TCPware® for OpenVMS Programmer's Guide

Part Number: N-6003-60-NN-A

January 2014

This document is a guide to the programming functions of TCPware for OpenVMS.

Revision/Update: This is a revised manual.

Operating System/Version: VAX/VMS V5.5-2 or later, OpenVMS VAX V6.0 or later, OpenVMS Alpha V6.1 or later, or OpenVMS I64 V8.2 or later

Software Version: 6.0

Process Software
Framingham, Massachusetts
USA

The material in this document is for informational purposes only and is subject to change without notice. It should not be construed as a commitment by Process Software. Process Software assumes no responsibility for any errors that may appear in this document.

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

The following third-party software may be included with your product and will be subject to the software license agreement.

Network Time Protocol (NTP). Copyright © 1992 by David L. Mills. The University of Delaware makes no representations about the suitability of this software for any purpose.

Point-to-Point Protocol. Copyright © 1989 by Carnegie-Mellon University. All rights reserved. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by Carnegie Mellon University. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED ``AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

RES_RANDOM.C. Copyright © 1997 by Niels Provos <provos@physnet.uni-hamburg.de> All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by Niels Provos.
4. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

Copyright © 1990 by John Robert LoVerso. All rights reserved. Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by John Robert LoVerso.

Kerberos. Copyright © 1989, DES.C and PCBC_ENCRYPT.C Copyright © 1985, 1986, 1987, 1988 by Massachusetts Institute of Technology. Export of this software from the United States of America is assumed to require a specific license from the United States Government. It is the responsibility of any person or organization contemplating export to obtain such a license before exporting. WITHIN THAT CONSTRAINT, permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

DNSSIGNER (from BIND distribution) Portions Copyright (c) 1995-1998 by Trusted Information Systems, Inc.
Portions Copyright (c) 1998-1999 Network Associates, Inc.
Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies. THE SOFTWARE IS PROVIDED "AS IS" AND TRUSTED INFORMATION SYSTEMS DISCLAIMS
ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL TRUSTED INFORMATION SYSTEMS BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

ERRWARN.C. Copyright © 1995 by RadioMail Corporation. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of RadioMail Corporation, the Internet Software Consortium nor the names of its contributors may be used

judgement as to where in this file the bootstrap license should appear.
4. The acronym "ISC", when used in this license or generally in the context of works covered by this license, is an abbreviation for the words "Internet Software Consortium."
5. A distribution, as referred to hereafter, is any file, collection of printed text, CD ROM, boxed set, or other collection, physical or electronic, which can be distributed as a single object and which contains one or more works covered by this license.
6. You may make distributions containing covered files and provide copies of such distributions to whomever you choose, with or without charge, as long as you obey the other terms of this license. Except as stated in (9), you may include as many or as few covered files as you choose in such distributions.
7. When making copies of covered works to distribute to others, you must not remove or alter the Bootstrap License. You may not place your own copyright message, license, or similar statements in the file prior to the original copyright message or anywhere within the Bootstrap License. Object files and executable files are exempt from the restrictions specified in this clause.
8. If the version of a covered source file as you received it, when compiled, would normally produce executable code that would print a copyright message followed by a message referring to an ISC web page or other ISC documentation, you may not modify the file in such a way that, when compiled, it no longer produces executable code to print such a message.
9. Any source file covered by this license will specify within the Bootstrap License the name of the ISC distribution from which it came, as well as a list of associated documentation files. The associated documentation for a binary file is the same as the associated documentation for the source file or files from which it was derived. Associated documentation files contain human-readable documentation which the ISC intends to accompany any distribution.

If you produce a distribution, then for every covered file in that distribution, you must include all of the associated documentation files for that file. You need only include one copy of each such documentation file in such distributions.

Absence of required documentation files from a distribution you receive or absence of the list of documentation files from a source file covered by this license does not excuse you from this from this requirement. If the distribution you receive does not contain these files, you must obtain them from the ISC and include them in any redistribution of any work covered by this license. For information on how to obtain required documentation not included with your distribution, see: http://www.isc.org.

If the list of documentation files was removed from your copy of a covered work, you must obtain such a list from the ISC. The web page at http://www.isc.org contains pointers to lists of files for each ISC distribution covered by this license.

It is permissible in a source or binary distribution containing covered works to include reformatted versions of the documentation files. It is also permissible to add to or modify the documentation files, as long as the formatting is similar in legibility, readability, font, and font size to other documentation in the derived product, as long as any sections labeled CONTRIBUTIONS in these files are unchanged except with respect to formatting, as long as the order in which the CONTRIBUTIONS section appears in these files is not changed, and as long as the manual page which describes how to contribute to the Internet Software Consortium (hereafter referred to as the Contributions Manual Page) is unchanged except with respect to formatting.

Documentation that has been translated into another natural language may be included in place of or in addition to the required documentation, so long as the CONTRIBUTIONS section and the Contributions Manual Page are either left in their original language or translated into the new language with such care and diligence as is required to preserve the original meaning.

10. You must include this license with any distribution that you make, in such a way that it is clearly associated with such covered works as are present in that distribution. In any electronic distribution, the license must be in a file called "ISC-LICENSE".

If you make a distribution that contains works from more than one ISC distribution, you may either include a copy of the ISC-LICENSE file that accompanied each such ISC distribution in such a way that works covered by each license are all clearly grouped with that license, or you may include the single copy of the ISC-LICENSE that has the highest version number of all the ISC-LICENSE files included with such distributions, in which case all covered works will be covered by that single license file. The version number of a license appears at the top of the file containing the text of that license, or if in printed form, at the top of the first page of that license.

11. If the list of associated documentation is in a seperated file, you must include that file with any distribution you make, in such a way that the relationship between that file and the files that refer to it is clear. It is not permissible to merge such files in the event that you make a distribution including files from more than one ISC distribution, unless all the Bootstrap Licenses refer to files for their lists of associated documentation, and those references all list the same filename.

12. If a distribution that includes covered works includes a mechanism for automatically installing covered works, following that installation process must not cause the person following that process to violate this license, knowingly or unknowingly. In the event that the producer of a distribution containing covered files accidentally or wilfully violates this clause, persons other than the producer of such a distribution shall not be held liable for such violations, but are not otherwise excused from any requirement of this license.

13. COVERED WORKS ARE PROVIDED "AS IS". ISC DISCLAIMS ALL WARRANTIES WITH REGARD TO COVERED WORKS INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

14. IN NO EVENT SHALL ISC BE LIABLE FOR ANY SPECIAL, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OF COVERED WORKS.

Use of covered works under different terms is prohibited unless you have first obtained a license from ISC granting use pursuant to different terms. Such terms may be negotiated by contacting ISC as follows:

Internet Software Consortium
950 Charter Street
Redwood City, CA 94063
Tel: 1-888-868-1001 (toll free in U.S.)
Tel: 1-650-779-7091
Fax: 1-650-779-7055
Email: info@isc.org
Email: licensing@isc.org

DNSSAFE LICENSE TERMS
This BIND software includes the DNSsafe software from RSA Data Security, Inc., which is copyrighted software that can only be distributed under the terms of this license agreement.

The DNSsafe software cannot be used or distributed separately from the BIND software. You only have the right to use it or distribute it as a bundled, integrated product.

The DNSsafe software can ONLY be used to provide authentication for resource records in the Domain Name System, as specified in RFC 2065 and successors. You cannot modify the BIND software to use the
DNSsafe software for other purposes, or to make its cryptographic functions available to end-users for other uses.

If you modify the DNSsafe software itself, you cannot modify its documented API, and you must grant RSA Data Security the right to use, modify, and distribute your modifications, including the right to use
any patents or other intellectual property that your modifications depend upon.

You must not remove, alter, or destroy any of RSA's copyright notices or license information. When distributing the software to the Federal Government, it must be licensed to them as "commercial computer software" protected under 48 CFR 12.212 of the FAR, or 48 CFR 227.7202.1 of the DFARS.

You must not violate United States export control laws by distributing the DNSsafe software or information about it, when such distribution is prohibited by law.

THE DNSSAFE SOFTWARE IS PROVIDED "AS IS" WITHOUT ANY WARRANTY WHATSOEVER. RSA HAS NO OBLIGATION TO SUPPORT, CORRECT, UPDATE OR MAINTAIN THE RSA SOFTWARE. RSA DISCLAIMS ALL WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO ANY MATTER WHATSOEVER, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS.
If you desire to use DNSsafe in ways that these terms do not permit, please contact:
RSA Data Security, Inc.
100 Marine Parkway
Redwood City, California 94065, USA
to discuss alternate licensing arrangements.

Secure Shell (SSH). Copyright © 2000. This License agreement, including the Exhibits ("Agreement"), effective as of the latter date of execution ("Effective Date"), is hereby made by and between Data Fellows, Inc., a California corporation, having principal offices at 675 N. First Street, 8th floor, San Jose, CA 95112170 ("Data Fellows") and Process Software, Inc., a Massachusetts corporation, having a place of business at 959 Concord Street, Framingham, MA 01701 ("OEM").

Portions copyright 1988 - 1994 Epilogue Technology Corporation.

Copyright (c) 1998-2007 The OpenSSL Project. All rights reserved.

Contents

Preface

Introducing This Guide

This guide describes the TCPware services and libraries from the programmer's perspective. It is for network application programmers.

What You Need to Know Beforehand

Before using TCPware, you should be familiar with:

- Computer networks in general.
- HP's OpenVMS operating system and file system.

How This Guide Is Organized

This guide has the following contents:

- PART I, *Introduction to Programming*—Introduces network programming and the TCPware programming functions.
- PART II, *UCX Compatibility Programming*—Describes TCPware's support for the BGDRIVER and VAXCRTL socket routines.
- PART III, *QIO Programming*—Describes the TCPDRIVER, UDPDRIVER, IPDRIVER, and INETDRIVER QIO programming interfaces.
- PART IV, *Programming Libraries*—Describes the FTP Library, Socket Library, and TELNET Library routines.
- PART V, *Application Programming Interfaces*—Describes the Simple Network Management Protocol (SNMP) extendible agent application programming interface (API) routines, and the Token Authentication API functions.
- PART VI, *ONC RPC Programming*—Describes the Remote Procedure Call (RPC) routines.
- PART VII, *Appendix*—Reference for programming the Socket Library routines for versions of OpenVMS earlier than 5.3.
- Index to this guide.

Online Help

You can use help at the DCL prompt to find the following:

- Topical help—Access TCPware help topics as follows:

```
$ HELP TCPWARE [topic]
```

The topic entry is optional. You can also enter topics and subtopics at the following prompt and its subprompts:

```
TCPWARE Subtopic?
```

Online help is also available from within certain TCPware components: FTP-OpenVMS Client and Server, Network Control Utility (NETCU), TELNET-OpenVMS Client, NSLOOKUP, and TRACEROUTE. Use the HELP command from within each component.

Example: `NETCU>HELP [topic]`

- Error messages help – Access help for TCPware error messages only as follows:

```
$ HELP TCPWARE MESSAGES
```

If the error message is included in the MESSAGES help, it identifies the TCPware component and provides a meaning and user action. See the `Instructions` under `MESSAGES`.

Obtaining Customer Support

You can use the following customer support services for information and help about TCPware and other Process Software products if you subscribe to our Product Support Services. (If you bought TCPware products through an authorized TCPware reseller, contact your reseller for technical support.) Contact Technical Support directly using the following methods:

- **Electronic Mail**

  E-mail relays your question to us quickly and allows us to respond, as soon as we have information for you. Send e-mail to **support@process.com**.Be sure to include your:

  – Name
  – Telephone number
  – Company name
  – Process Software product name and version number
  – Operating system name and version number

  Describe the problem in as much detail as possible. You should receive an immediate automated response telling you that your call was logged.

- **Telephone**

  If calling within the continental United States or Canada, call Process Software Technical Support toll-free at 1-800-394-8700. If calling from outside the continental United States or Canada, dial 1-508-628-5074. Please be ready to provide your name, company name, and telephone number.

- **World Wide Web**

  There is a variety of useful technical information available on our World Wide Web home page, **http://www.process.com** (select **Customer Support**).

- **Internet Newsgroup**

  You can also access the VMSnet newsgroup, **vmsnet.networks.tcp-ip.tcpware**. Process Software's Engineering and Technical Support professionals monitor and respond to this open forum newsgroup on a timely basis.

License Information

TCPware for OpenVMS includes a software license that entitles you to install and use it on one machine. Please read and understand the *Software License Agreement* before installing the product. If you want to use TCPware on more than one machine, you need to purchase additional licenses. Contact Process Software or your distributor for details.

Maintenance Services

Process Software offers a variety of software maintenance and support services. Contact us or your distributor for details about these services.

Reader's Comments Page

TCPware guides may include Reader's Comments as their last page. If you find an error in this guide or have any other comments about it, please let us know. Return a completed copy of the Reader's Comments page, or send e-mail to **techpubs@process.com**.

Please make your comments specific, including page references whenever possible. We would appreciate your comments about our documentation.

Documentation Set

The documentation set for TCPware for OpenVMS consists of the following:

- ***Release Notes*** for the current version of TCPware for OpenVMS—For all users, system mangers, and application programmers. The *Release Notes* are available online on your TCPware for OpenVMS media and are accessible before or after software installation.
- ***Installation & Configuration Guide***—For system managers and those installing the software. The guide provides installation and configuration instructions for the TCPware for OpenVMS products.
- ***User's Guide***—For all users. This guide includes an introduction to TCPware for OpenVMS products as well as a reference for the user functions arranged alphabetically by product, utility, or service.
- ***Management Guide***—For system managers. This guide contains information on functions not normally available to the general network end user. It also includes implementation notes and troubleshooting information.
- ***Network Control Utility (NETCU) Command Reference***—For users and system managers. This reference covers all the commands available with the Network Control Utility (NETCU) and contains troubleshooting information.
- ***Programmer's Guide***—For network application programmers. This guide gives application programmers information on the callable interfaces between TCPware for OpenVMS and application programs.
- **Online help**—
  - Topical help, using **HELP TCPWARE** *[**topic**]*
  - Error messages help, using **HELP TCPWARE MESSAGES**

Conventions Used

Convention	Meaning
host	Any computer system on the network. The local host is your computer. A remote host is any other computer.
`monospaced type`	System output or user input. User input is in **`bold type`**. Example: `Is this configuration correct?` **`YES`** Monospaced type also indicates user input where the case of the entry should be preserved.
italic type	Variable value in commands and examples. For example, *username* indicates that you must substitute your actual username. Italic text also identifies documentation references.
[*directory*]	Directory name in an OpenVMS file specification. Include the brackets in the specification.
[optional-text]	(Italicized text and square brackets) Enclosed information is optional. Do not include the brackets when entering the information. Example: **`START/IP line`** ***`address [info]`*** This command indicates that the *info* parameter is optional.
{*value* | *value*}	Denotes that you should use only one of the given values. Do not include the braces or vertical bars when entering the value.
Note!	Information that follows is particularly noteworthy.
CAUTION!	Information that follows is critical in preventing a system interruption or security breach.
`key`	Press the specified key on your keyboard.
`Ctrl/key`	Press the control key and the other specified key simultaneously.
`Return`	Press the Return or Enter key on your keyboard.

