists.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
-------------	-------------

EADDRINUSE	Indicates that the socket has already joined the multicast group on the selected interface.
EADDRNOTAVAIL	Indicates that the interface IP address cannot be found or does not support multicasting.
EBADF	Indicates that the <i>s</i> parameter is not a valid socket descriptor.
EFAULT	Indicates that using <i>optval</i> and <i>optlen</i> parameters would result in an attempt to access memory outside the caller's virtual storage.
EIBMIUCVERR	Indicates that an IUCV error occurred.
EINVAL	Indicates that the multicast IP address specified is not a valid Class D IP address (designated by the high order four bits being set to B'1110').
EOPNOTSUPP	Indicates that the <i>s</i> parameter is not a socket descriptor that supports the <i>optname</i> parameter.
ETOOMANYREFS	Indicates that the socket has already joined the maximum number of multicast groups (IP_MAX_MEMBERSHIPS=20).

Examples: The following are examples of the setsockopt() call. See “getsockopt()” on page 45 for examples of how the getsockopt() options set are queried.

```
int rc;
int s;
int optval;
struct linger l;
int setsockopt(int s, int level, int optname, char *optval, int optlen);
:
:
/* I want out of band data in the normal input queue */
optval = 1;
rc = setsockopt(s, SOL_SOCKET, SO_OOBINLINE, (char *) &optval
, sizeof(int));
:
:
/* I want to linger on close */
l.l_onoff = 1;
l.l_linger = 100;
rc = setsockopt(s, SOL_SOCKET, SO_LINGER, (char *) &l, sizeof(l));
```

See Also: fcntl(), getprotobyname(), getsockopt(), ioctl(), socket().

sockdb_sock_debug()

The sock_debug() call controls the socket library tracing facility. If enabled, all socket calls and interrupts are traced, with output directed to the virtual console.

If you include the SOCKDEBUG statement in the TCPIP DATA file, tracing is active by default. A call to sock_debug() is not required.

```
#include <manifest.h>
#include <socket.h>

void sock_debug(int onoff)
```

sockdb_sock_debug()

Parameter	Description
<i>onoff</i>	TRUE tracing is enabled. FALSE tracing is disabled

sock_debug_bulk_perf0 ()

The `sock_debug_bulk_perf0()` call controls the generation of a performance report when any socket configured for bulk mode is closed. The report is directed to the virtual console.

If you include the `SOCKDEBUGBULKPERF0` statement in the TCPIP DATA file, performance reports are enabled by default. A call to `sock_debug_bulk_perf0()` is not required.

```
#include <manifest.h>
#include <socket.h>

void sock_debug_bulk_perf0(int onoff)
```

Parameter	Description
<i>onoff</i>	TRUE A performance report is generated. FALSE A performance report is not generated.

If this parameter is omitted a performance report is not produced.

Example: The following is an example of the `sock_debug_bulk_perf0()` `socket()` call report.

```
Bulkmode performance for socket 3:
  Doing TESTSTOR (ie. testing addressability of buffers, etc.)
  Received 14601460 bytes,
  10001 datagrams, 846 IUCV's 11.8
  datagrams>IUCV.
```

In this example, 3 is the socket descriptor for the socket running in bulk mode. Doing TESTSTOR indicates that the library was checking for addressing errors on socket calls, and 14,601,460 bytes of data were received in 10,001 datagrams. 846 calls were done to read the datagrams on the socket for an average of 11.8 datagrams for each IUCV.

sock_do_bulkmode()

The `sock_do_bulkmode()` controls the initial BULKMODE setting for datagram sockets.

If you include the `SOCKBULKMODE` statement in the TCPIP DATA file, bulk mode is enabled by default. A call to `sock_do_bulkmode()` is not required.

For a complete description of bulk mode, see `setibmssockopt()`.

sock_do_bulkmode()

```
#include <manifest.h>
#include <socket.h>

void sock_do_bulkmode(int onoff)
```

Parameter	Description
<i>onoff</i>	TRUE All sockets with type SOCK_DGRAM are created with bulk mode enabled. FALSE All sockets with type SOCK_DGRAM are created with bulk mode disabled. A call to setibmssockopt() is required to enable bulk mode for a specific socket.

sock_do_teststor()

The sock_do_teststor() call is used to check for calls that attempt to access storage outside the caller's address space.

If you include the statement SOCKTESTSTOR in the TCPIP DATA file, address checking is active by default. A call to sock_do_teststor() is not necessary.

```
#include <manifest.h>
#include <socket.h>

void sock_do_teststor(int onoff)
```

Parameter	Description
<i>onoff</i>	TRUE The address of the message buffer and the message buffer are checked for addressability for each socket call. The error condition, EFAULT is set if there is an addressing problem. FALSE Address checking is not done. If an error occurs when <i>onoff</i> is 0, normal runtime error handling reports the exception condition.

shutdown()

```
int shutdown(s, how)
int s;
int how;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>how</i>	Specifies the condition of the shutdown. The values 0, 1, or 2 set the condition.

Description: The shutdown() call shuts down all or part of a duplex connection. The *how* parameter sets the condition for shutting down the connection to socket *s*.

how can have a value of 0, 1, or 2, where:

shutdown()

- 0 ends communication from socket *s*.
- 1 ends communication to socket *s*.
- 2 ends communication both to and from socket *s*.

Return Values: The value 0 indicates success; the value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
EBADF	Indicates that <i>s</i> is not a valid socket descriptor.
EINVAL	Indicates that the <i>how</i> parameter was not set to one of the valid values. Valid values are 0, 1, and 2.

See Also: `accept()`, `close()`, `connect()`, `socket()`.

socket()

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

int socket(domain, type, protocol)
int domain;
int type;
int protocol;
```

Parameter	Description
<i>domain</i>	Specifies the address domain requested. It is either <code>AF_INET</code> or <code>AF_IUCV</code> .
<i>type</i>	Specifies the type of socket created, either <code>SOCK_STREAM</code> , <code>SOCK_DGRAM</code> , or <code>SOCK_RAW</code> .
<i>protocol</i>	Specifies the protocol requested. Some possible values are 0, <code>IPPROTO_UDP</code> , or <code>IPPROTO_TCP</code> .

Description: The `socket()` call creates an endpoint for communication and returns a socket descriptor representing the endpoint. Different types of sockets provide different communication services.

The *domain* parameter specifies a communications domain within which communication is to take place. This parameter selects the address family (format of addresses within a domain) which is used. The families supported are `AF_INET`, which is the internet domain and `AF_IUCV`, which is the IUCV domain. These constants are defined in the `SOCKET.H` header file.

The *type* parameter specifies the type of socket created. The type is analogous with the semantics of the communication requested. These socket type constants are defined in the `SOCKET.H` header file. The types supported are:

Socket Type	Description
<code>SOCK_STREAM</code>	Provides sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data. This type is supported in both the <code>AF_INET</code> and <code>AF_IUCV</code> domains.
<code>SOCK_DGRAM</code>	Provides datagrams, which are connectionless

messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times. This type is supported in only the AF_INET domain.

SOCK_RAW

Provides the interface to internal protocols (such as IP and ICMP). This type is supported in only the AF_INET domain.

The *protocol* parameter specifies a particular protocol to be used with the socket. In most cases, a single protocol exists to support a particular type of socket in a particular addressing family (not true with raw sockets). If the *protocol* field is set to 0, the system selects the default protocol number for the domain and socket type requested. Protocol numbers are found in the ETC PROTO file. Alternatively, the *getprotobyname* call can be used to get the protocol number for a protocol with a known name. The *protocol* field must be set to 0, if the *domain* parameter is set to AF_IUCV. Currently, *protocol* defaults are TCP for stream sockets and UDP for datagram sockets. There is no default for raw sockets.

SOCK_STREAM

Stream sockets model duplex byte streams. They provide reliable, flow-controlled connections between peer applications. Stream sockets are either active or passive. Active sockets are used by clients who initiate connection requests with connect(). By default, socket() creates active sockets. Passive sockets are used by servers to accept connection requests with the connect() call. An active socket is transformed into a passive socket by binding a name to the socket with the bind() call and by indicating a willingness to accept connections with the listen() call. Once a socket is passive, it cannot be used to initiate connection requests.

In the AF_INET domain, the bind() call applied to a stream socket lets the application specify the networks from which it is willing to accept connection requests. The application can fully specify the network interface by setting the *internet address* field in the *address* structure to the internet address of a network interface. Alternatively, the application can use a *wildcard* to specify that it wants to receive connection requests from any network. This is done by setting the *internet address* field in the *address* structure to the constant INADDR_ANY as defined in the SOCKET.H header file.

Once a connection has been established between stream sockets, any of the data transfer calls can be used (read(), write(), send(), recv(), readv(), writev(), sendto(), recvfrom(), sendmsg(), recvmsg()). Usually, the read-write or send-recv pairs are used for sending data on stream sockets. If out-of-band data is to be exchanged, the send-recv pair is normally used.

SOCK_DGRAM

Datagram sockets model datagrams. They provide connectionless message exchange with no guarantees on reliability. Messages sent have a maximum size. Datagram sockets are not supported in the AF_IUCV domain.

There is no active or passive analogy to stream sockets with datagram sockets. Servers must still call bind() to name a socket and to specify from which network interfaces it wishes to receive packets. Wildcard addressing, as described for stream sockets, applies for datagram sockets also. Because datagram sockets are connectionless, the listen() call has no meaning for them and must not be used with them.

socket()

Once an application has received a datagram socket it can exchange datagrams using the `sendto()` and `recvfrom()` or `sendmsg()` and `recvmsg()` calls. If the application goes one step further by calling `connect()` and fully specifying the name of the peer with which all messages will be exchanged, then the other data transfer calls `read()`, `write()`, `readv()`, `writev()`, `send()`, `recv()` can also be used. For more information on placing a socket into the connected state see “`connect()`” on page 27 .

Datagram sockets allow messages to be broadcast to multiple recipients. Setting the destination address to be a broadcast address is network interface dependent (depends on class of address and whether sub-nets are being used). The constant `INADDR_BROADCAST`, defined in `socket.h`, can be used to broadcast to the primary network if the primary network configured supports broadcast. Datagram sockets allow messages to be multicast by setting the destination address to a multicast address.

SOCK_RAW

Raw sockets give the application an interface to lower layer protocols, such as IP and ICMP. This interface is often used to bypass the transport layer when direct access to lower layer protocols is needed. Raw sockets are also used to test new protocols. Raw sockets are not supported in the `AF_IUCV` domain.

Raw sockets are connectionless and data transfer semantics are the same as those described previously for datagram sockets. The `connect` call can be used similarly to specify the peer.

Outgoing packets have an IP header prefixed to them. IP options can be set and inspected using the `setsockopt()` and `getsockopt()` calls respectively. Incoming packets are received with the IP header and options intact.

Sockets are deallocated with the `close()` call.

Raw sockets allow messages to be multicast by setting the destination address to a multicast address.

Note: When you use only `AF_IUCV` sockets and no `AF_INET` sockets, you need to run `SET TIMER REAL` on pre-VM/XA SP™ systems.

The following limitations apply:

- Only `SOCK_STREAM` sockets are supported in the `AF_IUCV` domain.
- The `setsockopt()` and `getsockopt()` calls are not supported for sockets in the `AF_IUCV` domain.
- The `flags` field in the `send()`, `recv()`, `sendto()`, `recvfrom()`, `sendmsg()`, `recvmsg()` calls is not supported in the `AF_IUCV` domain.

Return Values: A nonnegative socket descriptor indicates success. The value `-1` indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
<code>EIBMIUCVERR</code>	Indicates that an IUCV error occurred.
<code>EMFILE</code>	Indicates that the socket descriptor table is already full.

EPROTONOSUPPORT Indicates that the *protocol* is not supported in this *domain* or this *protocol* is not supported for this *socket type*.

Examples: The following are examples of the socket() call.

```
int s;
struct protoent *p;
struct protoent *getprotobyname(char *name);
int socket(int domain, int type, int protocol);
:
:
/* Get stream socket in internet domain with default protocol */
s = socket(AF_INET, SOCK_STREAM, 0);
:
:
/* Get stream socket in iucv domain with default protocol */
s = socket(AF_IUCV, SOCK_STREAM, 0);
:
:
/* Get raw socket in internet domain for ICMP protocol */
p = getprotobyname("iucv");
s = socket(AF_INET, SOCK_RAW, p->p_proto);
```

See Also: accept(), bind(), close(), connect(), fcntl(), getprotobyname(), getsockname(), getsockopt(), ioctl(), maxdesc(), read(), readv(), recv(), recvfrom(), recvmsg(), select(), selectex(), send(), sendmsg(), sendto(), shutdown(), write(), writev().

takesocket()

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

int takesocket(clientid, hisdesc)
struct clientid *clientid;
int hisdesc;
```

Parameter	Description
<i>clientid</i>	Points to the <i>clientid</i> of the application from whom you are taking a socket.
<i>hisdesc</i>	Specifies the descriptor of the socket to be taken.

Description: The takesocket() call acquires a socket from another program. Your program obtains the other program's client ID and socket descriptor through your program's startup parameter list. After successfully calling takesocket(), your application uses an agreed-upon mechanism to signal the other application so that it can close the socket.

Typically, takesocket() is used by an agent program that handles one socket at a time, taking its sockets from a master program that obtains them by means of the accept() call.

Return Values: A nonnegative socket descriptor indicates success. The value -1 indicates an error. The value of errno indicates a specific error.

Errno Value	Description
EACCES	Indicates that the other application did not give the socket to your application.

takesocket()

EBADF	Indicates that the <i>hisdesc</i> parameter does not specify a valid socket descriptor owned by the other application. The socket has already been taken.
EFAULT	Indicates that using the <i>clientid</i> parameter as specified would result in an attempt to access storage outside the caller's virtual storage.
EINVAL	Indicates that the <i>clientid</i> parameter does not specify a valid client identifier.
EMFILE	Indicates that the socket descriptor table is already full.
ENOBUFS	Indicates that the operation cannot be performed because of the shortage of SCB or SKCB control blocks in the TCPIP virtual machine.
EPFNOSUPPORT	Indicates that the domain field of the <i>clientid</i> parameter is not AF_INET.

See Also: getclientid(), givesocket().

tcperror()

```
#include <tcperrno.h>
```

```
void tcperror(s)  
char *s;
```

Parameter	Description
<i>s</i>	Specifies a NULL or NULL-terminated character string.

Description: When a socket call produces an error, the call returns a negative value and the variable *errno* is set to an error value found in the TCPERRNO.H header file. The `tcperror()` call prints a short error message describing the last error that occurred. If *s* is non-NULL, `tcperror()` 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.

The error messages printed by `tcperror()` are stored in an array of strings called `tcp_errlist`. The entry `tcp_errlist [errno]` is the message that `tcperror()` prints.

The `tcperror()` function is equivalent to the `perror()` function in UNIX[®].

Return Values: None.

Examples: The following are examples of the `tcperror()` call.

Example 1

```
if ((s=socket(AF_INET, SOCK_DGRAM, 0)) < 0) {  
    tcperror("socket()");  
    exit(2);  
}
```

If the `socket()` call produces the error ENOMEM, `socket()` returns a negative value and *errno* is set to ENOMEM. When `tcperror()` is called, it prints the string:

```
socket(): not enough memory (ENOMEM)
```


Example 2

```
if ((s=socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    tcperror(NULL);
```

If the `socket()` call produces the error `enomem`, `socket()` returns a negative value and `errno` is set to `ENOMEM`. When `tcperror()` is called, it prints the string:

```
Not enough memory (ENOMEM)
```

Example 3

```
if ((s=socket(AF_INET, SOCK_DGRAM, 0)) < 0){
    printf("error creating socket s: %s\n", tcp_errlist [errno]);
    exit(1);
}
```

If the `socket()` call produces the error `ENOMEM`, `socket()` returns a negative value and `errno` is set to `enomem`. The program then prints:

```
error creating socket s: not enough memory (ENOMEM)
```

write()

```
int write(s, buf, len)
int s;
char *buf;
int len;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>buf</i>	Points to the buffer holding the data to be written.
<i>len</i>	Specifies the length in bytes of the buffer pointed to by the <i>buf</i> parameter.

Description: The `write()` call writes data on a socket with descriptor *s*. The `write()` call applies only to connected sockets.

This call writes up to *len* bytes of data. If writing the number of bytes requested is not possible, the call waits for writing to be possible. This blocks the caller, unless the socket is in nonblocking mode. See “`ioctl()`” on page 54 or “`fcntl()`” on page 31 for a description of how to set nonblocking mode.

Return Values: If successful, the number of bytes written is returned. The value `-1` indicates an error. The value of `errno` indicates the specific error.

Errno Value	Description
EBADF	Indicates that <i>s</i> is not a valid socket descriptor.
EFAULT	Indicates that using the <i>buf</i> and <i>len</i> parameters would result in an attempt to access memory outside the caller’s virtual storage.
ENOBUFS	Indicates that no buffer space is available to send the message.
EWOULDBLOCK	Indicates that <i>s</i> is in nonblocking mode, and no data is available to read.

write()

See Also: connect(), fcntl(), getsockopt(), ioctl(), read(), readv(), recv(), recvfrom(), recvmsg(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), writev().

writev()

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

int writev(s, iov, iovcnt)
int s;
struct iovec *iov;
int iovcnt;
```

Parameter	Description
<i>s</i>	Specifies the socket descriptor.
<i>iov</i>	Points to an array of iovec buffers.
<i>iovcnt</i>	Specifies the number of buffers pointed to by the <i>iov</i> parameter.

Description: The writev() call writes data on a socket with descriptor *s*. The data is gathered from the buffers specified by *iov*[0]...*iov*[*iovcnt*-1]. The *iovec* structure is defined in *uio.h* and contains the following fields:

Element	Description
<i>iov_base</i>	Points to the buffer.
<i>iov_len</i>	Specifies the length of the buffer.

The writev() call applies only to connected sockets.

This call writes *len* bytes of data. If it is not possible to write the number of bytes requested, the writev() call waits for the ability to write. This blocks the caller, unless the socket is in nonblocking mode. See “fcntl()” on page 31 or “ioctl()” on page 54 for a description of how to set nonblocking mode.

Return Values: If successful, the number of bytes written from the buffer(s) is returned. The value -1 indicates an error. The value of *errno* indicates the specific error.

Errno Value	Description
EBADF	Indicates that <i>s</i> is not a valid socket descriptor.
EFAULT	Indicates that using the <i>iov</i> and <i>iovcnt</i> parameters would result in an attempt to access memory outside the caller’s virtual storage.
ENOBUFS	Indicates that no buffer space is available to send the message.
EWOULDBLOCK	Indicates that <i>s</i> is in nonblocking mode, and no data is available to read.

See Also: connect(), fcntl(), getsockopt(), ioctl(), write(), read(), readv(), recv(), recvmsg(), recvfrom(), select(), selectex(), send(), sendmsg(), sendto(), setsockopt(), socket(), write().

Sample C Socket Programs

This section provides examples of the following programs:

- C socket TCP client (see topic 89)
 - C socket TCP server (see topic 90)
 - C socket UDP server (see topic 92)
 - C socket UDP client (see topic 93)
-

C Socket TCP Client

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

```

/*
 * Include Files.
 */
#define VM
#include <manifest.h>
#include <bsdtypes.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: %shostname 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, "Gethostbynamefailed\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");

```

C Socket TCP Client

```
/*
 * Put the server information into the server structure.
 * The port must be put into network byte order.
 */
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)
{
    tcperror("Socket()");
    exit(-1);
}

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

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

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

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

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

C Socket TCP Server

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

```
/*
 * Include Files.
 */
#define VM
#include <manifest.h>
#include <bsdtypes.h>
#include <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 onlyone: 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)
    {
        tcperror("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)
    {
        tcperror("Bind()");
        exit(-1);
    }

    /*
    * Listen for connections. Specify the backlog as 1.
    */
    if (listen(s, 1) != 0)
    {
        tcperror("Listen()");
        exit(-1);
    }

    /*
    * Accept a connection.
    */
    namelen = sizeof(client);
    if ((ns = accept(s, &client,&namelen)) == -1)
    {
        tcperror("Accept()");
        exit(-1);
    }
}

```

C Socket TCP Server

```
    }

    /*
     * Receive the message on the newly connected socket.
     */
    if (recv(ns, buf, sizeof(buf),0) == -1)
    {
        tcperror("Recv()");
        exit(-1);
    }

    /*
     * Send the message back to the client.
     */
    if (send(ns, buf, sizeof(buf),0) < 0)
    {
        tcperror("Send()");
        exit(-1);
    }

    close(ns);
    close(s);

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

C Socket UDP Server

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

```
#include <manifest.h>
#include <bsdtypes.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 buff[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
 * 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);
}

```

C Socket UDP Client

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

```

#include <manifest.h>
#include <bsdtypes.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>

main(argc, argv)

```

C Socket UDP Client

```
int argc;
char **argv;
{

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

    /* argv[1] is internet address of server argv[2] is port of server.
     * 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);
    }
9
    /* 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()");
        exit(2);
    }

    /* Deallocate the socket */
    close(s);
}
```

Chapter 3. TCP/UDP/IP API (Pascal Language)

This chapter describes the Pascal language application program interface (API) provided with TCP/IP Level 320 for 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, TcpFReceive, 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 143.

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 14. 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 3 on page 97.

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

Figure 14. 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 3. TCP Connection States

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

Connection Information Record

The connection information record is used as a parameter 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 15. For more information about the use of each field, see “`TcpOpen` and `TcpWaitOpen`” on page 134 and “`TcpStatus`” on page 137

```

StatusInfoType =
  record
    Connection: ConnectionType;
    OpenAttemptTimeout: integer;
    Security: SecurityType;
    Compartment: CompartmentType;
    Precedence: PrecedenceType;
    BytesToRead: integer;
    UnackedBytes: integer;
    ConnectionState: ConnectionStateType;
    LocalSocket: SocketType;
    ForeignSocket: SocketType;
  end;

```

Figure 15. Pascal Declaration of Connection Information Record

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

Pascal Language

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 16.

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

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

Figure 16. Pascal Declaration of Socket Type

Field	Description
Address	Specifies the internet address.
Port	Specifies the port.

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 113. 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 17 on page 99.

```

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
      );
    FRECEIVEerror:
      (
        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 case;
end record;

```

Figure 17. Notification Record (Part 1 of 2)

Pascal Language

```
RAWIPpacketsDELIVERED:
(
  RawIpDataLength: integer;
  RawIpFullLength: integer;
);
RAWIPspaceAVAILABLE:
(
  RawIpSpaceInBytes: integer;
);
RESOURCESavailable: ();
MSGreceived: ();
TIMERexpired:
(
  Datum: integer;
  AssociatedTimer: TimerPointerType
);
UDPdatagramDELIVERED:
(
  DataLength: integer;
  ForeignSocket: SocketType;
  FullLength: integer
);
UDPdatagramSPACEavailable: ();
UDPresourcesAVAILABLE: ();
URGENTpending:
(
  BytesToRead: integer;
  UrgentSpan: integer
);
USERdefinedNOTIFICATION:
(
  UserData: UserNotificationDataType
);
end;
```

Figure 17. Notification Record (Part 2 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 "TcpFSend, TcpSend, and TcpWaitSend" on page 131.

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
 - REMOTEreset
 - 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.

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 DATAdelivered 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 CONNECTIONstateCHANGED with NewState set to NONEXISTENT notification for this connection (see 2 on page 101). 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 CONNECTIONstateCHANGED 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

Pascal Language

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 (Smsgs) arrive. The GetSmsg call is used to retrieve queued Smsgs. 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.

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 18.

```

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

```

Figure 18. 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 86 on page 405.

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.

Pascal Language

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 17 on page 99.

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 4. Pascal Language Interface Summary—Notifications

Procedure Call	Description	Page
<code>GetNextNote</code>	Retrieves the next notification.	113
<code>Handle</code>	Specifies the types of notifications that your program can process.	114
<code>NotifyIo</code>	Requests that an <code>IOinterrupt</code> notification be sent to you when an I/O interrupt occurs on a given virtual address.	118
<code>Unhandle</code>	Specifies notification types that your program can no longer process.	142
<code>UnNotifyIo</code>	Indicates that you no longer wish to be sent a notification when an I/O interrupt occurs on a given virtual address.	142

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 5. Pascal Language Interface Summary—TCP/UDP Initialization

Procedure Call	Description	Page
<code>TcpNameChange</code>	Identifies the name of the virtual machine running the TCP/IP program when the virtual machine has a name other than <code>TCPIP</code> .	133
<code>BeginTcpIp</code>	Establishes communication with the TCP/IP services.	110
<code>StartTcpNotice</code>	Establishes communication with the TCP/IP services.	126

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 6. Pascal Language Interface Summary—TCP/UDP Termination

Procedure Call	Description	Page
<code>EndTcpIp</code>	Terminates communication with the TCP/IP services.	111

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 7. Pascal Language Interface Summary—Handling External Interrupts

Procedure Call	Description	Page
<code>TcpExtRupt</code>	Notifies the TCP interface of the arrival of a simulated external interrupt.	128
<code>RTcpExtRupt</code>	A version of <code>TcpExtRupt</code> .	123
<code>TcpVmcfRupt</code>	Notifies the TCP interface of the arrival of a simulated external VMCF interrupt.	138
<code>RTcpVmcfRupt</code>	A version of <code>TcpVmcfRupt</code> .	123

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 8. Pascal Language Interface Summary—TCP Communication Procedures

Procedure Call	Description	Page
<code>TcpOpen</code>	Initiates a TCP connection.	134
<code>TcpWaitOpen</code>	Initiates a TCP connection and waits for the establishment of the connection.	134
<code>TcpFSend</code>	Sends TCP data.	131
<code>TcpSend</code>	Sends TCP data.	131
<code>TcpWaitSend</code>	Sends TCP data and waits until TCPIP accepts it.	131
<code>TcpFReceive</code>	Establishes a buffer to receive TCP data.	128
<code>TcpReceive</code>	Establishes a buffer to receive TCP data.	128
<code>TcpWaitReceive</code>	Establishes a buffer to receive TCP data and waits for the reception of the data.	128
<code>TcpClose</code>	Begins the TCP one-way closing sequence.	127
<code>TcpAbort</code>	Shuts down a TCP connection immediately.	127
<code>TcpStatus</code>	Obtains the current status of a TCP connection.	137

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.

Pascal Language

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 9. Pascal Language Interface Summary—Ping Interface

Procedure Call	Description	Page
PingRequest	Sends an Internet Control Message Protocol (ICMP) echo request.	119

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 10. Pascal Language Interface Summary—Monitor Procedures

Procedure Call	Description	Page
MonCommand	Instructs TCP to read a specific file and execute the commands that it contains.	116
MonQuery	Performs control functions and retrieves internal TCPIP control blocks.	117

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 11. Pascal Language Interface Summary—UDP Communication Procedures

Procedure Call	Description	Page
UdpOpen	Requests communication with UDP on a specified socket.	139
UdpSend	Sends a UDP datagram to a specified foreign socket.	141
UdpNReceive	Notifies the TCPIP virtual machine that you are willing to receive UDP datagram data.	139
UdpReceive	Notifies the TCPIP virtual machine that you are willing to receive UDP datagram data.	140
UdpClose	Terminates use of a UDP socket.	138

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 12. Pascal Language Interface Summary—Raw IP Interface

Procedure Call	Description	Page
RawIpOpen	Informs the TCPIP virtual machine that the client wants to send and receive IP packets of a specified protocol.	120
RawIpReceive	Specifies a buffer to receive raw IP packets of a specified protocol.	120
RawIpSend	Sends raw IP packets of a specified protocol.	121
RawIpClose	Informs the TCPIP virtual machine that the client no longer handles the protocol.	119

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 13. Pascal Language Interface Summary—Timer Routines

Procedure Call	Description	Page
<code>CreateTimer</code>	Allocates a timer.	111
<code>ClearTimer</code>	Resets a timer.	111
<code>SetTimer</code>	Sets a timer to expire after a specified interval.	126
<code>DestroyTimer</code>	Deallocates a timer.	111

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 14. Pascal Language Interface Summary—Host Lookup Routines

Procedure Call	Description	Page
<code>GetHostNumber</code>	Converts a host name to an internet address using static tables.	112
<code>GetHostResol</code>	Converts a host name to an internet address using a domain name resolver.	112
<code>GetHostString</code>	Converts an internet address to a host name using static tables.	113
<code>GetIdentity</code>	Returns environment information.	113
<code>IsLocalAddress</code>	Determines if an internet address is local.	115
<code>IsLocalHost</code>	Determines if a host name is local, remote, loopback, or unknown.	115

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 123 and “RTcpVmcfRupt” on page 123.

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 15. Pascal Language Interface Summary—Other Routines

Procedure Call	Description	Page
GetSmsg	Retrieves one queued special message (Smsg).	114
ReadXlateTable	Reads a binary translation table file.	122
SayCalRe	Converts a return code into a descriptive message.	124
SayConSt	Converts a connection state into a descriptive message.	124
SayIntAd	Converts an internet address into a name or dotted-decimal form.	124
SayIntNum	Converts an internet address into its dotted-decimal form.	125
SayNotEn	Converts a notification enumeration type into a descriptive message.	125
SayPorTy	Converts a port number into a descriptive message or into EBCDIC.	125
SayProTy	Converts the protocol type into a descriptive message or into EBCDIC.	125
AddUserNote	Adds a USERdefinedNOTIFICATION notification to the note queue.	110

Procedure Calls

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

BeginTcplp

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 4. Virtual Machine Communication Facility Interface” on page 147.

```
procedure BeginTcpIp
(
  var ReturnCode: integer
);
external;
```

Parameter	Description
ReturnCode	Indicates success or failure of call. Possible return values are: <ul style="list-style-type: none">• OK• ABNORMALcondition

- fatalerror
- NOtcpIPservice
- TCPipshutdown
- VIRTUALmemoryTOOsmall

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

ClearTimer

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

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

Parameter	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 126.

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

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

DestroyTimer

The DestroyTimer procedure deallocates or *fre*es a timer that you created.

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

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

EndTcplp

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

```
procedure EndTcpIp;  
external;
```

The EndTcpIp procedure has no parameters.

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.

```
procedure GetHostNumber  
(  
  const Name: string;  
  var HostNumber: InternetAddressType  
);  
external;
```

Parameter	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.

```
procedure GetHostResol  
(  
  const Name: string;  
  var HostNumber: InternetAddressType  
);  
external;
```

Parameter	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;
```

Parameter	Description
Address	Specifies the address to be converted.
Name	Set to the corresponding host, gateway, or network name, or to null string if no match found.

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 `TCPIPUSERID` 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;
```

Parameter	Description
UserId	Specifies the user ID of the VM user or the job name of a batch job that has invoked <code>GetIdentity</code> .
HostName	Specifies the host machine name.
DomainName	Specifies the network domain name.
TcpIpServiceName	Specifies the user ID of the TCPIP virtual machine.

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;
```

Parameter	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: <ul style="list-style-type: none">• OK• NOoutstandingNOTIFICATIONS• NOTyetBEGUN

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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;
```

Parameter	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 17 on page 99.

```

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

```

Parameter	Description
Notifications	Specifies the set of notification types to be handled.
ReturnCode	Indicates the success or failure of the call. Possible return values are: <ul style="list-style-type: none"> • OK • NOTyetBEGUN • TCPipSHUTDOWN • ABNORMALcondition • FATALerror

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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.

```

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

```

Parameter	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: <ul style="list-style-type: none"> • OK • NONlocalADDRESS • TCPipSHUTDOWN • ABNORMALcondition • FATALerror

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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:

Host Class	Description
HOSTlocal	Specifies an internet address for the local host.

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).

```
procedure IsLocalHost
(
  const Name: string;
  var Class: HostClassType
);
external;
```

Parameter	Description
Name	Specifies the host name.
Class	Specifies the host class.

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.

```
procedure MonCommand
(
  const FileSpec: SpecOfFileType;
  var ReturnCode: integer
);
external;
```

Parameter	Description
FileSpec	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 Figure 18 on page 105.
ReturnCode	Indicates the success or failure of the call. Possible return values are: <ul style="list-style-type: none">• OK• ABNORMALcondition• ERRORinPROFILE• HASnoPASSWORD• INCORRECTpassword• INVALIDuserID• INVALIDvirtualADDRESS• MINIDISKinUSE• MINIDISKnotAVAILABLE• NOTyetBEGUN• PROFILEnotFOUND• SOFTWAREerror• TCPipSHUTDOWN

- UNAUTHORIZEDuser
- UNIMPLEMENTEDrequest

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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;

```

Parameter	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 19.

```

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 19. 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 A. Pascal Return Codes" on page 405.

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;
```

Parameter	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 A. Pascal Return Codes" on page 405.

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;
```

Parameter	Description
ForeignAddress	Specifies the address of the foreign host to be pinged.
Length	Specifies the length of the ping packet, excluding the IP header. The range of values for this field are 8 to 512 bytes.
Timeout	Specifies how long to wait for a response, in seconds.
ReturnCode	Indicates the success or failure of the call. Possible return values are: <ul style="list-style-type: none"> • OK • ABNORMALcondition • BADlengthARGUMENT • CONNECTIONalreadyEXISTS • NObufferSPACE • NOTyetBEGUN

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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;
```

Parameter	Description
ProtocolNo	Specifies the number of the IP protocol.

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 A. Pascal Return Codes” on page 405.

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
);
external;
```

Parameter	Description
ProtocolNo	Specifies the number of the IP protocol.
ReturnCode	Indicates the success or failure of the call. Possible return values are: <ul style="list-style-type: none">• OK• LOCALportNOTavailable• NOTyetBEGUN• SOFTWAREError• TCPipSHUTDOWN• UNAUTHORIZEDuser

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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.

```

procedure RawIpReceive
(
    ProtocolNo: integer;
    Buffer: Address31Type;
    BufferLength: integer;
var
    ReturnCode: integer
);
external;

```

Parameter	Description
ProtocolNo	Specifies the number of the IP protocol.
Buffer	Specifies the address of your buffer.
BufferLength	Specifies the length of your buffer. If you specify a length greater than 8492 bytes, only the first 8492 bytes are used.
ReturnCode	Indicates the success or failure of the call. Possible return values are: <ul style="list-style-type: none"> • OK • NOsuchCONNECTION • NOTyetBEGUN • SOFTWAREerror • TCPipSHUTDOWN • UNAUTHORIZEDuser

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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

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

Parameter	Description
ProtocolNo	Specifies the number of the IP protocol.
Buffer	Specifies the address of your buffer containing packets to send.
DataLength	Specifies the total length of data in your buffer.
NumPackets	Specifies the number of packets in your buffer.
ReturnCode	Indicates the success or failure of the call. Possible return values are: <ul style="list-style-type: none">• OK• BADlengthARGUMENT• NObufferSPACE• NOsuchCONNECTION• NOTyetBEGUN• SOFTWAREerror• TCPipSHUTDOWN• UNAUTHORIZEDuser

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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.

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

Parameter	Description
TableName	Specifies the name of the translate table. ReadXlateTable tries to read TableName TCPXLBIN. If that file exists but it has a bad format, ReadXlateTable returns with a ReturnCode FILEformatINVALID. If <i>user_id</i> TCPXLBIN does not exist, ReadXlateTable tries to read TCPIP TCPXLBIN. ReturnCode reflects the status of reading that file.

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: <ul style="list-style-type: none"> • 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 parameters.

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

```
LA R13,PASCSAVE
LA R1,VMCFPARAM
L R15,=V(RTCPVMCF)
BALR R14,R15
.
.
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 RTCPVMCF.
VMCFPARAM DC A(ENV)
DC A(VMCFBUF) Address of your VMCF interrupt buffer.
```

The RTcpVmcfRupt procedure has no parameters.

SayCalRe

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

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

Parameter	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;
```

Parameter	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;
```

Parameter	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;
```

Parameter	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;
```

Parameter	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;
```

Parameter	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;
```

Parameter	Description
Protocol	Specifies the number of the protocol to be described.

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;
```

Parameter	Description
T	Specifies a timer pointer, as returned by a previous <code>CreateTimer</code> call.
AmountOfTime	Specifies the time interval in seconds.
Data	Specifies an integer value to be returned with the <code>TIMERExpired</code> notification.

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 4. Virtual Machine Communication Facility Interface” on page 147. 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;
```

Parameter	Description
-----------	-------------

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: <ul style="list-style-type: none"> • OK • ABNORMALcondition • ALREADYclosing • NOtcpIPservice • VIRTUALmemoryTOOsmall • FATALerror

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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.

```

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

```

Parameter	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: <ul style="list-style-type: none"> • OK • ABNORMALcondition • FATALerror • NOsuchCONNECTION • NOTyetBEGUN • TCPipSHUTDOWN

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 A. Pascal Return Codes” on page 405.

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;
```

Parameter	Description
Connection	Specifies the connection number, as returned by <code>TcpOpen</code> or <code>TcpWaitOpen</code> in the <code>Connection</code> field of the <code>StatusInfoType</code> record.
ReturnCode	Indicates the success or failure of the call. Possible return values are: <ul style="list-style-type: none">• <code>OK</code>• <code>ABNORMALcondition</code>• <code>ALREADYclosing</code>• <code>NOsuchCONNECTION</code>• <code>NOTyetBEGUN</code>• <code>TCPipSHUTDOWN</code>

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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 138.
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 123 and “`RTcpVmcfRupt`” on page 123.

```
procedure TcpExtRupt
(
    const RuptCode: integer
);
external;
```

Parameter	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.

```
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
);
external;
```

Parameter	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 A. Pascal Return Codes” on page 405.

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 2 on page 101). 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.

```
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;
```

Parameter	Description
Connection	Specifies the connection number, as returned by <code>TcpOpen</code> or <code>TcpWaitOpen</code> in the <code>Connection</code> field of the <code>StatusInfoType</code> record.
Buffer	Specifies the address of the buffer containing the data to send.

TcpFSend, TcpSend, and TcpWaitSend

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 A. Pascal Return Codes” on page 405.

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.

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

Parameter	Description
NewNameOfTcp	Specifies the name of the virtual machine running TCP/IP.

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.

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

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

Parameter	Description
ConnectionInfo	Specifies a connection information record.
Connection	Set this field to <code>UNSPECIFIEDconnection</code> . When the call returns, the field contains the number of the new connection if <code>ReturnCode</code> is OK.
ConnectionState	For active open, set this field to <code>TRYINGtoOPEN</code> . For passive open, set this field to <code>LISTENING</code> .

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)
- WRONGsecORprc (TcpWaitOpen only)
- ZEROresources

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

TcpOption

The TcpOption procedure sets an option for a TCP connection.

```
procedure TcpOption
(
  Connection: ConnectionType
  OptionName: integer;
  OptionValue: integer;
var
  ReturnCode: integer
);
external;
```

Parameter	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 A. Pascal Return Codes” on page 405.

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
);
external;

```

Parameter **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 A. Pascal Return Codes” on page 405.

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 123 and “RTcpVmcfRupt” on page 123.

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

Parameter	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;
```

Parameter	Description
ConnIndex	Specifies the ConnIndex value returned from UdpOpen.
ReturnCode	Indicates the success or failure of the call. Possible return values are: <ul style="list-style-type: none">• OK• ABNORMALcondition• FATALerror• NOsuchCONNECTION• NOTyetBEGUN• SOFTWAREerror• TCPIPshutdown

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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.

```

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

```

Parameter	Description
ConnIndex	Specifies the ConnIndex value returned from UdpOpen.
BufferAddress	Specifies the address of your buffer that will be filled with a UDP datagram.
BufferLength	Specifies the length of your buffer. If you specify a length larger than 8192 bytes, only the first 8192 bytes are used.
ReturnCode	Indicates the success or failure of the call. Possible return values are: <ul style="list-style-type: none"> • OK • ABNORMALcondition • FATALerror • NOsuchCONNECTION • NOTyetBEGUN • RECEIVEstillPENDING • TCPIPshutdown

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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.

```

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

```

UdpOpen

Parameter	Description
LocalSocket	Specifies the local socket (address and port pair).
ConnIndex	Specifies the ConnIndex value returned from UdpOpen.
ReturnCode	Indicates the success or failure of the call. Possible return values are: <ul style="list-style-type: none">• OK• ABNORMALcondition• FATALerror• LOCALportNOTavailable• NOTyetBEGUN• SOFTWAREerror• TCPipSHUTDOWN• UDPlocalADDRESS• UDPzeroRESOURCES

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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;
```

Parameter	Description
ConnIndex	Specifies the ConnIndex value returned from UdpOpen.
DatagramAddress	Specifies the address of your buffer that will be filled with a UDP datagram.

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 A. Pascal Return Codes” on page 405.

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
);
external;

```

Parameter	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: <ul style="list-style-type: none"> • OK • BADlengthARGUMENT • NObufferSPACE • NOsuchCONNECTION • NOTyetBEGUN • SOFTWAREerror • TCPipSHUTDOWN • UDPunspecADDRESS • UDPunspecPORT

UdpSend

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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.

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

Parameter	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: <ul style="list-style-type: none">• OK• ABNORMALcondition• FATALerror• INVALIDrequest• NOTyetBEGUN• TCPipSHUTDOWN
-------------------	--

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

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.

```
procedure UnNotifyIo
(
    DeviceAddress: integer;
var   ReturnCode: integer
);
external;
```

Parameter	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: <ul style="list-style-type: none">• OK• NOsuchCONNECTION• SOFTWAREerror
-------------------	---

For a description of Pascal ReturnCodes, see “Appendix A. Pascal Return Codes” on page 405.

Sample Pascal Program

```

{*****}
{*
{* Memory-to-memory Data Transfer Rate Measurement
{*
{*
{* Pseudocode: Establish access to TCP/IP Services
{* Prompt user for operation parameters
{* 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;

```

Sample Pascal Program

```
        DropEmulation;
        Close (Input);
        Close (Output);
    end;

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 parameters }
    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.
```

Chapter 4. Virtual Machine Communication Facility Interface

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 20 on page 148.

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:

ANINTEGR	DS	F
CONN	DS	H
CALLCODE	DS	X
RETCODE	DS	X

Figure 20. 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.

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 A. Pascal Return Codes” on page 405). Further information is given in the description of each function.

VMCF Functions

Table 16 lists the available VMCF functions, with descriptions, to communicate with the TCPIP virtual machine.

Table 16. Available VMCF Functions

Function	Code	Description
AUTHORIZE	Control	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.
UNAUTHORIZE	Control	Terminates VMCF activity.
SEND	Data	Directs a message or block of data to another virtual machine.
SEND/RECV	Data	Directs a message or block of data to another virtual machine, and requests a reply.
RECEIVE	Data	Allows you to accept selective messages or data sent using the SEND or SEND/RECV functions.
REPLY	Data	Allows you to direct data back to the originator of a SEND/RECV function, simulating duplex communication.
REJECT	Data	Allows you to reject specific SEND or SEND/RECV requests pending for your virtual machine.

