IBM Transmission Control Protocol/ Internet Protocol for 4690 Application Interface Guide

Document Number (TCPAPPIF-SCRIPT)

Version Code: 1.2
Copy Printed: November 6, 1997 at 2:39 p.m.

Note

Before using this information and the products it supports, be sure to read the general information under "Notices and Format Information" on page 13.

Second Edition (June 1996)

This is the first edition of the *IBM Transmission Control Protocol/ Internet Protocol for 4690 Application Interface Guide*. This edition applies to Version 1 of the licensed program IBM 4690 Operating System, program number 5696-538, and to all subsequent releases and modifications until otherwise indicated in new editions. Changes are made periodically to the information herein.

Requests for additional copies of this document may be directed to Bob Niedergerke, at (919) 301-5781, or via e-mail at bobn@vnet.ibm.com. Mike Yawn is the management contact for this document, and may be reached at (919) 301-5465, or via e-mail at cmyawn@vnet.ibm.com. The fax transmission number is (919) 301-5891.

You can also order this document through your IBM representative or the IBM branch office serving your locality. They may retrieve a softcopy from the DSSFORUM repository, as TCPAPPIF LIST3820.

Written requests or comments may be addressed to:

IBM Corporation
Department CYR, Building 650
PO Box 12195
Research Triangle Park, North Carolina, 27709-2195 USA.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Contents

Notices and Format Information

Trademarks

The following terms are trademarks of IBM Corporation in the United States or other countries:

AIX	IBM	OS/2
AT	Operating System/2	Presentation Manager

The following terms are trademarks of other companies:

Trademark	**Owned By**
DEC	Digital Equipment Corporation
High C	MetaWare Inc.
IEEE	Institute of Electrical and Electronics Engineers
Network File System	Sun Microsystems, Inc.
NFS	Sun Microsystems, Inc.
Project Athena	Massachusetts Institute of Technology
Portmapper	Sun Microsystems, Inc.
UNIX	UNIX System Laboratories, Inc.
VT100	Digital Equipment Corporation
VT220	Digital Equipment Corporation
X Window System	Massachusetts Institute of Technology
Windows	Microsoft Corporation

Conventions Used in This Book

The following conventions appear throughout this book:

- Commands are presented in lowercase bold, but you can enter them in either uppercase or lowercase.
- Subcommands are presented in lowercase bold, and you must enter them in lowercase.
- File names are presented in uppercase, but you can enter them in either uppercase or lowercase.
- Periods in numbers separate the whole and the decimal portions of the numerals.
- Numbers over four digits appear 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 Read a Syntax Diagram

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

Syntax diagrams use the following symbols:

Symbol	**Description**
►►	Marks the beginning of the command or subcommand syntax
►	Marks the continuation of the command or subcommand
|	Marks the beginning and end of a fragment or part of the command or subcommand syntax
►◄	Marks the end of the command or subcommand syntax

Required parameters are displayed on the main path. Optional parameters are displayed below the main path. Default parameters are displayed above the main path.

Parameters are classified as keywords or variables. Keywords appear in uppercase or lowercase, but you must type them as they are shown in the syntax diagram. A command or a subcommand, for example, is a keyword. See "Conventions Used in This Book" on page 13 for the guidelines about entering commands and subcommands.

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

In the following example, *infile* is a variable. Replace it with the value that you want.

```
►►──rpcgen──infile──►◄
```

Include all punctuation such as colons, semicolons, commas, quotation marks, and minus signs shown in the diagram.

Choose One Required Item from a Stack: A stack of parameters with a parameter on the main path means that you must choose one from the stack.

```
                                              ┌────────────────┐
►►──snmp──┬─get──┬──host──community_name────▼─mib_variable─┴───►◄
          └─next─┘
```

Choose One Optional Item from a Stack: A stack of parameters without a parameter on the main path means that you do not have to choose any from the stack; but if you do, you cannot choose more than one.

```
►►──mode──┬──────────┬──►◄
          ├─ascii──┤
          └─binary─┘
```

Specify a Sequence More Than Once: An arrow above the main path that returns to a previous point means the sequence of items included by the arrow can be specified more than once.

```
               ┌─────────────┐
►►──mkfontdr──▼─directory─┴───►◄
```

1.0 General Information

This document specifies the functional description and major design points for the implementation of TCP/IP (Transmission Control Protocol/Internet Protocol) support for the 4690 Operating System.

TCP/IP Support of 4690OS Store Controllers

The following TCP/IP functions are provided as part of the the TCP/IP Support of 4690OS Store Controllers:

- TCP/IP on Token Ring

  This allows the 4690 controller with TCP/IP to communicate via token ring with other TCP/IP enabled platforms.

- TCP/IP on Ethernet

  This allows the 4690 controller with TCP/IP to communicate via ethernet with other TCP/IP enabled platforms.

- SNMP Agent

  Provides the basis for network management within a TCP/IP internetworking environment. SNMP defines the communication protocols between network resources and a network manager in order to control and exchange information within the network. The list of available information which the SNMP agent can provide is referred to as the MIB or Management Information Base. Requests from the SNMP manager to SNMP agent are used to retrieve status of a network resource. The SNMP agent can also provide an unsolicited notification of significant events referred to as traps.

- NETSTAT

  Command used to query TCP/IP about network status of the local host.

- PING

  Sends out an ICMP datagram to a specified destination and just waits for it to come back.

- ROUTE

  Used to define network routers.

- IFCONFIG

  Used to configure network interfaces.

- Socket Library (C-Language)

  A socket is a logical connection between two or more applications. Application interface to sockets via the socket library.

- BOOTP Client/Server

  This allows the 4690 to obtain (BOOTP Client) or provide (BOOTP Server) network initialization data.

- SNMP DPI

  The SNMP agent distributed program interface (RFC 1128) allows C language programs to extend the standard MIB to include customer specific variables and generate enterprise specific traps.

- Telnet Client

  This allows the 4690 to remotely logon to another platform using Telnet protocols and VT100 emulation.

- Telnet Server

This allows another platform to remotely logon to the 4690 using Telnet protocols and VT100, IBM3151, or ANSI emulation.

- LPR Client

  This allows the 4690 to send print jobs to a remote printer.

- File Transfer Protocol (FTP) Client and Server

  This allows file transfer between TCP/IP hosts.

- Trivial File Transfer Protocol (TFTP) Server

  This allows a client on another platform to transfer files using the TFTP protocol.

- File Transfer Protocol (FTP) Application Programming Interface Library

  This provides C language routines for some FTP client functions.

- SUN Remote Procedure Call Library (RPC) (C Language)
- Network File System (NFS) Server

  This allows NFS clients to transparently access files on a 4690 disk as if they were local. Only byte-stream access is provided. There is no record level function in NFS.

2.0 Functional Specifications and Design

2.1 High-Level Functional Description

TCP/IP support within the 4690 Operating System is implemented as a communications device driver, similar to the SNA driver and the Netbios driver. The TCP/IP protocol is supported on the 4690 token-ring and ethernet data link control (DLC) and can execute concurrently with Netbios, TCC, and SNA protocols.

User-level access to the driver is provided through the socket programming interface. A socket runtime library will be provided that can be linked to user-written applications.

In addition to the TCP/IP driver and socket library, a Simple Network Management Protocol (SNMP) agent (server) is implemented.

The Internet superserver, INETD, is also supported. This allows a single server to be active awaiting clients requests for other servers. INETD will start the server program on behalf of the client request.

BOOTP client/server - allows 4690 to obtain (BOOTP client) or provide (BOOTPD server) network initialization data.

SNMP DPI - SNMP agent Distributed Program Interface (RFC 1128) allows C language programs to extend the standard MIB to include customer specific variables and generate enterprise specific traps.

Telnet Server - allows Telnet clients to remotely logon to 4690 using the telnet protocols and VT100 emulation.

LPR client - allows 4690 to send print jobs to a remote printer. Implemented as command line interface, as well as addressability through the 4690 print spooler.

File Transfer Protocol (FTP) - allows file transfer between TCP/IP hosts. Implement client and server functions.

Trivial File Transfer Protocol (TFTP) - allows TFTP clients to transfer files from 4690. Implement server functions.

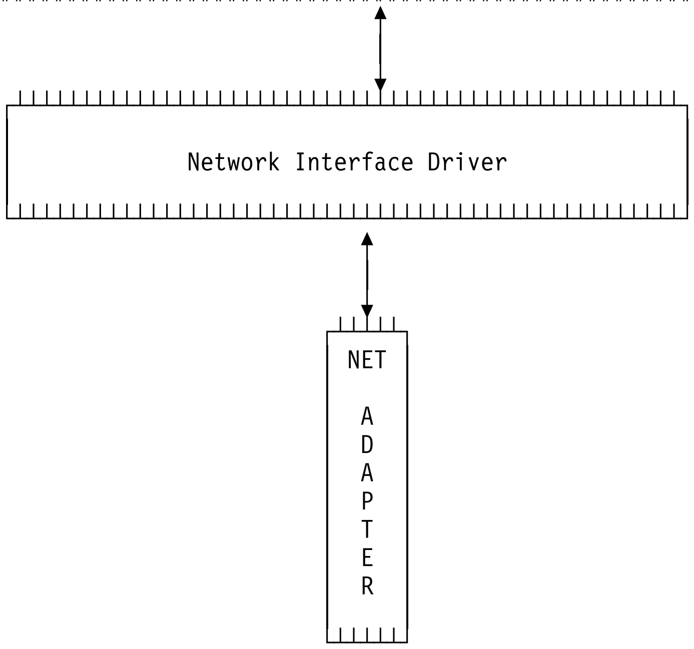
File Transfer Protocol (FTP) API - provides C language routines for some FTP client functions.

Network File System (NFS) server - allows NFS clients to transparently access files on a 4690 disk as if they were local. Byte-stream access only is offered. No record-level functions are supported with NFS.

Sun Remote Procedure Call Library (RPC) is supported for the C Language.

The following diagram depicts the entire suite of protocols.

```
%%%%%%%%                             %%%%%%%
% SNMP %         Servers             % FTP %  Clients  User Appls
%  DPI %                             % API %
%%%%%%%%######%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#      #     #     %  B  %     %  T  %     %     %   %          %
#  S   #  I  #     %  O  %  T  %  E  %     %  B  %   %  NFSD    %
#  N   #  N  #  F  %  O  %  F  %  L  %     %  O  %   %          %
#  M   #  E  #  T  %  T  %  T  %  N  %  F  %  O  % L % Sun RPC  %
#  P   #  T  #  P  %  P  %  P  %  E  %  T  %  T  % P %          %
#  D   #  D  #  D  %  D  %  D  %  T  %  P  %  P  % R %          %
##############%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#                           Socket                              #
#                           Library                             #
#################################################################
#                                                               #
#                          TCP/UDP/IP                           #
#                                                               #
#                           Driver                              #
#                                                               #
#################################################################
```



2.2 4690 TCP/IP Setup

Assuming the 4690 TCP/IP driver is installed and operational, most of the 4690 TCP/IP deliverables can be started by an operator or program control (in the case of client programs) or as background tasks (servers) setup through 4690 controller background application configuration or initiated by the INETD superserver. Some servers have dependencies on data files for start-up, e.g. FTPD requires a TRUSERS file which provides userid, password and access rights for FTP users. In addition, some servers have dependencies on other executables, e.g. NFSD requires an active PORTMAPPER process to register its procedures.

2.3 4690 TCP/IP Deliverables

2.3.1 Operation

For the most part, client programs will operate under manual or program control and servers will operate as background processes. This section examines the operation of each of the deliverables at a more detailed level.

2.3.1.1 BOOTP client

The BOOTP client program (ADX_SPGM:ADXHSIBL.286) is used to access TCP/IP host initialization data from a BOOTP server in the network. Specifically it is used to obtain the host's IP address, domain name, name server IP address, and network router address. The client sends a broadcast request indicating its hardware (token-ring adapter address) to a BOOTP server on the network and waits for a reply. If no reply is received from a server in 25 seconds, the client gives up and does not update any network initialization data.

When a server does reply, the BOOTP client will take the following actions:

1. Generate a batch file named ADX_SDT1:ADXBPxxZ.BAT containing an IFCONFIG command with the received host IP address and subnet mask as parameters. In addition, if the gateway (router) IP address was sent by the server, a ROUTE command is also added to this batch file.
2. Update the RESOLV file with the name of the domain and IP address of the name server for the host.

Note: The BOOTP client does not execute the IFCONFIG and ROUTE commands. It only builds the ADXBPxxZ.BAT batch file. That file must be run after BOOTP completes execution if network initialization is to occur.

2.3.1.2 BOOTP server

The BOOTP server (BOOTPD) is named ADX_SPGM:ADXHSIAL.286. It services requests from BOOTP clients and supplies network initialization data which it gathers from the BOOTPTAB (ADX_SDT1:ADXHSIAF.DAT).

The BOOTPTAB is a data file that contains the network initialization data for specific hosts, either based on hardware adapter address or host IP address. A sample BOOTPTAB is delivered with the BOOTP server program.

2.3.1.3 FTP client

The FTP client program, ADX_SPGM:ADXHSIGL.286, is used for file transfer between two TCP/IP hosts. It has a command line interface for interpreting user requests for file transfer settings and specifications.

In addition, the FTP client can access the NETRC file (ADX_SDT1: ADXHSIGF.DAT) for auto-login and macro definition.

2.3.1.4 FTP server

The FTP server (FTPD) is named ADX_SPGM:ADXHSIFL.286. It implements the FTP protocol and services the FTP client for file transfers.

Client access is validated by FTPD by searching the TRUSERS file, ADX_SDT1:ADXHSIUF.DAT, for userid, password, and access privileges. A sample TRUSERS file is delivered with the FTP server.

Under control of INETD, a separate instance of the FTP server will be initiated per client. If INETD is not used and the FTP server is currently connected to a client, other client connection requests will be rejected.

2.3.1.5 TFTP server

The TFTP server (TFTPD) is named ADX_SPGM:ADXHSITL.286. It implements the TFTP protocol and services TFTP clients for file transfers.

2.3.1.6 Network File System (NFS) server

The NFS server program (ADX_SPGM:ADXHSINL.286) provides transparent data access to NFS clients. A 4690 directory can be mounted by a remote host and file access granted to the remote user/application.

NFSD grants access based upon the contents of the EXPORTS file, ADX_SDT1:ADXHSIXF.DAT. This file contains the exported directories and the users which have access to them. A sample EXPORTS file is delivered with the NFS server.

Upon program start-up, NFSD registers its procedures with the PORTMAPPER program, ADX_SPGM:ADXHSIPL.286. It is essential that PORTMAPPER is executing before the NFS server is started. NFSD will not execute without the PORTMAPPER.

2.3.1.7 Telnet Server

The telnet server program is implemented on top of the existing 4690 remote operator system driver. It executes as a controller application and provides control of the 4690 main console and keyboard to a telnet client. Access is limited to a single client at any given time.

VT100, IBM3151, and ANSI terminal emulation are supported by the telnet server. Some keyboard mapping may be necessary to allow entry of control and function key sequences from a VT100 client.

2.3.1.8 Enhanced Telnet Server

The enhanced telnet server is implemented on top of the existing 4690 auxiliary console driver. It executes as a controller application and provides control of a 4690 auxiliary console to a telnet client. A maximum of eight clients can be connected at one time, which are configured via Controller Configuration.

VT220, VT100, ANSI, and HFT terminal emulation are supported by the enhanced telnet server. Non-USA keyboards are supported, and a user-configurable keying sequence to generate Alt+SysRq is available.

2.3.1.9 LPR client

The remote printer client program LPR (ADX_SPGM:ADXHSIRL.286) is implemented to support a command line interface as well as addressability through the 4690 print spooler driver.

As part of the 4690 print spooler, LPR will be invoked as a result of a print request to one of the 4690 printers. LPR is implemented in the back-end of the driver, i.e. the despool portion, so that it can implicitly be initiated by a "print" command. Once a file is queued by the spooler for the remote printer, LPR will send the file to the print server for printing. The printing of the file will be considered complete and removed from the print queue as soon as its data has been sent to the remote host. Printing status at the remote server will not be available from the 4690.

2.3.1.10 SNMP DPI

The SNMP Distributed Program Interface (DPI) is a set of C language runtimes which provide functions to extend the network management capabilities of the 4690 SNMP agent.

User programs which use the SNMP DPI interface directly with the SNMP agent to define enterprise-specific MIB variables and generate customer-specific TRAPs.

The SNMP DPI is delivered as ADXHSIDL.L86 suitable for linking with C language programs written for the MetaWare High C language compiler and linked with the 4690 application linker, LINK86.286.

2.3.1.11 FTP API

The FTP Application Programming Interface (API) is a set of C language runtimes which provide several of the FTP client capabilities to an application program.

The FTP API is delivered as ADXHSITL.L86 suitable to linking with C language programs written for the MetaWare High C language compiler and linked with the 4690 application linker, LINK86.286.

2.4 4690 TCP/IP File Index

Table 1 (Page 1 of 3). 4690 TCP/IP File Index

4690 Physical File Name	**4690 System-Defined Logical Name**	**Description**
ADX_SPGM:ADXHSI0L.286	(none)	TCP/IP Driver
ADX_SPGM:ADXHSI1L.286	(none)	SNMP Agent
ADX_SPGM:ADXHSI2L.286	(none)	TCPSTART.286 Driver Init
ADX_SPGM:ADXHSI3L.286	IFCONFIG	IFCONFIG.286 Driver Init
ADX_SPGM:ADXHSI4L.286	ROUTE	ROUTE.286 Driver Init
ADX_SPGM:ADXHSI5L.286	PING	PING.286 Network Support Program
ADX_SPGM:ADXHSI6L.286	NETSTAT	NETSTAT.286 Network Support Program
ADX_SPGM:ADXHSI7L.286	FINGER	FINGER.286 Network Support Program
ADX_SPGM:ADXHSI8L.286	(none)	MAKE_PW.286 Network Support Program
ADX_SPGM:ADXHSI9L.286	INETD	INETD Superserver
ADX_UPGM:ADXHSISL.L86	(none)	C Language Socket Library
ADX_SDT1:ADXIPxxZ.BAT	(none)	Driver Initialization Batch File, xx= Controller ID

Table 1 (Page 2 of 3). 4690 TCP/IP File Index

4690 Physical File Name	**4690 System-Defined Logical Name**	**Description**
ADX_SDT1:ADXHSIPF.DAT	PROTOCOL	*protocol* File
ADX_SDT1:ADXHSISF.DAT	SERVICES	*services* File
ADX_SDT1:ADXHSINF.DAT	NETWORKS	*networks* File
ADX_SDT1:ADXHSIRF.DAT	RESOLV	*resolv* File
ADX_SDT1:ADXHSIHF.DAT	HOSTS	*hosts* File
ADX_SDT1:ADXHSIDF.DAT	(none)	SNMP TRAP destination file
ADX_SDT1:ADXHSIQF.DAT	(none)	SNMP ASCII text community name file
ADX_SDT1:ADXHSIEF.DAT	(none)	SNMP encrypted community name file
ADX_SDT1:ADXHSIIF.DAT	(none)	INETD data file
ADX_SPGM:ADXHSIAL.286	BOOTPD	BOOTP server
ADX_SPGM:ADXHSIBL.286	BOOTP	BOOTP client
ADX_SPGM:ADXHSIFL.286	(none)	FTP server
ADX_SPGM:ADXHSIGL.286	FTP	FTP client
ADX_SPGM:ADXHSINL.286	(none)	NFS server
ADX_SPGM:ADXHSIPL.286	(none)	Portmapper
ADX_SPGM:ADXHSIRL.286	LPR	LPR client
ADX_SPGM:ADXHSIIL.286	(none)	Telnet server (keyboard process)
ADX_SPGM:ADXHSISL.286	(none)	Telnet server (screen process)
ADX_SPGM:ADXHSIUL.286	(none)	Enhanced Telnet server
ADX_SPGM:ADXHSITL.286	TFTPD	TFTP server
ADX_SPGM:ADXHSIXL.286	REXEC	REXEC client
ADX_UPGM:ADXHSIDL.L86	(none)	SNMP DPI Library
ADX_UPGM:ADXHSITL.L86	(none)	FTP API Library
ADX_SDT1:ADXBPxxZ.BAT	(none)	BOOTP client batch file output, xx = Controller ID
ADX_SDT1:ADXHSIAF.DAT	(none)	BOOTPD Bootptab File
ADX_SDT1:ADXHSIGF.DAT	(none)	FTP client NETRC File
ADX_SDT1:ADXHSIUF.DAT	(none)	FTP server TRUSERS File
ADX_SDT1:ADXHSIXF.DAT	(none)	NFSD EXPORTS File
ADX_SDT1:ADXHSITF.DAT	(none)	NFSD MTAB File
ADX_SDT1:ADXHSIMF.DAT	(none)	Telnet Server messages File

Table 1 (Page 3 of 3). 4690 TCP/IP File Index

4690 Physical File Name	4690 System-Defined Logical Name	Description
ADX_SDT1:ADXHSIZF.DAT	(none)	Optional SNMP environment variable definitions file
ADX_SDT1:ADXHSIRL.286	(none)	LPR Print Spooler
ADX_SDT1:ADXHSIVL.286	VT100	VT100 Client

The TCP/IP software components are all derived from Berkeley Software Distribution (BSD) 4.3 UNIX. The software was ported from the TCP/IP for OS/2 EE product and modified to operate on the 4690 Operating System.

2.5 System Hardware and Software Requirements

2.5.1 Hardware

4690 TCP/IP requires controller memory to support the load and use of the communications driver. In addition, memory must also be available to execute the SNMP application. At a minimum, an estimated 350K bytes of memory will be required for the initialized driver (install size is about 100K and dynamic memory allocation is between 100-200K byte) and an estimated 200K bytes of memory will be required for SNMP. Additional controller memory must be available for each of the client and server applications executing in the controller. For instance, controller memory for user-written applications is not included in these estimates. There are more detailed estimates in Chapter 3.

4690 TCP/IP is be implemented for use on a Token-Ring or Ethernet network.

2.5.2 Software

4690 TCP/IP will require 4690 Version 1 Release 1 level of Operating System.

2.6 4690 TCP/IP Installation

Initial installation of the 4690 TCP/IP driver and applications will be done via a batch file, INSTALL.BAT, and 4690 Apply Software Maintenance. Updating existing installations can be performed using 4690 Apply Software Maintenance.

2.7 4690 TCP/IP Configuration

2.7.1 Driver Initialization

A single configuration batch file (ADX_SDT1:ADXIPxxZ.BAT) containing three commands and their parameters is used by the TCP/IP driver to initialize itself. This file is built by the user with a standard ASCII text editor and contains the ADXHSI2L.286 (TCPSTART), ADXHSI3L.286 (IFCONFIG) and ADXHSI4L.286 (ROUTE) commands with their parameters.

The TCPSTART command does not require any parameters. It must be the first command in the configuration file and is used to indicate to the driver it should allocate memory for operation. (At IPL the driver is loaded, but does not become operational until TCPSTART is executed.) The IFCONFIG and ROUTE commands will be executed next by the driver during its initialization.

2.7.2 Network Data Files

In addition to the driver initialization configuration file, there are several other data files required:

- ADX_SDT1:ADXHSIPF.DAT - *protocol* file
- ADX_SDT1:ADXHSISF.DAT - *services* file
- ADX_SDT1:ADXHSIRF.DAT - *resolv* file
- ADX_SDT1:ADXHSIHF.DAT - *hosts* file

The *protocol* file specifies to the driver which protocols are available, e.g. TCP, UDP, ICMP. The *services* file is static and does not necessarily require user modification; it lists all servers and their well-known port numbers.

The data in the *resolv* and *hosts* files is created by the user. The *resolv* file provides the name and location of a network name server (if one exists) to resolve host names to IP addresses. The *hosts* file contains a local list of host names and addresses and is used if resolution through a name server fails or if the *resolv* file does not exist.

2.7.3 SNMP Data Files

When SNMP is to be used, two data files may be created by the user.

If TRAPs are to be sent by the 4690 SNMP agent (server) to the SNMP monitor (client), a data file must be created by the user listing the name or address of that SNMP monitor and the transport protocol used to send the TRAP. This file is named ADX_SDT1:ADXHSIDF.DAT.

Also, if access to the 4690 network management objects is to be restricted to certain SNMP monitors, a community name file can be generated by the user containing address masks and network management passwords.

This community name file can be encrypted for secure access to 4690 management objects by network monitors. The program used to generate the encrypted community name file is called ADX_SPGM:ADXHSI8L.286 (MAKE_PW). The input file is an ASCII text file named ADX_SDT1:ADXHSIQF.DAT (PW.SRC) created by the user. The encrypted output file is named ADX_SDT1:ADXHSIEF.DAT (SNMP.PW).

2.7.4 INETD Data File

The Internet superserver requires a data file to identify the available server applications, protocol, and server names that can be activated by INETD. Modifications to this data file are required if user-written server applications are to be activated by INETD.

The file name for the INETD data file is ADX_SDT1:ADXHSIIF.DAT.

2.7.5 Logical and Physical Names

2.7.5.1 System-Defined

Logical names are defined for use by the driver and applications when the driver is loaded at IPL. When possible, logical names reflecting the UNIX standard filenames are system-defined as their actual 4690 physical filename, e.g. *hosts* is a logical name for ADX_SDT1:ADXHSIHF.DAT (physical filename). The 4690 file naming convention is used for all files associated with TCP/IP, its applications and servers.

2.7.5.2 User-Defined

The user can define logical names for the *syscont*, *sysloc*, and *hostname* environment variables used by the SNMP agent.

The logical name assignments do not necessarily have to be unique. Refer to the section on 'SNMP agent environment names' in the Users Information section of this document.

2.8 4690 TCP/IP Operation

2.8.1 Driver

The 4690 TCP/IP driver will be loaded at IPL time based on the existence of the configuration batch file, ADX_SDT1:ADXIPxxZ.BAT, which contains the TCPSTART, IFCONFIG, and ROUTE commands. If that file exists, the driver will be installed and will initialize itself by invoking the batch file to run. As a result, the driver initialization commands will execute.

Note that the link monitoring application, ADXHSNLL.286, will NOT have knowledge of any activity of TCP/IP nor can the driver be started by that program.

Once loaded, the driver will remain resident. Real-time removal of the driver is not possible.

2.8.1.1 SNMP Agent

The SNMP agent, ADX_SPGM:ADXHSI1L.286, can be started in command mode by invoking the program by name. Alternately, it can be configured as a background application to be started manually or automatically at IPL.

2.9 Socket Programming Interface Library

The 4690 TCP/IP driver is accessible by C language applications exclusively through the socket programming interface. The socket library is delivered as an L86 module named ADX_UPGM:ADXHSISL.L86. 4690 TCP/IP applications written in the C programming language and compiled with the MetaWare High C compiler can be linked with the socket library to produce executable code.

The MetaWare High C compiler with the C language runtime library for developing 4690 C applications is available from Integrated Systems, Inc., as the EPOS Developer's Kit.

A socket library for linking 4690 POS BASIC applications is not provided.

2.9.1 Supported socket calls

The supported set of C language socket calls for 4690 TCP/IP are listed below.

Table 2. 4690 C Language Socket Calls

accept()	bind()	bswap()[1]
connect()	dn_comp()	dn_expand()
endhostent()	endnetent()	endprotoent()
endservent()	gethostbyaddr()	gethostbyname()
gethostent()	gethostid()	getnetbyaddr()
getnetbyname()	getnetent()	getpeername()
getprotobyname()	getprotoent()	getservbyname()
getservbyport()	getservent()	getsockname()
getsockopt()	htonl()	htons()
inet_addr())	inet_lnaof()	inet_makeaddr()
inet_netof()	inet_network()	inet_ntoa()
ioctl()	listen()	lswap()[1]
ntohl()	ntohs()	port_cancel()[2]
readv()	recv()	recvfrom()
res_init()	res_mkquery()	res_send()
rexec()	select()	send()
sendto()	sethostent()	setnetent()
setprotoent()	setservent()	setsockopt()
shutdown()	sock_init()[1]	socket()
soclose()[3]	writev()	

2.10 SNMP Agent - ADX_SPGM:ADXHSI1L.286

The SNMP agent is a UDP server application primarily driven by a SNMP monitor. The SNMP agent maintains several network management objects which can be accessed by the SNMP monitor. Objects compose the Management Information Base and are divided into eight groups. 4690 TCP/IP will support the objects in seven of the eight groups (Exterior Gateway Protocol group will not be supported). Those groups are:

1. System
2. Interfaces
3. Address Translation
4. Internet Protocol
5. Internet Control Message Protocol

[1] OS/2 and 4690 socket extension call. Not a BSD 4.3 socket call.

[2] 4690 socket extension call.

[3] Used instead of BSD 4.3 close() socket call.

6. Transmission Control Protocol
7. User Datagram Protocol

2.10.1 Management Information Base II (MIB-II)

The objects supported for each group represent the MIB. The 4690 SNMP agent will provide objects corresponding to the MIB-II definition. Management objects in the 4690 cannot be SET by an SNMP monitor; they must be accessed read-only.

2.10.2 TRAPs

The SNMP TRAPs that are supported are the following:

- Cold Start - notifies an SNMP monitor that the agent has just been started and initialized and all management variables and tables are in the reset state.
- Authentication Failure - indicates an attempt by an unauthorized network management station to access the SNMP agent's objects.

2.11 Internet Superserver (INETD) - ADX_SPGM:ADXHSI9L.286

INETD is implemented as a server application which awaits incoming requests from client applications. If possible, the INETD superserver will start the server application requested by the client and pass control to it. This allows a single application to service all clients and replaces the need to have multiple servers concurrently active.

2.12 Compiling and Linking considerations

4690 TCP/IP applications may be compiled with the Metaware High C** compiler using the *big* memory model. The following variables should be defined with the -def option on the compiler (e.g. HCDX86O) command line:

- OS2
- far
- near

An executable program may then be created using the LINK86 program. You should link with the following libraries:

HCBE.L86	High C big memory model library
ADXHSISL.L86	4690 TCP/IP Socket library

Depending on what type of TCP/IP application you are developing, you may also need to link with the following libraries:

ADXHSIDL.L86	4690 SNMP DPI library
ADXHSIRL.L86	4690 Sun RPC library
ADXHSITL.L86	4690 FTP API library
FLEXLIB.L86	FlexOS library

Notes:

1. For information about the use of a particular API, see the chapter on that API.
2. For more information about the compile and link options, refer to the High C compiler documentation.
3. For a sample socket application, see Appendix A, “Sample socket application: Echo server” on page 365.

2.13 Miscellaneous

2.13.1 Network Support Programs

Along with the TCP/IP driver, socket library, and SNMP agent, several network support programs are delivered.

1. ADXHSI5L.286 (PING) - a diagnostic tool that sends an echo request to a foreign host to determine if it is accessible.
2. ADXHSI6L.286 (NETSTAT) - provides information about local system, TCP connections, UDP and IP statistics, driver memory usage, and socket information.
3. ADXHSI7L.286 (FINGER) - client application to obtain information about users on a foreign host.
4. ADXHSI8L.286 (MAKE_PW) - SNMP password encryption program for community name file.

3.0 Users Information

This is the User's Guide for 4690 TCP/IP services. This document contains:

1. Hardware and Software Requirements
2. Identification of Software components of this package
3. Installation and Setup instructions
4. Configuration and Usage of TCP/IP applications
5. Error messages

3.1 Hardware and Software Requirements

3.1.1 Hardware Requirements

3.1.1.1 Adapter - Token-Ring

4690 TCP/IP was designed to operate with the IBM Token-Ring adapter installed on the 4690 Controller. PS/2 token-ring adapters can be used. Additionally, 4690 TCP/IP can concurrently operate with the Multiple Controller Feature, i.e. Distributed Data Application (DDA) and/or SNA protocols on the token-ring network. Note that increasing the token-ring adapter shared-ram can help network performance when running multiple protocols on the token-ring. Adapter shared ram can be set on the IBM Token-Ring Adapter/2 16/4 (PS/2 only) to 16K (default), 32K, and 64K, but only to the extent that it does not conflict with other adapters installed in the 4690 controller. Refer to the Adapter/2 16/4 installation and setup for more information on how to change the shared ram for your adapter using the PS/2 hardware reference diskette.

3.1.1.2 Adapter - Ethernet

Beginning with 4690 Operating System maintenance level 9500 and together with 4690 TCP/IP level 9505, the TCP/IP protocols are supported on the ethernet network. Contact your IBM representative for a current list of ethernet adapters supported by the 4690 Operating System. All 4690 TCP/IP functions and applications are available on the ethernet LAN, however only one LAN type can be selected for the 4690 Operating System. Therefore, 4690 TCP/IP can only use a single interface at any given time.

3.1.1.3 Fixed Disk

4690 TCP/IP requires approximately 3M bytes of free fixed-disk storage.

3.1.1.4 Memory

Memory requirements for 4690 TCP/IP vary depending on the number and type of TCP/IP applications which are active, however, at a minimum approximately 500K bytes of RAM should be available. This allows for about 400K for the TCP/IP driver (code and data), and one or two TCP/IP applications, e.g. FTP and Telnet. Specific memory requirements per function are provided in the Table 1. Note that TCP/IP driver memory is required as a minimum and all other memory calculations for optional applications should be added to the driver memory.

```
Function             Memory (Kbytes)
-----------------------
TCP/IP driver           400 (note 1)
FTP client              220
FTP server              200
TFTP server             200
INETD superserver        50
BOOTP client            120
BOOTP server            200
Telnet client
 - VT100*               200
Telnet server
 - ADXHSIIL             375
 - ADXHSIUL             104
SNMP agent              230
Portmapper*             220
NFS* server             275
Rexec client            200
LPR client              180
```

Note 1. Driver memory is 400K for default of up to 48 maximum sockets. The maximum number of sockets is configurable up to 128. Each socket above 48 allocates another 2K of memory. (To configure more than 48 sockets, see the section below on "Configuring the interface".)

3.2 Software Requirements

4690 TCP/IP requires Version 1 Release 1 (V1R1) of the 4690 Operating System. The support for ethernet networks is provided in 4690 Operating System maintenance 9500. The support for the enhanced telnet server is provided in 4690 Operating System maintenance 9630.

4690 TCP/IP can operate in one of two 4690 system configurations:

- Configuration A

  4690 LAN environment in which DDA (Distributed Data Application) is being used for controller communications, file mirroring, and file services. In this environment the 4690 controller can have a role of Master, File Server, Alternate Master, Alternate File Server or Subordinate.

- Configuration B

  Non-DDA LAN environment in which the store controller operates independent of any other controller on the LAN. Typically, this environment consists of a single store controller connected on the LAN with other machines. It is normally designated as the Master controller although it has no knowledge of Alternate or Subordinate controllers.

3.2.1 4690 TCP/IP Files

Most 4690 TCP/IP software program files, data files and programming libraries adhere to the 4690 Operating System file naming conventions allowing this software to be maintained through 4690 Apply Software Maintenance (local) and ADCS/HCP (remote). In some cases logical names are defined when 4690 TCP/IP is loaded.

3.2.2 4690 TCP/IP installation, setup and configuration

In order to use 4690 TCP/IP, you should perform the following steps:

- Install the program and data files from diskette
- Configure the interface
- optionally define remote host names or remote name server

3.2.3 Installing 4690 TCP/IP in 4690 DDA LAN system - Configuration A

Follow these instructions for installing 4690 TCP/IP on a 4690 system configured to use 4690 DDA on the LAN as described in "Software Requirements - Configuration A" above.

4690 TCP/IP is installed using 4690 Apply Software Maintenance (ASM). Insert diskette #1 into drive A: and invoke a:install from a 4690 command prompt. When asked if installation is local or remote, select local. If the install program completes successfully, you can proceed with the installation using 4690 ASM using the steps below:

1. From System Main Menu select Installation and Update Aids
2. Select Apply Software Maintenance
3. Select Transfer Maintenance from Diskette
4. Select 4690 TCP/IP
5. Insert diskette #1 and continue with additional diskettes as prompted
6. When transfer completes, select Activate Maintenance
7. Select 4690 TCP/IP - Accept (if this is first installation, accept is equivalent to Test since there is nothing to backup)
8. Remove the diskette in drive A and prepare for IPL

Once the 4690 controller IPLs, TCP/IP will be installed.

3.2.4 Installing 4690 TCP/IP in 4690 non-DDA system - Configuration B

Follow these instructions for installing 4690 TCP/IP on a 4690 non-DDA environment as described in "Software Requirements - Configuration B" above.

You must run store controller configuration to use 4690 TCP/IP in a non-DDA environment:

- From System Main Menu select Installation and Update Aids
- Select Change Configuration
- Select Controller Configuration. Answer N (no) and press enter to the following question on the screen:

  Are you configuring a Store System that uses the IBM 4690 Multiple Controller Feature (LAN) to support the Data Distribution Application?
- Select LAN media type, either ethernet or token-ring
- Answer Y (yes) and press enter to the following question on the screen:

  Are you configuring a store system that uses SNA communication on a LAN?

You can optionally do any other store controller configuration from this point on, however it is not necessary to run 4690 TCP/IP.

- When finished with controller configuration, press ESC to return to the CONFIGURATION screen and select Activate Configuration.
- Select Controller Configuration

  Controller configuration should successfully activate the changes you just made. You can then exit (F3=QUIT) the configuration application and continue with 4690 TCP/IP installation.

4690 TCP/IP is installed using 4690 Apply Software Maintenance (ASM). Insert diskette #1 into drive A: and invoke a:install from a 4690 command prompt. When asked if installation is local or remote, select local. If the install program completes successfully, you can proceed with the installation using 4690 ASM using the steps below:

1. From System Main Menu select Installation and Update Aids
2. Select Apply Software Maintenance
3. Select Transfer Maintenance from Diskette
4. Select 4690 TCP/IP
5. Insert diskette #1 and continue with additional diskettes as prompted
6. When transfer completes, select Activate Maintenance
7. Select 4690 TCP/IP - Accept (if this is first installation, accept is equivalent to Test since there is nothing to backup)
8. Remove the diskette in drive A and prepare for IPL

Once the 4690 controller IPLs, the controller configuration will become active and TCP/IP will be installed.

3.2.5 Configuring the interface

The 4690 TCP/IP protocol driver is loaded during controller IPL based on the existence of one file named ADXIP*xx*Z.BAT in subdirectory C:\ADX_SDT1, where *xx* is the 2 character 4690 controller ID. This file has local distribution attribute and during installation it is copied onto the master controller ONLY.

IMPORTANT!

You must have the configuration batch file ADXIPxxZ.BAT on each machine that you want the run 4690 TCP/IP. An example batch file is copied onto the master controller during 4690 TCP/IP installation, however you should update it for your particular network configuration.

The file C:\ADX_SDT1\ADXIPxxZ.BAT should contain (at a minimum) the following lines:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% adxhsi2l                       %
% ifconfig lan0 <ip_address>     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

where the ip_address field contains this machine's (host) internet address in dotted decimal form, for example 9.67.39.81 is a valid internet address. The assignment of internet addresses for a proprietary network are enterprise defined. Internet addresses for the public Internet must be obtained from the administrators of the Internet. All TCP/IP hosts in the network should have a unique internet address.

The ADXHSI2L command is equivalent to the tcpstart command, however ADXHSI2L can accept a single parameter which indicates the maximum number of sockets that can be active at any one time. If no parameter is specified (as shown above), the default is 48 sockets. For most cases this is sufficient. If more sockets are desired, enter a value between 49 and 128 after the ADXHSI2L command. Note that additional memory allocated for each socket above 48 (about 2K per socket) regardless of whether or not the socket has an active connection. Also, once set to a value, the maximum number of sockets cannot be changed by a subsequent invocation of ADXHSI2L, i.e. the maximum can be set only once after a machine IPL. The maximum sockets value represents the combined total of both UDP and TCP sockets.

3.2.6 Identifying a network router

If your network contains an IP router, file ADXIPxxZ.BAT can identify it using the route command, for example:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% route add default <router_ip_address> 1  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

where the router_ip_address identifies the internet address of the network router in dotted decimal form. The route command should follow the ifconfig command in the configuration batch file ADXIPxxZ.BAT.

3.2.7 Using TCP/IP host names

It is usually difficult to remember internet addresses in dotted decimal form, therefore TCP/IP allows a host name to be used in place of the internet address. To resolve a name to an internet address, TCP/IP does the following:

1. contact the name server host designated in the resolv file using the domain name system (DNS) protocols.
2. lookup the name in the local hosts file.

3.2.8 Resolv file

An attempt will be made to contact the name server ONLY if the resolv file C:\ADX_SDT1\ADXHSIRF.DAT exists and identifies the domain and the address of the name server. During the installation of 4690 TCP/IP a sample resolv file is copied onto the 4690 controller.

IMPORTANT!

You should either update the resolv file C:\ADX_SDT1\ADXHSIRF.DAT with the domain name and internet address of your name server or erase this file. If you do not erase (or rename) this file or if it contains incorrect information (i.e. the internet address of a name server that does not exist), name resolution will be attempted anyway resulting in unnecessary delays.

Note: A 4690 controller CANNOT be a name server, therefore the resolv file should not identify a 4690 host internet address.

3.2.9 hosts file

If a resolv file does not exist, or if the name server cannot resolve the name to an internet address, the local hosts file C:\ADX_SDT1\ADXHSIHF.DAT is searched for a match. The hosts file correlates a hostname to an internet address. You may have many host name and internet address pairs in the hosts file. If you are not using a name server, you should update the hosts file with names and addresses of those hosts with whom this machine will normally communicate.

A hosts file C:\ADX_SDT1\ADXHSIHF.DAT might consist of the following entries:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 9.67.39.80   cc      # 4690 controller CC        %
% 9.67.39.81   dd      # 4690 controller DD        %
% 9.67.39.82   isp     # TCP/IP In-Store Processor %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A sample host file is copied onto the controller when 4690 TCP/IP is installed.

3.3 Setup and Usage of other TCP/IP applications

3.3.1 INETD superserver - ADXHSI9L.286

TCP/IP servers are designed to be "ready" to accept client connections whenever they are listening to their port number. Standard servers, e.g. ftp, telnet, listen to well-known ports awaiting incoming connections from clients.

The inetd superserver is a TCP/IP server that can listen to multiple TCP ports awaiting incoming client connections for specific servers, e.g. ftp server, telnet server. When a client attempts to connect to that server, inetd automatically starts the server as a 4690 background application. Using inetd allows you to have a single server continuously running instead of many servers resulting in a savings of resources. In addition, if using inetd to start servers, multiple instances of the same server can be concurrently active.

The file that identifies which servers inetd can start is C:\ADX_SDT1\ADXHSIIF.DAT. This file is copied onto the controller during 4690 TCP/IP installation and contains an entry for ftp server and the regular telnet server. It is necessary to change this file to use the enhanced telnet server, as described in the section on the server below.

Getting the inetd superserver started can be accomplished by configuring it as a 4690 background application which gets started when the controller IPLs. The inetd superserver executable filename is ADX_SPGM:ADXHSI9L.286. For information about defining 4690 background applications, see the IBM 4690 Store Systems User's Guide.

IMPORTANT!

The inetd superserver can only start TCP servers. UDP servers must be started by other means, e.g. configuring as a 4690 background application which starts at IPL. Therefore if you add servers to the inetd data file, ensure that they are TCP based. Also, if the server is a user written TCP socket application and you would like inetd to start it on demand, the application should be coded to expect the socket number passed to it as the second input parameter (the first being the string "BACKGRND") in the argument list.

3.3.2 FTP client - ADXHSIGL.286

The ftp client is an interactive file transfer program invoked at the 4690 command line by issuing the command ADXHSIGL, or using the logical name FTP.

Once invoked, the ftp client presents the user an ftp prompt. File Transfer commands/settings can be issued at the ftp prompt. For a list of valid commands type 'help' at the ftp prompt.

The netrc file is an optional file which can be used by the ftp client to automate logon to certain servers and provide macro definition for frequently used command strings. The 4690 netrc file is named C:\ADX_SDT1\ADXHSIGF.DAT. A sample netrc file is copied onto the controller during 4690 TCP/IP installation.

Note: Some ftp client operations can also be invoked with user-written applications through the use of the FTP Application Programming Interface (FTP API). See the section below regarding FTP API for more information.

3.3.3 FTP server - ADXHSIFL.286

The 4690 ftp server extends file transfer capabilities to an ftp client. Only a single ftp client may be logged on at any given time to the ftp server. Under control of the inetd superserver, a new instance of the ftp server will be started for each client logon request thus allowing multiple clients to be concurrently connected to multiple servers. However, if inetd is not used and the ftp server is currently connected to a client, additional client requests will be rejected until the server concludes its connection and again begins to await incoming connections.

The client must first logon to the server using a userid and (optional) password. The client userid and password is validated by the server when it finds a matching entry in the TRUSERS file. This file is named C:\ADX_SDT1\ADXHSIUF.DAT and should contain an entry for each client with whom the server will maintain an ftp session. Connection requests from unauthorized clients will be rejected. You should update this file with the ftp clients you wish to grant access.

In addition to userid and password information in the TRUSERS file, each entry lists the read/write access privileges for that user. In this way certain users can be limited to more restricted access, e.g. read-only, or more liberal access, e.g. read-write. Access can be granted per drive/directory for physical drives A: C: D: and RAM drives T: U: V: W:. In addition, the 4690 printer can be used as a file transfer destination for copying files directly from the client to prn device.

A sample TRUSERS file is copied onto the controller during 4690 TCP/IP installation. It contains an entry for user 'anonymous' (no password required) and gives read/write access to C:\ and its subdirectories.

IMPORTANT!

The TRUSERS file must exist in order for the ftp server to successfully initialize. If the server cannot access file C:\ADX_SDT1\ADXHSIUF.DAT it will reject the client logon request, log a system message and then terminate.

An additional command has been added to the 4690 ftp server to support the starting of 4690 background applications from an ftp client. This capability is provided through the use of the "quote" ftp client command with the ADXSTART server command provided the user at the client has the necessary execution authorization.