Chapter 1 Network Programming Overview

Introduction

This chapter introduces TCP/IP network programming. It describes TCP/IP programming generally and outlines areas where TCPware provides added functions or has specific requirements. This chapter includes sample client and server programs and describes the following:

- TCP/IP programming concepts
- Data representation and exchange
- Programming services options
- Network programming with sockets
- Multicasting
- Sample application programs

For details on TCP/IP network programming, see the list of reference texts in the *User's Guide* and the following books:

- Comer, Douglas E. & David L. Stevens [1993], *Internetworking with TCP/IP, Volume III: Client-Server Programming and Applications (BSD Socket Version)*, Prentice-Hall
- Stevens, W. Richard [1990], *UNIX Network Programming*, Prentice-Hall

This chapter is designed for an audience of experienced programmers who need information specific to TCPware programming.

TCP/IP Programming Concepts

TCP/IP programming requires determining the following:

- Whether to use a connection-oriented (TCP) or connectionless (UDP) networking service
- Creating sockets
- Naming the communication endpoints (internet addresses and port numbers)

Connection-Oriented Services and TCP

Connection-oriented services and protocols support applications that send multiple messages between peer applications. These services and protocols require that applications establish a logical connection (*virtual circuit*) between them before they can exchange data. They provide data transfer that is reliable, ordered, full duplex, and flow controlled. They:

- Verify receipt of data.
- Compute checksums.
- Provide sequence numbers to ensure correct segment order.

- Retransmit lost segments.
- Inform users of dropped network connections. TCP is such a connection-oriented protocol, designed for applications that need reliable data delivery between similar and dissimilar systems. Examples of applications that use TCP are:
  - FTP
  - TELNET
  - SMTP

Since these services and protocols are more complex than connectionless services, they have higher overhead. In a connection-oriented service:

- The source and destination uniquely identify each connection.
- You can multiplex connections across the network to paired host processes.
- Data streams break into portions encapsulated with control information, such as addresses. Encapsulated pieces pass through the network to the peer host.

With TCP, the number of blocks and size (in number of bytes) of a send operation need not equal that of a receive operation. The protocol may bundle several sends into one receive: a receive operation can receive more or fewer bytes than in a single send operation.

For example, if you do five send operations, you may not need five receive operations to get all the data. If a client application sends five 10-byte blocks of data (five send operations) over the network and the server program initiates a 100-byte read operation, the program can receive all 50 bytes at one time.

TCP (unlike DECnet) does not support the concept of messages. TCP is a byte-stream protocol that does not distinguish record or read/write boundaries. The individual applications have to perform messaging as needed. The client and server have to agree on a message protocol and implement it. Here are three possible strategies:

Messages of fixed length	Data acquisition applications are examples of this type of strategy. When you use this strategy, make sure the application reads a complete message-length of data from the network. After the read, the application must go back and process any residual data left over from the last complete message. For example, if you issue a read for 100 bytes and do not get 100 bytes, you need to update the pointers and the length, and go back until you read the full 100 bytes.
Messages preceded by byte count length	Other protocols prefix each message with a byte count. This could be a 16- or 32-bit quantity. RPC is an example of an application protocol that uses this strategy. If you use this technique, use the conversion routines for native- and network-byte order described in the *Native Byte Order and Network Byte Order*. This guarantees that the client and server agree on the byte order sequences for that number.
Messages separated by a <CR><LF> sequence	FTP is an application that uses this type of messaging strategy. A program receiving data searches for <CR><LF> characters and processes the data preceding these characters as one record. If there is more data in the buffer, the program looks for the next <CR><LF> sequence and processes the data preceding it as a separate message.

Connectionless Services and UDP

Connectionless services present data with a destination address, and the network delivers it on a best-effort basis. This is independent of other data exchanged between the same pair of hosts. Connectionless services are unreliable but have lower overhead than connection-oriented services. In a connectionless service:

- The client or server must perform data tracking and adaptive retransmission strategies.
- Applications cannot depend on the underlying transport for reliable delivery.
- Each message or portion of a message contains all delivery data.
- Operation is best in LANs since WANs can introduce more errors.

UDP is a full-duplex, connectionless datagram delivery protocol with low overhead. UDP is an excellent choice for applications that need the highest performance and can tolerate this level of service. Examples of applications that use UDP are:

- NFS
- DNS
- SNMP

UDP can lose, delay, or duplicate requests, or deliver them out of order. The application requesting the service must detect and correct transmission errors. For example, an application may retransmit a request if it does not receive a reply. Applications typically do this by enabling a short timeout period and retransmitting the request if the application does not get a response within the timeout period.

Under UDP, a server could receive multiple requests since the reply might simply be lost. This case can cause problems if some operations are not repeatable. Use UDP on LANs where transmission errors are less frequent and round-trip times more predictable. UDP is also well-suited for broadcast and multicast applications, and applications that cannot tolerate the overhead of virtual circuits.

NFS often uses UDP. Here is an example of what could happen during an NFS file deletion request:

1 The client issues a delete request.

2 The server receives and processes the request.

3 The server deletes the file and returns a reply that it deleted the file.

4 The reply gets lost.

5 Not having received a confirmation, the client reissues the original delete request.

6 The server gets the request and replies that the file no longer exists.

7 The server's reply confuses the client.

Note that you might need to add application design features if some operations are not repeatable. Remote Procedure Calls (RPCs) that NFS uses handle this situation by keeping a cache of recent requests and replies. In this case, if NFS receives a duplicate request, it sends the cached reply that it successfully deleted the file.

For DNS and SNMP, these issues are not as severe. For example, DNS resolves host name and internet addresses. It does not matter how many times you reissue a request; the response is the same and you get the information you need.

Socket Concepts

Most TCP/IP programming uses a Berkeley System Distribution (BSD) UNIX abstraction called *sockets*. Network programs use sockets to exchange data across the network. Socket programming is synchronous: each socket deals with only one connection at a time.

Socket programming requires a socket descriptor that is analogous to the file descriptor used in file programming. In socket programming (as in file programming), you create the socket descriptor, use it to read or write, then destroy the descriptor by closing the connection. The operation of creating a socket descriptor involves naming the communication endpoints.

Naming Communication Endpoints

Naming communication endpoints involves assigning:

Internet addresses	Internet addresses uniquely identify the source and destination host interfaces. Any exchange of information involves two addresses, one for the source and one for the destination.
Port numbers	Port numbers uniquely identify a source and destination port. Similar to DECnet object numbers, port numbers identify the particular application or service used. The TCP protocol specifies sockets that have a protocol port number and an IP address. This protocol uses the AF_INET address family type. TCP/IP programming uses well-known port numbers to contact a known service. For example, to do a file transfer you need to open a connection to port 21. Port 21 is the FTP server port number. Other services have specific port numbers as defined by the current *Assigned Numbers* RFC. If you write an application, you need to assign a port number. Any client that wants to use the service you create connects to that port number.

The *Assigned Numbers* RFC also designates a clearinghouse for assigning port numbers. Get a unique port number from this center when you develop a network service. However, if you are developing a private application, you can create your own port number so long as another service does not use it.

Sockets define endpoint addresses in data structures coded in C. Most application programs use predefined address structures, such as `sockaddr_in`, as defined in the *BSD Socket Data Structures*.

Data Representation and Exchange

This section discusses concepts about how data is formatted and exchanged between systems: data encoding schemes and native as opposed to network byte order.

Data Encoding Schemes

Different hardware and operating system platforms represent data with different encoding schemes. For example, representing floating point and integer values differ on various hardware platforms.

Your programs must decide on a compatible encoding scheme and perform any required conversion between network format and local hardware representation format. Another approach is to exchange data in ASCII instead of binary representation.

Native Byte Order and Network Byte Order

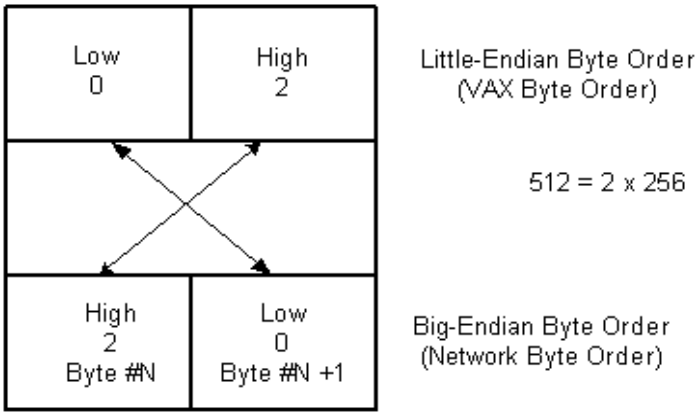
The sequence for storing binary data on a given machine is *native byte order*. The sequence for transmitting binary data over the network is *network byte order*.

Some machines are little-endian; others are big-endian. Little-endian machines (VAX systems, for example) store binary data with the least significant byte first.

Big-endian machines (Sun systems, for example) store and transmit binary data with the most significant byte first. Network byte order is always in the big-endian format. Socket libraries require network addresses in network byte order.

Figure 1-1 shows the format for storing the decimal value 512 as a 16-bit (2 byte) quantity on little-endian and big-endian systems.

Figure 1-1 Big- and Little-Endian Storage of 512 as a Word



When communicating between a client and server, both need to agree to the byte sequence for transmitting data. Depending on what type of system you are on, the host's native byte order may not be the same as network byte order. For example, the byte order for VAX systems is little-endian, the opposite of network byte order.

Several routines convert between network and native byte order. Use these routines to make sure that the data is in the right format. Always use these conversion routines, even if the local host byte order is the same as network byte order. Doing so guarantees that the order of information is correct and you can port the source code.

The function `htons` converts a short integer (16 bits) from host native to network byte order. The function `ntohs` converts a short integer from network to host native byte order. The `htonl` and `ntohl` functions convert long integers (32 bits) between the two.

For example, if you use a messaging strategy that uses a byte count length (as described in the "Messages preceded by byte count length" bulleted item), use these routines to encode the byte count length field. This way the order of information is correct, whether you are communicating between a VAX and Sun system or between two VAX systems.

Programming Services Options

This section describes the interface and services options you can use when writing TCP/IP networking applications. It includes information on the following:

- Device drivers
- VAX C and DEC C socket libraries and UCX Compatibility
- TCPware Socket Library routines
- System Queue Input/Output (QIO) interface calls
- FTP Library routines
- TELNET Library routines
- ONC RPC Services

Use the DEC C or VAX C socket libraries with C-based applications. Note that any high-level language that passes C-like arguments can call these routines.

Programming with sockets (rather than using the QIO interface discussed in the *System Queue Input/Output (QIO) Calls* subsection) makes network programs more portable across UNIX environments and more compatible with other socket-based TCP/IP applications. Sockets are easy to program since they look like standard subroutine calls and hide some of the complexity of the QIO interface system calls.

However, writing event-driven programs is much easier with the QIO interface than with sockets. This is true since sockets do not fit the standard OpenVMS event-driven IO model. The QIO interface provides access to the full range of TCPware functions; the Socket Library provides access to a subset of the available TCPware functions.

Device Drivers

TCPware uses the standard OpenVMS network interface device drivers that operate the hardware controller. These device drivers include DEC's Ethernet device drivers and third-party device drivers (PNDRIVER for Proteon's proNET controller; and NADRIVER, NBDRIVER, and NCDRIVER for the HYPERchannel driver).

Because TCP-OpenVMS uses these standard OpenVMS device drivers, other applications using these drivers (such as DECnet, but not other TCP/IP implementations) can continue to run at the same time and use the same hardware.

VAX C and DEC C Socket Library and UCX Compatibility Services

The VAX C and DEC C socket routines are the preferred methods of network programming in that they offer the greatest flexibility. These socket routines are available because of the TCPware UCX Compatibility Services.

See HP's *VAX C Run Time Library Manual* or *DEC C Language Reference Manual* for information on these socket library routines.

The TCPware UCX Compatibility Services provides the QIO interface that the HP TCP/IP Services for OpenVMS BGDRIVER support. The UCX Compatibility Services provide support for:

- Any application the UCX BGDRIVER supports.
- VMS 5.3 (or later) VAX C Run-Time Library (VAXCRTL) Socket Routines.
- OpenVMS VAX, OpenVMS Alpha and OpenVMS I64 DEC C Run-Time Library (DEC/CRTL) Socket Routines.

If you developed an application for UCX, there is no need to modify it to make it work with TCPware. You can take your image (compiled and linked against UCX) and run it as is.

See the *HP TCP/IP Services for VMS Programming Manual* for programming information. Then see Chapter 2, *UCX Compatibility Services*, for programming information specific to TCPware's UCX compatibility.

TCPware Socket Library

Note! The TCPware Socket Library is intended for use on VAX systems running pre-Version 5.3 VMS, or if you are using Open Network Computing (ONC) Remote Procedure Call (RPC).Services. If you are running VMS Version 5.3 or later, use the VAX C or DEC C socket routines, discussed in the previous section.

The TCPware Socket Library is a collection of C subroutines that closely emulate the UNIX socket functions. The Socket Library supports a subset of the UNIX socket functions, including stream and datagram sockets. These subroutines let you migrate UNIX C programs using the UNIX socket functions to the TCPware environment with versions of VMS earlier than 5.3.

The Socket Library routines use the TCPDRIVER and UDPDRIVER programming interfaces, although an alternate set of socket routines is available that uses the INETDRIVER programming interface. This INETDRIVER interface allows a mix of socket routines and INETDRIVER QIO calls. This alternate set of socket routines is available in the TCPWARE_SOCKLIB.OLB socket library and is not part of the

TCPWARE_SOCKLIB_SHR.EXE shareable run-time library (RTL).

The SOCKLIB RTL or OLB libraries support VAX C or DEC C for older versions of VMS. Some of the routines are renamed to avoid conflict with other library routine names but perform the same functions. For example, `socket_close` is analogous to the UNIX `close` function, and `socket_read` and `socket_write` are analogous to the UNIX `read` and `write` functions.

The TCPware Socket Library uses socket descriptors that are addresses of internal data structures, not socket numbers. The socket descriptor space is not compatible with the file descriptor space, so standard VAX C or DEC C routines will not function with TCPware socket descriptors. Another implication of this is that the TCPware `select` call uses a list of socket descriptors instead of a bit mask. Use the TCPware FD macros to manipulate socket descriptor lists since these macros manage this difference transparently.

See Appendix A, *TCPware Socket Library*, for programming information.

System Queue Input/Output (QIO) Calls

Programmers can create applications that use the system Queue Input/Output calls (QIOs) for an interface. These calls use standard OpenVMS system services and support any high-level programming language. QIOs support all OpenVMS asynchronous features such as Asynchronous System Traps (ASTs) and event flags.

BGDRIVER, TCPDRIVER, UDPDRIVER, IPDRIVER, and INETDRIVER are TCPware's proprietary QIO programming interfaces, as discussed in the following subsections.

BGDRIVER

TCPware provides BGDRIVER . The BGDRIVER is the preferred programming interface.

See Chapter 2, *UCX Compatibility Services*.

TCPDRIVER, UDPDRIVER, and IPDRIVER

These drivers are proprietary to Process Software and are described as follows:

IPDRIVER implements IP and ICMP	It uses the network device drivers to send and receive datagrams. The Address Resolution Protocol (ARP) and Reverse ARP (RARP), which map internet addresses and physical addresses, are also implemented within this driver. Functions are provided to open and close a port and to transmit and receive datagrams. (See the *IPDRIVER Services* chapter for programming information.) IPDRIVER uses ports to demultiplex received datagrams. When an IP datagram is received, IPDRIVER validates the header and searches for a port opened on the protocol number in the datagram's internet header. IPDRIVER discards the datagram if no port is open for that protocol or if that port has no outstanding receive.
TCPDRIVER implements TCP	It uses the IP device driver to send and receive TCP segments. Functions are provided to open and close, receive and send data over, and perform special control functions on a connection. (See the *TCPDRIVER Services* chapter in Part 3 for programming information.)