VMCF Interface

Table 16. Available VMCF Functions (continued)

Function	Code	Description
Note:		
Data		Indicates a data transfer
Control		Indicates a VMCF control function

VMCF TCPIP Communication CALLCODE Requests

Table 17 lists the equate values and available calls for initiating a VMCF TCPIP request; it also includes a description of each CALLCODE request.

Table 17. VMCF TCPIP CALLCODE Requests

Call Code	Equates	Description
CONNECTIONclosing	00	Data may no longer be transmitted on this connection since the TCP/IP service is in the process of closing down the process.
LISTENING	01	Waiting for a foreign site to open a connection.
NONEXISTENT	02	The connection no longer exists.
OPEN	03	Data can go either way on the connection.
RECEIVINGonly	04	Data can be received but not sent on this connection, because the client has done a one-way close.
SENDINGonly	05	Data can be sent out but not received on this connection. This means that the foreign site has done a one way close.
TRYINGtoOPEN	06	Trying to contact a foreign site to establish a connection.
ABORTtcp	100	Terminates a TCP connection.
BEGINtcpIPservice	101	Initializes a TCP/IP connection between your program and the TCPIP virtual machine.
CLOSEtcp	102	Initiates the closing of a TCP connection.
CLOSEudp	103	Initiates the closing of a UDP connection.
ENDtcpIPservice	104	Terminates the use of TCPIP services. All existing TCP connections are reset, all open UDP ports are canceled, and all IP protocols are released.
HANDLEnotice	105	Specifies the types of notifications to be received from TCPIP.
IshostLOCAL	106	Determines whether a given internet address is one of your host's local addresses.
MONITORcommand	107	Instructs TCPIP to obey a file of commands.
MONITORquery	108	Obtains status information from the TCPIP virtual machine or requests that it performs certain functions.
OPENTcp	110	Initiates a TCP connection.
OPENudp	111	Initiates a UDP connection.
OPTIONtcp	112	Sets an option for a TCP connection.
RECEIVEtcp	113	Tells TCPIP that you are ready to receive data on a specified TCP connection.

Table 17. VMCF TCPIP CALLCODE Requests (continued)

Call Code	Equates	Description
NRECEIVEudp	115	Tells TCPIP that your program is ready to receive a UDP datagram on a particular port.
SENDtcp	118	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.
SENDudp	119	Sends a UDP datagram.
STATUStcp	120	Obtains a Connection Information Record giving the current status of a TCP connection.
FRECEIVEtcp	121	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.
FSENDtcp	122	Sends data to a TCP connection. FSENDtcp waits for available receiving buffer space in the TCPIP virtual machine before completing the VMCF transaction.
CLOSErawIP	123	Tells TCPIP that your program does not handle the protocol any longer. Any queued incoming packets are discarded.
OPENrawIP	124	Initiates a connection and tells TCPIP virtual machine that your program is ready to send and receive packets of a specified IP protocol.
RECEIVERawIP	125	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.
SENDrawIP	126	Tells TCPIP virtual machine to send raw IP packets of a given protocol number.
PINGreq	127	Sends an ICMP echo request to a specified host and wait a specified time for a response.

VMCF TCPIP Communication CALLCODE Notifications

Table 18 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 18. VMCF TCPIP CALLCODE Notifications

Notification Code	Equates	Description
BUFFERspaceAVAILABLE	10	Notification that there is space available to send data on this connection. The space is currently set to 8192 bytes of buffer space.
CONNECTIONstateCHANGED	11	Notification that the state of the connection between the TCPIP virtual machine and your program has changed.
DATAdelivered	12	Notification that the TCPIP virtual machine data was delivered to your program, after issuing a RECEIVEtcp or FRECEIVEtcp call.

VMCF Interface

Table 18. VMCF TCPIP CALLCODE Notifications (continued)

Notification Code	Equates	Description
URGENTpending	15	Notification that there is queued data on a TCP connection not yet received by your program.
UDPdatagramDELIVERED	16	Notification that UDP datagram has been delivered to your program after issuing a NRECEIVEudp call to the TCPIP virtual machine.
UDPdatagramSPACEavailable	17	Notification that buffer space is available to process the data, after an error occurred performing a SENDudp call.
RAWIPpacketsDELIVERED	24	Notification that your buffer has received the raw IP packets.
RAWIPspaceAVAILABLE	25	Notification that buffer space is available to process the data. This notification is sent after the SENDrawip call was rejected by TCPIP virtual machine.
RESOURCESavailable	28	Notification 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.
UDPresourcesAVAILABLE	29	Notification that the resources needed to initiate a UDP connection are now available. This notification is sent only if a previous OPENudp call received a UDPzeroRESOURCES return code.
PINGresponse	30	Notification that your ping request from the PINGreq call has been received or that the time-out limit or your request has been reached.
DUMMYprobe	31	Notification that the TCPIP virtual machine is monitoring your machine
ACTIVEprobe	32	Notification that the TCPIP virtual machine is monitoring your machine for responsiveness

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.

```

FUNC:    SEND
VADA:    0
LENA:    1
VADB:    0 or, if your application supports probe messages (see the
          descriptions of the DUMMYprobe and ACTIVEprobe CALLCODE
          notifications), X'80000000'
LENB:    0
CALLCODE: BEGINtcpIPservice

```

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 BeginTcpIp Pascal procedure (page 110).

HANDLEnotice

Your program performs the 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 21.

Figure 21 shows the equates used for notification mask in the HANDLEnotice call.

MaskBUFFERspaceAVAILABLE	EQU	X'00000001'
MaskCONNECTIONstateCHANGED	EQU	X'00000002'
MaskDATAdelivered	EQU	X'00000004'
MaskURGENTpending	EQU	X'00000020'
MaskUDPdatagramDELIVERED	EQU	X'00000040'
MaskUDPdatagramSPACEavailable	EQU	X'00000080'
MaskRAWIPpacketsDELIVERED	EQU	X'00004000'
MaskRAWIPspaceAVAILABLE	EQU	X'00008000'
MaskRESOURCESavailable	EQU	X'00040000'
MaskUDPresourcesAVAILABLE	EQU	X'00080000'
MaskPINGresponse	EQU	X'00100000'

Figure 21. Equates for Notification Mask in the HANDLEnotice Call

Each HANDLEnotice call must specify all the notification types to be handled. Notification types specified in previous HANDLEnotice calls are not stored.

```

FUNC:    SEND
VADA:    0
LENA:    1
VADB:    Note mask
LENB:    0
CALLCODE: HANDLEnotice

```

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 Handle Pascal procedure (see "Handle" on page 114).

ENDtcpIPservice

Your program performs the 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.

VMCF Interface

```
FUNC:    SEND
VADA:    0
LENA:    1
VADB:    0
LENB:    0
CALLCODE: ENDtcpIPservice
```

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 22 gives the assembler format of the record. Figure 23 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 134.

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 22. 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 23. Miscellaneous Assembler Constants

```
FUNC:    SEND/RCV
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 98 and see also “CALLCODE Notifications” on page 163.

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 131.

```

FUNC:      SEND
VADA:      Address of data
LENA:      Length of data
VADB:      1 if push desired, else 0
LENB:      1 if urgent data, else 0
CONN:      Connection number from open
CALLCODE:  SENDtcp or FSENDtcp

```

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 131.

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. If

VMCF Interface

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.

FRECEIVtcp

The FRECEIVtcp 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 FRECEIVtcp function results in an outstanding VMCF transaction. There is a limit of 50 outstanding VMCF transactions for each virtual machine; you can therefore have FRECEIVtcp functions outstanding on only 50 connections at one time. If your application needs more outstanding receives, use the RECEIVtcp function.

Your program does not need to wait for a response from FRECEIVtcp. It can issue functions involving other connections, before receiving a response from FRECEIVtcp.

For general information about receiving TCP data, see the TcpFReceive Pascal procedure under "TcpFReceive, TcpReceive, and TcpWaitReceive" on page 128.

```
FUNC:    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: FRECEIVtcp
```

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.

RETCODE
OK

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 101.

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 128.

```

FUNC:      SEND
VADA:      0
LENA:      1
VADB:      0
LENB:      Length of buffer to receive data
CONN:      Connection number from open
CALLCODE:  RECEIVetcp

```

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt buffer indicates whether the request was successful. If the request was successful, with a RETCODE of OK, the data is delivered to your buffer and a notification of DATAdelivered is sent to your program. If the request was not successful, then the return code is one of those listed for the TcpReceive Pascal procedure.

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 127.

```

FUNC:      SEND
VADA:      0
LENA:      1
VADB:      0
LENB:      0
CONN:      Connection number from open
CALLCODE:  CLOSEtcp

```

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 TcpClose Pascal procedure on page 127.

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 127.

```
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 22 on page 154. For more information about the connection status call, see the Pascal procedure, “TcpStatus” on page 137.

```
FUNC:    SEND/RCV
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 136.

```
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 139.

```

FUNC:      SEND
VADA:      0
LENA:      1
VADB:      Local port number or UNSPECIFIEDport
LENB:      Local address
CONN:      Connection number: An arbitrary number, which your program
           uses in subsequent actions involving this port.
CALLCODE:  OPENudp

```

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 141.

```

FUNC:      SEND
VADA:      Address of datagram data
LENA:      Length of datagram data (up to 8492 bytes)
VADB:      Destination port number
LENB:      Destination address
CONN:      Connection number
CALLCODE:  SENDudp

```

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 139.

```

FUNC:      SEND
VADA:      0
LENA:      1
VADB:      0
LENB:      Size of your buffer to receive datagram
CONN:      Connection number
CALLCODE:  NRECEIVEudp

```

VMCF Interface

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 138.

```
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 120.

```
FUNC:    SEND/RCV
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 121.

```
FUNC:    SEND/RCV
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 RAWIPpacketsDELIVERED notification when a packet arrives. For information about the RAWIPpacketsDELIVERED notification record, see the Pascal procedure, “RawIpReceive” on page 120, and the section on the Pascal NotificationInfoType under “Notification Record” on page 98.

```

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 119.

```

FUNC:      SEND/RECV
VADA:      0
LENA:      1
VADB:      0
LENB:      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 115.

VMCF Interface

```
FUNC:    SEND
VADA:    0
LENA:    1
VADB:    Internet address to be tested
LENB:    0
CONN:    UNSPECIFIEDconnection
CALLCODE: IShostLOCAL
```

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 116, 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 24. Assembler Format of the SpecOfFileType Record for VM

```
FUNC:    SEND/RECV
VADA:    Address of SpecOfFile record
LENA:    Length of SpecOfFile record
VADB:    0
LENB:    0
CONN:    UNSPECIFIEDconnection
CALLCODE: MONITORcommand
```

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 117. Assembler formats of constants and records used with MONITORquery are:

```
COMMANDcpCMD      EQU 6
COMMANDdropCONNECTION EQU 8
QUERYhomeONLY     EQU 9
```

Figure 25. Equates for MonQueryRecordType used in the MONITORquery Call

```

QueryType  DS   X
* For QueryType = QUERYhomeONLY: No other fields
* For QueryType = COMMANDcpCMD:
CpCmd      DS   H      Length of command
           DS   100C   Command
* For QueryType = COMMANDdropCONNECTION:
           ORG   CpCmd
Connection DS   H

```

Figure 26. Assembler Format of the MonQueryRecordTypefor VM

The Pascal type HomeOnlyListType is an array of 64 InternetAddressType elements found in the COMMMAC MACLIB file. The size of InternetAddressType is a fullword.

```

FUNC:      SEND/RCV
VADA:      Address of MonQueryRecord describing request
LENA:      Length of MonQueryRecord
VADB:      Address of return buffer
LENB:      Length of return buffer
CONN:      UNSPECIFIEDconnection
CALLCODE:  MONITORquery

```

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 COMMANDdropCONNECTION), 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 117.

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 119.

```

FUNC:      SEND
VADA:      0
LENA:      1
VADB:      Internet address of foreign host
LENB:      Length of ping packet
ANINTEGR:  Timeout
CALLCODE:  PINGreq

```

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 98.

VMCF Interface

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.

FUNC: SEND
 JOBNAME: Name of the TCPIP virtual machine
 LENA: Length of data being delivered
 VADB: Non-zero if data was pushed, else zero.
 LENB: The offset of the byte following the last byte of urgent data, measured from the first byte of data delivered to your buffer. If zero or negative then there is no urgent data pending.
 CONN: Connection number
 CALLCODE: DATADELIVERED
 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. 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 UDPzeroRESOURCES 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 5. 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 *VM/ESA 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 *VM/ESA CMS Application Development Reference for Assembler* or *VM/ESA 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 19. Socket functions available using IUCV

ACCEPT	READ
BIND	READV
CLOSE	RECV
CONNECT	RECVFROM
FCNTL	RECVMSG
GETCLIENTID	SELECT
GETHOSTID	SELECTEX
GETHOSTNAME	SEND
GETPEERNAME	SENDMSG
GETSOCKNAME	SENDTO
GETSOCKOPT	SETSOCKOPT
GIVESOCKET	SHUTDOWN
IOCTL	SOCKET (AF_INET sockets only)
LISTEN	TAKESOCKET
MAXDESC	WRITE
	WRITEV

Socket Programming with IUCV

TCP/IP sockets are manipulated by using the following assembler macros:

Macro	Library	Description
IUCV	HCPGPI	Provides the mechanisms for setting values in the IUCV input parameter list and for executing the IUCV instruction
IPARML	HCPGPI	Mapping macro for the IUCV parameter list and the external interrupt buffer.
HNDIUCV	DMSGPI	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.
CMSIUCV	DMSGPI	Used to perform IUCV CONNECT and SEVER functions. It enables multiple IUCV or APPC/VM applications to run at the same without interference.
IUCVINI	GCTGPI	Similar to HNDIUCV, but for the GCS execution environment.
IUCVCOM	GCTGPI	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.

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:

IPTYPE	Interrupt Name	Description
X'02'	Connection Complete	Acknowledgement that TCP/IP has accepted your request to establish an IUCV connection (IUCV CONNECT)
X'03'	Connection Severed	Your IUCV connection has been deleted by TCP/IP
X'07'	Message complete	The requested socket function has completed

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. *VM/ESA 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 *VM/ESA 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 *VM/ESA CMS Application Development Reference for Assembler* or *VM/ESA 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:

Keyword	Value
USERID	The user ID of the TCP/IP virtual machine.
PRTY	NO
PRMDATA	YES
QUIESCE	NO
MSGLIM	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.
USERDTA	Binary zeros
CONTROL	NO

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 173 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 174.

The initialization message is sent using an IUCV SEND with the following parameters:

Keyword	Value
TRGCLS	0
DATA	BUFFER
BUFLLEN	20
TYPE	2WAY
ANSLEN	8
PRTY	NO
BUFFER	Points to a buffer in the following format:

Offset	Name	Length	Comments
0		8	Constant 'IUCVAPI '. The trailing blank is required.
8		2	Halfword integer. Maximum number of sockets that can be established on this IUCV connection. minimum: 50, Default: 50.
10	<i>apitype</i>	2	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 “Overlapping Socket Requests” on page 174.
12	<i>subtaskname</i>	8	Eight printable characters. The combination of your user ID and <i>subtaskname</i> uniquely identifies the TCP/IP client using this path. This value is displayed by the NETSTAT CLIENT command.

ANSBUF Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0		4	Reserved
4	<i>maxsock</i>	4	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.

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:

Reason Code	Explanation
IUCVCHECKRC	IUCV error detected. This code is used only before or during processing of the initialization message.
SHUTTINGDOWN	TCP/IP service is being shut down. This code is used only in response to the Pending Connection interrupt.
BAD PATH ID	An attempt was made to exceed the maximum number of IUCV connections support by the target TCPIP virtual machine.
NULL SAVED NAME	A software error occurred in TCP/IP. This code is used only before or during processing of the initialization message.
BAD INIT MSG LEN	Your program sent an initialization message that was not of the expected length.
REQUIREDCONSTANT	The first 8 bytes of your initialization message were not "IUCVAPI ".
BAD API TYPE	The <i>apitype</i> field in your initialization message contained an incorrect value.
RESTRICTED	Your virtual machine is not permitted to use TCP/IP.

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:

Keyword	Value
TRGCLS	The high-order halfword specifies the socket call. For most calls, the low-order halfword specifies the socket descriptor.
DATA	BUFFER or PRMMSG, depending on call
BUFLIST	If DATA=BUFFER, then either YES or NO as desired. If DATA=PRMMSG, not applicable.
BUFFER	If DATA=BUFFER, points to the buffer (or buffer list) in the format required by the call. If DATA=PRMMSG, not applicable.
BUFLEN	If DATA=BUFFER, length of buffer. If DATA=PRMMSG, not applicable.
PRMMSG	If DATA=PRMMSG, data as required by the call. DATA=PRMMSG is not allowed when ANSLIST=YES. If DATA=BUFFER, not applicable.
TYPE	2WAY
ANSLIST	Either YES or NO as desired. DATA=PRMMSG is not allowed when ANSLIST=YES.
ANSBUF	Points to a buffer to contain the reply from TCP/IP.
ANSLEN	Length of the reply buffer
PRTY	NO
SYNC	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.

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 IPADRJCT 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 200) 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.

Errno Value	Description
1000	An unrecognized socket call constant was found in the high-order halfword of the Target Message Class.
1001	A request or reply length field is incorrect
1002	The socket number assigned by your program for ACCEPT, SOCKET, or TAKESOCKET is out of range.
1003	The socket number assigned by your program for ACCEPT, SOCKET, or TAKESOCKET is already in use.
1008	This request conflicts with a request already queued on the same socket (see "Overlapping Socket Requests" on page 174).
1009	The request was canceled by the CANCEL call (see "CANCEL" on page 178).
1010	Returned by the Offload function when a beginthread failure occurs.

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 178 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 20 shows the C structures used, and the corresponding assembler language syntax.

Table 20. C Structures in Assembler Language Format

C Structure	Assembler Language Equivalent
<pre>struct sockaddr_in { short sin_family; ushort sin_port; struct in_addr sin_addr; char sin_zero[8]; };</pre>	<pre>FAMILY DS H PORT DS H ADDR DS F ZERO DC XL8'00'</pre>
<pre>struct timeval { long tv_sec; long tv_usec; };</pre>	<pre>TVSEC DS F TVUSEC DS F</pre>
<pre>struct linger { int l_onoff; int l_linger; };</pre>	<pre>ONOFF DS F LINGER DS F</pre>
<pre>struct ifreq { #define IFNAMSIZ 16 char ifr_name[IFNAMSIZ]; union { struct sockaddr ifru_addr; struct sockaddr ifru_dstaddr; struct sockaddr ifru_broadaddr; short ifru_flags; int ifru_metric; caddr_t ifru_data; } ifr_ifru; };</pre>	<pre>NAME DS CL16 ADDR.FAMILY DS H ADDR.PORT DS H ADDR.ADDR DS F ADDR.ZERO DC XL8'00' ORG ADDR.FAMILY DST.FAMILY DS H DST.PORT DS H DST.ADDR DS F DST.ZERO DC XL8'00' ORG ADDR.FAMILY BRD.FAMILY DS H BRD.PORT DS H BRD.ADDR DS F BRD.ZERO DC XL8'00' ORG ADDR.FAMILY FLAGS DS H ORG ADDR.FAMILY METRIC DS F</pre>

Table 20. C Structures in Assembler Language Format (continued)

C Structure	Assembler Language Equivalent
<pre>struct ifconf { int ifc_len; union { caddr_t ifcu_buf; struct ifreq *ifcu_req; } ifc_ifcu; };</pre>	<pre>IFCLEN DS F IGNORED DS F</pre>
<pre>struct clientid { int domain; char name[8]; char subtaskname[8]; char reserved[20]; };</pre>	<pre>DOMAIN DS F NAME DS CL8 SUBTASK DS CL8 RESERVED DC XL20'00'</pre>

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 2. C Sockets Application Program Interface” on page 5.

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.

```
ns = accept(s, addr, addrlen)
int ns, s;
struct sockaddr_in *addr;
int *addrlen;
```

Keyword	Value
TRGCLS	High-order halfword = 1 Low-order halfword = s
DATA	PRMMSG
PRMMSG	High-order fullword = 0 Low-order fullword = socket number for the new socket, chosen by your program, in the range 0 through <i>maxsock</i> . See “Initializing the IUCV Connection” on page 171.
ANSLEN	24

ACCEPT

ANSBUF Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0	<i>ns</i>	4	The new socket number assigned to this connection. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>ns</i> is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.
8	<i>*addr</i>	16	The remote address and port of the new socket. See Table 20 on page 176 for format.

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.

```
rc = bind(s, name, namelen)
int rc, s;
struct sockaddr_in *name;
int namelen;
```

Keyword	Value
TRGCLS	High-order halfword = 2 Low-order halfword = <i>s</i>
DATA	BUFFER
BUFLen	16
BUFFER	Points to a buffer in the following format:

Offset	Name	Length	Comments
0	<i>*name</i>	16	The local address and port to which the socket is to be bound. See Table 20 on page 176 for format.

ANSLEN 8

ANSBUF Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

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.

CANCEL

Keyword	Value
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:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code. Possible <i>errno</i> values are: 3 Specifies that the call cannot be found. TCP/IP might have already responded to it. 22 Specifies that the call is not a type that may be canceled.

CLOSE

The CLOSE call shuts down the socket and frees the resources that are allocated to the socket.

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

Keyword	Value
TRGCLS	High-order halfword = 3 Low-order halfword = <i>s</i>
DATA	PRMMSG
PRMMSG	Binary zeros
ANSLEN	8
ANSBUF	Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

CONNECT

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.

```
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:

Offset	Name	Length	Comments
0	<i>*name</i>	16	The remote address and port to which the socket is to be connected. See Table 20 on page 176 for format.

ANSLEN	8
ANSBUF	The pointer to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

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 186 for another way to control blocking for a socket.

```
retval = fcntl(s, cmd, arg)
int retval;
int s, cmd, arg;
```

Keyword	Value
TRGCLS	High-order halfword = 5 Low-order halfword = <i>s</i>
DATA	PRMMSG
PRMMSG	High-order fullword: F_GETFL (X '00000003') F_SETFL (X '00000004')

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:

Offset	Name	Length	Comments
0	<i>retval</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

GETCLIENTID

The GETCLIENTID call returns the identifier by which the calling application is known to the TCP/IP address space. The client ID structure that is returned is used in the GIVESOCKET and TAKESOCKET calls.

```
rc = getclientid(domain, clientid)
int rc, domain;
struct clientid *clientid;
```

Keyword	Value
TRGCLS	High-order halfword = 30 Low-order halfword = 0
DATA	PRMMSG
PRMMSG	Binary zeros
ANSLEN	48
ANSBUF	Points to the buffer that is filled with a reply in the following format:

GETCLIENTID

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the GETCLIENTID 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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	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.
8	<i>*clientid</i>	40	See Table 20 on page 176 for format.

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:

Offset	Name	Length	Comments
0	<i>hostid</i>	4	The default home internet address.
4		4	Your program should ignore this field.

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
DATA	PRMMSG
PRMMSG	Binary zeros

ANSLEN *namelen + 8*

ANSBUF Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	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.
8	<i>*name</i>	<i>namelen</i>	The host name, not null-terminated.

GETPEERNAME

The GETPEERNAME call returns the name of the remote socket to which the local socket is connected.

```
rc = getpeername(s, name, namelen)
int rc, s;
struct sockaddr_in *name;
int *namelen;
```

Keyword	Value
TRGCLS	High-order halfword = 9 Low-order halfword = s
DATA	PRMMSG
PRMMSG	Binary zeros
ANSLEN	24
ANSBUF	Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	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.
8	<i>*name</i>	16	The remote address and port to which the socket is connected. See Table 20 on page 176 for format.

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

GETSOCKNAME

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.

```
rc = getsockname(s, name, namelen)
int rc, s;
struct sockaddr_in *name;
int *namelen;
```

Keyword	Value
TRGCLS	High-order halfword = 10 Low-order halfword = <i>s</i>
DATA	PRMMSG
PRMMSG	Binary zeros
ANSLEN	24
ANSBUF	Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	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.
8	<i>*name</i>	16	The local address and port to which the socket is bound. See Table 20 on page 176 for format.

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.

```
rc = getsockopt(s, level, optname, optval, &optlen)
int rc, s, level, optname, optlen;
char *optval;
```

Keyword	Value
TRGCLS	High-order halfword = 11 Low-order halfword = <i>s</i>
DATA	PRMMSG
PRMMSG	High-order fullword = <i>level</i> . Possible values are:

Value	C Symbol	Comments
X'FFFF'	SOL_SOCKET	Socket option
X'0006'	IPPROTO_TCP	TCP protocol option

Low-order fullword = *optname*. Possible values are:

Value	Option Name	Returned Value								
X'0001'	SO_DEBUG	Returns current setting.								
X'0004'	SO_REUSEADDR	Returns current setting.								
X'0008'	SO_KEEPAVIVE	Returns current setting.								
X'0010'	SO_DONTROUTE	Returns current setting.								
X'0020'	SO_BROADCAST	Returns current setting.								
X'0080'	SO_LINGER	Returns current setting in a C language <i>struct linger</i> . See Table 20 on page 176 for the assembler language equivalent.								
X'0100'	SO_OOINLINE	Returns current setting.								
X'1001'	SO_SNDBUF	Returns the size of the TCP/IP send buffer.								
X'1007'	SO_ERROR	Returns any pending error code and clears any error status conditions.								
X'1008'	SO_TYPE	Socket type is returned: <table border="1" style="margin-left: 20px;"> <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>	Value	Type	1	Stream	2	Datagram	3	Raw
Value	Type									
1	Stream									
2	Datagram									
3	Raw									
X'0001'	TCP_NODELAY	Returns current setting. Note: This option applies only to <i>level=IPPROTO_TCP</i>								

ANSLEN 16 for option SO_LINGER, 12 for all other options

ANSBUF Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	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.
8	<i>*optval</i>	4 or 8	The value of the requested option. If the option SO_LINGER was requested, 8 bytes are returned. For all other options, 4 bytes are returned.

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:

GIVESOCKET

- 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		
TRGCLS	High-order halfword = 31 Low-order halfword = s		
DATA	BUFFER		
BUFLen	40		
BUFFER	Points to the message in the following format:		
Offset	Name	Length	Comments
0	<i>*clientid</i>	40	See Table 20 on page 176 for format.
ANSLEN	8		
ANSBUF	The pointer to the buffer that is filled with a reply in the following format:		
Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

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.

```
rc = ioctl(s, cmd, arg)
int rc, s;
unsigned long cmd;
char *arg;
```

Keyword	Value
TRGCLS	High-order halfword = 12 Low-order halfword = <i>s</i>
DATA	BUFFER
BUFLLEN	Request <i>arg</i> length + 4
BUFFER	The pointer to the message in the format described in the following format:

Offset	Name	Length	Comments
0	<i>cmd</i>	4	The type of request. See Table 21 for values.
4	<i>*arg</i>	See Table 21.	The request data, if any.

ANSLEN	Reply <i>arg</i> length + 8
ANSBUF	The pointer to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	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.
8	<i>*arg</i>	See Table 21.	The response data, if any.

Table 21. Values for *cmd* Argument in *ioctl* Call

C Symbol	Value	Request <i>arg</i> Length	Reply <i>arg</i> Length	Comments
FIONBIO	X'8004A77E'	4	0	Request <i>arg</i> data is a fullword integer.
FIONREAD	X'4004A77F'	0	4	Reply <i>arg</i> data is a fullword integer.
SIOCADDRT	X'8030A70A'	48	0	For IBM use only.
SIOCATMARK	X'4004A707'	0	4	Reply <i>arg</i> data is a fullword integer.
SIOCDELRT	X'8030A70B'	48	0	For IBM use only.
SIOCGIFADDR	X'C020A70D'	32	32	<i>arg</i> data is the C language <i>struct ifreq</i> . See Table 20 on page 176 for the assembler language equivalent.
SIOCGIFBRDADDR	X'C020A712'	32	32	<i>arg</i> data is the C language <i>struct ifreq</i> . See Table 20 on page 176 for the assembler language equivalent.

IOCTL

Table 21. Values for cmd Argument in ioctl Call (continued)

C Symbol	Value	Request arg Length	Reply arg Length	Comments
SIOCGIFCONF	X'C008A714'	8	*	Request <i>arg</i> data is the C-language <i>struct ifconf</i> . See Table 20 on page 176 for the assembler language equivalent. Your program sets <i>ifc_len</i> to the reply length. The other field is ignored. Response <i>arg</i> data is an array of C language <i>struct ifreq</i> structures, one for each defined interface. Note: * = the maximum number of interfaces multiplied by 32.
SIOCGIFDSTADDR	X'C020A70F'	32	32	<i>arg</i> data is the C language <i>struct ifreq</i> . See Table 20 on page 176 for the assembler language equivalent.
SIOCGIFFLAGS	X'C020A711'	32	32	<i>arg</i> data is the C language <i>struct ifreq</i> . See Table 20 on page 176 for the assembler language equivalent.
SIOCGIFMETRIC	X'C020A717'	32	32	For IBM use only.
SIOCGIFNETMASK	X'C020A715'	32	32	<i>arg</i> data is the C language <i>struct ifreq</i> . See Table 20 on page 176 for the assembler language equivalent.
SIOCSIFDSTADDR	X'8020A70E'	32	0	For IBM use only.
SIOCSIFFLAGS	X'8020A710'	32	0	For IBM use only.
SIOCSIFMETRIC	X'8020A718'	32	0	For IBM use only.
SIOCGIBMOPT	C048D900	72	*	For IBM use only.
SIOCSIBMOPT	8048D900	*	0	For IBM use only.

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 = <i>s</i>
DATA	PRMMSG	
PRMMSG		High-order fullword = 0
		Low-order fullword = <i>backlog</i>
ANSLEN	8	
ANSBUF		Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

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 171.

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 = <i>s</i>
DATA	PRMMSG
PRMMSG	Binary zeros
ANSLEN	<i>len</i> + 24
ANSBUF	Points to the buffer that is filled with a reply in the following format:

READ, READV

Offset	Name	Length	Comments
0	<i>cc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>cc</i> 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	<i>*buf</i>	<i>len</i>	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 = recvfrom(s, buf, len, flags, from, fromlen)
int cc, s;
char *buf;
int len, flags;
struct sockaddr_in *from;
int *fromlen;
```

Keyword	Value
TRGCLS	High-order halfword = 16 Low-order halfword = <i>s</i>
DATA	PRMMSG
PRMMSG	High-order fullword = 0. Low-order fullword = <i>flags</i> :
	MSG_OOB (X'00000001')
	MSG_PEEK (X'00000002')
ANSLEN	<i>len</i> + 24
ANSBUF	Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>cc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>cc</i> is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.
8	<i>*from</i>	16	The source address and port of the message. See Table 20 on page 176 for format.
24	<i>*buf</i>	<i>len</i>	The received data.

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 174 for more information about multiple-request socket programs.

```
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:

SELECT, SELECTEX

bitmask = X'00000001' << (*descriptor_number* modulo 32)

(“<<” 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:

Keyword	Value
TRGCLS	High-order halfword = 19 Low-order halfword = descriptor set size in bytes (<i>fdsize</i>). See “Descriptor Sets” on page 191.
DATA	BUFFER
BUFLEN	(3* <i>fdsize</i>)+28
BUFFER	The pointer to the message in the following format:

Offset	Name	Length	Number of file descriptors
0	<i>nfds</i>	4	To improve processing efficiency, <i>nfds</i> should be no greater than one plus the largest descriptor number actually in use.
4		4	Set this field to zero if you want select to block. Otherwise set this field to any nonzero value and fill in <i>*timeval</i> .
8		4	If any descriptor bits are set in <i>readfds</i> , your program sets this field to a nonzero value. If no descriptor bits are set in <i>readfds</i> , your program can set this field to zero, to improve processing efficiency.
12		4	If any descriptor bits are set in <i>writfds</i> , your program sets this field to a nonzero value. If no descriptor bits are set in <i>writfds</i> , your program can set this field to zero to improve processing efficiency.
16		4	If any descriptor bits are set in <i>exceptfds</i> , your program sets this field to a nonzero value. If no descriptor bits are set in <i>exceptfds</i> your program can set this field to zero to improve processing efficiency.
20	<i>*timeval</i>	8	See Table 20 on page 176 for format. If field at offset 4 is zero, then set this field to binary zeros.
28	<i>*readfds</i>	<i>fdsize</i>	If field at offset 8 is zero, then this field is not used.
28 + <i>fdsize</i>	<i>*writfds</i>	<i>fdsize</i>	If field at offset 12 is zero, then this field is not used.
28 + (2* <i>fdsize</i>)	<i>*exceptfds</i>	<i>fdsize</i>	If field at offset 16 is zero, then this field is not used.

ANSLEN (3**fdsize*)+16

ANSBUF The pointer to the buffer that is filled in with a reply in the following format:

Offset	Name	Length	Comments
0	<i>nfound</i>	4	The total number of ready sockets (in all bit masks). A value of zero indicates an expired time limit. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>nfound</i> is -1, this field contains a reason code.
8		8	Your program ignores this field. Note: The rest of the reply buffer is filled only if the call was successful.
16	<i>*readfds</i>	<i>fdsize</i>	If field at offset 8 in request data was zero, then your program ignores this field.
16 + <i>fdsize</i>	<i>*writefds</i>	<i>fdsize</i>	If field at offset 12 in request data was zero, then your program ignores this field.
16 + (2* <i>fdsize</i>)	<i>*exceptfds</i>	<i>fdsize</i>	If field at offset 16 in request data was zero, then your program ignores this field.

SEND

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.

```
cc = send(s, msg, len, flags)
int cc, s;
char *msg;
int len, flags;
```

Keyword	Value
TRGCLS	High-order halfword = 20 Low-order halfword = <i>s</i>
BUFLEN	<i>len</i> + 20
DATA	BUFFER
BUFFER	The pointer to the message in the following format:

Offset	Name	Length	Comments
0	<i>flags</i>	4	MSG_OOB (X'00000001') MSG_DONTROUTE (X'00000004')
4		16	Your program should set this field to binary zeros.

SEND

Offset	Name	Length	Comments
20	<i>*msg</i>	<i>len</i>	The data to be sent.
ANSLEN	8		
ANSBUF	The pointer to the buffer that is filled in with a reply in the following format:		
Offset	Name	Length	Comments
0	<i>cc</i>	4	The number of bytes sent. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>cc</i> is -1, this field contains a reason code.

SENDMSG

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 193 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:

Offset	Name	Length	Comments
0	<i>flags</i>	4	MSG_OOB (X'00000001') MSG_DONTROUTE (X'00000004')
4	<i>*to</i>	16	See Table 20 on page 176 for format.
20	<i>*msg</i>	<i>len</i>	The data to be sent.

ANSLEN	8
ANSBUF	The pointer to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>cc</i>	4	The number of bytes sent. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>cc</i> is -1, this field contains a reason code.

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:

Offset	Name	Length	Comments
0	<i>level</i>	4	X'FFFF' - SOL_SOCKET - Socket option X'0006' - IPPROTO_TCP - TCP protocol option

SETSOCKOPT

Offset	Name	Length	Comments
4	<i>optname</i>	4	Option to set. See Table 22 for values.
8	<i>*optval</i>	4 or 8	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.

Table 22. Option name values for SETSOCKOPT

Value	Option Name	Option Value
X'0001'	SO_DEBUG	On (1) or Off (0). Option may be set, but has no effect.
X'0004'	SO_REUSEADDR	Yes (1) or No (0).
X'0008'	SO_KEEPALIVE	Yes (1) or No (0).
X'0010'	SO_DONTROUTE	Yes (1) or No (0). Option may be set, but has no effect. Use MSG_DONTROUTE on write-type calls instead.
X'0020'	SO_BROADCAST	Yes (1) or No (0).
X'0080'	SO_LINGER	Value is a C language <i>struct linger</i> . See Table 20 on page 176 for the assembler language equivalent.
X'0100'	SO_OOBINLINE	Yes (1) or No (0). Note: The following option applies only to <i>level=IPPROTO_TCP</i>
X'0001'	TCP_NODELAY	Yes (1) or No (0).

ANSLEN 8

ANSBUF Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

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

Low-order halfword = *s*

DATA PRMMSG

PRMMSG High-order fullword = 0
Low-order fullword = *how*:

0 = receive
1 = send
2 = both

ANSLEN 8

ANSBUF Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

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 = socket(domain, type, protocol)
int s, domain, type, 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:

Offset	Name	Length	Comments
0	<i>domain</i>	4	The only valid value is AF_INET (X'00000002')
4	<i>type</i>	4	Fullword integer: SOCK_STREAMX'00000001' SOCK_DGRAMX'00000002' SOCK_RAWX'00000003'
8	<i>protocol</i>	4	Fullword integer: IPPROTO_ICMP X'00000001' IPPROTO_TCP X'00000006' IPPROTO_UDP X'00000011' IPPROTO_RAW X'000000FF'

SOCKET

Offset	Name	Length	Comments
12	<i>s</i>	4	Socket number for the new socket, chosen by your program, in the range 0 through <i>maxsock</i> . See "Initializing the IUCV Connection" on page 171.
ANSLEN 8			
ANSBUF Points to the buffer that is filled with a reply in the following format:			
Offset	Name	Length	Comments
0	<i>s</i>	4	The socket number assigned to this communications end point. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>s</i> 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	<i>*clientid</i>	40	See Table 20 on page 176 for format.
40	<i>hisdesc</i>	4	
44	<i>s</i>	4	Socket number for the new socket, chosen by your program, in the range 0 through <i>maxsock</i> . See "Initializing the IUCV Connection" on page 171.

ANSLEN 8

ANSBUF The pointer to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>s</i>	4	The socket number assigned to this communications end point. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>s</i> is -1, this field contains a reason code.