An example of starting background application named testpgm.286 is:

```
  ftp>  quote adxstart testpgm.286
```

The previous command would cause the 4690 ftp server to start testpgm.286 as a 4690 background application. Note that the ftp server returns a message to the ftp client stating that the program has been started. The server does not wait for the background program to complete.

Also, note that the program name must be fully path-qualified and is limited to 22 characters. If using path-qualified program names from a unix client, you may need to surround the entire quote string in double quotes if the backslash character is required. Optionally you can use forward slash for the path name delimiter.

Program parameters can be specified immediately following the program name, each separated by at least one blank. The length of the parameter list cannot exceed 46 characters. The program will be started with default priority of 200 and cannot be changed.

User authority to start background applications using the 'quote' command is granted through the TRUSERS file. A special execution access privilege is granted to clients using the 'ex:' tag for the user. This tag is similar to the 'rd:' and 'wr:' tags used to grant read and write privileges to users. The sample TRUSERS file, ADXHSIUF.DAT, contains an entry for user=anonymous (no password required) and grants write access (wr:) for drive c:\ (and all subdirectories of the root directory) and read access (rd:) for drive c:\ (and all subdirectories of the root directory) and background application initiation access (ex:). This entry is shown as:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% user: anonymous          %
% wr: c:\                  %
% rd: c:\                  %
% ex:                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Note that the 'ex:' tag does not require additional parameters. Any user without execution authority will be returned a message by the ftp server indicating they are not authorized to start background applications.

3.3.3.1 FTP Server Timeout

A receive_data timeout has been added to the 4690 FTP server to prevent a server hang condition in the event that either the FTP client has crashed or the physical link between the client and server is disconnected.

The timeout is set by the definition of a User-Defined Logical Filename. To define the logical name, use 4690 Controller Configuration to add the name ADXFTPTO. The definition for this name can be a value between 1 and 14400 and represents the maximum number of SECONDS to wait for a client to transmit data to the server. Note that this timeout value is only relevant to the server while it is receiving data. It does not affect either the ftp control connection IDLE time (which is also configurable from the client using the ftp site command) or have any bearing on retransmission of data when the server is sending data to a client that may not be responding.

If the ADXFTPTO logical filename is not defined, or it is defined with an invalid value, a default timeout of 2 hours is used. If the timeout expires, the FTP server closes all connections with the client and terminates.

3.3.4 NFS server - ADXHSINL.286

The Network File System server is a UDP-based server which allows remote access to files located in 4690 directories. An NFS client mounts an exported 4690 directory and accesses files as if they were local to the client machine. Multiple NFS clients can mount the same or different 4690 exported directories all at the same time, however, only a single NFS server is required to be active to handle and keep track of all client requests.

Note: 4690 TCP/IP contains an NFS server capability only. No NFS
client support is included.

The NFS server determines which 4690 directories it can export and which users can mount them using the exports file.

A sample exports file is copied onto the controller during 4690 TCP/IP installation as C:\ADX_SDT1\ADXHSIXF.DAT. You should modify this file to add/change/delete exported directories and access privileges as well as list additional users.

IMPORTANT!

The exports file is REQUIRED in order for the 4690 NFS server to initialize successfully. If the 4690 NFS server cannot obtain access to the exports file, it will log an error in the system log and terminate.

IMPORTANT!

The remote procedure call (RPC) Portmapper is a mandatory prerequisite for the NFS server. It must be active before the NFS server is started so that NFS procedures can be registered. The Portmapper is program file C:\ADX_SPGM\ADXHSIPL.286 and can be started during a controller IPL. Likewise, you can configure the NFS server to start during IPL, however ENSURE THAT YOU DEFINE THE PORTMAPPER BACKGROUND APPLICATION FIRST FOLLOWED BY THE NFS SERVER. The NFS server must be started after the Portmapper is active. Defining the Portmapper first in the configuration will cause it to get started before the NFS server. The NFS server program file name is C:\ADX_SPGM\ADXHSINL.286.

3.3.5 Experiences with 4690 NFS server

3.3.5.1 Read/Write Block Size

The NFS client decides the maximum block size for reads and writes when it mounts the remote filesystem (drive). Some systems may attempt a block size of 8K bytes by default. 4690 NFS server cannot support a block size of 8K, however if larger block sizes are required above the typical 4K blocks, the 4690 server will support up to 8000 bytes (not 8192) for read/write block size.

3.3.5.2 Performance

In many cases, a typical NFS server (unix-based) will be running on a high-performance network server machine with multiple clients concurrently mounted to multiple filesystems. In addition, the high-performance server might be a dedicated file server in the network and therefore may have little or no other contention for its CPU.

Be aware that NFS clients are designed with this in mind, i.e. that there is a high-performance server available. It is therefore highly likely that a client (especially for large data transfers) will send multiple block-read requests almost simultaneously (if reading from the 4690 files) or transmit multiple block-write requests almost simultaneously (if writing to the 4690 files). In addition, most clients have relatively short timeout values if a response from the server is not received. When the timeout expires, a retransmission of the block-read or block-write occurs. This timeout/retransmission sequence may actually make the load worse on the potentially already loaded server. Performance of the 4690 NFS server can be degraded under these conditions, especially considering the 4690 controller may be executing many applications of higher priority than NFS and also that the machine characteristics, i.e. processor and disk, may be on different ends of the performance spectrum. For example, consider an RS/6000 NFS client and a 4690 store controller PC/AT NFS server whose processor, CPU, and disk speeds are significantly different.

There are two approaches you might take if you suspect that your NFS performance could stand room for improvement.

First, to help alleviate potential performance degradation, you should change the NFS read/write timeout for the mounted filesystem (4690 drive) to a higher value than the default. (Of course, trying the default first may be worthwhile to determine if there is cause for suspicion.) For example, on AIX V3.2 the NFS client timeout is 0.7 seconds. A recommended timeout of 7-10 seconds might help keep retransmissions low due to timer expiration.

Second, consider using ftp when large file transfer is required.

3.3.6 Telnet client

The telnet client support in 4690 TCP/IP allows a 4690 to logon to a remote system using the telnet protocol. The remote system must have a telnet server capable of supporting VT100 type terminals. The telnet client is a VT100 emulator and the program file name is C:\ADX_SPGM\ADXHSIVL.286.

Invocation of the 4690 telnet client is done from a 4690 command prompt using the ADXHSIVL command, or logical name VT100.

While in a telnet session, the escape sequence is CTRL-], i.e. hold down the Ctrl key and press the] (right bracket) key at the same time. This allows you to set/display different telnet operating parameters. You can also exit the emulator by typing 'quit' at the VT100 prompt.

3.3.7 BOOTP client - ADXHSIBL.286

The bootp client program (C:\ADX_SPGM\ADXHSIBL.286) is used to access TCP/IP host initialization data from a bootp server in the network. Specifically, it can be used to obtain the 4690 IP address, domain name, name server IP address, and network router address. The bootp client sends a broadcast request indicating its hardware (LAN adapter address) to a bootp server on the network and waits for a reply. If no reply is received from a server within 25 seconds, the client gives up and does not update any network initialization data.

When a server does reply, the BOOTP client will take the following actions:

1. Generate a batch file named C:\ADX_SDT1\ADXBPxxZ.BAT containing an ifconfig command that will configure the token-ring lan0 interface for the received host IP address and, optionally, the subnet mask if one was sent. In addition, if a network router (gateway) IP address was sent by the bootp server, a route command is also added to this batch file.
2. Update the resolv file with the name of the domain and IP address of the name server this host should use to resolve hostnames to internet addresses.

IMPORTANT!

The BOOTP client does not actually execute the ifconfig and route commands. It only builds the ADXBPxxZ.BAT batch file. That file must be run after BOOTP completes execution if network initialization is to occur.

An example of how to use the 4690 bootp client is the following:

1. Copy the ifconfig and route (if used) commands contained in the configuration batch file C:\ADX_SDT1\ADXIPxxZ.BAT to another batch file named C:\ADX_SDT1\ADXBPxxZ.BAT. (You do not need to copy the adxhsi2l command to ADXBPxxZ.BAT file.)
2. Delete the ifconfig and route (if used) commands from file ADXIP*xx*Z.BAT and replace with a single command which invokes the bootp client, i.e. adxhsibl. Follow that with a call to the bootp batch file ADXBP*xx*Z, where *xx* is the 2 character 4690 controller ID.

The two batch files would then look like this, for controller CC:

ADXIPCCZ.BAT contents:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%
% adxhsi2l                %
% adxhsibl                %
% adx_sdt1:adxhsibz       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

ADXBPCCZ.BAT contents:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ifconfig lan0 <ip_address> ...            %
% route add default <router_ip_address> ... %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Using this example, when the bootp server is available and does respond, the bootp client updates batch file ADXBPxxZ.BAT. It then returns control to the configuration batch file ADXIPxxZ.BAT which then invokes the newly created configuration.

However, in the event that the bootp server is unavailable and/or does not respond to the bootp client request in the timeout period, the 4690 would be able to initialize its interface based on current contents of file ADXBPxxZ.BAT.

3.3.7.1 Using bootp to update/create the SNMP trap destination file.

The 4690 bootp client can recognize a generic tag which specifies the host destination for SNMP traps. The generic tag is placed in the bootptab file on the server; it can be specified as Tnnn: where nnn is any integer between 128 and 254. The string associated with the generic tag must begin with the characters snmp: followed by the host name or IP address of the SNMP trap destination.

When the 4690 bootp client recognizes the generic tag and determines that it is being used to specify the SNMP trap destination, it creates or updates the SNMP trap destination file, C:\ADX_SDT1\ADXHSIDF.DAT, with the name or IP address of the host which follows the snmp: prefix in the tag string. This allows automatic setting of the trap destination host during network initialization.

3.3.8 BOOTP server - ADXHSIAL.286

The 4690 bootp server program name is C:\ADX_SPGM\ADXHSIAL.286. It services requests from BOOTP clients and supplies network initialization data which it gathers from the bootptab file which is named C:\ADX_SDT1\ADXHSIAF.DAT. Since it is a UDP-based server, it cannot be directly started from the inetd superserver. You can, however, configure it to be a 4690 background application which starts during controller IPL.

The bootptab file is a data file that contains the network initialization data for specific hosts, either based on hardware adapter address or host IP address. A sample bootptab is copied onto the controller during 4690 TCP/IP installation.

3.3.9 TFTP server - ADXHSITL.286

TFTP is a simple file transfer protocol. The TFTP server services requests from TFTP clients by transferring files to or from the clients. TFTP is mainly used for bootpstrapping medialess clients. FTP should continue to be used for normal file transfer purposes. In addition, the 4690 TFTP server supports multicast file transfer.

Since the TFTP server is a UDP-based server, it cannot be directly started from the inetd superserver. You can, however, configure it to be a 4690 background application which starts during controller IPL.

3.3.10 SNMP agent - ADXHSI1L.286

The 4690 Simple Network Management (SNMP) agent (server) is designed to operate with an SNMP network management station (client). Fundamentally, the SNMP network management station polls or queries the SNMP agent for information about the network, e.g number of UDP datagrams transmitted. The information is stored by the agent in a database called the Management Information Base, or MIB. The 4690 SNMP agent supports the MIB-II definition for managed objects (excluding the Exterior Gateway Protocol group) for GET and GET-NEXT requests. The setting (SET-request) of MIB objects is not supported. In addition to providing access to MIB-II network objects, the SNMP agent can asynchronously send TRAPs to the network management station whenever a significant network event occurs.

Currently there are two different types of TRAPs which can be sent by the SNMP agent:

- Cold-Start - notifies the network management station that this SNMP agent has just been started.
- Authentication Failure - indicated that an attempt was made by an unauthorized network management station to access this SNMP agent's MIB objects.

Since the SNMP agent is a UDP-based server, it cannot be started by the inetd superserver, however it can be configured as a 4690 background application that starts during controller IPL. See the IBM 4690 Store Systems User's Guide for information about configuring background applications.

3.3.10.1 Community Names File

Access to the 4690 MIB-II objects is only afforded to authorized network management stations. Those stations are defined in the community names file.

The 4690 SNMP agent uses an encrypted community names file to validate a network management station. That file is named C:\ADX_SDT1\ADXHSIEF.DAT. The encrypted file is built from an ascii text file created by the user. The ascii text file is named C:\ADX_SDT1\ADXHSIQF.DAT.

To build the encrypted community names file do the following:

1. create the ascii text community names file as file C:\ADX_SDT1\ADXHSIQF.DAT. (A sample file is copied onto the controller during 4690 TCP/IP installation.
2. Invoke the encryption program ADXHSI8L from a 4690 command line. This program assumes the input file is C:\ADX_SDT1\ADXHSIQF.DAT and creates the encrypted output file C:\ADX_SDT1\ADXHSIEF.DAT.

It is recommended that you build and maintain the ascii text community names file in a secure location. Only the encrypted community names file needs to exist on the 4690 controller.

IMPORTANT!

The encrypted community names file must exist in order for the 4690 SNMP agent to successfully initialize. If the SNMP agent cannot access the encrypted community names file, it will log a system message and terminate.

3.3.10.2 SNMP agent environment names

In addition to the community names file, three environment names must be defined to the 4690 system. These names correspond directly to MIB objects and are user-definable.

The names are the following:

syscont	- specifies who to contact in case of problems, for example, Joe Smith
sysloc	- specifies the physical location of this host, for example, Store 101CC
hostname	- specifies name of host on which you are running. This name must be able to be resolved to an IP address either by a name server or using the hosts file.

These three environment names are defined by one of two methods:
1) User-Defined Logical Filenames in 4690
   Controller Configuration or,
2) Listed in file C:\ADX_SDT1\ADXHSIZF.DAT.

The format of the SNMP environment names file, C:\ADX_SDT1\ADXHSIZF.DAT, is the following:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% syscont   Contact_Name     # comment      %
% hostname  Host_Name        # comment      %
% sysloc    System_Location  # comment      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A sample SNMP environment names file, C:\ADX_SDT1\ADXHSIZF.DAT, is copied onto the 4690 controller during 4690 TCP/IP installation. Note that precedence is given to the logical filename definition of the SNMP environment name before determining if the name is defined in the SNMP environment names file.

IMPORTANT!

The three SNMP environment names must exist on each machine in which the 4690 SNMP agent is to run either as user-defined logical filenames or within the SNMP environment names file (or some combination of both). However, the assignment for each name does not have to be unique across all machines, e.g. a syscont of Joe Smith may be the same on every machine.

3.3.10.3 SNMP Trap destination file

The SNMP trap destination file identifies the network management station to which traps will be sent. This file is named C:\ADX_SDT1\ADXHSIDF.DAT. A sample trap destination file is copied onto the controller during 4690 TCP/IP installation.

If the trap destination file does not exist, traps will not be sent.

3.3.10.4 MIB-II

Extension of the MIB objects can be accomplished using the SNMP Distributed Program Interface (SNMP DPI) for 4690. The SNMP DPI allows enterprise-specific MIB objects to be created and also provides an application the capability to generate traps. More information about the SNMP DPI can be found in Programming Interfaces section below.

3.3.11 Telnet server - ADXHSIIL.286

The 4690 telnet server can support a VT100, ANSITERM, or IBM 3151 telnet client. It is important to note the following characteristics of the 4690 telnet server:
1) Only ONE client can access the 4690 telnet server at a time.
2) Once logged on to the 4690, the client shares the main system
   keyboard and display. This operation is similar to the
   4690 Remote Operator capabilities.

For an alternative telnet server which does not have these limitations refer to the section below on "Enhanced Telnet Server".

3.3.11.1 Operation

A telnet entry should be included in the inetd data file, C:\ADX_SDT1\ADXHSIIF.DAT, so that the inetd superserver can start the telnet server when a client attempts a connection. The example inetd data file copied onto the controller during 4690 TCP/IP installation already contains a telnet entry. Note that the inetd superserver will start the telnet server automatically whenever a client issues a connection request.

3.3.11.2 Logging On and Off / Userids and Passwords

The telnet server uses the 4690 system userids and passwords to grant login access to telnet clients. For example, the default 4690 system userid is 99999999 with password of 99999999. When the telnet client is prompted for userid: and password:, the server will validate the input against the 4690 system maintained userids and passwords.

To log off normally, enter the telnet command mode on the client terminal using the telnet escape sequence (usually ctrl-right bracket) and type quit. Note that ctrl-d will not close the connection. In addition, telnet sessions which are idle can be automatically disconnected by the 4690 telnet server. The timeout for disconnection due to inactivity can be set by defining a user-defined logical filename of ADXTNDTO and assigning it a value (greater than 0) that corresponds to the number of idle minutes telnet server will wait before automatic disconnect. The default timeout is infinite and is used if the ADXTNDTO logical filename does not exist or is defined to an invalid value.

If a client attempts to logon to the server and someone else is already logged on, an option of killing the active telnet server process is provided. If you choose to kill the other process, the active session will be disconnected and you will receive a message telling you to try to log in again.

3.3.11.3 Keyboard

The 4690 Sys-Req function is mapped to the tilda-backquote key transmitted by the telnet client.

Other function keys transmitted from PS/2 style keyboards are known to operate consistent with function keys entered from the 4690 keyboard.

Note: Due to time delays in some network environments (i.e. WANs), the Telnet server may need to issue more than one read to get an entire function key sequence. For most networks (LANs), a 50ms delay is plenty of time for the entire key sequence to arrive. If you are running over a WAN or a slow network, you may need to define the logical name TNDESCTO to the number of milliseconds you want the Telnet server to wait for an entire function key sequence. The default is 50ms. Suggested values would be anywhere from 25 to 1000 milliseconds (1/4 to 1 full second).

3.3.11.4 Locking out the local keyboard

When logged in through the Telnet server, it is possible to lock the local keyboard. This allows only the remotely logged in user to have control of the console. The executable file is ADX_SPGM:ADXHSILL.286. The local keyboard can be locked by issuing the following command:

```
ADXHSILL LOCK
```

The local keyboard will be implicitly unlocked upon normal or abnormal termination of the Telnet server, or it can be unlocked explicitly with the following command:

```
ADXHSILL UNLOCK
```

Also, you can lock and unlock the local keyboard from any screen without using the ADXHSILL command directly. Typing Ctrl-B will (b)lock the local keyboard and typing Ctrl-U will un(b)lock it.

3.3.11.5 Terminfo/Termcap files

The 4690 telnet server does not use terminfo or termcap files. There are three basic terminal types supported (correspond to TERM environment variable in unix):

- vt100
- ansi
- ibm3151

Also, terminal types of aixterm, xterm, vt100-am are supersets of vt100 and are supported using the vt100 protocols. If an attempt is made to logon to the 4690 telnet server from an unknown terminal type, the connection will be closed and the client is sent a message indicating that the 4690 telnet server cannot support the terminal.

3.3.11.6 Log files

The 4690 telnet server is actually composed of two processes working together. Program ADXHSIIL.286 is the keyboard input process, and program ADXHSISL.286 is the screen output process. Both of these programs can log messages (information or otherwise) into files ADXHSIIL.LOG and ADXHSISL.LOG in subdirectory ADX_UPGM.

3.3.11.7 Telnet server considerations

"Real" vt100 terminals are 24 rows by 80 columns displays. Unfortunately, stuffing 25 rows into 24 rows is not possible, and no trade-off seems prudent in trying to decide which row to omit. Therefore, it is best if a 25 row vt100 emulator is used. Using a 24 row emulator will work, but display results appear unpredictable when the 25th row is received, e.g. sometimes it is displayed and row 1 scrolls off the display, other times row 25 is not displayed at all. Some terminal definitions of vt100-am (auto margins) are defined with 25 rows.

Most ansi terminal types are 25 row by 80 column displays. These seem to work well with 4690 telnet server. In addition, if the connection is made to the server from an ansi terminal type, the color attributes of fields (if you have a color display) will be sent to the terminal.

One disadvantage to ansi terminal types is that there appears to be no consistent definition of function keys and other special keys, e.g. PgUp, PgDn. If you have an ansi terminal type and the function keys are not working, we will try to support it. We will handle these requests on a case-by-case basis.

Included on the installation diskette is an ansi terminfo file for the AIX operating system on RS/6000. It is called ansiterm.ti and can be compiled using tic in /usr/lib/terminfo directory (must have root access to do this). You can use this terminfo file to define your terminal as ansi, i.e. export TERM=ansi. If your client is on a color display, use ansi to allow color information to be sent from the 4690 telnet server. VT100 terminal emulation is black-and-white only.

3.3.12 Enhanced Telnet server - ADXHSIUL.286

The 4690 enhanced telnet server supports VT220, VT100, ANSI and HFT. It may also support other terminal types, especially where they are similar to any of the above types.

Unlike the regular telnet server, the enhanced server allows multiple clients to be connected simultaneously. This is configured through the auxiliary console section of controller configuration.

To use this telnet server the 4690 Operating System must have 9630 or later maintenance applied.

3.3.12.1 Operation

A telnet entry should be included in the inetd data file, C:\ADX_SDT1\ADXHSIIF.DAT, so that the inetd superserver can start the telnet server when a client attempts a connection. The example inetd data file copied onto the controller during 4690 TCP/IP installation already contains an entry for the regular telnet server, as follows

```
telnet  tcp ADX_SPGM:ADXHSIIL.286
```

This should be changed for the enhanced telnet server as follows

```
telnet  tcp ADX_SPGM:ADXHSIUL.286
```

Note that the inetd superserver will start the telnet server automatically whenever a client issues a connection request.

3.3.12.2 Logging On and Off / Userids and Passwords

When a connection is made to the telnet server, the logon screen is presented, and you must logon as if you were at the main console, or another auxiliary console.

To log off normally, enter the telnet command mode on the client terminal using the telnet escape sequence (usually ctrl-]) and type quit.

Only the number of sessions configured using Controller Auxiliary Console configuration can be supported. If an additional client attempts to connect when there are already the maximum number of sessions in operation, then the connection is automatically refused.

From the Background Application Control menu you can see a separate copy of ADXHSIUL.286 operating for each connected session. It is also possible to terminate connections from this screen using F8 STOP.

By default the telnet server will request a device status report from the telnet client after a minute of inactivity. Most clients will automatically respond to this, and the server will keep the connection open. If after a further time interval there is no response, the server will terminate the connection on the assumption that the client has gone away.

If you should wish to change the time interval at which this request is sent, or disable it altogether, this can be done by defining the user logical name ADXTNDTO to the number of minutes you wish the interval to be, or to 0 to disable.

3.3.12.3 Keyboard

By default the backquote character (`) is mapped to the 4690 SysRq function but this can be changed by defining a User Logical File Name of 'ADXTNDSR' which contains the ASCII value for the single character you would rather be mapped to 4690 SysRq. Also, if you are using a VT220 client, you may u se ALT+F12 as an alternative way to generate a 4690 SysRq.

For example, use Controller Configuration, User Logical File Names, and define the name ADXTNDSR to have the value 1. Now when you telnet into the 4690 controller CTRL+A will take you to the system keys menu.

Note: Due to time delays in some network environments (i.e. WANs), the telnet server may need to issue more than one read to get an entire function key sequence. For most networks (LANs), a 50ms delay is plenty of time for the entire key sequence to arrive. If you are running over a WAN or a slow network, you may need to define the logical name TNDESCTO to the number of milliseconds you want the telnet server to wait for an entire function key sequence. The default is 50ms. Suggested values would be anywhere from 250 to 1000 milliseconds (1/4 to 1 full second).

3.3.12.4 Locking out the local keyboard

The enhanced telnet server operates in the same fashion as an auxiliary console. It does not lock out the local keyboard.

3.3.12.5 Terminfo/Termcap files

The 4690 enhanced telnet server does not use terminfo or termcap files. It has been tested with the following terminal types:

- vt220
- vt100
- ansi
- HFT

Other terminal types which are similar may also work. The enhanced telnet server does not reject connections based upon the terminal type. A valid terminal type must be selected at the client.

3.3.12.6 Log files

The enhanced 4690 telnet server does not log messages to any files. Information messages will be displayed on the background application control screen in the MESSAGE field for the corresponding telnet server.

3.3.12.7 Telnet server considerations

See the comments on the regular telnet server about 24 row displays. Furthermore, during testing several telnet clients were found not to correctly support insert and delete escape sequences. This is most noticeable in 4690 Command Mode where the output on the client window when using the delete key may not accurately reflect what has actually happened on the 4690 system.

The IBM OS/2 Warp telnet client correctly supports both 25 row displays, and the necessary insert and delete escape sequences. Furthermore, since it supports VT220, Alt+F12 can be used to emulate 4690 SysRq.

The telnet client in IBM TCP/IP for DOS and Windows 2.1.1.4 will work in ANSI mode; however, it presently does not interpret the insert and delete escape sequences correctly. It is hoped this will be fixed in a future CSD.

Presently all support in the enhanced telnet server is black and white.

3.3.13 LPR client - ADXHSIRL.286

The lpr client program is invoked from a 4690 command line as ADXHSIRL or logical name LPR, and accepts the same input parameters as the lpr command. All required command parameters must be explicitly provided to the 4690 lpr program. Type lpr -? for a list of valid command line parameters.

3.3.14 Rexec client - ADXHSIXL.286

The rexec client program is invoked from a 4690 command line as ADXHSIXL or logical name REXEC, and accepts the same input parameters as the rexec program. All required command parameters must be explicitly provided to the 4690 rexec program. Type rexec -? for a list of valid command line parameters.

3.4 Error messages

The following list of messages can be logged by components of 4690 TCP/IP. The message text and explanation is provided below.

The TCP/IP driver can log the following system messages:

```
W875 TCP/IP INITIALIZATION FAILURE
    B4/S005/E001  FN=nnnnnnnn RC=rrrrrrrr
```

This message will be logged if there is an error executing the the initialization batch file, ADX_SDT1:ADXIPxxZ.BAT. Unique Data is formatted as:

FN Indicates filename of initialization file that cannot be executed.

RC Indicates return code from the driver.

```
W876 TCP/IP TOKEN RING NETWORK ERROR
    B4/S005/Exxx  FN=nnnnnnnn RC=rrrrrrrr
```

This message will be logged if the TCP/IP driver is notified of a token-ring network error by the token-ring driver. Unique data is formatted as:

EXXX Events 002 thru 005 indicate the following:

- o 002 - Token-Ring adapter check.

  This indicates that the token-ring adapter has failed (usually hardware). This is a non-recoverable error.

- o 003 - Set_User_Appendage failed.

  This indicates that the TCP/IP driver cannot register itself with the token-ring network driver.

- o 004 - Open Service_Access_Point (SAP) failed.

  This indicates that the Open SAP request issued by the TCP/IP driver to the token-ring driver failed. No TCP/IP network data can be sent or received on the token-ring when the Open SAP request fails.

- o 005 - Receive_Modify failed.

  This indicates that the TCP/IP driver cannot register its receive data location with the token-ring driver. No TCP/IP data can be sent or received.

FN Indicates network operation in progress when error occurred.

RC Indicates return code from the driver.

```
  W877 TCP/IP CRITICAL COMMUNICATIONS FAILURE
       B4/S005/Exxx  RC=rrrrrrrr
```

This message indicates that the executing TCP/IP driver has detected an internal critical error. This error is non-recoverable. Unique data is formatted as:

EXXX Events 006 and 007 indicate the following:

- o 006 - Out of memory buffers.

  This indicates that the TCP/IP driver has run out of internal memory buffers and cannot obtain any more.
- o 007 - Memory Allocation failure.

  This indicates that the TCP/IP driver cannot obtain system memory when it attempts to allocate the storage for its own use.

RC Indicates return code from the driver.

```
  W878 TCP/IP FILE ACCESS ERROR
       B4/S005/E001  FN= ssssssss RC=rrrrrrrr
```

This message indicates that a component of TCP/IP (driver or appl) cannot access a file or successfully complete certain functions. FN is the name of the file or function detecting the error. RC is the operating system or driver return code.

The SNMP agent can log the following system messages:

```
  W879 SNMP TRAP - COLD STARTED
       B4/S003/E001
```

This message will be logged when the SNMP agent is started. It indicates that the Cold-Start trap has been sent to the SNMP network monitor. This is a severity 4 message.

```
  W880 SNMP TRAP - AUTHENTICATION FAILURE
       B4/S003/E002  IP ADDR=xxx.xxx.xxx.xxx
```

This message will be logged if the SNMP agent detects that an unauthorized SNMP network monitor has attempted to access its MIB variables. The Authentication-Failure trap is sent to the agent's authorized network monitor and this severity 3 message is logged. The unqiue data identifies the internet address of the unauthorized host in dotted-decimal format.

```
  W881 SNMP CRITICAL MEMORY FAILURE
```

B4/S003/E003 RC=rrrrrrrr

This message indicates that the SNMP agent cannot allocate sufficient memory to execute. The agent program will log this severity 2 message and exit.

RC	Indicates return code from the agent application.

W882 SNMP COMMUNITY NAME FILE NOT FOUND
B4/S003/E004 FN=ssssssss

This message indicates that the SNMP agent could not locate the community name file in encrypted form. This file should be generated using ADXHSI8L.286 from the un-encrypted community name file ADX_SDT1:ADXHSIQF.DAT. The encrypted file output is named ADX_SDT1:ADXHSIEF.DAT and must be build before the SNMP agent is started. The agent program will log this severity 2 message and exit.

W883 SNMP LOGICAL NAME NOT FOUND
B4/S003/E004 NAME=ssssssss

This message indicates that the SNMP agent cannot find an entry in the 4690 configuration for the logical name identified in the NAME field of the unique data. The agent program will log this severity 2 message and exit.

Certain applications can log the following system messages:

W978 TCPIP APPLICATION EXPERIENCED UNEXPECTED FAILURE
B4/S019/Eddd RC=rrrrrrrr CALL=bbcc INFO=ssssssssssss

This message indicates that the 4690 TCP/IP application experienced an non-recoverable error before exiting.

W979 TCPIP APPLICATION DETECTED USER ERROR
B4/S019/Eddd RC=rrrrrrrr CALL=bbcc INFO=ssssssssssss

This message indicates that the 4690 TCP/IP application detected a user error. The application may or may not continue.

W980 TCPIP APPLICATION CONTINUES WITH ERROR
B4/S019/Eddd RC=rrrrrrrr CALL=bbcc INFO=ssssssssssss

This message indicated that the 4690 TCP/IP application detected a non-fatal system error. The application continues.

Eddd ddd is the decimal value of an 8-bit binary number ppppeeee.
pppp is the program number of the 4690 TCP/IP application that logged this message.

eeee is the event number.

RC=rrrrrrrr
: rrrrrrrr is a 32-bit number in hex. This could be the return code a of a function call that failed.

CALL=bbcc
: bbcc is a 16-bit number is hex. The upper 8-bits identifies the type of a function call and the lower 8-bits identifies the specific function call.

INFO=ssssssssssss
: ssssssssssss is any additional information that does not have fixed format.

4.0 Commands

This section describes the commands and subcommands used with TCP/IP for 4690OS. The commands are listed alphabetically.

4.1 adxhsibl(bootp)

The **bootp** command finds the internet address for a client host from a server. The **bootp** command causes the local host hardware address (which has been uniquely assigned to the interface hardware adapter) to broadcast over the local TCP/IP network.

4.1.1 Syntax

```
►►──bootp──►◄
```

Displaying bootp Help

```
►►──bootp── -?──►◄
```

-?	Displays the list of parameters and their meanings.

4.2 adxhsial(bootpd)

The **bootpd** command starts the BOOTP server. The BOOTP server receives the hardware address and looks for a match in the ADX_SDT1:ADXHSIAF (BOOTPTAB) file. When the server finds a match, it sends an internet address, a subnetwork, and other information to the client. The boot protocol works over a single network, but not through a router. A BOOTPTAB table must be configured with the hardware and internet address pairs plus network parameters. A sample ADXHSIAF.DAT (BOOTPTAB) file resides in the ADX_SDT1 directory.

The services file in the ADX_SDT1 directory must include:

```
bootps   67/udp   #bootp server
bootpc   68/udp   #bootp client
```

4.2.1 Syntax

```
               ┌──────┐
►►──bootpd──▼─ -d─┴────►◄
```

-d Displays debugging information, such as hardware address, on the server terminal. You can specify more than one -d parameter to display any debug information. Each additional -d increases the amount of debugging information displayed. You can specify -d up to five times.

4.3 adxhsi7l(finger)

The **finger** command displays information about the users on a remote host.

4.3.1 Syntax

```
            ┌──────────────────────┐
►►──finger──▼──┬──────────┬──host──┴──►◄
               └──user@───┘
```

Displaying finger Help

```
►►──finger── -?──►◄
```

user@	Specifies the user name to be queried on the remote host. This parameter is optional; however, if you specify a user, the host must be followed by an @. Without *user@*, the **finger** command displays all users currently logged on at the host. With *user@*, only detailed information about the user will be displayed. **Note:** A space is required between multiple *user@host* entries.
host	Specifies a host from which you request user information. This parameter is required.
-?	Displays help information.

4.4 adxhsigl(ftp)

The **ftp** command transfers files between your workstation and a remote host that is running an FTP server.

4.4.1 Syntax

```
►►──ftp──┬──────┬──┬──────┬──┬──────┬──┬──────┬──┬──────┬──┬──────────────────┬──►◄
         └─ -d──┘  └─ -g──┘  └─ -n──┘  └─ -i──┘  └─ -v──┘  └─host──┬────────┬─┘
                                                                  └─port───┘
```

Displaying ftp Help

```
►►──ftp── -?──►◄
```

-d	Enables debugging.
-g	Disables file name globbing (extension).
-n	Specifies that FTP does not look in the NETRC file. You must enter the user ID and password for the remote host.
-i	Disables interactive prompting.
-v	Toggles verbose mode on. When verbose mode is on, FTP displays all responses from the remote server.
host	Specifies the remote host to which you are connecting.
port	Specifies the destination port to which you are connecting. If *port* is not specified, you are connected by default to the well-known FTP port as specified in the services file.
-?	Displays help information.

4.4.2 Subcommands

The following subcommands can be used in the FTP command shell.

4.4.2.1 ! Subcommand

The **!** subcommand enters the 4690OS command processor while keeping the FTP command shell in resident memory. This subcommand can also invoke the 4690 command shell to issue a single command and immediately return to FTP.

```
►►──!──┬─────────────────────────────┬──►◄
       └─command──┬──────────────┬───┘
                  └─parameters───┘
```

command	Specifies the 4690OS command that you want to issue
parameters	Specifies any parameters required by the 4690OS command

If you enter the **!** subcommand without parameters, you will enter a 4690OS command shell. To return to the FTP command shell from the 4690OS command shell, type `exit`.

4.4.2.2 $ Subcommand

The $ subcommand issues FTP macros created with the **macdef** subcommand.

```
►►──$──┬──────────────────────────────────┬──►◄
       └─macro_name──┬──────────────┬─────┘
                     └─parameters───┘
```

macro_name	Specifies the macro name.
parameters	Specifies any parameters to be passed to the macro. You can specify more than one parameter.

4.4.2.3 account Subcommand

The **account** subcommand sends the account information to the remote host. You can issue the **account** subcommand with the abbreviation **ac**.

```
►►──account──┬────────────────┬──►◄
             └─account_name───┘
```

account_name	Specifies the account name on the remote host. If you do not specify the account name, FTP prompts you for it in non-echo mode.

4.4.2.4 append Subcommand

The **append** subcommand adds a file on your workstation to a file on the remote host. **append** transfers a file from your workstation to the remote host and adds it to the specified file or to a file of the same name. You can issue the **append** subcommand with the abbreviation **ap**.

```
►►──append──┬──────────────────────────────────────┬──►◄
            └─source_file──┬────────────────────┬──┘
                           └─destination_file───┘
```

source_file	Specifies the name of the file on your workstation that is to be transferred and appended to a file on the remote host. If you do not specify this value, FTP will prompt you for it.
destination_file	Specifies the name of the remote host file to which you want to append the *source_file*.

4.4.2.5 ascii Subcommand

The **ascii** subcommand sets the file transfer type to ASCII. Using the **ascii** subcommand is the same as issuing the **type** subcommand with the ascii parameter. You can issue the **ascii** subcommand with the abbreviation **as**.

```
►►──ascii──►◄
```

4.4.2.6 bell Subcommand

The **bell** subcommand toggles the bell sound on and off. If the **bell** is on, it will sound after each file transfer is complete. You can issue the **bell** subcommand with the abbreviation **be**.

```
►►──bell──►◄
```

4.4.2.7 binary Subcommand

The **binary** subcommand sets the file transfer type to binary. The **binary** file transfer type is useful for image transfers, such as executable files. Using the **binary** subcommand is the same as issuing the **type** subcommand with the binary parameter. You can issue the **binary** subcommand with the abbreviation **bi**.

```
►►──binary──►◄
```

4.4.2.8 bye Subcommand

The **bye** subcommand ends the FTP session and leaves the FTP command shell. The **bye** subcommand closes any open connection. Using **bye** is the same as issuing the **quit** subcommand. You can issue the **bye** subcommand with the abbreviation **by**.

```
►►──bye──►◄
```

4.4.2.9 cd Subcommand

The **cd** subcommand changes the working directory on the remote host.

```
►►──cd──┬──────────────────┬──►◄
        └─directory_name───┘
```

directory_name	Specifies the name of the file directory on the remote host that becomes the current working directory for file transfer tasks. If you do not specify this value, FTP will prompt you for it.

4.4.2.10 cdup Subcommand

The **cdup** subcommand changes the current working directory on the remote host to the current parent directory on the remote host. You can issue the **cdup** subcommand with the abbreviation **cdu**.

```
►►──cdup──►◄
```

4.4.2.11 close Subcommand

The **close** subcommand ends the FTP connection but does not leave the FTP command shell. Using the **close** subcommand is the same as issuing the **disconnect** subcommand. You can issue the **close** subcommand with the abbreviation **cl**.

```
►►──close──►◄
```

4.4.2.12 cr Subcommand

The **cr** subcommand strips the carriage return character from a carriage return or line feed sequence when receiving records during ASCII-type file transfers. Issue the **cr** subcommand to set carriage return stripping off. Issue the **cr** subcommand again to set carriage return stripping on.

```
►►──cr──►◄
```

4.4.2.13 debug Subcommand

The **debug** subcommand sets debugging on or off. The initial setting is for **debug** to be off. You can issue the **debug** subcommand with the abbreviation **deb**.

```
►►──debug──►◄
```

4.4.2.14 delete Subcommand

The **delete** subcommand deletes a file from the remote host. You can issue the **delete** subcommand with the abbreviation **del**.

```
►►──delete──┬───────────┬──►◄
            └─file_name─┘
```

file_name Specifies the name of the file to be deleted from the remote host. If you do not specify this value, FTP will prompt you for it.

4.4.2.15 dir Subcommand

The **dir** subcommand displays a listing of the files and directories in the directory on the remote host.

```
►►──dir──┬────────────────────────────────────────┬──►◄
         │                     ┌─ - ──────────┐   │
         └─┬─path─────────┬────┴──────────────┴───┘
           ├─pattern──────┤    └─file_name────┘
           └─path──pattern┘
```

path Specifies a path to a different directory, a specific file, or both.

pattern Specifies the pattern of the file names to be listed. Patterns are any combination of ASCII characters. The following two characters have special meaning:

- * The asterisk means that any character or group of characters can occupy that position in the pattern.
- ? The question mark means that any single character can occupy that position in the pattern.

file_name Specifies the name of a file on your workstation to which you want to write the output. If you specify a file name, you must also specify a path or pattern. If you do not specify a file name or if you specify a hyphen (-), the output is displayed on the screen.

4.4.2.16 disconnect Subcommand

The **disconnect** subcommand ends the FTP connection but does not leave the FTP command shell. Using the **disconnect** subcommand is the same as issuing the **close** subcommand. You can issue the **disconnect** subcommand with the abbreviation **dis**.

```
►►──disconnect──►◄
```

4.4.2.17 form Subcommand

The **form** subcommand sets the file transfer format. You can issue the **form** subcommand with abbreviation **f**.

```
►►──form──format──►◄
```

format Specifies the file transfer format (only nonprint is supported).

4.4.2.18 get Subcommand

The **get** subcommand transfers a file from a remote host to your workstation. The current settings for the **type** and **struct** subcommands are used with **get**.

```
►►──get──┬─────────────────────────────────────────┬──►◄
         └─source_file──┬──────────────────────┬───┘
                        ├─destination_file─────┤
                        ├─con──────────────────┤
                        └─prn──────────────────┘
```

source_file Specifies the name of the file on the remote host that is to be transferred to your workstation. If you do not specify this value, FTP will prompt you for it.

destination_file
Specifies the name given to the source file when it is stored on your workstation. If *destination_file* is not specified, the *source_file* name is used and changed, if necessary, to conform to 4690OS file-naming conventions. If the name of the file being received is the same as a file that already

exists on your workstation, your existing file is overwritten by the incoming file, unless **runique** is set to on. If **runique** is set to on, a unique file name is created for the incoming file, and your existing file is unchanged. If you are unsure whether **runique** is set to on, use the **status** subcommand to check the setting.

con	Specifies that the file is to be displayed on your screen.
prn	Specifies that the file is to be sent to a printer or special device.

4.4.2.19 glob Subcommand

The **glob** subcommand toggles the file name expansion for the **mdelete**, **mget**, and **mput** subcommands. You can use patterns when **glob** is on; this is the initial setting. You can issue the **glob** subcommand with the abbreviation **gl**.

```
►►──glob──►◄
```

4.4.2.20 hash Subcommand

The **hash** subcommand toggles hash mark printing. When **hash** is on, FTP displays hash marks (#) to indicate data transfer progress. You can issue the **hash** subcommand with the abbreviation **ha**.

```
►►──hash──►◄
```

4.4.2.21 help Subcommand

The **help** subcommand displays help information for the FTP command shell. Using the **help** subcommand is the same as issuing the **?** subcommand. You can issue the **help** subcommand with the abbreviation **he**.

```
►►──help──┬──────────────┬──►◄
          └─subcommand─┘
```

subcommand   Specifies the subcommand for which you are requesting help.

4.4.2.22 lcd Subcommand

The **lcd** subcommand changes or displays the current working directory on your workstation. You can issue the **lcd** subcommand with the abbreviation **lc**.

```
►►──lcd──┬──────────────┬──►◄
         └─local_path─┘
```

local_path   Specifies the name of a directory on your workstation that you want to make your current directory. If you do not specify a local path, the name of the current working directory on your workstation is displayed.

4.4.2.23 ls Subcommand

The **ls** subcommand provides a listing of the files in the current working directory on the remote host.

```
►►──ls──┬───────────────────────────────────────┬──►◄
        │                  ┌─ - ───────┐          │
        └─┬─path─────────┬─┴─file_name─┴─┘
          ├─pattern──────┤
          └─path──pattern─┘
```

path   Specifies a path to a different directory, specific file, or both.

pattern   Specifies the pattern of the file names to be listed. Patterns are any combination of ASCII characters. The following two characters have special meaning:

- *   The asterisk means that any character or group of characters can occupy that position in the pattern.
- ?   The question mark means that any single character can occupy that position in the pattern.

file_name   Specifies the name of a file on your workstation to which you want to write the output. If you specify a file name, you must also specify a path or pattern. If you do not specify a file name, or if you specify a hyphen (-), the output is displayed on the screen.

4.4.2.24 macdef Subcommand

The **macdef** subcommand defines a macro name and begins macro input mode. Macros remain defined until you issue the **close** subcommand. FTP limits you to 16 macros and a total 4096 characters in all defined macros. You can issue the **macdef** subcommand with the abbreviation **ma**.

```
►►──macdef──┬──────────────┬──►◄
            └─macro_name───┘
```

macro_name Specifies the macro name.

4.4.2.25 mdelete Subcommand

The **mdelete** subcommand deletes a group of files from the remote host. The **glob** subcommand must be set to on to use this subcommand. If you are unsure whether **glob** is set to on, use the **status** subcommand to check the setting. You can issue the **mdelete** subcommand with the abbreviation **mde**.

```
             ┌────────────┐
►►──mdelete──▼┬──────────┬┴──►◄
              └─pattern──┘
```

pattern Specifies the name pattern of the files to be deleted from the remote host. Patterns are any combination of ASCII characters. The following two characters have special meaning:

* The asterisk means that any character or group of characters can occupy that position in the pattern.

? The question mark means that any single character can occupy that position in the pattern.

You can specify more than one pattern with the **mdelete** subcommand. If you do not specify this value, FTP will prompt you for it.

4.4.2.26 mget Subcommand

The **mget** subcommand transfers a group of files from the remote host to your workstation. The **glob** subcommand must be set to on to use this subcommand. If you are unsure whether **glob** is set to on, use the **status** subcommand to check the setting. You can issue the **mget** subcommand with the abbreviation **mg**.

```
          ┌────────────┐
►►──mget──▼┬──────────┬┴──►◄
           └─pattern──┘
```

pattern Specifies the name pattern of the files to be transferred from the remote host to your workstation. Patterns are any combination of ASCII characters. The following two characters have special meaning:

* The asterisk means that any character or group of characters can occupy that position in the pattern.

? The question mark means that any single character can occupy that position in the pattern.

You can specify more than one pattern with the **mget** subcommand. If you do not specify this value, FTP will prompt you for it.

4.4.2.27 mkdir Subcommand

The **mkdir** subcommand creates a directory on the remote host. You can issue the **mkdir** subcommand with the abbreviation **mk**.

```
►►──mkdir──┬───────────────┬──►◄
           └─directory─┘
```

directory Specifies the path to the directory that you are creating. If you do not specify a directory, FTP prompts you for the path.

4.4.2.28 mode Subcommand

The **mode** subcommand sets the file transfer mode. You can issue the **mode** subcommand with the abbreviation **mo**.

```
►►──mode──mode_name──►◄
```

mode_name Specifies the file transfer mode (only stream is supported).