UDPDRIVER implements UDP	It uses the IP device driver to send and receive UDP datagrams. Functions are provided to open and close a receive port, and to receive and send data. (See the *UDPDRIVER Services* chapter for programming information.)

INETDRIVER

TCPware provides INETDRIVER, the Stanford Research Institute (SRI) QIO interface. Some vendors of TCP/IP products (such as Process Software's MultiNet) use the SRI QIO interface as a direct socket type interface. The INET device driver maps UNIX socket calls to OpenVMS QIO requests. It supports stream (TCP) and datagram (UDP) services.

INETDRIVER Services provide an asynchronous I/O implementation of the UNIX socket calls within the OpenVMS $QIO and $QIOW system services. These system services allow Asynchronous System Trap (AST) routines and event flags to be associated with I/O requests. This allows for efficient socket operations.

The INETDRIVER interfaces directly with the TCP and UDP protocols in the transport layers. It does not replace the TCPDRIVER or UDPDRIVER services, but provides another way to communicate with them.

See Chapter 6, *INETDRIVER Services*, for programming information.

FTP Library Routines

The FTP-OpenVMS library routines provide a programming interface to the FTP protocol. Network programmers use the FTP-OpenVMS library routines in applications to provide FTP capabilities.

See Chapter 7, *FTP Library*, for programming information.

TELNET Library Routines

The TELNET-OpenVMS library routines provide a programming interface to the TELNET protocol. Network programmers use the TELNET-OpenVMS library routines in applications to provide TELNET capabilities.

See Chapter 9, *TELNET Library*, for programming information.

ONC RPC Services

TCPware provides Open Network Computing (ONC) Remote Procedure Call (RPC) Services. ONC RPC Services are a set of software programming tools with which you can develop distributed applications. These tools implement the RPC and XDR (External Data Representation) protocols.

A distributed application executes different parts of its programs on different hosts in a network. Computers on the network share the processing workload, with each computer performing the tasks for which it is best equipped. For example, a distributed database application might consist of a central database running on a server and numerous client workstations. The workstations send requests to the server. The server carries out the requests and sends the results back to the workstations. The workstations use the results in other modules of the application.

Remote procedure calls (RPCs) allow programs to invoke procedures on remote hosts as if the procedures were local. ONC RPC Services hide the networking details from the application. This facilitates distributed processing because it relieves the application programmer from performing low-level network tasks such as establishing connections, addressing sockets, and converting data from one machine's format to another.

The XDR protocol provides a means for the local and remote host to agree on a way of representing data. XDR is a standard that resolves differences of data representation between different operating systems and hardware architectures.

ONC RPC Services consist of the following components:

- Shareable run-time libraries (RTLs)
- RPCGEN compiler
- Port Mapper
- RPCINFO command

The Port Mapper maps well-known RPC program numbers to UDP/TCP port numbers. The Port Mapper helps ONC RPC client programs connect to ports the ONC RPC server uses. A Port Mapper runs on each host that implements ONC RPC Services. The Port Mapper is part of TCPware's Network Control Process (NETCP).

Note! You must use the TCPware Socket Library if you are using ONC RPC Services.

Part VI in this guide is devoted to ONC RPC programming.

Network Programming with Sockets

This section provides a quick overview of socket programming. It includes information on the following:

- Using socket calls in network programming
- Socket system calls
- Messaging over stream (TCP) connections
- BSD socket data structures

Using Socket Calls in Network Programming

Figure 1-2 is an example of calls made by a client and server that use TCP to communicate. In this example, the server starts and waits for new connections on a well-known port. It accepts each new connection, processes the client's requests, and closes the connection.

In the example, the client creates a socket and uses `connect` to connect to the server. The client then uses `write` to send requests to the server and `read` to receive replies from the server. The client calls the `close` routine when it is finished using the connection.

The server uses a `socket` call to create a socket, then uses `bind` to specify the local well-known port for the application. Next, the server calls `listen` (which prepares the socket for incoming connections) and enters a loop. In the loop, the server calls the `accept` routine and waits for the next connection request to arrive. The server uses `read` and `write` to interact with and `close` to end communication with the client. The server then returns to `accept` and waits for the next connection request.

Servers typically use passive sockets to wait for incoming network connections. Applications (typically clients) use active sockets to initiate a connection.

Socket System Calls

When creating a socket, you create either a stream socket or a datagram socket. Stream sockets (created by specifying SOCK_STREAM) correspond to the TCP protocol. Datagram sockets (created by specifying SOCK_DGRAM) correspond to the UDP protocol.

SOCK_RAW corresponds to the IP layer directly. Note that SOCK_RAW is not described in this chapter because programming is not usually done at this level. Using TCP or UDP is far superior.

The IP layer only allows you to have 256 connections or users at the same time. TCP and UDP allow you to have about 2^{32} possible connections between any two hosts. A connection is uniquely identified by the source internet address and port number, and the destination internet address and port number.

Many operating systems provide the BSD socket call interface, including OpenVMS v5.3 and later. Most non-socket implementations have comparable networking services. In OpenVMS, most implementations include a BSD socket library interface as well as a QIO interface. The QIO interface can be almost a one-to-one correspondence to the socket calls, and usually provides a greater range of functions than available through the socket library. While the QIO interface can be a bit different from the BSD interface, it still provides the same level of services.

Figure 1-2 Sequence of Socket Calls Between Client and Server

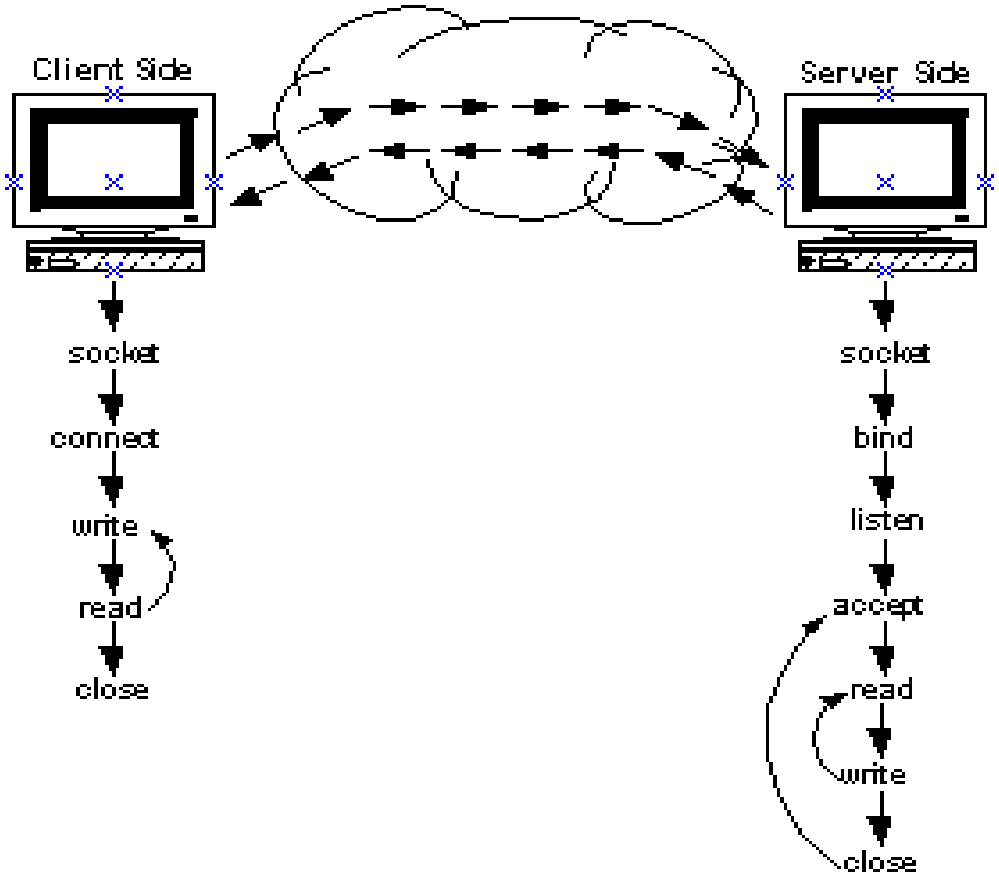


Table 1-1 describes typically-used socket calls.

Table 1-1 Socket Calls

Call	Description
accept	Accepts an incoming connection. `Accept` actually blocks and waits for the connection to come in. Typically in the case of a server, it sits in block mode until someone actually does a `connect`. Valid only for stream TCP/IP sockets.
bind	Used primarily by servers to name their local endpoint of the connection. Servers are contacted at well-known port numbers: they "bind" to that port number. In a bind operation, you specify the internet address as well as the port number. Most servers specify the internet address as zero (or INADDR_ANY). The servers do not want to bind to any particular internet address, just to a particular port. If a host has multiple internet addresses, it does not matter where the request comes from so long as it goes to the specified port.
close	Closes communication over a socket. `Close` is similar to closing a file. `Close` makes sure that the data has been sent over to the peer and then does the actual close operation. For the Socket Library, `close` is called `socket_close`.
connect	Names the remote endpoint and, in the case of TCP, establishes a connection. When you specify `connect`, you also specify an internet address and port number. This connects you actively to the remote peer, provided someone there is willing to accept the connection. For example, when an FTP client issues an open, it does a connect operation on the specified internet address and port 21. This establishes the connection to the remote server.
getpeername getsockname	Obtain the names of the communication endpoints. These calls return the internet address and port number of either the local end of the connection or the remote end of the connection. The local end of the connection is the port number and internet address on which you have done a bind operation. The remote endpoint is the system to which you connected or that connected to you.
getsockopt setsockopt	Obtain or set socket options. For example, you can set one option that determines if KEEPALIVE operations are done on TCP connections.
listen	Used by servers and prepares the socket for incoming connections. `Listen` declares that the server is willing to accept connections on this socket. Any connection going to that bound port is eligible to be processed by that process.

read recv recvfrom	`Read, recv,` and `recvfrom` are different forms of the same request. Use these calls to read data. `Recvfrom` is primarily used by UDP and returns the internet address and port number of the sender of the datagram. Typically, you use this information to send back a reply in a subsequent `sendto`. For the Socket Library, read is called `socket_read` and `recv` is called `socket_recv`.
select	Performs asynchronous I/Os. The `select` call allows you to service many connections from within a particular process. There is also a timeout parameter with the `select` call that limits the amount of time you can wait before it comes back to you. In this way, `select` allows you to wait indefinitely, wait until timeout, or poll on a specific socket or set of sockets.
socket	Comparable to opening or creating a file. Creates a socket to which you can send I/O requests.
write send sendto	`Write, send`, and `sendto` are different forms of the same request. Use these calls to send data out over the network. `Sendto` is used primarily by UDP and allows you to specify the destination internet address and the port number. When using UDP, the `sendto` request allows you to send requests to any number of hosts. For the TCPware Socket Library, `write` is called `socket_write` and `send` is called `socket_send`.

BSD Socket Data Structures

The BSD sockets support a group of data structures and subroutines for the socket interface that typically are used to communicate with the socket layer. These include the `sockaddr_in` structure, `hostent` structure, and `servent` structure. The following include files are used:

Include Files	**Description**
`in.h`	defines the `sockaddr_in` structure. Almost all socket operations require use of this file.
`inet.h`	defines address conversion subroutines, such as `inet_addr()`, `inet_ntoa()`, and so on.
`netdb.h`	defines network database structures including the `hostent` and `servent` structures.
`socket.h`	defines the `sockaddr` structure, SOCK_STREAM, SOCK_DGRAM, AF_INET, and other symbols used when calling the Socket Library subroutines. All socket operations require use of this file.

See Chapter 8, *Socket Library*, for details on the files and the following sections.

sockaddr_in Structure

One of the more important socket data structures is the `sockaddr_in` structure. Use the `sockaddr_in` structure in calls to name a communication endpoint. The `sockaddr_in` structure communicates the internet address in the `sin_addr` field. It also communicates the port number in the `sin_port` field.

The `in.h` include file defines the `sockaddr_in` structure as shown in Example 1-1. In this example:

1 `sin_family` is the address family (AF_INET for example)

2 `sin_port` is the port number (in network byte order)

3 `sin_addr` is the internet address (in network byte order)

4 `sin_zero` is the internet address where the remainder of the eight bytes are unused and should be set to zero

Specify both the port and address in network byte order. For example, when doing a bind operation or a connect operation, specify the `sockaddr_in` structure with the information filled in. When using `getsockname` or `getpeername`, you provide the address of the structure and the subroutine then would fill it in for you.

Example 1-1 Sockaddr_in Structure As Defined in the in.h File

```
struct  in_addr  {
      unsigned long              addr;
};

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

hostent Structure

The `gethostbyname` and `gethostbyaddr` routines use the `hostent` structure for doing host by name or host by internet address lookups. For example, if you want to do an

```
FTP>open hostname
```

the underlying socket interface does not support use of a host name. In this case, use the `gethostbyname` routine to take that name and return the corresponding internet address (among other information).

The `netdb.h` include file defines the `hostent` structure as shown in Example 1-2.

Example 1-2 Hostent Structure as Defined in the netdb.h Include File

```
struct hostent {
char       *h_name;                 /* official host name */
char       **h_aliases;             /* alias list */
int        h_addrtype;              /* host address type */
int        h_length;                /* address length */
char       **h_addr_list;           /* list of addresses */
#define  h_addr  h_addr_list[0];    /* first address */
};
```

The `hostent` structure has:

- A host name field that points to the ASCII host name.
- A pointer to a list of alias names if the host has alias names.
- A pointer to a list of internet addresses. In a more complex environment, a host might have multiple internet addresses.
- Other information, such as the address format and address family. For TCP/IP, the address is four bytes long and the address family is AF_INET (decimal value of 2).

`servent` Structure

The `servent` structure is used when looking up services by name or port. Although you can hard-code the name or port (such as port 21), you can use the following routines to obtain the service name and port:

`getservbyname`	to obtain the service port when given its name
`getservbyport`	to obtain the service name when given its port number

Use these routines to obtain port numbers and service names. Doing so makes changing port numbers easy; you simply edit the services definition file, TCPWARE:SERVICES.

The `netdb.h` include file defines the `servent` structure as shown in Example 1-3.

Example 1-3 Servent Structure as Defined in netdb.h File