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.

```
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:

Offset	Name	Length	Comments
0		20	Your program sets this parameter to binary zeros.
20	<i>*buf</i>	<i>len</i>	The data to be sent.

ANSLEN 8

ANSBUF Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>cc</i>	4	The number of bytes sent. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>cc</i> is -1, this field contains a reason code.

LASTERRNO

LASTERRNO

As explained in “TCP/IP Response to an IUCV Request” on page 175, 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:

Offset	Name	Length	Comments
0	<i>rc</i>	4	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 <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

Chapter 6. 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 246. Figure 27 on page 202 and Figure 28 on page 203 provide an overview of the high-level RPC client and server processes from initialization through cleanup.

RPCs

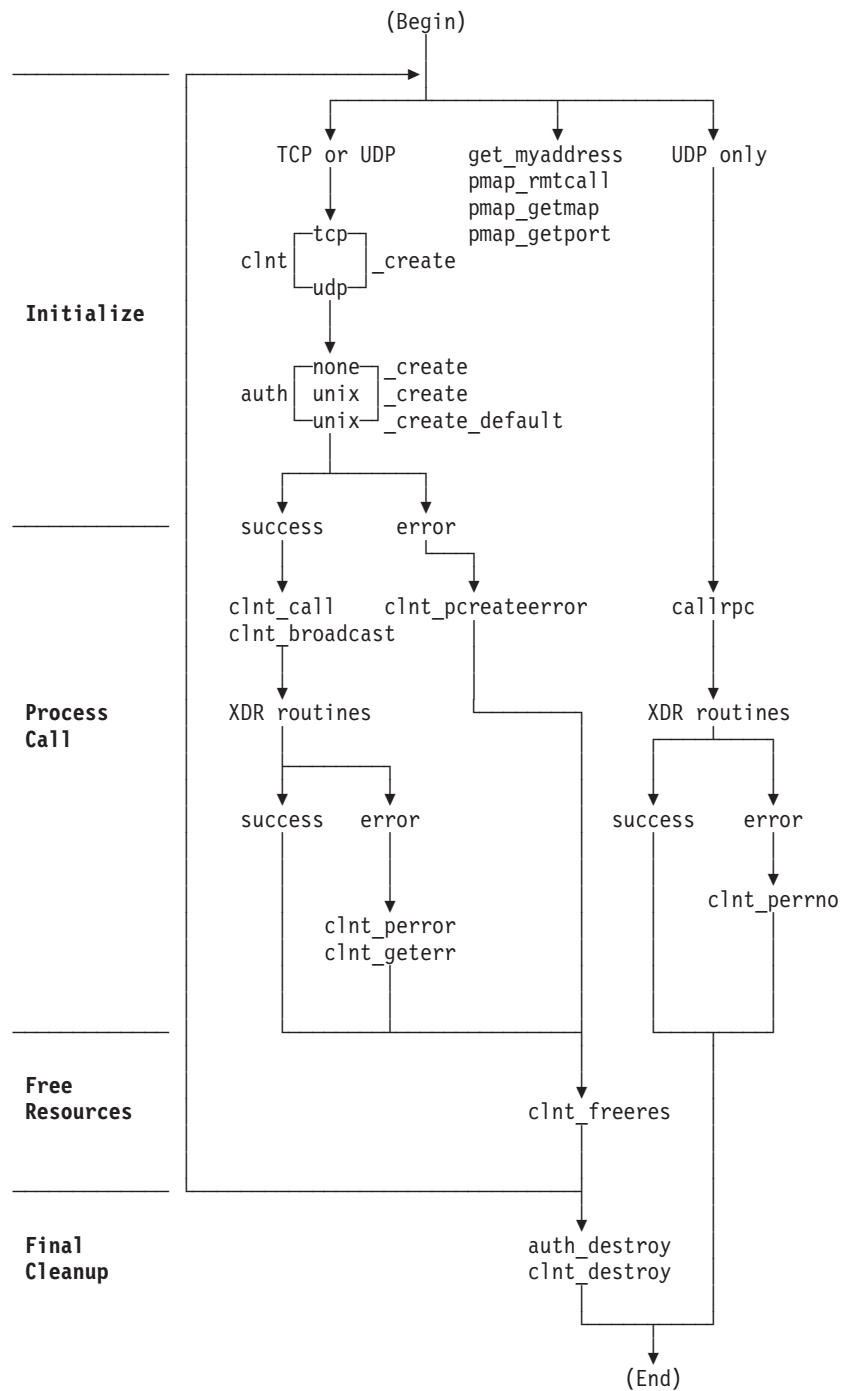


Figure 27. Remote Procedure Call (Client)

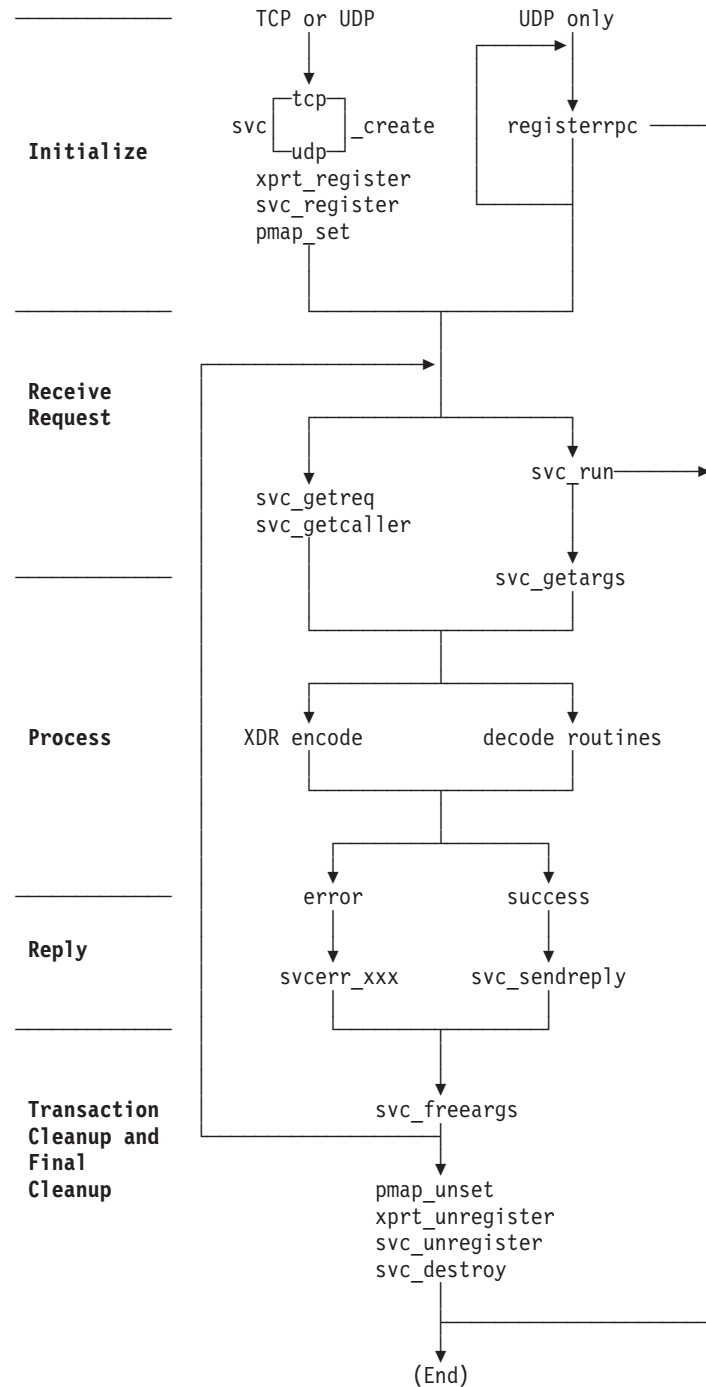


Figure 28. Remote Procedure Call (Server)

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

RPCs

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, see *TCP/IP User's Guide*.

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

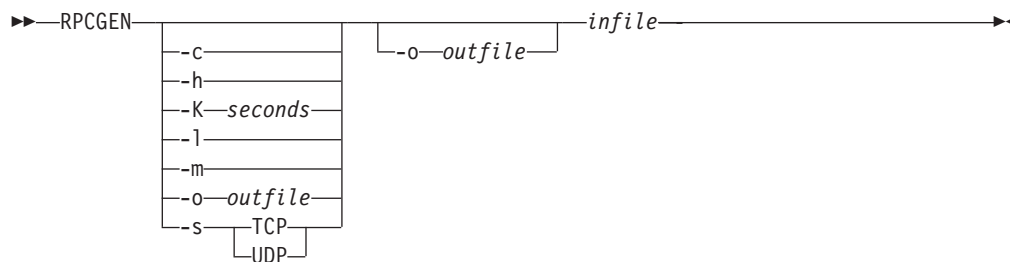
RPCGEN is a tool that generates C code to implement an RPC protocol. The input to RPCGEN is a language similar to C, known as RPC language. For RPCGEN to work correctly you must have access to the CC EXEC that is a part of the C compiler and have accessed the TCPMAINT.592 disk.

RPCGEN *infile* is normally used when you want to generate all four of the following output files. For example, if the *infile* is named *proto.x*, RPCGEN generates:

- A header file called `PROTO.H`
- XDR routines called `PROTOX.C`
- Server-side stubs called `PROTOS.C`
- Client-side stubs called `PROTOS.C`

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*.



Parameter	Description
-c	Compiles into XDR routines.
-Dname[=value]	Define a symbol "name". Equivalent to the #define directive in the source. If no "value" is given, "name" is defined as 1. This option may be called more than once.
-h	Compiles into C data definitions (a header file). The -T option can be used in conjunction to produce a header file which supports RPC dispatch tables.
-K seconds	If the server was started by inetd, specify the time in seconds after which the server should exit if there is no further activity. This option is useful for customization. If seconds is 0, the server exits after serving that given request. If seconds is -1, the server hangs around for ever after being started by inetd. This option is valid only with the -I option.
-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.

The options -c, -h, -l, -m, -s and -t are used exclusively to generate a particular type of file, while the options -D, -I, -L and -T are global and can be used with the other options.

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:

RPCs

```
enum cInt_stat {
    RPC_SUCCESS=0,          /* call succeeded */
    /*
     * local errors
     */
    RPC_CANTENCODEARGS=1,  /* can't encode arguments */
    RPC_CANTDECODERES=2,   /* can't decode results */
    RPC_CANTSEND=3,        /* failure in sending call */
    RPC_CANTRECV=4,        /* failure in receiving result */
    RPC_TIMEDOUT=5,        /* call timed out */
    /*
     * remote errors
     */
    RPC_VERSMISMATCH=6,    /* RPC versions not compatible */
    RPC_AUTHERROR=7,       /* authentication error */
    RPC_PROGUNAVAIL=8,     /* program not available */
    RPC_PROGVERSMISMATCH=9, /* program version mismatched */
    RPC_PROGUNAVAIL=10,    /* procedure unavailable */
    RPC_CANTDECODEARGS=11, /* decode arguments error */
    RPC_SYSTEMERROR=12,    /* generic "other problem" */
    /*
     * callrpc errors
     */
    RPC_UNKNOWNHOST=13,    /* unknown host name */
    /*
     * create errors
     */
    RPC_PMAPFAILURE=14,    /* the pmaper failed in its call */
    RPC_PROGNOTREGISTERED=15, /* remote program is not registered */
    /*
     * unspecified error
     */
    RPC_FAILED=16,         /* call failed */
    RPC_UNKNOWNPROTO=17    /* unknown protocol */
};
```

Remote Procedure Call Library

RPC requires the RPCLIB TXTLIB and the COMMTXT TXTLIB for compiling and linking. These TXTLIBs are normally named in a GLOBAL TXTLIB statement in the PROFILE EXEC or in the TCPLOAD EXEC.

Porting

This section contains information about porting RPC applications.

Remapping C Identifiers with RPC.H

To conform to the VM requirement that C identifiers are 8 characters or less in length, a header file called MANIFEST.H remaps the RPC long names to 8-character derived names for internal processing. This file is implicitly included by the RPC.H file and must be present to compile and link.

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 233.

RPC Global Variables

This section describes the two RPC global variables, `prc_createerr` and `svc_fds`.

`rpc_createerr`

Description: A global variable that is set when any RPC client creation routine `#include <rpc.h>`

```
struct rpc_createerr rpc_createerr;
```

fails. Use `clnt_pcreateerror()` to print the message.

See Also: `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

`svc_fds`

```
int svc_fds;
```

Description: A global variable that specifies the read descriptor bit mask 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()`.

Remote Procedure and eXternal Data Representation Calls

This section provides the syntax, parameters, 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;
```

Parameter	Description
-----------	-------------

auth_destroy()

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()

```
&numsign.include <rpc.h>.
AUTH *
authnone_create()
```

The `authnone_create()` call has no parameters.

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()

```
#include <rpc.h>

AUTH *
authunix_create(host, uid, gid, len, aup_gids)
char *host;
int uid;
int gid;
int len;
int *aup_gids;
```

Parameter	Description
-----------	-------------

<i>host</i>	Specifies a pointer to the symbolic name of the host where the desired server is located.
<i>uid</i>	Identifies the user's user ID.
<i>gid</i>	Identifies the user's group ID.
<i>len</i>	Specifies the length of the information pointed to by <i>aup_gids</i> .
<i>aup_gids</i>	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 parameters.

Description: The `authunix_create_default()` call calls `authunix_create()` with default parameters.

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;
```

Parameter	Description
<i>host</i>	Specifies a pointer to the symbolic name of the host where the desired server is located.
<i>prognum</i>	Identifies the program number of the remote procedure.
<i>versnum</i>	Identifies the version number of the remote procedure.
<i>procnum</i>	Identifies the procedure number of the remote procedure.
<i>inproc</i>	Specifies the XDR procedure used to encode the arguments of the remote procedure.
<i>in</i>	Specifies a pointer to the arguments of the remote procedure.
<i>outproc</i>	Specifies the XDR procedure used to decode the results of the remote procedure.
<i>out</i>	Specifies a pointer to the results of the remote procedure. <code>clnt_perrno()</code> can be used to translate the return code into messages.

Description: The `callrpc()` call calls the remote procedure described by *prognum*, *versnum*, and *procnum* running on the *host* system. `callrpc()` encodes and decodes the parameters for transfer.

For more information on `callrpc()` cannot call the procedure `xdr_enum`; see “`xdr_enum()`” on page 233. For more information on, `callrpc()` uses UDP as its transport layer, see “`clntudp_create()`” on page 217.

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()

clnt_broadcast()

```
#include <rpc.h>

enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)
u_long prognum;
u_long versnum;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
resultproc_t eachresult;
```

Parameter	Description
<i>prognum</i>	Identifies the program number of the remote procedure.
<i>versnum</i>	Identifies the version number of the remote procedure.
<i>procnum</i>	Identifies the procedure number of the remote procedure.
<i>inproc</i>	Specifies the XDR procedure used to encode the arguments of the remote procedure.
<i>in</i>	Specifies a pointer to the arguments of the remote procedure.
<i>outproc</i>	Specifies the XDR procedure used to decode the results of the remote procedure.
<i>out</i>	Specifies a pointer to the results of the remote procedure.
<i>eachresult</i>	Specifies the procedure called after each response.

Note: resultproc_t is a type definition:

```
#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:

```
#include <rpc.h>

bool_t eachresult(out, addr)
char *out;
struct sockaddr_in *addr;
```

Parameter	Description
<i>out</i>	Has the same function as it does for clnt_broadcast(), except that the output of the remote procedure is decoded.
<i>addr</i>	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().

clnt_call()

```
#include <rpc.h>

enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
CLIENT *clnt;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
```

Parameter	Description
<i>clnt</i>	Points to a client handle that was previously obtained using <code>clntraw_create()</code> , <code>clnttcp_create()</code> , or <code>clntudp_create()</code> .
<i>procnum</i>	Identifies the remote procedure number.
<i>inproc</i>	Identifies the XDR procedure used to encode <i>procnum</i> 's arguments.
<i>in</i>	Points to the remote procedure's arguments.
<i>outproc</i>	Specifies the XDR procedure used to decode the remote procedure's results.
<i>out</i>	Points to the remote procedure's results.
<i>tout</i>	Specifies the time allowed for the server to respond in units of 0.1 seconds.

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;
```

Parameter	Description
<i>clnt</i>	Specifies the pointer to a client handle that was previously obtained using <code>clntraw_create()</code> , <code>clnttcp_create()</code> , or <code>clntudp_create()</code> .
<i>request</i>	Determines the operation (either <code>CLSET_TIMEOUT</code> , <code>CLGET_TIMEOUT</code> , <code>CLGET_SERVER_ADDR</code> , <code>CLSET_RETRY_TIMEOUT</code> , or <code>CLGET_RETRY_TIMEOUT</code>).
<i>info</i>	Points to information used by the request.

clnt_control()

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 parameter 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()

```
#include <rpc.h>
```

```
CLIENT *  
clnt_create(host, prognum, versnum, protocol)  
char *host;  
u_long prognum;  
u_long versnum;  
char *protocol;
```

Parameter	Description
<i>host</i>	Points to the name of the host at which the remote program resides.
<i>prognum</i>	Specifies the remote program number.
<i>versnum</i>	Specifies the version number of the remote program.
<i>protocol</i>	Points to the protocol, which can be either <code>tcp</code> or <code>udp</code> .

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()

```
#include <rpc.h>

void
clnt_destroy(clnt)
CLIENT *clnt;
```

Parameter	Description
<i>clnt</i>	Points to a client handle that was previously created using <code>clntudp_create()</code> , <code>clnttcp_create()</code> , or <code>clntraw_create()</code> .

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()

```
#include <rpc.h>

bool_t
clnt_freeres(clnt, outproc, out)
CLIENT *clnt;
xdrproc_t outproc;
char *out;
```

Parameter	Description
<i>clnt</i>	Points to a client handle that was previously obtained using <code>clntraw_create()</code> , <code>clnttcp_create()</code> , or <code>clntudp_create()</code> .
<i>outproc</i>	Specifies the XDR procedure used to decode the remote procedure's results.
<i>out</i>	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()`.

clnt_geterr()

```
#include <rpc.h>

void
clnt_geterr(clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;
```

Parameter	Description
-----------	-------------

clnt_geterr()

clnt Points to a client handle that was previously obtained using `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_screateerror()`, `clnt_sperrno()`, `clnt_sperror()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clnt_pcreateerror()

```
#include <rpc.h>

void
clnt_pcreateerror(s)
char *s;
```

Parameter	Description
<i>s</i>	Specifies a NULL or NULL-terminated character string. If <i>s</i> is non-NULL, <code>clnt_pcreateerror()</code> prints the string <i>s</i> followed by a colon, followed by a space, followed by the error message, and terminated with a newline character. If <i>s</i> is NULL or points to a NULL string, just the error message and the newline character are output.

Description: 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 `clntraw_create()`, `clnttcp_create()`, or `clntudp_create()` calls fail.

See Also: `clnt_create()`, `clnt_geterr()`, `clnt_perrno()`, `clnt_perror()`, `clnt_screateerror()`, `clnt_sperrno()`, `clnt_sperror()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clnt_perrno()

```
#include <rpc.h>

void
clnt_perrno(stat)
enum clnt_stat stat;
```

Parameter	Description
<i>stat</i>	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_screateerror()`, `clnt_sperrno()`, `clnt_sperror()`.

clnt_perror()


```
#include <rpc.h>

void
clnt_perror(clnt, s)
CLIENT *clnt;
char *s;
```

Parameter	Description
<i>clnt</i>	Points to a client handle that was previously obtained using <code>clntudp_create()</code> , <code>clnttcp_create()</code> , or <code>clntraw_create()</code> .
<i>s</i>	Points to a string that is to be printed in front of the message. The string is followed by a colon.

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_geterr()`, `clnt_pcreateerror()`, `clnt_perrno()`, `clnt_screateerror()`, `clnt_sperrno()`, `clnt_serror()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clnt_screateerror()

```
#include <rpc.h>

char *
clnt_screateerror(s)
char *s;
```

Parameter	Description
<i>s</i>	Specifies a NULL or NULL-terminated character string. If <i>s</i> is non-NULL, <code>clnt_screateerror()</code> prints the string <i>s</i> followed by a colon, followed by a space, followed by the error message, and terminated with a new-line character. If <i>s</i> is NULL or points to a NULL string, just the error message and the new-line character are printed.

Description: The `clnt_screateerror()` 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.

See Also: `clnt_create()`, `clnt_geterr()`, `clnt_perrno()`, `clnt_perror()`, `clnt_pcreateerror()`, `clnt_sperrno()`, `clnt_serror()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clnt_sperrno()

```
#include <rpc.h>

char *
clnt_sperrno(stat)
enum clnt_stat stat;
```

Parameter	Description
<i>stat</i>	Specifies the client status.

clnt_sperrno()

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.

See Also: `callrpc()`, `clnt_geterr()`, `clnt_pcreateerror()`, `clnt_spcreateerror()`, `clnt_sperror()`, `clnt_perrno()`, `clnt_perror()`.

clnt_sperror()

```
#include <rpc.h>

char *
clnt_sperror(clnt, s)
CLIENT *clnt;
char *s;
```

Parameter	Description
<i>clnt</i>	Points to a client handle that was previously obtained using <code>clnt_create()</code> , <code>clntudp_create()</code> , <code>clnttcp_create()</code> , or <code>clntraw_create()</code> .
<i>s</i>	Specifies a NULL or a NULL-terminated character string. If <i>s</i> is non-NULL, <code>clnt_sperror()</code> prints the string <i>s</i> followed by a colon, followed by a space, followed by the error message, and terminated with a newline character. If <i>s</i> is NULL or points to a NULL string, just the error message and the newline character are output.

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.

See Also: `clnt_call()`, `clnt_create()`, `clnt_geterr()`, `clnt_pcreateerror()`, `clnt_perrno()`, `clnt_perror()`, `clnt_spcreateerror()`, `clnt_sperrno()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clntraw_create()

```
#include <rpc.h>

CLIENT *
clntraw_create(prognum, versnum)
u_long prognum;
u_long versnum;
```

Parameter	Description
<i>prognum</i>	Specifies the remote program number.
<i>versnum</i>	Specifies the version number of the remote program.

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

clntraw_create()

machine, which simulates RPC programs within one virtual machine. For more information, see “svcrow_create()” on page 228.

Return Values: NULL indicates failure.

See Also: clnt_call(), clnt_destroy(), clnt_freeres(), clnt_geterr(), clnt_pcreateerror(), clnt_perror(), clnt_screateerror(), clnt_sperror(), clntudp_create(), clnttcp_create(), svcraw_create().

clnttcp_create()

```
#include <rpc.h>
```

```
CLIENT *  
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)  
struct sockaddr_in *addr;  
u_long prognum;  
u_long versnum;  
int *sockp;  
u_int sendsz;  
u_int recvsz;
```

Parameter	Description
<i>addr</i>	Points to the internet address of the remote program. If the <i>addr</i> port number is zero (<i>addr</i> → <i>sin_port</i>), <i>addr</i> is set to the port on which the remote program is receiving.
<i>prognum</i>	Specifies the remote program number.
<i>versnum</i>	Specifies the version number of the remote program.
<i>sockp</i>	Points to the socket. If <i>sockp</i> is <i>RPC_ANYSOCK</i> , then this routine opens a new socket and sets <i>sockp</i> .
<i>sendsz</i>	Specifies the size of the send buffer. Specify 0 to choose the default.
<i>recvsz</i>	Specifies the size of the receive buffer. Specify 0 to choose the default.

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_screateerror(), clnt_sperror(), clntraw_create(), clntudp_create().

clntudp_create()

clntudp_create()

```
#include <rpc.h>

CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
struct timeval wait;
int *sockp;
```

Parameter	Description
addr	Points to the internet address of the remote program. If the <i>addr</i> port number is zero (<i>addr</i> → <i>sin_port</i>), <i>addr</i> is set to the port on which the remote program is receiving. The remote portmap service is used for this.
<i>prognum</i>	Specifies the remote program number.
<i>versnum</i>	Specifies the version number of the remote program.
<i>wait</i>	Indicates that UDP resends the call request at intervals of <i>wait</i> time, until either a response is received or the call times out. The time-out length is set using the <i>clnt_call()</i> procedure.
<i>sockp</i>	Points to the socket. If <i>sockp</i> is <i>RPC_ANYSOCK</i> , this routine opens a new socket and sets <i>sockp</i> .

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 32 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_freeres()*, *clnt_geterr()*, *clnt_pcreateerror()*, *clnt_perror()*, *clnt_screateerror()*, *clnt_sperror()*, *clntraw_create()*, *clnttcp_create()*.

get_myaddress()

```
#include <rpc.h>

void
get_myaddress(addr)
struct sockaddr_in *addr;
```

Parameter	Description
<i>addr</i>	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_set()*, *pmap_unset()*.

getrpcport()

```
#include <rpc.h>

u_short
getrpcport(host, prognum, versnum, protocol)
char *host;
u_long prognum;
u_long versnum;
int protocol;
```

Parameter	Description
<i>host</i>	Points to the name of the foreign host.
<i>prognum</i>	Specifies the program number to be mapped.
<i>versnum</i>	Specifies the version number of the program to be mapped.
<i>protocol</i>	Specifies the transport protocol used by the program (IPPROTO_TCP or IPPROTO_UDP).

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>

struct pmaplist *
pmap_getmaps(addr)
struct sockaddr_in *addr;
```

Parameter	Description
<i>addr</i>	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()

pmap_getport()

```
#include <rpc.h>

u_short
pmap_getport(addr, prognum, versnum, protocol)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
int protocol;
```

Parameter	Description
<i>addr</i>	Points to the internet address of the foreign host.
<i>prognum</i>	Identifies the program number to be mapped.
<i>versnum</i>	Identifies the version number of the program to be mapped.
<i>protocol</i>	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*).

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()

```
#include <rpc.h>

enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum, inproc, in, outproc, out, tout, portp)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
u_long *portp;
```

Parameter	Description
<i>addr</i>	Points to the internet address of the foreign host.
<i>prognum</i>	Identifies the remote program number.
<i>versnum</i>	Identifies the version number of the remote program.
<i>procnum</i>	Identifies the procedure to be called.
<i>inproc</i>	Identifies the XDR procedure used to encode the arguments of the remote procedure.
<i>in</i>	Points to the arguments of the remote procedure.
<i>outproc</i>	Identifies the XDR procedure used to decode the results of the remote procedure.

pmap_rmtcall()

<i>out</i>	Points to the results of the remote procedure.
<i>tout</i>	Identifies the time-out period for the remote request.
<i>portp</i>	If the call from the remote portmap service is successful, <i>portp</i> contains the port number of the triple (<i>prognum</i> , <i>versnum</i> , <i>procnum</i>).

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()

```
#include <rpc.h>

bool_t
pmap_set(prognum, versnum, protocol, port)
u_long prognum;
u_long versnum;
int protocol;
u_short port;
```

Parameter	Description
<i>prognum</i>	Identifies the local program number.
<i>versnum</i>	Identifies the version number of the local program.
<i>protocol</i>	Specifies the transport protocol used by the local program.
<i>port</i>	Identifies the port to which the local program is mapped.

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()

```
#include <rpc.h>

bool_t
pmap_unset(prognum, versnum)
u_long prognum;
u_long versnum;
```

Parameter	Description
<i>prognum</i>	Identifies the local program number.
<i>versnum</i>	Identifies the version number of the local program.

pmap_unset()

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;
```

Parameter	Description
<i>prognum</i>	The program number to register.
<i>versnum</i>	Identifies the version number to register.
<i>procnum</i>	Specifies the procedure number to register.
<i>procname</i>	Specifies the procedure that is called when the registered program is requested. <i>procname</i> must accept a pointer to its arguments, and return a static pointer to its results.
<i>inproc</i>	Specifies the XDR routine used to decode the arguments.
<i>outproc</i>	Specifies the XDR routine that encodes 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 233 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;
```

Parameter	Description
<i>xprt</i>	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: `svccraw_create()`, `svctcp_create()`, `svcudp_create()`.

svc_freeargs()

```
#include <rpc.h>
```

```
bool_t
svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Parameter	Description
<i>xprt</i>	Points to the service transport handle.
<i>inproc</i>	Specifies the XDR routine used to decode the arguments.
<i>in</i>	Points to the input arguments.

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;
```

Parameter	Description
<i>xprt</i>	Points to the service transport handle.
<i>inproc</i>	Specifies the XDR routine used to decode the arguments.
<i>in</i>	Points to the decoded arguments.

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()

svc_getcaller()

```
#include <rpc.h>

struct sockaddr_in *
svc_getcaller(xprt)
SVCXPRT *xprt;
```

Parameter	Description
<i>xprt</i>	Points to the service transport handle.

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()

```
#include <rpc.h>

void
svc_getreq(rdfs)
int rdfs;
```

Parameter	Description
<i>rdfs</i>	Specifies the read descriptor bit mask.

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.

See Also: `svc_run()`.

svc_register()

```
#include <rpc.h>

bool_t
svc_register(xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
u_long prognum;
u_long versnum;
void (*dispatch) ();
int protocol;
```

Parameter	Description
<i>xprt</i>	Points to the service transport handle.
<i>prognum</i>	Specifies the program number to be registered.
<i>versnum</i>	Specifies the version number of the program to be registered.
<i>dispatch</i>	Specifies the dispatch routine associated with <i>prognum</i> and <i>versnum</i> .

Specifies the structure of the dispatch routine is:

svc_register()

```
#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 `svccraw_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 parameters.

```
#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_getreq()`.

See Also: `registerrpc()`, `svc_getreq()`.

svc_sendreply()

```
#include <rpc.h>

bool_t
svc_sendreply(xprt, outproc, out)
SVCXPRT *xprt;
xdrproc_t outproc;
char *out;
```

Parameter	Description
<i>xprt</i>	Points to the caller's transport handle.
<i>outproc</i>	Specifies the XDR procedure used to encode the results.
<i>out</i>	Points to the results.

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()

svc_unregister()

```
#include <rpc.h>

void
svc_unregister(prognum, versnum)
u_long prognum;
u_long versnum;
```

Parameter	Description
<i>prognum</i>	Specifies the program number that is removed.
<i>versnum</i>	Specifies the version number of the program that is removed.

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()

```
#include <rpc.h>

void
svcerr_auth(xprt, why)
SVCXPRT *xprt;
enum auth_stat why;
```

Parameter	Description
<i>xprt</i>	Points to the service transport handle.
<i>why</i>	Specifies the reason the call is refused.

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_noprogram()`, `svcerr_progvers()`, `svcerr_systemerr()`, `svcerr_weakauth()`.

svcerr_decode()

```
#include <rpc.h>

void
svcerr_decode(xprt)
SVCXPRT *xprt;
```

Parameter	Description
<i>xprt</i>	Points to the service transport handle.

Description: The `svcerr_decode()` call is called by a service dispatch routine that cannot decode its parameters.

See Also: `svcerr_auth()`, `svcerr_noproc()`, `svcerr_noprogram()`, `svcerr_progvers()`, `svcerr_systemerr()`, `svcerr_weakauth()`.

svcerr_noproc()

```
#include <rpc.h>
```

```
void
svcerr_noproc(xprt)
SVCXPRT *xprt;
```

Parameter	Description
<i>xprt</i>	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_noprogram()`, `svcerr_progvers()`, `svcerr_systemerr()`, `svcerr_weakauth()`.

svcerr_noprogram()

```
#include <rpc.h>
```

```
void
svcerr_noprogram(xprt)
SVCXPRT *xprt;
```

Parameter	Description
<i>xprt</i>	Points to the service transport handle.

Description: The `svcerr_noprogram()` 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;
u_long low_vers;
u_long high_vers;
```

Parameter	Description
<i>xprt</i>	Points to the service transport handle.
<i>low_vers</i>	Specifies the low version number that does not match.
<i>high_vers</i>	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_noprogram()`, `svcerr_progvers()`, `svcerr_systemerr()`, `svcerr_weakauth()`.

svcerr_systemerr()

svcerr_systemerr()

```
#include <rpc.h>

void
svcerr_systemerr(xprt)
SVCXPRT *xprt;
```

Parameter	Description
<i>xprt</i>	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;
```

Parameter	Description
<i>xprt</i>	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 parameters.

See Also: `svcerr_auth()`, `svcerr_decode()`, `svcerr_noproc()`, `svcerr_noprog()`, `svcerr_progvers()`, `svcerr_systemerr()`.

svcrow_create()

The `svcrow_create()` call has no parameters.

```
#include <rpc.h>

SVCXPRT *
svcrow_create()
```

Description: The `svcrow_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 216 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;
u_int send_buf_size;
u_int recv_buf_size;
```

Parameter	Description
<i>sock</i>	Specifies the socket descriptor. If <i>sock</i> is <code>RPC_ANYSOCK</code> , a new socket is created. If the socket is not bound to a local TCP port, it is bound to an arbitrary port.
<i>send_buf_size</i>	Specifies the size of the send buffer. Specify 0 to choose the default.
<i>recv_buf_size</i>	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()`, `svccraw_create()`, `svccudp_create()`.

svccudp_create()

```
#include <rpc.h>

SVCXPRT *
svccudp_create(sock, sendsz, recvsz)
int sock;
u_int sendsz;
u_int recvsz;
```

Parameter	Description
<i>sock</i>	Specifies the socket descriptor. If <i>sock</i> is <code>RPC_ANYSOCK</code> , a new socket is created. If the socket is not bound to a local TCP port, it is bound to an arbitrary port.
<i>sendsz</i>	Specifies the size of the send buffer.
<i>recvsz</i>	Specifies the size of the receive buffer.

Description: The `svccudp_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()`, `svccraw_create()`, `svctcp_create()`.

xdr_accepted_reply()

xdr_accepted_reply()

```
#include <rpc.h>

bool_t
xdr_accepted_reply(xdrs, ar)
XDR *xdrs;
struct accepted_reply *ar;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>ar</i>	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()`, `registerrpc()`, `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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>arrp</i>	Specifies the address of the pointer to the array.
<i>sizep</i>	Points to the element count of the array.
<i>maxsize</i>	Specifies the maximum number of elements accepted.
<i>elsize</i>	Specifies the size of each of the array's elements, found using <code>sizeof()</code> .
<i>elproc</i>	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()

xdr_authunix_parms()

```
#include <rpc.h>

bool_t
xdr_authunix_parms(xdrs, aupp)
XDR *xdrs;
struct authunix_parms *aupp;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>aupp</i>	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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>bp</i>	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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>sp</i>	Points to a pointer to the byte string.
<i>sizep</i>	Points to the byte string size.
<i>maxsize</i>	Specifies the maximum size of the byte string.

xdr_bytes()

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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>chdr</i>	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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>cmsg</i>	Points to the call message.

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()

```
#include <rpc.h>

bool_t
xdr_double(xdrs, dp)
XDR *xdrs;
double *dp;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>dp</i>	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()

```
#include <rpc.h>

bool_t
xdr_enum(xdrs, ep)
XDR *xdrs;
enum_t *ep;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>ep</i>	Points to the enumerated number. <i>enum_t</i> can be any enumeration type such as <code>enum colors</code> , with <code>colors</code> declared as <code>enum colors (black, brown, red)</code> .

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

```
#include <rpc.h>

enum colors (black, brown, red)
void
static xdr_enum_t(xdrs, ep)
XDR *xdrs;
enum colors *ep;
{
    xdr_enum(xdrs, ep)
}
```

server RPCs.

For example, an RPC client would contain the following lines:

xdr_enum()

```
⋮  
error = callrpc(argv[1], ENUMRCVPROG, VERSION, ENUMRCVPROC, xdr_enum_t, &innumber, xdr_enum_t,  
                &outnumber);
```

```
⋮
```

An RPC server would contain the following line:

Return Values: The value 1 indicates success; the value 0 indicates an error.

```
⋮
```

```
registerrpc(ENUMRCVPROG, VERSION, ENUMRCVPROC, xdr_enum_t, xdr_enum_t);
```

```
⋮
```

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(),
registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_float()

```
#include <rpc.h>  
  
bool_t  
xdr_float(xdrs, fp)  
XDR *xdrs;  
float *fp;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>fp</i>	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.

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;  
u_int len;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>len</i>	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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>ip</i>	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()

```
#include <rpc.h>

bool_t
xdr_long(xdrs, lp)
XDR *xdrs;
long *lp;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>lp</i>	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()

xdr_opaque()

```
#include <rpc.h>

bool_t
xdr_opaque(xdrs, cp, cnt)
XDR *xdrs;
char *cp;
u_int cnt;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>cp</i>	Points to the opaque object.
<i>cnt</i>	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()

```
#include <rpc.h>

bool_t
xdr_opaque_auth(xdrs, ap)
XDR *xdrs;
struct opaque_auth *ap;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>ap</i>	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.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_pmap()

```
#include <rpc.h>

bool_t
xdr_pmap(xdrs, regs)
XDR *xdrs;
struct pmap *regs;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>regs</i>	Points to the portmap parameters.

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()

```
#include <rpc.h>

bool_t
xdr_pmaplist(xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>rp</i>	Points to a pointer to the portmap data array.

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()

```
#include <rpc.h>

bool_t
xdr_pointer(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>pp</i>	Points to a pointer.
<i>size</i>	Specifies the size of the target.
<i>proc</i>	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()

xdr_reference()

```
#include <rpc.h>

bool_t
xdr_reference(xdrs, pp, size, proc)
XDR *xdrs;
u_int size;
xdrproc_t proc;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>pp</i>	Points to a pointer.
<i>size</i>	Specifies the size of the target.
<i>proc</i>	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()

```
#include <rpc.h>

bool_t
xdr_rejected_reply(xdrs, rr)
XDR *xdrs;
struct rejected_reply *rr;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>rr</i>	Points to the rejected reply.

Description: The `xdr_rejected_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()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_replymsg()

```
#include <rpc.h>

bool_t
xdr_replymsg(xdrs, rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>rmsg</i>	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()

```
#include <rpc.h>

bool_t
xdr_short(xdrs, sp)
XDR *xdrs;
short *sp;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>sp</i>	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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>sp</i>	Points to a pointer to the string.
<i>maxsize</i>	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()

xdr_u_int()

```
#include <rpc.h>

bool_t
xdr_u_int(xdrs, up)
XDR *xdrs;
unsigned *up;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>up</i>	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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>ulp</i>	Points to the unsigned long integer.

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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>usp</i>	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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>dscmp</i>	Points to the union's discriminant. <i>enum_t</i> can be any enumeration type.
<i>unp</i>	Points to the union.
<i>choices</i>	Points to an array detailing the XDR procedure to use on each arm of the union.
<i>dfault</i>	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:

```
#include <rpc.h>

enum colors (black, brown, red);

bool_t
xdr_union(xdrs, dscmp, unp, choices, dfault)
XDR *xdrs;
enum colors *dscmp;
char *unp;
struct xdr_discrim *choices;
xdrproc_t dfault;
```

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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.

xdr_vector()

<i>basep</i>	Specifies the base of the array.
<i>nelem</i>	Specifies the element count of the array.
<i>elemsize</i>	Specifies the size of each of the array's elements, found using <code>sizeof()</code> .
<i>xdr_elem</i>	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 parameters.

```
#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* parameter of the `clnt_call` function when the user does not need to move data.

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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>sp</i>	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-parameter XDR routines, and `xdr_string()` is a three-parameter 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;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>addr</i>	Points to the memory location.
<i>size</i>	Specifies the maximum size of <i>addr</i> .
<i>op</i>	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;
u_int sendsize;
u_int recvsize;
char *handle;
int (*readit) ();
int (*writeit) ();
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>sendsize</i>	Indicates the size of the send buffer. Specify 0 to choose the default.
<i>recvsize</i>	Indicates the size of the receive buffer. Specify 0 to choose the default.
<i>handle</i>	Specifies the first parameter passed to <code>readit()</code> and <code>writeit()</code> .
<i>readit()</i>	Called when a stream's input buffer is empty.
<i>writeit()</i>	Called when a stream's output buffer is full.

Description: The `xdrrec_create()` call initializes the XDR stream pointed to by *xdrs*.

Notes:

1. The *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.

xdrrec_endofrecord()

xdrrec_endofrecord()

```
#include <rpc.h>
bool_t
xdrrec_endofrecord(xdrs, sendnow)
XDR *xdrs;
int sendnow;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.
<i>sendnow</i>	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.

xdrrec_eof()

```
#include <rpc.h>
bool_t
xdrrec_eof(xdrs)
XDR *xdrs;
```

Parameter	Description
<i>xdrs</i>	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()

```
#include <rpc.h>
bool_t
xdrrec_skiprecord(xdrs)
XDR *xdrs;
```

Parameter	Description
<i>xdrs</i>	Points to an XDR stream.

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()

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

void
xdrstdio_create(xdrs, file, op)
XDR *xdrs;
FILE *file;
enum xdr_op op;
```

Parameter	Description
-----------	-------------

<i>xdrs</i>	Points to an XDR stream.
<i>file</i>	Specifies the file name for the I/O stream.
<i>op</i>	Determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

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()

```
#include <rpc.h>

void
xprt_register(xprt)
SVCXPRT *xprt;
```

Parameter	Description
-----------	-------------

<i>xprt</i>	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;
```

Parameter	Description
-----------	-------------

<i>xprt</i>	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`.

See also: `svc_fds`.

Sample RPC Programs

This appendix provides examples of the following programs:

- RPC client (see topic 246)
 - RPC server (see topic 246)
 - RPC raw data stream (see topic 248)
-

RPC 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>
#include <socket.h>

#define intrcvprog ((u_long)150000)
#define version    ((u_long)1)
#define intrcvproc ((u_long)1)

main(argc, argv)
    int argc;
    char *argv[];
{
    int innumber;
    int outnumber;
    int error;

    if (argc != 3) {
        fprintf(stderr, "usage: %s hostname integer\n", argv[0]);
        exit (-1);
    } /* endif */
    innumber = atoi(argv[2]);
    /*
     * Send the integer to the server. The server should
     * return the same integer.
     */
    error = callrpc(argv[1], intrcvprog, version, intrcvproc, xdr_int,
                   (char *)&innumber, xdr_int, (char *)&outnumber);

    if (error != 0) {
        fprintf(stderr, "error: callrpc failed: %d \n", error);
        fprintf(stderr, "intrcvprog: %d version: %d intrcvproc: %d",
                intrcvprog, version, intrcvproc);
        exit(1);
    } /* endif */

    printf("value sent: %d   value received: %d\n", innumber, outnumber);
    exit(0);
}
```

RPC Server

The following is an example of an RPC server program.

```
/* GENERIC SERVER      */
/* RECEIVE AN INTEGER OR FLOAT AND RETURN THEM RESPECTIVELY */
/* PORTMAPPER MUST BE RUNNING */

#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);
    printf("Intrcv Registration with Port Mapper completed\n");

    /*OR MULTIPLE PROGRAMS*/
    registerrpc(fltrcvprog,fltvers,fltrcvproc,floatrcv,xdr_float,xdr_float);
    printf("Floatrcv Registration with Port Mapper completed\n");

    /*
     * svc_run will handle all requests for programs registered.
     */
    svc_run();
    printf("Error:svc_run returned!\n");
    exit(1);
}

/*
 * Procedure called by the server to receive and return an integer.
 */
int *
intrcv(in)
    int *in;
{
    int *out;

    printf("integer received: %d\n",*in);
    out = in;
    printf("integer being returned: %d\n",*out);
    return (out);
}

/*
 * Procedure called by the server to receive and return a float.
 */
float *
floatrcv(in)
    float *in;
{
    float *out;

    printf("float received: %e\n",*in);
    out=in;
    printf("float being returned: %e\n",*out);
    return(out);
}

```

RPC Raw Data Stream

The following is an example of an RPC raw data stream program.

```

/*RAWEX                                     */
/* AN EXAMPLE OF THE RAW CLIENT/SERVER USAGE */
/* PORTMAPPER MUST BE RUNNING              */
/*
 * 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\n", argv[0]);
        exit(-1);
    }
    in = atoi(argv[1]);

    /*
     * Create the raw transport handle for the server.
     */
    transp = svcraw_create();
    if (transp == NULL) {
        fprintf(stderr, "can't create an RPC server transport\n");
        exit(-1);
    }

    /* In case the program is already registered, deregister it */
    pmap_unset(rawprog, rawvers);

    /* Register the server program with PORTMAPPER */
    if (!svc_register(transp,rawprog,rawvers,raw2, 0)) {
        fprintf(stderr, "can't register service\n");
        exit(-1);
    }
    /*
     * The following registers the transport handle with internal

```

```

    * data structures.
    */
xprt_register(transp);

/*
 * Create the client transport handle.
 */
if ((clnt = clntraw_create(rawprog, rawvers)) == NULL ) {
    clnt_pcreateerror("clntudp_create");
    exit(-1);
}
total_timeout.tv_sec = 60;
total_timeout.tv_usec = 0;
printf("Argument:  %d\n",in);

/*
 * Make the call from the client to the server.
 */
cs=clnt_call(clnt,rawproc,xdr_int,
             (char *)&in,xdr_int,(char *)&bout,total_timeout);

printf("Result:  %d",bout);
if(cs!=0) {
    clnt_perror(clnt,"Client call failed");
    exit(1);
}
exit(0);
}

/*
 * Service procedure called by the server when it receives the client
 * request.
 */
void raw2(rqstp,transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
    int in,out;
    if (rqstp->rq_proc=rawproc) {
        /*
         * Unpack the integer passed by the client.
         */
        svc_getargs(transp,xdr_int,&in);
        printf("Received:  %d\n",in);
        /*
         * Send the integer back to the client.
         */
        out=in;
        printf("Sent:  %d\n",out);
        if (!svc_sendreply(transp, xdr_int,&out)) {
            printf("Can't reply to RPC call.\n");
            exit(1);
        }
    }
}
}

```

RPC Raw Data Stream

Chapter 7. 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 447.

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)
- OLDXLIB 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.

X Window System Interface

Figure 29 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

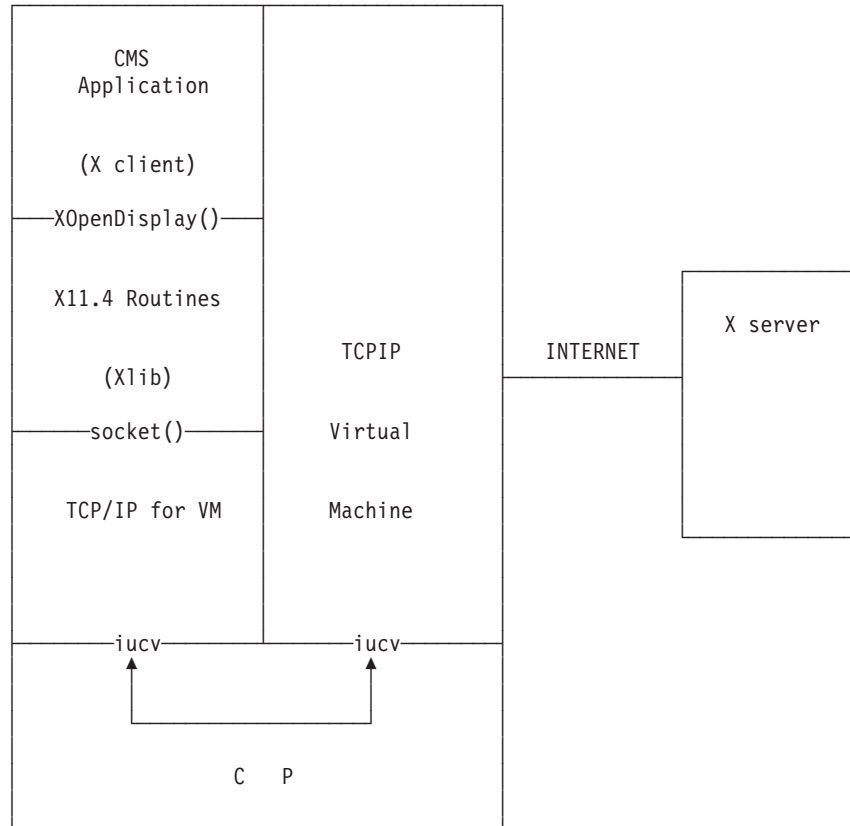


Figure 29. VM X Window System Application to Server

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 30 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 30. 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—▶▶  
  
▶▶┌.target_screen└▶▶
```

Parameter	Description
<i>internet_address</i>	Specifies the internet address of the host machine on which the X Window System server is running.
<i>target_server</i>	Specifies the number of the display server on the host machine.
<i>target_screen</i>	Specifies the screen to be used on the same target server.

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(IBM CPP))
```


X Window System Interface

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

GLOBAL TXTLIB XMLIB XTLIB X11LIB COMMTXT CMSLIB

Table 23 describes the various TXTLIBs associated with the X Window System support in the VM/CMS environment.