4.4.2.29 mput Subcommand

The **mput** subcommand transfers a group of files from your workstation to a remote host. The **glob** subcommand must be set to on to use this subcommand. If you are unsure whether **glob** is set to on, use the **status** subcommand to check the setting. You can issue the **mput** subcommand with the abbreviation **mp**.

```
          ┌─────────────┐
►►──mput──▼─────────────┴──►◄
           └─pattern─┘
```

pattern Specifies the name pattern of the files to be copied from your workstation to the remote host. Patterns are any combination of ASCII characters. The following two characters have special meaning:

* The asterisk means that any character or group of characters can occupy that position in the pattern.

? The question mark means that any single character can occupy that position in the pattern.

You can specify more than one pattern with the **mput** subcommand. If you do not specify this value, FTP will prompt you for it.

4.4.2.30 nmap Subcommand

The **nmap** subcommand toggles file name mapping. The initial setting for **nmap** is off. File name mapping occurs with the **put** and **mput** subcommands and with the **get** and **mget** subcommands when they are issued without a local file name. You can issue the **nmap** subcommand with the abbreviation **nm**.

```
►►──nmap──┬─────────────────────────┬──►◄
          └─inpattern──outpattern─┘
```

inpattern	Specifies the character pattern of the file names.
outpattern	Specifies the character pattern of the remote host file names.

4.4.2.31 ntrans Subcommand

The **ntrans** subcommand toggles file name character translation. The initial setting for **ntrans** is off. If you do not specify *inchars* with the **ntrans** subcommand, the current status of **ntrans** is displayed. You can issue the **ntrans** subcommand with the abbreviation **nt**.

```
►►──ntrans──┬─────────────────────────┬──►◄
            └─inchars──┬────────────┬─┘
                       └─outchars─┘
```

inchars	Specifies the character for the file name on the workstation
outchars	Specifies the character for the remote file name

4.4.2.32 open Subcommand

The **open** subcommand establishes a connection to a remote host. You can issue the **open** subcommand with the abbreviation **o**.

```
►►──open──┬──────────────────┬──►◄
          └─host──┬────────┬─┘
                  └─port─┘
```

host	Specifies the remote host to which you want to connect. If you do not specify *host*, FTP prompts you for a host.
port	Specifies the destination port to which you are connecting. If you do not specify a port, you are connected by default to the well-known FTP port as specified in the services file.

4.4.2.33 prompt Subcommand

The **prompt** subcommand toggles interactive prompting. The initial setting for **prompt** is on. You can issue the **prompt** subcommand with the abbreviation **prom**.

```
►►──prompt──►◄
```

4.4.2.34 proxy Subcommand

The **proxy** subcommand forwards subcommands to another server to allow logical connections between two servers; this connection allows file transfers between the servers. You can issue the **proxy** subcommand with the abbreviation **prox**.

```
►►──proxy──┬────────────┬──►◄
           └─subcommand─┘
```

subcommand	Specifies an FTP subcommand. If you do not specify this value, FTP will prompt you for it.

4.4.2.35 put Subcommand

The **put** subcommand transfers a file from your workstation to a remote host. Using the **put** subcommand is the same as issuing the **send** subcommand. You can issue the **put** subcommand with the abbreviation **pu**.

```
►►──put──┬─────────────────────────────────────┬──►◄
         └─source_file──┬──────────────────┬───┘
                        ├─destination_file─┤
                        ├─con──────────────┤
                        └─prn──────────────┘
```

source_file	Specifies the name of the file on your workstation that is to be transferred to the remote host. If you do not specify this value, FTP will prompt you for it.
destination_file	Specifies the name given to the *source_file* when it is stored on the remote host. If *destination_file* is not specified, the *source_file* name is used and changed, if necessary, to conform to 4690OS file-naming conventions. If the name of the file being received is the same as a file that already exists on the remote host, the existing file is overwritten by the incoming file, unless **sunique** is on. If **sunique** is on, a unique file name is created for the incoming file, and the existing file is unchanged.
con	Specifies that the file is to be displayed on the server screen.
prn	Specifies that the file is to be sent to a destination printer or special device.

4.4.2.36 pwd Subcommand

The **pwd** subcommand displays the name of the current working directory on the remote host. You can issue the **pwd** subcommand with the abbreviation **pw**.

```
►►──pwd──►◄
```

4.4.2.37 quit Subcommand

The **quit** subcommand ends the FTP session and exits the FTP command shell. Using the **quit** subcommand is the same as issuing the **bye** subcommand. You can issue the **quit** subcommand with the abbreviation **qui**.

```
►►──quit──►◄
```

4.4.2.38 quote Subcommand

The **quote** subcommand sends the specified text to the remote host verbatim.

Note: Using **quote** with commands that involve data transfers can produce unpredictable results.

You can issue the **quote** subcommand with the abbreviation **quo**.

```
►►──quote──┬──────────┬──►◄
           └─argument─┘
```

argument	Specifies the information to send to the remote host. If you do not specify this value, FTP will prompt you for it.

4.4.2.39 recv Subcommand

The **recv** subcommand transfers a file from a remote host to your workstation. The current settings for the **type** and **struct** subcommands are used with **recv**. You can issue the **recv** subcommand with the abbreviation **rec**.

```
►►──recv──┬─────────────────────────────────────┬──►◄
          └─source_file──┬────────────────────┬─┘
                         ├─destination_file─┤
                         ├─con──────────────┤
                         └─prn──────────────┘
```

source_file	Specifies the name of the file on the remote host that is to be transferred to your workstation. If you do not specify this value, FTP will prompt you for it.
destination_file	Specifies the name given to the *source_file* when it is stored on your workstation. If *destination_file* is not specified, the *source_file* name is used and changed, if necessary, to conform to 4690OS file-naming conventions. If the name of the file being received is the same as a file that already exists on your workstation, your existing file is overwritten by the incoming file, unless **runique** is set to on. If **runique** is set to on, a unique file name is created for the incoming file, and your existing file is unchanged. If you are unsure whether **runique** is set to on, use the **status** subcommand to check the setting.
con	Specifies that the file is to be displayed on your screen.
prn	Specifies that the file is to be sent to a printer or special device.

4.4.2.40 remotehelp Subcommand

The **remotehelp** subcommand identifies the services and their respective syntax specifications. You can issue the **remotehelp** subcommand with the abbreviation **rem**.

```
►►──remotehelp──┬───────────┬──►◄
                └─command───┘
```

command Identifies the host command for which you want to view help information.

4.4.2.41 rename Subcommand

The **rename** subcommand renames a file on the remote host. You can issue the **rename** subcommand with the abbreviation **ren**.

```
►►──rename──┬───────────────────────────┬──►◄
            └─oldname──┬─────────────┬──┘
                       └─newname─────┘
```

oldname Specifies the current name of a file in the working directory of the remote host.

newname Specifies the new name for the file. If the file name specified by the *newname* already exists, an error message is displayed.

If you do not specify either of these parameters, FTP will prompt you for them.

4.4.2.42 reset Subcommand

The **reset** subcommand clears the reply queue and resets the command reply sequencing between the local processor and the remote server. You can issue the **reset** subcommand with the abbreviation **res**.

```
►►──reset──►◄
```

4.4.2.43 rmdir Subcommand

The **rmdir** subcommand removes a directory from the remote host. You can issue the **rmdir** subcommand with the abbreviation **rm**.

```
►►──rmdir──┬─────────────┬──►◄
           └─directory───┘
```

directory Specifies the directory that you want to remove from the remote host. If you do not specify a directory, FTP prompts you for one.

4.4.2.44 runique Subcommand

The **runique** subcommand toggles the creation of unique file names for local destination files during the **get**, **mget**, and **recv** operations. The initial setting for **runique** is off.

If **runique** is set to off, FTP will overwrite existing files. If **runique** is set to on, FTP will not overwrite existing files. You can issue the **runique** subcommand with the abbreviation **ru**.

```
►►──runique──►◄
```

4.4.2.45 send Subcommand

The **send** subcommand transfers a file from your workstation to a remote host. Using the **send** subcommand is the same as issuing the **put** subcommand.

```
►►──send──┬──────────────────────────────────────┬──►◄
          └─source_file──┬────────────────────┬─┘
                         ├─destination_file───┤
                         ├─con────────────────┤
                         └─prn────────────────┘
```

source_file	Specifies the name of the file on your workstation that is to be transferred to the remote host. If you do not specify this value, FTP will prompt you for it.
destination_file	Specifies the name given to the *source_file* when it is stored on the remote host. If *destination_file* is not specified, the *source_file* name is used and changed, if necessary, to conform to 4690OS file-naming conventions. If the name of the file being received is the same as a file that already exists on the remote host, the existing file is overwritten by the incoming file, unless **sunique** is on. If **sunique** is on, a unique file name is created for the incoming file, and the existing file is unchanged.
con	Specifies that the file is to be displayed on the server screen or console.
prn	Specifies that the file is to be sent to a destination printer or special device.

4.4.2.46 sendport Subcommand

The **sendport** subcommand toggles the use of FTP PORT commands. The initial setting for **sendport** is to use PORT commands. You can issue the **sendport** subcommand with the abbreviation **sendp**.

```
►►──sendport──►◄
```

4.4.2.47 site Subcommand

The **site** subcommand sends service-specific information to a remote host. You can issue the **site** subcommand with the abbreviation **si**.

```
►►──site──parameters──►◄
```

parameters Specifies the service-specific information. To identify services and their respective syntax specifications, issue the **remotehelp** subcommand.

4.4.2.48 status Subcommand

The **status** subcommand displays the following information:

- Connection status
- Transfer mode
- Transfer type
- Form
- Structure
- Flags

You can issue the **status** subcommand with the abbreviation **sta**.

```
►►──status──►◄
```

4.4.2.49 struct Subcommand

The **struct** subcommand specifies the file transfer structure. You can issue the **struct** subcommand with the abbreviation **str**.

```
►►──struct──┬─file───┬──►◄
            └─record─┘
```

file Specifies a file structure that is a continuous sequence of data bytes. This structure is supported for both ASCII and binary file transfer types.

record Specifies a file structure that is not currently supported by TCP/IP.

4.4.2.50 sunique Subcommand

The **sunique** subcommand toggles the creation of unique file names for the destination files during **put**, **mput**, and **send** operations. The initial setting of **sunique** is off.

If **sunique** is off, FTP will overwrite existing files. If **sunique** is on, FTP will not overwrite existing files. You can issue the **sunique** subcommand with the abbreviation **su**.

```
►►──sunique──►◄
```

4.4.2.51 trace Subcommand

The **trace** subcommand toggles the flag that determines whether transmitted packets are traced. The initial setting for **trace** is off. You can issue the **trace** subcommand with the abbreviation **tr**.

```
►►──trace──►◄
```

4.4.2.52 type Subcommand

The **type** subcommand sets the file transfer type. You can issue the **type** subcommand with the abbreviation **ty**.

```
►►──type──┬──────────┬──►◄
          ├─ascii────┤
          └─binary───┘
```

The setting that you specify will remain in effect until you either change it or quit FTP. If you specify **type** without a parameter, FTP will display a message indicating the current transfer type.

ascii	Specifies the file transfer type as ASCII. ASCII is the initial setting for FTP's file transfer type. Specifying the **type** subcommand with the ascii parameter is the same as issuing the **ascii** subcommand.
binary	Specifies the file transfer type as binary (image). Specifying the **type** subcommand with the binary parameter is the same as issuing the **binary** subcommand.

4.4.2.53 user Subcommand

The **user** subcommand identifies you to the remote host. You can issue the **user** subcommand with the abbreviation **u**.

```
►►──user──┬──────────────────────────┬──►◄
          └─userid──┬─────────────┬──┘
                    └─password────┘
```

userid	Specifies your name to the remote host.
password	Specifies the password associated with your user ID.

4.4.2.54 verbose Subcommand

The **verbose** subcommand toggles the flag that determines whether responses from the FTP server are displayed. The initial setting for **verbose** is on. You can issue the **verbose** subcommand with the abbreviation **v**.

```
►►──verbose──►◄
```

4.4.2.55 ? Subcommand

The **?** subcommand displays help information for the FTP command shell. Using the **?** subcommand is the same as issuing the **help** subcommand.

```
►►──?──┬──────────────┬──►◄
       └─subcommand─┘
```

subcommand Specifies the subcommand for which you are requesting help.

4.5 adxhsifl(ftpd)

The **ftpd** command uses the FTPD.EXE program to start the FTP server. It runs as a task until you shut down the server.

4.5.1 Syntax

```
►►──ftpd──►◄
```

4.6 adxhsi3l(ifconfig)

The **ifconfig** command assigns an address to a network interface and configures the network interface parameters.

4.6.1 Syntax

```
                          ┌─metric 0─┐  ┌─mtu 1500─┐                      ┌─ -trailers─┐
►►──ifconfig──interface───┼──────────┼──┼──────────┼──┤ Address Family ├──┼────────────┼────────►
                          └─metric n─┘  └─mtu n────┘                      └─trailers───┘

   ┌─arp───┐  ┌─bridge───┐  ┌─snap───┐
►──┼───────┼──┼──────────┼──┼────────┼──┬─────────┬──broadcast broadcast_address──────────►
   └─ -arp─┘  └─ -bridge─┘  └─ -snap─┘  └─ -allrs─┘

   ┌─ -802.3─┐  ┌─icmpred───┐
►──┼─────────┼──┼───────────┼──►◄
   └─802.3───┘  └─ -icmpred─┘
```

Address Family:

```
                                                ┌─up───┐
├──┬────┬──┬──────────────────────────────────┬──┼──────┼──┬──────────────┬──┤
   └─af─┘  └─address──┬──────────────────┬───┘  └─down─┘  └─netmask mask─┘
                      └─dest_address─────┘
```

interface	The name of the interface you are configuring (lan0 or lo).
	Note: Specifying lo creates a local loopback interface. The local loopback interface bypasses the network interface drivers to provide a direct internal connection back to the internet protocol support. For example, if you type `ifconfig lo 2.2.2.2`, you can use the address 2.2.2.2 as a local loopback.
metric *n*	Sets the metric for the interface to *n*. The value for *n* is a number (1–15). The default is 0 (directly connected). This routing metric is used by the Routing Information Protocol (RIP).
	The higher the metric, the greater the number of hops to the destination network or host.
mtu *n*	Sets the maximum transmission unit of the interface to *n*. The value *n* represents a number. The default is 1500. The maximum is 2000.
af	Specifies the name of the address family supported.
	Because an interface can receive transmissions in different protocols, with each protocol requiring a separate naming scheme, you must specify the address family. However, specifying the address family can change the interpretation of the remaining parameters. The only address family currently supported is inet.
address	Specifies the address assigned to a particular interface in the standard dotted-decimal notation.
dest_address	Specifies the address of the correspondent on the receiving end of a point-to-point link.
up	Enables an interface after the interface has been marked as being down with an ifconfig statement.
	Interfaces are automatically marked as being up when the first address is set on an interface.
down	Marks an interface as being down. When an interface is marked as being down, the system does not attempt to transmit messages through that interface. In some cases, the reception of messages is also disabled.
	This action does not automatically disable routes using the interface.
netmask *mask*	Specifies how much of the internet address to reserve for use as a subnetwork address. This parameter is used for networks only.

For example, the subnetwork capability of TCP/IP divides a single network into multiple logical networks. An organization can have a single internet network address that is known to users outside the organization, but it can configure its internal network into different departmental subnetworks. The subnetwork portion of an internet address is then divided into a subnetwork number and a host number, for example:

network_number subnet_number host_number

where:

network_number	Is the network portion of the internet address
subnet_number	Is the subnetwork portion of the local address
host_number	Is the host portion of the local address

The *mask* value includes the network portion of the local address and the subnetwork portion, which is taken from the host field of the address. The mask can be specified as a single hexadecimal number with a leading `X'0'` or with a dotted-decimal notation address.

The mask contains a 1 for each bit position of the 32-bit address that is to be used for the network and subnetwork and a 0 for the bit position to be used by the host. The mask should contain at least the standard network portion, but the bits of the mask do not have to be contiguous. The subnetwork field should be contiguous with the network portion.

-trailers
: Disables trailer-link level encapsulation. This is the default.

trailers
: Requests the use of trailer-link level encapsulation when sending.

For example, if a network interface supports trailers, the system, when possible, encapsulates outgoing messages, which minimizes the number of memory-to-memory copy operations that the receiver must perform.

On networks that support ARP, this parameter indicates that the system should request that other systems use trailers when sending to this host. Trailer encapsulations are sent to other hosts that have made such requests.

arp
: Enables ARP in mapping between network level addresses and physical or station addresses. This is the default.

ARP is currently used for mapping between internet addresses and Ethernet addresses or IBM token-ring addresses.

-arp
: Disables ARP.

bridge
: Enables routing field support. This is the default.

-bridge
: Disables routing field support.

snap
: Sends token-ring headers with the extended snap format. This is the Institute of Electrical and Electronic Engineers (IEEE) standard and is necessary to communicate with workstations using the extended snap format, such as AIX*. This is the default.

-snap
: Does not send token-ring headers with the extended snap format.

-allrs
: Sets the token-ring broadcast indicator to single-route broadcast. The default is all-routes broadcast. See the *IBM LAN Technical Reference*, SC30-3383, for more information about all-routes and single-route broadcasting.

broadcast *broadcast_address*
: Specifies the address to use to represent broadcasts to the network. The default broadcast address is an internet address with a local address that has a value of all 1s.

-802.3
: Disables IEEE 802.3. Enables Ethernet DIX 2. This is the default.

802.3	Enables IEEE 802.3.
icmpred	Allows TCP/IP to add routes obtained by ICMP redirects. This is the default.
-icmpred	Prevents TCP/IP from adding routes obtained by ICMP redirects.

The **ifconfig** command displays the current configuration for a network interface when only an interface is supplied. If an address family is specified using *af*, **ifconfig** reports only the details specific to that address family.

To receive help for the command syntax, use the **ifconfig** command alone, without specifying an interface, address, or parameter.

4.7 adxhsi9l(inetd)

The **inetd** command enables a super server that allows you to start more than one server from a single 4690OS session and to use a specific server when needed. The **inetd** command supports the following servers:

- adxhsifl (FTPD)
- adxhsiil(TELNETD)

Note: INETD will start the servers listed in the adxhsiif.dat file located in subdirectory ADX_SDT1.

4.7.1 Syntax

```
►►──inetd──►◄
```

4.8 adxhsirl(lpr)

The **lpr** command allows you to transfer the contents of a file on your workstation to a network host that provides print spooling services.

4.8.1 Syntax

```
►►──lpr──┬────┬──┬────┬──┬─────────────────┬──►◄
         └─-f─┘  └─-n─┘  ├─┬─────────────┬─┤
                         │ └─-p printer──┘ │
                         ├─┬────────────┬──┤
                         │ └─-s server──┘  │
                         └─filename────────┘
```

Displaying lpr Help

```
►►──lpr── -?──►◄
```

-f	When the print server is running on a UNIX** system, the -f parameter formats the file using the UNIX **pr** command. When the print server is running under 4690OS, LPR passes the file through unchanged.
-n	Displays no messages unless an error occurs.
-p *printer*	Specifies the name of the printer to which the file is sent. Optionally, the LPR_PRT logical name may be defined to specify a default printer to use when -p is not specified.
-s *server*	Specifies the name or internet address of a network host with print spooling capabilities. Optionally, the LPR_SRV logical name may be defined to specify a default print server to use when -s is not specified.
	If a print server is not specified with the **lpr** command, LPR displays an error message and ends.
filename	Specifies the name of the file to be sent to the printer.
-?	Displays help information.

4.9 adxhsi6l(netstat)

The **netstat** command displays the network status of the local workstation. The **netstat** command provides information about TCP connections, user datagram protocol (UDP) and internal protocol (IP) statistics, memory buffers, and sockets.

4.9.1 Syntax

```
                    ┌──────┐
►►──netstat──-──▼──────┴──►◄
                  ├─ m─┤
                  ├─ t─┤
                  ├─ u─┤
                  ├─ i─┤
                  ├─ s─┤
                  ├─ r─┤
                  ├─ c─┤
                  ├─ n─┤
                  ├─ a─┤
                  └─ p─┘
```

Displaying netstat Help

```
►►──netstat── -?──►◄
```

More than one parameter can be specified with the **netstat** command. Do not enter spaces between the parameters when you use the **netstat** command with multiple parameters. The **netstat** command ignores any entry after a space.

-m	Information about memory buffer usage
-t	Information about TCP connections
-u	Information about UDP statistics
-i	Information about IP statistics
-s	Information about sockets
-r	Routing tables and corresponding network interfaces
-c	Information about internet control message protocol (ICMP) statistics
-n	Information about LAN interfaces
-a	The address of the network interfaces
-p	Contents of the address resolution protocol table
-?	Help information

4.10 adxhsinl(nfsd)

The **nfsd** command starts the NFS (adxhsinl) server. Verify that PORTMAP (adxhsipl) is running before starting the NFS server.

4.10.1 Syntax

```
►►──nfsd──►◄
```

4.11 adxhsi5l(ping)

The **ping** command sends an echo request to a remote host to determine if the host is accessible.

4.11.1 Syntax

```
►►──ping──┬──────┬──┬──────┬──┬──────┬──host──┬─────────────────────────────┬──►◄
          └─ -d──┘  └─ -r──┘  └─ -v──┘        └─data_size──┬──────────────┬─┘
                                                           └─npackets─────┘
```

Displaying ping Help

```
►►──ping── -?──►◄
```

-d	Starts the socket-level debugging process.
-r	Bypasses the routing tables and sends packets directly to a host on an attached network. If the host is not on a directly-connected network, PING cannot make a connection. This parameter can be used to ping a local host through an interface that no longer has a route through it.
-v	Specifies verbose output.
host	Specifies the IP address or host name of the remote host to which you want to send the echo request.
data_size	Sets the number of data bytes for the echo request (the default number of data bytes is 56, with an additional 8-byte header attached).
npackets	Sets the number of echo requests that are sent to the remote host.

These parameters are position dependent; you cannot specify the number of packets without specifying the data size.

Note: If you do not specify *npackets*, the echo request is sent continuously until one of the following actions stops the echo request:

- Pressing the Ctrl and C keys simultaneously
- Pressing the Ctrl and Break keys simultaneously
- Closing the task

-?	Displays help information.

4.12 adxhsipl(portmap)

The **portmap** command starts a protocol to define a network service that permits clients to look up the port number of any remote program supported by the server.

4.12.1 Syntax

```
►►──portmap──►◄
```

4.13 adxhsixl(rexec)

The **rexec** command issues a command on a remote host. The **rexec** command sends a single command to the remote host.

4.13.1 Syntax

```
►►──rexec──host──┬────────────────┬──┬────────────────┬──┬──────┬──┬──────┬──command──►◄
                 └─ -l loginname──┘  └─ -p password───┘  └─ -k──┘  └─ -n──┘
```

Displaying rexec Help

```
►►──rexec── -?──►◄
```

host	Specifies the remote host on which the command is to be issued.
-l *loginname*	Specifies the user ID on *host*. If you do not specify a login name, the values in the ADX_SDT1:ADXHSIGF.DAT file are used.
-p *password*	Specifies the password that is associated with the login name. If you do not specify a password, the values in the ADX_SDT1:ADXHSIGF.DAT file are used. If the ADX_SDT1:ADXHSIGF.DAT file does not provide the password value, **rexec** prompts you for the password. You can enter the password in a nonecho mode.
-k	Ignores the local keyboard input. This is helpful for running noninteractive input, especially from a batch file.
-n	Specifies not to use the ADX_SDT1:ADXHSIGF.DAT file for automatic login.
command	Specifies the command to be issued on the remote host. The command must be in the syntax used by the remote host.
-?	Displays help information.

4.14 adxhsi4l(route)

The **route** command is used to modify the network routing tables. Use the **route** command only if you are an experienced TCP/IP user.

4.14.1 Syntax

Adding to the Route Table

```
►►──route──┬──────┬──┬──────┬──add──┬─net────┬──┬─destination─┬──router──metric──►◄
           └─ -f──┘  └─ -h──┘       ├─subnet─┤  └─default─────┘
                                    └─host───┘
```

Deleting from the Route Table

```
►►──route──┬──────┬──┬──────┬──delete──┬─net────┬──┬─destination─┬──router──┬────────┬──►◄
           └─ -f──┘  └─ -h──┘          ├─subnet─┤  └─default─────┘          └─metric─┘
                                       └─host───┘
```

Route Table Help

```
►►──route──?──►◄
```

-f	Empties the routing tables of all network and subnet route entries. If this is used in conjunction with another parameter, the tables are emptied before the other parameters take effect.
-h	Empties the routing tables of all host route entries. If this is used in conjunction with another parameter, the tables are emptied before other parameters take effect.
add	Adds a route. If you specify the add parameter, *metric* is required.
delete	Deletes a route.
net	Specifies that a network is to be added or deleted.
subnet	Specifies that a subnet is to be added or deleted.
host	Specifies that a host is to be added or deleted.
destination	Specifies the internet address of the host, network, or subnet.
default	Specifies all destinations not defined with another routing table entry.
router	Specifies the internet address of the next hop in the path to the destination.
metric	Specifies the number of hops to the destination. The *metric* parameter is required for adding to the route table.
?	Displays help information.

4.15 adxhsirl(rpcinfo)

The **rpcinfo** command makes an RPC call to the RPC server and reports the status of the server, which is registered and operational with the Portmapper.

4.15.1 Syntax

rpcinfo for a Host

```
                    ┌─local_host─┐
►►──rpcinfo── -p ───┼────────────┼──►◄
                    └─host───────┘
```

rpcinfo for a Host Using UDP

```
►►──rpcinfo──┬──────────────┬── -u host prognum──┬─────────┬──►◄
             └─ -n portnum──┘                    └─versnum─┘
```

rpcinfo for a Host Using TCP

```
►►──rpcinfo──┬──────────────┬── -t host prognum──┬─────────┬──►◄
             └─ -n portnum──┘                    └─versnum─┘
```

rpcinfo for a Broadcast to Hosts Using UDP

```
►►──rpcinfo──┬──────┬──prognum──versnum──►◄
             └─ -b──┘
```

-p *host* Queries the Portmapper about the specified host and prints a list of all registered RPC programs. If the host is not specified, the system defaults to the local host name.

-n *portnum*
Specifies the port number to be used for the -t and -u parameters. This value replaces the port number that is given by the Portmapper.

-u *host prognum versnum*
Sends an RPC call to procedure 0 of *prognum* and *versnum* on the specified host using UDP and reports whether a response is received.

-t *host prognum versnum*
Sends an RPC call to procedure 0 of *prognum* and *versnum* on the specified host using TCP and reports whether a response is received.

-b *prognum versnum*
Sends an RPC broadcast to procedure 0 of the specified *prognum* and *versnum* using UDP and reports all hosts that respond.

4.16 adxhsi1l(snmpd)

The **snmpd** command starts the SNMP agent. The **snmpd** command runs as a task until you shut down the server.

4.16.1 Syntax

```
►►──snmpd──►◄
```

4.17 adxhsitl(tftpd)

The **tftpd** command starts the Internet Trivial File Transfer (TFTP) server as defined in RFC1350. While the RFC1350 TFTP server unicasts files to a single client lockstep, the MTFTP server multicasts files to multiple clients, lockstepped with a single acknowledging client. The MTFTP server is started IF AND ONLY IF either the -m or the -n options are used.

The TFTP server operates at the port indicated by the *tftp* service description in *services*. If the port number is not present in *services*, the default (Assigned Numbers RFC) of 69 is used.

The MTFTP service names are:

mtftps - MTFTP Server listening port

mtftpc - MTFTP Client destination port

If the ports are absent from *services*, the default values of mtftps = 75 and mtftpc = 76 are used. In addition, the MTFTP server requires a configuration file that specifies the IP multicast address for each file that is expected to be multicasted. The configuration file has the same format as *bootptab*, except that only the *td*, *hd*, *bf*, and the MTFTP tags are recognized. For each file the fields must be specified in the same entry (without the *tc* tag).

The 4690 TFTP implementation allows at most one file transfer at one time. In order to support bootstrapping of multiple clients, the 4690 TFTP also allows concurrent transfer of boot images by caching parts of the images. This way one boot image can be transferred to multiple clients at the same time, allowing them to boot concurrently. The files allowed for concurrent transfer are those found in the configuration file.

Tftpd can be invoked at the command prompt like any other regular command, or defined as a background process. The program automatically detects whether it was invoked at the command prompt or the background and automatically selects the appropriate mode.

The use of TFTP does not require an account or password on the remote system. Due to the lack of authentication information, the client must specify the full path. "/../" as parts of filenames are discarded. The client is allowed to create new files or write to existing files unless the *-o* flag is used. Each directory on the path of the filename must already exist. The *-p* flag is used to further restrict access. The files requested by the clients must be in one of the paths specified by the flag. *Tftpd* can transfer only one file at a time.

All requests to the MTFTP server are regarded as read requests of binary files. MTFTP supports only sending of binary files for bootstrapping purposes. The multicast address used to transfer a file is determined by the MTFTP tag and the bootfile name in the configuration file. If the last character in the request file name is numeric, the last character is not matched against the last character in the configuration file, and the multicast address defined in the MTFTP tag is incremented by the value of the numeric character. The resulting IP multicast address is used to transfer the boot file. For example, if the configuration has file "bootimg1" with multicast address "225.2.10.0", a request for "bootimg4" will cause the file "bootimg4" to be multicast to the address "225.2.10.4".

If more than one server is set up as an MTFTP server, the multicast address used by each file must be unique. Each *mtftpd* still multicasts only one file at a time.

4.17.1 Syntax

```
►►──tftpd──┬───────────────────┬──┬─────────────────┬──┬────────────┬──┬──────────┬──┬────┬──►
           └─ -b num-of-pages ─┘  └─ -f config-file ─┘  └─ -d debug ─┘  └─ -g hops ─┘  └─ -m ─┘

►──┬────┬──┬────┬──┬────────────────┬──┬─────────┬──┬────┬──┬────┬──┬──────────────┬──►◄
   └─ -n ─┘  └─ -o ─┘  └─ -p secure-dir ─┘  └─ -s size ─┘  └─ -u ─┘  └─ -v ─┘  └─ -w wait-time ─┘
```

-b *number-of-pages*	Sets the number of 32K pages used for caching for concurrent TFTP transfer. The default and mininum number is 2, and 10 is max.
-d *debug-level*	Sets the *debug-level* variable that controls the amount of debugging messages generated. For example, *-d4* or *-d 4* will set the debugging level to 4. For compatibility, omitting the numeric parameter (i.e. just -d) will simply increment the debug level by one. Levels above 4 do not generate more messages than level 4.
-f *config-file*	Sets the configuration file to be used by TFTPD for concurrent transfer of bootfiles, and also by MTFTPD. The default is *bootptab* (ADX_SDT1:ADXHSIAF.DAT). Both physical and logical filenames may be specified.
-g *hops*	Sets the number of gateway hops between the MTFTP server and clients. By default, MTFTP file transfer cannot be routed through a gateway.
-m	Starts the MTFTP server, in addition to the TFTP server.
-n	Starts the MTFTP server, instead of the TFTP server. This flag overrides both the -u and -v flags.
-o	Prohibits overwriting of existing files.
-p *secure-dirs*	Enables security. Limit TFTP accesses to the *secure-dir* directories. Up to five directories are allowed. The directory names are case-sensitive.
-s *size*	Increases the transfer size from the default of 512 to *size*. *Size* should be a multiple of 16. The maximum is 1456. This option applies only to the MTFTP server.
-u	Allow TFTP server for concurrent and non-concurrent transfer of boot files defined in the bootptab. TFTP transfers of files absent from the bootptab are also allowed. This flag is overridden by the -n flag.
-v	Only allow TFTP server for concurrent transfer of boot files defined in the bootptab. TFTP transfers of files absent from the bootptab are disabled. This flag is overridden by the -n flag.
-w *wait-time*	Delays the start of an MTFTP transfer by *wait-time* seconds after a request is received. The default is 0.

4.18 ADXHSIVL(VT100)

The **VT100** command logs you on to a remote host, emulating a VT100.

If you do not specify a host on the **VT100** command, you enter the Telnet command shell. In the command shell, you can establish the operating environment and designate the host and port to which you want to connect.

5.0 Sockets

This chapter describes the C socket API provided with TCP/IP for 4690OS. Use the socket routines to interface with the TCP, UDP, ICMP, and IP protocols. These socket routines let a program communicate across networks with other programs. For example, you can use socket routines when you write a client program that communicates with a server program running on another computer.

To use the sockets, you must know C language programming. For more information about C sockets, see the *IBM AIX Operating System Technical Reference, Volume 1*.

5.1 Programming with Sockets

The 4690OS socket API provides a standard interface to the transport and internetwork layer interfaces of TCP/IP for 4690OS. 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.

5.1.1 Socket Programming Concepts

Before programming with the sockets API, consider the concepts in this section.

5.1.1.1 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 operating system. It is represented by an integer called a socket descriptor.

The socket interface provides applications with a network interface that hides the details of the physical network.

5.1.1.2 Socket Types

The socket types are defined in the <SYS\SOCKET.H> header file.

The stream socket (SOCK_STREAM) interface defines a reliable connection-oriented service. Data is sent without errors or duplication and is received in the same order as it is sent. 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 the File Transfer Protocol (FTP).

The datagram socket (SOCK_DGRAM) 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 32 768). No disassembly and reassembly of packets is performed. An example of an application that uses datagram sockets is the Network File System** (NFS**.)

The raw socket (SOCK_RAW) 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 or for gaining access to some of the more advanced facilities of an existing protocol.

You can extend the socket interface and therefore, define new socket types to provide additional services. An example of this is the transaction-type socket defined for interfacing to the Versatile Message Transfer Protocol (VMTP).[4] Transaction-type sockets are not supported by TCP/IP for 4690OS.

Because socket interfaces isolate you from the communication functions of the various protocol layers, the interfaces are largely independent of the underlying network. In the 4690 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 might change, but the socket interface will stay the same. For example, stream sockets may eventually interface to the International Standards Organization (ISO) Open System Interconnection (OSI) transport class 4 protocol. This means that applications will not have to be rewritten as underlying protocols change[5].

5.1.1.2.1 Guidelines for Choosing Socket Types: If you are communicating with an existing application, you must use the same protocols as the existing application. For example, if you interface to an application that uses TCP, you must use stream sockets. For new applications, consider the following factors:

- Consider 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 may be acceptable if the application does not require reliability, or if the application implements the reliability on top of the socket's interface. The tradeoff is the increased performance available over stream sockets.
- Consider performance. The overhead associated with reliability, flow control, packet reassembly, and connection maintenance degrades the performance of stream sockets so that they do not perform as well as datagram sockets.
- Consider the amount of data to be transferred. Datagram sockets limit the amount of data transferred but stream sockets do not. If you send less than 2048 bytes at a time, use datagram sockets. As the amount of data in a single transaction increases, use stream sockets.

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

5.1.1.3 Address Family

An address family defines different styles of address or communication domain. All hosts in the same address family understand and use the same scheme of address socket endpoints. TCP/IP for 4690OS supports one address family: AF_INET. The AF_INET domain defines address in the internet domain. AF_INET is also referred to as a PF_INET. Both are equivalent (PF stands for Protocol Family). The address families are defined in the <SYS\SOCKET.H> header file.

5.1.1.4 Socket Address

The *sockaddr* structure in the <SYS\SOCKET.H> header file defines a socket address. It has two fields, as shown in the following example:

[4] David R. Cheriton and Carey L. Williamson, "VMTP as the Transport Layer for High-Performance Distributed Systems," *IEEE Communications*, June 1989, Vol. 27, No. 6.

[5] This does not imply an IBM statement of direction.

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

The *sa_family* field contains the address family. It is AF_INET for the internet 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.

5.1.1.4.1 Addressing within an Internet Domain: A socket address in an internet address family comprises four fields: the address family (AF_INET), an internet address, a port, and a character array. The following *sockaddr_in* structure, which is in the <NETINET\IN.H> header file, defines the structure of an internet socket address.

```
struct in_addr
{
        u_long s_addr;
};

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

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 zeros.

5.1.1.5 Internet Address

An internet address is a 32-bit quantity that represents a network interface. Every internet address within an administered AF_INET domain must be unique. A common misunderstanding is that every host must have only one internet address. In fact, a host has as many internet addresses as it has network interfaces. For more information about internet address formats, see *Internetworking With TCP/IP Volume I: Principles, Protocols, and Architecture*.

5.1.1.6 Ports

The system software uses a port to differentiate between different applications using the same protocol (TCP or UDP). It is an additional qualifier that the software uses 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 more information, see the ADX_SDT1:ADXHSISF.DAT file.

5.1.1.7 Network Byte Order

You usually specify ports and addresses to calls using the network-byte ordering convention. Network byte order is also known as *big endian byte* ordering, as in Motorola** microprocessors (compared with *little endian byte* ordering in Intel** microprocessors). Using network byte ordering for data exchanged between hosts allows hosts using unique architectures to exchange address information. See pages 94, 96, and 98 for examples of using the htons() call to put ports into network byte order. For more information about network byte order, see: 5.4.1, “accept()” on page 107, 5.4.2, “bind()” on page 110, 5.4.27, “htonl()” on page 149, 5.4.28, “htons()” on page 150, 5.4.38, “ntohl()” on page 165, and 5.4.39, “ntohs()” on page 166.

Note: The sockets 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).

5.1.2 How to Apply Socket Calls

With a few socket calls, you can write a powerful network application as in the following steps and examples:

1. First, you initialize the application with sockets using the sock_init() call, as shown in Figure 1. For a more detailed description, see 5.4.57, “sock_init()” on page 194.

```
int rc;
int sock_init();
⋮
rc = sock_init();
```

Figure 1. An Application Uses the sock_init() Call

   The code fragment shown in Figure 1 initializes the process with the socket library and checks whether INET.SYS is running.

2. Next, you must get a socket descriptor for the application using the socket() call, as shown in Figure 2. For a more detailed description, see: 5.4.58, “socket()” on page 195.

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

Figure 2. An Application Uses the socket() Call

   The code fragment shown in Figure 2 allocates a socket descriptor *s* in the internet address family. 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. TCP/IP for 4690OS supports one address family: AF_INET. The *type* parameter is a constant that specifies the type of socket, which can be SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW. The *protocol* parameter is a constant that specifies the protocol to use. The parameter is ignored unless *type* is set to SOCK_RAW. If the *protocol* field is set to 0, the system selects the default protocol number for the domain and socket type requested. If successful, socket() returns a positive integer socket descriptor.

3. Once an application has a socket descriptor, it can explicitly bind() a unique name to the socket, as shown in the example in Figure 3. For a more detailed description, see 5.4.2, “bind()” on page 110.

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

/* clear the structure so that the sin_zero field is clear */
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));
```

Figure 3. An Application Uses the bind() Call

For a server to be able to listen() for connections on a stream socket, or recvfrom() on a datagram socket, the server must first bind() a socket to a specific address family, port, and interface. 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. Figure 3 shows two useful network utility routines:

- inet_addr() takes an internet address in dotted decimal form and returns it in network byte order. For a more detailed description, see 5.4.29, “inet_addr()” on page 151.
- htons() takes a port number in host byte order and returns the port in network byte order. For a more detailed description, see 5.4.28, “htons()” on page 150.

Figure 4 shows how the bind() call on the server side uses the network utility routine getservbyname() to find a well-known port number for specific service from the ADX_SDT1:ADXHSISF.DAT file. It also shows wildcard value INADDR_ANY. If a host has several network addresses (multihomed host), messages sent to any of the addresses should be deliverable to a socket.

```
int rc;
int s;
struct sockaddr_in myname;
int bind(int s, struct sockaddr_in name, int namelen);
struct servent *sp;
⋮
sp = getservbyname("login","tcp");  /* get application specific */
                                    /* well-known port          */

/* clear the structure to be sure the sin_zero field is clear */
memset(&myname,0,sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = INADDR_ANY;
myname.sin_port = sp->s_port;
⋮
rc = bind(s,(struct sockaddr *)&myname,sizeof(myname));
```

Figure 4. A bind() Call Uses the getservbyname() Call

4. 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 shown in Figure 5.

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

Figure 5. An Application Uses the listen() Call

The listen() call tells the TCP/IP software that the server is ready to accept connections and that a maximum of five connection requests can be queued for the server. The system ignores additional requests. For a more detailed description, see 5.4.36, “listen()” on page 162.

5. Clients using stream sockets call connect() to start a connection request as shown in Figure 6.

```
int s;
struct sockaddr_in servername;
int rc;
int connect(int s, struct sockaddr *name, int namelen);
⋮
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 6. An Application Uses the connect() Call

The connect() call tries to connect socket *s* to the server with name *servername*. This could be the server used in the previous bind() example. The caller optionally blocks until the server accepts the connection. On successful return from the server, the system associates socket *s* with the connection to the server. For a more detailed description, see 5.4.4, “connect()” on page 114.

```
int s;
struct sockaddr_in servername;
char *hostname = "serverhost";
int rc;
int connect(int s, struct sockaddr_in *name, int namelen);
struct servent *sp;
struct hostent *hp;
⋮
sp = getservbyname("login","tcp");  /* get application specific */
                                    /* well-known port          */

hp = gethostbyname(hostname);

/* clear the structure so that the sin_zero field is clear */
memset(&servername,0,sizeof(servername));
servername.sin_family = AF_INET;
servername.sin_addr.s_addr = *((u_long *)hp->h_addr);
servername.sin_port = sp->s_port;
⋮
rc = connect(s,(struct sockaddr *)&servername,sizeof(servername));
```

Figure 7. An Application Uses the gethostbyname() Call

Figure 7 shows an example of a network utility routine, the gethostbyname() call, to find out the internet address of *serverhost* from the name server or the ADX_SDT1:ADXHSIHF.DAT file.

6. Servers using stream sockets accept a connection request with the accept() call as shown in Figure 8.

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

Figure 8. An Application Uses the accept() Call

If connection requests are not pending on socket *s*, the accept() call optionally blocks the server. When the server accepts a connection request on socket *s*, it returns the name of the client and length of the client name, 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 more detailed description, see 5.4.1, “accept()” on page 107.

7. Clients and servers have many calls from which to choose for data transfer. The readv(), writev(), and send() calls can be used only on sockets that are in the connected state. The sendto() and recvfrom() calls can be used at any time. Figure 9 illustrates the use of send().

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

Figure 9. An Application Uses the send(). Calls

Figure 9 also shows an application sending data on a connected socket and receiving data in response. You can use the *flags* field to specify additional options to send(), such as sending out-of-band data.

8. If the socket is not in a connected state, additional address information must be passed to sendto() and may be optionally returned from recvfrom(). An example of the use of the sendto() and recvfrom() calls is shown in Figure 10.

```
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
struct sockaddr_in to;
struct sockaddr from;
int addrlen;
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 s;
⋮
to.sin_family = AF_INET;
to.sin_addr   = inet_addr("129.5.24.1");
to.sin_port   = htons(1024);
⋮
bytes_sent = sendto(s, data_sent, sizeof(data_sent), 0,
                    &to, sizeof(to));
⋮
addrlen = sizeof(from); /* must be initialized */
bytes_received = recvfrom(s, data_received,
   sizeof(data_received), 0, &from, &addrlen);
```

Figure 10. 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. See 5.4.43, “recvfrom()” on page 172 and 5.4.50, “sendto()” on page 184 for more information about these additional parameters. You usually use sendto() and recvfrom() for datagram sockets and send() for stream sockets.

9. Use the writev() and readv() calls to scatter and gather data. Scattered data can be located in multiple data buffers. The writev() call gathers the scattered data and sends it. The readv() call receives data and scatters it into multiple buffers.

10. 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.
11. In addition to select(), applications can use the ioctl() call to help perform asynchronous (nonblocking) socket operations.
12. You deallocate a socket descriptor, s, with the soclose() call. For a more detailed description, see 5.4.59, “soclose()” on page 198. Figure 11 shows the soclose() call.