```
struct servent  {
char      *s_name;          /* official service */
char      **s_aliases;      /* alias list */
int       s_port;           /* port number */
char      *s_proto;         /* protocol to use */
};
```

Multicasting

Multicasting, as specified in RFC 1112, *Host Extensions for IP Multicasting*, is the transmission of an IP datagram to a multicast host group. A multicast host group is a set of zero or more hosts identified by a single class D IP destination address. Using a multicast host group allows applications running on your host to receive multicast IP datagrams destined for that host group.

TCPware implements the highest level of conformance specified in RFC 1112 – level 2 "full support for multicasting."

The following TCPware programming interfaces support IP multicasting for network interfaces that support multicasting (such as Ethernet, FDDI, and Token Ring):

- UDPDRIVER
- IPDRIVER
- BGDRIVER (SOCK_DGRAM sockets)
- INETDRIVER (SOCK_DGRAM sockets)
- Socket Library

Sending IP Multicast Datagrams

To send a multicast datagram, specify an IP multicast address in the range 224.0.0.0 to 239.255.255.255 as the destination address in a send request. This range covers the class D IP addresses that identify multicast host groups. Note that 224.0.0.0 is unassigned and 224.0.0.1 is assigned to the permanent group of all IP hosts, including gateways, used to address all multicast hosts on the directly connected network.

Each multicast transmission is sent from a single network interface, even if the host has more than one multicast-capable interface. The system manager establishes the default interface to use for multicasting by defining the appropriate routes. If no route is defined for the multicast address, TCPware uses the default gateway's interface. If you do not specify a default gateway, TCPware uses the first available interface. An application can issue a request to explicitly set the interface to use for subsequently transmitted multicast datagrams.

When you send a multicast datagram, TCPware by default delivers a local copy of it if the multicast address is joined by one or more receivers. An application can issue a request to disable the local loopback of multicast datagrams.

TCPware sends IP multicast datagrams with a time-to-live (TTL) of 1 by default, which prevents them from being forwarded beyond a single subnetwork. An application can issue a request to specify the TTL for subsequent multicast datagrams to be set to any value from 0 to 255, in order to control the scope of the multicasts.

Receiving IP Multicast Datagrams

Before an application can receive IP multicast datagrams, it must become a member of one or more IP multicast groups. The system manager can explicitly have the host join multicast groups by issuing the NETCU ADD MULTICAST command (and the NETCU REMOVE MULTICAST command to leave a group). Or, an application can ask the host to join (or leave) a multicast group by issuing a request.

See *Multicasting Commands* in the *NETCU Command Reference* for the multicasting commands.

The memberships requested by an application do not necessarily determine which datagrams that application receives. The IP layer accepts incoming multicast datagrams if anyone claimed a membership in the destination group of the datagram. However, delivery of a multicast datagram to a particular application is based on the destination port (for UDP) or protocol type, just as with unicast and broadcast datagrams. To receive multicast datagrams sent to a particular port/protocol, you must bind to that local port, leaving the local address unspecified.

Every membership is associated with a single interface, and it is possible to join the same group on more than one interface. If you do not specify a local interface's address when joining a group, TCPware uses the default multicast interface. At present, you can have a maximum of 32 memberships per socket. TCPware drops the memberships an application adds when you close the socket or port or the application exits. However, more than one socket or port can claim membership in a particular group, and the host remains a member of that group until the last claim is dropped.

Writing Application Programs

This section includes information on the following:

- Writing a stream client
- Writing a stream server
- Writing a datagram client
- Writing a datagram server
- Writing servers

Writing a Stream Client

When writing a stream client, use the following sequence, with socket library (and system QIO equivalent) routines given:

1. Create a socket by calling the `socket` routine and requesting a `SOCK_STREAM` socket (SYS$ASSGN).
2. Open the connection to the server by calling the `connect` routine (IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP).

   You need to specify a `sockaddr_in` structure with the destination internet address and destination port number filled in where the:

   Internet address is the address of the server and can be obtained by calling `gethostbyname` or `inet_addr` (`inet_addr` converts an ASCII dotted internet address to binary representation).

   Port number is the well-known port for the server and can be obtained by calling `getservbyname`.
3. Send and receive data as needed by calling the standard `send/recv` and `read/write` routines (IO$_WRITEVBLK or IO$_READVBLK). When you use `send/recv`, you can specify flag options, some of which allow you to:

   Send urgent data, or

   Peek at incoming data at the head of the queue to determine what to do before the data is actually read.
4. Close the connection by calling the `close` routine (IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN and then SYS$DASSGN).

The programs in Stream Client Sample Programs Included with TCPware that demonstrate stream client applications are included in the TCPWARE_ROOT:[TCPWARE.EXAMPLES] directory.

Table 1-2 Stream Client Sample Programs Included with TCPware

Program	Description
DAYTIMED.C	DAYTIME client that uses the TCPware Socket Library
DISCARD.C	DISCARD client that uses the INETDRIVER
FINGER.C	FINGER client that uses the TCPDRIVER
TCPSAMPLE.FOR	DISCARD FORTRAN client that uses the TCPDRIVER
WHOIS.C	WHOIS client that use the TCPware Socket Library

Writing a Stream Server

When writing a stream server, use the following sequence, with socket library (and system QIO equivalent) routines given:

1. Create a socket by calling the `socket` routine and requesting a `SOCK_STREAM` socket (SYS$ASSGN).
2. Bind the socket to the well-known port for the service by calling the `bind` routine (IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP). Specify zero for the internet address. Use the `getservbyname` routine to determine the required port.
3. Set the socket to accept incoming connections by calling the `listen` routine, which has two parameters: the socket that you want to listen on, and a backlog parameter, a value that indicates how many connections you can have at a given time (the range is 1 to 5).

4 For example, if you have a single-threaded server that processes connections serially, the server listens, accepts a connection, and processes that connection. Meanwhile other connections can come in on the well-known port. In this case, the connection is established and placed in a queue waiting to be accepted. However, you can do nothing with the connection until you perform the next step.

5 Wait for a connection (call `accept`), where:

- `accept` returns a new socket for the connection (you do the I/O on this socket).
- The original socket is still listening for more connections.

6 Receive and send data as needed to provide the service by calling `read` or `write` and `send` or `recv` (IO$_READVBLK or IO$_WRITEVBLK).

7 Close the connection socket by calling the `close` routine (IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN).

8 Go to step 4.

The programs in Table 1-3 that demonstrate stream server applications are included in the TCPWARE_ROOT:[TCPWARE.EXAMPLES] directory.

Table 1-3 Stream Server Sample Programs Included with TCPware

Program	Description
DISCARDD.C	DISCARD server that uses the TCPDRIVER or Socket Library
FINGERD.C	FINGER server that uses the TCPDRIVER
TCPSAMPLE.FOR	DISCARD FORTRAN server that uses the TCPDRIVER

Writing a Datagram Client

Writing a datagram client consists of the same type of operations as writing a stream client. Use the following sequence, with socket library (and system QIO equivalent) routines given:

1 Create a socket by calling the `socket` routine and requesting a `SOCK_DGRAM` socket (SYS$ASSGN).

2 (Optionally) issue a `connect` call if you are only going to exchange information with a particular remote peer (IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP).

In this case, when you issue a `connect` call, the process reserves internal socket fields that identify the internet address and port number to which the data is to go. This means you can send data using standard `write` and `send` calls instead of having to use the `sendto` call to specify the destination. Note that:

- `connect` names the communication endpoint. There is no need to specify it with each send.
- No connection is opened with UDP.

3 Send a request datagram calling the `sendto` or `send` routines (use `send` only if connected). (IO$_WRITEVBLK.)

If you did not issue a `connect` call, you need to issue a `sendto` call that specifies the buffer you want to send and where to send the buffer.

4 Start a timer. For example, the program might give the server five seconds to respond. If there is no response within that time, the program reissues the request.

The program could also indicate to the user that there is a problem if, for example, the server does not respond after five attempts to reach the server.

5 Receive a reply datagram using the call `recvfrom` (or `recv` if connected). Cancel the timer if the proper reply was received; otherwise reissue the `recvfrom` or `recv`. (IO$_READVBLK.)

6 If the application needs to request more data, go to step 3.

7 Close the socket by calling the `close` routine (IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN and then SYS$DASSGN).

The programs in Table 1-4 that demonstrate datagram client applications are included in the TCPWARE_ROOT:[TCPWARE.EXAMPLES] directory.

Table 1-4 Datagram Client Sample Programs Included with TCPware

Program	Description
UDPSAMPLE.FOR	DISCARD FORTRAN client that uses the UDPDRIVER

Writing a Datagram Server

Writing a datagram server consists of the same type of operations as writing a stream server, with socket library (and system QIO equivalent) routines given:

1 Create a socket by calling the `socket` routine and requesting a `SOCK_DGRAM` socket (SYS$ASSGN).

2 Bind the socket to the well-known port for the service by calling the `bind` routine (IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP). This call says the socket is willing to accept datagrams sent to this particular port.

3 Wait for a request to arrive using the call `recvfrom` (IO$_READVBLK). In this call you specify the buffer, the length of the buffer, and the `sockaddr_in` structure that will be filled in with the information about who sent you that datagram.

4 Decode and service the request.

5 Send the reply using the call `sendto` (IO$_WRITEVBLK.).

6 Go to step 3.

The programs in Datagram Server Sample Programs Included with TCPware that demonstrate datagram server applications are included in the TCPWARE_ROOT:[TCPWARE.EXAMPLES] directory.

Table 1-5 Datagram Server Sample Programs Included with TCPware

Program	Description
BG_UDP_SERVER.C	DISCARD server that uses the BGDRIVER
UDPDRIVER.C	DISCARD server that uses the UDPDRIVER
UDPSAMPLE.FOR	DISCARD FORTRAN server that uses the UDPDRIVER
UDP_SOCK_SERVER.C	DISCARD server that uses the HP Socket Library

Writing Servers

Servers that need to service several connections at once are complex. They require use of multiplexing services because they must not block waiting for any one connection. Use:

- BSD `select` operation (if using a socket library interface)
- ASTs or EFNs for SYS$QIO (if using a QIO interface)

Such servers also typically require a context block per connection. This block includes the socket or channel number and other information about the state which that connection is in and any information processing that

connection requires.

You can also use a master server (such as NETCP in TCPware or `inetd` in UNIX) if you are writing a server. Initiating the server with the master server allows you to:

- Avoid a separate listening process for each service.

  For example, you might have several different servers, for each of which you would need a unique detached process. A master server avoids doing this because there is one process that listens for any of these ports. Once a connection comes in on one of these ports, the server creates a process to service that connection. The master server process does the listen and then the accept. When the accept has completed, it creates a detached process and that detached process services that connection. In the meantime, the master server has gone back to waiting for another connection to come in.

- Simplify support for multiple connections. This means you do not need to write multithreaded servers.

  When you use the master server process, you add server information to its database as to:

  – Whether the application uses TCP or UDP.
  – The well known port number or port name to listen on.
  – Information about the process that needs to be created (for example where the image is located, what privileges it must run with, and so on).

The server is started automatically when a connection is established or a datagram is sent to the well-known port. The server's input, output, and error files are assigned to the socket (or, in VMS, the device name). This simplifies the server because it does not need to do `socket`, `bind`, `listen`, and `accept` operations since these have been done by the master server. All the server does is service the one connection and events occurring on it.

For details on using the TCPware master server, see Appendix A, *TCPware Socket Library*, of this guide, and the ADD SERVICE command in the *NETCU Command Reference*.

Chapter 2 UCX Compatibility Services

Introduction

This chapter describes the UCX Compatibility Services.

The UCX Compatibility Services provide the following functions:

- BGDRIVER, the front-end QIO interface to TCPware that provides support for the HP TCP/IP Services for OpenVMS (formerly the VMS/ULTRIX Connection, or UCX) QIO functions.
- TCPware's master server supports BG devices (which are marked record-oriented). These extensions should allow UCX servers activated by the UCX master server to run under TCPware.

See the ADD SERVICE command in the *NETCU Command Reference* for details on setting services in TCPware.

- Support for the VMS 5.3 and later VAX C Run-Time Library (VAXCRTL) Socket Routines as described in the *VAX C Run-Time Library Reference Manual* available from HP.
- Support for OpenVMS Alpha and OpenVMS I64 DEC C Runtime Library (DECCRTL) Socket Routines.

For documentation on the HP TCP/IP Services for OpenVMS, see Bookreader help in the *Telecommunications and Networking* library, the *HP TCP/IP Services for OpenVMS VAX* collection, and the *HP TCP/IP for OpenVMS Services for OpenVMS System Services and C Socket Programming* book. This includes sections on writing Internet applications, using the OpenVMS System Services (including QIO calls) and DEC C Socket Routines (including a reference), and on-line programming examples.

Also see the *HP TCP/IP Services for VMS Programming Manual* from HP for details on the BGDRIVER $QIO interface.

Note! The UCX Compatibility Services are intended to be 100% compatible with HP's BGDRIVER.

If you discover that our BGDRIVER is not 100% compatible with the UCX BGDRIVER $QIO interface, please send us a sample program demonstrating the incompatibility.

The following files are included as part of the UCX QIO compatibility support in the directory TCPWARE_COMMON:[TCPWARE]:

BGDRIVER.EXE:	device driver that emulates the QIO functions.
UCX$INETDEF.H:	VAX C header file containing the UCX INET functions.
UCX$IPC.OLB:	transfer vectors used to resolve the socket routine references to the VAX C/DEC C Run-Time Library.

UCX$IPC_SHR.EXE:	Run-Time library used by VAXCRTL/DECCTRL to support the VAX C/DEC C Socket Routines.

If you have an application that was compiled and linked against UCX, you should be able to run that image on TCPware with no modifications (you do not need to re-link against TCPware).

Note! If you make changes to that application and want to compile and link against TCPware, follow the instructions under the *Sample Programs* section. The resulting image should run on TCPware or UCX systems.

Multicasting

UCX Compatibility Services includes the following `setsockopt` and `getsockopt` options at the UCX$C_IPOPT level for multicasting support:

UCX$C_IP_ADD_MEMBERSHIP and UCX$C_IP_DROP_MEMBERSHIP	The `setsockopt` operation adds and drops a multicast membership. The following structure is specified:
	`struct IPMREQDEF {`
	`/* Multicast group address */`
	`unsigned long int IMR$L_MULTIADDR;`
	`/* Local interface address */`
	`unsigned long int IMR$L_INTERFACE;`
	`} ;`
	—IMR$L_MULTIADDR contains the multicast internet address to be added or dropped and —IMR$L_INTERFACE contains the local interface's internet address on which the multicast address is added or dropped.
	If IMR$L_INTERFACE is specified as INADDR_ANY (0), the default multicast interface is used.

UCX$C_IP_MULTICAST_IF	The `setsockopt` operation sets the interface for subsequent multicast datagrams. The longword option value specifies the local internet address of the interface to be used. A `getsockopt` operation of this option returns the currently set interface (or 0 if none was set).
UCX$C_IP_MULTICAST_LOOP	The `setsockopt` operation enables or disables the local loopback of multicast datagrams. By default, this option is enabled. Specify a byte value of 1 to enable, 0 to disable. A `getsockopt` operation of this option returns the current multicast loopback setting.

UCX$C_IP_MULTICAST_TTL	The `setsockopt` operation sets the time-to-live (TTL) value for multicast datagrams. By default, this value is 1. A `getsockopt` operation of this option returns the current multicast TTL.

Logicals

TCPware defines the following logicals for UCX compatibility:

UCX$DEVICE TCPIP$DEVICE	defined as `BG:`, which is the name of the UCX device drive.
UCX$INET_HOST TCPIP$INET_HOST	defined to be the host name, which is the same setting as TCPWARE_DOMAINNAME logical.
UCX$IPC_SHR TCPIP$IPC_SHR	provides the linkage to the TCPware version of the UCX$IPC_SHR Run-Time library.

Note! The VAX C/DEC C Socket Routines `getnetbyname` and `getnetbyaddr` are supported and read the TCPWARE:NETWORKS. file.

IOCTL commands that set interface characteristics are not supported. Sensing of interfaces (SIOCGIFCONF, SIOCGIFADDR, SIOCGIFBRDADDR, SIOCGIFDSTADDR, SIOCGIFFLAGS, and SIOCGIFNETMASK) is supported.

Sample Programs

The following sample programs using UNIX-like sockets are included in the TCPWARE_COMMON:[TCPWARE.EXAMPLES] directory:

BGDRIVER_TCP_CLIENT.C	BGDRIVER_UDP_CLIENT.C
BGDRIVER_TCP_SERVER.C	BGDRIVER_UDP_SERVER.C

The BGDRIVER_TCP_CLIENT.C and BGDRIVER_TCP_SERVER.C pair of programs provides a self-declared ECHO server that sequentially accepts client connections and echoes back the client messages. The BGDRIVER_UDP_CLIENT.C and BGDRIVER_UDP_SERVER.C pair of programs provide a self-declared DISCARD server that can receive (and discard) datagrams from multiple clients. These programs are functionally equivalent to the socket programs in Chapter 8, *Socket Library*.

To build any one of these applications using DEC C, enter:

```
$ CC/DECC/PREFIX=ALL/DEFINE=TCPWARE filename
$ LINK filename
Ctrl/Z
```

To build any one of these applications using VAX C, enter:

```
$ CC/VAXC/DEFINE=TCPWARE filename
$ LINK filename, TCPWARE:UCX$IPC/LIB, SYS$INPUT/OPTIONS-
_$  SYS$SHARE:VAXCRTL/SHARE
Ctrl/Z
```

You can build these programs on both TCPware and UCX systems. The `/DEFINE=TCPWARE` uses code to point to a TCPware include directory for building on a TCPware system.

Debugging and Tracing

TCPware provides a call tracing facility that can be used to debug and trace the use of the sockets API for many applications. This facility works for both the TCPware socket library and the API that the newer versions of the C compiler work with. This does NOT log QIO operations. To enable the tracing define the TCPWARE_SOCKET_TRACE logical name. The value of the logical name can be used in the following ways:

- As a bit mask for types of operations to trace. Bit 0 (zero) signifies control operations, bit 1signifies read operations and bit 2 signifies write operations. When these values are used the information is written to SYS$OUTPUT:.
- As a partial or full file name. When used as a partial file name the default name specified to open the file is: SYS$SCRATCH:TCPWARE_SOCKET_<process_name>.LOG. Control, read and write operations are logged when logging is done to a file.

Chapter 3 TCPDRIVER Services

Introduction

This chapter describes the Transmission Control Protocol (TCP) device driver (TCPDRIVER) services. It describes the user interface of this implementation of TCP only. RFC 793 contains the full TCP specifications. There are no implementation-specific restrictions for the Transmission Control Protocol (TCP). The material presented here does not explain or describe the TCP protocol.

TCP is a connection-oriented, end-to-end reliable protocol. It provides reliable communication between pairs of processes in computers attached to interconnected networks. TCP services are available through the OpenVMS Queue I/O (SYS$QIO and SYS$QIOW) system services, which:

- Open and close a connection.
- Send and receive data over the connection.
- Perform status checks on the connection.

SYS$QIOW is the synchronous and SYS$QIO the asynchronous form of VMS system services. Use each form depending on the requirements of your application.

See the appropriate OpenVMS documentation for more information on the OpenVMS I/O system services and related services, such as the asynchronous system trap (AST) and event flag services.

Note! The QIO calls described in this chapter are used for direct access to the TCPDRIVER. If you are porting an application that uses the BGDRIVER or INETDRIVER QIO interface, you may not need to make modifications. Use TCPware's BGDRIVER (see Chapter 2) or INETDRIVER (see Chapter 6).

Sequence of Operations

The sequence of operations to open a connection are as follows:

1 Assign an I/O channel to TCP0: with the Assign I/O Channel (SYS$ASSIGN) system service. SYS$ASSIGN creates a new device unit and assigns to it the channel.

2 Open the connection with the IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP function of one of the Queue I/O (SYS$QIO or SYS$QIOW) system services.

3 Perform read requests with the IO$_READVBLK function and write requests using the IO$_WRITEVBLK function as desired.

4 Close your end of the connection with the IO$_SETMODE | IO$M_CTRL | IOM$_SHUTDOWN function.

5 Perform additional read requests until the peer returns SS$_VCCLOSED status.

6 Deassign the I/O channel using the Deassign I/O Channel (SYS$DASSGN) system service.

Other Operations

In addition to the sequence of operations described in the preceding section, TCPDRIVER includes other operations that:

Place a connection into listen state and specify the maximum number of incoming connections that can be queued waiting to be accepted, then wait for and accept an incoming connection	IO$_CREATE
Read data immediately	IO$_READVBLK \| IO$M_NOW
Read the received data without removing it from the received data queue	IO$_READVBLK \| IO$M_DATACHECK
Read only completely filled data buffers	IO$READVBLK \| IO$M_PACKED
Send data stored in a list of buffers	IO$_WRITEVBLK \| IO$M_EXTEND
Read the active connections status	IO$_SENSEMODE
Read the connection characteristics for the channel	IO$_SENSEMODE \| IO$M_CTRL
Read the TCP counters	IO$_SENSEMODE \| IO$M_RD_COUNT
Set the connection characteristics	IO$_SETMODE \| IO$M_CTRL
Request delivery of an attention AST	IO$_SETMODE \| IO$M_ATTNAST
Abort a connection	IO$_SETMODE \| IO$M_CTRL | IO$M_SHUTDOWN \| IO$M_ABORT
Cancel any pending I/O requests	SYS$CANCEL

TCPDRIVER System Service Call Format

The format for the TCPDRIVER SYS$QIO system service call is as follows:

status = SYS$QIO(*efn*, *chan*, *func*, *iosb*, *astadr*, *astprm*, *p1*, *p2*, *p3*, *p4*, *p5*, *p6*)

The SYS$QIO or SYS$QIOW system calls issue TCP functions. SYS$QIO is for asynchronous service completion. It specifies that the service return to the caller immediately after queuing the I/O request. SYS$QIOW is for synchronous service completion. It specifies that the service place the calling process in a wait state and only return to the caller after completing the I/O operation. The OpenVMS IODEF module provides definitions for the SYS$QIO function codes.

Note! The vertical bar (|) used in some of the functions described in this chapter is the C bit-wise inclusive OR operator.

TCPDRIVER System Service Call Arguments

You invoke TCPDRIVER system service calls with the standard OpenVMS QIO mechanism.

See the appropriate OpenVMS documentation (for example, the *Introduction to VMS System Services* volume) for more information on the QIO mechanism.

The following sections describe each system call argument.

efn

OpenVMS usage:	**ef_number**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Number of the event flag set by completion of the I/O operation. The argument is optional.

chan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	read only
mechanism:	by value

I/O channel assigned to the TCP device to which you are directing the request.

func

OpenVMS usage:	**function_code**
type:	word (unsigned)
access:	read only

mechanism:	by value

Device-specific function code and the modifier if appropriate for each operation.

Note! *TCPDRIVER System Service Call Function Codes* describes each *func*.

iosb

OpenVMS usage:	**io_status_block**
type:	quadword (unsigned)
access:	write only
mechanism:	by reference

I/O status block that receives the final completion status of the I/O operation. Structured as in Figure 3-1. Table 3-1 describes the status block fields in detail.

Figure 3-1 I/O Status Block

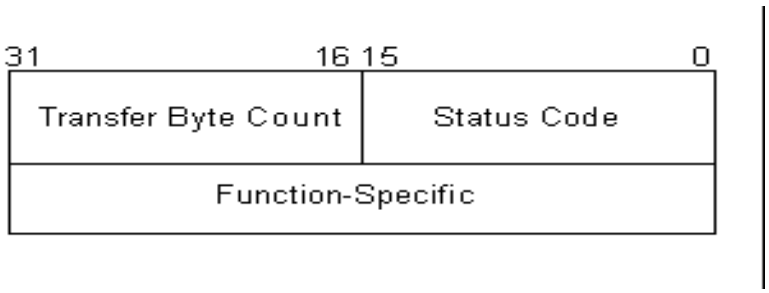


Table 3-1 I/O Status Block Fields

Field Name	**Description**
Function-Specific	Varies for each function code.
Status Code	The SS$ status code or special error status code. If the low bit (0) of the OpenVMS error code is clear, the network has returned an error.
Transfer Byte Count	Number of bytes of data transferred in the I/O operation.

astadr

OpenVMS usage:	**ast_procedure**
type:	procedure entry mask
access:	call without stack unwinding
mechanism:	by reference

Address of the asynchronous system trap (AST) routine executed when the I/O is completed.

astprm

OpenVMS usage:	**user_arg**
type:	longword (unsigned)
access:	read only
mechanism:	by value

AST parameter to be passed to the AST routine.

p1 to p6

OpenVMS usage:	**varying_arg**
type:	longword (unsigned)
access:	read only or write only
mechanism:	by reference or by value

Function-specific parameters, as described for each function.

TCPDRIVER System Service Call Function Codes

System service call function codes specify what action the QIO performs. This section describes the TCPDRIVER function codes.

IO$_CREATE

Provides for socket style listens and accepts.

Note! This function and socket style listens and accepts for TCPDRIVER are new to TCPware starting with version 5.3. Earlier releases do not support this function and return an SS$_ILLIOFUNC status if used.

Use the IO$_CREATE function to perform one of three operations:

ACCEPT	Accept an incoming connection
ACCEPT-WAIT	Wait for an incoming connection
LISTEN	Place a connection into listen state and specify the maximum number of incoming connections that can be queued waiting to be accepted

A typical flow for a program using this function might be as follows:

1 Assign a channel to the TCP: device (SYS$ASSIGN). This is the listening channel.

2 Issue an IO$_SETMODE | IO$M_CTRL function on the listening channel to specify the local port number for the service.

3 Issue an IO$_CREATE function on the listening channel to specify a large *backlog* (greater than 128 is recommended)—see theFormat for L.

4 Issue an IO$_CREATE function (with all parameters set to zero) on the listening channel to await an incoming connection. (You could alternatively use an asynchronous QIO and AST routine). This function blocks until a connection is available—see the Format for A.

5 Assign a channel to the TCP: device (SYS$ASSIGN) and issue an IO$_CREATE | IO$M_NOW function on the listening channel with *newchan* being the newly assigned channel—see the Format for A.

6 The newly assigned channel (step 5) has the accepted connection and should be serviced (reads and writes).

7 Loop back to step 4 to continue accepting connections.

Ideally, steps 4 through 7 should be designed such that multiple connections can be serviced simultaneously.

Format for LISTEN

status =SYS$QIO(*efn*, *chan*, IO$_CREATE, *iosb*, *astadr*, *astprm*, 0, 0, 0, *backlog*, 0, 0)

Description for LISTEN

This function places the connection into a listen state. If you did not previously specify a local port number for the channel (see theIO$_SETMODE | IO$M_CTRL function), a unique port number is automatically assigned. The function completes execution immediately as it does not wait for an incoming connection.

Argument for LISTEN

p4=backlog

OpenVMS usage:	**unsigned word**
type:	word (unsigned)

access:	read only
mechanism:	by value

Connection backlog, or maximum number of connections that can be queued. It is recommended that you specify a large backlog (greater than 128) for most applications.

Status for LISTEN

SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_FILALRACC	Port is busy Only ports that are in the closed state can be used
SS$_ILLIOFUNC	Invalid parameters or modifiers were specified (returned in status, not in *iosb*)
SS$_NORMAL	Success Connection put in listen state

Format for ACCEPT-WAIT

status =SYS$QIO(*efn*, *chan*, IO$_CREATE, *iosb*, *astadr*, *astprm*, 0, 0, 0, 0, 0, 0)

Description for ACCEPT-WAIT

This function waits for an incoming connection to be available on a listening connection. A listen function as specified in the LISTEN operation above must have been previously issued. You can use the IO$M_NOW modifier to probe if a connection is available; SS$_NODATA is returned if no connection is available.

Status for ACCEPT-WAIT

SS$_DEVINACT	Device not in listen state or not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_NODATA	No connection is currently available to be accepted Only returned if IO$_CREATE \| IO$M_NOW was specified as the function code
SS$_NORMAL	Success Connection put in listen state

SS$_THIRDPARTY	TCPDRIVER was shut down by a third party

Format for ACCEPT

status =SYS$QIO(*efn*, *chan*, IO$_CREATE | IO$M_NOW, *iosb*, *astadr*, *astprm*, 0, 0, *newchan*, 0, 0, 0)

Description for ACCEPT

This function accepts an incoming connection. A listen function as specified in the LISTEN operation above must have been previously issued on the channel (*chan*). As this function does not block, a wait should generally precede it for an incoming connection function, as specified in the ACCEPT-WAIT operation above.

Argument for ACCEPT

p3=newchan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	read only
mechanism:	by value

OpenVMS channel for a newly created TCP: device. This channel should be created by using SYS$ASSIGN to assign a new channel to the TCP0: device before issuing this function. The accepted connection uses this channel.

Status for ACCEPT

SS$_DEVACTIVE	I/O channel specified in *newchan* is not inactive Only inactive channels can be used
SS$_DEVINACT	Connection is not in listen state or the device is not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_IVDEVNAM	I/O channel specified in *newchan* is not a TCP: device
SS$_NODATA	No connection is available to accept

IO$_READVBLK

Receives data, including urgent data, writing the received data to the specified user buffer.

The IO$M_DATACHECK modifier peeks at the next received data without removing it from the received data queue.

The IO$M_NOW modifier executes the function immediately regardless of the amount of available data. If no data is available, the function returns the SS$_NODATA status.

The IO$M_PACKED modifier requests that IO$_READVBLK return completely filled data buffers. IO$_READVBLK completes reads with this modifier only after receiving the requested number of bytes, unless the port closes or you specify IO$M_NOW.

Note! Use the IO$M_PACKED modifier with caution, as the read request only completes execution after receiving the requested number of bytes, or if the connection is closed or reset by the peer.

Format

status =SYS$QIO(*efn*, *chan*, IO$_READVBLK, *iosb*, *astadr*, *astprm*, *buffer*, *size*, 0, 0, 0, 0)

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	write only
mechanism:	by reference

Address of the user's buffer that receives the data.

p2=size

OpenVMS usage:	**word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by value

User's buffer size in bytes. Specifies the maximum amount of data written to the buffer. The value should not be greater than 64000 bytes.

Status

SS$_CANCEL	Request canceled
SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_NODATA	No data available and IO$M_NOW was specified.
SS$_NORMAL	Success Requested data was received.
SS$_PATHLOST	Route to peer lost Network or route used in communicating with peer no longer working Connection closed
SS$_THIRDPARTY	Connection broken by third party Usually indicates that TCPware was shut down Connection closed
SS$_TIMEOUT	Connection timed-out Connection closed
SS$_VCBROKEN	Connection broken (either locally or by source) and closed
SS$_VCCLOSED	Peer closed end of the connection and no more data is available, or the connection was never opened

The number of bytes received is returned in the high-order word of the first longword of the I/O status block. This may be less than the number of bytes requested.

The second longword of the I/O status block contains flag bits as follows:

- If bit 1 (mask value of 2) is set, there is still more data to be read before the end of the urgent data marker. This indicates that IO$_READVBLK has not read all of the urgent data.
- If bit 2 (mask value of 4) is set, the data was delivered with the urgent data marker in advance of the data. There may still be more data to be read before the end of the urgent data depending on the state of bit 1. This bit is always set if bit 1 is set.

  All other bits in the second longword of the I/O status block are clear (0).

IO$_SENSEMODE

Reads the active connections status and returns status information for all of the active and listening TCPDRIVER connections.

Format

status=SYS$QIO(*efn*, *chan*, IO$_SENSEMODE, *iosb*, *astadr*, *astprm*, *buffer*, *address*, 0, 0, 0, 0)

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	write only
mechanism:	by reference

Optional address of the 8-byte device characteristics buffer. Data returned is: the device class (DC$_SCOM) in the first byte, the device type (0) in the second byte, and the default buffer size, which is the maximum datagram size, in the high-order word of the first longword. IO$_SENSEMODE returns the second longword as 0.

p2=address

OpenVMS usage:	**vector_word_unsigned**
type:	word (unsigned)
access:	write only
mechanism:	by descriptor

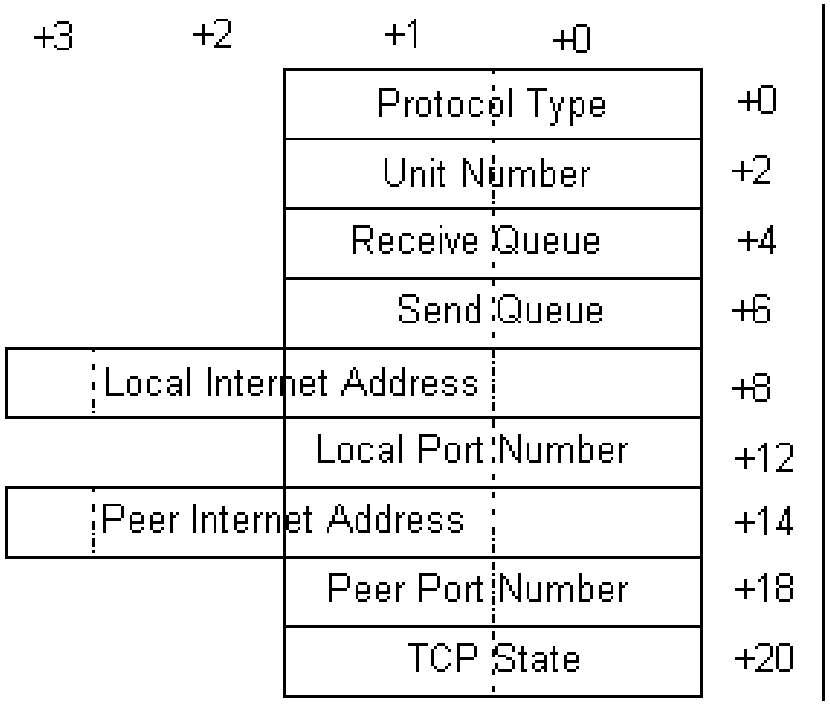
Address of the descriptor for the buffer to receive the status information on the active connections.

Figure 3-2 shows the 22 bytes of information returned for each connection.

Protocol type	Word value is 2 for TCPDRIVER connections, 4 for INETDRIVER stream sockets, and 5 for BGDRIVER stream sockets.
Unit number	Word value is the TCPDRIVER, INETDRIVER, or BGDRIVER device unit number for the connection.

Receive queue	Word value is the number of bytes received from the peer waiting to be delivered to the user through the IO$_READVBLK function.
Send queue	Word value is the number of bytes waiting to be transmitted to or to be acknowledged by the peer.
Local internet address	Longword value is the local internet address (or 0 if the connection is not open and no local internet address was specified for the connection).
Local port number	Word value is the local port number.
Peer internet address	Longword value is the peer's internet address (or 0 if the connection is not open and no peer internet address was specified for the connection).
Peer port number	Word value is the peer's port number, or 0 if the connection is not open and you did not specify a peer port number for the connection.
TCP state	Word value is the Transmission Control Protocol connection state mask. See theIO$_SETMODE \| IO$M_CTRL description for the mask value definitions.

Figure 3-2 Connection Status Information



Status

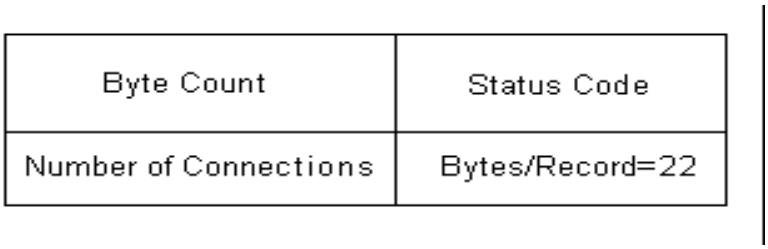
SS$_BUFFEROVF	Buffer too small for all connections Truncated buffer returned
SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_NORMAL	Success Status information returned

The byte count for the status information buffer is returned in the high-order word of the first longword of the I/O status block. This may be less than the bytes requested. See Figure 3-3 for more information.

The size in bytes of each connection's record (22 bytes) is returned in the low order word of the second longword of the I/O status block.

The total number of active connections is returned in the high-order word of the second longword of the I/O status block. This can be greater than the number of reported connections if the buffer is full.

Figure 3-3 I/O Status Block



IO$_SENSEMODE | IO$M_CTRL

Reads the connection characteristics for the channel.

Format

status=SYS$QIO(*efn*, *chan*, IO$_SENSEMODE | IO$M_CTRL, *iosb*, *astadr*, *astprm*, *buffer*, *address*, 0, 0, 0, 0)

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	write only
mechanism:	by reference

Optional address of the 8-byte device characteristics buffer. Data returned is: device class (DC$_SCOM) in the first byte, device type (0) in the second byte, and default buffer size (the maximum window size) in the high-order word of the first longword. IO$_SENSEMODE | IO$M_CTRL returns the second longword as 0.

p2=address

OpenVMS usage:	**vector_word_unsigned**
type:	word (unsigned)
access:	write only
mechanism:	by descriptor

Address of the descriptor for the extended characteristics buffer to receive the characteristics. Information returned in the buffer is formatted the same as the extended characteristics buffer used to set the connection characteristics. See the IO$_SENSEMODE function for details.

If bit 12 (mask 4096) is set in the parameter identifier (PID), the PID is followed by a counted string. If bit 12 is clear, the PID is followed by a longword value. While TCPware currently never returns a counted string for a parameter, this may change in the future.