Table 23. TXTLIBs Associated with X Window System Support

TXTLIB	Routine(s)
X11LIB	X11.4 X Window System client routines and X11.4 Miscellaneous Utility routines
OLDXLIB	X10 Compatibility routines
XTLIB	X11.4 Intrinsic routines
XAWLIB	Athena widget routines
XMLIB	OSF/Motif-based widget routines

X Window System Subroutines

This section provides information about X Window System in tabular form for quick reference.

The following tables list the subroutines supported by TCP/IP Level 320 for VM.

The subroutines are grouped according to the type of function provided.

Opening and Closing a Display

Table 24 provides the subroutines for opening and closing a display.

Table 24. Opening and Closing Display

Subroutine	Description
XCloseDisplay()	Closes a display.
XFree()	Frees in-memory data created by Xlib function.
XNoOp()	Executes a NoOperation protocol request.
XOpenDisplay()	Opens a display.

Creating and Destroying Windows

Table 25 provides the subroutines for creating and destroying windows.

Table 25. Creating and Destroying Windows

Subroutine	Description
XConfigureWindow()	Configures the specified window.
XCreateSimpleWindow()	Creates unmapped InputOutput subwindow.
XCreateWindow()	Creates unmapped subwindow.
XDestroySubwindows()	Destroys all subwindows of specified window.
XDestroyWindow()	Unmaps and destroys window and all subwindows.

Manipulating Windows

Table 26 provides the subroutines for manipulating windows.

Table 26. Manipulating Windows

Subroutine	Description
<code>XCirculateSubwindows()</code>	Circulates a subwindow up or down.
<code>XCirculateSubwindowsUp()</code>	Raises the lowest mapped child of the window.
<code>XCirculateSubwindowsDown()</code>	Lowers the highest mapped child of the window.
<code>XIconifyWindow()</code>	Sends a <code>WM_CHANGE_STATE</code> ClientMessage to the root window of the specified screen.
<code>XLowerWindow()</code>	Lowers the specified window.
<code>XMapRaised()</code>	Maps and raises the specified window.
<code>XMapSubwindows()</code>	Maps all subwindows of the specified window.
<code>XMapWindow()</code>	Maps the specified window.
<code>XMoveResizeWindow()</code>	Changes the specified window's size and location.
<code>XMoveWindow()</code>	Moves the specified window.
<code>XRaiseWindow()</code>	Raises the specified window.
<code>XReconfigureWMWindow()</code>	Issues a <code>ConfigureWindow</code> request on the specified top-level window.
<code>XResizeWindow()</code>	Changes the specified window's size.
<code>XRestackWindows()</code>	Restacks a set of windows from top to bottom.
<code>XSetWindowBorderWidth()</code>	Changes the border width of the window.
<code>XUnmapSubwindows()</code>	Unmaps all subwindows of the specified window.
<code>XUnmapWindow()</code>	Unmaps the specified window.
<code>XWithdrawWindow()</code>	Unmaps the specified window and sends a synthetic <code>UnmapNotify</code> event to the root window of the specified screen.

Changing Window Attributes

Table 27 provides the subroutines for changing window attributes.

Table 27. Changing Window Attributes

Subroutine	Description
<code>XChangeWindowAttributes()</code>	Changes one or more window attributes.
<code>XSetWindowBackground()</code>	Sets the window's background to a specified pixel.
<code>XSetWindowBackgroundPixmap()</code>	Sets the window's background to a specified pixmap.
<code>XSetWindowBorder()</code>	Changes the window's border to a specified pixel.

X Window System Interface

Table 27. Changing Window Attributes (continued)

Subroutine	Description
XSetWindowBorderPixmap()	Changes the window's border tile.
XTranslateCoordinates()	Transforms coordinates between windows.

Obtaining Window Information

Table 28 provides the subroutines for obtaining window information.

Table 28. Obtaining Window Information

Subroutine	Description
XGetGeometry()	Gets the current geometry of the specified drawable.
XGetWindowAttributes()	Gets the current attributes for the specified window.
XQueryPointer()	Gets the pointer coordinates and the root window.
XQueryTree()	Obtains the IDs of the children and parent windows.

Obtaining Properties and Atoms

Table 29 provides the subroutines for obtaining properties and atoms.

Table 29. Properties and Atoms

Subroutine	Description
XGetAtomName()	Gets a name for the specified atom ID.
XInternAtom()	Gets an atom for the specified name.

Manipulating Window Properties

Table 30 provides the subroutines for manipulating the properties of windows.

Table 30. Manipulating Window Properties

Subroutine	Description
XChangeProperty()	Changes the property for the specified window.
XDeleteProperty()	Deletes a property for the specified window.
XGetWindowProperty()	Gets the atom type and property format for the window.
XListProperties()	Gets the specified window's property list.
XRotateWindowProperties()	Rotates the properties in a property array.

Setting Window Selections

Table 31 provides the subroutines for setting window selections.

Table 31. Setting Window Selections

Subroutine	Description
XConvertSelection()	Converts a selection.

Table 31. Setting Window Selections (continued)

Subroutine	Description
XGetSelectionOwner()	Gets the selection owner.
XSetSelectionOwner()	Sets the selection owner.

Manipulating Colormaps

Table 32 provides the subroutines for manipulating color maps.

Table 32. Manipulating Colormaps

Subroutine	Description
XAllocStandardColormap()	Allocates an XStandardColormap structure.
XCopyColormapAndFree()	Creates a new colormap from a specified colormap.
XCreateColormap()	Creates a colormap.
XFreeColormap()	Frees the specified colormap.
XQueryColor()	Queries the RGB value for a specified pixel.
XQueryColors()	Queries the RGB values for an array of pixels.
XSetWindowColormap()	Sets the colormap of the specified window.

Manipulating Color Cells

Table 33 provides the subroutines for manipulating color cells.

Table 33. Manipulating Color Cells

Subroutine	Description
XAllocColor()	Allocates a read-only color cell.
XAllocColorCells()	Allocates read/write color cells.
XAllocColorPlanes()	Allocates read/write color resources.
XAllocNamedColor()	Allocates a read-only color cell by name.
XFreeColors()	Frees colormap cells.
XLookupColor()	Looks up a colorname.
XStoreColor()	Stores an RGB value into a single colormap cell.
XStoreColors()	Stores RGB values into colormap cells.
XStoreNamedColor()	Sets a pixel color to the named color.

Creating and Freeing Pixmaps

Table 34 provides the subroutines for creating and freeing pixmaps.

Table 34. Creating and Freeing Pixmaps

Subroutine	Description
XCreatePixmap()	Creates a pixmap of a specified size.
XFreePixmap()	Frees all storage associated with specified pixmap.

Manipulating Graphics Contexts

Table 35 provides the subroutines for manipulating graphics contexts.

Table 35. Manipulating Graphics Contexts

Subroutine	Description
XChangeGC()	Changes the components in the specified Graphics Context (GC).
XCopyGC()	Copies the components from a source GC to a destination GC.
XCreateGC()	Creates a new GC.
XFreeGC()	Frees the specified GC.
XGetGCValues()	Returns the GC values in the specified structure.
XGContextFromGC()	Obtains the GContext resource ID for GC.
XQueryBestTile()	Gets the best fill tile shape.
XQueryBestSize()	Gets the best size tile, stipple, or cursor.
XQueryBestStipple()	Gets the best stipple shape.
XSetArcMode()	Sets the arc mode of the specified GC.
XSetBackground()	Sets the background of the specified GC.
XSetClipmask()	Sets the clip_mask of the specified GC to a specified pixmap.
XSetClipOrigin()	Sets the clip origin of the specified GC.
XSetClipRectangles()	Sets the clip_mask of GC to a list of rectangles.
XSetDashes()	Sets the dashed line style components of a specified GC.
XSetFillRule()	Sets the fill rule of the specified GC.
XSetFillStyle()	Sets the fill style of the specified GC.
XSetFont()	Sets the current font of the specified GC.
XSetForeground()	Sets the foreground of the specified GC.
XSetFunction()	Sets display function in the specified GC.
XSetGraphicsExposures()	Sets the graphics-exposure flag of the specified GC.
XSetLineAttributes()	Sets the line-drawing components of the GC.
XSetPlaneMask()	Sets the plane mask of the specified GC.
XSetState()	Sets the foreground, background, plane mask, and function in GC.
XSetStipple()	Sets the stipple of the specified GC.
XSetSubwindowMode()	Sets the subwindow mode of the specified GC.
XSetTile()	Sets the fill tile of the specified GC.
XSetTSTOrigin()	Sets the tile or stipple origin of the specified GC.

Clearing and Copying Areas

Table 36 provides the subroutines for clearing and copying areas.

Table 36. *Clearing and Copying Areas*

Subroutine	Description
XClearArea()	Clears a rectangular area of window.
XClearWindow()	Clears the entire window.
XCopyArea()	Copies the drawable area between drawables of the same root and the same depth.
XCopyPlane()	Copies single bit-plane of the drawable.

Drawing Lines

Table 37 provides the subroutines for drawing lines.

Table 37. *Drawing Lines*

Subroutine	Description
XDraw()	Draws an arbitrary polygon or curve that is defined by the specified list of Vertices as specified in <i>vlist</i> .
XDrawArc()	Draws a single arc in the drawable.
XDrawArcs()	Draws multiple arcs in a specified drawable.
XDrawFilled()	Draws arbitrary polygons or curves and then fills them.
XDrawLine()	Draws a single line between two points in a drawable.
XDrawLines()	Draws multiple lines in the specified drawable.
XDrawPoint()	Draws a single point in the specified drawable.
XDrawPoints()	Draws multiple points in the specified drawable.
XDrawRectangle()	Draws an outline of a single rectangle in the drawable.
XDrawRectangles()	Draws an outline of multiple rectangles in the drawable.
XDrawSegments()	Draws multiple line segments in the specified drawable.

Filling Areas

Table 38 provides the subroutines for filling areas.

Table 38. *Filling Areas*

Subroutine	Description
XFillArc()	Fills a single arc in drawable.
XFillArcs()	Fills multiple arcs in drawable.
XFillPolygon()	Fills a polygon area in the drawable.

X Window System Interface

Table 38. Filling Areas (continued)

Subroutine	Description
XFillRectangle()	Fills a single rectangular area in the drawable.
XFillRectangles()	Fills multiple rectangular areas in the drawable.

Loading and Freeing Fonts

Table 39 provides the subroutines for loading and freeing fonts.

Table 39. Loading and Freeing Fonts

Subroutine	Description
XFreeFont()	Unloads the font and frees the storage used by the font.
XFreeFontInfo()	Frees the font information array.
XFreeFontNames()	Frees a font name array.
XFreeFontPath()	Frees data returned by XGetFontPath.
XGetFontPath()	Gets the current font search path.
XGetFontProperty()	Gets the specified font property.
XListFontsWithInfo()	Gets names and information about loaded fonts.
XLoadFont()	Loads a font.
XLoadQueryFont()	Loads and queries font in one operation.
XListFonts()	Gets a list of available font names.
XQueryFont()	Gets information about a loaded font.
XSetFontPath()	Sets the font search path.
XUnloadFont()	Unloads the specified font.

Querying Character String Sizes

Table 40 provides the subroutines for querying the character size of a string.

Table 40. Querying Character String Sizes

Subroutine	Description
XFreeStringList()	Frees the in-memory data associated with the specified string list.
XQueryTextExtents()	Gets a 1-byte character string bounding box from the server.
XQueryTextExtents16()	Gets a 2-byte character string bounding box from the server.
XTextExtents()	Gets a bounding box of a 1-byte character string.
XTextExtents16()	Gets a bounding box of a 2-byte character string.
XTextPropertyToStringList()	Returns a list of strings representing the elements of the specified XTextProperty structure.

Table 40. Querying Character String Sizes (continued)

Subroutine	Description
XTextWidth()	Gets the width of an 8-bit character string.
XTextWidth16()	Gets the width of a 2-byte character string.

Drawing Text

Table 41 provides the subroutines for drawing text.

Table 41. Drawing Text

Subroutine	Description
XDrawImageString()	Draws 8-bit image text in the specified drawable.
XDrawImageString16()	Draws 2-byte image text in the specified drawable.
XDrawString()	Draws 8-bit text in the specified drawable.
XDrawString16()	Draws 2-byte text in the specified drawable.
XDrawText()	Draws 8-bit complex text in the specified drawable.
XDrawText16()	Draws 2-byte complex text in the specified drawable.

Transferring Images

Table 42 provides the subroutines for transferring images.

Table 42. Transferring Images

Subroutine	Description
XGetImage()	Gets the image from the rectangle in the drawable.
XGetSubImage()	Copies the rectangle on the display to image.
XPutImage()	Puts the image from memory into the rectangle in the drawable.

Manipulating Cursors

Table 43 provides the subroutines for manipulating cursors.

Table 43. Manipulating Cursors

Subroutine	Description
XCreateFontCursor()	Creates a cursor from a standard font.
XCreateGlyphCursor()	Creates a cursor from font glyphs.
XDefineCursor()	Defines a cursor for a window.
XFreeCursor()	Frees a cursor.
XQueryBestCursor()	Gets useful cursor sizes.
XRecolorCursor()	Changes the color of a cursor.
XUndefineCursor()	Undefines a cursor for a window.

Handling Window Manager Functions

Table 44 provides the subroutines for handling the window manager functions.

Table 44. Handling Window Manager Functions

Subroutine	Description
XAddToSaveSet()	Adds a window to the client's save-set.
XAllowEvents()	Allows events to be processed after a device is frozen.
XChangeActivePointerGrab()	Changes the active pointer grab.
XChangePointerControl()	Changes the interactive feel of the pointer device.
XChangeSaveSet()	Adds or removes a window from the client's save-set.
XGetInputFocus()	Gets the current input focus.
XGetPointerControl()	Gets the current pointer parameters.
XGrabButton()	Grabs a mouse button.
XGrabKey()	Grabs a single key of the keyboard.
XGrabKeyboard()	Grabs the keyboard.
XGrabPointer()	Grabs the pointer.
XGrabServer()	Grabs the server.
XInstallColormap()	Installs a colormap.
XKillClient()	Removes a client.
XListInstalledColormaps()	Gets a list of currently installed colormaps.
XRemoveFromSaveSet()	Removes a window from the client's save-set.
XReparentWindow()	Changes the parent of a window.
XSetCloseDownMode()	Changes the close down mode.
XSetInputFocus()	Sets the input focus.
XUngrabButton()	Ungrabs a mouse button.
XUngrabKey()	Ungrabs a key.
XUngrabKeyboard()	Ungrabs the keyboard.
XUngrabPointer()	Ungrabs the pointer.
XUngrabServer()	Ungrabs the server.
XUninstallColormap()	Uninstalls a colormap.
XWarpPointer()	Moves the pointer to arbitrary point on the screen.

Manipulating Keyboard Settings

Table 45 provides the subroutines for manipulating keyboard settings.

Table 45. Manipulating Keyboard Settings

Subroutine	Description
XAutoRepeatOff()	Turns off the keyboard auto-repeat.
XAutoRepeatOn()	Turns on the keyboard auto-repeat.
XBell()	Sets the volume of the bell.

Table 45. Manipulating Keyboard Settings (continued)

Subroutine	Description
XChangeKeyboardControl()	Changes the keyboard settings.
XChangeKeyboardMapping()	Changes the mapping of symbols to keycodes.
XDeleteModifiermapEntry()	Deletes an entry from the XModifierKeymap structure.
XFreeModifiermap()	Frees XModifierKeymap structure.
XGetKeyboardControl()	Gets the current keyboard settings.
XGetKeyboardMapping()	Gets the mapping of symbols to keycodes.
XGetModifierMapping()	Gets keycodes to be modifiers.
XGetPointerMapping()	Gets the mapping of buttons on the pointer.
XInsertModifiermapEntry()	Adds an entry to the XModifierKeymap structure.
XNewModifiermap()	Creates the XModifierKeymap structure.
XQueryKeymap()	Gets the state of the keyboard keys.
XSetPointerMapping()	Sets the mapping of buttons on the pointer.
XSetModifierMapping()	Sets keycodes to be modifiers.

Controlling the Screen Saver

Table 46 provides the subroutines for controlling the screen saver.

Table 46. Controlling the Screen Saver

Subroutine	Description
XActivateScreenSaver()	Activates the screen saver.
XForceScreenSaver()	Turns the screen saver on or off.
XGetScreenSaver()	Gets the current screen saver settings.
XResetScreenSaver()	Resets the screen saver.
XSetScreenSaver()	Sets the screen saver.

Manipulating Hosts and Access Control

Table 47 provides the subroutines for manipulating hosts and toggling the access control.

Table 47. Manipulating Hosts and Access Control

Subroutine	Description
XDisableAccessControl()	Disables access control.
XEnableAccessControl()	Enables access control.
XListHosts()	Gets the list of hosts.
XSetAccessControl()	Changes access control.

X Window System Interface

Handling Events

Table 48 provides the subroutines for handling events.

Table 48. Handling Events

Subroutine	Description
XCheckIfEvent()	Checks event queue for the specified event without blocking.
XCheckMaskEvent()	Removes the next event that matches a specified mask without blocking.
XCheckTypedEvent()	Gets the next event that matches event type.
XCheckTypedWindowEvent()	Gets the next event for the specified window.
XCheckWindowEvent()	Removes the next event that matches the specified window and mask without blocking.
XEventsQueued()	Checks the number of events in the event queue.
XFlush()	Flushes the output buffer.
XGetMotionEvents()	Gets the motion history for the specified window.
XIfEvent()	Checks the event queue for the specified event and removes it.
XMaskEvent()	Removes the next event that matches a specified mask.
XNextEvent()	Gets the next event and removes it from the queue.
XPeekEvent()	Peeks at the event queue.
XPeekIfEvent()	Checks the event queue for the specified event.
XPending()	Returns the number of events that are pending.
XPutBackEvent()	Pushes the event back to the top of the event queue.
XSelectInput()	Selects events to be reported to the client.
XSendEvent()	Sends an event to a specified window.
XSync()	Flushes the output buffer and waits until all requests are completed.
XWindowEvent()	Removes the next event that matches the specified window and mask.

Enabling and Disabling Synchronization

Table 49 provides the subroutines for toggling synchronization.

Table 49. Enabling and Disabling Synchronization

Subroutine	Description
XSetAfterFunction()	Sets the previous after function.
XSynchronize()	Enables or disables synchronization.

Using Default Error Handling

Table 50 provides the subroutines for using the default error handling.

Table 50. Using Default Error Handling

Subroutine	Description
XDisplayName()	Gets the name of the display currently being used.
XGetErrorText()	Gets the error text for the specified error code.
XGetErrorDatabaseText()	Gets the error text from the error database.
XSetErrorHandler()	Sets the error handler.
XSetIOErrorHandler()	Sets the error handler for unrecoverable I/O errors.

Communicating with Window Managers

Table 51 provides the subroutines for communicating with window managers.

Table 51. Communicating with Window Managers

Subroutine	Description
XAllocClassHints()	Allocates storage for an XClassHint structure.
XAllocIconSize()	Allocates storage for an XIconSize structure.
XAllocSizeHints()	Allocates storage for an XSizeHints structure.
XAllocWMHints()	Allocates storage for an XWMHints structure.
XGetClassHint()	Gets the class of a window.
XFetchName()	Gets the name of a window.
XGetCommand()	Gets a window's WM_COMMAND property.
XGetIconName()	Gets the name of an icon window.
XGetIconSizes()	Gets the values of icon size atom.
XGetNormalHints()	Gets size hints for window in normal state.
XGetRGBColormaps()	Gets colormap associated with specified atom.
XGetSizeHints()	Gets the values of type WM_SIZE_HINTS properties.
XGetStandardColormap()	Gets colormap associated with specified atom.
XGetTextProperty()	Gets a window's property of type TEXT.
XGetTransientForHint()	Gets WM_TRANSIENT_FOR property for window.
XGetWMClientMachine()	Gets the value of a window's WM_CLIENT_MACHINE property.
XGetWMColormapWindows()	Gets the value of a window's WM_COLORMAP_WINDOWS property.
XGetWMHints()	Gets the value of the window manager's hints atom.
XGetWMName()	Gets the value of the WM_NAME property.
XGetWMIconName()	Gets the value of the WM_ICON_NAME property.

X Window System Interface

Table 51. Communicating with Window Managers (continued)

Subroutine	Description
XGetWMNormalHints()	Gets the value of the window manager's hints atom.
XGetWMProtocols()	Gets the value of a window's WM_PROTOCOLS property.
XGetWMSizeHints()	Gets the values of type WM_SIZE_HINTS properties.
XGetZoomHints()	Gets values of the zoom hints atom.
XSetCommand()	Sets the value of the command atom.
XSetClassHint()	Sets the class of a window.
XSetIconName()	Assigns a name to an icon window.
XSetIconSizes()	Sets the values of icon size atom.
XSetNormalHints()	Sets size hints for a window in normal state.
XSetRGBColormaps()	Sets the colormap associated with the specified atom.
XSetSizeHints()	Sets the values of the type WM_SIZE_HINTS properties.
XSetStandardColormap()	Sets the colormap associated with the specified atom.
XSetStandardProperties()	Specifies a minimum set of properties.
XSetTextProperty()	Sets a window's properties of type TEXT.
XSetTransientForHint()	Sets WM_TRANSIENT_FOR property for window.
XSetWMClientMachine()	Sets a window's WM_CLIENT_MACHINE property.
XSetWMColormapWindows()	Sets a window's WM_COLORMAP_WINDOWS property.
XSetWMHints()	Sets the value of the window manager's hints atom.
XSetWMIconName()	Sets the value of the WM_ICON_NAME property.
XSetWMName()	Sets the value of the WM_NAME property.
XSetWMNormalHints()	Sets the value of the window manager's hints atom.
XSetWMProperties()	Sets the values of properties for a window manager.
XSetWMProtocols()	Sets the value of the WM_PROTOCOLS property.
XSetWMSizeHints()	Sets the values of type WM_SIZE_HINTS properties.
XSetZoomHints()	Sets the values of the zoom hints atom.
XStoreName()	Assigns a name to a window.

Manipulating Keyboard Event Functions

Table 52 on page 269 provides the subroutines for manipulating the functions of keyboard events.

Table 52. Keyboard Event Functions

Subroutine	Description
XKeycodeToKeysym()	Converts a keycode to a keysym value.
XKeysymToKeycode()	Converts a keysym value to keycode.
XKeysymToString()	Converts a keysym value to keysym name.
XLookupKeysym()	Translates a keyboard event into a keysym value.
XLookupMapping()	Gets the mapping of a keyboard event from a keymap file.
XLookupString()	Translates the keyboard event into a character string.
XRebindCode()	Changes the keyboard mapping in the keymap file.
XRebindKeysym()	Maps the character string to a specified keysym and modifiers.
XRefreshKeyboardMapping()	Refreshes the stored modifier and keymap information.
XStringToKeysym()	Converts the keysym name to the keysym value.
XUseKeymap()	Changes the keymap files.
XGeometry()	Parses window geometry given padding and font values.
XGetDefault()	Gets the default window options.
XParseColor()	Obtains RGB values from color name.
XParseGeometry()	Parses standard window geometry options.
XWMGeometry()	Obtains a window's geometry information.

Manipulating Regions

Table 53 provides the subroutines for manipulating regions.

Table 53. Manipulating Regions

Subroutine	Description
XClipBox()	Generates the smallest enclosing rectangle in the region.
XCreateRegion()	Creates a new empty region.
XEmptyRegion()	Determines whether a specified region is empty.
XEqualRegion()	Determines whether two regions are the same.
XIntersectRegion()	Computes the intersection of two regions.
XDestroyRegion()	Frees storage associated with the specified region.
XOffsetRegion()	Moves the specified region by the specified amount.
XPointInRegion()	Determines if a point lies in the specified region.
XPolygonRegion()	Generates a region from points.

X Window System Interface

Table 53. Manipulating Regions (continued)

Subroutine	Description
XRectInRegion()	Determines if a rectangle lies in the specified region.
XSetRegion()	Sets the GC to the specified region.
XShrinkRegion()	Reduces the specified region by a specified amount.
XSubtractRegion()	Subtracts two regions.
XUnionRegion()	Computes the union of two regions.
XUnionRectWithRegion()	Creates a union of source region and rectangle.
XXorRegion()	Gets the difference between the union and intersection of regions.

Using Cut and Paste Buffers

Table 54 provides the subroutines for using cut and paste buffers.

Table 54. Using Cut and Paste Buffers

Subroutine	Description
XFetchBuffer()	Gets data from a specified cut buffer.
XFetchBytes()	Gets data from the first cut buffer.
XRotateBuffers()	Rotates the cut buffers.
XStoreBuffer()	Stores data in a specified cut buffer.
XStoreBytes()	Stores data in first cut buffer.

Querying Visual Types

Table 55 provides the subroutines for querying visual types.

Table 55. Querying Visual Types

Subroutine	Description
XGetVisualInfo()	Gets a list of visual information structures.
XListDepths()	Determines the number of depths that are available on a given screen.
XListPixmapFormats()	Gets the pixmap format information for a given display.
XMatchVisualInfo()	Gets visual information matching screen depth and class.
XPixmapFormatValues()	Gets the pixmap format information for a given display.

Manipulating Images

Table 56 provides the subroutines for manipulating images.

Table 56. Manipulating Images

Subroutine	Description
XAddPixel()	Increases each pixel in a pixmap by a constant value.
XCreateImage()	Allocates memory for the XImage structure.
XDestroyImage()	Frees memory for the XImage structure.
XGetPixel()	Gets a pixel value in an image.
XPutPixel()	Sets a pixel value in an image.
XSubImage()	Creates an image that is a subsection of a specified image.

Manipulating Bitmaps

Table 57 provides the subroutines for manipulating bitmaps.

Table 57. Manipulating Bitmaps

Subroutine	Description
XCreateBitmapFromData()	Includes a bitmap in the C program.
XCreatePixmapFromBitmapData()	Creates a pixmap using bitmap data.
XDeleteContext()	Deletes data associated with the window and context type.
XFindContext()	Gets data associated with the window and context type.
XReadBitmapFile()	Reads in a bitmap from a file.
XSaveContext()	Stores data associated with the window and context type.
XUniqueContext()	Allocates a new context.
XWriteBitmapFile()	Writes out a bitmap to a file.

Using the Resource Manager

Table 58 provides the subroutines for using the resource manager.

Table 58. Using the Resource Manager

Subroutine	Description
Xpermalloc()	Allocates memory that is never freed.
XrmDestroyDatabase()	Destroys a resource database and frees its allocated memory.
XrmGetFileDatabase()	Creates a database from a specified file.
XrmGetResource()	Retrieves a resource from a database.
XrmGetStringDatabase()	Creates a database from a specified string.
XrmInitialize()	Initializes the resource manager.
XrmMergeDatabases()	Merges two databases.
XrmParseCommand()	Stores command options in a database.
XrmPutFileDatabase()	Copies the database into a specified file.

X Window System Interface

Table 58. Using the Resource Manager (continued)

Subroutine	Description
XrmPutLineResource()	Stores a single resource entry in a database.
XrmPutResource()	Stores a resource in a database.
XrmPutStringResource()	Stores string resource in a database.
XrmQGetResource()	Retrieves a quark from a database.
XrmQGetSearchList()	Gets a resource search list of database levels.
XrmQGetSearchResource()	Gets a quark search list of database levels.
XrmQPutResource()	Stores binding and quarks in a database.
XrmQPutStringResource()	Stores string binding and quarks in a database.
XrmQuarkToString()	Converts a quark to a character string.
XrmStringToQuark()	Converts a character string to a quark.
XrmStringToQuarkList()	Converts character strings to a quark list.
XrmStringToBindingQuarkList()	Converts strings to bindings and quarks.
XrmUniqueQuark()	Allocates a new quark.

Manipulating Display Functions

Table 59 provides the subroutines for manipulating the display functions.

Table 59. Display Functions

Subroutine	Description
AllPlanes() XAllPlanes()	Returns all bits suitable for use in plane argument.
BitMapBitOrder() XBitMapOrder()	Returns either the most or least significant bit in each bitmap unit.
BitMapPad() XBitMapPad()	Returns the multiple of bits padding each scanline.
BitMapUnit() XBitMapUnit()	Returns the size of a bitmap's unit in bits.
BlackPixel() XBlackPixel()	Returns the black pixel value of the screen specified.
BlackPixelOfScreen() XBlackPixelOfScreen()	Returns the black pixel value of the screen specified.
CellsOfScreen() XCellsOfScreen()	Returns the number of colormap cells.
ConnectionNumber() XConnectionNumber()	Returns the file descriptor of the connection.
CreatePixmapCursor() XCreatePixmapCursor()	Creates a pixmap of a specified size.
CreateWindow() XCreateWindow()	Creates an unmapped subwindow for a specified parent window.
DefaultColormap() XDefaultColormap()	Returns a default colormap ID for allocation on the screen specified.
DefaultColormapOfScreen() XDefaultColormapOfScreen()	Returns the default colormap ID of the screen specified.
DefaultDepth() XDefaultDepth()	Returns the depth of the default root window.

Table 59. Display Functions (continued)

Subroutine	Description
DefaultDepthOfScreen() XDefaultDepthOfScreen()	Returns the default depth of the screen specified.
DefaultGC() XDefaultGC()	Returns the default GC of the default root window.
DefaultGCOfScreen() XDefaultGCOfScreen()	Returns the default GC of the screen specified.
DefaultScreen() XDefaultScreen()	Obtains the default screen referenced in the XOpenDisplay routine.
DefaultScreenofDisplay() XDefaultScreenofDisplay()	Returns the default screen of the display specified.
DefaultRootWindow() XDefaultRootWindow()	Obtains the root window for the default screen specified.
DefaultVisual() XDefaultVisual()	Returns the default visual type of the screen specified.
DefaultVisualOfScreen() XDefaultVisualOfScreen()	Returns the default visual type of the screen specified.
DisplayCells() XDisplayCells()	Displays the number of entries in the default colormap.
DisplayHeight() XDisplayHeight()	Displays the height of the screen in pixels.
DisplayHeightMM() XDisplayHeightMM()	Displays the height of the screen in millimeters.
DisplayOfScreen() XDisplayOfScreen()	Displays the type of screen specified.
DisplayPlanes() XDisplayPlanes()	Displays the depth (number of planes) of the root window of the screen specified.
DisplayString() XDisplayString()	Displays the string passed to XOpenDisplay when the current display was opened.
DisplayWidth() XDisplayWidth()	Displays the width of the specified screen in pixels.
DisplayWidthMM() XDisplayWidthMM()	Displays the width of the specified screen in millimeters.
DoesBackingStore() XDoesBackingStore()	Indicates whether the specified screen supports backing stores.
DoesSaveUnders() XDoesSaveUnders()	Indicates whether the specified screen supports save unders.
EventMaskOfScreen() XEventMaskOfScreen()	Returns the initial root event mask for a specified screen.
HeightMMOfScreen() XHeightMMOfScreen()	Returns the height of a specified screen in millimeters.
HeightOfScreen() XHeightOfScreen()	Returns the height of a specified screen in pixels.
ImageByteOrder() XImageByteOrder()	Specifies the required byte order for each scanline unit of an image.
IsCursorKey()	Returns TRUE if keysym is on cursor key.
IsFunctionKey()	Returns TRUE if keysym is on function keys.
IsKeypadKey()	Returns TRUE if keysym is on keypad.

X Window System Interface

Table 59. Display Functions (continued)

Subroutine	Description
IsMiscFunctionKey()	Returns TRUE if keysym is on miscellaneous function keys.
IsModifierKey()	Returns TRUE if keysym is on modifier keys.
IsPFKey()	Returns TRUE if keysym is on PF keys.
LastKnownRequestProcessed() XLastKnownRequestProcessed()	Extracts the full serial number of the last known request processed by the X server.
MaxCmapsOfScreen() XMaxCmapsOfScreen()	Returns the maximum number of colormaps supported by the specified screen.
MinCmapsOfScreen() XMinCmapsOfScreen()	Returns the minimum number of colormaps supported by the specified screen.
NextRequest() XNextRequest()	Extracts the full serial number to be used for the next request to be processed by the X Server.
PlanesOfScreen() XPlanesOfScreen()	Returns the depth (number of planes) in a specified screen.
ProtocolRevision() XProtocolRevision()	Returns the minor protocol revision number (zero) of the X server associated with the display.
ProtocolVersion() XProtocolVersion()	Returns the major version number (11) of the protocol associated with the display.
QLength() XQLength()	Returns the length of the event queue for the display.
RootWindow() XRootWindow()	Returns the root window of the current screen.
RootWindowOfScreen() XRootWindowOfScreen()	Returns the root window of the specified screen.
ScreenCount() XScreenCount()	Returns the number of screens available.
XScreenNumberOfScreen()	Returns the screen index number of the specified screen.
ScreenOfDisplay() XScreenOfDisplay()	Returns the pointer to the screen of the display specified.
ServerVendor() XServerVendor()	Returns the pointer to a null-determined string that identifies the owner of the X server implementation.
VendorRelease() XVendorRelease()	Returns the number related to the vendor's release of the X server.
WhitePixel() XWhitePixel()	Returns the white pixel value for the current screen.
WhitePixelOfScreen() XWhitePixelOfScreen()	Returns the white pixel value of the specified screen.
WidthMMOfScreen() XWidthMMOfScreen()	Returns the width of the specified screen in millimeters.
WidthOfScreen() XWidthOfScreen()	Returns the width of the specified screen in pixels.

Extension Routines

Table 60 lists the X Window System Extension Subroutines.

Table 60. *Extension Routines*

Subroutine	Description
XAllocID()	Returns a resource ID that can be used when creating new resources.
XESetCloseDisplay()	Defines a procedure to call when XCloseDisplay is called.
XESetCopyGC()	Defines a procedure to call when a GC is copied.
XESetCreateFont()	Defines a procedure to call when XLoadQueryFont is called.
XESetCreateGC()	Defines a procedure to call when a new GC is created.
XESetError()	Suppresses the call to an external error handling routine and defines an alternative routine for error handling.
XESetErrorString()	Defines a procedure to call when an I/O error is detected.
XESetEventToWire()	Defines a procedure to call when an event must be converted from the host to wire format.
XESetFreeFont()	Defines a procedure to call when XFreeFont is called.
XESetFreeGC()	Defines a procedure to call when a GC is freed.
XESetWireToEvent()	Defines a procedure to call when an event is converted from the wire to the host format.
XFreeExtensionList()	Frees memory allocated by XListExtensions.
XListExtensions()	Returns a list of all extensions supported by the server.
XQueryExtension()	Indicates whether named extension is present.

MIT Extensions to X

Table 61 lists the routines that allow an application to use these extensions:

Table 61. *MIT Extensions to X*

Subroutine	Description
XShapeQueryExtension	Queries to see if the server supports the SHAPE extension.
XShapeQueryVersion	Checks the version number of the server SHAPE extension.
XShapeCombineRegion	Converts the specified region into a list of rectangles and calls XShapeRectangles.
XShapeCombineRectangles	Performs a CombineRectangles operation.

X Window System Interface

Table 61. MIT Extensions to X (continued)

Subroutine	Description
XShapeCombineMask	Performs a CombineMask operation.
XShapeCombineShape	Performs a CombineShape operation.
XShapeOffsetShape	Performs an OffsetShape operation.
XShapeQueryExtents	Sets the extents of the bounding and clip shapes.
XShapeSelectInput	Selects Input Events.
XShapeInputSelected	Returns the current input mask for extension events on the specified window.
XShapeGetRectangles	Gets a list of rectangles describing the region specified.
XMITMiscQueryExtension	Queries to see if the server supports the MITMISC extension.
XMITMiscSetBugMode	Sets the compatibility mode switch.
XMITMiscGetBugMode	Queries the compatibility mode switch.
XmbufQueryExtension	Queries to see if the server supports the MULTIBUF extension.
XmbufGetVersion	Gets the version number of the extension.
XmbufCreateBuffers	Requests that multiple buffers be created.
XmbufDestroyBuffers	Requests that the buffers be destroyed.
XmbufDisplayBuffers	Displays the indicated buffers.
XmbufGetWindowAttributes	Gets the multibuffering attributes.
XmbufChangeWindowAttributes	Sets the multibuffering attributes.
XmbufGetBufferAttributes	Gets the attributes for the indicated buffer.
XmbufChangeBufferAttributes	Sets the attributes for the indicated buffer.
XmbufGetScreenInfo	Gets the parameters controlling how mono and stereo windows may be created on the indicated screen.
XmbufCreateStereoWindow	Creates a stereo window.

Associate Table Functions

Table 62 lists the Associate Table functions.

Table 62. Associate Table Functions

Subroutine	Description
XCreateAssocTable()	Returns a pointer to the newly created associate table.
XDeleteAssoc()	Deletes an entry from the specified associate table.
XDestroyAssocTable()	Frees memory allocated to the specified associate table.
XLookUpAssoc()	Obtains data from the specified associate table.
XMakeAssoc()	Creates an entry in the specified associate table.

Miscellaneous Utility Routines

Table 63 lists the Miscellaneous Utility routines.

Table 63. Miscellaneous Utility Routines

Subroutine	Description
XctCreate()	Creates an XctData structure for parsing a Compound Text string.
XctFree()	Frees all data associated with the XctData structure.
XctNextItem()	Parses the next <i>item</i> from the Compound Text string.
XctReset()	Resets the XctData structure to reparse the Compound Text string.
XmuAddCloseDisplayHook()	Adds a callback for the given display.
XmuAddInitializer()	Registers a procedure, to be invoked the first time XmuCallInitializers is called on a given application context.
XmuAllStandardColormaps()	Creates all of the appropriate standard colormaps.
XmuCallInitializers()	Calls each of the procedures that have been registered with XmuAddInitializer.
XmuClientWindow()	Finds a window, at or below the specified window.
XmuCompareISOLatin1()	Compares two strings, ignoring case differences.
XmuConvertStandardSelection()	Converts many standard selections.
XmuCopyISOLatin1Lowered()	Copies a string, changing all Latin-1 uppercase letters to lowercase.
XmuCopyISOLatin1Uppered()	Copies a string, changing all Latin-1 lowercase letters to uppercase.
XmuCreateColormap()	Creates a colormap.
XmuCreatePixmapFromBitmap()	Creates a pixmap of the specified width, height, and depth.
XmuCreateStippledPixmap()	Creates a two pixel by one pixel stippled pixmap of specified depth on the specified screen.
XmuCursorNameToIndex()	Returns the index in the standard cursor font for the name of a standard cursor.
XmuCvtFunctionToCallback()	Converts a callback procedure to a callback list containing that procedure.
XmuCvtStringToBackingStore()	Converts a string to a backing-store integer.
XmuCvtStringToBitmap()	Creates a bitmap suitable for window manager icons.
XmuCvtStringToCursor()	Converts a string to a Cursor.
XmuCvtStringToJustify()	Converts a string to an XtJustify enumeration value.
XmuCvtStringToLong()	Converts a string to an integer of type long.

X Window System Interface

Table 63. Miscellaneous Utility Routines (continued)

Subroutine	Description
XmuCvtStringToOrientation()	Converts a string to an XtOrientation enumeration value.
XmuCvtStringToShapeStyle()	Converts a string to an integer shape style.
XmuCvtStringToWidget()	Converts a string to an immediate child widget of the parent widget passed as an argument.
XmuDeleteStandardColormap()	Removes the specified property from the specified screen.
XmuDQAddDisplay()	Adds the specified display to the queue.
XmuDQCreate()	Creates and returns an empty XmuDisplayQueue.
XmuDQDestroy()	Releases all memory associated with the specified queue.
XmuDQLookupDisplay()	Returns the queue entry for the specified display.
XmuDQNDisplays()	Returns the number of displays in the specified queue.
XmuDQRemoveDisplay()	Removes the specified display from the specified queue.
XmuDrawLogo()	Draws the <i>official</i> X Window System logo.
XmuDrawRoundedRectangle()	Draws a rounded rectangle.
XmuFillRoundedRectangle()	Draws a filled rounded rectangle.
XmuGetAtomName()	Returns the name of an Atom.
XmuGetColormapAllocation()	Determines the best allocation of reds, greens, and blues in a standard colormap.
XmuGetHostname()	Returns the host name.
XmuInternAtom()	Caches the Atom value for one or more displays.
XmuInternStrings()	Converts a list of atom names into Atom values.
XmuLocateBitmapFile()	Reads a file in standard bitmap file format.
XmuLookupAPL()	Maps a key event to an APL string. This function is similar to XLookupString.
XmuLookupArabic()	Maps a key event to a Latin/Arabic (ISO 8859-6) string. This function is similar to XLookupString.
XmuLookupCloseDisplayHook()	Determines if a callback is installed.
XmuLookupCyrillic()	Maps a key event to a Latin/Cyrillic (ISO 8859-5) string. This function is similar to XLookupString, except that it maps a
XmuLookupGreek()	Maps a key event to a Latin/Greek (ISO 8859-7) string. This function is similar to XLookupString.
XmuLookupHebrew()	Maps a key event to a Latin/Hebrew (ISO 8859-8) string. This function is similar to XLookupString.