```
⋮
/* close the socket */
soclose(s);
⋮
```

Figure 11. An Application Uses the soclose() Call

5.1.2.1 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.

When you make a connection, it exists until you close the socket. During the connection, either data is delivered or TCP/IP returns an error code.

See Figure 12 for the general sequence of calls to follow for most socket routines using TCP sockets.

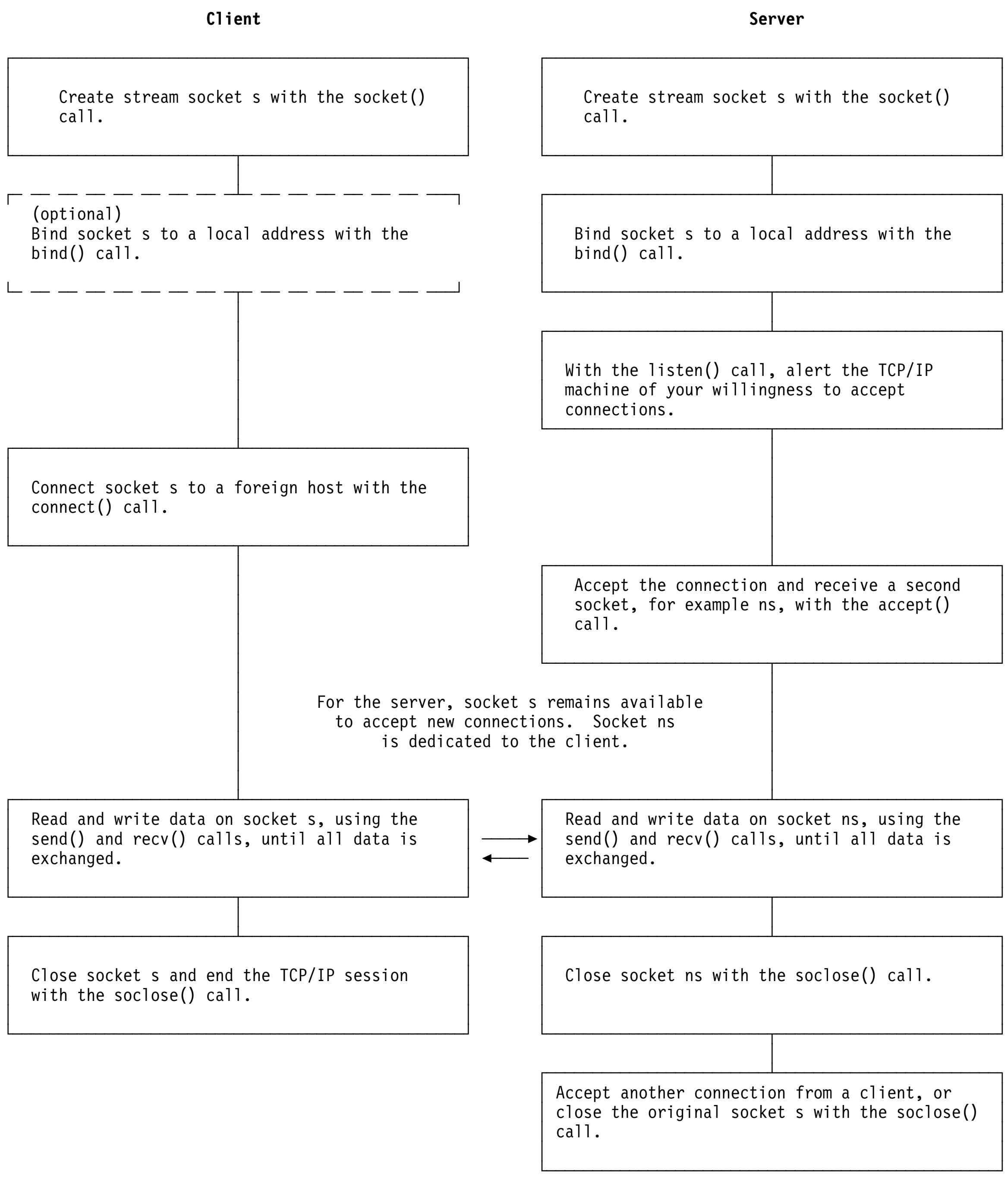

Client
Server
Create stream socket s with the socket() call.
Create stream socket s with the socket() call.
(optional) Bind socket s to a local address with the bind() call.
Bind socket s to a local address with the bind() call.
With the listen() call, alert the TCP/IP machine of your willingness to accept connections.
Connect socket s to a foreign host with the connect() call.
Accept the connection and receive a second socket, for example ns, with the accept() call.
For the server, socket s remains available to accept new connections. Socket ns is dedicated to the client.
Read and write data on socket s, using the send() and recv() calls, until all data is exchanged.
Read and write data on socket ns, using the send() and recv() calls, until all data is exchanged.
Close socket s and end the TCP/IP session with the soclose() call.
Close socket ns with the soclose() call.
Accept another connection from a client, or close the original socket s with the soclose() call.

Figure 12. A Typical TCP Socket Session

5.1.2.2 A Typical UDP Socket Session

You cannot clearly distinguish UDP socket processes by server and client roles, as you can with TCP socket processes. Instead, the distinction is between connected and unconnected sockets. An unconnected socket can communicate with any host; but a connected socket, because it has a dedicated destination, can send data to and receive data from only one host.

Both connected and unconnected sockets send their data over the network without verification. Consequently, when the UDP interface accepts a packet, the arrival of the packet and the integrity of the packet cannot be guaranteed.

See Figure 13 for the general sequence of calls to follow for most socket routines using UDP sockets.

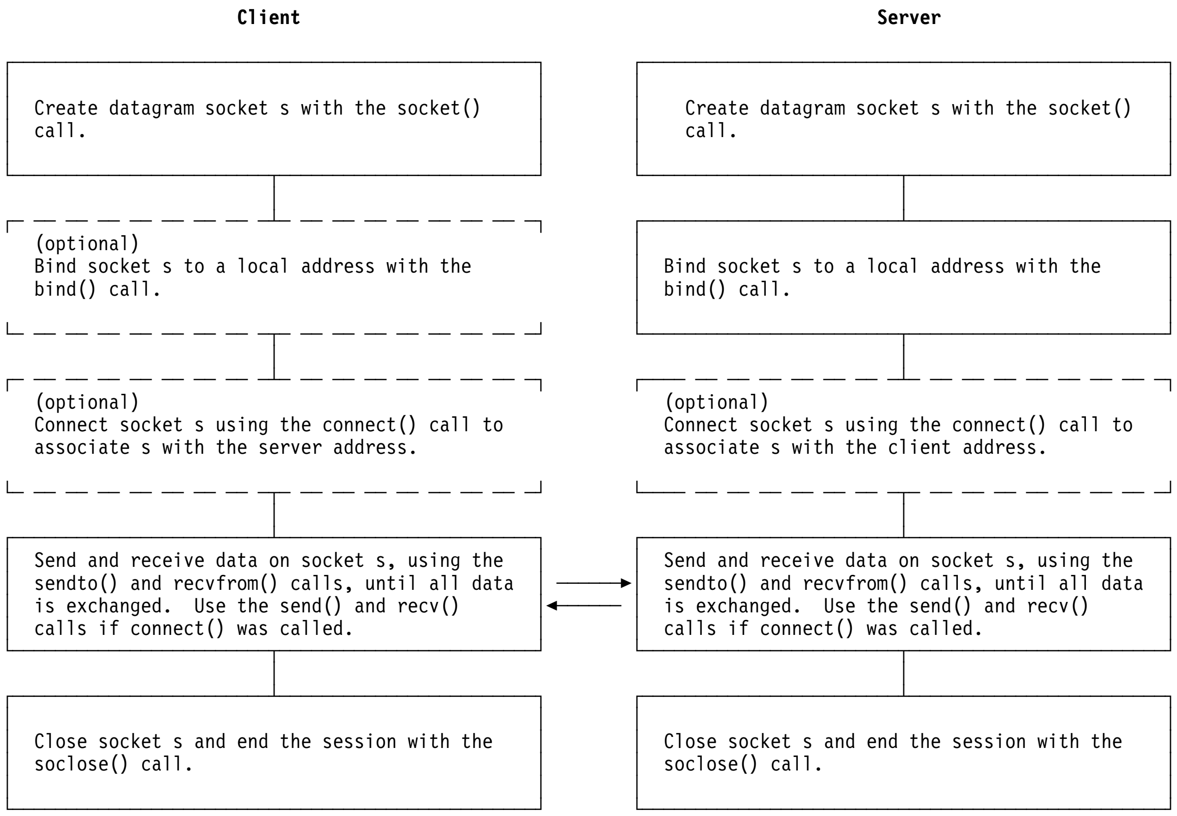

Client
Server
Create datagram socket s with the socket() call.
Create datagram socket s with the socket() call.
(optional) Bind socket s to a local address with the bind() call.
Bind socket s to a local address with the bind() call.
(optional) Connect socket s using the connect() call to associate s with the server address.
(optional) Connect socket s using the connect() call to associate s with the client address.
Send and receive data on socket s, using the sendto() and recvfrom() calls, until all data is exchanged. Use the send() and recv() calls if connect() was called.
Send and receive data on socket s, using the sendto() and recvfrom() calls, until all data is exchanged. Use the send() and recv() calls if connect() was called.
Close socket s and end the session with the soclose() call.
Close socket s and end the session with the soclose() call.

Figure 13. A Typical UDP Socket Session

5.1.2.3 Network Utility Routines

The 4690OS socket API also provides a set of network utility routines to perform useful tasks such as internet address translation, domain name resolution, network byte order translation, and access to the database of useful network information. This section describes a few network utility routines.

5.1.2.3.1 Host Names Information: The following is a list of host calls:

- gethostbyname()
- gethostbyaddr()
- sethostent()
- gethostent()
- endhostent()

The gethostbyname() call takes an internet host name and returns a *hostent* structure, which contains the name of the host, aliases, host address family, and host address. The *hostent* structure is defined in the <NETDB.H> header file. The gethostbyaddr() call maps the internet host address into a *hostent* structure.

The database for these calls is provided by the name server or the ADX_SDT1:ADXHSIHF.DAT file if a name server is not present or is unable to resolve the host name. Because of the differences in the databases and their access protocols, the information returned can differ.

The sethostent(), gethostent(), and endhostent() calls provide sequential access to the ADX_SDT1:ADXHSIHF.DAT file.

5.1.2.3.2 Network Names Information: The following is a list of network calls:

- getnetbyname()
- getnetbyaddr()
- setnetent()
- getnetent()
- endnetent()

The getnetbyname() call takes a network name and returns a *netent* structure, which contains the name of the network, aliases, network address family, and network number. The *netent* structure is defined in the <NETDB.H> header file. The getnetbyaddr() call maps the network number into a *netent* structure.

The database for these calls is provided by the ADX_SDT1:ADXHSINF.DAT file.

The setnetent(), getnetent(), and endnetent() calls provide sequential access to the ADX_SDT1:ADXHSINF.DAT file.

5.1.2.3.3 Protocol Names Information: The following is a list of protocol calls:

- getprotobyname()
- setprotoent()
- getprotoent()
- endprotoent()

The getprotobyname() call takes the protocol name and returns a *protoent* structure, which contains the name of the protocol, aliases, and protocol number. The *protoent* structure is defined in the <NETDB.H> header file.

The database for these calls is provided by the ADX_SDT1:ADXHSIPF.DAT file.

The setprotoent(), getprotoent(), and endprotoent() calls provide sequential access to the ADX_SDT1:ADXHSIPF.DAT file.

5.1.2.3.4 Service Names Information: The following is a list of service calls:

- getservbyname()
- getservbyport()
- setservent()
- getservent()
- endservent()

The getservbyname() call takes the service name and protocol, and returns a *servent* structure that contains the name of the service, aliases, port number, and protocol. The *servent* structure is defined in the <NETDB.H> header file. The getservbyport() call maps the port number and protocol into a *servent* structure.

The database for these calls is provided by the ADX_SDT1:ADXHSIPS.DAT file.

The setservent(), getservent(), and endservent() calls provide sequential access to the ADX_SDT1:ADXHSIPS.DAT file.

5.1.2.3.5 Network Byte Order Translation: Ports and addresses are usually specified to calls using the network byte ordering convention. The following calls translate integers from host to network byte order and from network to host byte order.

htonl()	Translates host to network, long integer (32-bit)
htons()	Translates host to network, short integer (16-bit)
ntohl()	Translates network to host, long integer (32-bit)
ntohs()	Translates network to host, short integer(16-bit)

5.1.2.3.6 Internet Address Manipulation: The following calls convert internet addresses and decimal notation and manipulate the network number and local network address portions of an internet address.

inet_addr()	Translates dotted decimal notation to a 32-bit internet address (network byte order)
inet_network()	Translates dotted decimal notation to a network number (host byte order), and zeros in the host part
inet_ntoa()	Translates 32-bit internet address (network byte order) to dotted decimal notation
inet_netof()	Extracts network number (host byte order) from 32-bit internet address (network byte order)
inet_lnaof()	Extracts local network address (host byte order) from 32-bit internet address (network byte order)
inet_makeaddr()	Constructs internet address (network byte order) from network number and local network address

5.1.2.3.7 Domain Name Resolution: Resolver calls are used to resolve the symbolic host name into an internet address and to extract more information about the host from the database.

The resolver calls determine whether the name server is present by referencing the ADX_SDT1:ADXHSIRF.DAT file. To resolve a name with no name server present, the resolver calls check the ADX_SDT1:ADXHSIHF.DAT file. file for an entry that maps the name to an address. To resolve a name in a name server network, the resolver calls query the domain name server database. If this query fails, the calls then check for an entry in the local ADX_SDT1:ADXHSIHF.DAT file.

The following resolver calls are used to make, send, and interpret packets for name servers in the internet domain:

- res_mkquery()
- res_init()

5.2 C Socket Library

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

Socket Header File	What It Contains
	Internet name server definition
NERRNO.H	Network error code definitions
NETDB.H	Data definitions for network utility calls
NET\IF.H	Definition for the Network Interface structure
NET\IF_ARP.H	Definition for the ARP protocol
NET\ROUTE.H	Definition for the routing table structure
NETINET\IN.H	Definition for Internet constants and structures
RESOLV.H	Resolver global definitions and variables
SYS\IOCTL.H	Definition for input-output control
	Definition for *fd_set* structure and FD_*xxx* macros
SYS\SOCKET.H	Data definitions and the socket structure
SYS\TIME.H	Definition of the *timeval* structure
TYPES.H	Data type definitions
UTILS.H	Definitions for byte swapping routines

To use the socket routines described in this chapter, you must have the socket library, ADXHSISL.L86, on your system.

5.3 Porting a Socket API Application

The IBM 4690OS socket implementation differs from the Berkeley socket implementation as follows:

- Sockets are not 4690OS files or devices. Socket numbers have no relationship to 4690OS file handles. Therefore, read(), write(), and close() do not work for sockets. Using read(), write(), or close() gives incorrect results. Use the send(), and soclose() functions instead.
- Some socket calls require that you call the sock_init() routine before you call them. Therefore, always call sock_init() at the beginning of programs using the socket interface.

  Error codes are accessed by the *tcperrno* variable. you can define *errno* to *tcperrno* using

  #define errno tcperrno.

  Include <NERRNO.H> for TCP/IP errno values.
- The select() call has a different interface. Unlike the Berkeley select() call, you cannot use the 4690OS select() call to wait for activity on devices other than sockets.
- ioctl() implementation might differ from the current Berkeley ioctl() implementation. For example, IBM has added a *lendata* parameter, which the current Berkeley ioctl() implementation does not support. Other functions of the IBM ioctl() call might also differ from the current Berkeley ioctl() implementation. In addition, the getsockopt() and setsockopt() might provide different support. See 5.4.35, “ioctl()” on page 158, 5.4.26, “getsockopt()” on page 145, and 5.4.55, “setsockopt()” on page 190 for more information.

You must define the variable OS2 by doing one of the following:

- Place `#define OS2` at the top of each file that includes TCP/IP header files.

  Note this is really 'OS2', not '4690'.

5.4 C Socket Calls

This section provides the syntax, parameters, and other appropriate information for each C socket call supported by TCP/IP for 4690OS.

5.4.1 accept()

The accept() call accepts a TCP connection request from a remote host.

5.4.1.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.1.2 Parameters

s

The socket descriptor

name

The socket address of the connecting client that is filled by accept() before it returns. The format of *name* 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.

namelen

Initially points to an integer that contains the size in bytes of the storage pointed to by *name*. On return, that integer contains the size of the data returned in the storage pointed to by *name*. If *name* is NULL, then *namelen* is ignored and can be NULL.

5.4.1.3 Return Values

A non-negative socket descriptor indicates success; the value –1 indicates an error. Get the specific error code by accessing the *errno* variable.

Possible Error Code	**Description**
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using *name* and *namelen* would result in an attempt to copy the address into a portion of the caller’s address space into which information cannot be written.
EINVAL	listen() was not called for socket *s*.
ENOBUFS	Not enough buffer space is available to create the new socket.
EOPNOTSUPP	The *s* parameter is not of type SOCK_STREAM.
EWOULDBLOCK	The *s* parameter is in nonblocking mode and no connections are on the queue.
ECONNABORTED	The software caused a connection abend.

5.4.1.4 Description

A server uses the accept() call 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 the error code 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 <NETINET\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, in bytes, of the buffer 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.

You use this call only with SOCK_STREAM sockets. You cannot screen clients without calling accept(). The application cannot tell the system from which requesters it will accept connections, but the caller can choose to close a connection immediately after discovering the requester's identity.

Use the select() call and set the bit in the read descriptor array to check the socket for incoming connection requests.

5.4.1.5 Examples

Two examples of the accept() call follow. In the first, the caller wants the client's address to be returned; in the second, the caller does not.

```
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);
```

5.4.1.6 Related Calls

bind()
connect()
getpeername()
listen()
socket()

5.4.2 bind()

The bind() call binds a local name to the socket.

5.4.2.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.2.2 Parameters

s
: Socket descriptor of any type that was returned by a previous socket() call

name
: Pointer to a *sockaddr* structure (buffer) containing the name that is to be bound to *s*

namelen
: Size of the buffer pointed to by *name* in bytes

5.4.2.3 Return Values

The value 0 indicates success; the value −1 indicates an error.

Possible Error Code	**Description**
EADDRINUSE	The address is already in use. See the SO_REUSEADDR option described under 5.4.26, "getsockopt()" on page 145 and the SO_REUSEADDR option described under 5.4.55, "setsockopt()" on page 190.
EADDRNOTAVAIL	The address specified is not valid on this host. For example, the internet address does not specify a valid network interface.
EAFNOSUPPORT	The address family is not supported.
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using *name* and *namelen* would result in an attempt to copy the address into a nonwriteable portion of the caller's address space.
EINVAL	The socket is already bound to an address. For example, you cannot bind a name to a socket that is in the connected state. This value is also returned if *namelen* is not the expected length.
ENOBUFS	No buffer space is available.

5.4.2.4 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.

Because *s* was created in the AF_INET domain, the format of the name buffer is expected to be *sockaddr_in* as defined in the header file <NETINET\IN.H>:

```
struct in_addr
{
        u_long s_addr;
};

struct sockaddr_in
{
        short   sin_family;
        u_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. The application can call getsockname() to discover the port number assigned. The *sin_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 multihomed 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 *sin_addr* is set to the constant INADDR_ANY, as defined in <NETINET\IN.H>, the caller is 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 zeros.

5.4.2.5 Examples

Note the following about the bind() call examples:

- Put the internet address and port 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.
- Set the *address* field using the inet_addr() utility routine, which takes a character string representing the dotted decimal address of an interface and returns the binary internet address representation in network byte order.
- Zero the structure before using it to ensure that the name requested does not set any reserved fields.

See 5.4.4, “connect()” on page 114 for examples of how a client might connect to servers.

```
int rc;
int s;
struct sockaddr_in myname;
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; /* all interfaces */
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));
```

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.

5.4.2.6 Related Calls

connect()
gethostbyname()
getsockname()
htons()
inet_addr()
listen()
port_cancel()
socket()

5.4.3 bswap()

The bswap() call swaps bytes in a short integer.

5.4.3.1 Syntax

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

u_short bswap(a)
u_short a;
```

5.4.3.2 Parameter

a

Unsigned short integer whose bytes are to be swapped

5.4.3.3 Return Value

The bswap() call returns the translated short integer.

5.4.3.4 Description

The bswap() call swaps bytes in a short integer.

5.4.3.5 Related Calls

htonl()
htons()
lswap()
ntohl()
ntohs()

5.4.4 connect()

The connect() call requests a connection to a remote host.

5.4.4.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.4.2 Parameters

s
: Socket descriptor

name
: Pointer to a *socket address* structure that contains the address of the socket to which a connection will be attempted

namelen
: Size, in bytes, of the *socket address* pointed to by *name*

5.4.4.3 Return Values

The value 0 indicates success; the value −1 indicates an error.

Possible Error Code	**Description**
EADDRNOTAVAIL	The calling host cannot reach the specified destination.
EAFNOSUPPORT	The address family is not supported.
EALREADY	The socket *s* is marked nonblocking, and a previous connection attempt has not completed.
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
ECONNREFUSED	The connection request was rejected by the destination host.
EFAULT	Using *name* and *namelen* would result in an attempt to copy the address into a portion of the caller's address space to which data cannot be written.
EINPROGRESS	The socket *s* is marked nonblocking, and the connection cannot be completed immediately. The EINPROGRESS value does not indicate an error condition.
EISCONN	The socket *s* is already connected.
ENETUNREACH	The network cannot be reached from this host.
ETIMEDOUT	The connection establishment timed out before a connection was made.
ENOBUFS	No buffer space available.
EOPNOTSUPP	The operation is not supported on the socket.

5.4.4.4 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 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 sets the error code 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, connect() returns −1 with the error code 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 lets the application use data transfer calls reserved for sockets that are in the connected state. In this case, readv(), writev(), send(), sendto(), and recvfrom() are available. Stream sockets can call connect() only once, but datagram sockets can call connect() multiple times to change their association. Datagram sockets can dissolve their association by connecting to an incorrect address such as the null address (all fields zeroed).

The *s* parameter is the socket used to originate the connection request. 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*.

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 <NETINET\IN.H>.

```
struct in_addr
{
        u_long s_addr;
};

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

Set the *sin_family* field to AF_INET. Set the *sin_port* field to the port to which the server is bound, and specify it in network byte order. Do not use the *sin_zero* field; set it to all zeros.

5.4.4.5 Examples

Note the following about these connect() call examples:

- Put the internet address and port 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.
- Set the address field using the inet_addr() utility routine, which takes a character string representing the dotted decimal address of an interface and returns the binary internet address representation in network byte order.
- To ensure that the name requested does not set any reserved fields, zero the structure before using it .

These examples could be used to connect to the servers shown in the examples listed with 5.4.2, “bind()” on page 110.

```
int s;
struct sockaddr_in servername;
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));
```

5.4.4.6 Related Calls

accept()
bind()
htons()
inet_addr()
listen()
select()
socket()

5.4.5 dn_comp()

The dn_comp() call compresses the expanded domain name.

5.4.5.1 Syntax

```
#include <types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>

int dn_comp(exp_dn, comp_dn, length, dnptrs, lastdnptr)
u_char *exp_dn;
u_char *comp_dn;
int length;
u_char **dnptrs;
u_char **lastdnptr;
```

5.4.5.2 Parameters

exp_dn
: Pointer to the location of an expanded domain name

comp_dn
: Pointer to an array containing the compressed domain name

length
: Length in bytes of the array pointed to by the *comp_dn* parameter

dnptrs
: List of pointers to previously compressed names in the current message

lastdnptr
: Pointer to the end of the array pointed to by *dnptrs*

5.4.5.3 Return Values

When successful, the dn_comp() call returns the size of the compressed domain name. If it fails, the call returns a value of −1.

5.4.5.4 Description

The dn_comp() call compresses the domain name pointed to by the *exp_dn* parameter and stores it in the area pointed to by the *comp_dn* parameter. It uses the global *_res* structure, which is defined in the <RESOLV.H> header file.

5.4.5.5 Related Calls

dn_expand()
res_init()
res_mkquery()
res_send()

5.4.6 dn_expand()

The dn_expand() call expands a compressed domain name to a full domain name.

5.4.6.1 Syntax

```
#include <types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>

int dn_expand(msg, eomorig, comp_dn, exp_dn, length)
u_char *msg;
u_char *eomorig;
u_char *comp_dn;
u_char *exp_dn;
int length;
```

5.4.6.2 Parameters

msg
: Pointer to the beginning of a message

eomorig
: Pointer to the end of the original message that contains the compressed domain name

comp_dn
: Pointer to the compressed domain name

exp_dn
: Pointer to a buffer that holds the resulting expanded domain name

length
: Length in bytes of the buffer pointed to by the *exp_dn* parameter

5.4.6.3 Return Values

If it succeeds, the dn_expand() call returns the size of the expanded domain name. If it fails, the call returns a value of −1.

5.4.6.4 Description

The dn_expand() call expands a compressed domain name to a full domain name, converting the expanded name to all uppercase letters. It uses the global *_res* structure, which is defined in the <RESOLV.H> header file.

5.4.6.5 Related Calls

dn_comp()
res_init()
res_mkquery()
res_send()

5.4.7 endhostent()

The endhostent() call closes the HOSTS file.

5.4.7.1 Syntax

```
void endhostent()
```

5.4.7.2 Description

The endhostent() call closes the ADX_SDT1:ADXHSIHF.DAT file, which contains information about known hosts.

5.4.7.3 Related Calls

gethostbyaddr()
gethostbyname()
gethostent()
sethostent()

5.4.8 endnetent()

The endnetent() call closes the NETWORKS file.

5.4.8.1 Syntax

```
void endnetent()
```

5.4.8.2 Description

The endnetent() call closes the ADX_SDT1:ADXHSINF.DAT file, which contains information about known networks.

5.4.8.3 Related Calls

getnetbyaddr()
getnetbyname()
getnetent()
setnetent()

5.4.9 endprotoent()

The endprotoent() call closes the PROTOCOL file.

5.4.9.1 Syntax

```
void endprotoent()
```

5.4.9.2 Description

The endprotoent() call closes the ADX_SDT1:ADXHSIPF.DAT file, which contains information about known protocols.

5.4.9.3 Related Calls

getprotobyname()
getprotoent()
setprotoent()

5.4.10 endservent()

The endservent() call closes the SERVICES file.

5.4.10.1 Syntax

```
void endservent()
```

5.4.10.2 Description

The endservent() call closes the ADX_SDT1:ADXHSISF.DAT file, which contains information about known services.

5.4.10.3 Related Calls

getservbyname()
getservbyport()
getservent()
setservent()

5.4.11 gethostbyaddr()

The gethostbyaddr() call returns information about a host specified by an internet address.

5.4.11.1 Syntax

```
#include <netdb.h>

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

5.4.11.2 Parameters

addr
: Pointer to a 32-bit internet address in network byte order

addrlen
: Size of *addr* in bytes

domain
: Address domain supported (AF_INET)

5.4.11.3 Return Values

This call returns a pointer to a *hostent* structure for the host address specified on the call. The <NETDB.H> header file defines the *hostent* structure and contains the following elements:

Element	**Description**
h_name	Official name of the host
h_aliases	Zero-terminated array of alternative names for the host
h_addrtype	Type of address being returned, always set to AF_INET
h_length	Length of the address in bytes
h_addr	Pointer to the network address of the host

The return value points to static data that later calls overwrite. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or EOF. The value of *h_errno* indicates the specific error.

h_errno Value	**Code**	**Description**
HOST_NOT_FOUND	1	The host specified by the *addr* parameter is not found.
TRY_AGAIN	2	The local server does not receive a response from an authorized server. Try again later.
NO_RECOVERY	3	This error code indicates an unrecoverable error.
NO_DATA	4	The requested *addr* is valid but does not have an internet address at the name server.
NO_ADDRESS	4	The requested *addr* is valid but does not have an internet address at the name server.

5.4.11.4 Description

The gethostbyaddr() call tries to resolve the host internet address through a name server, if one is present. If a name server is not present or cannot resolve the host name, gethostbyaddr() sequentially searches the ADX_SDT1:ADXHSIHF.DAT file until a matching host address is found or an EOF marker is reached.

5.4.11.5 Related Calls

endhostent()
gethostbyname()
gethostent()
sethostent()

5.4.12 gethostbyname()

The gethostbyname() call returns information about a host specified by a host name.

5.4.12.1 Syntax

```
#include <netdb.h>

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

5.4.12.2 Parameter

name
Name of the host being queried

5.4.12.3 Return Values

This call returns a pointer to a *hostent* structure for the host name specified on the call. The <NETDB.H> header file defines the *hostent* structure, which contains the following elements:

Element	**Description**
h_name	Official name of the host
h_aliases	Zero-terminated array of alternative names for the host
h_addrtype	Type of address being returned, always set to AF_INET
h_length	Length of the address in bytes
h_addr	Pointer to the network address of the host

The return value points to static data that later calls overwrite. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or EOF. The value of *h_errno* indicates the specific error.

h_errno Value	**Code**	**Description**
HOST_NOT_FOUND	1	The host specified by the *name* parameter is not found.
TRY_AGAIN	2	The local server does not receive a response from an authorized server. Try again later.
NO_RECOVERY	3	This error code indicates an unrecoverable error.
NO_DATA	4	The requested *name* is valid but does not have an internet address at the name server.
NO_ADDRESS	4	The requested *name* is valid but does not have an internet address at the name server.

5.4.12.4 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 or cannot resolve the host name, gethostbyname() sequentially searches the ADX_SDT1:ADXHSIHF.DAT file until it finds a matching host name or reaches an EOF marker.

5.4.12.5 Related Calls

endhostent()
gethostbyaddr()
gethostent()
sethostent()

5.4.13 gethostent()

The gethostent() call returns a pointer to the next entry in the HOSTS file.

5.4.13.1 Syntax

```
#include <netdb.h>

struct hostent *gethostent()
```

5.4.13.2 Return Values

The return value points to static data that later calls overwrite. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or EOF. The <NETDB.H> header file defines the *hostent* structure, which contains the following elements:

Element	**Description**
h_name	Official name of the host
h_aliases	Zero-terminated array of alternative names for the host
h_addrtype	Type of address being returned, always set to AF_INET
h_length	Length of the address in bytes
h_addr	Pointer to the network address of the host

5.4.13.3 Description

The gethostent() call reads the next line of the ADX_SDT1:ADXHSIHF.DAT file and returns a pointer to the next entry in the HOSTS file.

5.4.13.4 Related Calls

endhostent()
gethostbyaddr()
gethostbyname()
sethostent()

5.4.14 gethostid()

The gethostid() call returns the unique 32-bit identifier of the current host.

5.4.14.1 Syntax

```
#include <types.h>

u_long gethostid()
```

5.4.14.2 Return Values

The gethostid() call returns the 32-bit identifier, in host byte order of the current host, which should be unique across all hosts.

5.4.14.3 Description

The gethostid() call gets the unique 32-bit identifier for the current host.

5.4.15 getnetbyaddr()

The getnetbyaddr() call returns the NETWORKS file entry that contains the specified address.

5.4.15.1 Syntax

```
#include <netdb.h>

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

5.4.15.2 Parameters

net
: Network address

type
: Address domain supported (AF_INET)

5.4.15.3 Return Values

The return value points to static data that later calls overwrite. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or EOF.

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

Element	**Description**
n_name	Official name of the network
n_aliases	Array, terminated with a NULL pointer, of alternative names for the network
n_addrtype	Type of network address being returned, always set to AF_INET
n_net	Network number, returned in host byte order

5.4.15.4 Description

The getnetbyaddr() call searches the ADX_SDT1:ADXHSINF.DAT file for the specified network address.

5.4.15.5 Related Calls

endnetent()
getnetbyname()
getnetent()
setnetent()

5.4.16 getnetbyname()

The getnetbyname() call returns the NETWORKS file entry that contains the specified name.

5.4.16.1 Syntax

```
#include <netdb.h>

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

5.4.16.2 Parameter

name
Pointer to a network name

5.4.16.3 Return Values

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; it contains the following elements:

Element	**Description**
n_name	Official name of the network
n_aliases	Array, terminated with a NULL pointer, of alternative names for the network
n_addrtype	Type of network address being returned, always set to AF_INET
n_net	Network number, returned in host byte order

The return value points to static data that later calls overwrite. A pointer to a *netent* structure indicates success; a NULL pointer indicates an error or EOF.

5.4.16.4 Description

The getnetbyname() call searches the ADX_SDT1:ADXHSINF.DAT file for the specified network name.

5.4.16.5 Related Calls

endnetent()
getnetbyaddr()
getnetent()
setnetent()

5.4.17 getnetent()

The getnetent() call returns the next entry in the NETWORKS file.

5.4.17.1 Syntax

```
#include <netdb.h>

struct netent *getnetent()
```

5.4.17.2 Return Values

The getnetent() call returns a pointer to the next entry in the ADX_SDT1:ADXHSINF.DAT file. The return value points to static data that later calls overwrite.

A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or EOF.

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

Element	**Description**
n_name	Official name of the network
n_aliases	Array, terminated with a NULL pointer, of alternative names for the network
n_addrtype	Type of network address being returned, always set to AF_INET
n_net	Network number, returned in host byte order

5.4.17.3 Description

The getnetent() call returns the next entry of the ADX_SDT1:ADXHSINF.DAT file.

5.4.17.4 Related Calls

endnetent()
getnetbyaddr()
getnetbyname()
setnetent()

5.4.18 getpeername()

The getpeername() call returns the name of the peer connected to socket *s*.

5.4.18.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.18.2 Parameters

s
: Socket descriptor.

name
: Internet address of the connected socket that is filled by getpeername() before it returns. The exact format of *name* is determined by the domain in which communication occurs.

namelen
: Size of the address structure pointed to by *namelen*, in bytes.

5.4.18.3 Return Values

The value 0 indicates success; the value –1 indicates an error.

Possible Error Code	**Description**
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the caller’s address space.
ENOTCONN	The socket is not in the connected state.
ENOBUFS	No buffer space is available.

5.4.18.4 Description

The getpeername() call returns the name of the peer connected to socket *s*. *namelen* must be initialized to indicate the size of the space pointed to by *name* and is set to the number of bytes copied into the space before the call returns. If the buffer of the local host is too small, the peer name is truncated.

5.4.18.5 Related Calls

accept()
connect()
getsockname()
socket()

5.4.19 getprotobyname()

The getprotobyname() call returns a protocol entry specified by a name in the PROTOCOL file.

5.4.19.1 Syntax

```
#include <netdb.h>

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

5.4.19.2 Parameter

name
    Pointer to the specified protocol

5.4.19.3 Return Values

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**
p_name	Official name of the protocol
p_aliases	Array, terminated with a NULL pointer, of alternative names for the protocol
p_proto	Protocol number

The return value points to static data that later calls overwrite. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or EOF.

5.4.19.4 Description

The getprotobyname() call searches the ADX_SDT1:ADXHSIPF.DAT file for the specified protocol name.

5.4.19.5 Related Calls

endprotoent()
getprotoent()
setprotoent()

5.4.20 getprotobynumber()

The getprotobynumber() call returns a protocol entry specified by a number in the PROTOCOL file.

5.4.20.1 Syntax

```
#include <netdb.h>

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

5.4.20.2 Parameter

proto
Protocol number

5.4.20.3 Return Values

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
p_name	Official name of the protocol
p_aliases	Array, terminated with a NULL pointer, of alternative names for the protocol
p_proto	Protocol number

The return value points to static data that later calls overwrite. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or EOF.

5.4.20.4 Description

The getprotobynumber() call searches the ADX_SDT1:ADXHSIPF.DAT file for the specified protocol number.

5.4.20.5 Related Calls

endprotoent()
getprotobyname()
getprotoent()
setprotoent()

5.4.21 getprotoent()

The getprotoent() call returns the next entry in the PROTOCOL file.

5.4.21.1 Syntax

```
#include <netdb.h>

struct protoent *getprotoent()
```

5.4.21.2 Return Values

The getprotoent() call returns a pointer to the next entry in the file, ADX_SDT1:ADXHSIPF.DAT.

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

Element	**Description**
p_name	Official name of the protocol
p_aliases	Array, terminated with a NULL pointer, of alternative names for the protocol
p_proto	Protocol number

The return value points to static data that later calls overwrite. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or EOF.

5.4.21.3 Description

The getprotoent() call searches for the next line in the ADX_SDT1:ADXHSIPF.DAT file.

5.4.21.4 Related Calls

endprotoent()
getprotobyname()
setprotoent()

5.4.22 getservbyname()

The getservbyname() call returns a service entry specified by a name in the SERVICES file.

5.4.22.1 Syntax

```
#include <netdb.h>

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

5.4.22.2 Parameters

name
: Pointer to the service name

proto
: Pointer to the specified protocol

5.4.22.3 Return Values

The 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**
s_name	Official name of the service
s_aliases	Array, terminated with a NULL pointer, of alternative names for the service
s_port	Port number of the service
s_proto	Required protocol to contact the service

The return value points to static data that later calls overwrite. A pointer to a *servent* structure indicates success. A NULL pointer indicates an error or EOF.

5.4.22.4 Description

The getservbyname() call searches the ADX_SDT1:ADXHSISF.DAT file for the specified service name, which must match the protocol if a protocol is stated.

5.4.22.5 Related Calls

endservent()
getservbyport()
getservent()
setservent()

5.4.23 getservbyport()

The getservbyport() call returns a service entry specified by a port number in the SERVICES file.

5.4.23.1 Syntax

```
#include <netdb.h>

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

5.4.23.2 Parameters

port
  Specified port

proto
  Pointer to the specified protocol

5.4.23.3 Return Values

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**
s_name	Official name of the service
s_aliases	Array, terminated with a NULL pointer, of alternative names for the service
s_port	Port number of the service
s_proto	Required protocol to contact the service

The return value points to static data that later calls overwrite. A pointer to a *servent* structure indicates success. A NULL pointer indicates an error or EOF.

5.4.23.4 Description

The getservbyport() call sequentially searches the ADX_SDT1:ADXHSISF.DAT file for the specified port number, which must match the protocol if a protocol is stated.

5.4.23.5 Related Calls

endservent()
getservbyname()
getservent()
setservent()

5.4.24 getservent()

The getservent() call returns the next entry in the SERVICES file.

5.4.24.1 Syntax

```
#include <netdb.h>

struct servent *getservent()
```

5.4.24.2 Return Values

The getservent() call returns a pointer to the next entry in the ADX_SDT1:ADXHSISF.DAT file. The *servent* structure is defined in the <NETDB.H> header file and contains the following elements:

Element	**Description**
s_name	Official name of the service
s_aliases	Array, terminated with a NULL pointer, of alternative names for the service
s_port	Port number of the service
s_proto	Required protocol to contact the service

The return value points to static data that later calls overwrite. A pointer to a *servent* structure indicates success. A NULL pointer indicates an error or EOF.

5.4.24.3 Description

The getservent() call searches for the next line in the ADX_SDT1:ADXHSISF.DAT file.

5.4.24.4 Related Calls

endservent()
getservbyname()
getservbyport()
setservent()

5.4.25 getsockname()

The getsockname() call gets the local socket name.

5.4.25.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.25.2 Parameters

s

Socket descriptor.

name

Address of the buffer into which getsockname() copies the name of *s*.

namelen

Must initially point to an integer that contains the size in bytes of the storage pointed to by *name*. Upon return, that integer contains the size of the data returned in the storage pointed to by *name*.

5.4.25.3 Return Values

The value 0 indicates success; the value −1 indicates an error. You can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside the caller's address space.
ENOBUFS	No buffer space is available.

5.4.25.4 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 is set to 0. 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().

5.4.25.5 Related Calls

accept()
bind()
connect()
getpeername()
sock_errno()
socket()

5.4.26 getsockopt()

The getsockopt() call gets socket options associated with a socket.

5.4.26.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.26.2 Parameters

s
: Socket descriptor.

level
: Level for which the option is set. Only SOL_SOCKET is supported.

optname
: Name of a specified socket option.

optval
: Pointer to option data.

optlen
: Pointer to the length of the option data.

5.4.26.3 Return Values

The value 0 indicates success; the value −1 indicates an error. You can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
EADDRINUSE	The address is already in use.
ENOTSOCK	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 address space.
ENOPROTOOPT	The *optname* parameter is not recognized, or the *level* parameter is not SOL_SOCKET.

5.4.26.4 Description

The getsockopt() call gets options associated with a socket. It can be called only for sockets in the AF_INET 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 name of the option and the level at which the option resides. To manipulate options at the socket level, the *level* parameter must be set to SOL_SOCKET, as defined in

<SYS\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 level is supported. The getprotobyname() call can be used to return the protocol number for a named protocol.

The *optval* and *optlen* parameters return data used by the particular get command. The *optval* parameter points to a buffer that is to receive the data requested by the get command. 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, SO_SNDBUF, and SO_RCVBUF, expect *optval* to point to an integer and *optlen* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is 0, the option is disabled. The SO_LINGER option expects *optval* to point to a *linger* structure, as defined in <SYS\SOCKET.H>. The SO_SNDBUF and SO_RCVBUF options expect *optval* to point to a long integer. 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 0 if the SO_LINGER option is being disabled. A nonzero value enables the option. The *l_linger* field specifies the amount of time to linger on close.

The following options are recognized at the socket level:

Option	**Description**
SO_BROADCAST	Toggles the ability to broadcast messages. Enabling this option lets the application send broadcast messages over *s*, if the interface specified in the destination supports broadcasting of packets. This option has no meaning for stream sockets.
SO_DEBUG	Toggles recording of debugging information.
SO_DONTROUTE	Toggles the routing bypass for outgoing messages. Enabling this option causes outgoing messages to bypass the standard routing algorithm and be directed to the appropriate network interface, according to the network portion of the destination address. When enabled, packets can be sent only to directly connected networks (networks for which this host has an interface). This option has no meaning for stream sockets.
SO_ERROR	Returns any pending error on the socket and clears the error status. You can use it to check for asynchronous errors on connected datagram sockets or for other asynchronous errors (errors that are not returned explicitly by one of the socket calls).
SO_KEEPALIVE	Toggles keep connection alive. TCP uses a timer called the keepalive timer. This timer is used to monitor idle connections that might have been disconnected because of a peer time-out or abnormal termination. If this option is toggled, a keepalive packet is sent to the peer every 120 minutes. This is used mainly to enable servers to close connections that have already disappeared as a result of clients going away without closing connections. This option has meaning only for stream sockets.
SO_LINGER	Lingers on close if data is present. Enabling this option when unsent data is present and when soclose() is called causes the calling application to be blocked during that call until the data is transmitted or the connection times out. If this option is disabled, INET waits to try to send the data. The data transfer is usually successful, but cannot be guaranteed, because INET waits only a finite amount of time trying to send the data. The soclose() call returns without blocking the caller. This option has meaning only for stream sockets.

SO_OOBINLINE	Toggles reception of out-of-band data. Enabling this option causes out-of-band data to be placed in the normal data input queue as it is received, making it available to both recv() and recvfrom() without having to specify the MSG_OOB flag in those calls. Disabling this option causes out-of-band data to be placed in the priority data input queue as it is received, making it available to both recv() and recvfrom() only by specifying the MSG_OOB flag in those calls. This option has meaning only for stream sockets.
SO_RCVBUF	Sets the size of the receive buffer to the value contained in the buffer pointed to by *optval*. This lets you change the size to meet specific application needs, such as increasing its size for high-volume connections and input.
SO_RCVLOWAT	Retrieves receive low-water mark information.
SO_RCVTIMEO	Retrieves receive time-out information.
SO_REUSEADDR	Toggles local address reuse. Enabling this option enables local addresses that are already in use to be bound. This alters the normal algorithm used in the bind() call. At connect time, the system checks that no local address and port have the same foreign address and port, and returns the error code EADDRINUSE if the association already exists.
SO_SNDBUF	Sets the size of the send buffer to the value contained in the buffer pointed to by *optval*. This lets you change the size to meet specific application needs. For example, you could increase the size for high-volume connections and output.
SO_SNDLOWAT	Retrieves send low-water mark information.
SO_SNDTIMEO	Retrieves send time-out information.
SO_TYPE	Returns the type of the socket. On return, the integer pointed to by *optval* is set to one of the following: SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW.
SO_USELOOPBACK	Bypasses hardware when possible.

5.4.26.5 Example

Examples of the getsockopt() call follow. See 5.4.55, “setsockopt()” on page 190 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  */
    }
}
```

5.4.26.6 Related Calls

getprotobyname()
setsockopt()
sock_errno()
socket()

5.4.27 htonl()

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

5.4.27.1 Syntax

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

u_long htonl(a)
u_long a;
```

5.4.27.2 Parameter

a

Unsigned long integer to be put into network byte order

5.4.27.3 Return Value

The htonl() call returns the translated long integer.

5.4.27.4 Description

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

5.4.27.5 Related Calls

bswap()
htons()
lswap()
ntohl()
ntohs()

5.4.28 htons()

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

5.4.28.1 Syntax

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

u_short htons(a)
u_short a;
```

5.4.28.2 Parameter

a
: Unsigned short integer to be put into network byte order

5.4.28.3 Return Value

The htons() call returns the translated short integer.

5.4.28.4 Description

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

5.4.28.5 Related Calls

bswap()
htonl()
lswap()
ntohl()
ntohs()

5.4.29 inet_addr()

The inet_addr() call constructs an internet address from character strings representing numbers expressed in standard dotted-decimal notation.

5.4.29.1 Syntax

```
#include <types.h>

u_long inet_addr(cp)
char *cp;
```

5.4.29.2 Parameter

cp
   A character string in standard dotted-decimal notation

5.4.29.3 Return Value

The internet address is returned in network byte order.

5.4.29.4 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 address space without any rearrangement of its bytes.

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

5.4.29.5 Related Calls

inet_lnaof()
inet_makeaddr()
inet_netof()
inet_network()
inet_ntoa()

5.4.30 inet_lnaof()

The inet_lnaof() call returns the local network portion of an internet address.

5.4.30.1 Syntax

```
#include <types.h>
#include <netinet\in.h>

u_long inet_lnaof(in)
struct in_addr in;
```

5.4.30.2 Parameter

in
    Host internet address

5.4.30.3 Return Value

The inet_lnaof() call returns the local network address in host byte order.

5.4.30.4 Description

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

5.4.30.5 Related Calls

inet_addr()
inet_makeaddr()
inet_netof()
inet_network()
inet_ntoa()

5.4.31 inet_makeaddr()

The inet_makeaddr() call constructs an internet address from a network number and a local address.

5.4.31.1 Syntax

```
#include <types.h>
#include <netinet\in.h>

struct in_addr inet_makeaddr(net, lna)
u_long net;
u_long lna;
```

5.4.31.2 Parameters

net
  Network number

lna
  Local network address

5.4.31.3 Return Value

The inet_makeaddr() call returns the internet address in network byte order.

5.4.31.4 Description

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

5.4.31.5 Related Calls

inet_addr()
inet_lnaof()
inet_netof()
inet_network()
inet_ntoa()

5.4.32 inet_netof()

The inet_netof() call returns the network number portion of the internet host address.

5.4.32.1 Syntax

```
#include <types.h>
#include <netinet\in.h>

u_long inet_netof(in)
struct in_addr in;
```

5.4.32.2 Parameter

in

Internet address in network byte order

5.4.32.3 Return Value

The inet_netof() call returns the network number in host byte order.

5.4.32.4 Description

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

5.4.32.5 Related Calls

inet_addr()
inet_lnaof()
inet_makeaddr()
inet_network()
inet_ntoa()

5.4.33 inet_network()

The inet_network() call constructs a network number from character strings representing numbers expressed in standard dotted-decimal notation.

5.4.33.1 Syntax

```
#include <types.h>

u_long inet_network(cp)
char *cp;
```

5.4.33.2 Parameter

cp
A character string in standard dotted-decimal notation

5.4.33.3 Return Value

The inet_network() call returns the network number in host byte order.

5.4.33.4 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.

5.4.33.5 Related Calls

inet_addr()
inet_lnaof()
inet_makeaddr()
inet_netof()
inet_ntoa()

5.4.34 inet_ntoa()

The inet_ntoa() call returns a pointer to a string in dotted-decimal notation.

5.4.34.1 Syntax

```
#include <types.h>
#include <netinet\in.h>

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

5.4.34.2 Parameter

in

Host internet address

5.4.34.3 Return Value

The inet_ntoa() call returns a pointer to the internet address expressed in dotted-decimal notation.