Table 63. Miscellaneous Utility Routines (continued)

Subroutine	Description
XmuLookupJISX0201()	Maps a key event to a string in the JIS X0201-1976 encoding. This function is similar to XLookupString.
XmuLookupKana()	Maps a key event to a string in the JIS X0201-1976 encoding. This function is similar to XLookupString.
XmuLookupLatin1()	This function is identical to XLookupString.
XmuLookupLatin2()	Maps a key event to a Latin-2 (ISO 8859-2) string. This function is similar to XLookupString.
XmuLookupLatin3()	Maps a key event to a Latin-3 (ISO 8859-3) string. This function is similar to XLookupString.
XmuLookupLatin4()	Maps a key event to a Latin-4 (ISO 8859-4) string. This function is similar to XLookupString.
XmuLookupStandardColormap()	Creates or replaces a standard colormap if one does not currently exist.
XmuLookupString()	Maps a key event into a specific key symbol set.
XmuMakeAtom()	Creates and initializes an opaque object.
XmuNameOfAtom()	Returns the name of an AtomPtr.
XmuPrintDefaultErrorMessage()	Prints an error message, equivalent to Xlib's default error message.
XmuReadBitmapData()	Reads a standard bitmap file description.
XmuReadBitmapDataFromFile()	Reads a standard bitmap file description from the specified file.
XmuReleaseStippledPixmap()	Frees a pixmap created with XmuCreateStippledPixmap.
XmuRemoveCloseDisplayHook()	Deletes a callback that has been added with XmuAddCloseDisplayHook.
XmuReshapeWidget()	Reshapes the specified widget, using the Shape extension.
XmuScreenOfWindow()	Returns the screen on which the specified window was created.
XmuSimpleErrorHandler()	A simple error handler for Xlib error conditions.
XmuStandardColormap()	Creates a standard colormap for the given screen.
XmuUpdateMapHints()	Clears the PPosition and PSize flags and sets the USPosition and USSize flags.
XmuVisualStandardColormaps()	Creates all of the appropriate standard colormaps for a given visual.

X Authorization Routines

Table 64 lists the X Authorization routines.

Table 64. Authorization Data Routines

Subroutine	Description
XauFileName()	Generates the default authorization file name.
XauReadAuth()	Reads the next entry from the authfile.
XuWriteAuth()	Writes an authorization entry to the authfile.
XauGetAuthByAddr()	Searches for an authorization entry.
XauLockAuth()	Does the work necessary to synchronously update an authorization file.
XauUnlockAuth()	Undoes the work of XauLockAuth.
XauDisposeAuth()	Frees storage allocated to hold an authorization entry.

X Intrinsic Routines

Table 65 provides the X Intrinsic Routines.

Table 65. X Intrinsic Routines

Routine	Description
CompositeClassPartInitialize	Initializes the CompositeClassPart of a Composite Widget.
CompositeDeleteChild	Deletes a child widget from a Composite Widget.
CompositeDestroy	Destroys a composite widget.
CompositeInitialize	Initializes a Composite Widget structure.
CompositeInsertChild	Inserts a child widget in a Composite Widget.
RemoveCallback	Removes a callback procedure from a callback list.
XrmCompileResourceList	Compiles an XtResourceList into an XrmResourceList.
XtAddActions	Declares an action table and registers it with the translation manager
XtAddCallback	Adds a callback procedure to the callback list of the specified widget.
XtAddCallbacks	Adds a list of callback procedures to the callback list of specified widget.
XtAddConverter	Adds a new converter.
XtAddEventHandler	Registers an event handler procedure with the dispatch mechanism when an event matching the mask occurs on the specified widget.
XtAddExposureToRegion	Computes the union of the rectangle defined by the specified exposure event and region.
XtAddGrab	Redirects user input to a model widget.

Table 65. X Intrinsic Routines (continued)

Routine	Description
XtAddInput	Registers a new source of events.
XtAddRawEventHandler	Registers an event handler procedure with the dispatch mechanism without causing the server to select for that event.
XtAddTimeout	Creates a timeout value in the default application context and returns an identifier for it.
XtAddWorkProc	Registers a work procedure in the default application context.
XtAppAddActionHook	Adds an actionhook procedure to an application context.
XtAppAddActions	Declares an action table and registers it with the translation manager.
XtAppAddConverter	Registers a new converter.
XtAppAddInput	Registers a new file as an input source for a specified application.
XtAppAddTimeout	Creates a timeout value and returns an identifier for it.
XtAppAddWorkProc	Registers a work procedure for a specified procedure.
XtAppCreateShell	Creates a top-level widget that is the root of a widget tree.
XtAppError	Calls the installed fatal error procedure.
XtAppErrorMsg	Calls the high-level error handler.
XtAppGetErrorDatabase	Obtains the error database and merges it with an application or widget-specified database.
XtAppGetErrorDatabaseText	Obtains the error database text for an error or warning for an error message handler.
XtAppGetSelectionTimeout	Gets and returns the current selection timeout (ms) value.
XtAppInitialize	A convenience routine for initializing the toolkit.
XtAppMainLoop	Processes input by calling XtAppNextEvent and XtDispatchEvent.
XtAppNextEvent	Returns the value from the top of a specified application input queue.
XtAppPeekEvent	Returns the value from the top of a specified application input queue without removing input from queue.
XtAppPending	Determines if the input queue has any events for a specified application.
XtAppProcessEvent	Processes applications that require direct control of the processing for different types of input.
XtAppReleaseCacheRefs	Decrements the reference count for the conversion entries identified by the refs argument.

X Window System Interface

Table 65. X Intrinsic Routines (continued)

Routine	Description
XtAppSetErrorHandler	Registers a procedure to call on fatal error conditions. The default error handler prints the message to standard error.
XtAppSetErrorMsgHandler	Registers a procedure to call on fatal error conditions. The default error handler constructs a string from the error resource database.
XtAppSetFallbackResources	Sets the fallback resource list that will be loaded at display initialization time.
XtAppSetSelectionTimeout	Sets the Intrinsic selection time-out value.
XtAppSetTypeConverter	Registers the specified type converter and destructor in all application contexts created by the calling process.
XtAppSetWarningHandler	Registers a procedure to call on nonfatal error conditions. The default warning handler prints the message to standard error.
XtAppSetWarningMsgHandler	Registers a procedure to call on nonfatal error conditions. The default warning handler constructs a string from error resource database.
XtAppWarning	Calls the installed nonfatal error procedure.
XtAppWarningMsg	Calls the installed high-level warning handler.
XtAugmentTranslations	Merges new translations into an existing widget translation table.
XtBuildEventMask	Retrieves the event mask for a specified widget.
XtCallAcceptFocus	Calls the <code>accept_focus</code> procedure for the specified widget.
XtCallActionProc	Searches for the named action routine and, if found, calls it.
XtCallbackExclusive	Calls customized code for callbacks to create pop-up shell.
XtCallbackNone	Calls customized code for callbacks to create pop-up shell.
XtCallbackNonexclusive	Calls customized code for callbacks to create pop-up shell.
XtCallbackPopdown	Pops down a shell that was mapped by callback functions.
XtCallbackReleaseCacheRef	A callback that can be added to a callback list to release a previously returned <code>XtCacheRef</code> value.
XtCallbackReleaseCacheRefList	A callback that can be added to a callback list to release a list of previously returned <code>XtCacheRef</code> value.
XtCallCallbackList	Calls all callbacks on a callback list.
XtCallCallbacks	Executes the callback procedures in a widget callback list.

Table 65. X Intrinsic Routines (continued)

Routine	Description
XtCallConverter	Looks up the specified type converter in the application context and invokes the conversion routine.
XtCalloc	Allocates and initializes an array.
XtClass	Obtains the class of a widget and returns a pointer to the widget class structure.
XtCloseDisplay	Closes a display and removes it from an application context.
XtConfigureWidget	Moves and resizes the sibling widget of the child making the geometry request.
XtConvert	Invokes resource conversions.
XtConvertAndStore	Looks up the type converter registered to convert from_type to to_type and then calls XtCallConverter.
XtConvertCase	Determines upper and lowercase equivalents for a KeySym.
XtCopyAncestorSensitive	Copies the sensitive value from a widget record.
XtCopyDefaultColormap	Copies the default colormap from a widget record.
XtCopyDefaultDepth	Copies the default depth from a widget record.
XtCopyFromParent	Copies the parent from a widget record.
XtCopyScreen	Copies the screen from a widget record.
XtCreateApplicationContext	Creates an opaque type application context.
XtCreateApplicationShell	Creates an application shell widget by calling XtAppCreateShell.
XtCreateManagedWidget	Creates and manages a child widget in a single procedure.
XtCreatePopupShell	Creates a pop-up shell.
XtCreateWidget	Creates an instance of a widget.
XtCreateWindow	Calls XcreateWindow with the widget structure and parameter.
XtDatabase	Obtains the resource database for a particular display.
XtDestroyApplicationContext	Destroys an application context.
XtDestroyGC	Deallocates graphics context when it is no longer needed.
XtDestroyWidget	Destroys a widget instance.
XtDirectConvert	Invokes resource conversion.
XtDisownSelection	Informs the Intrinsic selection mechanism that the specified widget is to lose ownership of the selection.
XtDispatchEvent	Receives X events and calls appropriate event handlers.

X Window System Interface

Table 65. X Intrinsic Routines (continued)

Routine	Description
XtDisplay	Returns the display pointer for the specified widget.
XtDisplayInitialize	Initializes a display and adds it to an application context.
XtDisplayOfObject	Returns the display pointer for the specified widget.
XtDisplayStringConversionWarning	Issues a warning message for conversion routines.
XtDisplayToApplicationContext	Retrieves the application context associated with a Display.
XtError	Calls the installed fatal error procedure.
XtErrorMsg	A low-level error and warning handler procedure type.
XtFindFile	Searches for a file using substitutions in a path list.
XtFree	Frees an allocated block of storage.
XtGetActionKeysym	Retrieves the KeySym and modifiers that matched the final event specification in a translation table entry.
XtGetApplicationNameAndClass	Returns the application name and class as passed to XtDisplayInitialize
XtGetApplicationResources	Retrieves resources that are not specific to a widget, but apply to the overall application.
XtGetConstraintResourceList	Returns the constraint resource list for a particular widget.
XtGetErrorDatabase	Obtains the error database and returns the address of the error database.
XtGetErrorDatabaseText	Obtains the error database text for an error or warning.
XtGetGC	Returns a read-only sharable GC.
XtGetKeysymTable	Returns a pointer to the KeySym to KeyCode mapping table for a particular display.
XtGetMultiClickTime	Returns the multi-click time setting.
XtGetResourceList	Obtains the resource list structure for a particular class.
XtGetSelectionRequest	Retrieves the SelectionRequest event which triggered the convert_selection procedure.
XtGetSelectionTimeout	Obtains the current selection timeout.
XtGetSelectionValue	Obtains the selection value in a single, logical unit.
XtGetSelectionValueIncremental	Obtains the selection value using incremental transfers.
XtGetSelectionValues	Takes a list of target types and client data and obtains the current value of the selection converted to each of the targets.

Table 65. X Intrinsic Routines (continued)

Routine	Description
XtGetSelectionValuesIncremental	A function similar to XtGetSelectionValueIncremental except that it takes a list of targets and client_data.
XtGetSubresources	Obtains resources other than widgets.
XtGetSubvalues	Retrieves the current value of a non-widget resource data associated with a widget instance.
XtGetValues	Retrieves the current value of a resource associated with a widget instance.
XtGrabButton	Passively grabs a single pointer button.
XtGrabKey	Passively grabs a single key of the keyboard.
XtGrabKeyboard	Actively grabs the keyboard.
XtGrabPointer	Actively grabs the pointer.
XtHasCallbacks	Finds the status of a specified widget callback list.
XtInitialize	Initializes the toolkit, application, and shell.
XtInitializeWidgetClass	Initializes a widget class without creating any widgets.
XtInsertEventHandler	Registers an event handler procedure that receives events before or after all previously registered event handler.
XtInsertRawEventHandler	Registers an event handler procedure that receives events before or after all previously registered event handler without selecting for the events.
XtInstallAccelerators	Installs accelerators from a source widget to destination widget.
XtInstallAllAccelerators	Installs all the accelerators from a widget and all the descendants of the widget onto one destination widget.
XtIsApplicationShell	Determines whether a specified widget is a subclass of an Application Shell widget.
XtIsComposite	Determines whether a specified widget is a subclass of a Composite widget.
XtIsConstraint	Determines whether a specified widget is a subclass of a Constraint widget.
XtIsManaged	Determines the managed state of a specified child widget.
XtIsObject	Determines whether a specified widget is a subclass of an Object widget.
XtIsOverrideShell	Determines whether a specified widget is a subclass of an Override Shell widget.
XtIsRealized	Determines if a widget has been realized.
XtIsRectObj	Determines whether a specified widget is a subclass of a RectObj widget.
XtIsSensitive	Determines the current sensitivity state of a widget.

X Window System Interface

Table 65. X Intrinsic Routines (continued)

Routine	Description
XtIsShell	Determines whether a specified widget is a subclass of a Shell widget.
XtIsSubclass	Determines whether a specified widget is in a specific subclass.
XtIsTopLevelShell	Determines whether a specified widget is a subclass of a TopLevelShell widget.
XtIsTransientShell	Determines whether a specified widget is a subclass of a TransientShell widget.
XtIsVendorShell	Determines whether a specified widget is a subclass of a VendorShell widget.
XtIsWidget	Determines whether a specified widget is a subclass of a Widget widget.
XtIsWMSHELL	Determines whether a specified widget is a subclass of a WMSHELL widget.
XtKeysymToKeyCodeList	Returns the list of KeyCodes that map to a particular KeySym.
XtLastTimestampProcessed	Retrieves the timestamp from the most recent call to XtDispatchEvent.
XtMainLoop	An infinite loop which processes input.
XtMakeGeometryRequest	A request from the child widget to a parent widget for a geometry change.
XtMakeResizeRequest	Makes a resize request from a widget.
XtMalloc	Allocates storage.
XtManageChild	Adds a single child to a parent widget list of managed children.
XtManageChildren	Adds a list of widgets to the geometry-managed, displayable, subset of its composite parent widget.
XtMapWidget	Maps a widget explicitly.
XtMenuPopupAction	Pops up a menu when a pointer button is pressed or when the pointer is moved into the widget.
XtMergeArgLists	Merges two ArgList structures.
XtMoveWidget	Moves a sibling widget of the child making the geometry request.
XtName	Returns a pointer to the instance name of the specified object.
XtNameToWidget	Translates a widget name to a widget instance.
XtNewString	Copies an instance of a string.
XtNextEvent	Returns the value from the header of the input queue.
XtOpenDisplay	Opens, initializes, and adds a display to an application context.
XtOverrideTranslations	Overwrites existing translations with new translations.

Table 65. X Intrinsic Routines (continued)

Routine	Description
XtOwnSelection	Sets the selection owner when using atomic transfer.
XtOwnSelectionIncremental	Sets the selection owner when using incremental transfers.
XtParent	Returns the parent widget for the specified widget.
XtParseAcceleratorTable	Parses an accelerator table into the opaque internal representation.
XtParseTranslationTable	Compiles a translation table into the opaque internal representation of type XtTranslations.
XtPeekEvent	Returns the value from the front of the input queue without removing it from the queue.
XtPending	Determines if the input queue has events pending.
XtPopdown	Unmaps a pop-up from within an application.
XtPopup	Maps a pop-up from within an application.
XtPopupSpringLoaded	Maps a spring-loaded pop-up from within an application.
XtProcessEvent	Processes one input event, timeout, or alternate input source.
XtQueryGeometry	Queries the preferred geometry of a child widget.
XtRealizeWidget	Realizes a widget instances.
XtRealloc	Changes the size of an allocated block of storage, sometimes moving it.
XtRegisterCaseConverter	Registers a specified case converter.
XtRegisterGrabAction	Registers button and key grabs for a widget's window according to the event bindings in the widget's translation table.
XtReleaseGC	Deallocates a shared GC when it is no longer needed.
XtRemoveActionHook	Removes an action hook procedure without destroying the application context.
XtRemoveAllCallbacks	Deletes all callback procedures from a specified widget callback list.
XtRemoveCallback	Deletes a callback procedure from a specified widget callback list only if both the procedure and the client data match.
XtRemoveCallbacks	Deletes a list of callback procedures from a specified widget callback list.
XtRemoveEventHandler	Removes a previously registered event handler.
XtRemoveGrab	Removes the redirection of user input to a modal widget.

X Window System Interface

Table 65. X Intrinsic Routines (continued)

Routine	Description
XtRemoveInput	Discontinues a source of input by causing the Intrinsic read routine to stop watching for input from the input source.
XtRemoveRawEventHandler	Removes previously registered raw event handler.
XtRemoveTimeout	Clears a timeout value by removing the timeout.
XtRemoveWorkProc	Removes the specified background work procedure.
XtResizeWidget	Resizes a sibling widget of the child making the geometry request.
XtResizeWindow	Resizes a child widget that already has the values for its width, height, and border width.
XtResolvePathname	Searches for a file using standard substitutions in a path list.
XtScreen	Returns the screen pointer for the specified widget.
XtScreenOfObject	Returns the screen pointer for the nearest ancestor of object that is of class Widget.
XtSetErrorHandler	Registers a procedure to call under fatal error conditions.
XtSetErrorMsgHandler	Registers a procedure to call under fatal error conditions.
XtSetKeyboardFocus	Redirects keyboard input to a child of a composite widget without calling XSetInputFocus.
XtSetKeyTranslator	Registers a key translator.
XtSetMappedWhenManaged	Changes the widget map_when_managed field.
XtSetMultiClickTime	Sets the multi-click time for an application.
XtSetSelectionTimeout	Sets the Intrinsic selection timeout.
XtSetSensitive	Sets the sensitivity state of a widget.
XtSetSubvalues	Sets the current value of a non-widget resource associated with an instance.
XtSetTypeConverter	Registers a type converter for all application contexts in a process.
XtSetValues	Modifies the current value of a resource associated with widget instance.
XtSetWarningHandler	Registers a procedure to be called on non-fatal error conditions.
XtSetWarningMsgHandler	Registers a procedure to be called on non-fatal error conditions.
XtSetWMColormapWindows	Sets the value of the WM_COLORMAP_WINDOWS property on a widget's window.

Table 65. X Intrinsic Routines (continued)

Routine	Description
XtStringConversionWarning	A convenience routine for old-format resource converters that convert from strings.
XtSuperclass	Obtains the superclass of a widget by returning a pointer to the superclass structure of the widget.
XtToolkitInitialize	Initializes the X Toolkit internals.
XtTranslateCoords	Translates an [x,y] coordinate pair from widget coordinates to root coordinates.
XtTranslateKey	The default key translator routine.
XtTranslateKeycode	Registers a key translator.
XtUngrabButton	Cancels a passive button grab.
XtUngrabKey	Cancels a passive key grab.
XtUngrabKeyboard	Cancels an active keyboard grab.
XtUngrabPointer	Cancels an active pointer grab.
XtUninstallTranslations	Causes the entire translation table for widget to be removed.
XtUnmanageChild	Removes a single child from the managed set of its parent.
XtUnmanageChildren	Removes a list of children from the managed list of the parent, but does not destroy the children widgets.
XtUnmapWidget	Unmaps a widget explicitly.
XtUnrealizeWidget	Destroys the associated with a widget and its descendants.
XtVaAppCreateShell	Creates a top-level widget that is the root of a widget tree using varargs lists.
XtVaAppInitialize	Initializes the Xtk internals, creates an application context, opens and initializes a display and creates the initial application shell instance using varargs lists.
XtVaCreateArgsList	Dynamically allocates a varargs list for use with XtVaNestedList in multiple calls.
XtVaCreateManagedWidget	Creates and manages a child widget in a single procedure using varargs lists.
XtVaCreatePopupShell	Creates a pop-up shell using varargs lists.
XtVaCreateWidget	Creates an instance of a widget using varargs lists.
XtVaGetApplicationResources	Retrieves resources for the overall application using varargs list.
XtVaGetSubresources	Fetches resources for widget sub-parts using varargs list.
XtVaGetSubvalues	Retrieves the current values of non-widget resources associated with a widget instance using varargs lists.
XtVaGetValues	Retrieves the current values of resources associated with a widget instance using varargs lists.

X Window System Interface

Table 65. X Intrinsic Routines (continued)

Routine	Description
XtVaSetSubvalues	Sets the current values of non-widget resources associated with a widget instance using varargs lists.
XtVaSetValues	Modifies the current values of resources associated with a widget instance using varargs lists.
XtWarning	Calls the installed non-fatal error procedure.
XtWarningMsg	Calls the installed high-level warning handler.
XtWidgetToApplicationContext	Gets the application context for given widget.
XtWindow	Returns the window of the specified widget.
XtWindowOfObject	Returns the window for the nearest ancestor of object that is of class Widget.
XtWindowToWidget	Translates a window and display pointer into a widget instance.

Athena Widget Support

Table 66 provides the Athena widget routines.

Table 66. Athena Widget Routines

Routine	Description
XawAsciiSave	Saves the changes made in the current text source into a file.
XawAsciiSaveAsFile	Saves the contents of the current text buffer into a named file.
XawAsciiSourceChanged	Determines if the text buffer in an AsciiSrc object has changed.
XawAsciiSourceFreeString	Frees the storage associated with the string from an AsciiSrc widget requested with a call to XtGetValues.
XawDialogAddButton	Adds a new button to a Dialog widget.
XawDialogGetValueString	Returns the character string in the text field of a Dialog Widget.
XawDiskSourceCreate	Creates a disk source.
XawFormDoLayout	Forces or defers a re-layout of the form.
XawInitializeWidgetSet	Forces a reference to vendor shell so that the one in this widget is installed.
XawListChange	Changes the list that is displayed.
XawListHighlight	Highlights an item in the list.
XawListShowCurrent	Retrieves the list element that is currently set.
XawListUnhighlight	Unhighlights an item in the list.
XawPanedAllowResize	Enables or disables a child's request for pane resizing.
XawPanedGetMinMax	Retrieves the minimum and maximum height settings for a pane.

Table 66. Athena Widget Routines (continued)

Routine	Description
XawPanedGetNumSub	Retrieves the number of panes in a paned widget.
XawPanedSetMinMax	Sets the minimum and maximum height settings for a pane.
XawPanedSetRefigureMode	Enables or disables automatic recalculation of pane sizes and positions
XawScrollbarSetThumb	Sets the position and length of a Scrollbar thumb.
XawSimpleMenuAddGlobalActions	Registers an XawPositionSimpleMenu global action routine.
XawSimpleMenuClearActiveEntry	Clears the SimpleMenu widget's internal information about the currently highlighted menu entry.
XawSimpleMenuGetActiveEntry	Gets the currently highlighted menu entry.
XawStringSourceCreate	Creates a string source.
XawTextDisableRedisplay	Disables redisplay while making several changes to a Text Widget.
XawTextDisplay	Displays batched updates.
XawTextDisplayCaret	Enables and disables the insert point.
XawTextEnableRedisplay	Enables redisplay.
XawTextGetInsertionPoint	Returns the current position of the insert point.
XawTextGetSelectionPos	Retrieves the text that has been selected by this text widget.
XawTextGetSource	Retrieves the current text source for the specified widget.
XawTextInvalidate	Redisplays a range of characters.
XawTextReplace	Modifies the text in an editable Text widget.
XawTextSearch	Searches for a string in a Text widget.
XawTextSetInsertionPoint	Moves the insert point to the specified source position.
XawTextSetLastPos	Sets the last position data in an AsciiSource Object.
XawTextSetSelection	Selects a piece of text.
XawTextSetSelectionArray	Assigns a new selection array to a text widget.
XawTextSetSource	Replaces the text source in the specified widget.
XawTextSinkClearToBackground	Clears a region of the sink to the background color
XawTextSinkDisplayText	Stub function that in subclasses will display text.
XawTextSinkFindDistance	Finds the Pixel Distance between two text Positions.
XawTextSinkFindPosition	Finds a position in the text.

X Window System Interface

Table 66. Athena Widget Routines (continued)

Routine	Description
XawTextSinkGetCursorBounds	Finds the bounding box for the insert cursor.
XawTextSinkInsertCursor	Places the InsertCursor.
XawTextSinkMaxHeight	Finds the Minimum height that will contain a given number of lines.
XawTextSinkMaxLines	Finds the Maximum number of lines that will fit in a given height.
XawTextSinkResolve	Resolves a location to a position.
XawTextSinkSetTabs	Sets the Tab stops.
XawTextSourceConvertSelection	Dummy selection converter.
XawTextSourceRead	Reads the source into a buffer.
XawTextSourceReplace	Replaces a block of text with new text.
XawTextSourceScan	Scans the text source for the number and type of item specified.
XawTextSourceSearch	Searches the text source for the text block passed.
XawTextSourceSetSelection	Allows special setting of the selection.
XawTextTopPosition	Returns the character position of the left-most character on the first line displayed in the widget.
XawTextUnsetSelection	Unhighlights previously highlighted text in a widget.
XawToggleChangeRadioGroup	Allows a toggle widget to change radio groups.
XawToggleGetCurrent	Returns the RadioData associated with the toggle widget that is currently active in a toggle group.
XawToggleSetCurrent	Sets the Toggle widget associated with the radio_data specified.
XawToggleUnsetCurrent	Unsets all Toggles in the radio_group specified.

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 60 on page 275.

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 the TCP/IP Level 3A0:

- SHAPE
- MITMISC
- MULTIBUF

See Table 61 on page 275

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 62 on page 276.

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 63 on page 277.

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 64 on page 280.

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

X Window System Interface

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 Intrinsic, 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
- Pixmap
- Errors and warnings.

In the VM/CMS environment, you must remap many of the X Widget and X Intrinsic 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 67 on page 295.

Table 67. X Intrinsic Header File Names

MIT Distribution Name	TCP/IP Name
CompositeI.h	ComposiI.h
CompositeP.h	ComposiP.h
ConstrainP.h	ConstraP.h
IntrinsicI.h	IntriniI.h
IntrinsicP.h	IntriniP.h
PassivGraI.h	PassivGr.h
ProtocolsP.h	ProtocoP.h
SelectionI.h	SelectiI.h
WindowObjP.h	WindowOP.h

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.filetype
```

- 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 *xnllanguage*XAPDF, 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	Paned
AsciiSrc	Scrollbar
AsciiText	Simple
Box	SimpleMenu
Clock	Sme (Simple Menu Entry)
Command	SmeBSB (BSB Menu Entry)
Dialog	Smeline
Form	StripChart
Grip	Text
Label	TextSink
List	TextSrc
Logo	Toggle
Mailbox	VPaned
MenuButton	Viewport

For a complete list of the widgets supported by the Athena widget set, see “Athena Widget Support” on page 290.

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 68.

Table 68. Athena Header File Names

MIT Distribution Name	TCP/IP Name
AsciiSinkP.h	AscSinkP.h
AsciiSrcP.h	AscSrcP.h
AsciiTextP.h	AscTextP.h
Command.h	ACommand.h
CommandP.h	ACommanP.h
Form.h	AForm.h
FormP.h	AFormP.h
Label.h	ALabel.h
LabelP.h	ALabelP.h
List.h	AList.h
ListP.h	AListP.h
MenuButtoP.h	MenuButP.h
Scrollbar.h	AScrollb.h
ScrollbarP.h	AScrollP.h
SimpleMenP.h	SimpleMP.h
StripCharP.h	StripChP.h
TemplateP.h	TemplatP.h
Text.h	AText.h
TextSinkP.h	TextSinP.h
TextP.h	ATextP.h
TextSrcP.h	ATextSrP.h
ViewportP.h	ViewporP.h

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:

ArrowButton	MenuShell
ArrowGadget	MessageBox
ArrowButtonGadget	PanedWindow
BulletinBoard	PushButton
CascadeButton	PushButtonGadget
CascadeButtonGadget	RowColumn
Command	Sash
DialogShell	Scale
DrawingArea	ScrollBar
DrawnButton	ScrolledWindow
FileSelectionBox	SelectionBox
FileSelectionDialog	SelectionDialog
Form	Separator
Frame	SeparatorGadget
Label	Text
LabelGadget	ToggleButton
List	ToggleButtonGadget
MainWindow	

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 67 on page 295. In porting applications to the CMS environment, the header file name changes shown in Table 69 on page 299 may have to be made.

Table 69. OSF/Motif Header File Names

OSF/Motif Distribution Name	TCP/IP Name
BulletinBP.h	BulletBP.h
CascadeBG.h	CascadBG.h
CascadeBGP.h	CascaBGP.h
CascadeBP.h	CascadBP.h
CutPasteP.h	CutPastP.h
MenuShellP.h	MenuSheP.h
MessageBP.h	MessagBP.h
RowColumnP.h	RowColuP.h
ScrollBarP.h	ScrollBP.h
ScrolledWP.h	ScrollWP.h
SelectioB.h	SelectiB.h
SelectioBP.h	SelectBP.h
SeparatoG.h	SeparatG.h
SeparatoGP.h	SeparaGP.h
SeparatorP.h	SeparatP.h
ToggleBGP.h	ToggleBGP.h
ToggleBP.h	ToggleBP.h

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 300)
- A simple program that uses the OSF/Motif-based widget set (see page “OSF/Motif-Based Widget Sample Program” on page 302).

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.

```

/*
 * 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 <Xlib.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...\n");
    system("CP SLEEP 60 SEC");
    fprintf(stderr, "Okay, back!\n");

/*
 * 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
```

Athena Widget Sample Program

```
* 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.
     */
}
```

Athena Widget Sample Program

```
    */
    XtAppMainLoop(app_con);
}
```

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.

```
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <Xm/Shell.h>
#include <Xm/PushButton.h>
static void CloseApp();
Widget Shell;
Widget Button;
void main(argc, argv)
int argc;
char *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)
Widget w;
caddr_t client_data;
caddr_t call_data;
{
    exit(0);
}
```

Chapter 8. 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:

Parameter	Description
<i>principal name</i>	Specifies the unique name of a user (client) or service.
<i>instance</i>	Specifies a label that is used to distinguish among variations of the <i>principal name</i> . An <i>instance</i> 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.

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:

NAME	DESCRIPTION
billb	Principal name
jis.admin	Principal name and instance
srz@inorg.chem.edu	Principal name, null instance, realm
trees.root@org.chem.edu	Principal name, instance, realm

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 31 summarizes the ticket-granting process for accessing a service.

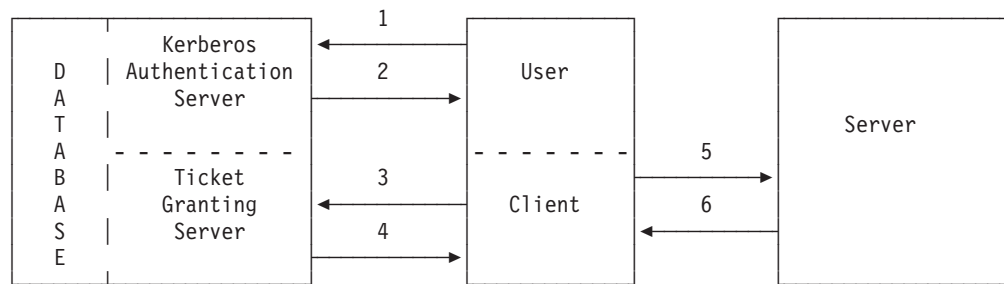


Figure 31. Protocol for Accessing a Service

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).

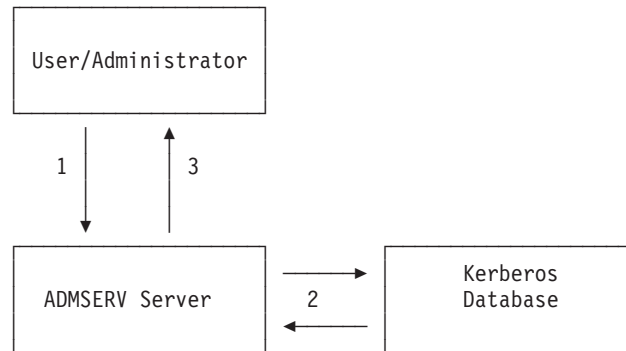
Kerberos Database

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 32 on page 307 summarizes the steps involving the Kerberos database and the *ADMSERV*.



1. The user or administrator makes a request to the ADMSERV server, using the KPASSWD user command or the KADMIN utility program.
2. The following two steps occur:
 - a. The ADMSERV verifies the identity of a requestor.
 - b. ADMSERV updates or retrieves the database entry.
3. The ADMSERV server informs the client of the result of the operation.

Notes:

1. The client does not need an initial ticket for this operation.
2. The ADMSERV has read/write access to the database, while the VMKORB server has read only access to the same database.

Figure 32. Protocol for Changing the Kerberos Database

Kerberos C Language Applications Library

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.

Kerberos Routines Reference

This section provides a reference for Kerberos routines. Table 70 provides the names, descriptions, and page numbers of the routines, located in the KRB TXTLIB, which are needed to interface with Kerberos.

Table 70. Kerberos krb_ Routines Reference

Kerberos krb_ Routine	Description	page
krb_get_cred()	Searches the caller's ticket file for a ticket containing the specified <i>principal name</i> , <i>instance</i> , and <i>realm</i> .	309
krb_kntoln()	Converts a Kerberos name to a local name.	309
krb_mk_err()	Constructs an application level error message that can be used in conjunction with the <code>krb_mk_priv()</code> and <code>krb_mk_safe()</code> routines.	310
krb_mk_priv()	Creates an encrypted, authenticated message from any arbitrary application data pointed to by <i>in</i> .	310
krb_mk_req()	Takes a pointer to a text structure in which an authenticator is to be built. It also takes the <i>principal name</i> , <i>instance</i> , and <i>realm</i> of the service to be used and an optional checksum.	311
krb_mk_safe()	Creates an authenticated, but unencrypted message from any arbitrary application data pointed to by <i>in</i> .	311
krb_rd_err()	Unpacks a message received from <code>krb_mk_err()</code> .	312
krb_rd_priv()	Decrypts and authenticates a message received from <code>krb_mk_priv()</code> .	313
krb_rd_req()	Finds out information about the <i>principal name</i> when a request has been made to a service.	314
krb_rd_safe()	Authenticates a message received from <code>krb_mk_safe()</code> .	315
krb_recvauth()	Called by the server to verify an authentication message received from a client.	315
krb_sendauth()	Prepares and transmits a ticket over a file descriptor.	316

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 318.

Kerberos Routines

This section provides the syntax, parameters, and other appropriate information for routines, located in the KRB TXTLIB, which are needed to interface with Kerberos.

krb_get_cred()

```
int krb_get_cred(service, instance, realm, c)
char *service;
char *instance;
char *realm;
CREDENTIALS *c;
```

Parameter	Description
<i>service</i>	Specifies the first part of the Kerberos name (<i>principal name</i>) of the target service.
<i>instance</i>	Specifies the second part of the Kerberos name of the target service.
<i>realm</i>	Specifies the third part of the Kerberos name of the target service.
<i>c</i>	Points to the structure, which is filled with credentials information.

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()

```
int krb_kntoln(ad, lname)
AUTH_DAT *ad;
char *lname;
```

krb_kntoln()

Parameter	Description
<i>ad</i>	Specifies an authentication structure containing a Kerberos name.
<i>lname</i>	Specifies a local name.

Description: The `krb_kntoln()` routine takes a Kerberos name in an `AUTH_DAT` 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()

```
long krb_mk_err(out, code, string)
unsigned char *out;
long code;
char *string;
```

Parameter	Description
<i>out</i>	Points to the output buffer area.
<i>code</i>	Specifies an application-specific error code.
<i>string</i>	Specifies an application-specific error string.

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_err()` 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()

```
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;
```

Parameter	Description
<i>in</i>	Points to an input structure containing application data.
<i>out</i>	Points to the output structure containing the encrypted data.
<i>in_length</i>	Specifies the length of the application data pointed to by <i>in</i> .
<i>schedule</i>	Specifies the session key schedule.
<i>key</i>	Points to a private session key.
<i>sender</i>	Specifies the fully qualified domain name of the sender.
<i>receiver</i>	Specifies the fully qualified domain name of the receiver.

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()

```
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;
```

Parameter	Description
<i>authent</i>	Points to the text structure in which an authenticator (including a service ticket) is to be built.
<i>service</i>	Specifies the first part of the Kerberos name (<i>principal name</i>) of the service.
<i>instance</i>	Specifies the second part of the Kerberos name of the service.
<i>realm</i>	Specifies the third part of the Kerberos name of the service.
<i>checksum</i>	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()

```
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;
```

Parameter	Description
<i>in</i>	Points to an input structure containing application data.
<i>out</i>	Points to the output structure containing the encrypted data.
<i>in_length</i>	Indicates the length of the application data pointed to by <i>in</i> .
<i>key</i>	Points to a private session key.
<i>sender</i>	Specifies the fully qualified domain name of the sender.
<i>receiver</i>	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()

```
int krb_rd_err(in, in_length, code, msg_data)
unsigned char *in;
unsigned long in_length;
long *code;
MSG_DAT *msg_data;
```

Parameter	Description
<i>in</i>	Points to the beginning of the received message.
<i>in_length</i>	Indicates the length of the received message pointed to by <i>in</i> .
<i>code</i>	Points to a value filled with the error value provided by the application.
<i>msg_data</i>	Points to the MSG_DAT structure, defined in KRB.H, which is filled by <code>krb_rd_err()</code> .

Description: The `krb_rd_err()` routine unpacks a message received from `krb_mk_err()`, and fills the following MSG_DAT fields:

Parameter	Description
<i>app_data</i>	Points to the application error text.
<i>app_length</i>	Indicates the <i>in_length</i> 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()

```
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;
```

Parameter	Description
<i>in</i>	Points to the beginning of the received message.
<i>in_length</i>	Indicates the length of the received message pointed to by <i>in</i> .
<i>schedule</i>	Specifies the session key schedule.
<i>key</i>	Points to a private session key.
<i>sender</i>	Specifies the fully qualified domain name of the sender to be checked against the message pointed to by <i>in</i> .
<i>receiver</i>	Specifies the fully qualified domain name of the receiver to be checked against the message pointed to by <i>in</i> .
<i>msg_data</i>	Points to the MSG_DAT structure, defined in KRB.H, which is filled by <code>krb_rd_priv()</code> .

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:

Parameter	Description
<i>app_data</i>	Points to the decrypted application data.
<i>app_length</i>	Indicates the length of the <i>app_data</i> field.
<i>time_sec</i>	Specifies the time stamps in the message.
<i>time_5ms</i>	Specifies the time stamps in the message.

`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_priv()

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_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;
```

Parameter	Description
<i>authent</i>	Specifies the authenticator of type KTEXT.
<i>service</i>	Specifies the first part of the Kerberos name (<i>principal name</i>).
<i>instance</i>	Specifies the second part of the Kerberos name.
<i>from_addr</i>	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.
<i>ad</i>	Points to the structure AUTH_DAT, which is filled with information obtained from the authenticator.
<i>filename</i>	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* parameters 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

- RD_AP_TIME
- RD_AP_NYV
- RD_AP_EXP

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_safe()

```
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;
```

Parameter	Description
<i>in</i>	Points to the beginning of the received message.
<i>in_length</i>	Indicates the length of the received message pointed to by <i>in</i> .
<i>key</i>	Points to a private session key.
<i>sender</i>	Specifies the fully qualified domain name of the sender to be checked against the message pointed to be <i>in</i> .
<i>receiver</i>	Specifies the fully qualified domain name of the receiver to be checked against the message pointed to be <i>in</i> .
<i>msg_data</i>	Points to the MSG_DAT structure, defined in KRB.H, which is filled by <code>krb_rd_safe()</code> .

Description: The `krb_rd_safe()` routine authenticates a message received from `krb_mk_safe()`, and, if successful, fills the following MSG_DAT fields:

Parameter	Description
<i>app_data</i>	Points to the decrypted application data.
<i>app_length</i>	Indicates the length of the <i>app_data</i> field.
<i>time_sec</i>	Specifies the time stamps in the message.
<i>time_5ms</i>	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 for the definition, current protocol version, and possible Kerberos message types.

krb_recvauth()

krb_recvauth()

```
int krb_recvauth(options, fd, ticket, service, instance, faddr, laddr, kdata, filename,  
schedule, version)  
long options;  
int fd;  
KTEXT ticket;  
char *service;  
char *instance;  
struct sockaddr_in *faddr;  
struct sockaddr_in *laddr;  
AUTH_DAT *kdata;  
char *filename;  
des_key_schedule schedule;  
char *version;
```

Parameter	Description
<i>options</i>	Indicates a bit-field of selected options. The only option valid for <code>krb_recvauth()</code> is <code>KOPT_DO_MUTUAL</code> .
<i>fd</i>	Specifies the socket descriptor from which to read the authentication message (and write to, if mutual authentication is requested).
<i>ticket</i>	Specifies a Kerberos ticket, which is part of the received message sent by the client.
<i>service</i>	Specifies the first part of the Kerberos name (<i>principal name</i>) of the target service.
<i>instance</i>	Specifies the second part of the Kerberos name of the target service.
<i>faddr</i>	Specifies the network address of the sending host (client).
<i>laddr</i>	Specifies the network address of the local server. Can be <code>NULL</code> , unless mutual authentication is requested.
<i>kdata</i>	Specifies the authentication information extracted from the message.
<i>filename</i>	Specifies the name of the file containing the server's keys. <i>filename</i> is passed to <code>krb_rd_req()</code> . If <i>filename</i> is <code>NULL</code> , the <code>ETC SRVTAB</code> file is used.
<i>schedule</i>	Specifies the session key schedule.
<i>version</i>	Specifies the version string, which should be large enough to hold a <code>KRB_SENDAUTH_VLEN</code> character string (defined in <code>KRB.H</code>).

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 318.

Return Values: The integer `KSUCCESS` indicates that the authentication was successful. `KFAILURE` indicates that the authentication has failed.

krb_sendauth()

krb_sendauth()

```
int krb_sendauth(options, fd, ticket, service, instance, realm, checksum, msg_data, cred,
schedule, laddr, faddr, version)
long options;
int fd;
KTEXT ticket;
char *service;
char *instance;
char *realm;
unsigned long checksum;
MSG_DAT *msg_data;
CREDENTIALS *cred;
struct sockaddr_in *faddr;
struct sockaddr_in *laddr;
des_key_schedule schedule;
char *version;
```

Parameter	Description
<i>options</i>	Indicates a bit-field of selected options.
<i>fd</i>	Specifies the socket descriptor to write to (and read from, if mutual authentication is requested).
<i>ticket</i>	Indicates the area where the ticket is returned when any option except KOPT_DONT_MK_REQ is requested. When KOPT_DONT_MK_REQ is requested, you must supply a value for <i>ticket</i> .
<i>service</i>	Specifies the first part of the Kerberos name (<i>principal name</i>) of the target service.
<i>instance</i>	Specifies the second part of the Kerberos name of the target service.
<i>realm</i>	Specifies the third part of the Kerberos name of the target service. If <i>realm</i> is NULL, the local <i>realm</i> is used.
<i>checksum</i>	Specifies any long integer supplied by the calling routine for mutual authentication purposes.
<i>msg_data</i>	Points to the MSG_DAT structure, defined in KRB.H, which is filled by krb_sendauth(), if the mutual authentication option is specified.
<i>cred</i>	Specifies the space you must allocate to hold the session key.
<i>schedule</i>	Specifies the space you must allocate to hold the session key schedule.
<i>laddr</i>	Specifies the network address of the client.
<i>faddr</i>	Specifies the network address of the server.
<i>version</i>	Specifies the version string, which should be large enough to hold a KRB_SENDAUTH_VLEN character string (as defined in KRB.H).

Description: 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.

```
/*
 * $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>\n", 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\n",
            SAMPLE_SERVICE);
        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\n", argv[1]);
        exit(1);
    }

    /* copy the hostname into dynamic storage */
    remote_host = malloc(strlen(hp->h_name) + 1);
    (void) strcpy(remote_host, hp->h_name);

    /* set up the address of the foreign socket for connect() */
    sin.sin_family = hp->h_addrtype;
    (void) memcpy((char *)sin_addr,
        (char *)hp->h_addr,

```

Kerberos Client

```
        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%s\n", buf);

close(sock);
exit(0);
}
```

Kerberos Server

The following is an example of a Kerberos server program.

```
/*
 * $Source: /mit/kerberos/src/appl/sample/RCS/sample_server.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_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 namelen = 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");
        exit(1);
    }
    if (bind(s, &myname, sizeof myname) < 0) {
        printf("sample_s: bind");
        exit(1);
    }
}

```

Kerberos Server

```
    }
    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, "getsocknamename: %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\n", krb_err_txt[status]);
        (void) sprintf(retbuf, "Kerberos error: %s\n",
            krb_err_txt[status]);
    } else {
        /* Check the version string (8 chars) */
        if (strncmp(version, "VERSION9", 8)) {
            /* didn't match the expected version */
            /* 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),\n at address %s,
            version %s,
            cksum %ld\n",
            auth_data.pname,
            auth_data.pinst,
            auth_data.prealm,
            lname,
            inet_ntoa(peername.sin_addr),
            version,
            auth_data.checksum);
    }
}
```

```
/* 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\n",
        count, len);
    exit(1);
}

/* close up and exit */
close(ns);
goto again;
exit(0);
}
```

Kerberos Server

Chapter 9. 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.

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 33.

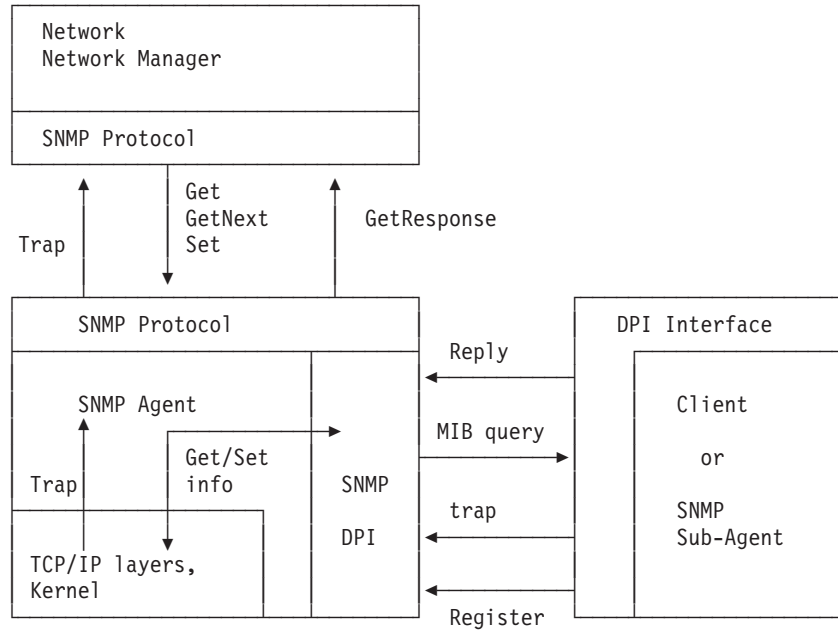


Figure 33. SNMP DPI overview

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).

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:

```
unsigned char *cp;

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

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

```
char *obj_id;

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

/* 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);
cp = mkDPIresponse(0,ret_value);
```

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

Subagents can support several different groups of the MIB tree. However, the subagent cannot jump from one group to another. You must first determine the reason for the request to then determine the path to traverse in the MIB tree. The second parameter contains this reason and is the group prefix of the MIB tree that is supported by the subagent.

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

SNMP Agent Distributed Program Interface

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:

```
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 parameters of the TRAP. The subagent can optionally provide a name, type, and value parameter. The DPI API library routine `mkDPItrap()` can be used to generate the TRAP packet.

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 “TCPLOAD EXEC” on page 2 and for the syntax of the SET LDRTBLS command, see “SET LDRTBLS Command” on page 4.

SNMP DPI Reference

The following table provides a reference for SNMP DPI. Table 71 describes each SNMP DPI routine supported by TCP/IP, and identifies the page in the book where you can find more information.

Table 71. SNMP DPI Reference

SNMP DPI Routine	Description	page
DPIdebug()	Used to turn some DPI internal tracing on or off.	329
fDPIparse()	Frees a parse tree previously created by a call to pDPIpacket().	330
mkDPIlist()	Creates the portion of the parse tree that represents a list of name and value pairs.	330
mkDPIregister()	Creates a register request packet and returns a pointer to a static buffer.	331
mkDPIresponse()	Creates a response packet.	331
mkDPIset()	Creates a representation of a parse tree name and value pair.	332
mkDPItrap()	Creates a trap request packet.	333
mkDPItrape()	Creates an extended trap. Basically the same as the mkDPItrap() routine but allows you to pass a list of variables and an enterprise object ID.	334
pDPIpacket()	Parses a DPI packet and returns a parse tree representation.	335
query_DPI_port()	Determines what TCP port is associated with DPI.	336

DPI Library Routines

This section provides the syntax, parameters, and other appropriate information for each DPI routine supported by TCP/IP Level 310 for VM.

DPIdebug()

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

void DPIdebug(onoff)
int *onoff;
```

DPIdebug()

Parameter	Description
<i>onoff</i>	Specifies an integer. A value of 0 turns tracing off and a value of 1 (or nonzero) turns tracing on.

Description: The DPIdebug() routine can be used to turn DPI internal tracing on or off.

fDPIparse()

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

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

Parameter	Description
<i>hdr</i>	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()

```
#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;
```

Parameter	Description
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`:

```
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()

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

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

Parameter	Description
-----------	-------------

<i>oid_name</i>	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.

```
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()

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

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

Parameter	Description
-----------	-------------

<i>ret_code</i>	Determines the error code to be returned.
<i>value_list</i>	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 parameter, `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 parameter 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 parameter 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.

```
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()

```
#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;
```

Parameter	Description
<code>oid_name</code>	Specifies the object identifier of the variable.
<code>type</code>	Specifies the type of the object identifier.
<code>len</code>	Indicates the length of the value.
<code>value</code>	Points to the first byte of the value of the object identifier.

Description: The `mkDPIset()` routine can be used to create the portion of a parse tree that represents a name and value pair (as would normally be returned in a response packet). It returns a pointer to a dynamically allocated parse tree representing the name, type, and value information. If 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* parameter 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 parameters. After a successful call to mkDPIset(), the application can dispose of the passed parameters without affecting the contents of the parse tree.

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

mkDPItrap()

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

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

Parameter	Description
<i>generic</i>	Specifies the generic field in the SNMP TRAP packet.
<i>specific</i>	Identifies the specific field in the SNMP TRAP packet.
<i>value_list</i>	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.

```
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()

```
        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()

```
#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;
```

Parameter	Description
-----------	-------------

generic	Specifies the generic field for the SNMP TRAP packet.
----------------	---

specific	Specifies the specific field for the SNMP TRAP packet.
-----------------	--

value_list	Specifies a pointer to a structure <code>dpi_set_packet</code> , which contains one or more variables to be sent with the SNMP TRAP packet. Or NULL if no variables are to be send.
-------------------	---

enterprise_oid	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.
-----------------------	---

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.

```
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,"1.3.6.1.4.1.2.2.1.4.1",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 */
```


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.

```
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.

pDPIDpacket()

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

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

Parameter	Description
<i>packet</i>	Specifies the DPI packet to be parsed.

Description: The `pDPIDpacket()` 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 `pDPIDpacket()`, the packet can be disposed of in any manner the application chooses, without affecting the contents of the parse tree.

Return Values: If `pDPIDpacket()` 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 `mkDPITrap()` 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;
};
```

pDPIpacket()

The *packet_type* field can have one of the following values, which are defined in the SNMP_DPI.H header file:

- SNMP_DPI_GET
- SNMP_DPI_GET_NEXT
- SNMP_DPI_SET

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

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

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

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

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

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

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

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()

```
#include <snmp_dpi.h>

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

Parameter	Description
-----------	-------------

<i>host_name</i>	Points to the SNMP agent's host name or internet address.
------------------	---

<i>community_name</i>	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.

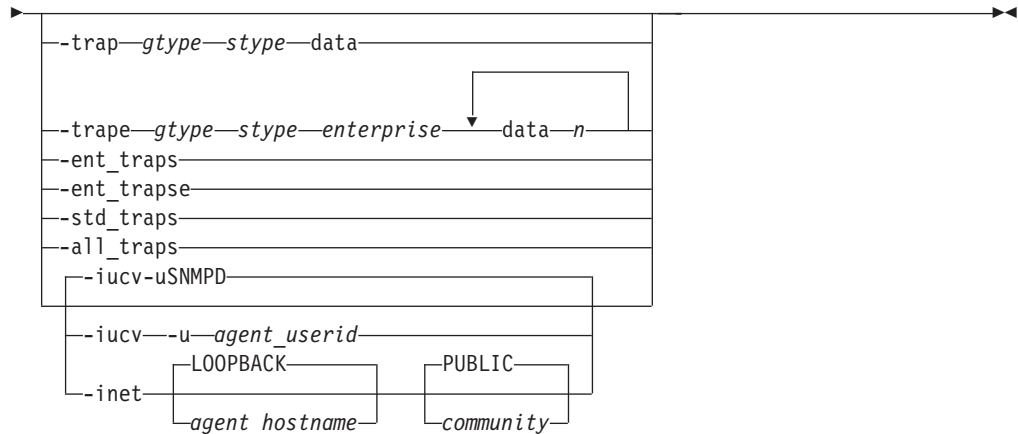
The DPISAMPLE program implements a set of variables in the DPISAMPLE table which consists of a set of objects in the IBM Research tree (1.4.1.2.2.1.4). See Figure 34 on page 339 for the object type and objectID.

The DPISAMPLE Program (Sample DPI Subagent)

The DPISAMPLE program accepts the following arguments:



The DPISAMPLE Program (Sample DPI Subagent)



Parameter	Description
?	Invokes output with an explanation about how the <i>dpisample</i> command is used. This option should be used in a C-shell environment.
-d n	Sets the debug level. The level, <i>n</i> , 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: <ul style="list-style-type: none"> <i>gtype</i> Specifies the type as generic. The available ranges are 0 — 6. <i>stype</i> Specifies the type as specific. data Passes data as an additional value for the variable <i>dpiSample.stype.0</i>. Data is interpreted depending on <i>stype</i>. The following list describes the available values for the <i>stype</i> parameter and their data descriptions: <ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> <i>gtype</i> Specifies the trap as generic. The available ranges are 0 — 6. <i>stype</i> Specifies the type as specific. <i>enterprise</i> 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.

The DPISAMPLE Program (Sample DPI Subagent)

-ent_traps	Generates nine enterprise-specific traps with <i>stype</i> values of 1 — 9, using the internal dpiSample variables as data.
-ent_trapse	Generates nine enterprise-specific traps with <i>stype</i> 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 <i>stype</i> of 1 — 9 (-ent_traps).
-iucv	Specifies that an AF_IUCV socket is to be used to connect to the SNMP agent. The <i>-iucv</i> parameter 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 <i>-inet</i> 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 34. 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.
*/
*/
```

Client Sample Program

```
/* Jun 1992 - Make it run on OS/2 as well */
/* Note: dpisample = dpisamp1 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 */
/* MIB@DESC.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 <netdb.h>
#include <inet.h>
extern char ebcdicto][, asciitoe] [];
#pragma linkage(cmxlte, OS)
#define DO_ETOA(a) cmxlte((a), ebcdictoascii, strlen((a)))
#define DO_ATOE(a) cmxlte((a), asciitoebcdic, strlen((a)))
#define DO_ERROR(a) tcperror((a))
#define LOOPBACK "loopback"
#define IUCV TRUE
#define max(a,b) (((a) > (b)) ? (a) : (b))
#define min(a,b) (((a) < (b)) ? (a) : (b))

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

#ifdef OS2
#define INCL_DOSPROCESS
#include <stdlib.h>
#include <types.h>
// #include <doscalls.h> /* GKS */
#include <os2.h> /* GKS */
#ifndef sleep
#define sleep(a) DosSleep(1000L * (a)) /*GKS*/
#endif
#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>
#define DO_ETOA(a) ; /* no need for this */
#define DO_ATOE(a) ; /* no need for this */
#define DO_ERROR(a) perror((a))
#define LOOPBACK "localhost"
#define IUCV FALSE
#ifdef AIX221
#define isdigit(c) (((c) >= '0') && ((c) <= '9'))
#else
// #include <sys/select.h>
#endif /* AIX221 */

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

#include <stdio.h>
#ifdef OS2
#include <dpi/snmp_dpi.h>
#else
#include "snmp_dpi.h"
#endif

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

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

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

#else /* 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 *programe, const int exit_rc);
static void init_connection(void);
static void init_variables(void);
static void await_and_read_packet(void);
static void handle_packet(void);
static void do_get(void);
static void do_set(void);
static void issue_traps(void);
static void issue_one_trap(void);
static void issue_one_trap(void);

```

Client Sample Program

```
static void issue_std_traps(void);
static void issue_ent_traps(void);
static void issue_ent_trapse(void);
static void do_register(void);
static void dump_bfr(const char *buf, const int len);
static struct dpi_set_packet *addtoaset(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 */
    -1, /* SNMP_TYPE_EMPTY */ /* 4 do not check type */
    SNMP_TYPE_INTERNET, /* 5 internet address */
    SNMP_TYPE_COUNTER, /* 6 counter */
    SNMP_TYPE_GAUGE, /* 7 gauge */
    SNMP_TYPE_TICKS, /* 8 time ticks */
    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 */
```


Client Sample Program

```

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

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

#ifdef _NO_PROTO                               /* for classic K&R C */
static void issue_traps()
#else /* _NO_PROTO */                          /* for ANSI-C compiler */
static void issue_traps(void)
#endif /* _NO_PROTO */
{
    switch (do_trap) {                         /* let's see which one(s) */

```

Client Sample Program

```
case ONE_TRAP:                /* only need to issue one trap */
    issue_one_trap();         /* go issue the one trap */
    break;
case ONE_TRAPE:               /* only need to issue one trape */
    issue_one_trapse();       /* go issue the one trape */
    break;
case STD_TRAPS:               /* only need to issue std traps */
    issue_std_traps();        /* standard traps gtypes 0-5 */
    break;
case ENT_TRAPS:               /* only need to issue ent traps */
    issue_ent_traps();        /* enterprise specific traps */
    break;
case ENT_TRAPSE:              /* only need to issue ent trapse */
    issue_ent_trapse();       /* enterprise specific trapse */
    break;
case ALL_TRAPS:               /* only need to issue std traps */
    issue_std_traps();        /* standard traps gtypes 0-5 */
    issue_ent_traps();        /* enterprise specific traps */
    issue_ent_trapse();       /* enterprise specific trapse */
    break;
default:
    break;
}                               /* end switch (do_trap) */
do_trap = 0;                    /* reset do_trap switch */
}

#ifdef _NO_PROTO                /* for classic K&R C */
static void await_and_read_packet() /* await packet from DPI agent */
#else /* _NO_PROTO */           /* for ANSI-C compiler */
static void await_and_read_packet(void) /* await packet from DPI agent */
#endif /* _NO_PROTO */
{
    int len, rc, bytes_to_read, bytes_read = 0;
#ifdef OS2
    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\n");
            close(dpi_fd);
            exit(1);
        } else {                       /* count bytes_read */
            bytes_read += len;
            bytes_to_read -= len;
        }
    } /* while (bytes_to_read > 0) */
}

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

    if (debug_lvl > 2) {
        printf("Received following SNMP-DPI packet:\n");
        dump_bfr(inbuf, PACKET_LEN(inbuf));
    }
    hdr = pDPIpacket(inbuf);           /* parse received packet */
    if (hdr == 0) {                    /* ignore if can't parse */
        printf("Ignore received packet, could not parse it!\n");
        return;
    }
    packet = NULL;
    var_type = 0;
    var_oid = "";
    var_gid = "";
    switch (hdr->packet_type) {
        /* extract pointers and/or data from specific packet types, */
        /* such that we can use them independent of packet type. */
        case SNMP_DPI_GET:
            if (debug_lvl > 0) printf("SNMP_DPI_GET for ");
            var_oid = hdr->packet_body.dpi_get->object_id;
            break;
        case SNMP_DPI_GET_NEXT:
            if (debug_lvl > 0) printf("SNMP_DPI_GET_NEXT for ");
            var_oid = hdr->packet_body.dpi_next->object_id;
            var_gid = hdr->packet_body.dpi_next->group_id;
            break;
        case SNMP_DPI_SET:
            if (debug_lvl > 0) printf("SNMP_DPI_SET for ");

```

Client Sample Program

```
var_value_len = hdr->packet_body.dpi_set->value_len;
var_value     = hdr->packet_body.dpi_set->value;
var_oid       = hdr->packet_body.dpi_set->object_id;
var_type      = hdr->packet_body.dpi_set->type;
break;
default: /* Return a GEN_ERROR */
    if (debug_lvl > 0) printf("Unexpected packet_type %d, genErr\n",
                             hdr->packet_type);
    packet = mkDPIresponse(SNMP_GEN_ERR, NULL);
    fDPIparse(hdr); /* return storage allocated by pDPIpacket() */
    send_packet(packet);
    return;
    break;
} /* end switch(hdr->packet_type) */
if (debug_lvl > 0) printf("objectID: %s \n",var_oid);

if (strlen(var_oid) <= strlen(OID)) { /* not in our tree */
    if (hdr->packet_type == SNMP_DPI_GET_NEXT) var_index = 0; /* OK */
    else { /* cannot handle */
        if (debug_lvl>0) printf("...Ignored %s, noSuchName\n",var_oid);
        packet = mkDPIresponse(SNMP_NO_SUCH_NAME, NULL);
        fDPIparse(hdr); /* return storage allocated by pDPIpacket() */
        send_packet(packet);
        return;
    }
} else { /* Extract our variable index (from OID.index.instance) */
    /* We handle any instance the same (we only have one instance) */
    var_index = atoi(&var_oid[strlen(OID)]);
}
if (debug_lvl > 1) {
    printf("...The groupID=%s\n",var_gid);
    printf("...Handle as if objectID=%s%d\n",OID,var_index);
}
switch (hdr->packet_type) {
case SNMP_DPI_GET:
    do_get(); /* do a get to return response */
    break;
case SNMP_DPI_GET_NEXT:
    { char toid]256[; /* space for temporary objectID */
      var_index++; /* do a get for the next variable */
      sprintf(toid,"%s%d",OID,var_index); /* construct objectID */
      var_oid = toid; /* point to it */
      do_get(); /* do a get to return response */
    } break;
case SNMP_DPI_SET:
    if (debug_lvl > 1) printf("...value_type=%d\n",var_type);
    do_set(); /* set new value first */
    if (packet) break; /* some error response was generated */
    do_get(); /* do a get to return response */
    break;
}
fDPIparse(hdr); /* return storage allocated by pDPIpacket() */
}

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

    switch (var_index) {
    case 0: /* table, cannot be queried by itself */
        printf("...Should not issue GET for table %s.0\n", OID);
        break;
    case 1: /* a number */
```

```

    data = mkDPISet(var_oid,SNMP_TYPE_NUMBER,sizeof(number),&number);
    break;
case 2: /* an octet_string (can have binary data) */
    data = mkDPISet(var_oid,SNMP_TYPE_STRING,ostring_len,ostring);
    break;
case 3: /* object id */
    data = mkDPISet(var_oid,SNMP_TYPE_OBJECT,objectID_len,objectID);
    break;
case 4: /* some empty variable */
    data = mkDPISet(var_oid,SNMP_TYPE_EMPTY,0,NULL);
    break;
case 5: /* internet address */
    data = mkDPISet(var_oid,SNMP_TYPE_INTERNET,sizeof(ipaddr),&ipaddr);
    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 = mkDPISet(SNMP_NO_ERROR,data);
} else { /* Could have been an error in mkDPISet though */
    if (debug_lvl > 0) printf("...Sending response noSuchName\n");
    packet = mkDPISet(SNMP_NO_SUCH_NAME,NULL);
} /* end if (data) */
if (packet) send_packet(packet);
}

#ifdef _NO_PROTO
static void do_set() /* handle SNMP_SET request */
#else /* _NO_PROTO */
static void do_set(void) /* handle SNMP_SET request */
#endif /* _NO_PROTO */
{
    unsigned long *ulp;
    long *lp;

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

```

Client Sample Program

```
    printf(" Returning badValue\n");
    packet = mkDPIresponse(SNMP_BAD_VALUE, NULL);
    if (packet) send_packet(packet);
    return;
}
switch (var_index) {
case 0: /* table, cannot set table. */
    if (debug_lvl > 0) printf("...Ignored set TABLE, noSuchName\n");
    packet = mkDPIresponse(SNMP_NO_SUCH_NAME, NULL);
    break;
case 1: /* a number */
    lp = (long *)var_value;
    number = *lp;
    break;
case 2: /* an octet_string (can have binary data) */
    free(ostring);
    ostring = (char *)malloc(var_value_len + 1);
    bcopy(var_value, ostring, var_value_len);
    ostring_len = var_value_len;
    ostring[var_value_len] = '\0'; /* so we can use it as a string */
    break;
case 3: /* object id */
    free(objectID);
    objectID = (char *)malloc(var_value_len + 1);
    bcopy(var_value, objectID, var_value_len);
    objectID_len = var_value_len;
    if (objectID[objectID_len - 1]) {
        objectID[objectID_len++] = '\0'; /* a valid one needs a null */
        if (debug_lvl > 0)
            printf("...added a terminating null to objectID\n");
    }
    break;
case 4: /* an empty variable, cannot set */
    if (debug_lvl > 0) printf("...Ignored set EMPTY, readOnly\n");
    packet = mkDPIresponse(SNMP_READ_ONLY, NULL);
    break;
case 5: /* Internet address */
    ulp = (unsigned long *)var_value;
    ipaddr = *ulp;
    break;
case 6: /* counter (unsigned) */
    ulp = (unsigned long *)var_value;
    counter = *ulp;
    break;
case 7: /* gauge (unsigned) */
    ulp = (unsigned long *)var_value;
    gauge = *ulp;
    break;
case 8: /* time ticks (unsigned) */
    ulp = (unsigned long *)var_value;
    ticks = *ulp;
    break;
case 9: /* a display_string (printable ascii only) */
    free(dstring);
    dstring = (char *)malloc(var_value_len + 1);
    bcopy(var_value, dstring, var_value_len);
    dstring[var_value_len] = '\0'; /* so we can use it as a string */
    DO_ATOE(dstring);
    break;
case 10: /* a request to execute a command */
    free(command);
    command = (char *)malloc(var_value_len + 1);
    bcopy(var_value, command, var_value_len);
    command[var_value_len] = '\0'; /* so we can use it as a string */
    DO_ATOE(command);
    if (strcmp("all_traps", command) == 0) do_trap = ALL_TRAPS;
    else if (strcmp("std_traps", command) == 0) do_trap = STD_TRAPS;
```

```

else if (strcmp("ent_traps",command) == 0) do_trap = ENT_TRAPS;
else if (strcmp("ent_trapse",command) == 0) do_trap = ENT_TRAPSE;
else if (strcmp("all_traps",command) == 0) do_trap = ALL_TRAPS;
else if (strcmp("quit",command) == 0) do_quit = TRUE;
else break;
if (debug_lvl > 0)
    printf("...Action requested: %s set to: %s\n",
        DPI_var]10[, command);
break;
default: /* NoSuchName */
if (debug_lvl > 0)
    printf("...Ignored set for %s, NoSuchName\n", var_oid);
packet = mkDPIresponse(SNMP_NO_SUCH_NAME,NULL);
break;
} /* end switch (var_index) */
if (packet) send_packet(packet);
}

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

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

    trap_gtype = 6;
    for (trap_stype = 1; trap_stype < 10; trap_stype++) {
        trap_data = temp_string;
        switch (trap_stype) {
            case 1 :
                sprintf(temp_string,"%ld",number);
                break;
            case 2 :
                sprintf(temp_string,"%s",ostring);
                break;
            case 3 :
                trap_data = objectID;
                break;
            case 4 :
                trap_data = "";
                break;
            case 5 :
                trap_data = dpi_hostname;
                break;
            case 6 :
                sleep(1); /* give manager a break */
                sprintf(temp_string,"%lu",counter);
                break;
            case 7 :
                sprintf(temp_string,"%lu",gauge);
                break;
            case 8 :

```

Client Sample Program

```
        sprintf(temp_string,"%lu",ticks);
        break;
    case 9 :
        trap_data = dstring;
        break;
    } /* end switch (trap_stype) */
    issue_one_trap();
}

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

    trape_gtype = 6;
    trape_eprise = ENTERPRISE_OID;
    for (n=11; n < (11+OID_COUNT_FOR_TRAPS); n++) {
        data = 0;
        trape_stype = n;
        for (i=1; i<=(n-10); i++)
            data = addtoset(data, i);
        if (data == 0) {
            printf("Could not make dpi_set_packet\n");
            return;
        }
        packet = mkDPITrape(trape_gtype,trape_stype,data,trape_eprise);
        if ((debug_lvl > 0) && (packet)) {
            printf("sending trape packet: %lu %lu enterprise=%s\n",
                trape_gtype, trape_stype, trape_eprise);
        }
        if (packet) send_packet(packet);
        else printf("Could not make trape packet\n");
    }
}

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

    for (i=0; i<trape_datacnt; i++) {
        sprintf(oid,"%s2.%d",OID,i);
        /* assume an octet_string (could have hex data) */
        data = mkDPILIST(data, oid, SNMP_TYPE_STRING,
            strlen(trape_data[i]), trape_data[i]);
    }
}
```



```

    if (data == 0) {
        printf("Could not make dpiset_packet\n");
    } else if (debug_lvl > 0) {
        printf("Preparing: ]oid=%s[ value: ", oid);
        printf("");
        for (cp = trape_data[i]; *cp; cp++) /* loop through data */
            printf("%2.2x",*cp);          /* hex print one byte */
        printf("H\n");
    }
}
packet = mkDPITrape(trape_gtype,trape_stype,data,trape_eprise);
if ((debug_lvl > 0) && (packet)) {
    printf("sending trape packet: %lu %lu enterprise=%s\n",
        trape_gtype, trape_stype, trape_eprise);
}
if (packet) send_packet(packet);
else printf("Could not make trape packet\n");
}

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

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

```

Client Sample Program

```
        break;
    case 5: /* internet address */
        ipaddr = lookup_host(trap_data);
        data = mkDPIset(oid, SNMP_TYPE_INTERNET, sizeof(ipaddr), &ipaddr);
        break;
    case 6: /* counter (unsigned) */
        ulnum = strtoul(trap_data, (char **)0, 10);
        data = mkDPIset(oid, SNMP_TYPE_COUNTER, sizeof(ulnum), &ulnum);
        break;
    case 7: /* gauge (unsigned) */
        ulnum = strtoul(trap_data, (char **)0, 10);
        data = mkDPIset(oid, SNMP_TYPE_GAUGE, sizeof(ulnum), &ulnum);
        break;
    case 8: /* time ticks (unsigned) */
        ulnum = strtoul(trap_data, (char **)0, 10);
        data = mkDPIset(oid, SNMP_TYPE_TICKS, sizeof(num), &ulnum);
        break;
    case 9: /* a display_string (ascii only) */
        DO_ETOA(trap_data);
        data = mkDPIset(oid, SNMP_TYPE_STRING, strlen(trap_data), trap_data);
        DO_ATOE(trap_data);
        break;
    default: /* handle as string */
        printf("Unknown specific trap type: %s, assume octet_string\n",
              trap_stype);
        data = mkDPIset(oid, SNMP_TYPE_STRING, strlen(trap_data), trap_data);
        break;
    } /* end switch (trap_stype) */
    break;
default: /* unknown trap */
    printf("Unknown general trap type: %s\n", trap_gtype);
    return;
    break;
} /* end switch (trap_gtype) */

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

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

    if (debug_lvl > 2) {
        printf("...Sending DPI packet:\n");
        dump_bfr(packet, PACKET_LEN(packet));
    }
#ifdef OS2
    rc = send(dpi_fd, packet, PACKET_LEN(packet), 0);
#else
    rc = write(dpi_fd, packet, PACKET_LEN(packet));
#endif
}
```

```

#endif
    if (rc != PACKET_LEN(packet)) DO_ERROR("send_packet: write");
    /* no need to free packet (static buffer in mkDPI.... routine) */
}

#ifdef _NO_PROTO
static void do_register() /* register our objectIDs with agent */
#else /* _NO_PROTO */
static void do_register(void) /* register our objectIDs with agent */
#endif /* _NO_PROTO */
{
    int i, rc;
    char toid]256[;

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

/* add specified variable to list of variable in the dpi_set_packet
*/
#ifdef _NO_PROTO
struct dpi_set_packet *addtoset(data, stype)
struct dpi_set_packet *data;
int stype;
#else /* _NO_PROTO */
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 */

```

Client Sample Program

```
        data = mkDPiIlist(data, var_oid, SNMP_TYPE_INETNET,
                          sizeof(ipaddr), &ipaddr);
        break;
    case 6: /* counter (unsigned) */
        data =mkDPiIlist(data, var_oid, SNMP_TYPE_COUNTER,
                          sizeof(counter), &counter);

        break;
    case 7: /* gauge (unsigned) */
        data = mkDPiIlist(data, var_oid, SNMP_TYPE_GAUGE,
                          sizeof(gauge), &gauge);

        break;
    case 8: /* time ticks (unsigned) */
        data = mkDPiIlist(data, var_oid, SNMP_TYPE_TICKS,
                          sizeof(ticks), &ticks);

        break;
    case 9: /* a display_string (printable ascii only) */
        DO_ETOA(dstring);
        data = mkDPiIlist(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("%d\n",number);
        break;
    case 2 :
        printf("");
        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\n", (ipaddr >> 24), ((ipaddr << 8) >> 24),
        * ((ipaddr << 16) >> 24), ((ipaddr << 24) >> 24));
        */
        break;
    case 6 :
        printf("%lu\n",counter);
        break;
    case 7 :
        printf("%lu\n",gauge);
        break;
    case 8 :
        printf("%lu\n",ticks);
    }
```

```

        break;
    case 9 :
        printf("%s\n",dstring);
        break;
    case 10 :
        printf("%s\n",command);
        break;
    } /* end switch(index) */
}

#ifdef _NO_PROTO                                /* for classic K&R C */
static void check_arguments(argc, argv)        /* check arguments */
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_s_type = 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_s_type = strtoul(argv[i+2],(char**)0,10);
                trape_eprise = argv[i+3];
                for (i = i + 4, j = 0;
                    (argc > i) && (j < MAX_TRAPE_DATA);
                    i++, j++) {
                    trape_data[j] = argv[i];
                }
                trape_datacnt = j;
                do_trap = ONE_TRAPE;
                break; /* -trape must be last option */
            }
            else usage(argv[0], 1);
        }
        else if (strcmp(argv[i],"-all_traps") == 0) {
            do_trap = ALL_TRAPS;
        }
        else if (strcmp(argv[i],"-std_traps") == 0) {
            do_trap = STD_TRAPS;
        }
        else if (strcmp(argv[i],"-ent_traps") == 0) {
            do_trap = ENT_TRAPS;
        }
        else if (strcmp(argv[i],"-ent_trapse") == 0) {
            do_trap = ENT_TRAPSE;
        }
#ifdef (VM) || defined(MVS)
        else if (strcmp(argv[i],"-inet") == 0) {
            use_iucv = 0;
        }
        else if (strcmp(argv[i],"-iucv") == 0) {
            use_iucv = TRUE;
        }
        else if (strcmp(argv[i],"-u") == 0) {

```

Client Sample Program

```
        use_iucv = TRUE; /* -u implies -iucv */
        i++;
        if (argc > i) {
            dpi_userid = argv[i];
        }
    #endif
    } else if (strcmp(argv[i], "?") == 0) {
        usage(argv[0], 0);
    } else {
        if (hname == NULL) hname = argv[i];
        else if (cname == NULL) cname = argv[i];
        else usage(argv[0], 1);
    }
    }
    if (hname == NULL) hname = LOOPBACK;          /* use default */
    if (cname == NULL) cname = PUBLIC_COMMUNITY_NAME; /* use default */
    #if defined(VM) || defined(MVS)
    if (dpi_userid == NULL) dpi_userid = SNMPAGENTUSERID;
    if (debug_lvl > 2)
        printf("hname=%s, cname=%s, userid=%s\n", hname, cname, dpi_userid);
    #else
    if (debug_lvl > 2)
        printf("hname=%s, cname=%s\n", hname, cname);
    #endif
    if (use_iucv != TRUE) {
        DO_ETOA(cname);                          /* for VM or MVS */
        dpi_port = query_DPI_port(hname, cname);
        DO_ATOE(cname);                          /* for VM or MVS */
        if (dpi_port == -1) {
            printf("No response from agent at %s(%s)\n", hname, cname);
            exit(1);
        }
    } else dpi_port == -1;
    dpi_hostname = hname;
}

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

#ifdef _NO_PROTO                                /* for classic K&R C */
static void init_variables()
#else /* _NO_PROTO */                          /* for ANSI-C compiler */
static void init_variables(void)
#endif /* _NO_PROTO */                          /* initialize our variables */
```

```

#endif /* _NO_PROTO */
{
    char ch, *cp;

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

}

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

    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_IUCV_NAME, siu.siucv_name, min(8,strlen(SNMP_IUCV_NAME)));
        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);
#ifdef VM || defined (MVS)
    }
#endif
}
#endif
if (dpi_fd < 0) {                /* exit on error */

```

Client Sample Program

```
        DO_ERROR("init_connection: socket");
        exit(1);
    }
    rc = connect(dpi_fd, sa, sasize);          /* connect to agent */
    if (rc != 0) {                             /* exit on error */
        DO_ERROR("init_connection: connect");
        close(dpi_fd);
        exit(1);
    }
}

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

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

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

Chapter 10. 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*—————▶▶

Parameter	Description
<i>domain_name</i>	Specifies the domain name of the sending host. The <i>domain_name</i> may be specified as either: <ul style="list-style-type: none">• 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.

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.

EHLO

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.

▶▶—EHLO—*domain_name*—◀◀

Parameter	Description
<i>domain_name</i>	Specifies the domain name of the sending host. The <i>domain_name</i> may be specified as either: <ul style="list-style-type: none"> • 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.

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:

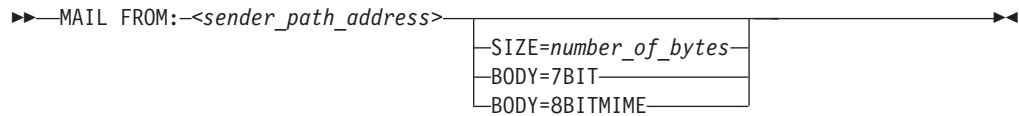
```
s: (wait for connection on TCP port 25)
c: (open connection to server)
s: 220 HOSTA.IBM.COM running IBM VM SMTP Level 320 on Sat, 1 May 99 ...
c: EHLO HOSTB.IBM.COM
s: 250-HOSTA.IBM.COM is my domain name.
s: 250-EXPN
s: 250-HELP
s: 250-8BITMIME
s: 250 SIZE 524288
...
```

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.

SMTP Virtual Machine Interfaces



Parameter	Description
<i>sender_path_address</i>	Specifies the full path address of the sender of the mail. Definitions for valid <i>sender_path_address</i> 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 360.
SIZE=number_of_bytes	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.
BODY=7BIT	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.
BODY=8BITMIME	Specifies that the message is encoded using all eight bits of each octet (byte) and may contain MIME headers.

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.



Parameter	Description
<i>recipient_path_address</i>	Specifies the full path address of a mail recipient. Definitions for valid <i>recipient_path_address</i> 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 360.

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 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.

SMTP Virtual Machine Interfaces

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).

▶▶—NOOP—▶▶

The NOOP command has no parameters.

HELP

The HELP command returns brief information about one or more SMTP commands.

```

>> HELP _____>>
      |_____|
      |command_name|
  
```

Parameter	Description
-----------	-------------

<i>command_name</i>	Identifies a specific SMTP command.
---------------------	-------------------------------------

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.

```

>> QUEU _____>>
      |_____|
      |DATE|
  
```

Parameter	Description
-----------	-------------

DATE	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.
-------------	---

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:

Queue Name	Description
------------	-------------

Spool	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 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 <i>Active</i> .
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 <i>Active</i> 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:

SMTP Virtual Machine Interfaces

- 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.

Queue Name Description

Process This queue is generally empty, as it contains queries that have not 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 sever, 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.

SMTP Virtual Machine Interfaces

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* 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* 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.

VERFY

The VRFY (“verify”) command determines if a given mailbox or user ID exists on the host where SMTP is running.

►►—VRFYverify_string—◄◄

Parameter	Description
verify_string	Specifies the name of a mailbox or user ID whose existence is to be verified.

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.

SMTP Virtual Machine Interfaces

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@abcvml.somewhere.com>

vrfy tcpadmin-list 250-<tcpmaint@abcvml.somewhere.com>
250-<tcpadmin@abcvml.somewhere.com>
250-<tcpadmin@adminpc.somewhere.com>
250 <maint@abcvml.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.

▶—EXPN*expand_string*—▶

Parameter	Description
-----------	-------------

<i>expand_string</i>	Specifies the name of the mailing list to be expanded.
----------------------	--

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.

▶—VERB

OFF
ON

—▶

Parameter	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” on page 369 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.

►►—TICK—*identifier*—◄◄

Parameter	Description
<i>identifier</i>	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>
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
```

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:

Digit	Description
0	Echo reply; used only in batch SMTP response files. Received commands are “echoed” in these files to provide contextual information for other reply codes.

SMTP Virtual Machine Interfaces

- 1 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.
- 2 Positive Completion reply; command accepted.
- 3 Positive Intermediate reply; data associated with the command should now be provided.
- 4 Temporary Negative Completion reply; try the command again, but at a later time.
- 5 Permanent Negative Completion reply; the command has been rejected.

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* (1024 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 360.

Batch SMTP Command Files

Batch SMTP command files are files that contain an SMTP envelope (as described in “SMTP Transactions” on page 359) 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.

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 send 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.

SMTP Virtual Machine Interfaces

```
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 369.

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:n 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
050 RCPT TO: <alice@host2.somewhere.com>
250 OK
050 DATA
354 Enter mail body. End by new line with just a '.'
250 Mail Delivered
050 QUIT
221
YOURHOST.YOURDOMAIN.EDU running IBM VM SMTP Level 320 closing connection
```

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 Function Level 320 on the TCPMAINT 592 “Client-code” minidisk) generates such a file and sends it to the SMTP virtual machine.

Sample content for a BSMTP 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:n EST
050-VERB ON
050
250 Verbose 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 <MSG@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.