5.4.34.4 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.

5.4.34.5 Related Calls

inet_addr()
inet_lnaof()
inet_makeaddr()
inet_network()
inet_ntoa()

5.4.35 ioctl()

The ioctl() call performs special operations on socket descriptor *s*.

5.4.35.1 Syntax

```
#include <types.h>
#include <sys\ioctl.h>
#include <net\route.h>
#include <net\if.h>
#include <net\if_arp.h>

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

5.4.35.2 Parameters

s
: Socket descriptor

cmd
: Command to perform

data
: Pointer to the data associated with *cmd*

lendata
: Length of the data in bytes

5.4.35.3 Return Values

The value 0 indicates success; the value −1 indicates an error. You can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EINVAL	The request is not valid or not supported.
EOPNOTSUPP	The operation is not supported on the socket.
EFAULT	Using the *data* and *lendata* parameters would result in an attempt to access memory outside the caller’s address space.

5.4.35.4 Description

Use the ioctl() call to control the operating characteristics of sockets. 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 requested. The following are valid ioctl() commands:

Option	**Description**
FIOASYNC	Sets or clears asynchronous input-output for a socket. *data* is a pointer to an integer. If the integer is 0, asynchronous input-output on the socket is cleared. Otherwise, the socket is set for asynchronous input-output.
FIONBIO	Sets or clears nonblocking input-output for a socket. *data* 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. *data* 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. *data* is a pointer to a *rtentry* structure, as defined in <NET\ROUTE.H>. 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. *data* 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.
SIOCDARP	Deletes an arp table entry. *data* is a pointer to a *arpreq* as defined in <NET\IF_ARP.H>. The arp table entry passed as an argument is deleted from the arp tables, if it exists.
SIOCDELRT	Deletes a routing table entry. *data* is a pointer to a *rtentry* structure, as defined in <NET\ROUTE.H>. If it exists, the routing table entry passed as an argument is deleted from the routing tables.
SIOCGARP	Gets the arp table entries. *data* is a pointer to an *arpreq*, as defined in <NET\IF_ARP.H>. The arp table entry passed as an argument is returned from the arp tables if it exists.
SIOCGIFADDR	Gets the network interface address. *data* is a pointer to an *ifreq* structure, as defined in <NET\IF.H>. The interface address is returned in the argument.
SIOCGIFBRDADDR	Gets the network interface broadcast address. *data* is a pointer to an *ifreq* structure, as defined in <NET\IF.H>. The interface broadcast address is returned in the argument.
SIOCGIFCONF	Gets the network interface configuration. *data* is a pointer to an *ifconf* structure, as defined in <NET\IF.H>. The interface configuration is returned in the argument.
SIOCGIFDSTADDR	Gets the network interface destination address. *data* is a pointer to an *ifreq* structure, as defined in <NET\IF.H>. The interface destination (point-to-point) address is returned in the argument.
SIOCGIFFLAGS	Gets the network interface flags. *data* is a pointer to an *ifreq* structure, as defined in <NET\IF.H>. The interface flags are returned in the argument.
SIOCGIFMETRIC	Gets the network interface routing metric. *data* is a pointer to an *ifreq* structure, as defined in <NET\IF.H>. The interface routing metric is returned in the argument.
SIOCGIFNETMASK	Gets the network interface network mask. *data* is a pointer to an *ifreq* structure, as defined in <NET\IF.H>. The interface network mask is returned in the argument.
SIOCSARP	Sets an arp table entry. *data* is a pointer to an *arpreq* as defined in <NET\IF_ARP.H>. The arp table entry passed as an argument is added to the arp tables.

SIOCSIFADDR	Sets the network interface address. *data* is a pointer to an *ifreq* structure, as defined in <NET\IF.H>. Sets the interface address to the value passed in the argument.
SIOCSIFBRDADDR	Sets the network interface broadcast address. *data* is a pointer to an *ifreq* structure, as defined in <NET\IF.H>. Sets the interface broadcast address to the value passed in the argument.
SIOCSIFDSTADDR	Sets the network interface destination address. *data* is a pointer to an *ifreq* structure, as defined in <NET\IF.H>. Sets the interface destination (point-to-point) address to the value passed in the argument.
SIOCSIFFLAGS	Sets the network interface flags. *data* is a pointer to an *ifreq* structure, as defined in <NET\IF.H>. Sets the interface flags to the values passed in the argument.
SIOCSIFMETRIC	Sets the network interface routing metric. *data* is a pointer to an *ifreq* structure, as defined in <NET\IF.H>. Sets the interface routing metric to the value passed in the argument.
SIOCSIFNETMASK	Sets the network interface network mask. *data* is a pointer to an *ifreq* structure, as defined in <NET\IF.H>. Sets the interface network mask to the value passed in the argument.

5.4.35.5 Example

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

5.4.35.6 Related Call

sock_errno()

5.4.36 listen()

The listen() call completes the binding necessary for a socket and creates a connection request queue for incoming requests.

5.4.36.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.36.2 Parameters

s
: Socket descriptor

backlog
: Maximum length for the queue of pending connections

5.4.36.3 Return Values

The value 0 indicates success; the value −1 indicates an error. You can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EOPNOTSUPP	The *s* parameter is not a socket descriptor that supports the listen() call.

5.4.36.4 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. After the queue is 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 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 *backlog* is less than 0, it is set to 0. If *backlog* is greater than SOMAXCONN, as defined in <SYS\SOCKET.H>, it is set to SOMAXCONN.

5.4.36.5 Related Calls

accept()
bind()
connect()
sock_errno()
socket()

5.4.37 lswap()

The lswap() call swaps bytes in a long integer.

5.4.37.1 Syntax

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

u_long lswap(a)
u_long a;
```

5.4.37.2 Parameter

a
  Unsigned long integer whose bytes are to be swapped

5.4.37.3 Return Value

The lswap() call returns the translated long integer.

5.4.37.4 Description

The lswap() call swaps bytes in a long integer.

5.4.37.5 Related Calls

bswap()
htonl()
htons()
ntohl()
ntohs()

5.4.38 ntohl()

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

5.4.38.1 Syntax

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

u_long ntohl(a)
u_long a;
```

5.4.38.2 Parameter

a

Unsigned long integer to be put into host byte order

5.4.38.3 Return Value

The ntohl() call returns the translated long integer.

5.4.38.4 Description

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

5.4.38.5 Related Calls

bswap()
htonl()
htons()
lswap()
ntohs()

5.4.39 ntohs()

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

5.4.39.1 Syntax

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

u_short ntohs(a)
u_short a;
```

5.4.39.2 Parameter

a
Unsigned short integer to be put into host byte order

5.4.39.3 Return Value

The ntohs() call returns the translated short integer.

5.4.39.4 Description

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

5.4.39.5 Related Calls

bswap()
htonl()
htons()
lswap()
ntohl()

5.4.40 port_cancel()

The port_cancel() call shuts down all sockets that are bound to a port, and frees resources allocated to those sockets.

5.4.40.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

int port_cancel(p)
int p;
```

5.4.40.2 Parameter

p
    Number of the port to clear

5.4.40.3 Return Values

The call returns the number of sockets that were closed.

5.4.40.4 Description

The port_cancel() call shuts down all sockets that are bound to the port *p* and frees resources allocated to those sockets.

5.4.40.5 Related Calls

bind()
socket()
soclose()

5.4.41 readv()

The readv() call reads data on a socket and stores it in a set of specified buffers.

5.4.41.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.41.2 Parameters

s
: Socket descriptor

iov
: Pointer to an array of iovec structure

iovcnt
: Number of buffers pointed to by the *iov* parameter

5.4.41.3 Return Values

If successful, the call returns the number of bytes read into the buffers. The value 0 indicates the connection is closed; the value –1 indicates an error. You can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using *iov* and *iovcnt* would result in an attempt to access memory outside the caller’s address space.
SOCEINVAL	*iovcnt* was not valid, or one of the fields in the *iov* array was not valid.
SOCEWOULDBLOCK	The *s* parameter is in nonblocking mode, and no data is available to read.

5.4.41.4 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 <SYS\SOCKET.H> and contains the following fields:

Element	**Description**
iov_base	Pointer to the buffer
iov_len	Length of the buffer

The readv() call returns up to the number of bytes in the buffers pointed to by the *iov* parameter. This number is the sum of all *iov_len* fields. 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, the readv() call waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode. See 5.4.35, “ioctl()” on page 158 for a description of how to set nonblocking mode.

The readv() call applies only to connected sockets.

5.4.41.5 Related Calls

connect()
getsockopt()
ioctl()
recv()
recvfrom()
select()
send()
sendto()
setsockopt()
sock_errno()
socket()
writev()

5.4.42 recv()

The recv() call receives data on a connected socket.

5.4.42.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.42.2 Parameters

s
: Socket descriptor.

buf
: Pointer to the buffer that receives the data.

len
: Length in bytes of the buffer pointed to by the *buf* parameter.

flags
: Set by specifying one or more of the following flags. If you specify more than one flag, use the logical OR operator (|) to separate them. Setting this parameter is supported only for sockets in the AF_INET domain. The flags are as follows:

  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 later receive operation sees the same data.

5.4.42.3 Return Values

If successful, the call returns the length (in bytes) of the message or datagram. The value 0 indicates that the connection is closed; the value −1 indicates an error. You can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *buf* and *len* parameters would result in an attempt to access memory outside the caller's address space.
EWOULDBLOCK	The *s* parameter is in nonblocking mode, and no data is available to read.

5.4.42.4 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.

The recv() 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 5.4.35, “ioctl()” on page 158 for a description of how to set nonblocking mode.

5.4.42.5 Related Calls

connect()
getsockopt()
ioctl()
readv()
recvfrom()
select()
send()
sendto()
setsockopt()
sock_errno()
socket()
writev()

5.4.43 recvfrom()

The recvfrom() call receives data on a datagram socket, regardless of its connection status.

5.4.43.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.43.2 Parameters

s
: Socket descriptor.

buf
: Pointer to the buffer that receives the data.

len
: Length in bytes of the buffer pointed to by the *buf* parameter.

flags
: Set by specifying one or more of the following flags. This parameter is supported only for sockets in the AF_INET domain. The flags are as follows:

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 later receive operation sees the same data.

name
: Pointer to a *socket address* structure from which data is received.

namelen
: Pointer to the size of *name* in bytes.

5.4.43.3 Return Values

If successful, the call returns the length, in bytes, of the message or datagram. The value −1 indicates an error; you can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *buf* and *len* parameters would result in an attempt to access memory outside the caller’s address space.
EWOULDBLOCK	The *s* parameter is in nonblocking mode, and no data is available to read.

5.4.43.4 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 *name* is nonzero, the source address of the message is returned. *namelen* is first initialized to the size of the buffer associated with *name*; on return, it is modified to indicate the actual number of bytes stored there.

The recvfrom() 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() call waits for a message to arrive and blocks the caller, unless the socket is in nonblocking mode. See 5.4.35, “ioctl()” on page 158 for a description of how to set nonblocking mode.

5.4.43.5 Related Calls

getsockopt()
ioctl()
readv()
recv()
select()
send()
sendto()
setsockopt()
sock_errno()
socket()
writev()

5.4.44 res_init()

The res_init() call reads the RESOLV file for the default domain name.

5.4.44.1 Syntax

```
include <types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>

void res_init()
```

5.4.44.2 Description

The res_init() call reads the ADX_SDT1:ADXHSIRF.DAT file for the default domain name and for the internet address of the initial hosts running the name server. If that file does not exist, the call attempts name resolution using the ADX_SDT1:ADXHSIHF.DAT file. One of these files should be operational.

The call stores domain name information in the global *_res* structure, which is defined in the <RESOLV.H> header file.

5.4.44.3 Related Calls

dn_comp()
dn_expand()
res_mkquery()
res_send()

5.4.45 res_mkquery()

The res_mkquery() call makes a query message for the name servers in the Internet domain.

5.4.45.1 Syntax

```
#include <types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>

int res_mkquery(op, dname, class, type, data, datalen, newrr,
                buf, buflen)
int op;
char *dname;
int class;
int type;
char *data;
int datalen;
struct rrec *newrr;
char *buf;
int buflen;
```

5.4.45.2 Parameters

op

The usual type is QUERY, but you can set the parameter to any query type defined in the <ARPA\NAMESER.H> header file.

dname

Pointer to the domain name. If *dname* points to a single label and the RES_DEFNAMES bit in the *_res* structure defined in the <RESOLV.H> header file is set, the call appends *dname* to the current domain name. The current domain name is defined in the ADX_SDT1:ADXHSIRF.DAT file.

class

One of the following values:

C_IN	ARPA Internet
C_CHAOS	Chaos network at MIT

type

One of the following type values for resources and queries:

T_A	Host address
T_NS	Authoritative server
T_MD	Mail destination
T_MF	Mail forwarder
T_CNAME	Canonical name
T_SOA	Start of authority zone
T_MB	Mailbox domain name
T_MG	Mail group member
T_MR	Mail rename name
T_NULL	NULL resource record
T_WKS	Well-known service
T_PTR	Domain name pointer

T_HINFO	Host information
T_MINFO	Mailbox information
T_MX	Mail routing information
T_UINFO	User information
T_UID	User ID
T_GID	Group ID

data
: Pointer to the data sent to the name server as a search key.

datalen
: Size of the parameter *data* in bytes.

newrr
: Reserved for future updates; currently an unused parameter.

buf
: Pointer to the query message.

buflen
: Length in bytes of the buffer pointed to by the *buf* parameter.

5.4.45.3 Return Values

If it succeeds, the res_mkquery() call returns the size of the query. If the query is larger than the value of *buflen*, the call fails and returns a value of −1.

5.4.45.4 Description

The res_mkquery() call makes a query message for the name servers in the Internet domain and puts that query message in the location pointed by the *buf* parameter. It uses global *_res* structure, which is defined in the <RESOLV.H> header file.

5.4.45.5 Related Calls

dn_comp()
dn_expand()
res_init()
res_send()

5.4.46 res_send()

The res_send() call sends a query to a local name server.

5.4.46.1 Syntax

```
#include <types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>

int res_send(msg, msglen, ans, anslen)
char *msg;
int msglen;
char *ans;
int anslen;
```

5.4.46.2 Parameters

msg
: Pointer to the beginning of a message

msglen
: Length in bytes of the buffer pointed to by the *msg* parameter

ans
: Pointer to the location where the received response is stored

anslen
: Length in bytes of the buffer pointed by the *ans* parameter

5.4.46.3 Return Values

If it succeeds, the call returns the length of the message. If it fails, the call returns a value of –1.

5.4.46.4 Description

The res_send() call sends a query to the local name server and calls the res_init() call if the RES_INIT option of the global *_res* structure is not set. It also handles time-outs and retries. It uses the global *_res* structure, which is defined in the <RESOLV.H> header file.

5.4.46.5 Related Calls

dn_comp()
dn_expand()
res_init()
res_mkquery()

5.4.47 rexec()

The rexec() call allows command execution on a remote host.

5.4.47.1 Syntax

```
#include <utils.h>

int rexec( host, port, user, passwd, cmd, err_sd2p)
char **host;
int port;
char *user, *passwd, *cmd;
int *err-sd2p;
```

5.4.47.2 Parameters

host
: Contains the name of a remote host.

port
: Specifies the well-known DARPA Internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the `getservbyname("exec","tcp")` library call.

user
: Points to a user ID valid at the remote host.

passwd
: Points to a password valid at the remote host.

cmd
: Points to the name of the command to be executed at the remote host.

err_sd2p
: Points to error socket descriptor. An auxiliary channel to a control process is set up, and a descriptor for it is placed in the *err_sd2p* parameter. The control process provides diagnostic output from the remote command on this channel. This diagnostoc information does not include remote authorization failure, since this connection is set up after authorization has been verified.

5.4.47.3 Return Values

Upon successful completion, the system returns a socket to the remote command.

If the rexec subroutine is unsuccessful, the system returns a -1 indicating that the specified host name does not exist.

5.4.47.4 Description

The rexec subroutine allows the calling process to start commands on a remote host.

If the rexec connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the calling process.

5.4.47.5 Example

```
int normsock;
char *host = NULL, *luser = NULL, *password = NULL, *cmd;
struct  servent *sp;

sp = getservbyname("exec", "tcp");

host = "remote _host";
luser = "my_userid";
password = "my_passwd";
cmd = "rempte_host_cmd"";

normsock = rexec(&host, sp->s_port, luser, password, cmd, &errsock);
if (normsock == -1)
       exit(-1);
```

5.4.48 select()

The select() call monitors read, write, and exception status on a group of sockets.

5.4.48.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

int select(s, noreads, nowrites, noexcepts, timeout)
int *s;
int noreads;
int nowrites;
int noexcepts;
long timeout;
```

5.4.48.2 Parameters

s	Pointer to an array of socket numbers in which the write socket numbers and the exception socket numbers follow the read socket numbers
noreads	Number of sockets to be checked for readability
nowrites	Number of sockets to be checked for readiness for writing
noexcepts	Number of sockets to be checked for exceptional pending conditions exceptional pending conditions
timeout	Maximum interval, in milliseconds, to wait for the selection to be complete

5.4.48.3 Return Values

The select() call returns the number of ready sockets. The value −1 indicates an error, and the value 0 indicates an expired time limit. If the return value is greater than 0, the socket numbers in *s* that were not ready are set to −1. You can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	The address is not valid.

5.4.48.4 Description

The select() call monitors activity on a set of various sockets until a time-out expires, to verify if any sockets are ready for reading or writing or if any exceptional conditions are pending.

If the time-out value is 0, select() does not wait before returning. If the time-out value is −1, select() does not time out; but it returns when a socket becomes ready. If the time-out value is a number of milliseconds, select() waits for the specified interval before returning.

See 5.3, “Porting a Socket API Application” on page 104, for information on how the 4690OS implementation of the select() call differs from the Berkeley implementation.

5.4.48.5 Example

```
#define MAX_TIMEOUT  1000

/* input_ready(insock)- Check to see if there is available input on
 * socket insock.
 * Returns 1 if input is available.
 *        0 if input is not available.
 *       −1 on error.
 */

 int input_ready(insock)
 int insock;                 /* input socket descriptor */

 {
   int socks[];    /* array of sockets */
   long timeout = MAX_TIMEOUT;

   /* put sockets to check in socks[] */
   socks[0] = insock;

   /* check for READ availability on this socket */
   return select(socks, 1, 0, 0, timeout);
 }
```

5.4.48.6 Related Calls

accept()
connect()
recv()
send()
sock_errno()

5.4.49 send()

The send() call sends data on a connected socket.

5.4.49.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

int send(s, msg, len, flags)
int s;
char *msg;
int len;
int flags;
```

5.4.49.2 Parameters

s
: Socket descriptor.

msg
: Pointer to the buffer containing the message to transmit.

len
: Length of the message pointed to by the *msg* parameter. The maximum is 32768.

flags
: Set by specifying one or more of the following flags. If you specify more than one flag, use the logical OR operator (|) to separate them. Setting this parameter is supported only for sockets in the AF_INET domain. The flags are as follows:

MSG_OOB
: Sends out-of-band data on sockets that support it. Only SOCK_STREAM sockets created in the AF_INET address family support out-of-band data.

MSG_DONTROUTE
: The SO_DONTROUTE option is turned on for the duration of the operation. Usually only diagnostic or routing programs use this.

5.4.49.3 Return Values

No indication of failure to deliver is implicit in a send() routine. The value −1 indicates locally detected errors; you can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *msg* and *len* parameters would result in an attempt to access memory outside the caller’s address space.
EINVAL	*len* is not the size of a valid address for the specified family.
ENOBUFS	No buffer space is available to send the message.
EWOULDBLOCK	The *s* parameter is in nonblocking mode, and no data is available to read.

5.4.49.4 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 sent, the send() call normally blocks, unless the socket is in nonblocking mode. (See 5.4.35, “ioctl()” on page 158 for a description of how to set nonblocking mode.) The select() call can be used to determine when to send more data.

5.4.49.5 Related Calls

connect()
getsockopt()
ioctl()
readv()
recv()
recvfrom()
select()
sendto()
sock_errno()
socket()
writev()

5.4.50 sendto()

The sendto() call sends packets on a datagram socket, regardless of its connection status.

5.4.50.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

int sendto(s, msg, len, flags, to, tolen)
int s;
char *msg;
int len;
int flags;
struct sockaddr *to;
int tolen;
```

5.4.50.2 Parameters

s
: Socket descriptor.

msg
: Pointer to the buffer containing the message to transmit.

len
: Length of the message pointed to by the *msg* parameter.

flags
: Set to 0 or MSG_DONTROUTE. Setting this parameter is supported only for sockets in the AF_INET 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.

to
: Address of the target.

tolen
: Size of the address pointed to by the *to* parameter.

5.4.50.3 Return Values

If it succeeds, sendto() returns the number of characters sent. The value −1 indicates an error; you can get the specific error code by calling sock_errno(). The return value of this call when used with datagram sockets does not imply failure to deliver.

Possible Error Code	**Description**
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *msg* and *len* parameters would result in an attempt to access memory outside the caller's address space.
EINVAL	*tolen* is not the size of a valid address for the specified address family.

EMSGSIZE	The message was too big to be sent as a single datagram. The default size is 8192 and the maximum size is 32 768.
ENOBUFS	No buffer space is available to send the message.
EWOULDBLOCK	The *s* parameter is in nonblocking mode and no data is available to read.
ENOTCONN	The socket is not connected.
EDESTADDRREQ	A destination address is required.

5.4.50.4 Description

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

5.4.50.5 Related Calls

readv()
recv()
recvfrom()
send()
select()
sock_errno()
socket()
writev()

5.4.51 sethostent()

The sethostent() call opens and rewinds the HOSTS file.

5.4.51.1 Syntax

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

5.4.51.2 Parameter

stayopen
  Allows the ADX_SDT1:ADXHSIHF.DAT file to stay open after each call

5.4.51.3 Return Value

A NULL pointer indicates an error or EOF.

5.4.51.4 Description

The sethostent() call opens and rewinds the ADX_SDT1:ADXHSIHF.DAT file. If the *stayopen* parameter is nonzero, the ADX_SDT1:ADXHSIHF.DAT file stays open after each of the gethost calls.

5.4.51.5 Related Calls

endhostent()
gethostbyaddr()
gethostbyname()
gethostent()

5.4.52 setnetent()

The setnetent() call opens and rewinds the NETWORKS file.

5.4.52.1 Syntax

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

5.4.52.2 Parameter

stayopen
: Causes the ADX_SDT1:ADXHSINF.DAT file to stay open after each call

5.4.52.3 Return Value

A NULL pointer indicates an error or EOF.

5.4.52.4 Description

The setnetent() call opens and rewinds the ADX_SDT1:ADXHSINF.DAT file, which contains information about known networks. If the *stayopen* parameter is nonzero, the ADX_SDT1:ADXHSINF.DAT file stays open after each of the getnet calls.

5.4.52.5 Related Calls

endnetent()
getnetbyaddr()
getnetbyname()
getnetent()

5.4.53 setprotoent()

The setprotoent() call opens and rewinds the PROTOCOL file.

5.4.53.1 Syntax

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

5.4.53.2 Parameter

stayopen
Causes the ADX_SDT1:ADXHSIPF.DAT file to stay open after each call

5.4.53.3 Return Value

A NULL pointer indicates an error or EOF.

5.4.53.4 Description

The setprotoent() call opens and rewinds the ADX_SDT1:ADXHSIPF.DAT file, which contains information about known protocols. If the *stayopen* parameter is nonzero, the ADX_SDT1:ADXHSIPF.DAT file stays open after each of the getproto calls.

5.4.53.5 Related Calls

endprotoent()
getprotobyname()
getprotobynumber()
getprotoent()

5.4.54 setservent()

The setservent() call opens and rewinds the SERVICES file.

5.4.54.1 Syntax

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

5.4.54.2 Parameter

stayopen
: Causes the ADX_SDT1:ADXHSISF.DAT file to stay open after each call

5.4.54.3 Return Value

A NULL pointer indicates an error or EOF.

5.4.54.4 Description

The setservent() call opens and rewinds the ADX_SDT1:ADXHSISF.DAT file, which contains information about known services and well-known ports. If the *stayopen* parameter is nonzero, the ADX_SDT1:ADXHSISF.DAT file stays open after each of the getserv calls.

5.4.54.5 Related Calls

endservent()
getservbyname()
getservbyport()
getservent()

5.4.55 setsockopt()

The setsockopt() call sets options associated with a socket.

5.4.55.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.55.2 Parameters

s
: Socket descriptor.

level
: Level for which the option is being set. Only SOL_SOCKET is supported.

optname
: Name of a specified socket option.

optval
: Pointer to option data.

optlen
: Length of the option data.

5.4.55.3 Return Values

The value 0 indicates success; the value −1 indicates an error. You can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
EADDRINUSE	The address is already in use.
ENOTSOCK	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 address space.
ENOPROTOOPT	The *optname* parameter is unrecognized, or the *level* parameter is not SOL_SOCKET.
EINVAL	*optlen* is not a valid size.
ENOBUFS	No buffer space is available.

5.4.55.4 Description

The setsockopt() call sets options associated with a socket. It can be called only for sockets in the AF_INET 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 name of the option and the level at which the option resides. To manipulate options at the socket level, the *level* parameter must be set to SOL_SOCKET, as defined in <SYS\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 level is 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* parameter 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*.

When you use socket-level options other than SO_LINGER, SO_SNDBUF, or SO_RCVBUF, *optval* points to an integer and *optlen* is set to the size of the integer. When the integer is nonzero, the option is enabled. When it is 0, the option is disabled. When you use the SO_LINGER option, *optval* points to a *linger* structure, as defined in <SYS\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 0 to disable the SO_LINGER option. 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 socket level:

Option	**Description**
SO_BROADCAST	Toggles the ability to broadcast messages. If you enable this option, the application can send broadcast messages over *s*, if the interface specified in the destination supports broadcasting of packets. This option has no meaning for stream sockets.
SO_DONTROUTE	Toggles the routing bypass for outgoing messages. When you enable this option, outgoing messages bypass the standard routing algorithm and are directed to the appropriate network interface according to the network portion of the destination address. When enabled, this option lets you send packets only to directly connected networks (networks for which this host has an interface). This option has no meaning for stream sockets.
SO_KEEPALIVE	Toggles keep connection alive. TCP uses a timer called the keepalive timer, which is used to monitor idle connections that might have been disconnected because of a peer failure or time-out. If you toggle this option, a keepalive packet goes to the peer every 120 minutes. This is used mainly to allow servers to close connections that have already disappeared as a result of clients going away without closing connections. This option has no meaning for stream sockets.
SO_LINGER	Lingers on close if data is present. If you enable this option while unsent data is present and soclose() is called, the calling application is blocked during the soclose() call until the data is transmitted or the connection has timed out. If this option is disabled, INET waits to try to send the data. Data transfer is usually successful, but it cannot be guaranteed because INET waits only a finite amount of time trying to send the data. The soclose() call returns without blocking the caller. Use this option only for stream sockets.

SO_OOBINLINE	Toggles the reception of out-of-band data. When you enable this option, out-of-band data is placed in the normal data input queue as it is received, making it available to recv() and recvfrom() without having to specify the MSG_OOB flag in those calls. Disabling this option causes out-of-band data to be placed in the priority data input queue as it is received, making it available to recv() and recvfrom() only by specifying the MSG_OOB flag in those calls. Use this option only for stream sockets.
SO_DEBUG	Toggles recording of debugging information.
SO_RCVBUF	Sets buffer size for input. This option sets the size of the receive buffer to the value contained in the buffer pointed to by *optval*. This lets you change the buffer size for specific application needs, such as increasing the buffer size for high-volume connections.
SO_RCVLOWAT	Sets receive low-water mark.
SO_RCVTIMEO	Sets receive time-out.
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 verifies at connect time that no local address and port have the same foreign address and port and returns the error EADDRINUSE if the association already exists.
SO_SNDBUF	Sets buffer size for output. This option sets the size of the send buffer to the value contained in the buffer pointed to by *optval*. This lets you change the send buffer size for specific application needs, such as increasing the buffer size for high-volume connections.
SO_SNDLOWAT	Sets send low-water mark.
SO_SNDTIMEO	Sets send time-out.
SO_USELOOPBACK	Bypasses when possible.

5.4.55.5 Example

```
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 5.4.26, “getsockopt()” on page 145 for examples of how the getsockopt() options set are queried.

5.4.55.6 Related Calls

getprotobyname()
getsockopt()
ioctl()
sock_errno()
socket()

5.4.56 shutdown()

The shutdown() call shuts down all or part of a duplex connection.

5.4.56.1 Syntax

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

5.4.56.2 Parameters

s

Socket descriptor

how

Condition of the shutdown

5.4.56.3 Return Values

The value 0 indicates success; the value −1 indicates an error. You can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EINVAL	The *how* parameter was not set to a valid value.

5.4.56.4 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*.

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

0	Ends communication from socket *s*
1	Ends communication to socket *s*
2	Ends communication both to and from socket *s*

5.4.56.5 Related Calls

accept()
connect()
sock_errno()
socket()
soclose()

5.4.57 sock_init()

The sock_init() call initializes the socket data structures and checks whether INET.SYS is running.

5.4.57.1 Syntax

```
int sock_init()
```

5.4.57.2 Return Values

The value 0 indicates success; the value 1 indicates an error.

5.4.57.3 Description

The sock_init() call initializes the socket data structures and checks whether INET.SYS is running. Therefore, sock_init() must be called at the beginning of each program that uses socket().

5.4.58 socket()

The socket() call creates an endpoint for communication and returns a socket descriptor representing the endpoint.

5.4.58.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.58.2 Parameters

domain
: The address domain requested. It must be AF_INET.

type
: Type of socket created: SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW.

protocol
: The protocol requested. Some possible values are 0, IPPROTO_UDP, or IPPROTO_TCP.

5.4.58.3 Return Values

A non-negative socket descriptor indicates success. The value −1 indicates an error; you can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
EPROTONOSUPPORT	The *protocol* is not supported in this *domain*, or this *protocol* is not supported for this socket *type*.
EPROTOTYPE	The *protocol* is the wrong type for the socket.

5.4.58.4 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 domain within which communication is to take place. This parameter selects the address family (format of addresses within a domain) which is used. The only family supported is AF_INET, which is the internet domain. This constant is defined in the <SYS\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 <SYS\SOCKET.H> header file. The types supported are:

SOCK_STREAM
: Provides sequenced, duplex byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data.

SOCK_DGRAM
Provides datagrams, which are connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times.

SOCK_RAW
Provides the interface to internal protocols (such as IP and ICMP).

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 ADX_SDT1:ADXHSIPF.DAT file. Alternatively, the getprotobyname() call can be used to get the protocol number for a protocol with a known name. Currently, *protocol* defaults are TCP for stream sockets and UDP for datagram sockets. There is no default for raw sockets.

SOCK_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 that 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. The application does this by setting the *internet address* field in the *address* structure to the constant INADDR_ANY, as defined in <SYS\SOCKET.H>.

After stream sockets become connected to each other, any of the data transfer calls can be used (send(), recv(), readv(), writev(), sendto(), or recvfrom()). Usually, a send-recv pair is used for sending data on stream sockets.

SOCK_DGRAM sockets model datagrams. They provide connectionless message exchange with no guarantees on reliability. Messages sent have a maximum size.

Unlike stream sockets, datagram sockets do not have active and passive uses. Servers must still call bind() to name a socket and to specify from which network interfaces it wants 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.

After an application receives a datagram socket, it can exchange datagrams using the sendto() and recvfrom() 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[6] can also be used. For more information about placing a socket into the connected state, see 5.4.4, “connect()” on page 114.

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 the class of address and whether subnets are being used). The constant INADDR_BROADCAST, defined in <SYS\SOCKET.H>, can be used to broadcast to the primary network if the primary configured network supports broadcasting.

[6] readv(), writev(), send() and recv()

SOCK_RAW sockets give the application an interface to lower-layer protocols, such as IP and ICMP. The application often uses this interface to bypass the transport layer when direct access to lower-layer protocols is needed and to test new protocols.

Raw sockets are connectionless; the data transfer abilities 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.

The soclose() call deallocates sockets.

5.4.58.5 Example

```
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 raw socket in internet domain for ICMP protocol */
p = getprotobyname("icmp");
s = socket(AF_INET, SOCK_RAW, p->p_proto);
```

5.4.58.6 Related Calls

accept()
bind()
connect()
getprotobyname()
getsockname()
getsockopt()
ioctl()
port_cancel()
readv()
recv()
recvfrom()
select()
send()
sendto()
shutdown()
sock_errno()
soclose()
writev()

5.4.59 soclose()

The soclose() call shuts down a socket and frees resources allocated to that socket.

5.4.59.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

int soclose(s)
int s;
```

5.4.59.2 Parameter

s

Descriptor of the socket to discard

5.4.59.3 Return Values

The value 0 indicates success; the value −1 indicates an error. You can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EALREADY	The socket *s* is marked nonblocking, and a previous connection attempt has not completed.
ENOTCONN	The socket is not connected.

5.4.59.4 Description

The soclose() 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.

5.4.59.5 Related Calls

accept()
port_cancel()
sock_errno()
socket()

5.4.60 writev()

The writev() call writes data on a socket with descriptor *s*.

5.4.60.1 Syntax

```
#include <types.h>
#include <sys\socket.h>

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

5.4.60.2 Parameters

s
: Socket descriptor

iov
: Pointer to an array of iovec buffers

iovcnt
: Number of buffers pointed to by the *iov* parameter

5.4.60.3 Return Values

If it succeeds, the call returns the number of bytes written from the buffers. The value −1 indicates an error. You can get the specific error code by calling sock_errno().

Possible Error Code	**Description**
SOCENOTSOCK	*s* is not a valid socket descriptor.
SOCEFAULT	Using the *iov* and *iovcnt* parameters would result in an attempt to access memory outside the caller's address space.
SOCEINVAL	*iovcnt* was not valid, or one of the fields in the *iov* array was not valid.
SOCENOBUFS	Buffer space is not available to send the message.
SOCEWOULDBLOCK	*s* is in nonblocking mode, and data is not available to read.
SOCEMSGSIZE	The message was too big to be sent as a single datagram. The default size is 8192 and the maximum size is 32 768.
SOCEDESTADDRREQ	A destination address is required.

5.4.60.4 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 SYS\SOCKET.H and contains the following fields:

Element	**Description**
iov_base	Pointer to the buffer
iov_len	Length of the buffer

The writev() call applies only to connected sockets.

This call writes *iov_len* bytes of data. If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, writev() blocks the caller until additional buffer space becomes available. If the socket is in a nonblocking mode, writev() returns a −1 and sets the error code to SOCEWOULDBLOCK. See 5.4.35, “ioctl()” on page 158 for a description of how to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, providing the datagram fits into the TCPIP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application sends 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets must place this call in a loop, calling this function until all data has been sent.

5.4.60.5 Related Calls

connect()
getsockopt()
ioctl()
readv()
recv()
recvfrom()
select()
send()
sendto()
setsockopt()
sock_errno()
socket()

6.0 Remote Procedure Calls (RPCs)

This chapter describes the high-level remote procedure calls (RPCs) implemented in TCP/IP for 4690OS, including the RPC programming interface to the C language and communication between processes.

The RPC protocol enables 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 protocol enables users to work with remote procedures as if the procedures were local. The remote procedure calls are defined through routines contained in the RPC protocol. Each call message is matched with a reply message. The RPC protocol is a message-passing protocol that implements other non-RPC protocols, such as batching and broadcasting remote calls. The RPC protocol also supports callback procedures and the select subroutine on the server side.

RPC provides an authentication process that identifies the server and client to each other. RPC includes a slot for the authentication parameters on every remote procedure call so that the caller can identify itself to the server. The client package generates and returns authentication parameters. RPC supports various types of authentication, such as the UNIX** systems.

In RPC, each server supplies a program that is a set of procedures. The combination of a host address, a program number, and a procedure number specifies one remote service procedure. In the RPC model, the client makes a procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, and sends a reply back to the client. The procedure call then returns to the client.

RPC is divided into two layers: intermediate and lowest. Generally, you use the RPC interface to communicate between processes on different workstations in a network. However, RPC works just as well for communication between different processes on the same workstation.

The Portmapper** program maps RPC program and version numbers to a transport-specific port number. The Portmapper program makes dynamic binding of remote programs possible.

To write network applications using RPC, programmers need a working knowledge of network theory and C programming language. For most applications, understanding the RPC mechanisms usually hidden by the RPCGEN protocol compiler is also helpful. However, RPCGEN makes understanding the details of RPC unnecessary. Figure 14 on page 202 and Figure 15 on page 203 give an overview of the high-level RPC client and server processes from initialization through cleanup.

For more information about the RPC and XDR protocols, see the Sun Microsystems publication, *Networking on the Sun Workstation: Remote Procedure Call Programming Guide*, RFC 1057 and RFC 1014.

6.1 The RPC Interface

The RPC model is similar to the local procedure call model. In the local model, the caller places the argument to a procedure in a specified location such as a result register. Then, the caller transfers control to the procedure. The caller eventually regains control, extracts the results of the procedure, and continues the execution.

RPC works in the same way: One thread of control winds logically through the caller and server processes as follows:

1. The caller process sends a call message that includes the procedure parameters to the server process and then waits for a reply message (blocks).
2. A process on the server side, which is dormant until the arrival of the call message, extracts the procedure parameters, computes the results, and sends a reply message. Then the server waits for the next call message.
3. A process on the caller side receives the reply message and extracts the results of the procedure. The caller then resumes the execution.

See Figure 14 and Figure 15 on page 203 for an illustration of the RPC model.

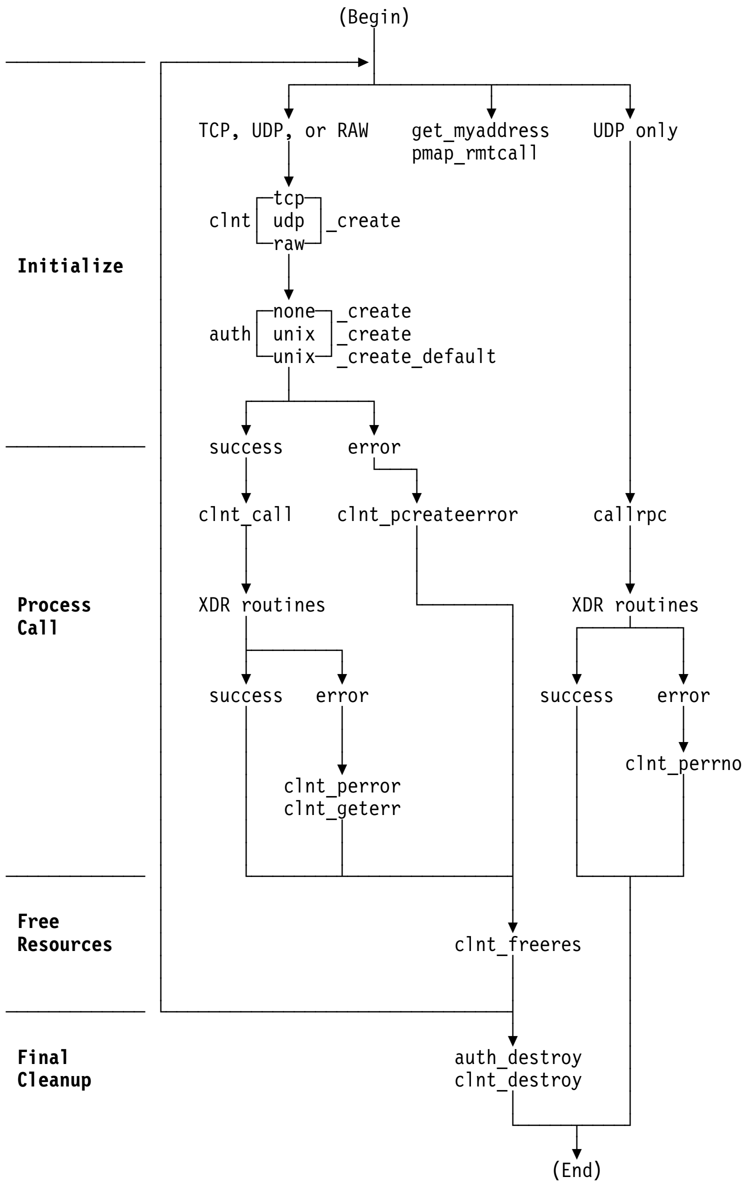


Figure 14. Remote Procedure Call (Client)

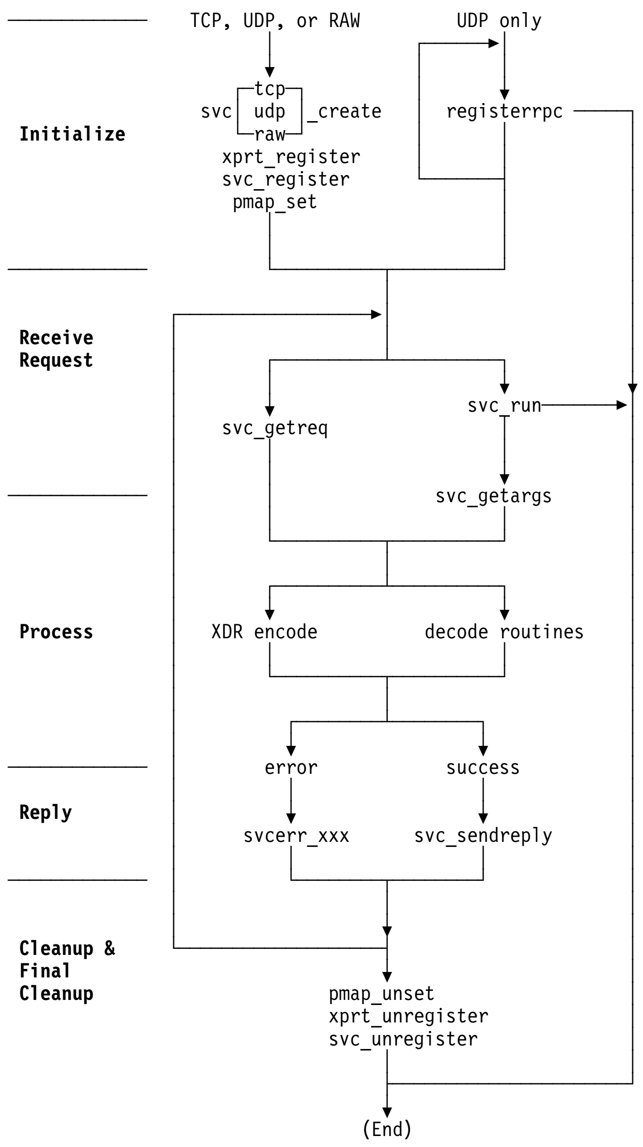


Figure 15. Remote Procedure Call (Server)

6.2 Remote Programs and Procedures

The RPC call message has three unsigned fields:

- Remote program number
- Remote program version number
- Remote procedure number

The three fields uniquely identify the procedure to be called. The program number defines a group of related remote procedures, each of which has a different procedure number. Each program also has a version number.

The central system authority administers the program number. A remote program number is assigned by groups of 0x20000000, as shown in the following list:

Program Number	**Description of Each Group**
0-1xxxxxxx	Is predefined and administered by the 4690OS TCP/IP system.
20000000-3xxxxxxx	Represents the user defined numbers
40000000-5xxxxxxx	Represents transient numbers
60000000-7xxxxxxx	Reserved
80000000-9xxxxxxx	Reserved
a0000000-bxxxxxxx	Reserved
c0000000-dxxxxxxx	Reserved
e0000000-fxxxxxxx	Reserved

6.3 Portmapper

The Portmapper protocol defines a network service that clients use to look up the port number of any remote program supported by the server. The client programs must find the port numbers of the server programs that they intend to use.

The Portmapper program:

- Maps RPC program and version numbers to transport specific port numbers.
- Makes dynamic binding of remote programs. This is desirable because the range of reserved port numbers is small, and the number of potential remote programs is large. When running only the Portmapper program on a reserved port, you can determine the port numbers of other remote programs by querying Portmapper.
- Supports both the UDP and TCP protocols.

The RPC client contacts Portmapper on port number 111 on either of these protocols.

6.3.1 Registering and Unregistering a Port with Portmapper

Portmapper is the only network service that must have a dedicated port (111). Other RPC network services can be assigned port numbers statically or dynamically, if the services register their ports with the host's local Portmapper. The RPC server can register or unregister their services by using the following calls:

svc_register()	Associates a program with the service dispatch routine

svc_unregister()	Removes all local mappings to dispatch routines and port numbers
registerrpc()	Registers a procedure with the local Portmapper and creates a control structure to remember the server procedure and its XDR routine

6.3.2 Contacting Portmapper

To find the port of a remote program, the client sends an RPC request to well-known port 111 of the server's host. If Portmapper has a port number entry for the remote program, Portmapper provides the port number in the RPC reply. The client then requests the remote program by sending an RPC request 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.

RPC also provides the following calls for interfacing with Portmapper:

pmap_getmaps()	Returns a list of current program-to-port mappings on the foreign host
pmap_getport()	Returns the port number associated with the remove program, version, and transport protocol
pmap_rmtcall()	Instructs Portmapper to make an RPC call to a procedure on the host
pmap_set()	Sets the mapping of a program to a port on the local machine
pmap_unset()	Removes mappings associated with the program and version number on the local machine
xdr_pmap()	Translates an RPC procedure identification
xdr_pmaplist()	Translates a variable number of RPC procedure identifications

6.3.3 Portmapper Procedures

The Portmapper program supports the following procedures:

Procedure	**Description**
PMAPPROC_NULL	Has no parameters. A caller can use the return code to determine if Portmapper is running.
PMAPPROC_SET	Registers itself with the Portmapper program on the same machine. It passes the: • Program number • Program version number • Transport protocol number • Port number The procedure has successfully established the mapping if the return value is TRUE. The procedure does not establish a mapping if one already exists.
PMAPPROC_UNSET	Unregisters the program and version numbers with Portmapper on the same machine.
PMAPPROC_GETPORT	Returns the port number when given a program number, version number, and transport protocol number. A port value of 0 indicates the program has not been registered.
PMAPPROC_DUMP	Takes no input, but returns a list of program, version, protocol, and port numbers.
PMAPPROC_CALLIT	Allows a caller to call another remote procedure on the same machine without knowing the remote procedure's port number. The PMAPPROC_CALLIT procedure sends a response only if the procedure is successfully run.

6.4 eXternal Data Representation (XDR)

An eXternal Data Representation (XDR) is a data representation standard that is independent of languages, operating systems, manufacturers, and hardware architecture. This standard enables networked computers to share data regardless of the machine on which the data is produced or consumed. The XDR language permits transfer of data between diverse computer architectures and has been used to communicate data between diverse machines.

An XDR approach to standardizing data representations is canonical. That is, XDR defines a single byte (big endian), a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standards. Similarly, any program running on any machine can read portable data by translating the XDR standard representations to its local equivalents.

The XDR standard is the backbone of the RPC, because data for remote procedure calls is sent using the XDR standard.

To use XDR routines, C programs must include the <RPC\XDR.H> header file, which is automatically included by the <RPC\RPC.H> header file.

6.4.1 Basic Block Size

The XDR language is based on the assumption that bytes (an octet) can be ported to, and encoded on, media that preserve the meaning of the bytes across the hardware boundaries of data. XDR does not represent bit fields or bit maps; it represents data in blocks of multiples of 4 bytes (32 bits). If the bytes needed to contain the data are not a multiple of four, enough (0 to 3) bytes to make the total byte count a multiple of four follow the *n* bytes. The bytes are read from, or written to, a byte stream in order. The order dictates that byte *m* precedes *m*+1. Bytes are ported and encoded from low order to high order in local area networks (LANs). Representing data in standardized formats resolves situations that occur when different byte-ordering formats exist on networked machines. This also enables machines with different structure-alignment algorithms to communicate with each other.

6.4.2 The XDR Subroutine Format

An XDR routine is associated with each data type. XDR routines have the following format:

```
xdr_xxx(xdrs,dp)
      XDR *xdrs;
      xxx *dp;
{
}
```

The routine has the following parameters:

xxx	XDR data type.
xdrs	Opaque handle that points to an XDR stream. The system passes the opaque handle pointer to the primitive XDR routines.
dp	Address of the data value that is to be encoded or decoded.

If they succeed, the XDR routines return a value of 1; if they do not succeed, they return a value of 0.

6.4.3 XDR Data Types and their Filter Primitives

The following basic and constructed data types are defined in the XDR standard:

- Integers
- Enumeration
- Booleans
- Floating-point decimals
- Opaque data
- Arrays
- Strings
- Structures
- Discriminated unions
- Void
- Constants
- Typedef
- Optional data
- Pointers

The XDR filter primitives are routines that define the basic and constructed data types. The XDR language provides RPC programmers with a specification for uniform representation that includes filter primitives for basic and constructed data types.

The basic data types include:

- Integers
- Enumeration
- Booleans
- Floating point decimals
- Void
- Constants
- Typedef
- Optional data

The constructed data types include:

- Arrays
- Opaque data
- Strings
- Byte arrays
- Structures
- Discriminated unions
- Pointers

The XDR standard translates both basic and constructed data types. For basic data types such as integer, XDR provides basic filter primitives that:

- Serialize information from the local host's representation to XDR representation
- Deserialize information from the XDR representation to the local host's representation

For constructed data types, XDR provides constructed filter primitives that allow the use of basic data types (such as integers and floating-point numbers) to create more complex constructs (such as arrays and discriminated unions).

6.4.3.1 Integer Filter Primitives

The XDR filters cover signed and unsigned integers, as well as signed and unsigned short and long integers.

The routines for XDR integer filters are:

xdr_int()	Translates between C integers and their external representations
xdr_u_int()	Translates between C unsigned integers and their external representations
xdr_long()	Translates between C long integers and their external representations
xdr_u_long()	Translates between C unsigned long integers and their external representations
xdr_short()	Translates between C short integers and their external representations
xdr_u_short()	Translates between C unsigned short integers and their external representations

6.4.3.2 Enumeration Filter Primitives

The XDR library provides a primitive for generic enumerations based on the assumption that a C enumeration value (enum) has the same representation. A special enumeration in XDR, known as the *Boolean*, provides a value of 0 or 1 represented internally in a binary notation.

The routines for the XDR library enumeration filters are:

xdr_enum()	Translates between C language enums and their external representations
xdr_bool()	Translates between Booleans and their external representations

6.4.3.3 Floating-Point Filter Primitives

The XDR library provides primitives that translate between floating-point data and their external representations. Floating-point data encodes an integer with an exponent. Floats and double-precision numbers compose floating-point data.

Note: Numbers are represented as Institute of Electrical and Electronics Engineers (IEEE) standard floating points. Routines might fail when decoding IEEE representations into machine specific representations.

The routines for the XDR floating-point filters are:

xdr_float()	Translates between C language floats and their external representations
xdr_double()	Translates between C language double-precision numbers and their external representations

6.4.3.4 Opaque Data Filter Primitive

Opaque data is composed of bytes of a fixed size that are not interpreted as they pass through the data streams. Opaque data bytes, such as handles, are passed between server and client without being inspected by the client. The client uses the data as it is and then returns it to the server. By definition, the actual data contained in the opaque object is not portable between computers.

The XDR library includes the following routine for opaque data:

xdr_opaque()	Translates between opaque data and its external representation

6.4.3.5 Array Filter Primitives

Arrays are constructed filter primitives that can be generic arrays or byte arrays. The XDR library provides filter primitives for handling both types of arrays.

6.4.3.5.1 Generic Arrays: These consist of arbitrary elements. You use them in much the same way as byte arrays, which handle a subset of generic arrays where the size of the elements is 1 and their external descriptions are predetermined. The primitive for generic arrays requires an additional parameter to define the size of the element in the array and to call an XDR routine to encode or decode each element in the array.

The XDR library includes the following routines for generic arrays:

xdr_array()	Translates between variable-length arrays and their corresponding external representations
xdr_vector()	Translates between fixed-length arrays and their corresponding external representations

6.4.3.5.2 Byte Arrays: These differ from strings by having a byte count. That is, the length of the array is set to an unsigned integer. They also differ in that byte arrays do not end with a null character. The XDR library provides a primitive for byte arrays. External and internal representations of byte arrays are the same.

The XDR library includes the following routine for byte arrays:

xdr_bytes()	Translates between counted byte string arrays and their external representations

6.4.3.6 String Filter Primitives

A string is a constructed filter primitive that consists of a sequence of bytes terminated by a null byte. The null byte does not figure into the length of the string. Externally, strings are represented by a sequence of American Standard Code Information Interchange (ASCII) characters. Internally, XDR represents them as pointers to characters with the designation `char *`.

The XDR library includes primitives for the following string routines:

xdr_string()	Translates between C language strings and their external representations
xdr_wrapstring()	Calls the xdr_string subroutine

6.4.3.7 Primitive for Pointers to Structures

The XDR library provides the primitive for pointers so that structures referenced within other structures can be easily serialized, deserialized, and released.

The XDR library includes the following routine for pointers to structures:

xdr_reference()	Provides pointer chasing within structures

6.4.3.8 Primitive for Discriminated Unions

A discriminated union is a C language union, which is an object that holds several data types. One arm of the union contains an enumeration value (enum_t), or discriminant, that holds a specific object to be processed over the system first.

The XDR library includes the following routine for discriminated unions:

xdr_union()	Translates between discriminated unions and their external representations

6.4.3.9 Passing Routines without Data

Sometimes an XDR routine must be supplied to the RPC system, but no data is required or passed. The XDR library provides the following primitive for this function:

xdr_void()	Supplies an XDR subroutine to the RPC system without sending data

6.4.4 XDR Nonfilter Primitives

Use the XDR nonfilter primitives to create, manipulate, implement, and destroy XDR data streams. These primitives allow you to:

- Describe the data stream position
- Change the data stream position
- Destroy a data stream

6.4.4.1 Creating and Using XDR Data Streams

You get XDR data streams by calling creation routines that take arguments specifically designed to the properties of the stream. There are existing XDR data streams for serializing or deserializing data in standard input and output streams, memory streams, and record streams.

Note: RPC clients do not have to create XDR streams, because the RPC system creates and passes these streams to the client.

The types of data streams include:

- Standard I/O streams
- Memory streams
- Record streams

6.4.4.1.1 Standard I/O Streams: XDR data streams serialize and deserialize standard input/output(I/O) by calling the standard I/O creation routine to initialize the XDR data stream pointed to by the *xdrs* parameter.

The XDR library includes the following routine for standard I/O data streams:

xdrstdio_create()	Initializes the XDR data stream pointed to by the *xdrs* parameter

6.4.4.1.2 Memory Streams: XDR data streams serialize and deserialize data from memory by calling the XDR memory creation routine to initialize, in local memory, the XDR stream pointed at by the *xdrs* parameter. In RPC, the UDP/IP implementation of remote procedure calls uses this routine to build entire call and reply messages in memory before sending the message to the recipient.

The XDR library includes the following routine for memory data streams:

xdrmem_create()	Initializes, in local memory, the XDR stream pointed to by the *xdrs* parameter

6.4.4.1.3 Record Streams: Record streams are XDR streams built on top of record fragments, which are built on TCP/IP streams. TCP/IP is a connection protocol for transporting large streams of data at one time rather than transporting a single data packet at a time.

The primary use of a record stream is to interface remote procedure calls to TCP connections. It can also be used to stream data into or out of normal files.

XDR provides the following routines for use with record streams:

xdrrec_create()	Provides an XDR stream that can contain long sequences of records
xdrrec_endofrecord()	Causes the current outgoing data to be marked as a record
xdrrec_skiprecord()	Causes the position of an input stream to move to the beginning of the next record
xdrrec_eof()	Checks the buffer for an input stream that identifies the end of file (EOF)

6.4.4.2 Manipulating an XDR Data Stream

XDR provides the following routines for describing the data stream position and changing the data stream position:

the current position in the data stream

6.4.4.3 Implementing an XDR Data Stream

You can create and implement XDR data streams. The following example shows the abstract data types (XDR handle) required for you to implement your own XDR streams. They contain operations applied to the stream (an operation vector for the particular implementation) and two private fields for using that implementation.

```
enum xdr_op  { XDR_ENCODE=0, XDR_DECODE=1, XDR_FREE=2 };
typedef struct xdr {
        enum xdr_op  x_op;
        struct xdr_ops {
                bool_t  (*x_getlong)(struct xdr *, long *);
                bool_t  (*x_putlong)(struct xdr *, long *);
                bool_t  (*x_getbytes)(struct xdr *, caddr_t, u_int);
            /* get some bytes from " */
                bool_t  (*x_putbytes)(struct xdr *, caddr_t, u_int);
            /* put some bytes to " */
                u_int   (*x_getpostn)(struct xdr *);
                bool_t  (*x_setpostn)(struct xdr *,u_int);
                long *  (*x_inline)(struct xdr *,u_int);
                void    (*x_destroy)(struct xdr *);
        } *x_ops;
        caddr_t         x_public;
        caddr_t         x_private;
        caddr_t         x_base;
        int             x_handy;
 } XDR;
```

The following parameters are pointers to XDR stream manipulation routines:

Parameter	**Description**
x_getlong	Gets long integer values from the data stream.
x_putlong	Puts long integer values into the data stream.
x_getbytes	Gets bytes from the data streams.
x_putbytes	Puts bytes into the data streams.
x_getpostn	Returns the stream offset.
x_setpostn	Repositions the offset.
x_inline	Points to an internal data buffer, used for any purpose.
x_destroy	Frees the private data structure.
x_ops	Specifies the current operation being performed on the stream. This field is important to the XDR primitives, but the stream's implementation does not depend on the value of this parameter.

The following fields are specific to a stream's implementation:

Field	**Description**
x_public	Specific user data that is private to the stream's implementation and that is not used by the XDR primitive
x_private	Points to the private data
x_base	Contains the position information in the data stream that is private to the user implementation
x_handy	Data can contain extra information as necessary

6.4.4.4 Destroying an XDR Data Stream

XDR provides a routine that destroys the XDR stream pointed to by the *xdrs* parameter and frees the private data structures allocated to the stream.

xdr_destroy()	Destroys the XDR stream pointed to by the *xdrs* parameter

The use of the XDR stream handle is undefined after it is destroyed.

6.5 RPC Intermediate Layer

The calls of the RPC intermediate layer are:

registerrpc()	Registers a procedure with the local Portmapper
callrpc()	Calls a remote procedure on the specified system
svc_run()	Accepts RPC requests and calls the appropriate service using svc_getreq()

The transport mechanism is the User Datagram Protocol (UDP). The UDP transport mechanism handles only arguments and results that are less than 8K bytes in length. At this level, RPC does not allow time-out specifications, choice of transport, or process control, in case of errors. If you need this kind of control, consider the lowest layer of RPC.

With only these three RPC calls, you can write a powerful RPC-based network application. The sequence of events follows:

1. Use the registerrpc() call to register your remote program with the local Portmapper. See 6.3, "Portmapper" on page 204 for more information. The following is an example of an RPC server:

   ```
   /* define remote program number and version */

   #define RMTPROGNUM (u_long)0x3fffffffL
   #define RMTPROGVER (u_long)0x1
   #define RMTPROCNUM (u_long)0x1

   #include <stdio.h>
   #include <rpc\rpc.h>


   main()
    {
     int *rmtprog();

     /* register remote program with portmapper */
     registerrpc(RMTPROGNUM, RMTPROGVER, RMTPROCNUM, rmtprog,
                        xdr_int, xdr_int);
     /* infinite loop, waits for RPC request from client */
     svc_run();
     printf("Error: svc_run should never reach this point \n");
     exit(1);
    }

   int *
   rmtprog(inproc)          /* remote program */
   int *inproc;

   {
    int *outproc;
    ...
    /* Process request */
    ...
    return (outproc);
   }
   ```

   The registerrpc() call registers a C procedure rmtprog, which corresponds to a given RPC procedure number.

   The registerrpc() call has six parameters:

   - The first three parameters, RMTPROGNUM, RMTPROGVER, and RMTPROCNUM, are the program, version, and procedure numbers of the remote procedure to be registered.
   - The fourth parameter, rmtprog, is the name of the local procedure that implements the remote procedure.
   - The last two parameters, xdr_int, are the XDR filters for the remote procedure's arguments and results.

   After registering a procedure, the RPC server goes into an infinite loop waiting for a client request to service.

2. The RPC client uses callrpc() to make a service request to the RPC server. The following is an example of an RPC client using the callrpc() call:

   ```
   /* define remote program number and version */
   ```

```
#define RMTPROGNUM (u_long)0x3fffffffL
#define RMTPROGVER (u_long)0x1
#define RMTPROCNUM (u_long)0x1

#include <stdio.h>
#include <rpc\rpc.h>


main()
{
  int inproc=100, outproc, rstat;

  ...

  /* service request to host RPCSERVER_HOST */
  if (rstat = callrpc("RPCSERVER_HOST", RMTPROGNUM,
              RMTPROGVER, RMTPROCNUM, xdr_int, (char *)&inproc,
              xdr_int, (char *)&outproc)!= 0)
     {
       clnt_perrno(rstat);   /* Why  callrpc() failed ? */
       exit(1);
     }
  ...

 }
```

The callrpc() call has eight parameters:

- The first is the name of the remote server machine.
- The next three parameters are the program, version, and procedure numbers.
- The fifth and sixth parameters are an XDR filter, and an argument to be encoded and passed to the remote procedure.
- The final two parameters are a filter for decoding the results returned by the remote procedure, and a pointer to the place where the procedure's results are to be stored.

You handle multiple arguments and results by embedding them in structures. The callrpc() call returns 0 if it succeeds, otherwise nonzero. The exact meaning of the returned code is in the <RPC\CLNT.H> header file and is an enum *clnt_stat* structure cast into an integer.

6.6 RPC Lowest Layer

Use the lowest layer of RPC in the following situations:

- You need to use TCP. The intermediate layer uses UDP, which restricts RPC calls to 8K bytes of data. TCP permits calls to send long streams of data.
- You want to allocate and free memory while serializing or deserializing messages with XDR routines. No RPC call at the intermediate level explicitly permits freeing memory. XDR routines are used for memory allocation as well as for serializing and deserializing.
- You need to perform authentication on the client side or the server side by supplying credentials or verifying them.

6.6.1.1 Server Side Program

The following is an example of the lowest layer of RPC on the server side program:

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L
#define LONGPROC   1
#define STRINGPROC 2

#define MAXLEN 100

#include <stdio.h>
#include <rpc\rpc.h>
#include <sys\socket.h>

main(argc, argv)
int argc;
char *argv[ ];

{
     int rmtprog();
     SVCXPRT *transp;

     ...

/* create TCP transport handle */
     transp = svctcp_create(RPC_ANYSOCK, 1024*10, 1024*10);
/* or create UDP transport handle */
/*   transp = svcudp_create(RPC_ANYSOCK);  */
     if (transp == NULL)   /* check transport handle creation */
      {
        fprintf(stderr, "can't create an RPC server transport\n");
        exit(-1);
      }

/* If exists, remove the mapping of remote program and port */
     pmap_unset(RMTPROGNUM, RMTPROGVER);

/* register remote program (TCP transport) with local portmapper */
     if (!svc_register(transp, RMTPROGNUM, RMTPROGVER, rmtprog,
                         IPPROTO_TCP))
/* or register remote program (UDP transport) with local portmapper */
/*   if (!svc_register(transp, RMTPROGNUM, RMTPROGVER, rmtprog,*/
                      /* IPPROTO_UDP)) */
      {
        fprintf(stderr, "can't register rmtprog() service\n");
        exit(-1);
      }

     svc_run();
     printf("Error:svc_run should never reaches this point \n");
     exit(1);

}

rmtprog(rqstp, transp)           /* code for remote program */
struct svc_req *rqstp;
SVCXPRT *transp;
{
   long in_long,out_long;
   char buf[100], *in_string=buf, *out_string=buf;
   ...
   switch((int)rqstp->rq_proc)   /* Which procedure ? */
    {
      case NULLPROC:
         if (!svc_sendreply(transp,xdr_void, 0))
          {
            fprintf(stderr,"can't reply to RPC call\n");
            exit(-1);
          }
         return;

      case LONGPROC:
         ...
         /* Process the request */
         if (!svc_sendreply(transp,xdr_long,&out_long))
          {
            fprintf(stderr,"can't reply to RPC call\n");
            exit(-1);
          }
         return;

      case STRINGPROC:   /* send received "Hello" message back */
                                   /* to client */
         svc_getargs(transp,xdr_wrapstring,(char *)&in_string);
         strcpy(out_string,in_string);

         /* send a reply back to a RPC client */
         if (!svc_sendreply(transp,xdr_wrapstring,
                                         (char *)&out_string))
          {
            fprintf(stderr,"can't reply to RPC call\n");
            exit(-1);
          }
         return;
      case ... :
         ...
         /* Any Remote procedure in RMTPROGNUM program */
         ...
      default:
         /* Requested procedure not found */
         svcerr_noproc(transp);
         return;
    }
}
```

The following steps describe the lowest layer of RPC on the server side program:

1. Service the transport handle.

   The svctcp_create() and svcudp_create() calls create TCP and UDP transport handles (SVCXPRT) respectively, used for receiving and replying to RPC messages. The SVCXPRT transport handle structure is defined in the <RPC\SVC.H> header file.

   If the argument of the svctcp_create() call is RPC_ANYSOCK, the RPC library creates a socket on which to receive and reply to remote procedure calls. The svctcp_create() and clnttcp_create() calls cause the RPC library calls to bind the appropriate socket, if it is not already bound.

   If the argument of the svctcp_create() call is not RPC_ANYSOCK, the svctcp_create() call expects its argument to be a valid socket number. If you specify your own socket, it can be bound or unbound. If it is bound to a port by you, the port numbers of the svctcp_create() and clnttcp_create() calls must match.

If the send and receive buffer size parameter of svctcp_create() is 0, the system selects a reasonable default.

2. Register the rmtprog service with Portmapper.

   If the rmtprog service terminated abnormally the last time it was used, the pmap_unset() call erases any trace of it before restarting. The pmap_unset() call erases the entry for RMTPROGNUM from the Portmapper's table.

   A service can register its port number with the local Portmapper service by specifying a nonzero protocol number in the svc_register() call. A programmer at the client machine can determine the server port number by consulting Portmapper at the server machine. You can do this automatically by specifying 0 as the port number in the clntudp_create() or clnttcp_create() calls.

   Finally, the program and version number are associated with the rmtprog procedure. The final argument to the svc_register() call is the protocol being used, which in this case is IPPROTO_TCP. Register at the program level, not at the procedure level.

3. Run the remote program RMTPROG.

   The rmtprog service routine must call and dispatch the appropriate XDR calls based on the procedure number. Unlike the registerrpc() call, which performs them automatically, the rmtprog routine requires two tasks:

   - When the NULLPROC procedure (currently 0) returns with no results, use it as a simple test for detecting whether a remote program is running.
   - Check for incorrect procedure numbers. If you detect one, call the svcerr_noproc() call to handle the error.

   As an example, the procedure STRINGPROC has an argument for a character string and returns the character string back to the client. The svc_getargs() call takes an SVCXPRT handle, the xdr_wrapstring() call, and a pointer that indicates where to place the input.

   The user service (rmtprog) serializes the results and returns them to the RPC caller through the svc_sendreply() call.

   Parameters of the svc_sendreply() call include the:

   - SVCXPRT handle
   - XDR routine, which indicates return data type
   - Pointer to the data to be returned

6.6.1.2 Client Side Program

The following is an example of the lowest layer of RPC on the client side program:

```
#define RMTPROGNUM  (u_long)0x3fffffffL
#define RMTPROGVER  (u_long)0x1L
#define STRINGPROC  (u_long)2

#include <stdio.h>
#include <rpc\rpc.h>
#include <sys\socket.h>
#include <netdb.h>

main(argc, argv)
int argc;
char *argv[ ];
{
   struct hostent *hp;
   struct timeval pertry_timeout, total_timeout;
   struct sockaddr_in server_addr;
   int sock = RPC_ANYSOCK;
   static char buf[100], *strc_in= "Hello", *strc_out=buf;
   char *parrc_in, *parrc_out;
   register CLIENT *clnt;
   enum clnt_stat cs;
   ...
   /* get the Internet address of RPC server host */
   if ((hp = gethostbyname("RPCSERVER_HOST")) == NULL)
    {
      fprintf(stderr,"Can't get address for %s\n","RPCSERVER_HOST");
      exit (-1);
    }

   pertry_timeout.tv_sec = 3;
   pertry_timeout.tv_usec = 0;

   /* set sockaddr_in structure */
   bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr.s_addr,
                       hp->h_length);
   server_addr.sin_family = AF_INET;
   server_addr.sin_port = 0;

   /* create clnt TCP handle */
   if ((clnt = clnttcp_create(&server_addr, RMTPROGNUM, RMTPROGVER,
                          &sock, 1024*10, 1024*10)) == NULL)
     {
       clnt_pcreateerror("clnttcp_create fail"); /* Why failed ? */
       exit(-1);
     }

/*
 *  create clnt UDP handle
 *  if ((clnt = clntudp_create(&server_addr, RMTPROGNUM, RMTPROGVER,
 *                            pertry_timeout, &sock)) == NULL)
 *   {
 *     clnt_pcreateerror("clntudp_create fail");
 *     exit(-1);
 *   }
 */
   total_timeout.tv_sec = 10;
   total_timeout.tv_usec = 0;
   ...

   /*call the remote procedure STRINGPROC associated with */
   /*client handle (clnt) */
   cs=clnt_call(clnt, STRINGPROC,xdr_wrapstring,
   (char *)&strc_in[j],
                xdr_wrapstring, (char *)&strc_out,total_timeout);
        if (cs != RPC_SUCCESS)
              printf("*Error* clnt_call fail :\n");

   clnt_destroy(clnt);  /* deallocate any memory associated  */
                        /* with clnt handle                  */
   ...
}
```

The following steps describe the lowest layer of RPC on the client side program:

1. Determine the internet address of the RPC server host.

   Use the gethostbyname() call to determine the internet address of the host, which is running the RPC server. Initialize the *socaddr_in* structure, found in the <NETINET\IN.H> header file.

   If you are not familiar with socket calls, see 5.0, “Sockets.”

2. Use the client RPC handle.

   The clnttcp_create() and clntudp_create() calls create TCP and UDP client RPC handles (CLIENT), respectively. The CLIENT structure is defined in the <RPC\CLNT.H> header file.

   There are six parameters for the clnttcp_create() call:

   - Server address
   - Program number
   - Version number
   - Pointer to a valid socket descriptor
   - Send buffer size
   - Receive buffer size

   Use the same parameters for the clntudp_create() call, except for the send and receive buffer size. Instead, specify a time-out value (between tries).

3. Call the remote procedure.

   The low-level version of the callrpc() call is the clnt_call(), which has seven parameters:

   - CLIENT pointer
   - Remote procedure number (STRINGPROC)
   - XDR call for serializing the argument
   - Pointer to the argument
   - XDR call for deserializing the return value from the RPC server
   - Pointer to where the return value is to be placed
   - Total time in seconds to wait for a reply

   For UDP transport, the number of tries is the clnt_call() time-out divided by the clntudp_create() time-out.

   The return code RPC_SUCCESS indicates a successful call; otherwise, an error has occurred. You find the RPC error code in the <RPC\CLNT.H> header file.

   The clnt_destroy() call always deallocates the space associated with the client handle. If the RPC library opened the socket associated with the client handle, the clnt_destroy() call closes it. If you open the socket, it stays open.

6.7 rpcgen Command

Use the **rpcgen** command to generate C code to implement an RPC protocol. The input to RPCGEN is a language similar to C, known as RPC language.

You normally use **rpcgen** *infile* to generate the following four 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 PROTOC.C

6.7.1 Syntax

```
►►──rpcgen──infile──►◄

►►──rpcgen──┬─ -c─┬──┬──────────────┬──┬──────────┬──►◄
            ├─ -h─┤  └─ -o outfile─┘  └─infile─┘
            ├─ -l─┤
            └─ -m─┘

►►──rpcgen── -s transport──┬──────────────┬──┬──────────┬──►◄
                           └─ -o outfile─┘  └─infile─┘
```

-c	Compiles into XDR routines.
-h	Compiles into C data definitions (a header file).
-l	Compiles into client-side stubs.
-m	Compiles into server-side stubs without generating a main routine.
-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.
-s *transport*	Compiles into server-side stubs, using the given transport.

For more information on the **rpcgen** command, see the Sun Microsystems publication, *Networking on the Sun Workstation: Remote Procedure Call Programming Guide*.

6.8 rpcinfo Command

The **rpcinfo** command makes an RPC call to the RPC server and reports the status of the server, which is registered and operational with Portmapper**.

6.8.1 Syntax

6.9 adxhsirl(rpcinfo)

The **rpcinfo** command makes an RPC call to the RPC server and reports the status of the server, which is registered and operational with the Portmapper.

6.9.1 Syntax

rpcinfo for a Host

```
                  ┌─local_host─┐
►►──rpcinfo── -p ─┼────────────┼──►◄
                  └─host───────┘
```

rpcinfo for a Host Using UDP

```
►►──rpcinfo──┬──────────────┬── -u host prognum──┬─────────┬──►◄
             └─ -n portnum──┘                    └─versnum─┘
```

rpcinfo for a Host Using TCP

```
►►──rpcinfo──┬──────────────┬── -t host prognum──┬─────────┬──►◄
             └─ -n portnum──┘                    └─versnum─┘
```

rpcinfo for a Broadcast to Hosts Using UDP

```
►►──rpcinfo──┬──────┬──prognum──versnum──►◄
             └─ -b──┘
```

-p *host* Queries the Portmapper about the specified host and prints a list of all registered RPC programs. If the host is not specified, the system defaults to the local host name.

-n *portnum*
: Specifies the port number to be used for the -t and -u parameters. This value replaces the port number that is given by the Portmapper.

-u *host prognum versnum*
: Sends an RPC call to procedure 0 of *prognum* and *versnum* on the specified host using UDP and reports whether a response is received.

-t *host prognum versnum*
: Sends an RPC call to procedure 0 of *prognum* and *versnum* on the specified host using TCP and reports whether a response is received.

-b *prognum versnum*
: Sends an RPC broadcast to procedure 0 of the specified *prognum* and *versnum* using UDP and reports all hosts that respond.

The *prognum* argument can be either a name or a number. If you specify a *versnum*, the **rpcinfo** command tries to call that version of the specified program. Otherwise, it tries to find all the registered version numbers for the program you specify by calling version 0; then it tries to call each registered version.

The following file is associated with the **rpcinfo** command:

ADX_UPGM\RPC.DAT	Contains a list of server names and their corresponding RPC program numbers and aliases

6.9.1.1.1 Example 1: Use the **rpcinfo** command as follows to display RPC services registered on the local host:

```
rpcinfo -p
```

6.9.1.1.2 Example 2: Use the **rpcinfo** command as follows to display RPC services registered on a remote host named `charm`:

```
rpcinfo -p charm
```

6.9.1.1.3 Example 3: Use the **rpcinfo** command as follows to display the status of a particular RPC program on the remote host named `charm`:

```
rpcinfo -u charm 100003 2
```

or

```
rpcinfo -u charm nfs 2
```

In the previous examples, the **rpcinfo** command shows one of the following:

```
Program 100003 Version 2 ready and waiting
```

or

```
Program 100003 Version 2 is not available
```

6.9.1.1.4 Example 4: Use the **rpcinfo** command as follows to display all hosts on the local network that are running a certain version of a specific RPC server:

```
rpcinfo -b 100003 2
```

or

```
rpcinfo -b nfsprog 2
```

In these examples, the **rpcinfo** command lists all hosts that are running Version 2 of the NFS daemon.

Note: The version number is required for the -b parameter.

6.10 enum clnt_stat Structure

The enum *clnt_stat* structure is defined in the <RPC\CLNT.H> file. RPCs frequently return enum *clnt_stat* information. The format of the enum *clnt_stat* structure follows:

```
enum clnt_stat  {
   RPC_SUCCESS=0,            /* call succeeded */
   /*
    * local errors
    */
   RPC_CANTENCODEARGS=1,     /* can't encode arguments */
   RPC_CANTDECODERES=2,      /* can't decode results */
   RPC_CANTSEND=3,           /* failure in sending call */
   RPC_CANTRECV=4,          /* failure in receiving result */
   RPC_TIMEDOUT=5,          /* call timed out */
   /*
    * remote errors
    */
   RPC_VERSMISMATCH=6,      /* RPC versions not compatible */
   RPC_AUTHERROR=7,         /* authentication error */
   RPC_PROGUNAVAIL=8,       /* program not available */
   RPC_PROGVERSMISMATCH=9,  /* program version mismatched */
   RPC_PROCUNAVAIL=10,      /* procedure unavailable */
   RPC_CANTDECODEARGS=11,   /* decode arguments error */
   RPC_SYSTEMERROR=12,      /* generic "other problem" */
   /*
    * callrpc errors
    */
   RPC_UNKNOWNHOST=13,      /* unknown host name */
   /*
    * create errors
    */
   RPC_PMAPFAILURE=14,       /* the pmapper failed in its call */
   RPC_PROGNOTREGISTERED=15, /* remote program is not registered */
   /*
    * unspecified error
    */
   RPC_FAILED=16
               };
```

6.11 Remote Procedure Call Library

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

RPC Header File	What It Contains
RPC\AUTH.H	Authentication interface
RPC\AUTH_UNI.H	Protocol for UNIX-style authentication parameters for RPC
RPC\CLNT.H	Client-side remote procedure call interface
RPC\PMAP_CLN.H	Supplies C routines to get to PORTMAP services
RPC\PMAP_PRO.H	Protocol for the local binder service, or pmap
RPC\RPC.H	Includes the RPC header files necessary to do remote procedure calling
RPC\RPC_MSG.H	Message definitions
RPC\RPCNETDB.H	Data definitions for network utility calls
RPC\RPCTYPES.H	RPC additions to <TYPES.H>
RPC\SVC.H	Server-side remote procedure call interface
RPC\SVC_AUTH.H	Service side of RPC authentication
RPC\XDR.H	eXternal Data Representation serialization routines

The RPC routines are in the ADXHSIRL.L86 file.

Put the following statement at the beginning of any file using RPC code:

```
#include <rpc\rpc.h>
```

You must define the OS2 variable by doing one of the following:

- Place `#define OS2` at the top of each file that includes TCP/IP header files.
- Use the `-def OS2` option when compiling the source for your application.

Note this is really 'OS2', not '4690'.

6.12 Porting an RPC API Application

The IBM 4690OS RPC implementation differs from the Sun Microsystems RPC implementation as follows:

- The global variables *svc_socks[]* and *noregistered* are used in place of the *svc_fds* global variable. See 6.13.33, "svc_socks []" on page 272 for the use of these variables.
- Functions that rely on file descriptor structures are not supported.
- The svc_getreq() call supports the *socks* and *noavail* global variables. In the Sun Microsystems implementation, the svc_getreq() call supports the *rdfds* global variable.
- TYPES.H for RPC has been renamed to RPCTYPES.H.

6.13 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 for 4690OS.

6.13.1 auth_destroy()

The auth_destroy() call destroys authentication information.

6.13.1.1 Syntax

```
#include <rpc\rpc.h>

void
auth_destroy(auth)
AUTH *auth;
```

6.13.1.2 Parameter

auth
    Pointer to authentication information

6.13.1.3 Description

The auth_destroy() call deletes the authentication information for *auth*. After you call this procedure, *auth* is undefined.

6.13.1.4 Related Calls

authnone_create()
authunix_create()
authunix_create_default()

6.13.2 authnone_create()

The authnone_create() call creates and returns a NULL RPC authentication handle.

6.13.2.1 Syntax

```
#include <rpc\rpc.h>

AUTH *
authnone_create()
```

6.13.2.2 Description

The authnone_create() call creates and returns an RPC authentication handle. The handle passes the NULL authentication on each call.

6.13.2.3 Related Calls

auth_destroy()
authunix_create()
authunix_create_default()

6.13.3 authunix_create()

The authunix_create() call creates and returns a UNIX-based authentication handle.

6.13.3.1 Syntax

```
#include <rpc\rpc.h>

AUTH *
authunix_create(host, uid, gid, len, aup_gids)
char *host;
int uid;
int gid;
int len;
int *aup_gids;
```

6.13.3.2 Parameters

host
Pointer to the symbolic name of the host where the desired server is located

uid
User's user ID

gid
User's group ID

len
Length of the information pointed to by *aup_gids*

aup_gids
Pointer to an array of groups to which the user belongs

6.13.3.3 Description

The authunix_create() call creates and returns an authentication handle that contains UNIX-based authentication information.

6.13.3.4 Related Calls

auth_destroy()
authnone_create()
authunix_create_default()

6.13.4 authunix_create_default()

The authunix_create_default() call calls authunix_create() with default parameters.

6.13.4.1 Syntax

```
#include <rpc\rpc.h>

AUTH *
authunix_create_default()
```

6.13.4.2 Description

The authunix_create_default() call calls authunix_create() with default parameters.

6.13.4.3 Related Calls

auth_destroy()
authnone_create()
authunix_create()

6.13.5 callrpc()

The callrpc() call calls remote procedures.

6.13.5.1 Syntax

```
#include <rpc\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;
```

6.13.5.2 Parameters

host
: Pointer to the symbolic name of the host where the desired server is located

prognum
: Program number of the remote procedure

versnum
: Version number of the remote procedure

procnum
: Procedure number of the remote procedure

inproc
: XDR procedure used to encode the arguments of the remote procedure

in
: Pointer to the arguments of the remote procedure

outproc
: XDR procedure used to decode the results of the remote procedure

out
: Pointer to the results of the remote procedure

6.13.5.3 Return Values

RPC_SUCCESS indicates success; otherwise, an error has occurred. The results of the remote procedure call return to *out*.

6.13.5.4 Description

The callrpc() call calls the remote procedure described by *prognum*, *versnum*, and *procnum* running on the *host* system. It encodes and decodes the parameters for transfer.

Notes:

1. You can use clnt_perrno() to translate the return code into messages.
2. callrpc() cannot call the procedure xdr_enum. See 6.13.53, “xdr_enum()” on page 294 for more information.
3. This procedure uses UDP as its transport layer. See 6.13.16, “clntudp_create()” on page 247 for more information.

6.13.5.5 Example

```
#define RMTPROGNUM (u_long)0x3fffffffL
#define RMTPROGVER (u_long)0x1
#define RMTPROCNUM (u_long)0x1

int inproc=100, outproc, rstat;
...
/* service request to host RPCSERVER_HOST */
if (rstat = callrpc("RPCSERVER_HOST", RMTPROGNUM, RMTPROGVER, RMTPROCNUM,
                    xdr_int, (char *)&inproc, xdr_int,
                    (char *)&outproc)!= 0)
   {
     clnt_perrno(rstat);
     exit(1);
   }
...
```

6.13.5.6 Related Calls

clnt_call()
clnt_perrno()
clntudp_create()

6.13.6 clnt_broadcast()

The clnt_broadcast() call broadcasts a remote program to all locally connected broadcast networks.

6.13.6.1 Syntax

```
#include <rpc\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;
caddr_t in;
xdrproc_t outproc;
caddr_t out;
resultproc_t eachresult;
```

6.13.6.2 Parameters

prognum
: Program number of the remote procedure

versnum
: Version number of the remote procedure

procnum
: Procedure number of the remote procedure

inproc
: XDR procedure used to encode the arguments of the remote procedure

in
: Pointer to the arguments of the remote procedure

outproc
: XDR procedure used to decode the results of the remote procedure

out
: Pointer to the results of the remote procedure

eachresult
: Procedure called after each response

Note: resultproc_t is a type definition:

```
typedef bool_t (*resultproc_t) ();
```

6.13.6.3 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.

6.13.6.4 Description

The clnt_broadcast() call broadcasts a remote program 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 <netinet\in.h>
#include <rpc\rpctypes.h>

bool_t eachresult(out, addr)
char *out;
struct sockaddr_in *addr;
```

6.13.6.5 Parameters

out
: Has the same function as it does for clnt_broadcast(), except that the output of the remote procedure is decoded

addr
: Pointer to the address of the machine that sent the results

6.13.6.6 Example

```
enum clnt_stat cs;
u_long prognum, versnum;
...
cs = clnt_broadcast(prognum, versnum, NULLPROC, xdr_void,
                    (char *)NULL, xdr_void, (char *)NULL, eachresult);
if ((cs != RPC_SUCCESS) && (cs != RPC_TIMEDOUT))
  {
   fprintf( " broadcast failed: \n");
   exit(-1);
  }
...
bool_t
eachresult(out, addr)
void *out;                                /* Nothing comes back */
struct sockaddr_in *addr;                    /* Reply from whom */
{
    register struct hostent *hp;
    ...
    hp = gethostbyaddr((char *) &addr->sin_addr, sizeof addr->sin_addr,
        AF_INET);
    printf("%s %s\n", inet_ntoa(addr->sin_addr), hp->h_name);
    ...
    return(FALSE);
}
```

6.13.6.7 Related Calls

callrpc()
clnt_call()

6.13.7 clnt_call()

The clnt_call() call calls the remote procedure associated with the client handle.

6.13.7.1 Syntax

```
#include <rpc\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;
```

6.13.7.2 Parameters

clnt
: Pointer to a client handle that was previously obtained using clntraw_create(), clnttcp_create(), or clntudp_create()

procnum
: Remote procedure number

inproc
: XDR procedure used to encode *procnum*'s arguments

in
: Pointer to the remote procedure's arguments

outproc
: XDR procedure used to decode the remote procedure's results

out
: Pointer to the remote procedure's results

tout
: Time allowed for the server to respond, in units of 0.1 seconds

6.13.7.3 Return Values

RPC_SUCCESS indicates success; otherwise, an error has occurred. The results of the remote procedure call are returned to *out*.

6.13.7.4 Description

The clnt_call() call calls the remote procedure (*procnum*) associated with the client handle (*clnt*).

6.13.7.5 Example

```
u_long procnum;
register CLIENT *clnt;
enum clnt_stat cs;
struct timeval  total_timeout;
int intsend, intrecv;

cs=clnt_call(clnt, procnum, xdr_int, &intsend,
   xdr_int, &intrecv, total_timeout);
if ( cs != RPC_SUCCESS)
       printf("*Error* clnt_call fail :\n");
```

6.13.7.6 Related Calls

callrpc()
clnt_perror()
clntraw_create()
clnttcp_create()
clntudp_create()

6.13.8 clnt_destroy()

The clnt_destroy() call destroys a client's RPC handle.

6.13.8.1 Syntax

```
#include <rpc\rpc.h>

void
clnt_destroy(clnt)
CLIENT *clnt;
```

6.13.8.2 Parameter

clnt

Pointer to a client handle that was previously created using clntudp_create(), clnttcp_create(), or clntraw_create()

6.13.8.3 Description

The clnt_destroy() call deletes a client RPC transport handle. This procedure involves the deallocation of private data resources, including *clnt*. After you use this procedure, *clnt* is undefined. Open sockets associated with *clnt* must be closed.

6.13.8.4 Related Calls

clntraw_create()
clnttcp_create()
clntudp_create()

6.13.9 clnt_freeres()

The clnt_freeres() call deallocates resources assigned for decoding the results of an RPC.

6.13.9.1 Syntax

```
#include <rpc\rpc.h>

bool_t
clnt_freeres(clnt, outproc, out)
CLIENT *clnt;
xdrproc_t outproc;
char *out;
```

6.13.9.2 Parameters

clnt
: Pointer to a client handle that was previously obtained using clntraw_create(), clnttcp_create(), or clntudp_create()

outproc
: XDR procedure used to decode the remote procedure's results

out
: Pointer to the results of the remote procedure

6.13.9.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.9.4 Description

The clnt_freeres() call de-allocates any resources that were assigned by the system to decode the results of an RPC.

6.13.9.5 Related Calls

clntraw_create()
clnttcp_create()
clntudp_create()

6.13.10 clnt_geterr()

The clnt_geterr() call copies the error structure from a client's handle to the local structure.

6.13.10.1 Syntax

```
#include <rpc\rpc.h>

void
clnt_geterr(clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;
```

6.13.10.2 Parameters

clnt

Pointer to a client handle that was previously obtained using clntraw_create(), clnttcp_create(), or clntudp_create()

errp

Pointer to the address into which the error structure is copied

6.13.10.3 Description

The clnt_geterr() call copies the error structure from the client handle to the structure at address *errp*.

6.13.10.4 Example

```
u_long procnum;
register CLIENT *clnt;
enum clnt_stat cs;
struct timeval  total_timeout;
int intsend = 100, intrecv;
struct rpc_err error;
...
total_timeout.tv_sec = 20;
total_timeout.tv_usec = 0;
...
cs=clnt_call(clnt, procnum, xdr_int, &intsend,
   xdr_int, &intrecv, total_timeout);
if ( cs != RPC_SUCCESS)
     {
          clnt_geterr(clnt, &error);
          clnt_perror(clnt, "recv from server");
      }
...
```

6.13.10.5 Related Calls

clnt_call()
clnt_pcreateerror()
clnt_perrno()
clnt_perror()
clntraw_create()

clnttcp_create()
clntudp_create()

6.13.11 clnt_pcreateerror()

The clnt_pcreateerror() call indicates why a client handle cannot be created.

6.13.11.1 Syntax

```
#include <rpc\rpc.h>

void
clnt_pcreateerror(s)
char *s;
```

6.13.11.2 Parameter

s

Pointer to a string that is to be printed in front of the message. The string is followed by a colon.

6.13.11.3 Description

The clnt_pcreateerror() call writes a message to the standard error device, indicating why a client handle cannot be created. Use this procedure after the clntraw_create(), clnttcp_create(), or clntudp_create() call fails.

For an example of the clnt_pcreateerror() call, see 6.13.15, “clnttcp_create()” on page 245.

6.13.11.4 Related Calls

clnt_geterr()
clnt_perrno()
clnt_perror()
clntraw_create()
clnttcp_create()
clntudp_create()

6.13.12 clnt_perrno()

The clnt_perrno() call writes a message to the standard error device corresponding to the condition indicated by *stat*.

6.13.12.1 Syntax

```
#include <rpc\rpc.h>

void
clnt_perrno(stat)
enum clnt_stat stat;
```

6.13.12.2 Parameter

stat
    The client status

6.13.12.3 Description

The clnt_perrno() call writes a message to the standard error device corresponding to the condition indicated by *stat*. Use this procedure after callrpc() and clnt_broadcast() if there is an error.

6.13.12.4 Related Calls

callrpc()
clnt_geterr()
clnt_pcreateerror()
clnt_perror()

6.13.13 clnt_perror()

The clnt_perror() call writes an error message indicating why RPC failed.

6.13.13.1 Syntax

```
#include <rpc\rpc.h>

void
clnt_perror(clnt, s)
CLIENT *clnt;
char *s;
```

6.13.13.2 Parameters

clnt

Pointer to a client handle that was previously obtained using clntudp_create(), clnttcp_create(), or clntraw_create().

s

Pointer to a string that is to be printed in front of the message. The string is followed by a colon.

6.13.13.3 Description

The clnt_perror() call writes a message to the standard error device, indicating why an RPC failed. Use this procedure after clnt_call() if there is an error.

For an example of the clnt_perror() call, see 6.13.10, “clnt_geterr()” on page 239.

6.13.13.4 Related Calls

clnt_call()
clnt_geterr()
clnt_pcreateerror()
clnt_perrno()
clntraw_create()
clnttcp_create()
clntudp_create()

6.13.14 clntraw_create()

The clntraw_create() call creates a client transport handle to use in a single task.

6.13.14.1 Syntax

```
#include <rpc\rpc.h>

CLIENT *
clntraw_create(prognum, versnum)
u_long prognum;
u_long versnum;
```

6.13.14.2 Parameters

prognum
Remote program number

versnum
Version number of the remote program

6.13.14.3 Return Value

NULL indicates failure.

6.13.14.4 Description

The clntraw_create() call creates a dummy client for the remote double (*prognum*, *versnum*). Because messages are passed using a buffer within the address space of the local process, the server should also use the same address space, which simulates RPC programs within one address space. See 6.13.42, “svcraw_create()” on page 281 for more information.

6.13.14.5 Related Calls

clnt_call()
clnt_destroy()
clnt_pcreateerror()
clnttcp_create()
clntudp_create()
svcraw_create()

6.13.15 clnttcp_create()

The clnttcp_create() call creates an RPC client transport handle for the remote program using TCP transport.

6.13.15.1 Syntax

```
#include <rpc\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;
```

6.13.15.2 Parameters

addr
: Pointer to the internet address of the remote program. If *addr* points to a port number of 0, *addr* is set to the port on which the remote program is receiving.

prognum
: Remote program number.

versnum
: Version number of the remote program.

sockp
: Pointer to the socket. If *sockp* is RPC_ANYSOCK, then this routine opens a new socket and sets *sockp*.

sendsz
: Size of the send buffer. Specify 0 to have clnttcp_create() pick a suitable default size.

recvsz
: Size of the receive buffer. Specify 0 to have clnttcp_create() pick a suitable default size.

6.13.15.3 Return Value

NULL indicates failure.

6.13.15.4 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.

6.13.15.5 Example

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L

register CLIENT *clnt;
int sock = RPC_ANYSOCK; /* can be also valid socket descriptor */
```

```
struct hostent *hp;
struct sockaddr_in server_addr;

/* get the internet address of RPC server */
if ((hp = gethostbyname("RPCSERVER_HOST") == NULL)
  {
    fprintf(stderr,"Can't get address for %s\n",argv[2]);
    exit (-1);
  }

bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr.s_addr, hp->h_length);
server_addr.sin_family = AF_INET;
server_addr.sin_port = 0;


/* create TCP handle */
if  ((clnt = clnttcp_create(&server_addr, RMTPROGNUM, RMTPROGVER,
                             &sock, 1024*10, 1024*10)) == NULL)
   {
     clnt_pcreateerror("clnttcp_create");
     exit(-1);
    }
```

6.13.15.6 Related Calls

clnt_destroy()
clnt_pcreateerror()
clntraw_create()
clntudp_create()

6.13.16 clntudp_create()

The clntudp_create() call creates an RPC client transport handle for the remote program using UDP transport.