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

SMTP Virtual Machine Interfaces

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-Warning: localhost: Could not confirm that host [ipaddr] is helodata
```

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 376.

REXX Parameter List

Inputs:

Table 72. Client Verification REXX Exit Parameter List

Argument	Description
ARG(1)	Parameter list defined as follows: <ul style="list-style-type: none"> • 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:

RC	Return String
----	---------------

Argument Description

RC The exit return code; this must be a 4-byte binary value.

Return String An exit-specified string; the returned value must have a length less than or equal to the maximum length passed to the exit.

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 73. Client Verification ASSEMBLER Exit Parameter List

Offset in Decimal	Len	In/Out	Type	Description
+0	4	Input	Char	Exit type
+4	4	Input	Int	Version number
+8	4	Input	Int	Mail record ID
+12	4	Input	Int	Port number of SMTP server
+16	4	Input	Int	IP address of SMTP server
+20	4	Input	Int	Port number of client
+24	4	Input	Int	IP address of client
+28	4	Input	Ptr	Address of SMTP command string
+32	4	Input	Int	Length of SMTP command string

SMTP Virtual Machine Interfaces

Table 73. Client Verificaiton ASSEMBLER Exit Parameter List (continued)

Offset in Decimal	Len	In/Out	Type	Description
+36	4	Input	Ptr	Address of HELO/EHLO name
+40	4	Input	Int	Length of HELO/EHLO name
+44	24			Not used
+68	4	Input	Int	Verify Client status
+72	4	Output	Ptr	Address of Return String
+76	4	Output	Int	Length of Return String
+80	4	Input	Int	Maximum length of Return String
+84	8			Not used
+92	4	Input/ Output	Char	User Word 1
+96	4	Input/ Output	Char	User Word 2
+100	4	Output	Int	Return code from exit

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

SMTP Virtual Machine Interfaces

- 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:

- 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 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 74. Client Verification Exit Return Codes

Return Code	Explanation
0	Do not verify client. A comment will not be inserted in the mail header.
1	Perform verification using the built-in client verification function.
2	Mark as verified. The following comment will be inserted in the mail header: <i>X-Comment: localhost: Mail was sent by host</i>
3	The following comment will be inserted in the mail header: <i>X-Comment: Return String</i>

where the value for *Return String* can be specified by the exit.

SMTP Virtual Machine Interfaces

Table 74. Client Verification Exit Return Codes (continued)

Return Code	Explanation
4	<p>Disable the exit. The following message will be displayed on the SMTP console:</p> <pre>VERIFYCLIENT EXIT function disabled</pre> <p>The exit will no longer be called. The client will not be verified and no comment will be inserted in the mail header.</p>
5	<p>Reject this command with: <code>550 Return String</code></p> <p>If a return string is not provided by the exit, then the default message will be displayed:</p> <pre>550 Access denied</pre> <p>All future communications on this connection will be rejected with this 550 message.</p>
Other	<p>Any return code other than the above causes SMTP to issue this message:</p> <pre>Unexpected return from user exit <i>exitname</i> <i>exittype</i>, RC = <i>rc</i></pre> <p>SMTP treats this return code as if it were a return code of 0.</p>

Client Verification Sample Exits

Sample Client Verification exit routines are supplied with TCP/IP on the TCPMAINT 591 disk. The supplied samples are:

SMTPVERX SEXEC

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.

SMTP Virtual Machine Interfaces

- 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 381.

REXX Parameter List

Inputs:

Table 75. Mail Forwarding REXX Exit Parameter List

Argument	Description
ARG(1)	Parameter list defined as follows: <ul style="list-style-type: none"> • 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)	MAIL FROM: string
ARG(5)	Client domain name
ARG(6)	Not used

Outputs:

The following are returned to the caller in the RESULT variable via a REXX RETURN statement:

RC	Return String
----	---------------

Argument Description

RC The exit return code; this must be a 4-byte binary value.

Return String An exit-specified string; the returned value must have a length less than or equal to the maximum length passed to the exit.

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 76. Mail Forwarding ASSEMBLER Exit Parameter List

Offset in Decimal	Len	In/Out	Type	Description
+0	4	Input	Char	Exit type
+4	4	Input	Int	Version number
+8	4	Input	Int	Mail record ID
+12	4	Input	Int	Port number of SMTP server
+16	4	Input	Int	IP address of SMTP server

Table 76. Mail Forwarding ASSEMBLER Exit Parameter List (continued)

Offset in Decimal	Len	In/Out	Type	Description
+20	4	Input	Int	Port number of client
+24	4	Input	Int	IP address of client
+28	4	Input	Ptr	Address of SMTP command string
+32	4	Input	Int	Length of SMTP command string
+36	4	Input	Ptr	Address of HELO/EHLO name
+40	4	Input	Int	Length of HELO/EHLO name
+44	4	Input	Ptr	Address of client domain name
+48	4	Input	Int	Length of client domain name
+52	4	Input	Ptr	Address of MAIL FROM: string
+56	4	Input	Int	Length of MAIL FROM: string
+60	8	Input	Char	File name of note on disk
+68	4	Input	Int	Verify Client status
+72	4	Output	Ptr	Address of Return String
+76	4	Output	Int	Length of Return String
+80	4	Input	Int	Maximum length of Return String
+84	8			Not used
+92	4	Input/ Output	Char	User Word 1
+96	4	Input/ Output	Char	User Word 2
+100	4	Output	Int	Return code from exit

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,

SMTP Virtual Machine Interfaces

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 77. Mail Forwarding Exit Return Codes

Return Code	Explanation
0	Accept and attempt mail delivery.
1	Reject mail with: 551 <i>Return String</i> If a return string is not provided by the exit, the following default message will be used: 551 User not local; please try <i>user@otherhost</i> If the server has already responded to the command, this return code will result in error mail being sent back to the sender.
2	Accept and forward to the local VM user ID specified by <i>Return String</i> . 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.
4	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.
5	Reject this command with: 550 <i>Return String</i> 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.
Other	Any return code other than the above causes SMTP to issue this message: Unexpected return from user exit <i>exitname exittype</i> , RC = <i>rc</i> SMTP treats this return code as if it were a return code of 0.

SMTP Virtual Machine Interfaces

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 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.

VERFY

The SMTP 'VERFY' 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 386.

REXX Parameter List

Inputs:

Table 78. SMTP Commands REXX Exit Parameter List

Argument	Description
ARG(1)	Parameter list defined as follows: <ul style="list-style-type: none"> • 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)	MAIL FROM: string
ARG(5)	Client domain name
ARG(6)	Batch VM user ID

SMTP Virtual Machine Interfaces

Outputs: The following are returned to the caller in the RESULT variable via a REXX RETURN statement:

RC	Return String
----	---------------

Argument Description

RC The exit return code; this must be a 4-byte numeric value.

Return String An exit-specified string; the returned value must have a length less than or equal to the maximum length passed to the exit.

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 79. SMTP Commands ASSEMBLER Exit Parameter List

Offset in Decimal	Len	In/Out	Type	Description
+0	4	Input	Char	Exit type
+4	4	Input	Int	Version number
+8	4	Input	Int	Mail record ID
+12	4	Input	Int	Port number of SMTP server
+16	4	Input	Int	IP address of SMTP server
+20	4	Input	Int	Port number of client
+24	4	Input	Int	IP address of client
+28	4	Input	Ptr	Address of SMTP command string
+32	4	Input	Int	Length of SMTP command string
+36	4	Input	Ptr	Address of HELO/EHLO name
+40	4	Input	Int	Length of HELO/EHLO name
+44	4	Input	Ptr	Address of client domain name
+48	4	Input	Int	Length of client domain name
+52	4	Input	Ptr	Address of MAIL FROM: string
+56	4	Input	Int	Length of MAIL FROM: string
+60	8	Input	Char	File name of note on disk
+68	4	Input	Int	Verify client status
+72	4	Output	Ptr	Address of Return String
+76	4	Output	Int	Length of Return String
+80	4	Input	Int	Maximum length of Return String
+84	8	Input	Char	Batch VM User ID
+92	4	Input/ Output	Char	User Word 1
+96	4	Input/ Output	Char	User Word 2
+100	4	Output	Int	Return code from exit

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 (.), 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.

SMTP Virtual Machine Interfaces

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 sever 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 80. SMTP Command Exit Return Codes

Return Code	Explanation
0	Accept command, and continue normal processing.

Table 80. SMTP Command Exit Return Codes (continued)

Return Code	Explanation
1	<p>Reject mail with: 551 <i>Return String</i></p> <p>If a return string is not provided by the exit, the following default message will be used:</p> <p>550 Command Rejected</p> <p>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.</p>
2	<p>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.</p>
4	<p>Disable the exit. The following message will be displayed on the SMTP console:</p> <p>SMTPCMDX EXIT function disabled</p> <p>The exit will no longer be called. The command will be attempted, and processing will continue.</p>
5	<p>Reject this command with: 550 <i>Return String</i></p> <p>If a return string is not provided by the exit, then the default message will be displayed:</p> <p>550 Access denied</p> <p>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.</p>
Other	<p>Any return code other than the above causes SMTP to issue this message:</p> <p>Unexpected return from user exit <i>exitname</i> <i>exittype</i>, RC = <i>rc</i></p> <p>SMTP treats this return code as if it were a return code of 0.</p>

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.

SMTP Virtual Machine Interfaces

The assembler exit will have better performance characteristics than the REXX exit. For best performance, EXECLOAD any REXX exits.

Chapter 11. 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.

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 Session Connection Exit

Telnet Exit Parameter List

Table 81. Telnet Session Connection Exit Parameter List

Offset	Len	Name	In/Out	Description
+0	1	SCREASON	Input	Reason Exit was called X'01' - Client connect
+1	1	SCFLAG1	Output	Flags 1... - Hide VM logo from client .1.. - Hide command simulation ..xx xxxx - Reserved
+2	2	Reserved		
+4	4	SCIPADDR	Input	IP address of client
+8	2	SCPORT	Input	Telnet server port number
+10	2	SCCMDL	Input	Length of command buffer
			Output	Length of command placed in buffer
+12	4	SCPCMD	Input	Address of command buffer
+16	4	SCRC	Output	Return code 0 = Give client VM logo 4 = Reject client, no message 8 = Perform command in SCPCMD; SCCMDL must be non-zero 12 = Same as 0, and disable exit 16 = Same as 4, and disable exit 20 = Same as 8, and disable exit All others will reject the client, and a message is displayed on the TCPIP virtual machine console.
+20	2	SCFPORT	Input	Client foreign port number
+22	2	Reserved		
+24	16	SCLUNAME	Input	Client-provided LU name
+40	4	SCLIPADD	Input	Local IP address to which client connected

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 VMFLASM 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:

```
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 82. Telnet Exit Parameter List

Offset	Len	Name	In/Out	Description
+0	1	PMXVERSN	Input	Parameter list version number X'01' - Version 1
+1	1	PMXREASN	Input	Reason exit called X'00' - Printer connected X'01' - Printer disconnected
+2	2	Reserved		
+4	4	PMXIPADD	Input	Client IP address
+8	2	PMXFPORT	Input	Client port number
+A	2	PMXLPORT	Input	Telnet server port number
+C	4	PMXLDEV	Input	Logical device number
+10	8	PMXLUNAM	Input	Logical unit name specified by client
+18	8	PMXUSER	Input	Associated user identifier. If no matching TN3270E configuration statement exists, contains “?”
+20	4	PMXVDEV	Input	Associated virtual device address. If no matching TN3270E configuration statement exists, contains “?”
+24	4	PMXRC	Output	Return code 0 = Accept client 4 = Reject client 8 = Same as 0, and disable exit 12 = Same as 4, and disable exit

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 TCPMAINT 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 TCPMAINT 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 TCPMAINT 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 TCPMAINT 198 disk.

Enable the sample exit by including the following in PROFILE TCPIP:

Telnet Printer Management Exit

```
InternalClientParms  
  TN3270EExit PMEXIT  
EndInternalClientParms
```

Chapter 12. 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.

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:
 - 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

FTP Server Exit

- 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 83. FTP Exit Audit Parameter List

Offset in Decimal	Len	In/Out	Type	Description
+0	4	Input	Char	Exit type (AUDX)
+4	4	Input	Int	Version number
+8	8	Input	Char	FTP server command
+16	4	Input	Ptr	Address of command argument string; the first halfword contains the length.
+20	8	Input	Char	Login user ID
+28	8	Input	Char	LOGONBY user ID
+36	4	Input	Int	IP address of client
+40	4	Input	Ptr	Address of current working directory name; the first halfword contains the length.
+44	4	Input	Ptr	Address of target directory or file; the first halfword contains the length.
+48	4	Input	Int	Number of bytes transferred.
+52	2	Input	Int	Port number of FTP server
+54	2	Input	Int	Port number of FTP client
+56	8	Input	Char	Event date (yyyymmdd)
+64	8	Input	Char	Event time (hh:mm:ss)
+72	8			Not used
+80	4	Output	Int	Return code from exit

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:

Directory Type	Working Directory passed to FTPEXIT
Minidisk	DSK TERI.191
Shared File System	SFS SERVK1:TERI.
Byte File System	BFS ../VMBFS:BFS:TERI/
Virtual Reader	RDR TERI.RDR

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.

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*.

FTP Server Exit

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

Return Code	Use / Description
0	Continue processing.
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:

ABOR	ACCT	ALLO	APPE	CDUP	CWD
DELE	HELP	LIST	MKD	MODE	NLST
NOOP	PASS	PASV	PORT	PWD	QUIT
REIN	REST	RETR	RMD	RNFR	RNTO
SITE	SYST	STAT	STOR	STOU	STRU
TYPE	USER	XCWD	XMKD	XPWD	UNKNOWN

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 84. FTP Exit Parameter List

Offset in Decimal	Len	In/Out	Type	Description
+0	4	Input	Char	Exit type (CMDX)
+4	4	Input	Int	Version number
+8	8	Input	Char	FTP server command
+16	4	Input	Ptr	Address of command argument string; the first halfword contains the length.

Table 84. FTP Exit Parameter List (continued)

Offset in Decimal	Len	In/Out	Type	Description
+20	8	Input	Char	Login user ID
+28	8	Input	Char	LOGONBY user ID
+36	4	Input	Int	IP address of client
+40	4	Input	Ptr	Address of current working directory name; The first halfword contains the length.
+44	8	-	-	Not used
+52	2	Input	Int	Port number of FTP server
+54	2	Input	Int	Port number of FTP client
+56	16	-	-	Not used
+72	4	Input	Int	Maximum length of return string
+76	4	Output	Int	Address of return string (message text)
+80	4	Output	Int	Return code from exit

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.

Current working directory name

This field contains the type of directory, followed by the working directory name. For example:

Directory Type	Working Directory passed to FTPEXIT
Minidisk	DSK TERI.191
Shared File System	SFS SERVK1:TERI.
Byte File System	BFS ../VMBFS:BFS:TERI/

FTP Server Exit

Directory Type	Working Directory passed to FTPEXIT
Virtual reader	RDR TERI.RDR
No directory defined	-

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:

Exit Type	: AUDX
FTP command	: XFER
Client IP Address	: 9.111.32.29
UserID	: TERI
ByUserID	: -
Bytes transferred	: 2141
Server Port	: 1021
Client Port	: 21400
Event Date	: 19990309
Event Time	: 14:44:34
Working Directory	: SFS SERVK1:TERI.
Command args	: RECEIVING
Target File	: PROFILE.EXEC

Return Codes from General Command Processing

Return Code	Use / Description
0	Accept command and continue processing
4	Reject client command with "502 <i>return_string</i> ". If <i>return_string</i> is not provided, return the default message "502 command rejected".
8	Accept command, continue normal processing, but disable command exit processing. The following message is displayed on the FTP server console: "FTP CMDX exit has been disabled".
12	Same as 4 and disable command exit processing. The following message is displayed on the FTP server console: "FTP CMDX exit has been disabled".

Return Code	Use / Description
Other	Any return code other than the above causes FTP to issue the message: "Unexpected return from user exit FTPEXIT CMDX, RC = rc". The server will treat this return code as if it were a return code of 0.

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 85. FTP Exit Parameter List

Offset in Decimal	Len	In/Out	Type	Description
+0	4	Input	Char	Exit type (DIRX)
+4	4	Input	Int	Version number
+8	8	Input	Char	FTP server command
+16	4	Input	Ptr	Address of command argument string; the first halfword contains the length.
+20	8	Input	Char	Login user ID
+28	8	Input	Char	LOGONBY user ID
+36	4	Input	Int	IP address of client
+40	4	Input	Ptr	Address of current working directory name; the first halfword contains the length.
+44	4	Input	Ptr	Address of target directory or file; the first halfword contains the length.
+48	4	-	-	Not used
+52	2	Input	Int	Port number of FTP server
+54	2	Input	Int	Port number of FTP client
+56	16	-	-	Not used
+72	4	Input	Int	Maximum length of return string
+76	4	Output	Int	Address of return string (message text)
+80	4	Output	Int	Return code from exit

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:

Directory Type	Working Directory Passed to FTPEXIT
Minidisk	DSK TERI.191
Shared File System	SFS SERVK1:TERI.
Byte File System	BFS ../VMBFS:BFS:TERI/
Virtual reader	RDR TERI.RDR
No directory defined	-

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
Working Directory     : BFS ../VMBFS:BFS:SCOTT/
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

Return Code	Use / Description
0	Accept command and continue normal processing
4	Reject client command with "502 <i>return_string</i> ". If <i>return_string</i> is not provided, display the default message "502 command rejected".
8	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".
12	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".
Other	Any return code other than the above causes FTP to issue the message: "Unexpected return from user exit FTPEXIT DIRX, RC = <i>rc</i> ". The server will treat this return code as if it were a return code of 0.

Appendix A. 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 124) 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 86.

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 86.

Table 86. Pascal Language Return Codes

Return Code	Numeric Value	Message Text
OK	0	OK.
ABNORMALcondition ¹	-1	Abnormal condition during inter-address communication. (VMCF. RC= <i>nn</i> User= <i>xxxxxxx</i>)
ALREADYclosing	-2	Connection already closing.
BADlengthARGUMENT	-3	Invalid length specified.
CANNOTsendDATA ²	-4	Cannot send data.
CLIENTrestart	-5	Client reinitialized TCP/IP service.
CONNECTIONalreadyEXISTS	-6	Connection already exists.
DESTINATIONunreachable	-7	Destination address is unreachable.
ERRORinPROFILE	-8	Error in profile file; details are in PROFILE.TCPEERROR.
FATALerror ³	-9	Fatal inter-address communications error. (VMCF. RC= <i>nn</i> User= <i>xxxxxxx</i>)
HASnoPASSWORD	-10	No password in RACF [®] directory.
INCORRECTpassword	-11	TCPIP not authorized to access file.
INVALIDrequest	-12	Invalid request.
INVALIDuserID	-13	Invalid user ID.
INVALIDvirtualADDRESS	-14	Invalid virtual address.
KILLEDbyCLIENT	-15	You aborted the connection.
LOCALportNOTavailable	-16	The requested local port is not available.
MINIDISKinUSE	-17	File is in use by someone else and cannot be accessed.
MINIDISKnotAVAILABLE	-18	File not available.

Pascal Return Codes

Table 86. Pascal Language Return Codes (continued)

Return Code	Numeric Value	Message Text
NObufferSPACE ⁴	-19	No more space for data currently available.
NOMoreINCOMINGdata	-20	The foreign host has closed this connection.
NONlocalADDRESS	-21	The internet address is not local to this host.
NOoutstandingNOTIFICATIONS	-22	No outstanding notifications.
NOsuchCONNECTION	-23	No such connection.
NOtcpIPservice	-24	No TCP/IP service available.
NOTyetBEGUN	-25	Not yet begun TCP/IP service.
NOTyetOPEN	-26	The connection is not yet open.
OPENrejected	-27	Foreign host rejected the open attempt.
PARAMlocalADDRESS	-28	TcpOpen error: invalid local address.
PARAMstate	-29	TcpOpen error: invalid initial state.
PARAMtimeout	-30	Invalid time-out parameter.
PARAMunspecADDRESS	-31	TcpOpen error: unspecified foreign address in active open.
PARAMunspecPORT	-32	TcpOpen error: unspecified foreign port in active open.
PROFILEnotFOUND	-33	TCPIP cannot read profile file.
RECEIVEstillPENDING	-34	Receive still pending on this connection.
REMOTEclose	-35	Foreign host unexpectedly closed the connection.
REMOtereset	-36	Foreign host aborted the connection.
SOFTWAREerror	-37	Software error in TCP/IP!
TCPipSHUTDOWN	-38	TCP/IP service is being shut down.
TIMEOUTconnection	-39	Foreign host is no longer responding.
TIMEOUTopen	-40	Foreign host did not respond within OPEN time-out
TOOmanyOPENS	-41	Too many open connections already exist.
UNAUTHORIZEDuser	-43	You are not authorized to issue this command.
UNEXPECTEDsyn	-44	Foreign host violated the connection protocol.
UNIMPLEMENTEDrequest	-45	Unimplemented TCP/IP request.
UNKNOWNhost	-46	Destination host is not known.

Table 86. Pascal Language Return Codes (continued)

Return Code	Numeric Value	Message Text
UNREACHABLEnetwork	-47	Destination network is unreachable.
UNSPECIFIEDconnection	-48	Unspecified connection.
VIRTUALmemoryTOOsmall	-49	Client virtual machine has too little storage.
WRONGsecORprc	-50	Foreign host disagreed on security or precedence.
YOURend	-55	Client has ended TCP/IP service.
Oresources	-56	TCP cannot handle any more connections now.
UDPlocalADDRESS	-57	Invalid local address for UDP.
UDPunspecADDRESS	-59	Address unspecified when specification necessary.
UDPunspecPORT	-60	Port unspecified when specification necessary.
UDPzeroRESOURCES	-61	UDP cannot handle any more traffic.
FSENDstillPENDING	-62	FSend still pending on this connection.
DROPPEDbyOPERATOR	-79	Connection dropped by operator.
ERRORopeningORreadingFILE	-80	Error opening or reading file.
FILEformatINVALID	-81	File format invalid.

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.

Explanatory Notes

Appendix B. C API System Return Codes

This appendix provides a reference for system calls. Table 87 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 87. System Return Codes

Message	Code	Description
EPERM	1	Permission denied.
ENOENT	2	No such file or directory.
ESRCH	3	No such process.
EINTR	4	Interrupted system call.
EIO	5	I/O error.
ENXIO	6	No such device or address.
E2BIG	7	Argument list too long.
ENOEXEC	8	Exec format error.
EBADF	9	Bad file number.
ECHILD	10	No children.
EAGAIN	11	No more processes.
ENOMEM	12	Not enough memory.
EACCES	13	Permission denied.
EFAULT	14	Bad address.
ENOTBLK	15	Block device required.
EBUSY	16	Device busy.
EEXIST	17	File exists.
EXDEV	18	Cross device link.
ENODEV	19	No such device.
ENOTDIR	20	Not a directory.
EISDIR	21	Is a directory.
EINVAL	22	Invalid argument.
ENFILE	23	File table overflow.
EMFILE	24	Too many open files.
ENOTTY	25	Inappropriate device call.
ETXTBSY	26	Text file busy.
EFBIG	27	File too large.
ENOSPC	28	No space left on device.
ESPIPE	29	Illegal seek.
EROFS	30	Read only file system.
EMLINK	31	Too many links.
EPIPE	32	Broken pipe.
EDOM	33	Argument too large.

C API System Return Codes

Table 87. System Return Codes (continued)

Message	Code	Description
ERANGE	34	Result too large.
EWouldBlock	35	Operation would block.
EINPROGRESS	36	Operation now in progress.
EALREADY	37	Operation already in progress.
ENOTSOCK	38	Socket operation on non-socket.
EDESTADDRREQ	39	Destination address required.
EMSGSIZE	40	Message too long.
EPROTOTYPE	41	Protocol wrong type for socket.
ENOPROTOOPT	42	Protocol not available.
EPROTONOSUPPORT	43	Protocol not supported.
ESOCKTNOSUPPORT	44	Socket type not supported.
EOPNOTSUPP	45	Operation not supported on socket.
EPFNOSUPPORT	46	Protocol family not supported.
EAFNOSUPPORT	47	Address family not supported by protocol family.
EADDRINUSE	48	Address already in use.
EADDRNOTAVAIL	49	Cannot assign requested address.
ENETDOWN	50	Network is down.
ENETUNREACH	51	Network is unreachable.
ENETRESET	52	Network dropped connection on reset.
ECONNABORTED	53	Software caused connection abort.
ECONNRESET	54	Connection reset by peer.
ENOBUFS	55	No buffer space available.
EISCONN	56	Socket is already connected.
ENOTCONN	57	Socket is not connected.
ESHUTDOWN	58	Cannot send after socket shutdown.
ETOOMANYREFS	59	Too many references: cannot splice.
ETIMEDOUT	60	Connection timed out.
ECONNREFUSED	61	Connection refused.
ELOOP	62	Too many levels of symbolic loops.
ENAMETOOLONG	63	File name too long.
EHOSTDOWN	64	Host is down.
EHOSTUNREACH	65	No route to host.
ENOTEMPTY	66	Directory not empty.
EPROCLIM	67	Too many processes.
EUSERS	68	Too many users.
EDQUOT	69	Disc quota exceeded.

Table 87. System Return Codes (continued)

Message	Code	Description
ESTALE	70	Stale NFS file handle.
EREMOTE	71	Too many levels of remote in path.
ENOSTR	72	Device is not a stream.
ETIME	73	Timer expired.
ENOSR	74	Out of streams resources.
ENOMSG	75	No message of desired type.
EBADMSG	76	Trying to read unreadable message.
EIDRM	77	Identifier removed.
EDEADLK	78	Deadlock condition.
ENOLCK	79	No record locks available.
ENONET	80	Machine is not on the network.
ERREMOTE	81	Object is remote.
ENOLINK	82	Link has been severed.
EADV	83	Advertise error.
ESRMNT	84	Srmount error.
ECOMM	85	Communication error on send.
EPROTO	86	Protocol error.
EMULTIHOP	87	Multihop attempted.
EDOTDOT	88	Cross mount point.
EREMCHG	89	Remote address changed.

C API System Return Codes

Appendix C. 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 88 lists the well-known port assignments for TCP.

Table 88. TCP Well-Known Port Assignments

Port Number	Keyword	Reserved for	Services Description
0		reserved	
5	rje	remote job entry	remote job entry
7	echo	echo	echo
9	discard	discard	sink null
11	systat	active users	active users
13	daytime	daytime	daytime
15	netstat	Netstat	who is up or Netstat
19	chargen	ttytst source	character generator
21	ftp	FTP	File Transfer Protocol
23	telnet	Telnet	Telnet
25	smtp	mail	Simple Mail Transfer Protocol
37	time	timeserver	timeserver
39	rlp	resource	Resource Location Protocol
42	nameserver	name	host name server
43	nicname	who is	who is
53	domain	name server	domain name server
57	mtp	private terminal access	private terminal access
69	tftp	TFTP	Trivial File Transfer Protocol
77	rje	netrjs	any private RJE service
79	finger	finger	finger
87	link	ttylink	any private terminal link
95	supdup	supdup	SUPDUP Protocol
101	hostname	hostname	nic hostname server, usually from SRI-NIC
109	pop	postoffice	Post Office Protocol
111	sunrpc	sunrpc	Sun remote procedure call
113	auth	authentication	authentication service
115	sftp	sftp	Simple File Transfer Protocol
117	uucp-path	UUCP path service	UUCP path service
119	untp	readnews untp	USENET News Transfer Protocol

Well-Known Port Assignments

Table 88. TCP Well-Known Port Assignments (continued)

Port Number	Keyword	Reserved for	Services Description
123	ntp	NTP	Network Time Protocol
160–223		reserved	
512	REXEC	REXEC	Remote Execution Protocol
514	RSH	RSHELL	Remote Shell Service
712	vexec	vice-exec	Andrew File System authenticated service
713	vlogin	vice-login	Andrew File System authenticated service
714	vshell	vice-shell	Andrew File System authenticated service
2001	filesrv		Andrew File System service
2106	venus.its		Andrew File System service, for the Venus process

UDP Well-Known Port Assignments

Table 89 lists the well-known port assignments for UDP.

Table 89. UDP Well-Known Port Assignments

Port Number	Keyword	Reserved for	Services Description
0		reserved	
5	rje	remote job entry	remote job entry
7	echo	echo	echo
9	discard	discard	sink null
11	users	active users	active users
13	daytime	daytime	daytime
15	netstat	Netstat	Netstat
19	chargen	ttytst source	character generator
37	time	timeserver	timeserver
39	rlp	resource	Resource Location Protocol
42	nameserver	name	host name server
43	nicname	who is	who is
53	domain	nameserver	domain name server
67	bootpd	BOOTP	BOOTP Daemon
69	tftp	TFTP	Trivial File Transfer Protocol
75			any private dial out service
77	rje	netrjs	any private RJE service
79	finger	finger	finger
111	sunrpc	sunrpc	Sun remote procedure call
123	ntp	NTP	Network Time Protocol
160–223		reserved	
531	rxd-control		rxd control port
2001	rauth2		Andrew File System service, for the Venus process

Well-Known Port Assignments

Table 89. UDP Well-Known Port Assignments (continued)

Port Number	Keyword	Reserved for	Services Description
2002	rfilebulk		Andrew File System service, for the Venus process
2003	rfilesrv		Andrew File System service, for the Venus process
2018	console		Andrew File System service
2115	ropcons		Andrew File System service, for the Venus process
2131	rupdsrv		assigned in pairs; bulk must be srv +1
2132	rupdbulk;		assigned in pairs; bulk must be srv +1
2133	rupdsrv1		assigned in pairs; bulk must be srv +1
2134	rupdbulk1;		assigned in pairs; bulk must be srv +1

Well-Known Port Assignments

Appendix D. 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 VM are based on the following RFCs:

RFC	Title	Author
768	<i>User Datagram Protocol</i>	J.B. Postel
791	<i>Internet Protocol</i>	J.B. Postel
792	<i>Internet Control Message Protocol</i>	J.B. Postel
793	<i>Transmission Control Protocol</i>	J.B. Postel
821	<i>Simple Mail Transfer Protocol</i>	J.B. Postel
822	<i>Standard for the Format of ARPA Internet Text Messages</i>	D. Crocker
823	<i>DARPA Internet Gateway</i>	R.M. Hinden, A. Sheltzer
826	<i>Ethernet Address Resolution Protocol: or Converting Network Protocol Addresses to 48.Bit Ethernet Address for Transmission on Ethernet Hardware</i>	D.C. Plummer
854	<i>Telnet Protocol Specification</i>	J.B. Postel, J.K. Reynolds
856	<i>Telnet Binary Transmission</i>	J.B. Postel, J.K. Reynolds
857	<i>Telnet Echo Option</i>	J.B. Postel, J.K. Reynolds
877	<i>Standard for the Transmission of IP Datagrams over Public Data Networks</i>	J.T. Korb
885	<i>Telnet End of Record Option</i>	J.B. Postel
903	<i>Reverse Address Resolution Protocol</i>	R. Finlayson, T. Mann, J.C. Mogul, M. Theimer
904	<i>Exterior Gateway Protocol Formal Specification</i>	D.L. Mills
919	<i>Broadcasting Internet Datagrams</i>	J.C. Mogul
922	<i>Broadcasting Internet Datagrams in the Presence of Subnets</i>	J.C. Mogul
950	<i>Internet Standard Subnetting Procedure</i>	J.C. Mogul, J.B. Postel
952	<i>DoD Internet Host Table Specification</i>	K. Harrenstien, M.K. Stahl, E.J. Feinler
959	<i>File Transfer Protocol</i>	J.B. Postel, J.K. Reynolds
974	<i>Mail Routing and the Domain Name System</i>	C. Partridge
1009	<i>Requirements for Internet Gateways</i>	R.T. Braden, J.B. Postel
1013	<i>X Window System Protocol, Version 11: Alpha Update</i>	R.W. Scheifler
1014	<i>XDR: External Data Representation Standard</i>	Sun Microsystems Incorporated
1027	<i>Using ARP to Implement Transparent Subnet Gateways</i>	S. Carl-Mitchell, J.S. Quarterman
1032	<i>Domain Administrators Guide</i>	M.K. Stahl

RFCs

RFC	Title	Author
1033	<i>Domain Administrators Operations Guide</i>	M. Lottor
1034	<i>Domain Names—Concepts and Facilities</i>	P.V. Mockapetris
1035	<i>Domain Names—Implementation and Specification</i>	P.V. Mockapetris
1042	<i>Standard for the Transmission of IP Datagrams over IEEE 802 Networks</i>	J.B. Postel, J.K. Reynolds
1044	<i>Internet Protocol on Network System's HYPERchannel: Protocol Specification</i>	K. Hardwick, J. Lekashman
1055	<i>Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP</i>	J.L. Romkey
1057	<i>RPC: Remote Procedure Call Protocol Version 2 Specification</i>	Sun Microsystems Incorporated
1058	<i>Routing Information Protocol</i>	C.L. Hedrick
1091	<i>Telnet Terminal-Type Option</i>	J. VanBokkelen
1094	<i>NFS: Network File System Protocol Specification</i>	Sun Microsystems Incorporated
1112	<i>Host Extensions for IP Multicasting</i>	S. Deering
1118	<i>Hitchhikers Guide to the Internet</i>	E. Krol
1122	<i>Requirements for Internet Hosts-Communication Layers</i>	R.T. Braden
1123	<i>Requirements for Internet Hosts-Application and Support</i>	R.T. Braden
1155	<i>Structure and Identification of Management Information for TCP/IP-Based Internets</i>	M.T. Rose, K. McCloghrie
1156	<i>Management Information Base for Network Management of TCP/IP-based Internets</i>	K. McCloghrie, M.T. Rose
1157	<i>Simple Network Management Protocol (SNMP),</i>	J.D. Case, M. Fedor, M.L. Schoffstall, C. Davin
1179	<i>Line Printer Daemon Protocol</i>	The Wollongong Group, L. McLaughlin III
1180	<i>TCP/IP Tutorial,</i>	T. J. Socolofsky, C.J. Kale
1183	<i>New DNS RR Definitions (Updates RFC 1034, RFC 1035)</i>	C.F. Everhart, L.A. Mamakos, R. Ullmann, P.V. Mockapetris,
1187	<i>Bulk Table Retrieval with the SNMP</i>	M.T. Rose, K. McCloghrie, J.R. Davin
1188	<i>Proposed Standard for the Transmission of IP Datagrams over FDDI Networks</i>	D. Katz
1198	<i>FYI on the X Window System</i>	R.W. Scheifler
1207	<i>FYI on Questions and Answers: Answers to Commonly Asked Experienced Internet User Questions</i>	G.S. Malkin, A.N. Marine, J.K. Reynolds
1208	<i>Glossary of Networking Terms</i>	O.J. Jacobsen, D.C. Lynch
1213	<i>Management Information Base for Network Management of TCP/IP-Based Internets: MIB-II,</i>	K. McCloghrie, M.T. Rose
1215	<i>Convention for Defining Traps for Use with the SNMP</i>	M.T. Rose
1228	<i>SNMP-DPI Simple Network Management Protocol Distributed Program Interface</i>	G.C. Carpenter, B. Wijnen
1229	<i>Extensions to the Generic-Interface MIB</i>	K. McCloghrie
1230	<i>IEEE 802.4 Token Bus MIB IEEE 802 4 Token Bus MIB</i>	K. McCloghrie, R. Fox
1231	<i>IEEE 802.5 Token Ring MIB IEEE 802.5 Token Ring MIB</i>	K. McCloghrie, R. Fox, E. Decker

RFC	Title	Author
1267	<i>A Border Gateway Protocol 3 (BGP-3)</i>	K. Lougheed, Y. Rekhter
1268	<i>Application of the Border Gateway Protocol in the Internet</i>	Y. Rekhter, P. Gross
1269	<i>Definitions of Managed Objects for the Border Gateway Protocol (Version 3)</i>	S. Willis, J. Burruss
1293	<i>Inverse Address Resolution Protocol</i>	T. Bradley, C. Brown
1270	<i>SNMP Communications Services</i>	F. Kastenholz, ed.
1323	<i>TCP Extensions for High Performance</i>	V. Jacobson, R. Braden, D. Borman
1325	<i>FYI on Questions and Answers: Answers to Commonly Asked New Internet User Questions</i>	G.S. Malkin, A.N. Marine
1350	<i>TFTP Protocol</i>	K.R. Sollins
1351	<i>SNMP Administrative Model</i>	J. Davin, J. Galvin, K. McCloghrie
1352	<i>SNMP Security Protocols</i>	J. Galvin, K. McCloghrie, J. Davin
1353	<i>Definitions of Managed Objects for Administration of SNMP Parties</i>	K. McCloghrie, J. Davin, J. Galvin
1354	<i>IP Forwarding Table MIB</i>	F. Baker
1356	<i>Multiprotocol Interconnect on X.25 and ISDN in the Packet Mode</i>	A. Malis, D. Robinson, R. Ullmann
1374	<i>IP and ARP on HIPPI</i>	J. Renwick, A. Nicholson
1381	<i>SNMP MIB Extension for X.25 LAPB</i>	D. Throop, F. Baker
1382	<i>SNMP MIB Extension for the X.25 Packet Layer</i>	D. Throop
1387	<i>RIP Version 2 Protocol Analysis</i>	G. Malkin
1389	<i>RIP Version 2 MIB Extension</i>	G. Malkin
1390	<i>Transmission of IP and ARP over FDDI Networks</i>	D. Katz
1393	<i>Traceroute Using an IP Option</i>	G. Malkin
1397	<i>Default Route Advertisement In BGP2 And BGP3 Versions of the Border Gateway Protocol</i>	D. Haskin
1398	<i>Definitions of Managed Objects for the Ethernet-like Interface Types</i>	F. Kastenholz
1440	<i>SIFT/UFT:Sender-Initiated/Unsolicited File Transfer</i>	R. Troth
1483	<i>Multiprotocol Encapsulation over ATM Adaptation Layer 5</i>	J. Heinanen
1540	<i>IAB Official Protocol Standards</i>	J.B. Postel
1583	<i>OSPF Version 2</i>	J.Moy
1647	<i>TN3270 Enhancements</i>	B. Kelly
1700	<i>Assigned Numbers</i>	J.K. Reynolds, J.B. Postel
1723	<i>RIP Version 2 — Carrying Additional Information</i>	G. Malkin
1813	<i>NFS Version 3 Protocol Specification</i>	B. Callaghan, B. Pawlowski, P. Stauback, Sun Microsystems Incorporated
2225	<i>Classical IP and ARP over ATM</i>	M. Laubach, J. Halpern

These documents can be obtained from:

RFCs

Government Systems, Inc.
Attn: Network Information Center
14200 Park Meadow Drive
Suite 200
Chantilly, VA 22021

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:RFC*nnnn*.TXT
RFC:RFC*nnnn*.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 E. 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

HIPPI	High Performance Parallel Interface
HPFS	High Performance File System
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronic Engineers
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
ILANS	IBM Token-Ring LAN Subsystem
IP	Internet Protocol
IPL	Initial Program Load
ISA	Industry Standard Adapter
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
IUCV	Inter-User Communication Vehicle
JES	Job Entry Subsystem
JIS	Japanese Institute of Standards
JCL	Job Control Language
LAN	Local Area Network
LAPS	LAN Adapter Protocol Support
LCS	IBM LAN Channel Station
LPD	Line Printer Daemon
LPQ	Line Printer Query
LPR	Line Printer Client
LPRM	Line Printer Remove
LPRMON	Line Printer Monitor
LU	Logical Unit
MAC	Media Access Control
Mbps	Megabits per second
MBps	Megabytes per second
MCA	Micro Channel [®] Adapter
MIB	Management Information Base
MIH	Missing Interrupt Handler
MILNET	Military Network
MHS	Message Handling System
MTU	Maximum Transmission Unit
MVS	Multiple Virtual Storage
MX	Mail Exchange
NCP	Network Control Program
NCS	Network Computing System
NDIS	Network Driver Interface Specification
NFS	Network File System
NIC	Network Information Center
NLS	National Language Support
NSFNET	National Science Foundation Network
OS/2[®]	Operating System/2 [®]
OSA	Open Systems Adapter
OSF	Open Software Foundation, Inc.
OSI	Open Systems Interconnection
OSIMF/6000	Open Systems Interconnection Messaging and Filing/6000
OV/MVS	OfficeVision/MVS [™]
OV/VM	OfficeVision/VM [™]
PAD	Packet Assembly/Disassembly
PC	Personal Computer
PCA	Parallel Channel Adapter
PDN	Public Data Network
PDU	Protocol Data Units

Abbreviations and Acronyms

PING	Packet Internet Groper
PIOAM	Parallel I/O Access Method
POP	Post Office Protocol
PROFS®	Professional Office Systems
PSCA	Personal System Channel Attach
PSDN	Packet Switching Data Network
PU	Physical Unit
PVM	Passthrough Virtual Machine
RACF	Resource Access Control Facility
RARP	Reverse Address Resolution Protocol
REXEC	Remote Execution
REXX	Restructured Extended Executor Language
RFC	Request For Comments
RIP	Routing Information Protocol
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
RSCS	Remote Spooling Communications Subsystem
SAA	System Application Architecture
SBCS	Single Byte Character Set
SDLC	Synchronous Data Link Control
SFS	Shared File System
SLIP	Serial Line Internet Protocol
SMIL	Structure for Management Information
SMTP	Simple Mail Transfer Protocol
SNA	Systems Network Architecture
SNMP	Simple Network Management Protocol
SOA	Start of Authority
SPOOL	Simultaneous Peripheral Operations Online
SQL	IBM Structured Query Language
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TFTP	Trivial File Transfer Protocol
TSO	Time Sharing Option
TTL	Time-to-Live
UDP	User Datagram Protocol
VGA	Video Graphic Array
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
WAN	Wide Area Network
XDR	eXternal Data Representation

Abbreviations and Acronyms

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

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 to 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,
522 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 document 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 on non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. 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.

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
ESCONN	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
VTAM	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 for VM or MVS 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.

ASYNCR. Asynchronous.

asynchronous (ASYNCR). 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.

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 BOOTPD 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 Telegraphique et Telephonique.

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 MVS, 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.

Comite Consultatif International Telegraphique et Telephonique (CCITT). The International Telegraph and Telephone Consultative Committee. A unit of the International Telecommunications Union (ITU) of the United Nations. CCITT produces technical standards, known as "recommendations," for all internationally controlled aspects of analog and digital communication.

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 VM operating system that manages the real processor's resources and is responsible for simulating System/370s or 390s for individual users.

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 VM//ESA 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 MVS 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 MVS, 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 VM and OS/2.

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. 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 VM and OS/2, a named set of records stored or processed as a unit. Synonymous with *data set* in MVS.

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*.

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 MVS 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.

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 VM or MVS host into an existing TCP/IP HYPERchannel network, or to connect TCP/IP hosts together to create a TCP/IP HYPERchannel network.

I

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*.

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.

intrinsic 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.

L

LaMail. The client that communicates with the OS/2 Presentation Manager to manage mail on the network.

LAN. Local area network.

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.

LLB. Local Location Broker.

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 Location Broker (LLB). In Network Computing System (NCS) Location Broker, a server that maintains information about objects on the local host and provides the Location Broker forwarding facility.

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.

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 MVS/370, the MVS/XA product, and the MVS/ESA product.

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.

NCK.** Network Computing Kernel.

NCP. Network Control Program.

NCS. Network Computing System.

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 Computing Kernel (NCK). In the Network Computing System (NCS), the combination of the remote procedure call runtime library and the Location Broker.

Network Computing System (NCS). A set of software components developed by Apollo, Incorporated, that conform to the Network Computing Architecture (NCA). NCS is made up of two parts: the nidl compiler and Network Computing Kernel (NCK). NCS is a programming tool kit that allows programmers to distribute processing power to other hosts.

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 Interface Definition Language (NIDL). A declarative language for the definition of interfaces that has two forms, a Pascal-like syntax and a C-like syntax. NIDL is a component of the Network Computing Architecture.

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.

NIDL. Network Interface Definition Language.

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.

OfficeVision (OV). IBM's new proprietary, integrated office management system used for sending, receiving, and filing electronic mail, and a variety of other office tasks. OfficeVision replaces PROFS

Offload host. Any device that is handling the TCP/IP processing for the MVS host where TCP/IP for MVS is installed. Currently, the only supported Offload host is the 3172-3.

Offload system. Represents both the MVS host where TCP/IP for MVS 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.

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.

PDN. Public Data Network.

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.

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.

Q

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

RACF. 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.

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.

SNA. Systems Network Architecture.

SNALINK. SNA Network Link.

SNA Network Link. An SNA network link function of TCP/IP for VM and MVS hosts running TCP/IP to communicate through an existing SNA backbone.

SNMP. Simple Network Management Protocol.

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.

SPOOL. Simultaneous peripheral operations online.

spooling. 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.

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 subnetwork.

subnet mask. A mask used in the IP protocol layer to separate the subnet address from the host portion of the address.

subnetwork. 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.

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.

T

TALK. An interactive messaging system that sends messages between the local host and a foreign host.

TCP. Transmission Control Protocol.

TCP/IP. Transmission Control Protocol/Internet 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.

TFTPD. 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 (TFTPD). 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.

Virtual Machine/System Product (VM/SP). An IBM-licensed program that manages the resources of a single computer so that multiple computing systems appear to exist. Each virtual machine is the functional equivalent of a *real* 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 VM and MVS 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.

ZAP disk. The virtual disk in the VM operating system that contains the user-written modifications to VTAM code.