6.13.16.1 Syntax

```
#include <rpc\rpc.h>
#include <netdb.h>

CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
struct timeval wait;
int *sockp;
```

6.13.16.2 Parameters

addr
: Pointer to the internet address of the remote program. If *addr* points to a port number of 0, *addr* is set to the port on which the remote program is receiving. The remote PORTMAP service is used for this.

prognum
: Remote program number.

versnum
: Version number of the remote program.

wait
: Interval at which UDP resends the call request, until either a response is received or the call times out. Set the time-out length using the clnt_call() procedure.

sockp
: Pointer to the socket. If *sockp* is RPC_ANYSOCK, this routine opens a new socket and sets *sockp*.

6.13.16.3 Return Value

NULL indicates failure.

6.13.16.4 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: Do not use this procedure with procedures that use large arguments or return large results. UDP RPC messages can contain only 2K bytes of encoded data.

6.13.16.5 Example

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L

register CLIENT *clnt;
int sock = RPC_ANYSOCK; /* can be also valid socket descriptor */
struct hostent *hp;
struct timeval pertry_timeout;
struct sockaddr_in server_addr;

/* get the internet address of RPC server */
if ((hp = gethostbyname("RPC_HOST") == NULL)
  {
    fprintf(stderr,"Can't get address for %s\n",argv[2]);
    exit (-1);
  }

pertry_timeout.tv_sec = 3;
pertry_timeout.tv_usec = 0;
bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr.s_addr, hp->h_length);
server_addr.sin_family = AF_INET;
server_addr.sin_port = 0;

/* create UDP handle */
if ((clnt = clntudp_create(&server_addr, RMTPROGNUM, RMTPROGVER,
                             pertry_timeout, &sock)) == NULL)
   {
    clnt_pcreateerror("clntudp_create");
    exit(-1);
   }
```

6.13.16.6 Related Calls

clnt_destroy()
clnt_pcreateerror()
clntraw_create()
clnttcp_create()

6.13.17 get_myaddress()

The get_myaddress() call returns the local host's internet address.

6.13.17.1 Syntax

```
#include <rpc\rpc.h>

void
get_myaddress(addr)
struct sockaddr_in *addr;
```

6.13.17.2 Parameter

addr

Pointer to the location where the local internet address is placed

6.13.17.3 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.

6.13.18 pmap_getmaps()

The pmap_getmaps() call returns a list of current program-to-port mappings on a specified remote host's Portmapper.

6.13.18.1 Syntax

```
#include <rpc\rpc.h>

struct pmaplist *
pmap_getmaps(addr)
struct sockaddr_in *addr;
```

6.13.18.2 Parameter

addr
: Pointer to the internet address of the remote host

6.13.18.3 Description

The pmap_getmaps() call returns a list of current program-to-port mappings on the remote host's Portmapper specified by *addr*.

6.13.18.4 Example

```
struct hostent *hp;
struct sockaddr_in pmapper_addr;
struct pmaplist *my_pmaplist = NULL;

if ((hp = gethostbyname("PMAP_HOST") == NULL)
  {
    fprintf(stderr,"Can't get address for %s\n","PMAP_HOST");
    exit (-1);
  }

bcopy(hp->h_addr, (caddr_t)&pmapper_addr.sin_addr.s_addr, hp->h_length);
pmapper_addr.sin_family = AF_INET;
pmapper_addr.sin_port = 0;

/*
 *  get the list of program, version, protocol and port number
 *  from remote portmapper
 *
 *     struct pmap {
 *              long unsigned pm_prog;
 *              long unsigned pm_vers;
 *              long unsigned pm_prot;
 *              long unsigned pm_port;
 *             };
 *     struct pmaplist {
 *             struct pmap     pml_map;
 *             struct pmaplist *pml_next;
 *            };
 */
 my_pmaplist = pmap_getmaps(&pmapper_addr);
 ...
```

6.13.18.5 Related Calls

pmap_getport()
pmap_rmtcall()
pmap_set()
pmap_unset()

6.13.19 pmap_getport()

The pmap_getport() call returns the port number associated with a remote program.

6.13.19.1 Syntax

```
#include <rpc\rpc.h>

u_short
pmap_getport(addr, prognum, versnum, protocol)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
u_long protocol;
```

6.13.19.2 Parameters

addr
: Pointer to the internet address of the remote host

prognum
: Program number to be mapped

versnum
: Version number of the program to be mapped

protocol
: Transport protocol used by the program

6.13.19.3 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.

6.13.19.4 Description

The pmap_getport() call returns the port number associated with the remote program (*prognum*), the version (*versnum*), and the transport protocol (*protocol*).

6.13.19.5 Related Calls

pmap_getmaps()
pmap_rmtcall()
pmap_set()
pmap_unset()

6.13.20 pmap_rmtcall()

The pmap_rmtcall() call instructs Portmapper to make an RPC call to a procedure on a host on your behalf.

6.13.20.1 Syntax

```
#include <rpc\rpc.h>
#include <netdb.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;
```

6.13.20.2 Parameters

addr
: Pointer to the internet address of the foreign host

prognum
: Remote program number

versnum
: Version number of the remote program

procnum
: Procedure to be called

inproc
: XDR procedure that encodes the arguments of the remote procedure

in
: Pointer to the arguments of the remote procedure

outproc
: XDR procedure that decodes the results of the remote procedure

out
: Pointer to the results of the remote procedure

tout
: Time-out period for the remote request

portp
: Port number of the triple (*prognum*, *versnum*, *procnum*), if the call from the remote PORTMAP service succeeds

6.13.20.3 Return Values

RPC_SUCCESS indicates success; otherwise, an error has occurred. The results of the remote procedure call return to *out*.

6.13.20.4 Description

The pmap_rmtcall() call instructs Portmapper to make an RPC call to a procedure on a host, on your behalf. Use this procedure only for ping-type functions.

6.13.20.5 Example

```
int inproc, outproc,rc;
u_long portp;
struct timeval total_timeout;
struct sockaddr_in *addr;
...
get_myaddress(addr);
...
total_timeout.tv_sec = 20;
total_timeout.tv_usec = 0;

rc = pmap_rmtcall(addr,RMTPROGNUM,RMTPROGVER,RMTPROCNUM,xdr_int,
                  &inproc,xdr_int,&outproc,total_timeout,&portp);
if (rc != 0)
 {
   fprintf(stderr,"error: pmap_rmtcall() failed: %d \n",rc);
   clnt_perrno(rc);
   exit(1);
 }
```

6.13.20.6 Related Calls

pmap_getmaps()
pmap_getport()
pmap_set()
pmap_unset()

6.13.21 pmap_set()

The pmap_set() call sets the mapping of a server program to a port on the local machine's Portmapper.

6.13.21.1 Syntax

```
#include <rpc\rpc.h>

bool_t
pmap_set(prognum, versnum, protocol, port)
u_long prognum;
u_long versnum;
u_long protocol;
u_short port;
```

6.13.21.2 Parameters

prognum
: Local program number

versnum
: Version number of the local program

protocol
: Transport protocol used by the local program

port
: Port to which the local program is mapped

6.13.21.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.21.4 Description

The pmap_set() call sets the mapping of the program (specified by *prognum*, *versnum*, and *protocol*) to *port* on the local machine's Portmapper. This procedure is automatically called by the svc_register() procedure.

6.13.21.5 Related Calls

pmap_getmaps()
pmap_getport()
pmap_rmtcall()
pmap_unset()

6.13.22 pmap_unset()

The pmap_unset() call removes the mappings on the local machine's Portmapper.

6.13.22.1 Syntax

```
#include <rpc\rpc.h>

bool_t
pmap_unset(prognum, versnum)
u_long prognum;
u_long versnum;
```

6.13.22.2 Parameters

prognum
: Local program number

versnum
: Version number of the local program

6.13.22.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.22.4 Description

The pmap_unset() call removes the mappings associated with *prognum* and *versnum* on the local machine's Portmapper. All ports for each transport protocol currently mapping the *prognum* and *versnum* are removed from the PORTMAP service.

6.13.22.5 Example

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L
...
/* remove the mapping of remote program */
/* and its port from local portmapper   */
pmap_unset(RMTPROGNUM, RMTPROGVER);
...
```

6.13.22.6 Related Calls

pmap_getmaps()
pmap_getport()
pmap_rmtcall()
pmap_set()

6.13.23 registerrpc()

The registerrpc() call registers a procedure with the local Portmapper.

6.13.23.1 Syntax

```
#include <rpc\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;
```

6.13.23.2 Parameters

prognum
: Program number to register.

versnum
: Version number to register.

procnum
: Procedure number to register.

procname
: Procedure that is called when the registered program is requested. *procname* must accept a pointer to its arguments and return a static pointer to its results.

inproc
: XDR procedure that decodes the arguments.

outproc
: XDR procedure that encodes the results.

Note: You cannot use xdr_enum() as an argument to registerrpc(). See 6.13.53, “xdr_enum()” on page 294 for more information.

6.13.23.3 Return Values

The value 0 indicates success; the value −1 indicates an error.

6.13.23.4 Description

The registerrpc() call registers a procedure with the local Portmapper and creates a control structure to remember the server procedure and its XDR routine. The svc_run() call uses the control structure. Procedures registered using registerrpc() are accessed using the UDP transport layer.

6.13.23.5 Example

```
#define RMTPROGNUM (u_long)0x3fffffffL
#define RMTPROGVER (u_long)0x1
#define RMTPROCNUM (u_long)0x1

main()
 {
  int *rmtprog();

  /* register remote program with portmapper */
  registerrpc(RMTPROGNUM, RMTPROGVER, RMTPROCNUM, rmtprog,
                   xdr_int, xdr_int);

  /* infinite loop, waits for RPC request from client */
  svc_run();
  printf("Error: svc_run should never reach this point \n");
  exit(1);
 }

int *
rmtprog(inproc)          /* remote program */
int *inproc;

{
 int *outproc;
 ...
 /* Process request */
 ...
 return (outproc);
}
```

6.13.23.6 Related Calls

svc_register()
svc_run()

6.13.24 rpc_createerr

rpc_createerr is a global variable set when any RPC client creation routine fails.

6.13.24.1 Syntax

```
#include <rpc\rpc.h>

struct  rpc_createerr rpc_createerr;
```

6.13.24.2 Description

rpc_createerr is a global variable that is set when any RPC client creation routine fails. Use clnt_pcreateerror() to print the message.

6.13.25 svc_destroy()

The svc_destroy() call destroys the RPC service transport handle.

6.13.25.1 Syntax

```
#include <rpc\rpc.h>

void
svc_destroy(xprt)
SVCXPRT *xprt;
```

6.13.25.2 Parameter

xprt
: Pointer to the service transport handle

6.13.25.3 Description

The svc_destroy() call deletes the RPC service transport handle *xprt*, which becomes undefined after this routine is called.

6.13.25.4 Related Calls

svcraw_create()
svctcp_create()
svcudp_create()

6.13.26 svc_freeargs()

The svc_freeargs() call frees storage allocated for argument decoding.

6.13.26.1 Syntax

```
#include <rpc\rpc.h>

bool_t
svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

6.13.26.2 Parameters

xprt
: Pointer to the service transport handle

inproc
: XDR routine that decodes the arguments

in
: Pointer to the input arguments

6.13.26.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.26.4 Description

The svc_freeargs() call frees storage allocated to decode the arguments received by svc_getargs().

6.13.26.5 Related Calls

svc_getargs()

6.13.27 svc_getargs()

The svc_getargs() call decodes arguments from an RPC request.

6.13.27.1 Syntax

```
#include <rpc\rpc.h>

bool_t
svc_getargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

6.13.27.2 Parameters

xprt
Pointer to the service transport handle

inproc
XDR routine that decodes the arguments

in
Pointer to the decoded arguments

6.13.27.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.27.4 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*.

6.13.27.5 Example

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L

...

SVCXPRT *transp;

transp = svcudp_create(RPC_ANYSOCK);
if (transp == NULL)
   {
      fprintf(stderr, "can't create an RPC server transport\n");
      exit(-1);
   }
pmap_unset(RMTPROGNUM, RMTPROGVER);
if (!svc_register(transp, RMTPROGNUM, RMTPROGVER, rmtprog, IPPROTO_UDP))
   {
      fprintf(stderr, "can't register rmtprog() service\n");
      exit(-1);
   }
printf("rmtprog() service registered.\n");

svc_run();
printf("Error:svc_run should never reach this point \n");
exit(1);
...
rmtprog(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
    int intrecv;

     switch((int)rqstp->rq_proc)
        {
             case PROCNUM1:
                      svc_getargs(transp, xdr_int, &intrecv);
                      ...
                      return;
             case PROCNUM2:
             ...
          }
...
}
```

6.13.27.6 Related Call

svc_freeargs()

6.13.28 svc_getcaller()

The svc_getcaller() call gets the network address of the client associated with the service transport handle.

6.13.28.1 Syntax

```
#include <rpc\rpc.h>

struct sockaddr_in *
svc_getcaller(xprt)
SVCXPRT *xprt;
```

6.13.28.2 Parameter

xprt
  Pointer to the service transport handle

6.13.28.3 Description

This call gets the network address of the client associated with the service transport handle.

6.13.29 svc_getreq()

The svc_getreq() call implements asynchronous event processing and returns control to the program after all sockets have been serviced.

6.13.29.1 Syntax

```
#include <rpc\rpc.h>

void
svc_getreq(socks, noavail)
int socks[];
int noavail;
```

6.13.29.2 Parameters

socks
Array of socket descriptors

noavail
Integer specifying the number of socket descriptors in the array

6.13.29.3 Description

Use the svc_getreq() call rather than svc_run() to do asynchronous event processing. The routine returns control to the program when all sockets in the *socks* array have been serviced.

6.13.29.4 Related Calls

svc_run()
svc_socks[]

6.13.30 svc_register()

The svc_register() call registers procedures on the local Portmapper.

6.13.30.1 Syntax

```
#include <rpc\rpc.h>
#include <netdb.h>

bool_t
svc_register(xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
u_long prognum;
u_long versnum;
void (*dispatch) ();
int protocol;
```

6.13.30.2 Parameters

xprt
: Pointer to the service transport handle.

prognum
: Program number to be registered.

versnum
: Version number of the program to be registered.

dispatch
: Dispatch routine associated with *prognum* and *versnum*. The structure of the dispatch routine is as follows:

```
dispatch(request, xprt)
struct svc_req *request;
SVCXPRT *xprt;
```

protocol
: Protocol used. The value is generally one of the following:

- 0 (zero)
- IPPROTO_UDP
- IPPROTO_TCP

When you use a value of 0, the service is not registered with Portmapper.

6.13.30.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.30.4 Description

The svc_register() call associates the specified program with the service dispatch routine *dispatch*.

Note: When you use a toy RPC service transport created with svcraw_create(), make a call to xprt_register() immediately after a call to svc_register().

6.13.30.5 Example

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L

SVCXPRT *transp;

/* register the remote program with local portmapper */
if (!svc_register(transp, RMTPROGNUM, RMTPROGVER, rmtprog, IPPROTO_UDP))
      {
        fprintf(stderr, "can't register rmtprog() service\n");
        exit(-1);
      }

/* code for remote program; rmtprog  */
rmtprog(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
  {
     ...
     ...
   }
```

6.13.30.6 Related Calls

registerrpc()
svc_unregister()
xprt_register()

6.13.31 svc_run()

The svc_run() call accepts RPC requests and calls the appropriate service.

6.13.31.1 Syntax

```
#include <rpc\rpc.h>

void
svc_run()
```

6.13.31.2 Description

The svc_run() call accepts RPC requests and calls the appropriate service using svc_getreq(). The svc_run() call does not return control to the caller.

6.13.31.3 Example

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L

...

SVCXPRT *transp;

transp = svcudp_create(RPC_ANYSOCK);
if (transp == NULL)
   {
      fprintf(stderr, "can't create an RPC server transport\n");
      exit(-1);
    }
pmap_unset(RMTPROGNUM, RMTPROGVER);
if (!svc_register(transp, RMTPROGNUM, RMTPROGVER, rmtprog, IPPROTO_UDP))
   {
      fprintf(stderr, "can't register rmtprog() service\n");
      exit(-1);
    }
printf("rmtprog() service registered.\n");

svc_run();

printf("Error:svc_run should never reach this point \n");
exit(1);
...
```

```
rmtprog(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
 {
 ...
 }
```

6.13.31.4 Related Calls

registerrpc()
svc_getreq()

6.13.32 svc_sendreply()

The svc_sendreply() call sends the results of an RPC to the caller.

6.13.32.1 Syntax

```
#include <rpc\rpc.h>

bool_t
svc_sendreply(xprt, outproc, out)
SVCXPRT *xprt;
xdrproc_t outproc;
char *out;
```

6.13.32.2 Parameters

xprt
: Pointer to the caller's transport handle

outproc
: XDR procedure that encodes the results

out
: Pointer to the results

6.13.32.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.32.4 Description

The service dispatch routine calls the svc_sendreply() call to send the results of the call to the caller.

6.13.32.5 Example

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L

...

SVCXPRT *transp;

transp = svcudp_create(RPC_ANYSOCK);
if (transp == NULL)
   {
      fprintf(stderr, "can't create an RPC server transport\n");
      exit(-1);
    }
pmap_unset(RMTPROGNUM, RMTPROGVER);
if (!svc_register(transp, RMTPROGNUM, RMTPROGVER, rmtprog, IPPROTO_UDP))
    {
       fprintf(stderr, "can't register rmtprog() service\n");
       exit(-1);
     }
printf("rmtprog() service registered.\n");
```

```
svc_run();

printf("Error:svc_run should never reach this point \n");
exit(1);
...


rmtprog(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{

    int intrecv;
    int replysend;
    switch((int)rqstp->rq_proc)
     {
        case PROCNUM0:
           svc_getargs(transp, xdr_int, &intrecv);
           ...
           /* process intrecv parameter */
           replysend = ( intrecv * 1000) + 100;
           /*  send reply to client */
           if (!svc_sendreply(transp, xdr_int, &replysend))
              {
                   fprintf(stderr,"can't reply to RPC call\n");
                   exit(-1);
               }
            return;
        case PROCNUM1:
          ...
          ...
       }
...
}
```

6.13.33 svc_socks []

svc_socks[] is an array of socket descriptors being serviced.

6.13.33.1 Syntax

```
#include <rpc\rpc.h>

int svc_socks[];
```

```
#include <rpc\rpc.h>

int noregistered;
```

6.13.33.2 Description

svc_socks[] is an array of socket descriptors being serviced. *noregistered* is an integer that specifies the number of socket descriptors in svc_socks[].

6.13.33.3 Related Call

svc_getreq()

6.13.34 svc_unregister()

The svc_unregister() call removes the local mapping.

6.13.34.1 Syntax

```
#include <rpc\rpc.h>

void
svc_unregister(prognum, versnum)
u_long prognum;
u_long versnum;
```

6.13.34.2 Parameters

prognum
: Program number of the removed program

versnum
: Version number of the removed program

6.13.34.3 Description

The svc_unregister() call removes all local mappings of (*prognum*, *versnum*) to dispatch routines and (*prognum*, *versnum*, *) to port numbers.

6.13.34.4 Example

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L
...
  /* unregister remote program from local portmapper */
  svc_unregister(RMTPROGNUM, RMTPROGVER);
...
```

6.13.34.5 Related Call

svc_register()

6.13.35 svcerr_auth()

The svcerr_auth() call sends an error reply when the service dispatch routine cannot execute an RPC request because of authentication errors.

6.13.35.1 Syntax

```
#include <rpc\rpc.h>

void
svcerr_auth(xprt, why)
SVCXPRT *xprt;
enum auth_stat why;
```

6.13.35.2 Parameters

xprt
: Pointer to the service transport handle

why
: Reason why the call is refused

6.13.35.3 Description

A service dispatch routine that refuses to run an RPC request because of authentication errors calls svcerr_auth().

6.13.35.4 Related Calls

svcerr_decode()
svcerr_noproc()
svcerr_noprog()
svcerr_progvers()
svcerr_systemerr()
svcerr_weakauth()

6.13.36 svcerr_decode()

The svcerr_decode() call sends an error reply when the service dispatch routine cannot decode its parameters.

6.13.36.1 Syntax

```
#include <rpc\rpc.h>

void
svcerr_decode(xprt)
SVCXPRT *xprt;
```

6.13.36.2 Parameter

xprt
: Pointer to the service transport handle

6.13.36.3 Description

A service dispatch routine that cannot decode its parameters calls svcerr_decode().

6.13.36.4 Related Calls

svcerr_auth()
svcerr_noproc()
svcerr_noprog()
svcerr_progvers()
svcerr_systemerr()
svcerr_weakauth()

6.13.37 svcerr_noproc()

The svcerr_noproc() call sends an error reply when the service dispatch routine cannot call the procedure requested.

6.13.37.1 Syntax

```
#include <rpc\rpc.h>

void
svcerr_noproc(xprt)
SVCXPRT *xprt;
```

6.13.37.2 Parameter

xprt
: Pointer to the service transport handle

6.13.37.3 Description

A service dispatch routine that does not implement the requested procedure calls the svcerr_noproc() call.

6.13.37.4 Related Calls

svcerr_auth()
svcerr_decode()
svcerr_noprog()
svcerr_progvers()
svcerr_systemerr()
svcerr_weakauth()

6.13.38 svcerr_noprog()

The svcerr_noprog() call sends an error code when the requested program is not registered.

6.13.38.1 Syntax

```
#include <rpc\rpc.h>

void
svcerr_noprog(xprt)
SVCXPRT *xprt;
```

6.13.38.2 Parameter

xprt
Pointer to the service transport handle

6.13.38.3 Description

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

6.13.38.4 Related Calls

svcerr_auth()
svcerr_decode()
svcerr_noproc()
svcerr_progvers()
svcerr_systemerr()
svcerr_weakauth()

6.13.39 svcerr_progvers()

The svcerr_progvers() call sends the low version number and high version number of RPC service when the version numbers of two RPC programs do not match.

6.13.39.1 Syntax

```
#include <rpc\rpc.h>

void
svcerr_progvers(xprt, low_vers, high_vers)
SVCXPRT *xprt;
u_long low_vers;
u_long high_vers;
```

6.13.39.2 Parameters

xprt
: Pointer to the service transport handle

low_vers
: Low version number

high_vers
: High version number

6.13.39.3 Description

A service dispatch routine calls the svcerr_progvers() call when the version numbers of two RPC programs do not match. The call sends the supported low version and high version of RPC service.

6.13.39.4 Related Calls

svcerr_decode()
svcerr_noproc()
svcerr_noprog()
svcerr_progvers()
svcerr_systemerr()
svcerr_weakauth()

6.13.40 svcerr_systemerr()

The svcerr_systemerr() call sends an error reply when the service dispatch routine detects a system error that has not been handled.

6.13.40.1 Syntax

```
#include <rpc\rpc.h>

void
svcerr_systemerr(xprt)
SVCXPRT *xprt;
```

6.13.40.2 Parameter

xprt
Pointer to the service transport handle

6.13.40.3 Description

A service dispatch routine calls the svcerr_systemerr() call when it detects a system error that is not handled by the protocol.

6.13.40.4 Related Calls

svcerr_auth()
svcerr_decode()
svcerr_noproc()
svcerr_noprog()
svcerr_progvers()
svcerr_weakauth()

6.13.41 svcerr_weakauth()

The svcerr_weakauth() call sends an error reply when the service dispatch routine cannot run an RPC because of weak authentication parameters.

6.13.41.1 Syntax

```
#include <rpc\rpc.h>

void
svcerr_progvers(xprt)
SVCXPRT *xprt;
```

6.13.41.2 Parameter

xprt
Pointer to the service transport handle

6.13.41.3 Description

A service dispatch routine calls the svcerr_weakauth() call when it cannot run an RPC because of correct but weak authentication parameters

Note: This is the equivalent of `svcerr_auth(`*`xprt`*`, AUTH_TOOWEAK).`

6.13.41.4 Related Calls

svcerr_auth()
svcerr_decode()
svcerr_noproc()
svcerr_noprog()
svcerr_progvers()
svcerr_systemerr()

6.13.42 svcraw_create()

The svcraw_create() call creates a local RPC service transport handle to simulate RPC programs within one host.

6.13.42.1 Syntax

```
#include <rpc\rpc.h>

SVCXPRT *
svcraw_create()
```

6.13.42.2 Return Value

NULL indicates failure.

6.13.42.3 Description

The svcraw_create() call creates a local RPC service transport used for timings, to which it returns a pointer. Because messages are passed using a buffer within the address space of the local process, the client process must also use the same address space. This allows the simulation of RPC programs within one host. See 6.13.14, “clntraw_create()” on page 244 for more information.

6.13.42.4 Related Calls

clntraw_create()
svc_destroy()
svctcp_create()
svcudp_create()

6.13.43 svctcp_create()

The svctcp_create() call creates a TCP-based service transport.

6.13.43.1 Syntax

```
#include <rpc\rpc.h>

SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
int sock;
u_int send_buf_size;
u_int recv_buf_size;
```

6.13.43.2 Parameters

sock
: Socket descriptor. If *sock* is RPC_ANYSOCK, a new socket is created. If the socket is not bound to a local TCP port, it is bound to an arbitrary port.

send_buf_size
: Size of the send buffer. Specify 0 if you want the call to pick a suitable default value.

recv_buf_size
: Size of the receive buffer. Specify 0 if you want the call to pick a suitable default value.

6.13.43.3 Return Value

NULL indicates failure.

6.13.43.4 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.

6.13.43.5 Example

```
...
SVCXPRT *transp;

transp = svctcp_create(RPC_ANYSOCK, 1024*10, 1024*10);
...
```

6.13.43.6 Related Calls

svc_destroy()
svcraw_create()
svcudp_create()

6.13.44 svcudp_create()

The svcudp_create() call creates a UDP-based service transport.

6.13.44.1 Syntax

```
#include <rpc\rpc.h>

SVCXPRT *
svcudp_create(sockp)
int sockp;
```

6.13.44.2 Parameter

sockp
: The socket number associated with the service transport handle. If *sockp* is RPC_ANYSOCK, a new socket is created. If the socket is not bound to a local port, it is bound to an arbitrary port.

6.13.44.3 Return Value

NULL indicates failure.

6.13.44.4 Description

The svcudp_create() call creates a UDP-based service transport to which it returns a pointer. *xprt*—>xp_sock contains the transport's socket descriptor. *xprt*—>xp_port contains the transport's port number.

6.13.44.5 Example

```
...
SVCXPRT *transp;

transp = svcudp_create(RPC_ANYSOCK);
...
```

6.13.44.6 Related Calls

svc_destroy()
svcraw_create()
svctcp_create()

6.13.45 xdr_accepted_reply()

The xdr_accepted_reply() call translates between an RPC reply message and its external representation.

6.13.45.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_accepted_reply(xdrs, ar)
XDR *xdrs;
struct accepted_reply *ar;
```

6.13.45.2 Parameters

xdrs
: Pointer to an XDR stream

ar
: Pointer to the reply to be represented

6.13.45.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.45.4 Description

The xdr_accepted_reply() call translates between an RPC reply message and its external representation.

6.13.46 xdr_array()

The xdr_array() call translates between an array and its external representation.

6.13.46.1 Syntax

```
#include <rpc\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;
```

6.13.46.2 Parameters

xdrs
: Pointer to an XDR stream

arrp
: Address of the pointer to the array

sizep
: Pointer to the element count of the array

maxsize
: Maximum number of elements accepted

elsize
: Size of each of the array’s elements, found using sizeof()

elproc
: XDR routine that translates an individual array element

6.13.46.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.46.4 Description

The xdr_array() call translates between an array and its external representation.

6.13.46.5 Example

```
struct myarray
   {
      int  *arrdata;
      u_int   arrlength;
    };

void
xdr_myarray(xdrsp,arrp)
XDR  *xdrsp;
```

```
struct myarray *arrp;
{
   xdr_array(xdrsp,(caddr_t *)&arrp->arrdata,&arrp->arrlength,
                              MAXLEN,sizeof(int),xdr_int);
}

...
static int arrc_in[10],arrc_out[10];
...
u_long procnum;
register CLIENT *clnt;
enum clnt_stat cs;
struct timeval  total_timeout;
...
total_timeout.tv_sec = 20;
total_timeout.tv_usec = 0;
...
myarrc_in.arrdata =  & arrc_in[0];
myarrc_in.arrlength = ( sizeof(arrc_in) / sizeof (int) );
myarrc_out.arrdata = & arrc_out[0];
myarrc_out.arrlength = ( sizeof(arrc_out) / sizeof (int) );

cs=clnt_call(clnt, procnum, xdr_myarray, (char *) &myarrc_in, xdr_myarray,
                                 (char *)&myarrc_out, total_timeout);
if ( cs != RPC_SUCCESS)
        printf("*Error* clnt_call fail :\n");
...
```

6.13.47 xdr_authunix_parms()

The xdr_authunix_parms() call translates between UNIX-based authentication information and its external representation.

6.13.47.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_authunix_parms(xdrs, aupp)
XDR *xdrs;
struct authunix_parms *aupp;
```

6.13.47.2 Parameters

xdrs

Pointer to an XDR stream

aupp

Pointer to the authentication information

6.13.47.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.47.4 Description

The xdr_authunix_parms() call translates between UNIX-based authentication information and its external representation.

6.13.48 xdr_bool()

The xdr_bool() call translates between a Boolean and its external representation.

6.13.48.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_bool(xdrs, bp)
XDR *xdrs;
bool_t *bp;
```

6.13.48.2 Parameters

xdrs
: Pointer to an XDR stream

bp
: Pointer to the Boolean

6.13.48.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.48.4 Description

The xdr_bool() call translates between a Boolean and its external representation.

6.13.49 xdr_bytes()

The xdr_bytes() call translates between byte strings and their external representations.

6.13.49.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_bytes(xdrs, sp, sizep, maxsize)
XDR *xdrs;
char **sp;
u_int *sizep;
u_int maxsize;
```

6.13.49.2 Parameters

xdrs
: Pointer to an XDR stream

sp
: Pointer to a pointer to the byte string

sizep
: Pointer to the byte string size

maxsize
: Maximum size of the byte string

6.13.49.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.49.4 Description

The xdr_bytes() call translates between byte strings and their external representations.

6.13.49.5 Example

```
struct mybytes
   {
      char    *bytdata;
      u_int   bytlength;
    };

void
xdr_mybytes(xdrsp,arrp)
XDR  *xdrsp;
struct mybytes *arrp;
{
   xdr_bytes(xdrsp,(caddr_t *)&arrp->bytdata,&arrp->bytlength,MAXLEN);
}

...
char *bytc_in ,*bytc_out;
...
```

```
u_long procnum;
register CLIENT *clnt;
enum clnt_stat cs;
struct timeval  total_timeout;
...
total_timeout.tv_sec = 20;
total_timeout.tv_usec = 0;
...

mybytc_in.bytdata =  bytc_in;
mybytc_in.bytlength = strlen(bytc_in)+1;
cs=clnt_call(clnt, procnum, xdr_mybytes, (caddr_t *) &mybytc_in,
             xdr_mybytes, (caddr_t *)&mybytc_out, total_timeout);
if ( cs != RPC_SUCCESS)
    printf("*Error* clnt_call fail :\n");
```

6.13.50 xdr_callhdr()

The xdr_callhdr() call translates between an RPC message header and its external representation.

6.13.50.1 Syntax

```
#include <rpc\rpc.h>

void
xdr_callhdr(xdrs, chdr)
XDR *xdrs;
struct rpc_msg *chdr;
```

6.13.50.2 Parameters

xdrs
: Pointer to the XDR stream

chdr
: Pointer to the call header

6.13.50.3 Description

The xdr_callhdr() call translates between an RPC message header and its external representation.

6.13.51 xdr_callmsg()

The xdr_callmsg() call translates between RPC call messages (header and authentication, not argument data) and their external representations.

6.13.51.1 Syntax

```
#include <rpc\rpc.h>

void
xdr_callmsg(xdrs, cmsg)
XDR *xdrs;
struct rpc_msg *cmsg;
```

6.13.51.2 Parameters

xdrs
: Pointer to the XDR stream

cmsg
: Pointer to the call message

6.13.51.3 Description

The xdr_callmsg() call translates between RPC call messages (header and authentication, not argument data) and their external representations.

6.13.52 xdr_double()

The xdr_double() call translates between C double-precision numbers and their external representations.

6.13.52.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_double(xdrs, dp)
XDR *xdrs;
double *dp;
```

6.13.52.2 Parameters

xdrs
Pointer to the XDR stream

dp
Pointer to a double-precision number

6.13.52.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.52.4 Description

The xdr_double() call translates between C double-precision numbers and their external representations.

6.13.53 xdr_enum()

The xdr_enum() call translates between C-enumerated groups and their external representations.

6.13.53.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_enum(xdrs, ep)
XDR *xdrs;
enum_t *ep;
```

6.13.53.2 Parameters

xdrs
: Pointer to the XDR stream

ep
: Pointer to the enumerated number

6.13.53.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.53.4 Description

The xdr_enum() call translates between C-enumerated groups and their external representations. When you call the procedures callrpc() and registerrpc(), create a stub procedure for both the server and the client before the procedure of the application program using xdr_enum(). Verify that this procedure looks like the following:

```
#include <rpc\rpc.h>

void
static xdr_enum_t(xdrs, ep)
XDR *xdrs;
enum_t *ep;
{
        xdr_enum(xdrs, ep)
}
```

The xdr_enum_t procedure is used as the *inproc* and *outproc* in both the client and server RPCs.

For example, an RPC client would contain the following lines:

```
⋮

error = callrpc(argv[1],ENUMRCVPROG,VERSION,ENUMRCVPROC,
                         xdr_enum_t,&innumber,xdr_enum_t,&outnumber);


⋮
```

An RPC server would contain the following line:

```
⋮

registerrpc(ENUMRCVPROG,VERSION,ENUMRCVPROC,xdr_enum_t,
                    xdr_enum_t);


⋮
```

6.13.54 xdr_float()

The xdr_float() call translates between C floating-point numbers and their external representations.

6.13.54.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_float(xdrs, fp)
XDR *xdrs;
float *fp;
```

6.13.54.2 Parameters

xdrs
: Pointer to the XDR stream

fp
: Pointer to the floating-point number

6.13.54.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.54.4 Description

The xdr_float() call translates between C floating-point numbers and their external representations.

6.13.55 xdr_getpos()

The xdr_getpos() call starts the get-position routine associated with the XDR stream, *xdrs*.

6.13.55.1 Syntax

```
#include <rpc\rpc.h>

u_int
xdr_getpos(xdrs)
XDR *xdrs;
```

6.13.55.2 Parameter

xdrs
    Pointer to the XDR stream

6.13.55.3 Return Value

The xdr_getpos() call returns an unsigned integer, which indicates the position of the XDR byte stream.

6.13.55.4 Description

The xdr_getpos() call starts the get-position routine associated with the XDR stream, *xdrs*.

6.13.55.5 Related Call

xdr_setpos

6.13.56 xdr_inline()

The xdr_inline() call returns a pointer to a continuous piece of the XDR stream's buffer.

6.13.56.1 Syntax

```
#include <rpc\rpc.h>

long *
xdr_inline(xdrs, len)
XDR *xdrs;
int len;
```

6.13.56.2 Parameters

xdrs
: Pointer to the XDR stream

len
: Length in bytes of the desired buffer

6.13.56.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.56.4 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() might return NULL if there is not enough space in the stream buffer to satisfy the request.

6.13.57 xdr_int()

The xdr_int() call translates between C integers and their external representations.

6.13.57.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_int(xdrs, ip)
XDR *xdrs;
int *ip;
```

6.13.57.2 Parameters

xdrs
: Pointer to the XDR stream

ip
: Pointer to the integer

6.13.57.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.57.4 Description

The xdr_int() call translates between C integers and their external representations.

6.13.58 xdr_long()

The xdr_long() call translates between C long integers and their external representations.

6.13.58.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_long(xdrs, lp)
XDR *xdrs;
long *lp;
```

6.13.58.2 Parameters

xdrs
: Pointer to an XDR stream

lp
: Pointer to the long integer

6.13.58.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.58.4 Description

The xdr_long() call translates between C long integers and their external representations.

6.13.59 xdr_opaque()

The xdr_opaque() call translates between fixed-size opaque data and its external representation.

6.13.59.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_opaque(xdrs, cp, cnt)
XDR *xdrs;
char *cp;
u_int cnt;
```

6.13.59.2 Parameters

xdrs
Pointer to an XDR stream

cp
Pointer to the opaque object

cnt
Size of the opaque object

6.13.59.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.59.4 Description

The xdr_opaque() call translates between fixed-size opaque data and its external representation.

6.13.60 xdr_opaque_auth()

The xdr_opaque_auth() call translates between RPC message authentications and their external representations.

6.13.60.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_opaque_auth(xdrs, ap)
XDR *xdrs;
struct opaque_auth *ap;
```

6.13.60.2 Parameters

xdrs
: Pointer to an XDR stream

ap
: Pointer to the opaque authentication information

6.13.60.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.60.4 Description

The xdr_opaque_auth() call translates between RPC message authentications and their external representations.

6.13.61 xdr_pmap()

The xdr_pmap() call translates an RPC procedure identification, such as is used in calls to Portmapper.

6.13.61.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_pmap(xdrs, regs)
XDR *xdrs;
struct pmap *regs;
```

6.13.61.2 Parameters

xdrs
: Pointer to an XDR stream

regs
: Pointer to the PORTMAP parameters

6.13.61.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.61.4 Description

The xdr_pmap() call translates an RPC procedure identification, such as is used in calls to Portmapper.

6.13.62 xdr_pmaplist()

The xdr_pmaplist() call translates a variable number of RPC procedure identifications, such as those Portmapper creates.

6.13.62.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_pmaplist(xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;
```

6.13.62.2 Parameters

xdrs
: Pointer to an XDR stream

rp
: Pointer to a pointer to the PORTMAP data array

6.13.62.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.62.4 Description

The xdr_pmaplist() call translates a variable number of RPC procedure identifications, such as those Portmapper creates.

6.13.63 xdr_reference()

The xdr_reference() call provides pointer chasing within structures.

6.13.63.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_reference(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;
```

6.13.63.2 Parameters

xdrs
: Pointer to an XDR stream

pp
: Pointer to a pointer

size
: Size of the target

proc
: XDR procedure that translates an individual element of the type addressed by the pointer

6.13.63.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.63.4 Description

The xdr_reference() call provides pointer chasing within structures.

6.13.64 xdr_rejected_reply()

The xdr_rejected_reply() call translates between rejected RPC reply messages and their external representations.

6.13.64.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_rejected_reply(xdrs, rr)
XDR *xdrs;
struct rejected_reply *rr;
```

6.13.64.2 Parameters

xdrs
: Pointer to an XDR stream

rr
: Pointer to the rejected reply

6.13.64.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.64.4 Description

The xdr_rejected_reply() call translates between rejected RPC reply messages and their external representations.

6.13.65 xdr_replymsg()

The xdr_replymsg() call translates between RPC reply messages and their external representations.

6.13.65.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_replymsg(xdrs, rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;
```

6.13.65.2 Parameters

xdrs
: Pointer to an XDR stream

rmsg
: Pointer to the reply message

6.13.65.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.65.4 Description

The xdr_replymsg() call translates between RPC reply messages and their external representations.

6.13.66 xdr_setpos()

The xdr_setpos() starts the set-position routine associated with a XDR stream, *xdrs*.

6.13.66.1 Syntax

```
#include <rpc\rpc.h>

int
xdr_setpos(xdrs, pos)
XDR *xdrs;
u_int pos;
```

6.13.66.2 Parameters

xdrs
: Pointer to an XDR stream

pos
: Position value obtained from xdr_getpos()

6.13.66.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.66.4 Description

The xdr_setpos() call starts the set-position routine associated with the XDR stream, *xdrs*.

6.13.66.5 Related Call

xdr_getpos

6.13.67 xdr_short()

The xdr_short() call translates between C short integers and their external representations.

6.13.67.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_short(xdrs, sp)
XDR *xdrs;
short *sp;
```

6.13.67.2 Parameters

xdrs
: Pointer to an XDR stream

sp
: Pointer to the short integer

6.13.67.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.67.4 Description

The xdr_short() call translates between C short integers and their external representations.

6.13.68 xdr_string()

The xdr_string() call translates between C strings and their external representations.

6.13.68.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_string(xdrs, sp, maxsize)
XDR *xdrs;
char **sp;
u_int maxsize;
```

6.13.68.2 Parameters

xdrs	Pointer to an XDR stream
sp	Pointer to a pointer to the string
maxsize	Maximum size of the string

6.13.68.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.68.4 Description

The xdr_string() call translates between C strings and their external representations.

6.13.69 xdr_u_int()

The xdr_u_int() call translates between C unsigned integers and their external representations.

6.13.69.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_u_int(xdrs, up)
XDR *xdrs;
unsigned *up;
```

6.13.69.2 Parameters

xdrs
: Pointer to an XDR stream

up
: Pointer to the unsigned integer

6.13.69.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.69.4 Description

The xdr_u_int() call translates between C unsigned integers and their external representations.

6.13.70 xdr_u_long()

The xdr_u_long() call translates between C unsigned long integers and their external representations.

6.13.70.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_u_long(xdrs, ulp)
XDR *xdrs;
u_long *ulp;
```

6.13.70.2 Parameters

xdrs
: Pointer to an XDR stream

ulp
: Pointer to the unsigned long integer

6.13.70.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.70.4 Description

The xdr_u_long() call translates between C unsigned long integers and their external representations.

6.13.71 xdr_u_short()

The xdr_u_short() call translates between C unsigned short integers and their external representations.

6.13.71.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_u_short(xdrs, usp)
XDR *xdrs;
u_short *usp;
```

6.13.71.2 Parameters

xdrs
: Pointer to an XDR stream

usp
: Pointer to the unsigned short integer

6.13.71.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.71.4 Description

The xdr_u_short() call translates between C unsigned short integers and their external representations.

6.13.72 xdr_union()

The xdr_union() call translates between a discriminated C union and its external representation.

6.13.72.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_union(xdrs, dscmp, unp, choices, dfault)
XDR *xdrs;
int *dscmp;
char *unp;
struct xdr_discrim *choices;
xdrproc_t dfault;
```

6.13.72.2 Parameters

xdrs
: Pointer to an XDR stream

dscmp
: Pointer to the union's discriminant

unp
: Pointer to the union

choices
: Pointer to an array detailing the XDR procedure to use on each arm of the union

dfault
: Default XDR procedure to use

6.13.72.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.72.4 Description

The xdr_union() call translates between a discriminated C union and its external representation.

6.13.73 xdr_vector()

The xdr_vector() call translates between a fixed-length array and its external representation.

6.13.73.1 Syntax

```
#include <rpc\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
```

6.13.73.2 Parameters

xdrs
: Pointer to the XDR stream

basep
: Pointer to the base of the array

nelem
: Element count of the array

elemsize
: Size of each of the array's elements, found by using the sizeof() operator

xdr_elem
: Pointer to the XDR routine that translates an individual array element

6.13.73.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.73.4 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.

6.13.74 xdr_void()

The xdr_void() call returns a value of 1.

6.13.74.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_void()
```

6.13.74.2 Return Value

The xdr_void() call always returns a value of 1.