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 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 Base Publications

Evaluation

- *z/VM: Licensed Program Specifications*, GC24-5943
- *z/VM: General Information*, GC24-5944

Installation and Service

- *z/VM: Installation Guide*, GC24-5945
- *z/VM: Service Guide*, GC24-5946
- *z/VM: VMSES/E Introduction and Reference*, GC24-5947

Planning and Administration

- *z/VM: Planning and Administration*, SC24-5948
- *z/VM: CMS File Pool Planning, Administration, and Operation*, SC24-5949
- *z/VM: Migration Guide*, GC24-5928
- *VM/ESA: REXX/EXEC Migration Tool for VM/ESA*, GC24-5752
- *z/VM: Running Guest Operating Systems*, SC24-5950
- *VM/ESA: Connectivity Planning, Administration, and Operation*, SC24-5756
- *z/VM: Group Control System*, SC24-5951
- *z/VM: Performance*, SC24-5952

Customization

- *z/VM: CP Exit Customization*, SC24-5953

Operation

- *z/VM: System Operation*, SC24-5954
- *z/VM: Virtual Machine Operation*, SC24-5955

Application Programming

- *z/VM: CP Programming Services*, SC24-5956
- *z/VM: CMS Application Development Guide*, SC24-5957
- *z/VM: CMS Application Development Guide for Assembler*, SC24-5958
- *z/VM: CMS Callable Services Reference*, SC24-5959
- *z/VM: CMS Macros and Functions Reference*, SC24-5960
- *z/VM: CMS Application Multitasking*, SC24-5961
- *VM/ESA: REXX/VM Primer*, SC24-5598
- *z/VM: REXX/VM User's Guide*, SC24-5962
- *z/VM: REXX/VM Reference*, SC24-5963
- *z/VM: OpenExtensions POSIX Conformance Document*, GC24-5976
- *z/VM: OpenExtensions User's Guide*, SC24-5977
- *z/VM: OpenExtensions Command Reference*, SC24-5978
- *z/VM: OpenExtensions Advanced Application Programming Tools*, SC24-5979
- *z/VM: OpenExtensions Callable Services Reference*, SC24-5980
- *z/VM: Reusable Server Kernel Programmer's Guide and Reference*, SC24-5964
- *z/VM: Enterprise Systems Architecture/Extended Configuration Principles of Operation*, SC24-5965
- *C for VM/ESA: Library Reference*, SC23-3908
- *OS/390: DFSMS Program Management*, SC27-0806
- *z/VM: Program Management Binder for CMS*, SC24-5934
- *Debug Tool User's Guide and Reference*, SC09-2137
- *External Security Interface (RACROUTE) Macro Reference for MVS and VM*, GC28-1366
- *VM/ESA: Programmer's Guide to the Server-Requester Programming Interface for VM*, SC24-5455
- *VM/ESA: CPI Communications User's Guide*, SC24-5595
- *Common Programming Interface Communications Reference*, SC26-4399
- *Common Programming Interface Resource Recovery Reference*, SC31-6821

End Use

- *z/VM: CP Command and Utility Reference*, SC24-5967
- *VM/ESA: CMS Primer*, SC24-5458
- *z/VM: CMS User's Guide*, SC24-5968
- *z/VM: CMS Command Reference*, SC24-5969
- *z/VM: CMS Pipelines User's Guide*, SC24-5970
- *z/VM: CMS Pipelines Reference*, SC24-5971
- *CMS/TSO Pipelines: Author's Edition*, SL26-0018
- *z/VM: XEDIT User's Guide*, SC24-5972
- *z/VM: XEDIT Command and Macro Reference*, SC24-5973
- *z/VM: Quick Reference*, SC24-5986

Diagnosis

- *z/VM: System Messages and Codes*, GC24-5974
- *z/VM: Diagnosis Guide*, GC24-5975
- *z/VM: VM Dump Tool*, GC24-5887
- *z/VM: Dump Viewing Facility*, GC24-5966

Publications for Additional Facilities

DFSMS/VM[®]

- *VM/ESA: DFSMS/VM Function Level 221 Planning Guide*, GC35-0121
- *VM/ESA: DFSMS/VM Function Level 221 Installation and Customization*, SC26-4704
- *VM/ESA: DFSMS/VM Function Level 221 Storage Administration Guide and Reference*, SH35-0111
- *VM/ESA: DFSMS/VM Function Level 221 Removable Media Services User's Guide and Reference*, SC35-0141
- *VM/ESA: DFSMS/VM Function Level 221 Messages and Codes*, SC26-4707
- *VM/ESA: DFSMS/VM Function Level 221 Diagnosis Guide*, LY27-9589

OSA/SF

- *S/390: Planning for the S/390 Open Systems Adapter (OSA-1, OSA-2) Feature*, GC23-3870
- *VM/ESA: Open Systems Adapter Support Facility User's Guide for OSA-2*, SC28-1992
- *S/390: Open Systems Adapter-Express Customer's Guide and Reference*, SA22-7403

Language Environment[®]

- *Language Environment for OS/390 & VM: Concepts Guide*, GC28-1945
- *Language Environment for OS/390 & VM: Migration Guide*, SC28-1944
- *Language Environment for OS/390 & VM: Programming Guide*, SC28-1939
- *Language Environment for OS/390 & VM: Programming Reference*, SC28-1940
- *Language Environment for OS/390 & VM: Writing Interlanguage Communication Applications*, SC28-1943
- *Language Environment for OS/390 & VM: Debugging Guide and Run-Time Messages*, SC28-1942

Publications for Optional Features

CMS Utilities Feature

- *VM/ESA: CMS Utilities Feature*, SC24-5535

TCP/IP Feature for z/VM

- *z/VM: TCP/IP Level 3A0 Planning and Customization*, SC24-5981
- *z/VM: TCP/IP Level 3A0 User's Guide*, SC24-5982
- *z/VM: TCP/IP Level 3A0 Programmer's Reference*, SC24-5983
- *z/VM: TCP/IP Level 3A0 Messages and Codes*, GC24-5984
- *z/VM: TCP/IP Level 3A0 Diagnosis Guide*, GC24-5985

OpenEdition[®] DCE Feature for VM/ESA[®]

- *OpenEdition DCE for VM/ESA: Introducing the OpenEdition Distributed Computing Environment*, SC24-5735
- *OpenEdition DCE for VM/ESA: Planning*, SC24-5737
- *OpenEdition DCE for VM/ESA: Configuring and Getting Started*, SC24-5734
- *OpenEdition DCE for VM/ESA: Administration Guide*, SC24-5730
- *OpenEdition DCE for VM/ESA: Administration Reference*, SC24-5731
- *OpenEdition DCE for VM/ESA: Application Development Guide*, SC24-5732

- *OpenEdition DCE for VM/ESA: Application Development Reference*, SC24-5733
- *OpenEdition DCE for VM/ESA: User's Guide*, SC24-5738
- *OpenEdition DCE for VM/ESA: Messages and Codes*, SC24-5736

LANRES/VM

- *LAN Resource Extension and Services/VM: Licensed Program Specifications*, GC24-5617
- *LAN Resource Extension and Services/VM: General Information*, GC24-5618
- *LAN Resource Extension and Services/VM: Guide and Reference*, SC24-5622

- "Network Management of TCP/IP Networks: Present and Future," A. Ben-Artzi, A. Chandna, V. Warriar.
- "Special Issue: Network Management and Network Security," *ConneXions-The Interoperability Report*, Volume 4, No. 8, August 1990.
- *The Art of Distributed Application: Programming Techniques for Remote Procedure Calls*, John R. Corbin, Springer-Verlog, 1991.
- *The Simple Book: An Introduction to Management of TCP/IP-based Internets*, Marshall T Rose, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

CD-ROM

The following CD-ROM contains all the IBM libraries that are available in IBM BookManager® format for current VM system products and current IBM licensed programs that run on VM. It also contains PDF versions of z/VM publications and publications for some related IBM licensed programs.

- *Online Library Omnibus Edition: VM Collection*, SK2T-2067

Note: Only unlicensed publications are included.

Other TCP/IP Related Publications

This section lists other publications, outside the z/VM 3.1.0 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
- "MIB II Extends SNMP Interoperability," C. Vanderberg, *Data Communications*, October 1990.
- "Network Management and the Design of SNMP," J.D. Case, J.R. Davin, M.S. Fedor, M.L. Schoffstall.

Index

A

- abbreviations and acronyms 421
- accept() 22
- ACCEPT (IUCV) 177
- address, socket 7
- address families, socket 6
- address information file 18
- Addressing within an internet domain 7
- Addressing within an IUCV domain 8
- AddUserNote 110
- administration server 306
- AF_INET address family 6
- AF_INET domain example 26
- AF_IUCV address family 6
- AF_IUCV domain example 27
- Aliases information file 18
- APITYPE=3 (multiple request) 174
- applications program interface (API)
 - IUCV sockets API 171
- ASCII to EBCDIC translation tables 18
- associate table functions 293
- asynchronous communication, sequence (Pascal API) 95
- auth_destroy() 207
 - authenticators 304
 - communicating 304
 - name structures 303
 - tickets 304
- authentication server 303, 305
- authnone_create() 208
- authunix_create() 208
- authunix_create_default() 208

B

- BeginTcpIp (Pascal) 110
- Berkeley socket implementation 17
- big endian byte ordering convention 7
- bind() 23
- BIND (IUCV) 178
- BUFFERspaceAVAILABLE (VMCF) 164
- byte order, network 7

C

- C socket application programming interface 5
- C socket calls
 - getibmssockopt()
 - call example 37
 - description 36
 - return values 38
 - ibmsflush() 51
 - setibmssockopt()
 - call example 72, 74
 - description 72
 - return values 74
 - structure elements 73
 - sock_debug() 79
 - sock_debug_bulk_perf0()
 - description 80

- C socket calls (*continued*)
 - sock_debug_bulk_perf0() (*continued*)
 - example 80
 - sock_do_bulkmode() 80
 - sock_do_teststor() 81
- C socket program library 89
- C socket programs, examples
 - TCP client 89
 - TCP server 90
 - UDP client 92
 - UDP server 93
- C Socket Quick Reference 19
- callrpc() 209
- ClearTimer 111
- client
 - Kerberos 179, 318, 320
 - remote procedure calls 201
 - SNMP DPI programs 337
- client verification exit, SMTP 373
- clnt_broadcast() 209
- clnt_call() 211
- clnt_destroy() 213
- clnt_freeres() 213
- clnt_geterr() 213
- clnt_pcreateerror() 214
- clnt_perrno() 214
- clnt_perror() 214
- clnt_screateerror() 215
- clnt_sperrno() 215
- clnt_sperror() 216
- clntcp_create() 212
- clntraw_create() 216
- clnttcp_create() 217
- clntudp_create() 217
- close() 27
- command exit, SMTP 384
- compiling and linking
 - general for all APIs 1
 - SNMP DPI 328
 - X Windows 254
- connect() 27
- CONNECT (IUCV) 180
- connection information record (Pascal) 97
- connection states (Pascal) 96
- CONNECTIONclosing (Pascal) 96
- CONNECTIONstateCHANGED (VMCF) 164
- CreateTimer 111
- CREDENTIALS structure 309

D

- DATA 363
- data structures
 - Pascal 96
 - VMCF 147
- DATAdelivered (VMCF) 164
- datagram socket, interface 5
 - TCPIP.DATA 79, 81
- debugging messages 19
- DestroyTimer 111

- directories
 - Kerberos 307
 - remote procedure calls 206
 - sockets 8
- DPI client program 337, 339

E

- EBCDIC to ASCII translation tables 18
- EHLO 361
- EndTcpIp (Pascal) 111
- envelope, SMTP
 - description 359
 - example 369
- environment variables 17
 - HOSTALIASES 18
 - SOCK_DEBUG 19
 - X-ADDR 18
 - X-SITE 18
 - X-XLATE 18
- ETC SERVICES file 413
- exit routines, SMTP 373, 390
- Exits, Server
 - Telnet 391
- EXPN 368
- extension routines (X window system) 292
- eXternal Data Representation protocol, general information 201

F

- FCNTL (IUCV) 180
- fDPIparse() 330
- file specification record (Pascal) 105
- ForeignSocket 98

G

- GET, SNMP DPI request 327
- get_myaddress() 218
- GET-NEXT, SNMP DPI request 327
- GETCLIENTID (IUCV) 181
- getdtablesize() 33
- gethostbyaddr() 33
- gethostbyname() 34
- gethostent() 35
- gethostid() 36
- GETHOSTID (IUCV) 182
- gethostname() 36
- Gethostname (REXX) 182
- GetHostNumber 112
- GetHostResol 112
- GetHostString 113
- GetIBMSockopt 36
- GetIdentity 113
- getnetbyaddr() 38
- getnetbyname() 39
- getnetent() 40
- GetNextNote 113
- getpeername() 40

GETPEERNAME (IUCV) 183
 getprotobyname() 41
 getprotobynumber() 41
 getprotoent() 42
 getservbyname() 43
 getservbyport() 43
 getservent() 44
 GetSmsg 114
 getsockname() 44
 GETSOCKNAME (IUCV) 183
 getsockopt() 45
 GETSOCKOPT (IUCV) 184
 givesocket() 49
 GIVESOCKET (IUCV) 185

H

Handle (Pascal) 114
 handling external interrupts 107
 HELO 360
 HELP 364
 host information file 18
 Host lookup routines 109
 HOSTALIAS environment variable 18
 HOSTS ADDRINFO 18
 HOSTS SITEINFO 18
 htonl() 51
 hton() 51

I

ibmsflush() 51
 inet_addr() 52
 inet_lnaof() 52
 inet_makeaddr() 53
 inet_netof() 53
 inet_network() 53
 inet_ntoa() 54
 initialization procedures, TCP/UDP
 (Pascal) 106
 inter-communication vehicle sockets 169
 interface, C socket 5
 internet addressing 7
 ioctl() 54
 IOCTL (IUCV) 186
 IPUSER variable, returned by socket
 call 173
 IsLocalAddress 115
 IsLocalHost 115
 IUCV, subsystem communication macros
 IUCV CONNECT 171
 IUCV PURGE 175
 IUCV REJECT 175, 200
 IUCV REPLY 175
 IUCV SEND 171
 IUCVMCOM SEVER 174
 IUCV socket API 171
 IUCV socket call, buffer formats
 ACCEPT 178
 BIND 178
 CANCEL 179
 CLOSE 179
 CONNECT 180
 FCNTL 181
 GETCLIENTID 182
 GETHOSTID 182
 GETHOSTNAME 183

IUCV socket call, buffer formats
 (continued)
 GETPEERNAME 183
 GETSOCKNAME 184
 GETSOCKOPT 185
 GIVESOCKET 186
 IOCTL 187
 LASTERRNO 200
 LISTEN 189
 READ 190
 READV 190
 RECV 191
 RECVMMSG 191
 RRCVFROM 191
 SELECT 192
 SELECT and SELECTEX
 DESCRIPTOR_SET macro 191
 descriptor sets 191
 FD_CLR macro 191
 FD_ISSET macro 191
 SELECTEX 192
 SEND 193
 SENDTO 195
 SHUTDOWN 197
 SOCKET 197
 TAKESOCKET 198
 WRITE 199
 WRITEV 199

IUCV socket calls

ACCEPT 177
 BIND 178
 CLOSE 179
 CONNECT 180
 FCNTL 180
 GETCLIENTID 181
 GETHOSTID 182
 GETHOSTNAME 182
 GETPEERNAME 183
 GETSOCKNAME 183
 GETSOCKOPT 184
 GIVESOCKET 185
 IOCTL 186
 LISTEN 188
 MAXDESC 189
 READ 189
 READV 189
 RECV 190
 RRCVFROM 190
 RECVMMSG 190
 SELECT 191
 SELECTEX 191
 SEND 193
 SENDMSG 194
 SENDTO 194
 SETSOCKOPT 195
 SHUTDOWN 196
 SOCKET 197
 TAKESOCKET 198
 WRITE 199
 WRITEV 199

IUCV sockets, general

connect parameters 171
 general information 169
 issuing socket calls 174
 lasterrno special request 200
 multiple-req socket program
 (apitype=3) 171, 174

IUCV sockets, general (continued)
 path severance 173
 response from initial message 172
 response from TCPIP 175
 restrictions 169
 send parameters, initial message 171
 sever, application initiated 173
 sever, clean_up of stream sockets 173
 sever, TCPIP initiated 173
 socket API 171
 socket call syntax 176
 waiting for response from TCPIP 175
 IUCV Sockets, prerequisite
 knowledge 169

K

Kerberos
 administration server 306
 applications library 307
 authentication server
 authenticators 304
 communicating 304
 name structures 303
 tickets 304
 authentication system 303, 325
 database 306
 encryption 305
 Kerberos routines
 krb_get_cred() 309
 krb_kntoln() 309
 krb_mk_err() 310
 krb_mk_priv() 310
 krb_mk_req() 311
 krb_mk_safe() 311
 krb_rd_err() 312
 krb_rd_priv() 313
 krb_rd_req() 314
 krb_rd_safe() 315
 krb_recvauth() 315
 krb_sendauth() 316
 Quick Reference routines 308
 sample programs
 client 318
 server 320
 ticket granting service 305
 user programs 308

L

LDRTBLS, SET command 4
 level parameter on C socket calls
 on getibmssockopt() 37
 on setibmssockopt() 72
 libraries
 Kerberos 306, 307
 remote procedure calls 206
 SNMP DPI 329
 sockets 16
 listen() 56, 188
 LISTENING (Pascal) 96
 little endian byte ordering convention 7

M

mail forwarding exit, SMTP 378
 MAILFROM 361

Management Information Base
 (MIB) 325, 327
 maxdesc() 57
 MAXDESC (IUCV) 189
 messages
 Pascal 104
 system 1
 mkDPIregister() 331
 mkDPIresponse() 331
 mkDPIset() 332
 mkDPItrap() 333
 MonCommand 116
 Monitor procedures 108
 monitor query 117
 MonQuery 117
 MSG_DAT fields 312, 313, 315, 317

N

names
 Kerberos 303, 304
 network, physical 5
 network byte order 7
 big endian byte ordering 7
 htonl() 51
 htons() 51
 little endian byte ordering 7
 ntohl() 58
 ntohs() 59
 NONEXISTENT (Pascal) 96
 NOOP 364
 notification record (Pascal) 98
 notifications
 notifications (Pascal) 105
 notifications, specifying those to
 receive (VMCF) 153
 notifications (VMCF) 163
 NotifyIo 118
 ntohl() 58
 ntohs() 59

O

onoff parameter on C socket calls
 on sock_debug() 80
 on sock_debug_bulk_perf0() 80
 on sock_do_bulkmode() 81
 on sock_do_teststor() 81
 OPEN (Pascal) 96
 OpenAttemptTimeout 97
 optlen parameter on C socket calls
 on getibmssockopt() 37
 on setibmssockopt() 72
 optname parameter on C socket calls
 on getibmssockopt() 37
 on setibmssockopt() 72
 optval parameter on C socket calls
 on getibmssockopt() 37
 on setibmssockopt() 72
 OSF/Motif 251, 296, 297

P

parameter, domain (C sockets) 8
 parameter, protocol (C sockets) 9
 parameter, type (C sockets) 8
 parse 330

Pascal
 API, description 95
 assembler calls
 RTcpExtRupt 123
 RTcpVmcfRupt 123
 asynchronous communication, general
 sequence 95
 Compiler, IBM VS Pascal &
 Library 95
 connection state type
 CONNECTIONclosing 96
 LISTENING 96
 NONEXISTENT 96
 OPEN 96
 RECEIVINGonly 96
 SENDINGonly 97
 TRYINGtoOPEN 97
 data structures 96
 description
 connection information record 97
 connection states 96
 file specification record 105
 notification record 98
 return codes 405, 409
 sample program 143
 software requirements 95
 password 303, 305, 306
 path addresses, SMTP 370
 pDPIpacket() 335
 PING interface 107
 PingRequest 119
 PINGresponse (VMCF) 167
 pmap_getmaps() 219
 pmap_getport() 219
 pmap_rmtcall() 220
 pmap_set() 221
 pmap_unset() 221
 port
 port assignments 204
 unspecified ports 135
 porting
 assessing system return messages 1
 printing system return messages 1
 remote procedure calls 206
 sockets 17
 portmap 204
 Portmapper 203
 procedure calls, Pascal
 descriptions 105
 handling external interrupts
 RTcpExtRupt 123
 RTcpVmcfRupt 123
 TcpExtRupt 128
 TcpVmcfRupt 138
 Host lookup routines
 GetHostNumber 112
 GetHostResol 112
 GetHostString 113
 GetIdentity 113
 IsLocalAddress 115
 IsLocalHost 115
 Monitor procedures
 MonCommand 116
 MonQuery 117
 notifications
 description 105
 GetNextNote 113

procedure calls, Pascal (*continued*)

notifications (*continued*)

 Handle 114

 Unhandle 142

Other routines

 AddUserNote 110

 GetSmsg 114

 ReadXlateTable 122

 SayCalRe 124

 SayConSt 124

 SayIntAd 124

 SayIntNum 125

 SayNotEn 125

 SayPorTy 125

 SayProTy 125

Raw IP interface

 RawIpClose 119

 RawIpOpen 120

 RawIpReceive 120

 RawIpSend 121

TCP communication procedures

 TcpAbort 127

 TcpClose 127

 TcpFReceive, TcpReceive, and

 TcpWaitReceive 128

 TcpFSend, TcpSend, and

 TcpWaitSend 131

 TcpOpen and TcpWaitOpen 134

 TcpOption 136

 TcpStatus 137

TCP/UDP initialization procedures

 BeginTcpIp 110

 StartTcpNotice 126

 TcpNameChange 133

TCP/UDP termination procedure,

 EndTcpIp 106

Timer routines

 ClearTimer 111

 CreateTimer 111

 DestroyTimer 111

 SetTimer 126

UDP communication procedures

 UdpClose 138

 UdpNReceive 139

 UdpOpen 139

 UdpReceive 140

 UdpSend 141

prototypes 17

Q

query_DPI_port() 336

QUEUE 365

quick reference tables

 Kerberos routines 308

 SNMP DPI routines 329

 socket calls 19

 X Windows 256

QUIT 364

R

Raw IP interface 108

raw sockets 6

RawIpClose (Pascal) 119

RawIpOpen (Pascal) 120

RAWIPpacketsDELIVERED (VMCF) 166

RawIpReceive (Pascal) 120
 RawIpSend (Pascal) 121
 RAWIPspaceAVAILABLE (VMCF) 167
 RCPT TO 362
 read() 59
 readv() 60
 ReadXlateTable 122
 RECEIVINgonly (Pascal) 96
 recv() 61
 recvfrom() 62
 recvmsg() 63
 REGISTER, SNMP DPI request 328
 registrerrpc() 222
 related protocols 417
 remote procedure calls (RPCs) 201, 245

- accessing system return messages 206
- auth_destroy() 207
- authnone_create() 208
- authunix_create() 208
- authunix_create_default() 208
- callrpc() 209
- clnt_broadcast() 209
- clnt_call() 211
- clnt_control() 211
- clnt_create() 212
- clnt_destroy() 213
- clnt_freeres() 213
- clnt_geterr() 213
- clnt_pcreateerror() 214
- clnt_perrno() 214
- clnt_perror() 214
- clnt_spccreateerror() 215
- clnt_sperrno() 215
- clnt_sperror() 216
- clntraw_create() 216
- clnttcp_create() 217
- clntudp_create() 217
- enum clnt_stat structure 205
- enumerations 207
- general information 201
- get_myaddress() 218
- getrpcport() 219
- interface 201
- library 206
- pmap_getmaps() 219
- pmap_getport() 219
- pmap_rmtcall() 220
- pmap_set() 221
- pmap_unset() 221
- porting 206
- Portmapper
 - contacting 204
 - target assistance 204
- printing system return messages 207
- registrerrpc() 222
- remapping file names 206
- rpc_createerr 207
- RPCGEN command 204
- svc_destroy() 222
- svc_fds() 207
- svc_freeargs() 223
- svc_getargs() 223
- svc_getcaller() 224
- svc_getreq() 224
- svc_register() 224
- svc_run() 225

remote procedure calls (RPCs) 201, 245
(continued)
 svc_sendreply() 225
 svc_unregister() 226
 svcerr_auth() 226
 svcerr_decode() 226
 svcerr_noproc() 226
 svcerr_noprog() 227
 svcerr_progvers() 227
 svcerr_systemerr() 227
 svcerr_weakauth() 228
 svcraw_create() 228
 svctcp_create() 228
 svcudp_create() 229
 xdr_accepted_reply() 229
 xdr_array() 230
 xdr_authunix_parms() 230
 xdr_bool() 231
 xdr_bytes() 231
 xdr_callhdr() 232
 xdr_callmsg() 232
 xdr_double() 232
 xdr_enum() 233
 xdr_float() 234
 xdr_inline() 234
 xdr_int() 235
 xdr_long() 235
 xdr_opaque() 235
 xdr_opaque_auth() 236
 xdr_pmap() 236
 xdr_pmaplist() 237
 xdr_pointer() 237
 xdr_reference() 237
 xdr_rejected_reply() 238
 xdr_replymsg() 238
 xdr_short() 239
 xdr_string() 239
 xdr_u_int() 239
 xdr_u_long() 240
 xdr_u_short() 240
 xdr_union() 241
 xdr_vector() 241
 xdr_void() 242
 xdr_wrapstring() 242
 xdrmem_create() 243
 xdrrec_create() 243
 xdrrec_endofrecord() 244
 xdrrec_eof() 244
 xdrrec_skiprecord() 244
 xdrstdio_create() 244
 xpirt_register() 245
 xpirt_unregister() 245
 resolver customization 18
 RESOURCEsavailable (VMCF) 167
 return codes

- Pascal 405, 409
- system
 - Accessing 1
 - Printing 1

 rpc_createerr 207
 RPC sample programs

- client 246
- raw data stream 248
- server 246

 RPCGEN command 204
 RSET 364

S

S, defines socket descriptor on C socket call

- on getibmssockopt() 37
- on ibmsflush() 51
- on setibmssockopt() 72

 SayCalRe 124
 SayConSt 124
 SayIntAd 98, 124
 SayIntNum 125
 SayNotEn 125
 SayPorTy 125
 SayProTy 125
 select() 64
 SELECT (IUCV) 191
 selectex() 67
 SELECTEX (IUCV) 191
 send() 68
 SEND (IUCV) 193
 SENDINGonly 97
 sendmsg() 69
 SENDMSG (IUCV) 194
 sendto() 70
 SENDTO (IUCV) 194
 server

- Kerberos 303, 320, 325
- NCS 306
- remote procedure calls 203, 246, 251
- sockets 90, 92

 SERVICES file 413
 SET, SNMP DPI request 327
 SET LDRTBLS command 4
 sethostent() 72
 setibmssockopt() 72
 setnetent() 75
 setprotoent() 75
 setservent() 75
 setsockopt() 76
 SETSOCKOPT (IUCV) 195
 SetTimer 126
 shutdown() 81
 SHUTDOWN (IUCV) 196
 MSG command (VMCF) 114
 SMTP exit routines 373, 390
 SMTP interface

- batch command files, format 371
- batch examples
 - converting to batch format 371
 - querying delivery queues 372
 - sending mail 371
- envelope, description of 369
- path addresses 370
- responses 369
- SMTP commands
 - DATA 363
 - EHLO 361
 - EXPN 368
 - HELO 360
 - HELP 364
 - MAILFROM 361
 - NOOP 364
 - QUEUE 365
 - QUIT 364
 - RCPT TO 362
 - RSET 364
 - TICK 369
 - VERB 368

SMTP interface (*continued*)
 VRFY 367

SMTP transactions 359

SMTPSEND EXEC 372

SNMP agent distributed program
 interface (DPI) 325, 337

SNMP DPI
 agents 325
 compiling and linking 328
 requests
 GET 327
 GET-NEXT 327
 REGISTER 328
 SET 327
 TRAP 328
 routines
 DPIdebug() 329
 fDPIparse() 330
 mkDPIlist() 330
 mkDPIregister() 331
 mkDPIresponse() 331
 mkDPIset() 332
 mkDPItrap() 333
 mkDPItrape() 334
 pDPIpacket() 335
 query_DPI_port() 336
 Quick Reference 329
 SOCK_DGRAM 8
 SOCK_RAW 8
 SOCK_STREAM 8
 software requirements 328
 subagents 325

SO_BULKMODE, on C socket calls. 36

SO_NONBLOCKLOCAL, on C socket
 calls. 36

Sock_debug() 79

Sock_debug_bulk_perf0() 80

SOCK_DEBUG environment variable 19

Sock_do_bulkmode() 80

Sock_do_teststor()Sock_debug_bulk_perf0() 81

socket() 82

SOCKET (IUCV) 197

socket calls
 accept() 10, 22
 bind() 9, 23
 close() 14, 27
 connect() 10, 27
 endhostent() 30
 endnetent() 31
 endprotoent() 31
 endservent() 31
 fcntl() 13, 31
 getclientid() 32
 getdtablesize() 33
 gethostbyaddr() 33
 gethostbyname() 34
 gethostent() 35
 gethostid() 36
 gethostname() 36
 getnetbyaddr() 38
 getnetbyname() 39
 getnetent() 40
 getpeername() 40
 getprotobyname() 41
 getprotobynumber() 41
 getprotoent() 42
 getservbyname() 43

socket calls (*continued*)
 getservbyport() 43
 getservent() 44
 getsockname() 44
 getsockopt() 45
 givesocket() 49
 htonl() 51
 htons() 51
 ibmsflush() 51
 inet_addr() 52
 inet_lnaof() 52
 inet_makeaddr() 53
 inet_netof() 53
 inet_network() 53
 inet_ntoa() 54
 ioctl() 13, 54
 listen() 10, 56
 maxdesc() 57
 ntohl() 58
 ntohs() 59
 read() 11, 59
 readv() 11, 60
 recv() 11, 61
 recvfrom() 11, 62
 recvmsg() 12, 63
 select() 12, 64
 selectex() 67
 send() 11, 68
 sendmsg() 12, 69
 sendto() 11, 70
 sethostent() 72
 setibmssockopt() 72
 setnetent() 75
 setprotoent() 75
 setservent() 75
 setsockopt() 76
 shutdown() 81
 socket() 8, 82
 takesocket() 85
 tcperror() 86
 write() 11, 87
 writev() 11, 88

SOCKET.H header file 7

socket record 98

sockets
 address 7
 address families
 AF_INET 6
 AF_IUCV 6
 Addressing within an internet
 domain 7
 Addressing within an IUCV
 domain 8
 C Socket application program
 interface 5
 connected 11, 16, 26
 definition 5
 domain parameter 8
 example program fragment series 8,
 14
 general information 5
 guidelines for using types of 6
 header files 17
 SOCKET.H 37
 interface
 datagram 5
 raw socket 6

sockets (*continued*)
 interface (*continued*)
 stream 5
 transaction 6
 library, C socket 16
 main socket calls 8
 porting 17
 programming concepts 5
 protocol parameter 9
 TCP socket 14
 type parameter 8
 typical TCP socket session 14, 16
 typical UDP socket session 16
 UDP socket 16
 unconnected 16

software requirements
 Pascal 95
 sockets 5
 X window system 251

SOL_SOCKET, on C socket calls. 36

STANDARD TCPXLBIN 18

StartTcpNotice (Pascal) 126

stream sockets 5

stubs 205

subroutines (X window system) 256

svc_destroy() 222

svc_fds() 207

svc_freeargs() 223

svc_getargs() 223

svc_getcaller() 224

svc_getreq() 224

svc_register() 224

svc_run() 225

svc_sendreply() 225

svc_unregister() 226

svcerr_auth() 226

svcerr_decode() 226

svcerr_noproc() 226

svcerr_noprog() 227

svcerr_progvers() 227

svcerr_systemerr() 227

svcerr_weakauth() 228

svcrawl_create() 228

svctcp_create() 228

svcudp_create() 229

syntax diagram
 examples
 default xiii
 fragment xiii
 return arrow xii
 symbols xii
 variable xii
 table xii, xiii

system return codes 409

T

table
 syntax diagram xii, xiii

takesocket() 85

TAKESOCKET (IUCV) 198

TCP communication procedures
 (Pascal) 107

TCP/IP initialization and termination
 procedures (VMCF)
 abort a TCP connection 158
 begin TCP/IP service 152
 close a TCP connection 157

- TCP/IP initialization and termination procedures (VMCF) *(continued)*
 - close a UDP port 160
 - determine whether an address is local 161
 - end TCP/IP service 153
 - instruct TCPIP to obey a file of commands 162
 - obtain current status of TCP connection 158
 - obtain status information from TCPIP 162
 - open a UDP port 159
 - open TCP connection 154
 - receive raw IP packets of a given protocol 161
 - receive TCP data with FRECEIVetcp function 156
 - receive TCP data with RECEIVetcp function 157
 - receive UDP data 159
 - send an ICMP echo request 163
 - send raw IP packets 160
 - send TCP data 155
 - send UDP data 159
 - specifying the notifications to receive 153
 - tell TCPIP that your program will no longer use a particular IP protocol 161
 - tell TCPIP that your program will use a particular IP protocol 160
- TCP/UDP initialization procedures (Pascal) 106
- TCP/UDP/IP API (Pascal) 95
 - connection information record 97
 - connection state 96
 - data structures 96
 - file specification record 105
 - handling external interrupts 107
 - notification record 98
 - notifications 105
 - socket record 98
 - software requirements 95
 - using procedure calls 105
- TCP/UDP termination procedure (Pascal) 106
- TcpAbort (Pascal) 127
- TcpClose (Pascal) 127
- tcperror() 86
- TcpExtRupt 128
- TcpFReceive (Pascal) 128
- TcpFSend (Pascal) 131
- TCPIP ATCPPSRC file (Pascal) 95
- TCPLOAD
 - EXEC 2
 - using 3
- TcpNameChange 133
- TcpOpen (Pascal) 104, 134
- TcpOption (Pascal) 136
- TcpReceive (Pascal) 128
- TcpSend (Pascal) 131
- TcpStatus (Pascal) 137
- TcpVmcfRupt 138
- TcpWaitOpen (Pascal) 104, 134
- TcpWaitReceive 128
- TcpWaitSend 131

- Textlib (TXTLIB) Files
 - CLIB 3
 - CMSLIB 3
 - COMMTXT 3
 - GLOBAL 3
 - IBMLIB 3
 - PASCAL 3
 - RPCLIB 3
 - SCEELKED 3
 - TCPASCAL 3
 - TCPLANG 3
- TICK 369
- ticket-granting server 305
- tickets 304, 305
- Timer routines 109
- transaction sockets 6
- transactions, SMTP 359
- Translation information file 18
- TRAP, SNMP DPI request 328
- TRYINGtoOPEN (Pascal) 97

U

- UDP communication procedure 108, 139
- UDP socket session 16
- UdpClose (Pascal) 138
- UDPdatagramDELIVERED (VMCF) 104, 165
- UDPdatagramSPACEavailable (VMCF) 166
- UdpNReceive 139
- UdpReceive (Pascal) 104, 140
- UDPresourcesAVAILABLE (VMCF) 167
- UdpSend (Pascal) 141
- Unhandle (Pascal) 142
- UnNotifyIo 142
- UnpackedBytes 98
- URGENTpending (VMCF) 165
- user exit routines, SMTP 373, 390

V

- variables, environment 17
- VERB 368
- Virtual Machine Communication Facility (VMCF) Interface
 - CALLCODE notifications
 - ACTIVEprobe 168
 - BUFFERspaceAVAILABLE 164
 - CONNECTIONstateCHANGED 164
 - DATAdelivered 164
 - DUMMYprobe 168
 - PINGresponse 167
 - RAWIPpacketsDELIVERED 166
 - RAWIPspaceAVAILABLE 167
 - RESOURCESavailable 167
 - UDPdatagramDELIVERED 165
 - UDPdatagramSPACEavailable 166
 - UDPresourcesAVAILABLE 167
 - URGENTpending 165
 - CALLCODE system queries
 - IShostLOCAL 161
 - MONITORcommand 162
 - MONITORquery 162
 - PINGreq 163
 - functions 149
 - general information
 - data structures 147, 154

- Virtual Machine Communication Facility (VMCF) Interface *(continued)*
 - general information *(continued)*
 - use of VMCF interrupt header fields 148
 - use of VMCF parameter list fields 148
 - IP CALLCODE requests
 - CLOSErawip 161
 - OPENrawip 160
 - RECEIVERawip 161
 - SENDrawip 160
 - TCP CALLCODE requests
 - ABORTtcp 158
 - CLOSEtcp 157
 - FRECEIVetcp 156
 - FSENDtcp 155
 - OPENTcp 154
 - OPTIONtcp 158
 - RECEIVetcp 157
 - SENDtcp 155
 - STATUStcp 158
 - TCP/IP initialization and termination procedures
 - abort a TCP connection 158
 - begin TCP/IP service 152
 - close a TCP connection 157
 - close a UDP port 160
 - determine whether an address is local 161
 - end TCP/IP service 153
 - instruct TCPIP to obey a file of commands 162
 - obtain current status of TCP connection 158
 - obtain status information from TCPIP 162
 - open a UDP port 159
 - open TCP connection 154
 - receive raw IP packets of a given protocol 161
 - receive TCP data with FRECEIVetcp function 156
 - receive TCP data with RECEIVetcp function 157
 - receive UDP data 159
 - send an ICMP echo request 163
 - send raw IP packets 160
 - send TCP data 155
 - send UDP data 159
 - specifying the notifications to receive 153
 - tell TCPIP that your program will no longer use a particular IP protocol 161
 - tell TCPIP that your program will use a particular IP protocol 160
 - TCP/UDP/IP initialization and termination procedures
 - BEGINtcpIPservice 152
 - ENDtcpIPservice 153
 - HANDLEnotice 153
 - TCPIP communication CALLCODE notifications 151
 - TCPIP communication CALLCODE requests 150

Virtual Machine Communication Facility
(VMCF) Interface *(continued)*
 UDP CALLCODE requests
 CLOSEudp 160
 NRECEIVEudp 159
 OPENudp 159
 SENDudp 159
 when to use 147
VRFY 367

W

well-known port assignments
 TCP 413
 UDP 414
windows
 changing attributes 257
 communicating with window
 managers 267
 controlling the screen saver 265
 creating and destroying 256
 cut and paste buffers 270
 default error handling 267
 display functions 272
 enabling and disabling
 synchronization 266
 handling events 266
 hosts and access control 265
 keyboard event functions 268
 keyboard settings 264
 manipulating bitmaps 271
 manipulating images 271
 manipulating properties 258
 manipulating regions 269
 manipulating windows 257
 obtaining information 258
 opening and closing 256
 querying visual types 270
 resource manager 271
 setting selections 258
 window manager functions 264
write() 87
WRITE (IUCV) 199
writev() 88
WRITEV (IUCV) 199

X

X-ADDR environment variable 18
X-SITE environment variable 18
X Window Quick Reference tables
 associate table functions 276
 Athena widget support 290
 authorization routines 280
 character string sizes, querying 262
 clearing and copying areas 261
 communicating with window
 managers 267
 controlling the screen saver 265
 default error handling 267
 drawing lines 261
 drawing text 263
 extension routines 275
 filling areas 261
 fonts, loading and freeing 262
 handling events 266

X Window Quick Reference tables
(continued)
 handling window manager
 functions 264
 manipulating of
 bitmaps 271
 color cells 259
 colormaps 259
 cursors 263
 display functions 272
 graphics contents 260
 hosts and access control 265
 images 271
 keyboard event functions 268
 keyboard settings 264
 regions 269
 miscellaneous utility routines 277
 MIT extensions to X 275
 Pixmap, creating and freeing 259
 querying visual types 270
 synchronization, enabling and
 disabling 266
 transferring images 263
 using cut and paste buffers 270
 using the resource manager 271
windows
 changing attributes 257
 creating and destroying 256
 display, opening and closing 256
 manipulating 257
 manipulating properties 258
 obtaining information 258
 properties and atoms 258
 selections, setting 258
 X intrinsics routiness 280

X window system
 application resource file 253
 application resources 295
 associate table functions 293
 authorization routines 293
 compiling and linking 254
 creating an application 254
 defining widgets 294
 EBCDIC-ASCII translation 252
 extension routines 292
 how the interface works 251
 interface 251
 MIT extensions 293
 running an application 255
 sample programs
 Athena widget set, use of 300
 OSF/Motif-based widget set, use
 of 302
 Xlib calls, use of 299
 software requirements 251
 subroutines
 changing window attributes 257
 clearing and copying areas 261
 communicating with window
 managers 267
 controlling the screen saver 251,
 265
 creating and destroying
 windows 256
 creating and freeing pixmaps 259
 drawing lines 261
 drawing text 263

X window system *(continued)*
 subroutines *(continued)*
 enabling and disabling
 synchronization 266
 filling areas 261
 handling events 266
 handling window manager
 functions 264
 loading and freeing fonts 262
 manipulating bitmaps 271
 manipulating color cells 259
 manipulating colormaps 259
 manipulating cursors 263
 manipulating display
 functions 272
 manipulating graphics
 contexts 260
 manipulating hosts and access
 control 265
 manipulating images 271
 manipulating keyboard event
 functions 268
 manipulating keyboard
 settings 264
 manipulating regions 269
 manipulating window
 properties 258
 manipulating windows 257
 obtaining window
 information 258
 opening and closing a
 display 256
 properties and atoms 258
 querying character string
 sizes 262
 querying visual types 270
 setting window selections 258
 transferring images 263
 using cut and paste buffers 270
 using default error handling 267
 using the resource manager 271
 target display, identifying 254
 utility routines 293
 widget support
 Athena 296
 OSF/MOTIF based 297
 X Defaults 253
 X Window System Toolkit 293
 X-*XLATE* environment variable 18
 xdr_accepted_reply() 229
 xdr_array() 230
 xdr_authunix_parms() 230
 xdr_bool() 231
 xdr_bytes() 231
 xdr_callhdr() 232
 xdr_callmsg() 232
 xdr_double() 232
 xdr_enum() 233
 xdr_float() 234
 xdr_inline() 234
 xdr_int() 235
 xdr_long() 235
 xdr_opaque() 235
 xdr_opaque_auth() 236
 xdr_pmap() 236
 xdr_pmaplist() 237
 xdr_pointer() 237

xdr_reference() 237
xdr_rejected_reply() 238
xdr_replymsg() 238
xdr_short() 239
xdr_string() 239
xdr_u_int() 239
xdr_u_long() 240
xdr_u_short() 240
xdr_union() 241
xdr_vector() 241
xdr_void() 242
xdr_wrapstring() 242
xdrmem_create() 243
xdrrec_create() 243
xdrrec_endofrecord() 244
xdrrec_eof() 244
xdrrec_skiprecord() 244
xdrstdio_create() 244
xpvt_register() 245
xpvt_unregister() 245

Readers' Comments — We'd Like to Hear from You

z/VM
TCP/IP Level 3A0
Programmer's Reference
Version 3 Release 1.0

Publication No. SC24-5983-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Complete	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to find	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to understand	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Well organized	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Applicable to your tasks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

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.

Readers' Comments — We'd Like to Hear from You
SC24-5983-00



Cut or Fold
Along Line

Fold and Tape

Please do not staple

Fold and Tape

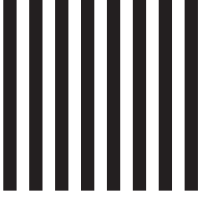


BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



Fold and Tape

Please do not staple

Fold and Tape

SC24-5983-00

Cut or Fold
Along Line



File Number: S370/4300/30XX-50
Program Number: 5654-A17



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC24-5983-00



Spine information:



z/VM

TCP/IP Programmer's Reference

Version 3 Release 1.0