6.13.74.3 Description

The xdr_void() call is used like a command that does not require any other XDR functions. You can place this call in the *inproc* or *outproc* parameter of the clnt_call() function when you do not need to move data.

6.13.74.4 Related Calls

callrpc()
clnt_broadcast()
clnt_call()
clnt_freeres()
pmap_rmtcall()
registerrpc()
svc_freeargs()
svc_getargs()
svc_sendreply()

6.13.75 xdr_wrapstring()

The xdr_wrapstring() call translates between strings and their external representations.

6.13.75.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_wrapstring(xdrs, sp)
XDR *xdrs;
char **sp;
```

6.13.75.2 Parameters

xdrs
: Pointer to an XDR stream

sp
: Pointer to a pointer to the string

6.13.75.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.75.4 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 start two-parameter XDR routines, and xdr_string() is a three-parameter routine.

6.13.76 xdrmem_create()

The xdrmem_create() call initializes the XDR stream pointed to by *xdrs*.

6.13.76.1 Syntax

```
#include <rpc\rpc.h>

void
xdrmem_create(xdrs, addr, size, op)
XDR *xdrs;
char *addr;
u_int size;
enum xdr_op op;
```

6.13.76.2 Parameters

xdrs
: Pointer to an XDR stream

addr
: Pointer to the memory location

size
: Maximum size of *addr*, in multiples of 4

op
: The direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE)

6.13.76.3 Description

The xdrmem_create() call initializes the XDR stream pointed to by *xdrs*. Data is written to, or read from, *addr*.

6.13.77 xdrrec_create()

The xdrrec_create() call initializes the XDR stream pointed to by *xdrs*.

6.13.77.1 Syntax

```
#include <rpc\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)();
```

6.13.77.2 Parameters

xdrs
: Pointer to an XDR stream.

sendsize
: Size of the send buffer. Specify 0 to choose the default.

recvsize
: Size of the receive buffer. Specify 0 to choose the default.

handle
: First parameter passed to *readit*() and *writeit*().

*readit***()**
: Called when a stream's input buffer is empty.

*writeit***()**
: Called when a stream's output buffer is full.

6.13.77.3 Description

The xdrrec_create() call initializes the XDR stream pointed to by *xdrs*.

Note: The caller must set the *op* field in the *xdrs* structure.

Warning: This XDR procedure implements an intermediate record string. Additional bytes in the XDR stream provide record boundary information.

6.13.78 xdrrec_endofrecord()

The xdrrec_endofrecord() call marks the data in the output buffer as a completed record.

6.13.78.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdrrec_endofrecord(xdrs, sendnow)
XDR *xdrs;
int sendnow;
```

6.13.78.2 Parameters

xdrs
: Pointer to an XDR stream

sendnow
: Specifies nonzero to write out data in the output buffer

6.13.78.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.78.4 Description

You can start the xdrrec_endofrecord() call only on streams created by xdrrec_create(). Data in the output buffer is marked as a complete record.

6.13.79 xdrrec_eof()

The xdrrec_eof() call marks the end of the file, after using the rest of the current record in the XDR stream.

6.13.79.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdrrec_eof(xdrs)
XDR *xdrs;
int empty;
```

6.13.79.2 Parameter

xdrs
Pointer to an XDR stream

6.13.79.3 Return Values

The value 1 indicates the current record has been consumed; the value 0 indicates continued input on the stream.

6.13.79.4 Description

You can start the xdrrec_eof() call only on streams created by xdrrec_create().

6.13.80 xdrrec_skiprecord()

The xdrrec_skiprecord() call discards the rest of the XDR stream's current record in the input buffer.

6.13.80.1 Syntax

```
#include <rpc\rpc.h>

bool_t
xdrrec_skiprecord(xdrs)
XDR *xdrs;
```

6.13.80.2 Parameter

xdrs
Pointer to an XDR stream

6.13.80.3 Return Values

The value 1 indicates success; the value 0 indicates an error.

6.13.80.4 Description

You can start the xdrrec_skiprecord() call only on streams created by xdrrec_create(). The XDR implementation is instructed to discard the remaining data in the input buffer.

6.13.80.5 Related Call

xdrrec_create()

6.13.81 xdrstdio_create()

The xdrstdio_create() call initializes the XDR stream pointed to by *xdrs*.

6.13.81.1 Syntax

```
#include <rpc\rpc.h>
#include <stdio.h>

void
xdrstdio_create(xdrs, file, op)
XDR *xdrs;
FILE *file;
enum xdr_op op;
```

6.13.81.2 Parameters

xdrs
: Pointer to an XDR stream

file
: File name for the input and output stream

op
: The direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE)

6.13.81.3 Description

The xdrstdio_create() call initializes the XDR stream pointed to by *xdrs*. Data is written to or read from the standard I/O stream or file.

6.13.82 xprt_register()

The xprt_register() call registers service transport handles with the RPC service package.

6.13.82.1 Syntax

```
#include <rpc\rpc.h>

void
xprt_register(xprt)
SVCXPRT *xprt;
```

6.13.82.2 Parameter

xprt
Pointer to the service transport handle

6.13.82.3 Description

The xprt_register() call registers service transport handles with the RPC service package. This routine also modifies the global variable svc_socks[].

6.13.82.4 Related Call

svc_register()

6.13.83 xprt_unregister()

The xprt_unregister() call unregisters the RPC service transport handle.

6.13.83.1 Syntax

```
#include <rpc\rpc.h>

void
xprt_unregister(xprt)
SVCXPRT *xprt;
```

6.13.83.2 Parameter

xprt

Pointer to the service transport handle

6.13.83.3 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_socks[].

7.0 File Transfer Protocol Application Programming Interface

This chapter describes the FTP API routines supported by TCP/IP for 4690OS. The File Transfer Protocol (FTP) API allows applications to have a client interface for file transfer. Applications written to this interface can communicate with multiple FTP servers at the same time. The interface supports a maximum of 256 simultaneous connections and enables third-party proxy transfers between pairs of FTP servers. Consecutive third-party transfers are allowed between any sequence of pairs of FTP servers.

The FTP API tracks the servers to which an application is currently connected. When a new request for FTP service is requested, the API checks whether a connection to the server exists and establishes one if it does not exist. If the server has dropped the connection since last use, the API re-establishes it.

7.1 FTP API Call Library

To use the FTP API described in this chapter, you must have the <FTPAPI.H> header file on your system. The FTP API routines are in the ADXHSITL.L86 file.

Put the following statement at the top of any file using FTP API code:

```
#include <ftpapi.h>
```

Define the 4690 variable to the compiler by doing the following:

- Place `#define OS2` at the top of each file that includes TCP/IP header files.

7.2 Return Values

Most functions return a value of −1 to indicate failure and a value of 0 to indicate success. Two functions do not return 0 and −1 values: ftplogoff(), which is of type void, and ping(), which returns an error code rather than storing the return value in *ftperrno*. When the value is −1, the global integer variable *ftperrno* is set to one of the following codes:

Return Code	**Description**
FTPSERVICE	Unknown service.
FTPHOST	Unknown host.
FTPSOCKET	Unable to obtain socket.
FTPCONNECT	Unable to connect to server.
FTPLOGIN	Login failed.
FTPABORT	Transfer aborted.
FTPLOCALFILE	Problem opening the local file.
FTPDATACONN	Problem initializing data connection.
FTPCOMMAND	Command failed.
FTPPROXYTHIRD	Proxy server does not support third party.
FTPNOPRIMARY	No primary connection for proxy transfer.

7.3 FTP API Calls

This section provides the syntax, parameters, and other appropriate information for each FTP API call supported by TCP/IP for 4690OS.

7.3.1 ftpappend()

The ftpappend() call appends information to a remote file.

7.3.1.1 Syntax

```
#include <ftpapi.h>

int ftpappend(host, userid, passwd, acct, local,
              remote, transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
int transfertype;
```

7.3.1.2 Parameters

host	Host running the FTP server.
userid	ID used for logon.
passwd	Password of the user ID.
acct	Account (when needed); can be NULL.
local	Local file name.
remote	Remote file name.
transfertype	Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

7.3.1.3 Return Values

The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.1.4 Description

The ftpappend() call appends information to a remote file.

7.3.1.5 Example

```
int rc;
rc=ftpappend("conypc","jason","ehgr1",NULL,"abc.doc","new.doc",T_ASCII);
```

The local ASCII file `abc.doc` is appended to the file `new.doc` in the current working directory at the host `conypc`.

7.3.2 ftpcd()

The ftpcd() call changes the current working directory on a host.

7.3.2.1 Syntax

```
#include <ftpapi.h>

int ftpcd(host, userid, passwd, acct, dir)
char *host;
char *userid;
char *passwd;
char *acct,
char *dir;
```

7.3.2.2 Parameters

host	Host running the FTP server
userid	ID used for logon
passwd	Password of the user ID
acct	Account (when needed); can be NULL
dir	New working directory

7.3.2.3 Return Values

The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.2.4 Description

The ftpcd() call changes the current working directory.

7.3.2.5 Example

```
int rc;
rc=ftpcd("conypc","jason","ehgr1",NULL,"mydir");
```

The current working directory is changed to `mydir` on the host `conypc` using the user ID `jason` and the password `ehgr1`.

7.3.3 ftpdelete()

The ftpdelete() call deletes files on a remote host.

7.3.3.1 Syntax

```
#include <ftpapi.h>

int ftpdelete(host, userid, passwd, acct, name)
char *host;
char *userid;
char *passwd;
char *acct;
char *name;
```

7.3.3.2 Parameters

host	Host running the FTP server
userid	ID used for logon
passwd	Password of the user ID
acct	Account (when needed); can be NULL
name	File to be deleted

7.3.3.3 Return Values

The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.3.4 Description

The ftpdelete() call deletes a file on a host.

7.3.3.5 Example

```
int rc;
rc=ftpdelete("conypc","jason","ehgr1",NULL,"abc.1");
```

The file `abc.1` is deleted on the host `conypc` using the user ID `jason` and the password `ehgr1`.

7.3.4 ftpdir()

The ftpdir() call gets a directory in wide format from a host.

7.3.4.1 Syntax

```
#include <ftpapi.h>

int ftpdir(host, userid, passwd, acct, local, pattern,)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *pattern;
```

7.3.4.2 Parameters

host	Host running the FTP server.
userid	ID used for logon.
passwd	Password of the user ID.
acct	Account (when needed); can be NULL.
local	Local file name.
pattern	The file name or pattern of the files to be listed on the foreign host. Patterns are any combination of ASCII characters. The following two characters have special meaning:

*	Shows that any character or group of characters can occupy that position in the pattern.
?	Shows that any single character can occupy that position in the pattern.

7.3.4.3 Return Values

The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.4.4 Description

The ftpdir() call gets a directory in wide format from a host.

7.3.4.5 Example

```
int rc;
rc=ftpdir("conypc","jason","ehgr1",NULL,"conypc.dir","*.c");
```

ftpdir() gets a directory of `*.c` files in wide format, and stores the directory in a local file, `conypc.dir`.

7.3.5 ftpget()

The ftpget() call gets a file from an FTP server.

7.3.5.1 Syntax

```
#include <ftpapi.h>

int ftpget(host, userid, passwd, acct, local, remote,
           mode, transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
char *mode;
int transfertype;
```

7.3.5.2 Parameters

host	Host running the FTP server.
userid	ID used for logon.
passwd	Password of the user ID.
acct	Account (when needed); can be NULL.
local	Local file name.
remote	Remote file name.
mode	Either *w* for write or *a* for append.
transfertype	Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

7.3.5.3 Return Values

The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.5.4 Description

The ftpget() call gets a file from an FTP server.

7.3.5.5 Example

```
int rc;
rc=ftpget("conypc","jason","ehgr1",NULL,"new.doc","abc.doc","w",T_ASCII);
```

The system copies the ASCII file `abc.doc` on the host `conypc` into the local current working directory as the file `new.doc`. If the file `new.doc` already exists in the local current working directory, the contents of the file `abc.doc` overwrite the file `new.doc`.

7.3.6 ftplogoff()

The ftplogoff() call closes all current connections.

7.3.6.1 Syntax

```
#include <ftpapi.h>

void ftplogoff()
```

7.3.6.2 Description

The ftplogoff() call closes all current connections. An application must call this before terminating.

7.3.7 ftpls()

The ftpls() call gets directory information in short format from a remote host and writes it to a local file.

7.3.7.1 Syntax

```
#include <ftpapi.h>

int ftpls(host, userid, passwd, acct, local, pattern)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *pattern;
```

7.3.7.2 Parameters

host	Host running the FTP server.
userid	ID used for logon.
passwd	Password of the user ID.
acct	Account (when needed); can be NULL.
local	Local file into which the information is placed.
pattern	The file name or pattern of the files to be listed on the foreign host. Patterns are any combination of ASCII characters. The following two characters have special meaning:

- * Shows that any character or group of characters can occupy that position in the pattern.
- ? Shows that any single character can occupy that position in the pattern.

7.3.7.3 Return Values

The value 0 indicates success; the value –1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.7.4 Description

The ftpls() call gets directory information in short format from a host and writes it to a local file.

7.3.7.5 Example

```
int rc;
rc=ftpls("conypc","jason","ehgr1",NULL,"conypc.dir","*.c");
```

ftpls() gets a directory of `*.c` files in short format and stores the names in the local file `conypc.dir`.

7.3.8 ftpmkd()

The ftpmkd() call creates a new directory on a target machine.

7.3.8.1 Syntax

```
#include <ftpapi.h>

int ftpmkd(host, userid, passwd, acct, dir)
char *host;
char *userid;
char *passwd;
char *acct;
char *dir;
```

7.3.8.2 Parameters

host	Host running the FTP server
userid	ID used for logon
passwd	Password of the user ID
acct	Account (when needed); can be NULL
dir	Directory to be created

7.3.8.3 Return Values

The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.8.4 Description

The ftpmkd() call creates a new directory on a host.

7.3.8.5 Example

```
int rc;
rc=ftpmkd("conypc","jason","ehgr1",NULL,"mydir");
```

The directory `mydir` is created on the host `conypc`, using the user ID `jason` and the password `ehgr1`.

7.3.9 ftpproxy()

The ftpproxy() call transfers a file between two remote servers without sending the file to the local host.

7.3.9.1 Syntax

```
#include <ftpapi.h>

int ftpproxy(host1, userid1, passwd1, acct1, host2, userid2,
              passwd2, acct2, fn1, fn2, transfertype)
char *host1;
char *userid1;
char *passwd1;
char *acct1;
char *host2;
char *userid2;
char *passwd2;
char *acct2;
char *fn1;
char *fn2;
int transfertype;
```

7.3.9.2 Parameters

host1	Target host running the FTP server.
userid1	ID used for logon on host 1.
passwd1	Password of the user ID on host 1.
acct1	Account for host 1 (when needed); can be NULL.
host2	Source host running the FTP server.
userid2	ID used for logon on host 2.
passwd2	Password of the user ID on host 2.
acct2	Account for host 2 (when needed); can be NULL.
fn1	File to be written on host 1.
fn2	File to be copied from host 2.
transfertype	Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

7.3.9.3 Return Values

The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.9.4 Description

The ftpproxy() call copies a file on a specified source host directly to a specified target host, without involving the requesting host in the file transfer. This call is functionally the same as the FTP client subcommand *proxy put*.

Note: For ftpproxy() to complete successfully, both the source and the target hosts must be running the FTP servers.

7.3.9.5 Example

```
int rc;
rc=ftpproxy("pc1","oleg","erst",NULL,  /* target host information*/
            "pc2","yan", "dssa1", NULL, /* source host information*/
            "\tmp\newdoc.1",            /* target file name */
            "\tmp\doc.1",               /* source file name */
            T_ASCII);                   /* ascii transfer */
```

The ASCII file `\tmp\doc.1` on the host `pc2` is copied to host `pc1` as the file `\tmp\newdoc.1`.

7.3.10 ftpput()

The ftpput() call transfers a file to a host.

7.3.10.1 Syntax

```
#include <ftpapi.h>

int ftpput(host, userid, passwd, acct, local, remote, transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
int transfertype;
```

7.3.10.2 Parameters

host	Host running the FTP server.
userid	ID used for logon.
passwd	Password of the user ID.
acct	Account (when needed); can be NULL.
local	Local file name.
remote	Remote file name.
transfertype	Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

7.3.10.3 Return Values

The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.10.4 Description

The ftpput() call transfers a file to an FTP server.

7.3.10.5 Example

```
int rc;
rc=ftpput("conypc","jason","ehgr1",NULL,"abc.doc","new.doc",T_ASCII);
```

The system copies the ASCII file `abc.doc` on the local current working directory to the current working directory of the host `conypc` as file `new.doc`. If the file `new.doc` already exists, the contents of the file `abc.doc` overwrite the file `new.doc`.

7.3.11 ftpputunique()

The ftpputunique() call transfers a file to a host and ensures it is created with a unique name.

7.3.11.1 Syntax

```
#include <ftpapi.h>

int ftpputunique(host, userid, passwd, acct, local, remote,
                 transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
int transfertype;
```

7.3.11.2 Parameters

host	Host running the FTP server.
userid	ID used for logon.
passwd	Password of the user ID.
acct	Account (when needed); can be NULL.
local	Local file name.
remote	Remote file name.
transfertype	Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

7.3.11.3 Return Values

The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.11.4 Description

The ftpputunique() call copies a local file to a file on a specified host. It guarantees that the new file will have a unique name and that the new file will not overwrite a file with the same name. If the file already exists on the host, a new and unique file name is created and used as the target of the file transfer.

7.3.11.5 Example

```
int rc;
rc=ftpputunique(
       "conypc","jason","ehgr1",NULL,"abc.doc", "new.doc",T_ASCII);
```

The ASCII file `abc.doc` is copied to the current working directory of the host `conypc` as file `new.doc`, unless the file `new.doc` already exists. If the file `new.doc` already exists, the file `new.doc` is given a new name unique within the current working directory on the host `conypc`. The name of the new file is displayed upon successful completion of the file transfer.

7.3.12 ftpquote()

The ftpquote() call sends a string to the server verbatim.

7.3.12.1 Syntax

```
#include <ftpapi.h>

int ftpquote(host, userid, passwd, acct, quotestr)
char *host;
char *userid;
char *passwd;
char *acct;
char *quotestr;
```

7.3.12.2 Parameters

host	Host running the FTP server
userid	ID used for logon
passwd	Password of the user ID
acct	Account (when needed); can be NULL
quotestr	Quote string to be passed to the FTP server verbatim

7.3.12.3 Return Values

The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.12.4 Description

The ftpquote() call sends a string to the server verbatim.

7.3.12.5 Example

```
int rc;
rc=ftpquote("conypc","jason","ehgr1",NULL,"site idle 2000");
```

The idle is set to time out in 2000 seconds. Your server might not support that amount of idle time.

7.3.13 ftprename()

The ftprename() call renames a file on a remote host.

7.3.13.1 Syntax

```
#include <ftpapi.h>

int ftprename(host, userid, passwd, acct, namefrom, nameto)
char *host;
char *userid;
char *passwd;
char *acct;
char *namefrom;
char *nameto;
```

7.3.13.2 Parameters

host	Host running the FTP server
userid	ID used for logon
passwd	Password of the user ID
acct	Account (when needed); can be NULL
namefrom	Original file name
nameto	New file name

7.3.13.3 Return Values

The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.13.4 Description

The ftprename() call renames a file on a host.

7.3.13.5 Example

```
int rc;
rc=ftprename("conypc","jason","ehgr1",NULL,"abc.1","cd.fg");
```

The file `abc.1` is renamed to `cd.fg` on host `conypc`, using user ID `jason`, with password `ehgr1`.

7.3.14 ftprmd()

The ftprmd() call removes a directory on a target machine.

7.3.14.1 Syntax

```
#include <ftpapi.h>

int ftprmd(host, userid, passwd, acct, dir)
char *host;
char *userid;
char *passwd;
char *acct;
char *dir;
```

7.3.14.2 Parameters

host	Host running the FTP server
userid	ID used for logon
passwd	Password of the user ID
acct	Account (when needed); can be NULL
dir	Directory to be removed

7.3.14.3 Return Values

The value 0 indicates success; the value –1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.14.4 Description

The ftprmd() call removes a directory on a host.

7.3.14.5 Example

```
int rc;
rc=ftprmd("conypc","jason","ehgr1",NULL,"mydir");
```

The directory, `mydir`, is removed on the host, `conypc`, using the user ID, `jason`, and the password, `ehgr1`.

7.3.15 ftpsite()

The ftpsite() call executes the **site** command. (For more information about the **site** command, see 4.4, “adxhsigl(ftp)” on page 56.

7.3.15.1 Syntax

```
#include <ftpapi.h>

int ftpsite(host, userid, passwd, acct, sitestr)
char *host;
char *userid;
char *passwd;
char *acct;
char *sitestr;
```

7.3.15.2 Parameters

host	Host running the FTP server
userid	ID used for logon
passwd	Password of the user ID
acct	Account (when needed); can be NULL
sitestr	Site string to be executed

7.3.15.3 Return Values

The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

7.3.15.4 Description

The ftpsite() call executes the site command.

7.3.15.5 Example

```
int rc;
rc=ftpsite("conypc","jason","ehgr1",NULL,"idle 2000");
```

The idle is set to time out in 2000 seconds. Your server might not support that amount of idle time.

7.3.16 ping()

The ping() call sends a ping to the remote host to determine if that host is responding.

7.3.16.1 Syntax

```
#include <ftpapi.h>

int ping(addr, len)
unsigned long addr;
int len;
```

7.3.16.2 Parameters

addr	Internet address of the host in network byte order
len	Length of the ping packets

7.3.16.3 Return Values

If the return value is positive, the return value is the number of milliseconds it took for the echo to return. If the return value is negative, it contains an error code.

The following are ping() call return codes and their corresponding descriptions:

Return Code	**Description**
PINGREPLY	Host does not reply
PINGSOCKET	Unable to obtain socket
PINGPROTO	Unknown protocol ICMP
PINGSEND	Send failed
PINGRECV	Recv failed

7.3.16.4 Description

The ping() call sends a ping to the host with ICMP Echo Request. The ping() call is useful to determine whether the host is alive before attempting FTP transfers, because time-out on regular connections is more than a minute. The ping() call returns within 3 seconds, at most, if the host is not responding.

7.3.16.5 Example

```
#include <stdio.h>
#include <netdb.h>
#include <ftpapi.h>

struct hostent *hp;     /* Pointer to host info */

main(argc, argv, envp)
int argc;
char *argv[];
char *envp[];
{
   int i;
   unsigned long addr;

   if (argc!=2) {
       printf("Usage: p <host>\n");
       exit(1);
   }

   hp = gethostbyname(argv[1]);
   if (hp) {
           memcpy( (char *)&addr, hp->h_addr, hp->h_length);
           i = ping(addr,256);
           printf("ping reply in %d milliseconds\n",i);
   } else {
           printf("unknown host\n");
           exit(2);
   }
   ftplogoff(); /* close all connections */
}
```

8.0 SNMP Agent Distributed Program Interface (DPI)

This chapter describes the SNMP DPI routines supported by TCP/IP for 4690OS. 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.

8.1 SNMP Agents and Subagents

SNMP agents are responsible for performing network management functions by network management stations. Examples of management functions are GET, GETNEXT, and SET, performed on the MIB elements.

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

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

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

A subagent of the SNMP agent determines the port number by sending a GET request for an MIB variable, which represents the value of the TCP port. The subagent is not required to create and parse SNMP packets, because the DPI API has a library routine query_DPI_port(). After the subagent obtains the value of the DPI TCP port, it should make a TCP connection to the appropriate port. After a successful 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.

8.2 Processing DPI Requests

The SNMP agent can initiate three DPI requests: GET, SET, and GETNEXT. 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 following section describes the requests that the SNMP subagent can initiate.

8.2.1 Processing a GET Request

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

```
u_char *cp;

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

If the object ID of the variable is supported, the subagent does not return an error; it returns the name, type, and value of the object ID, using the mkDPIset() and mkDPIresponse() routines. The following is an example of an object ID, whose type is string, being returned:

```
char *obj_id;

u_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);
```

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

8.2.3 Processing a GETNEXT Request

Parsing a GETNEXT 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, but they cannot jump from one group to another. You must first determine the reason for the request in order to 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 the subagent supports.

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

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

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

1. 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)
   ```

2. Subagent 2 is then called:

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

3. Subagent 1 is then called:

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

8.2.4 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:

```
u_char *cp;

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

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

8.2.5 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 routines mkDPItrap() and mkDPItrape() can be used to generate TRAP packets.

8.3 DPI Library

To use the DPI library routines provided with TCP/IP for 4690OS, you must have the <DPI\SNMP_DPI.H> header file on your system.

The ADXHSIDL.L86 file contains the DPI library routines.

You must define the 4690 variable to the compiler by doing the following:

- Place `#define OS2` at the top of each file that includes TCP/IP header files.

For more information about SNMP, see 3.0, “Users Information” on page 31.

8.4 DPI Library Routines

This section provides the syntax, parameters, and other appropriate information for each DPI routine supported by TCP/IP for 4690OS.

8.4.1 fDPIparse()

The fDPIparse() routine frees a parse tree that was previously created by a call to pDPIpacket().

8.4.1.1 Syntax

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

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

8.4.1.2 Parameter

hdr Parse tree

8.4.1.3 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.

8.4.1.4 Related Call

pDPIpacket()

8.4.2 mkDPIregister()

The mkDPIregister() routine creates a register request packet and returns a pointer to a static buffer.

8.4.2.1 Syntax

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

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

8.4.2.2 Parameter

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

8.4.2.3 Return Values

If successful, mkDPIregister() 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.

8.4.2.4 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.

8.4.2.5 Example

```
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 */
```

8.4.3 mkDPIresponse()

The mkDPIresponse() routine creates a response packet.

8.4.3.1 Syntax

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

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

8.4.3.2 Parameters

ret_code	Error code to be returned
value_list	Pointer to a parse tree containing the name, type, and value information to be returned

8.4.3.3 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.

8.4.3.4 Description

The mkDPIresponse() routine creates a response packet. The *ret_code* parameter 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 <DPI\SNMP_DPI.H> header file for a description of these messages.

If *ret_code* does not indicate an error, then *value_list* points to a parse tree created by mkDPIset(), which represents the name, type, and value information being returned. If an error is indicated, *value_list* 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.

8.4.3.5 Example

```
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 */
```

8.4.4 mkDPIset()

The mkDPIset() routine creates a representation of a parse tree name and value pair.

8.4.4.1 Syntax

```
#include <dpi\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;
```

8.4.4.2 Parameters

oid_name	Pointer to the object identifier
type	Type of the object identifier
len	Length of the value
value	Pointer to the first byte of the value of the object identifier

8.4.4.3 Return Values

The mkDPIset() routine returns a pointer to a parse tree containing the name, type, and value information.

8.4.4.4 Description

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

The *type* parameter can have the following values (defined in the <DPI\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.

8.4.5 mkDPItrap()

The mkDPItrap() routine creates a TRAP request packet.

8.4.5.1 Syntax

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

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

8.4.5.2 Parameters

generic	Generic field in the SNMP TRAP packet
specific	Specific field in the SNMP TRAP packet
value_list	The name and value pair to be placed into the SNMP packet

8.4.5.3 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.

8.4.5.4 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.

8.4.5.5 Example

```
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,
        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);
```

8.4.6 pDPIpacket()

The pDPIpacket() routine parses a DPI packet and returns a parse tree representing its contents.

8.4.6.1 Syntax

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

struct snmp_dpi_hdr *pDPIpacket(packet)
u_char *packet;
```

8.4.6.2 Parameter

packet Pointer to the DPI packet to be parsed

8.4.6.3 Return Values

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

Note: The parse tree structures are defined in the <DPI\SNMP_DPI.H> header file.

8.4.6.4 Description

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

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

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

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

The *packet_type* field can have one of the following values, defined in the <DPI\SNMP_DPI.H> header file:

- SNMP_DPI_GET
- SNMP_DPI_GET_NEXT
- SNMP_DPI_SET

The *packet_type* field indicates the request that is being made of the DPI client. For each of these requests, the remainder of the *packet_body* will be 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 GETNEXT 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 is contained in a *dpi_set_packet* structure.

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

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

- SNMP_TYPE_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.

8.4.7 query_DPI_port()

The query_DPI_port() routine determines what TCP port is associated with DPI.

8.4.7.1 Syntax

```
#include <dpi\snmp_dpi.h>

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

8.4.7.2 Parameters

host_name	Pointer to the SNMP agent’s host name or internet address
community_name	Pointer to the community name to be used when making a request

8.4.7.3 Return Values

If it succeeds, the routine returns an integer representing the TCP port number; it returns a −1 if the port cannot be determined.

8.4.7.4 Description

A DPI client uses the query_DPI_port() routine to determine the TCP port number associated with the DPI. The client needs this port number to connect to the SNMP agent. The port number is obtained through an SNMP GET request. *host_name* and *community_name* are the arguments that are passed to the query_DPI_port() routine.

Appendix A. Sample socket application: Echo server

The following is a simple socket application that demonstrates the use of the 4690OS TCP/IP socket API. It is made up of two parts, a client (CLIENT.C) and a server (SERVER.C).

A.1 Compiling and linking

The MetaWare High C** compiler was used to create these programs. The following commands were used to compile CLIENT.C and SERVER.C:

- hcdx86 client.c -def ON4680 -def OS2 -def IS_MAIN
- hcdx86 server.c -def ON4680 -def OS2 -def IS_MAIN

To create executable files CLIENT.286 and SERVER.286, the following link commands were used:

- link86 client[stack[add[1000]]]=client, adxhsisl.l86 [s], hcbe.l86 [s], flexlib.l86 [s]
- link86 server[stack[add[1000]]]=server, adxhsisl.l86 [s], hcbe.l86 [s], flexlib.l86 [s], adxapacl.l86

Note that the server application uses the ADX_CSERVE call to display messages to the background process status screen, thus necessitating the use of the adxapacl.l86 library. Your link command may vary depending on which libraries you need for your application, and where these libraries are located on your system.

A.2 Source code: CLIENT.C

```
/*
 * Include Files.
 */
#ifdef ON4680
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <h\types.h>
#include <h\utils.h>
#include <netinet\in.h>
#include <h\sys\socket.h>
#include <h\ioctl.h>
#include <h\netdb.h>

extern tcperrno;
#include <errno.h>
#include <h\nerrno.h>
#define errno tcperrno

extern void sock_init();
extern int soclose();

void tcperror();
#define perror tcperror

#endif

#ifdef UNIX
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netdb.h>
#include <fcntl.h>
#include <errno.h>

extern int close();

#endif

extern u_short bswap();
extern int connect();
extern int recv();
extern int send();
extern int socket();
extern void exit();


/* Client Main. */
main(argc, argv)
int argc;
char **argv;
{
unsigned short port;       /* port client will connect to              */
char buf[550];             /* data buffer for sending and receiving    */
struct hostent *hostnm;    /* server host name information             */
struct sockaddr_in server; /* server address                           */
int s;                     /* client socket                            */
char message[100];
int i;
unsigned long in;
int rc;

/* Check Arguments Passed. Should be hostname and port. */
if (argc < 3)
   {
   printf("Usage: %s hostname port\n", argv[0]);
   exit(1);
   }
if (argc == 3) strcpy(message,"Hello World");
else
   {
   strcpy(message,"");
   for ( i = 3; i < argc; i++)
       {
       strcat(message,argv[i]);
       strcat(message," ");
       }
   }

#ifdef ON4680
/* Initialize with sockets. */
sock_init();
#endif
```

```
/* The host name is the first argument. Get the server address. */

/* First see if internet address was entered */
in = inet_addr(argv[1]);
hostnm=gethostbyaddr((char *) &in,sizeof(struct in_addr),AF_INET);
if (hostnm == (struct hostent *) 0)
   {
   /* See if host name was entered */
   hostnm = gethostbyname(argv[1]);
   if (hostnm == (struct hostent *) 0)
      {
      printf("inet_addr returned %8.8lx\n",in);
      printf("gethostbyaddr failed\n");
      printf("gethostbyname failed\n");
      exit(2);
      }
   }

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

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

/* 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)
   {
   perror("Socket()");
   exit(3);
   }

/* Connect to the server. */
if (connect(s, &server, sizeof(server)) < 0)
   {
   perror("Connect()");
   exit(4);
   }

/* send message to server */
printf("\nSending message to server \n\n==> %s\n\n",buf);
if (send(s, buf, strlen(buf), 0) < 0)
   {
   perror("Send()");
   exit(5);
   }

/* Receive the message returned by the server */
printf("receiving message returned by server \n");
rc = recv(s, buf, sizeof(buf), 0);
if (rc < 0)
   {
```

```
   perror("Recv()");
   exit(6);
   }
buf[rc]=0x00;
printf("\n==> %s\n\n",buf);

/* Close the socket. */
printf("closing socket\n");
#ifdef ON4680
soclose(s);
#else
close(s);
#endif

printf("Client Ended Successfully\n");
return 0;

}

#ifdef ON4680
void tcperror(msg)
char *msg;
{
printf("%s: errno = %d\n",msg,errno);
}
#endif
```

A.3 Source code: SERVER.C

```
/*********************************************************************/
/*                                                                   */
/*                                                                   */
/*                                                                   */
/*                                                                   */
/*                                                                   */
/*                                                                   */
/*********************************************************************/

/*
 * Include Files.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <h\types.h>
#include <h\utils.h>
#include <netinet\in.h>
#include <h\sys\socket.h>
#include <h\ioctl.h>
#include <h\netdb.h>

extern tcperrno;             /* tcp/ip errno returned as tcperrno */
#include <errno.h>
#include <h\nerrno.h>
#define errno tcperrno       /* refer to tcperrno as errno for portability */
```

```
extern void sock_init();     /* socket library routines */
extern int soclose();
extern u_short bswap();
extern int recv();
extern int send();
extern int socket();
extern void exit();
extern int accept();
extern int bind();
extern int listen();
extern int port_cancel();
extern int select();
extern ADX_CSERVE();
/* extern ADX_CTIMER_SET(); */

void dispmsg();
void tcperror();
#define perror tcperror

int tcpip_open();            /* routine to open tcpip link */
int tcpip_recv();            /* routine to receive message from tcpip link */
int tcpip_send();            /* routine to send message to tcpip link */
int tcpip_close();           /* routine to close tcpip link */
void ctl_break();            /* routine to intercept Ctrl-Break key seq */
extern void s_exit();
extern long e_termevent();

int s=0;                     /* socket for accepting connections      */
int ns=0;                    /* socket connected to client            */
unsigned short port;         /* port server binds to                  */
char message[80];            /* character array for displaying messages */

int debug = FALSE;           /* indicates debug mode or not */
int backgrnd = FALSE;        /* indicates running as background task or not */
int inetd = FALSE;           /* if initiated by inetd */

/*
 * Server Main.
 */
main(argc, argv)
int argc;
char **argv;

{
char msg_in[550] = "";
char msg_out[550] = "";
int rc;
int first_time = TRUE;       /* first open includes socket/bind/listen */
int last_time  = FALSE;      /* close both sockets */
int end_session = FALSE;     /* session terminated by client */
int end_prog = FALSE;        /* end the program */
long emask;                  /* event mask for trapping ctrl-c */
int i;                       /* loop counter */
int digit = FALSE;           /* used to test parameters for all digits */
char work[100];              /* work string */
```

```
/* set up a clean up routine in case this programn is aborted */
emask = e_termevent(ctl_break,0L);
if (emask < 0L)
   {
   sprintf(message,"e_terminate failed rc %lx.",emask);
   dispmsg(message);
   s_exit(emask);
   }

/***************** Determine if running in foreground ***********************/

if(argc > 1)
   {
   if((strcmp("backgrnd", argv[1]) == 0) || (strcmp("BACKGRND", argv[1]) == 0))
      {
      backgrnd = TRUE;
      dispmsg ("Running in background mode");
      }
   }

/***************** Determine if running debug mode ************/

if ((argc > 1) && (!backgrnd))       /* no debug in backgrnd mode */
   {
   for (i = 1; i < argc; i++)
       {
       if (strcmp("-d",argv[i]) == 0) debug = TRUE;
       if (strcmp("-D",argv[i]) == 0) debug = TRUE;
       if (debug) dispmsg("Running in background mode");
       }
   }

/* default port number is 10001 */
port = 10001;

/***************** Check for port number over ride ************/

if (argc > 1)
   {
   for (i = 1; i < argc; i++)
       {
       if ((strncmp("-p",argv[i],2) == 0) || (strncmp("-P",argv[i],2) == 0))
          {
          port = atoi((argv[i]+2));
          }
       }
   }

sprintf(message,"Using port %d",port);
dispmsg(message);

/********** Determine if initiated by inetd *******************/

if (backgrnd)                /* if running as a background task  */
   {
   strcpy(work,argv[2]);
   digit=TRUE;
   for (i=0;i<strlen(work);i++)
```

```
         {
         if (!isdigit(work[i])) digit=FALSE;
         }
      if (digit)
         {
         ns=atoi(argv[2]);
         if ((ns >= 0) && (ns < 256))
            {
            inetd=TRUE;
            sock_init();
            printf("Initiated by inetd, pf");
            if (debug) dispmsg("Initiated by inetd");
            rc=0;
            }
         }
      }

   while (!end_prog)
      {
      if (!inetd)
         {
         /* wait for connection */
         /*      first_time = 1 if socket is to be opened & bound to port */
         /*                   0 if socket and port are already bound      */
         rc = tcpip_open(first_time);
         if (rc < 0)
            {
            sprintf(message,"tcpip_open failed - rc = %d, errno = %d",rc,errno);
            dispmsg(message);
            last_time = TRUE;
            first_time = TRUE;
            }
         else
            {
            first_time = FALSE;
            last_time = FALSE;
            strcpy(msg_in,"");
            }
         }

      if (rc >= 0)   /* if open successful or initiated by inetd */
         {
         /* loop until session closed */
         end_session = FALSE;
         while (!end_session)
            {
            /* receive message from socket */
            rc = tcpip_recv(msg_in, sizeof(msg_in));
            if (rc < 0)
               {
               sprintf(message,"tcpip_recv failed - rc = %d, errno = %d",rc,errno);
               dispmsg(message);
               }
            else if (rc == 0)
               {
               sprintf(message,"connection closed from client end");
               dispmsg(message);
               end_session = TRUE;
               }
            else
               {
               msg_in[rc] = 0x00;
               sprintf(message,"%d bytes received",rc);
               dispmsg(message);
               sprintf(message,"==> |%s|",msg_in);
               dispmsg(message);
               }

            /* if message received */
            if (rc > 0)
               {
               strcpy(msg_out, msg_in);
               strcat(msg_out," is returned");
               rc = tcpip_send(msg_out, strlen(msg_out));
               if (rc < 0)
                  {
                  sprintf(message,"tcpip_send failed - rc = %d, errno = %d",rc,errno);
                  dispmsg(message);
                  }
               }
            } /* end of while waiting for EOT */
         if (debug)
            {
            sprintf(message,"session closed");
            dispmsg(message);
            }
         } /* end of rc from tcpip_open OK */
      rc = tcpip_close(last_time);
      if (rc < 0)
         {
         sprintf(message,"tcpip_close failed - rc = %d, errno = %d",rc,errno);
         dispmsg(message);
         }
      if (inetd) end_prog=TRUE;
      } /* end_prog is FALSE */
      return 0;
   }

   /*********************************************************************/
   /* tcpip_open - opens tcpip link if neccesary & waits for connect     */
   /*********************************************************************/
   /*    wait for connection                                             */
   /*    parameters                                                      */
   /*         first_time - 1 if socket is to be opened & bound to port   */
   /*                    - 0 if socket and port are already bound        */
   /*********************************************************************/
   int tcpip_open(first_time)
   int first_time;
   {
   struct sockaddr_in client; /* client address information            */
   struct sockaddr_in server; /* server address information            */
   int namelen;               /* length of client name                 */
   int rc=1;

   if (first_time)
      {
      /* Initialize with sockets. */
      sock_init();

      /* Get a socket for accepting connections. */
      rc = socket(AF_INET, SOCK_STREAM, 0);
      if (rc < 0)
         {
         printf("rc = %d ",rc);
         perror("Socket()");
         }
      else
         {
         s = rc;
         if (debug)
            {
            sprintf(message,"Socket %d received from socket()",s);
            dispmsg(message);
            }
         }

      /* Bind the socket to the server address. */
      if (rc >= 0)
         {
         server.sin_family = AF_INET;
         server.sin_port   = htons(port);
         server.sin_addr.s_addr = INADDR_ANY;

         if (debug)
            {
            sprintf(message,"Binding to port %d to socket %d",port,s);
            dispmsg(message);
            }

         rc = bind(s, &server, sizeof(server));
         if (rc < 0)                             /* if bind fails */
            {
            if (debug)
               {
               sprintf(message,"Canceling sockets bound to port %d",port);
               dispmsg(message);
               }
            rc = port_cancel(htons(port));       /* kick anyone off of our port */
            rc = bind(s, &server, sizeof(server));
            if (rc < 0) perror("Bind()");
            }
         }
      } /* end of first_time processing

   /* Listen for connections. Specify the backlog as 1. */
   if (rc >= 0)
      {
      if (debug)
         {
         sprintf(message,"Listening to socket %d",s);
         dispmsg(message);
         }
      rc = listen(s, 0);
      if (rc != 0)
```

```
      {
      perror("Listen()");
      }
   }

/* Accept a connection. */
if (rc >= 0)
   {
   namelen = sizeof(client);
   sprintf(message,"Accepting connection on socket %d",s);
   dispmsg(message);
   rc = accept(s, &client, &namelen);
   if (rc < 0 )
      {
      perror("Accept()");
      }
   else
      {
      ns = rc;
      if (debug)
         {
         sprintf(message,"Connection accepted on socket %d",ns);
         dispmsg(message);
         }
      }
   }

return rc;
}

/*************************************************************************/
/* tcpip_recv - receive message from client                             */
/*    parameters                                                         */
/*         buf        - address of buffer to receive data read          */
/*         buflen     - length of buffer                                 */
/*************************************************************************/
int tcpip_recv(buf,buflen)
char *buf;
int buflen;
{
int rc;
int sa[1];

/* wait for socket to become ready to read */
rc = 0;
while (rc == 0)
   {
   sa[0] = ns;
   printf("ns = %d %d\n",ns, sa[0]);
   if (debug)
      {
      sprintf(message,"initiating select() waiting on socket %d",ns);
      dispmsg(message);
      }
   rc = select(&sa,1,0,0,(long) 10000);
   if (debug)
      {
      sprintf(message,"returned from select(), rc = %d",rc);
      dispmsg(message);
      }
   if (rc < 0)
      {
      perror("select()");
      }
   }

/* Receive the message on connected socket. */
if (rc > 0)
   {
   if (debug)
      {
      sprintf(message,"Receiving with an %d byte buffer",buflen);
      dispmsg(message);
      }
   rc = recv(ns, buf, buflen, 0);
   if (rc < 0)
      {
      perror("recv()");
      }
   }
return rc;
}

/*********************************************************************/
/* tcpip_send - send message to client                               */
/*    parameters                                                     */
/*         buf        - address of buffer containing data to send    */
/*         buflen     - length of buffer                             */
/*********************************************************************/
int tcpip_send(buf,buflen)
char *buf;
int buflen;
{
int rc;

/* Send the message back to the client. */
if (debug)
   {
   sprintf(message,"Sending a %d byte buffer on socket %d",buflen,ns);
   dispmsg(message);
   }
rc = send(ns, buf, buflen, 0);
if (rc < 0)
   {
   perror("Send()");
   }
return rc;
}

/*********************************************************************/
/* tcpip_close - close session socket &, if requested, port socket   */
/*********************************************************************/
/*    close the connection                                           */
/*    parameters                                                     */
/*         last_time  - 1 if both sockets are to be closed           */
/*                    - 0 if session socket only is to be closed     */
/*                                                                      */
/************************************************************************/
int tcpip_close(last_time)
int last_time;
{
int rc;
if (debug)
   {
   if (last_time)
      {
      dispmsg("Last time - closing both sockets");
      }
   else
      {
      dispmsg("NOT Last time - only closing session socket");
      }
   }

if (debug)
   {
   sprintf(message,"closing socket %d",ns);
   dispmsg(message);
   }
rc = soclose(ns);
if (rc < 0)
   {
   perror("soclose(ns)");
   }

if (last_time)
   {
   if (debug)
      {
      sprintf(message,"closing socket %d",s);
      dispmsg(message);
      }
   rc = soclose(s);
   if (rc < 0)
      {
      perror("soclose(s)");
      }
   }

return rc;
}

/*****************************************************************************/
/*                        Ctl-Break handler                                  */
/*****************************************************************************/
void ctl_break()
{
  sprintf(message,"Server ending due to Ctl-Break.");
  dispmsg(message);
  soclose(ns);                                /* close the sockets */
  soclose(s);
  s_exit(0l);
}
```

```
/**************************************************************************/
/*  DISPMSG - Routine to display a message on the background screen or on   */
/*            the foreground screen                                         */
/**************************************************************************/

void dispmsg(msg)
char *msg;
{
long rc;
int msg_size;

if (backgrnd)
   {
   msg_size=strlen(msg);
   if (msg_size > 46)
      {
      msg_size = 46;
      msg[msg_size] = 0x00;
      }
   ADX_CSERVE(&rc,26,msg,msg_size);
   }
else
   {
   printf("%s\n",msg);
   }
/* ADX_CTIMER_SET(&rc,(unsigned int) 0,(long) 5000); */ /* delay 5 sec */
return;
}


/***************************************************************************/
/*                       perror routine                                     */
/***************************************************************************/
void tcperror(msg)
char *msg;
{
sprintf(message,"%s: errno = %d",msg,errno);
dispmsg(message);
}
```

Appendix B. Index