Table 3-2 shows the read connection characteristics.

Table 3-2 P2 Read Connection Characteristics

PID	Meaning
0	**Local internet address.** Internet address is returned as the longword value. Only returned if the connection is open or a local internet address has been specified.
1	**Local port number.** Port number is returned in the low order word of the longword value. Local port number is either the default port number or the last specified local port number.
2	**Peer internet address.** Internet address is returned as the longword value. Only returned if the peer's internet address has been specified or the connection is open.
3	**Peer port number.** Port number is returned in the low order word of the longword value. Only returned if the peer's port number has been specified or the connection is open.
4	**Connection time-out value.** Time-out value (in seconds) is returned as the longword value.
5	**Connection window size.** Window size (in bytes) is returned as the longword value.
6	**TCP state.** State is returned as a longword bit mask value. Mask values and states are shown in Table 3-3**.** Only one bit can be set at any time.
7	**Passive open access control.** Value indicating shared (non-zero) or exclusive (zero) access to the same port number is returned as the longword value.
10	**Socket options.** For the values of this option, see the socket options entry in P2 Set Connection Characteristics .

Note! The order in which IO$_SENSEMODE | IO$M_CTRL returns the parameters can change. It is recommended that you search the buffer for the desired parameter.

Table 3-3 TCP State Mask Values

Mask Value	State	Mask Value	State	Mask Value	State
1	LISTEN	16	FIN-WAIT-1	256	LAST-ACK
2	SYN-SENT	32	FIN-WAIT-2	512	TIME-WAIT
4	SYN-RECEIVED	64	CLOSE-WAIT	1024	CLOSED
8	ESTABLISHED	128	CLOSING		

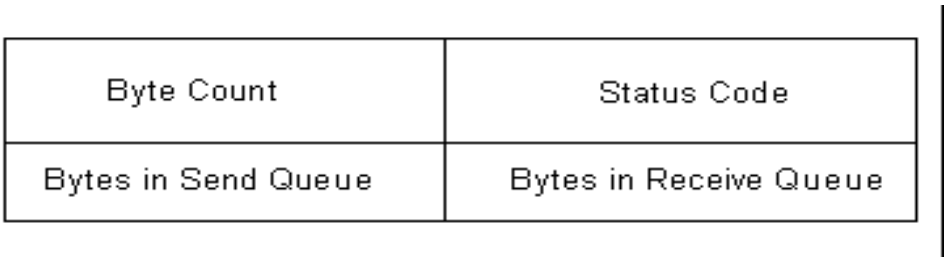
Status

SS$_BUFFEROVF	Buffer too small for all characteristics Truncated characteristics buffer is returned
SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started

SS$_NORMAL	Success Characteristics returned

The byte count for the characteristics buffer is returned in the high-order word of the first longword of the I/O status block. This may be less than the bytes requested. The number of bytes in the receive queue is returned in the low order word of the second longword in the I/O status block. The number of bytes in the read queue is returned in the high-order word of the second longword in the I/O status block. Figure 3-4 shows the I/O Status Block.

Figure 3-4 I/O Status Block



Note! You can use the SYS$GETDVI system service to obtain the local port number, peer port number, and peer internet address. The DEVDEPEND field stores the local port number (low order word) and peer port number (high-order word). The DEVDEPEND2 field stores the peer internet address.

IO$_SENSEMODE | IO$M_RD_COUNT

Reads the TCP counters. You can add the IO$M_CLR_COUNT modifier to this function to zero the counters after they are read. However, you must have the OPER privilege to use this modifier.

Format

status=SYS$QIO(*efn*, *chan*, IO$_SENSEMODE | IO$M_RD_COUNT, *iosb*, *astadr*, *astprm*, 0, *address*, 0, 0, 0, 0)

Argument

p2=address

OpenVMS usage:	**vector_longword_unsigned**
type:	longword (unsigned)
access:	write only
mechanism:	by descriptor

Address of the descriptor for the buffer to receive the counters. These counters relate only to the TCP level and are for all connections opened since the counters were last zeroed.

The counters (longword values) in the order in which they are returned:

1 Number of seconds since last zeroed

2 Number of segments transmitted (excludes retransmissions)

3 Number of segments retransmitted (includes keep-alive check transmissions)

4 Number of segments transmitted with transmission errors

5 Number of segments received

6 Number of segments received in error (invalid TCP header or checksum)

7 Number of receive data buffers concatenated (data segments are concatenated whenever possible)

8 Number of delayed acknowledgments transmitted (included in segments transmitted count)

9 Number of window updates transmitted due to receive request completions (included in segments transmitted count)

10 Number of out-of-sequence segments received

11 Number of transmit data buffers concatenated

12 Number of keep-alives transmitted (included in number of segments re-transmitted)

Status

SS$_BUFFEROVF	Buffer too small for all counters Truncated buffer is returned

SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_NORMAL	Success Counters returned

The byte count for the counters buffer is returned in the high-order word of the first longword of the I/O status block. This may be less than the bytes requested.

IO$_SETMODE | IO$M_ATTNAST

Requests the delivery of an attention AST whenever urgent data is available or the connection is broken.

This function enables an attention AST to be delivered only once. The function is subject to AST quotas. After the function delivers the AST, you must issue another IO$_SETMODE | IO$M_ATTNAST to re-enable AST delivery.

When a connection fully closes, all attention ASTs for the connection are disabled.

Note! If the existence of urgent data triggers the attention AST, and that data is not read before you re-enable the AST, the AST triggers again immediately.

Format

status=SYS$QIO(*efn*, *chan*, IO$_SETMODE | IO$M_ATTNAST, *iosb*, *astadr*, *astprm*, *address*, *param*, *mode*, 0, 0, 0)

Arguments

p1=address

OpenVMS usage:	**ast_procedure**
type:	procedure entry mask
access:	call without stack unwinding
mechanism:	by reference

Address of an AST service routine or 0 to disable attention ASTs.

p2=param

OpenVMS usage:	**user_arg**
type:	longword (unsigned)
access:	read only
mechanism:	by value

AST parameter to be passed to the AST service routine.

p3=mode

OpenVMS usage:	**access mode**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Access mode at which to deliver the AST. The access mode is maximized with the requestor's access mode.

Status

SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_EXQUOTA	AST quota exceeded
SS$_NORMAL	Success Attention AST enabled or previous ASTs disabled

IO$_SETMODE | IO$M_CTRL

Sets the connection characteristics.

You specify the connection characteristics in the extended characteristics buffer. The buffer consists of a series of six-byte entries. The first word of each entry stores the parameter identifier (PID) followed by a longword that stores one of the values associated with that parameter.

Format

status=SYS$QIO(*efn*, *chan*, IO$_SETMODE | IO$M_CTRL, *iosb*, *astadr*, *astprm*, 0, *address*, 0, 0, 0, 0)

Argument

p2=address

OpenVMS usage:	**vector_word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by descriptor

Address of the descriptor for the extended characteristics buffer. Counted strings consist of a word with the size of the character string followed by the character string.

Figure 3-5 shows the format of the descriptor for the extended characteristics buffer.

Figure 3-5 P2 Set Characteristics Buffer

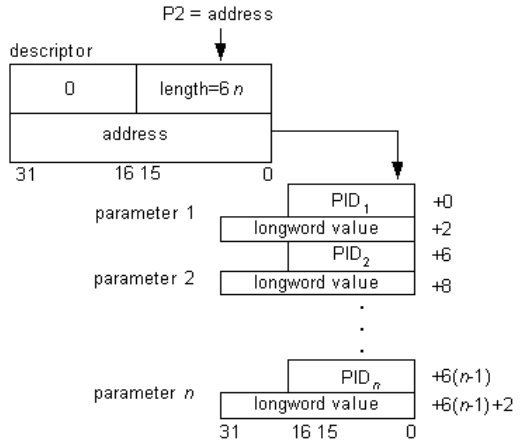


Table 3-4 lists the connection characteristics you can set. These characteristics remain set, even after you close the connection, until you change them with another IO$_SETMODE operation.

Table 3-4 P2 Set Connection Characteristics

PID	Meaning
0	**Local internet address.** Longword with the local host's internet or multicast address. If used, must be a valid local internet or multicast address. Set this parameter only if the port is not open. Specify in internet byte order, reversed from the normal VMS byte order. For example, internet address 2.3.4.5 is stored as ^X05040302. To determine if an internet address is local, issue an IO$_SETMODE \| IO$M_CTRL function specifying the address in this characteristic. If the address is not local, the function returns an `SS$_BADPARAM` status.
1	**Local port number.** Low order word of the longword containing the port number. If omitted, the function uses the unique port number generated when you assign the UDP device unit. Set this parameter only if the port is not open. If you use `0`, the function generates a new, unique number. Specify in normal VMS byte order.
2	**Peer internet address.** Longword with the peer host's internet address. If you specify both the peer's internet address and port number after opening the port (or when the port is opened) the port is considered fully specified (or opened as fully specified) and all further receive requests receive datagrams from that address and port only. Transmitted requests that do not provide a source/destination buffer are sent to that address and port. Specify in internet byte order, as the local internet address characteristic above. Using `0` functionally omits the address.
3	**Peer port number.** Low order word of the longword containing the port number. (See PID 2 above.) Specify in normal VMS byte order. Using `0` functionally omits the port number.
4	**Connection time-out value.** Longword specifying the connection time-out value in seconds. You can also specify the connection time-out value as P6 in the IO$_SETMODE \| IO$M_CTRL | IO$M_STARTUP and IO$_WRITEVBLK functions. The value specified by P6 has precedence over the value specified by the extended characteristics buffer. The default value is 5 minutes (300 seconds). The minimum is 20 seconds.
5	**Connection window size.** Longword specifying the connection's window size (in bytes). Typical values for connection window sizes range from 4096 to 32768 and specify how much data the TCP layer will buffer for the application. If omitted, the default window size configured during network start up is used for the connection.
7	**Passive open access control.** Specifies whether others may (shared access) or may not (exclusive access) use the same local port number in passive opens. To specify exclusive access (the default), the longword value must be 0. To specify shared access (used primarily by servers), the longword value must be non-zero. This parameter only applies to passive opens.
9	**Change ownership.** Longword that allows the current owner of the device unit to prepare to pass ownership to another process. Removes the owner's process ID from the connection. If the value field is non-zero, specifies the new owner UIC for the connection and changes the connection protection so that only system and owner have access (S:RWLP,O:RWLP). Changes the following fields for the connection: owner process ID, owner UIC, and device protection. Primarily the NETCP master server but also other programs use it.

10	**Set socket options.** Low order word of the longword value field containing the mask value of the socket options to set. Specify the following options, as defined in the TCPWARE_INCLUDE:SOCKET.H header file (with values in hexadecimal): SO_DEBUG, with a value of 0001, enables debugging (ignored) SO_REUSEADDR, with a value of 0004, allows reuse of local address (controllable through the passive open access control parameter) SO_KEEPALIVE, with a value of 0008, keeps connections alive SO_DONTROUTE, with a value of 0010, prevents routing (ignored) SO_BROADCAST, with a value of 0020, enables use of broadcasts (ignored) SO_USELOOPBACK, with a value of 0040, bypasses hardware when possible (ignored) SO_LINGER, with a value of 0080, lingers on a close
11	**Clear socket options.** Low order word of the longword value field containing the mask of the socket options to clear. (See the options under PID 10 above.)
12	**No delay.** Determines whether TCPDRIVER delays sending data to coalesce data from several small sends into a large packet for transmission over the network. The default is for TCPDRIVER to delay. Most applications should not change this option since the delay is very short and makes for better overall network performance. TCPDRIVER uses the parameter value passed to disable the delay (if the value is non-zero) or enable the delay (if the value is 0).
13	**IP options.** Counted string containing the IP options to be included in datagrams.

Note! The multicasting-related parameters (14 through 20) are supported but cannot be used with TCP (a connection-oriented protocol). Therefore, the settings are meaningless.

Status

SS$_BADPARAM	Bad parameter or value specified PID returned in low order word of second longword
SS$_DEVACTIVE	Connection open on unit; local port number, peer's internet address, and peer's port number cannot be changed
SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_INSFMEM	Insufficient memory to complete request
SS$_IVBUFLEN	Extended characteristics buffer length invalid
SS$_NOPRIV	Setting IP layer options (PIDs 13 through 20) requires privileges Application must either be running under SYSTEM UIC or have SYSPRV, BYPASS, or OPER privilege

SS$_NORMAL	Successful operation Characteristics set
SS$_UNREACHABLE	Default interface for multicast group address could not be found Specify local interface's IP address when joining multicast group

IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN

Closes your end of the connection. Lets the other end of the connection know that you completed sending data. You cannot perform any IO$_WRITEVBLK functions after issuing IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN. However, you can perform IO$_READVBLK functions on the connection until the other end of the connection closes and you have read all the data.

Note! A connection does not close until it closes on both ends. In other words, your end of the connection does not close until the peer has acknowledged the IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN.

The recommended procedure for closing a connection is to issue IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN and then read until SS$_VCCLOSED status is returned. Use the IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN | IO$M_ABORT function to abort a connection. This function returns all requests for the connection with the SS$_VCBROKEN status. If the connection is open, it is reset.

Format

status=SYS$QIO(*efn*, *chan*, IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN, *iosb*, *astadr*, *astprm*, 0, 0, 0, 0, 0, 0*)*

Arguments

None.

Status

SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_NORMAL	Successful operation Connection closed
SS$_PATHLOST	Route to peer lost Network or route used in communicating with peer no longer working Connection closed
SS$_THIRDPARTY	Connection broken by third party Usually indicates that TCPware was shut down Connection closed
SS$_TIMEOUT	Connection timed-out Connection closed
SS$_VCBROKEN	Connection broken (either locally or by source) and closed
SS$_VCCLOSED	Connection already closed or in process of being closed

IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP

Opens an active connection. For an active open connection, specify the peer's internet address and port number. If you omit the local port number, the function uses a unique port number assigned when the unit was created.

Use the IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP | ^X0800 function to open a passive connection. You can omit the peer's internet address or port number. The function completes execution only after the connection is fully established.

Format

status=SYS$QIO(*efn*, *chan*, IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP, *iosb*, *astadr*, *astprm*, 0, *address*, 0, 0, 0, *time*)

Arguments

p2=address

OpenVMS usage:	**vector_word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by descriptor

Address of the descriptor for the extended characteristics buffer. See theIO$_SENSEMODE | IO$M_CTRL description for details on this buffer.

p6=time

OpenVMS usage:	**longword_unsigned**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Optional connection time-out value in seconds. The default is the value specified in the extended characteristics buffer (or 5 minutes, if none was specified). The minimum value is 20 seconds.

Status

SS$_BADPARAM	Bad parameter or value specified
SS$_CANCEL	Request canceled
SS$_DEVACTIVE	Connection open on unit
SS$_DUPUNIT	Connection not unique Check local port number, peer's internet address, and peer's port number
SS$_IVBUFLEN	Buffer length invalid
SS$_NORMAL	Successful operation Connection open Use write and read requests to send or receive data
SS$_NOPRIV	Insufficient privilege Tried to open passive connection on port number below 1024 without BYPASS or SYSPRIV privilege
SS$_PATHLOST	Route to peer lost Network or route used in communicating with peer no longer working Connection closed
SS$_THIRDPARTY	Connection broken by third party Connection closed
SS$_TIMEOUT	Connection timed-out and closed Active open not completed within connection timeout
SS$_VCBROKEN	Connection broken (either locally or by peer) and closed Active open returns if peer refused connection

IO$_WRITEVBLK

Sends data stored in the specified user buffer. Completes execution after queuing the data for transmission.

You can use the IO$M_OUTBAND modifier with IO$_WRITEVBLK to transmit urgent data.

Format

status=SYS$QIO(*efn*, *chan*, IO$_WRITEVBLK, *iosb*, *astadr*, *astprm*, *buffer*, *size*, 0, 0, 0, *time)*

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	read only
mechanism:	by reference

Address of the user's buffer. The user's buffer stores the data to be sent.

p2=size

OpenVMS usage:	**word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by value

User's buffer size in bytes. Amount of data that the user is sending. If 0, the IO$_WRITEVBLK completes execution after transmitting all previously written data to the peer. However, the peer has not necessarily received this data yet. The value should not be greater than 64000 bytes.

p6=time

OpenVMS usage:	**longword_unsigned**
type:	longword (unsigned)

access:	read only
mechanism:	by value

Optional new-connection timeout value (in seconds).

Status

SS$_BADPARAM	Bad parameter or value specified If P6 is not zero, value specified is not acceptable; if IO$M_EXTEND modifier specified, P2 is not 24 (size of `msghdr` structure)
SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_IVBUFLEN	Extended characteristics buffer length invalid
SS$_NORMAL	Successful operation Data queued for sending to destination
SS$_PATHLOST	Route to peer lost Network or route used in communicating with peer no longer working Connection closed
SS$_THIRDPARTY	Connection broken by third party Usually indicates that TCPware was shut down Connection closed
SS$_TIMEOUT	Connection timed-out and fully closed Data could not be delivered to destination within connection time-out
SS$_VCBROKEN	Connection broken (either locally or by source) and fully closed
SS$_VCCLOSED	You issued close function to close end of connection, or connection never opened

The number of bytes sent is returned in the high-order word of the first longword of the I/O status block.

IO$_WRITEVBLK | IO$M_EXTEND

Sends data stored in a list of buffers.

Format

status=SYS$QIO(*efn*, *chan*, IO$_WRITEVBLK | IO$M_EXTEND, *iosb*, *astadr*, *astprm*, *msghdr*, *size*, 0, 0, 0, 0*)*

Arguments

p1=msghdr

OpenVMS usage:	**vector_longword_unsigned**
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of the `msghdr` structure. Points to the address of the structure storing the array of additional structures. Each of these array structures contains the address and size of one of the user buffers. The `msghdr` structure is as follows:

```
struct msghdr {
   char     *msg_name;            /*optional address*/
   int      msg_namelen;          /*size of address*/
   struct   iovec *msg_iov;       /*scatter/gather array*/
   int      msg_iovlen;           /*elements in msg_iov*/
   char     *msg_accrights;       /*access rights*/
   int      msg_accrightslen;
};

struct iovec {
   char     *iov_base;            /*address of buffer*/
   int      iov_len;              /*length of buffer*/
};
```

In the `msghdr` structure, this function uses only `msg_iov` and `msg_iovlen`. It ignores the values of the other elements.

`msg_iov` stores the address of the array of `iovec` structures and `msg_iovlen` is the number of iovec elements in the array. The iovec array stores the address and length of each buffer (in bytes) that holds the data to be sent.

p2=size

OpenVMS usage:	**longword_unsigned**
type:	longword (unsigned)

access:	read only
mechanism:	by value

Size of the `msghdr` structure, which must be 24 bytes. The following is an example using the scatter-gather write function. Figure 3-6 shows a diagram for the example.

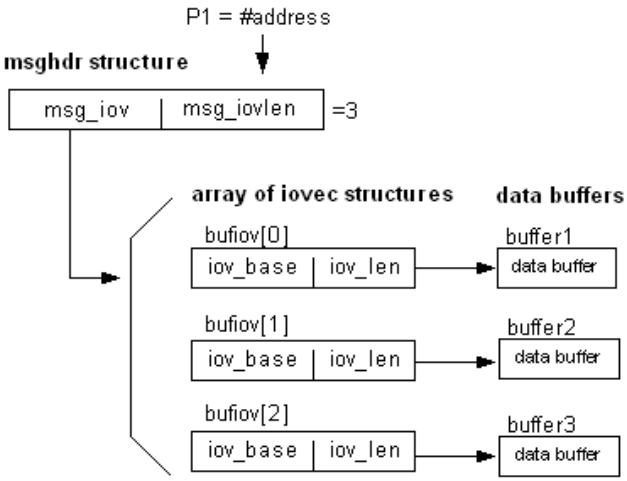
```
struct  msghdr  buflst;
struct  iovec   bufiov[3];
char    buffer1[100];
char    buffer2[20];
char    buffer3[200];

buflst.msg_iov = bufiov;
buflst.msg_iovlen = 3;

bufiov[0].iov_base = buffer1;
bufiov[0].iov_len = sizeof(buffer1)
bufiov[1].iov_base = buffer2;
bufiov[1].iov_len = sizeof(buffer2)
bufiov[2].iov_base = buffer3;
bufiov[2].iov_len = sizeof(buffer13)

istat = SYS$QIOW(0,chan,
   IO$_WRITEVBLK | IO$M_EXTEND,&iosb,0,0,buflst,sizeof(buflst),
   0,0,0);
```

Figure 3-6 Scatter-Gather Write Array Structure



Status

See IO$_WRITEVBLK

SYS$ASSIGN

Assigns a channel to a device.

Format

status = SYS$ASSIGN(*devnam*, *chan*, *[acmode]*, *[mbxnam]*)

Arguments

devnam

OpenVMS usage:	**device_name**
type:	character_coded text string
access:	read only
mechanism:	by descriptor-fixed length string descriptor

Address of a character string descriptor pointing to the device name string (TCP0:).

chan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	write only
mechanism:	by reference

Address of a word into which SYS$ASSIGN writes the channel number.

acmode

OpenVMS usage:	**access_mode**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Optional access mode associated with the channel. The most privileged access mode used is that of the caller.

mbxnam

OpenVMS usage:	**device_name**
type:	character-coded text string
access:	read only
mechanism:	by descriptor-fixed length string descriptor

Optional logical mailbox associated with the device. (Not supported by TCPDRIVER.)

Status

See HP's *VMS System Services Reference Manual* for a complete list of status messages.

SYS$CANCEL

Cancels any I/O that is pending on a socket. The I/O will be completed with an *iosb* status of SS$_CANCEL.

Outstanding I/O operations are automatically cancelled at image exit.

Format

status = SYS$CANCEL(*chan*)

Argument

chan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	read only
mechanism:	by value

Number of the channel to be canceled.

Status

See HP's *VMS System Services Reference Manual* for a complete list of status messages.

SYS$DASSGN

Releases a channel.

When you deassign a channel, any outstanding I/O is completed with an *iosb* status of SS$_CANCEL. If a connection is open on the channel, it is aborted.

I/O channels are automatically deassigned at image exit.

Format

status = SYS$DASSGN(*chan*)

Argument

chan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	read only
mechanism:	by value

Number of the channel to be deassigned.

Status

See HP's *VMS System Services Reference Manual* for a complete list of status messages.

Sample Programs

C Programs

The following sample programs are included in the TCPWARE_COMMON:[TCPWARE.EXAMPLES] directory:

- TCPDRIVER_CLIENT.C
- TCPDRIVER_SERVER.C
- FINGER.C

The first two pair of programs shows the use of stream sockets with SYS$QIO system service calls to the TCPDRIVER. They are functionally the same as the TCP_SOCKET_CLIENT.C and TCP_SOCKET_SERVER.C socket library programs.

The client sequence of operations is as follows:

1 Assigns an I/O channel to TCP0, using SYS$ASSIGN.

2 Opens a connection, using (IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP).

3 Exchanges data, using IO$_WRITEVBLK and IO$_READVBLK.

4 Closes the connection, using (IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN).

5 Performs additional reads until the peer returns an SS$_VCCLOSED status.

6 Deassigns the channel, using SYS$DASSGN.

7 The server sequence of operations is as follows:

Assigns an I/O channel to TCP0, using SYS$ASSIGN.

Causes a passive open of the connection, using (IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP | 0x0800). You can alternately use TCPDRIVER's IO$_CREATE function that listens and accepts connections with UNIX-like functionality (use the /DEFINE=USE_CREATE command in the **CC** line of the build process).

Exchanges data, using IO$_WRITEVBLK and IO$_READVBLK.

Closes the connection, using (IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN).

Deassigns the channel, using SYS$DASSGN.

The FINGER.C file for the FINGER protocol also contains sample source code using TCPDRIVER services.

FINGER

Link the files to the TCPware Socket Library as follows for Alpha, VAX and I64 systems for DEC C:

```
$ LINK FINGER SYS$INPUT/OPTIONS SYS$SHARE:TCPWARE_SOCKLIB_SHR/SHARE
$ Ctrl/Z
```

To build on a VAX with VAX C:

```
$ CC FINGER.C
$ LINK FINGER,SYS$INPUT/OPTIONS
     TCPWARE:UCX$IPC.OLB/LIBRARY
     SYS$SHARE:VAXCRTL/SHARE
$ Ctrl/Z
$ FINGER :== $<device:[directory]>FINGER
```

To build on an Alpha, VAX or I64 system with DEC C:

```
$ CC/PREFIX=ALL FINGER.C
$ LINK FINGER
$ FINGER :== $<device:[directory]>FINGER
```

Then type:

```
$ FINGER username@host
```

or

```
$ FINGER @host
```

FINGERD

To build on a VAX with VAX C:

```
$ CC FINGERD.C
$ LINK FINGERD,SYS$INPUT/OPTIONS
     TCPWARE:UCX$IPC.OLB/LIBRARY
     SYS$SHARE:VAXCRTL.EXE/SHARE
```

To build on an Alpha,VAX or I64 system with DEC C:

```
$ CC/PREFIX=ALL FINGERD.C
$ LINK FINGERD,SYS$INPUT/OPTIONS
    SYS$SHARE:TCPWARE_SOCKLIB_SHR/SHARE
```

Use the NETCU ADD SERVICE and (optionally) the ADD ACCESS commands to "start" the server. The ADD ACCESS commands can be used to restrict access to the FINGER server to specific hosts or networks. For example:

```
$ NETCU ADD SERVICE FINGER TCP <path>FINGERD -
/PRIV=(NOSAME,NETMBX,TMPMBX,WORLD,SYSPRV) /ACCESS=<n>
(and other qualifiers as you may want)
```

Add these commands to the TCPWARE:SERVERS.COM file so that the server starts whenever TCPware starts.

FORTRAN Program

The TCPWARE_COMMON:[TCPWARE.EXAMPLES] directory also includes a TCPSAMPLE.FOR FORTRAN language file that transmits or receives data. This program links to the TCPware Socket Library as follows:

ALPHA and I64

```
$ FORTRAN/NOALIGN TCPWARE_COMMON:[TCPWARE.EXAMPLES]TCPSAMPLE
$ LINK TCPSAMPLE, SYS$INPUT/OPTIONS
    SYS$SHARE:TCPWARE_SOCKLIB_SHR/SHARE
Ctrl/Z
```

VAX

```
$ FORTRAN TCPWARE_COMMON:[TCPWARE.EXAMPLES]TCPSAMPLE
$ LINK TCPSAMPLE, SYS$INPUT/OPTIONS
    SYS$SHARE:TCPWARE_SOCKLIB_SHR/SHARE
Ctrl/Z
```

Chapter 4 UDPDRIVER Services

Introduction

This chapter describes the User Datagram Protocol (UDP) device driver (UDPDRIVER) services. It describes the user interface of TCPware for the OpenVMS UDP implementation only. RFC 768 contains the protocol specification for UDP.

UDP allows an application on one machine to send a datagram to an application on another machine. The datagram includes a protocol port number that identifies the receiving application from among other applications executing on the remote machine.

The UDP protocol is transaction-oriented so it does not guarantee reliable delivery of data. If your applications require ordered and reliable delivery of data, you should use TCP.

UDP services are available through the OpenVMS Queue I/O (SYS$QIO and SYS$QIOW) system services. These system services:

- Open and close a port.
- Send and receive data over the port.
- Perform status checks on the port.

SYS$QIOW is the synchronous and SYS$QIO the asynchronous form of VMS system services. Use each form depending on your application's requirements.

See the appropriate OpenVMS documentation for details on the OpenVMS I/O system services and related services such as asynchronous system traps (ASTs) and event flags.

Note! The QIO calls described in this chapter are used for direct access to the UDPDRIVER. If you are porting an application that uses the BGDRIVER or INETDRIVER QIO interface, there may be no need to make modifications. Use TCPware's BGDRIVER (see Chapter 2) or INETDRIVER (see Chapter 7).

Sequence of Operations

Perform the following sequence of operations to open a port:

1 Assign an I/O channel to the UDP0: device using the SYS$ASSIGN system service. SYS$ASSIGN creates a new device unit and assigns to it the channel for the port.

2 Open the port with the IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP function of the SYS$QIO or SYS$QIOW system services.

3 Perform read requests with the IO$_READVBLK function and write requests with the IO$_WRITEVBLK function as desired.

4 Close the port with the IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN function.

5 Deassign the I/O channel with the SYS$DASSGN system service.

Other Operations

In addition to the sequence of operations described in the preceding section, UDPDRIVER includes other operations that:

Read a datagram immediately with	IO$_READVBLK \| IO$M_NOW
Read the next received message without removing it from the received message queue with	IO$_READVBLK \| IO$M_DATACHECK
Suppress checksum generation at the UDP level with	IO$_WRITEVBLK \| IO$M_NOFORMAT
Request a list of buffers to be sent as a single datagram with each request with	IO$_WRITEVBLK \| IO$M_EXTEND
Read the open ports status with	IO$_SENSEMODE
Read the port characteristics with	IO$_SENSEMODE \| IO$M_CTRL
Read the UDP counters with	IO$_SENSEMODE \| IO$M_RD_COUNT
Clear these UDP counters after they have been read with	IO$_SENSEMODE \| IO$M_RD_COUNT | IO$M_CLR_COUNT You must have OPER privilege to clear the counters.
Cancel any pending I/O requests with	SYS$CANCEL

User Datagram Protocol Implementation Notes

There are no implementation specific restrictions for the User Datagram Protocol (UDP). The material presented here does not explain or describe the UDP protocol.

UDPDRIVER System Service Call Format

The format for the UDPDRIVER SYS$QIO system service call is as follows:

status= SYS$QIO*[*W*]*(*efn*, *chan*, *func*, *iosb*, *astadr*, *astprm*, *p1*, *p2*, *p3*, *p4*, *p5*, *p6*)

SYS$QIO and SYS$QIOW are used to issue UDP functions.

SYS$QIO is for asynchronous service completion, specifying that the service return to the caller immediately after queuing the I/O request.

SYS$QIOW is for synchronous service completion, specifying that the service place the calling process in a wait state and only return to the caller after completing the I/O operation.

The OpenVMS IODEF module provides definitions for the SYS$QIO function codes.

Note! The vertical bar (|) used in some of the functions described in this chapter is the C bit-wise inclusive OR operator.

UDPDRIVER System Service Call Arguments

You invoke UDPDRIVER system service calls with the standard OpenVMS QIO mechanism.

See the appropriate OpenVMS documentation (for example, the *Introduction to VMS System Services* volume) for details on the QIO mechanism.

The following sections describe each system call argument.

efn

OpenVMS usage:	**ef_number**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Number of the event flag set by completion of the I/O operation. The argument is optional.

chan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel assigned to the UDP device to which you are directing the request.

func

OpenVMS usage:	**function_code**
type:	longword (unsigned)
access:	read only

mechanism:	by value

Device-specific function code and the modifier, if appropriate, for each operation.

Note! *UDPDRIVER System Service Call Function Codes* describes each *func*.

iosb

OpenVMS usage:	**io_status_block**
type:	quadword (unsigned)
access:	write only
mechanism:	by reference

I/O status block that receives the final completion status of the I/O operation, structured as in Figure 4-1.

Figure 4-1 I/O Status

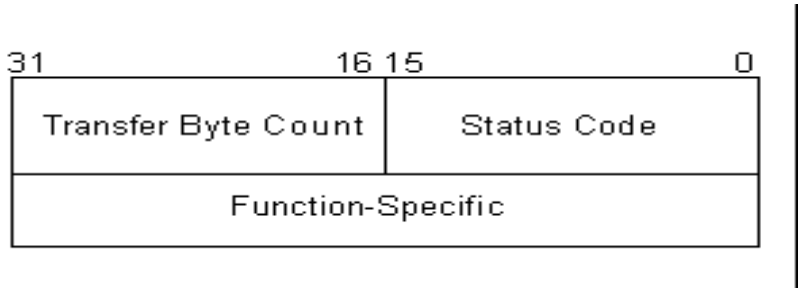


Table 4-1 describes the status block fields in more detail.

Table 4-1 I/O Status Block Fields

Field Name	**Description**
Transfer Byte Count	Number of bytes of data transferred in the I/O operation.
Status Code	SS$ status code or special error status code. If the low bit (0) of the OpenVMS error code is clear, the network has returned an error.
Function-Specific	Varies for each function code.

astadr

OpenVMS usage:	**ast_procedure**
type:	procedure entry mask
access:	call without stack unwinding
mechanism:	by reference

Address of the asynchronous system trap (AST) routine executed when the I/O is completed.

astprm

OpenVMS usage:	**user_arg**
type:	longword (unsigned)
access:	read only
mechanism:	by value

AST parameter to be passed to the AST routine.

p1 to p6

OpenVMS usage:	**varying_arg**
type:	longword (unsigned)
access:	read only or write only
mechanism:	by reference or by value

Function-specific parameters, as described for each function.

UDPDRIVER System Service Call Function Codes

System service call function codes specify what action the QIO performs. This section describes the following function codes:

IO$_READVBLK	IO$_WRITEVBLK \| IO$M_EXTEND

IO$_WRITEVBLK	IO$_SENSEMODE \| IO$M_RD_COUNT
IO$_SENSEMODE	IO$_SETMODE \| IO$M_CTRL | IO$M_STARTUP
IO$_SETMODE \| IO$M_CTRL	IO$_SETMODE \| IO$M_CTRL | IO$M_SHUTDOWN
IO$_SENSEMODE \| IO$M_CTRL	

IO$_READVBLK

Receives a datagram. The received datagram is written to the specified user buffer.

Use the IO$M_NOW modifier to execute the function immediately regardless of the amount of available data. If no data is available, the function returns SS$_NODATA status.

Use the IO$M_DATACHECK modifier with a read request to peek at the next received message without removing it from the received message queue.

Unless you set a local internet address (see P2 Set Port Characteristics), datagrams sent to any local interface address, any local broadcast address (TCPware supports both the standard -1 and nonstandard 0 broadcast address forms), or any joined multicast host group address are eligible to be received.

Note! UDPDRIVER holds a maximum of five unsolicited datagrams. It is good programming practice to have a read request pending for incoming datagrams.

Format

status=SYS$QIO(***efn, chan,* **IO$_READVBLK,*iosb, astadr, astprm,* ***buffer, size, udp_address,*** **0, 0,** ***receive_timeout*****)**

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	write only
mechanism:	by reference

Address of the user's buffer that receives the data.

p2=size

OpenVMS usage:	**word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by value

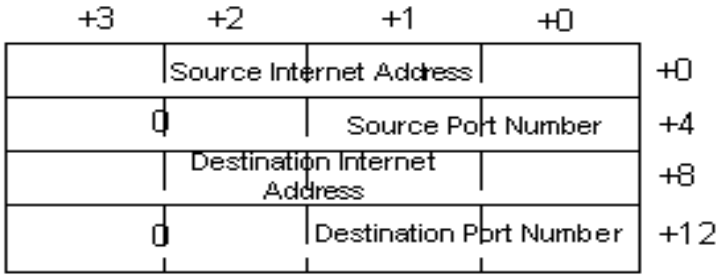
Length (in bytes) of the buffer to which the *address* argument points. This is the amount of data the user is willing to receive. The value must be at least 20 bytes.

p3=udp_address

OpenVMS usage:	**vector_longword_unsigned**
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Address of an optional 4-longword buffer. Receives the source and destination IP addresses and UDP port numbers from the received datagrams. Figure 4-2 shows the buffer format.

Figure 4-2 Read Function P3 Buffer Format



Note! IO$_READVBLK returns the source and destination internet addresses in internet byte order, which is reversed from the normal VMS byte order. For example, internet address 2.3.4.5 is stored as ^X05040302.

p6=receive_timeout

OpenVMS usage:	**longword_unsigned**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Contains the receive time-out time (in seconds). If this value is non-zero, the SS$_TIMEOUT status is returned if a datagram isn't received within this time.

Status

SS$_ABORT	Request aborted due to closed connection
SS$_BUFFEROVF	User's buffer too small for entire datagram Truncated datagram is returned Remainder of datagram is lost
SS$_CANCEL	Request cancelled
SS$_DEVINACT	Device is not active or port not opened
SS$_NODATA	No datagram available and IO$M_NOW was specified
SS$_NORMAL	Success Datagram received
SS$_THIRDPARTY	Port closed by third party Usually indicates that TCPware was shut down.
SS$_TIMEOUT	Receive time out time specified and no datagram received within allowed time

The number of bytes received are returned in the high-order word of the first longword of the I/O status block. This may be less than the bytes requested.

IO$_SENSEMODE

Reads the active ports status and returns status information for all of the open UDPDRIVER ports.

Format

status=SYS$QIO(*efn*, *chan*, **IO$_SENSEMODE,** *iosb*, *astadr*, *astprm*, ***buffer, address*, 0, 0, 0, 0)**

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	write only
mechanism:	by reference

Optional address of the 8-byte device characteristics buffer. The data returned is: the device class (DS$_SCOM) in the first byte, the device type (0) in the second byte, and the default buffer size, which is the maximum datagram size, in the high-order word of the first longword. IO$_SENSEMODE returns the second longword as 0.

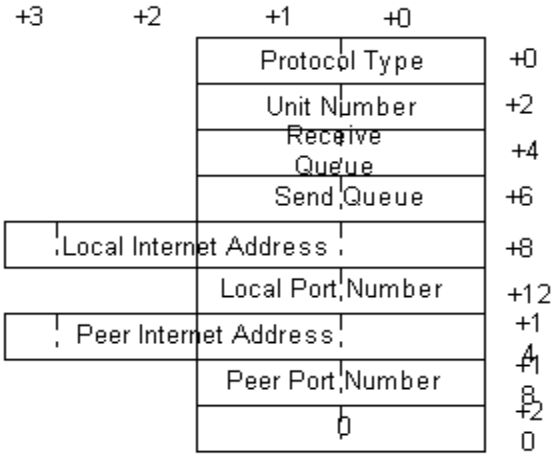
p2=address

OpenVMS usage:	**vector_word_unsigned**
type:	word (unsigned)
access:	write only
mechanism:	by descriptor

Address of the descriptor for the buffer to receive the status information on the open ports. The buffer receives 22 bytes of information for each open UDP port. Figure 4-3 shows the 22 bytes of status information returned.

Figure 4-3 P2 Status Information

Protocol type	Word value is 3 for UDPDRIVER ports, 4 for INETDRIVER datagram sockets, and 5 for BGDRIVER datagram sockets.
Unit number	Word value is the UDPDRIVER, INETDRIVER or BGDRIVER device unit number for the port.
Receive queue	Word value is the number of bytes received on the port waiting to be delivered to the user (via the read function).
Send queue	Word value is the number of bytes waiting to be transmitted for the port.
Local internet address	Longword value is the local internet address specified for the opened port. 0 means that none was specified.
Local port number	Word value is the local port number for the opened port.
Peer internet address	Longword value is the peer's internet address for a fully specified opened port. 0 means the port is not fully specified.
Peer port number	Word value is the peer's port number for a fully specified opened port. 0 means the port is not fully specified.



Status

SS$_BUFFEROVF	Buffer too small for all connections Truncated buffer returned

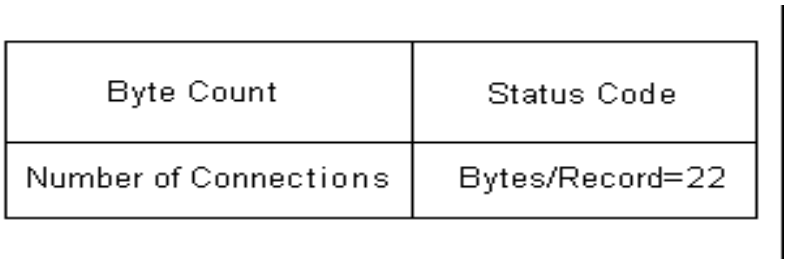
SS$_DEVINACT	Device not active Contact your system manager to determine why TCPware (or UPDRIVER) was not started
SS$_NORMAL	Success Status information returned

The byte count for the status information buffer is returned in the high-order word of the first longword of the I/O status block. This may be less than the bytes requested. See Figure 4-4 for more information.

The size in bytes of each connection's record (22 bytes) is returned in the low order word of the second longword of the I/O status block.

The total number of active connections is returned in the high-order word of the second longword of the I/O status block. This can be greater than the number of reported connections if the buffer is full.

Figure 4-4 I/O Status Block



IO$_SENSEMODE | IO$M_CTRL

Reads the port characteristics for the channel.

Format

status=SYS$QIO(*efn*, *chan*, **IO$_SENSEMODE | IO$M_CTRL,***iosb*, *astadr*, *astprm*,***buffer, address,* 0, 0, 0, 0)**

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	write only
mechanism:	by reference

Optional address of the 8-byte device characteristics buffer. The data returned is: the device class (DC$_SCOM) in the first byte, the device type (0) in the second byte, and the default buffer size, which is the maximum datagram size, in the high-order word of the first longword. IO$_SENSEMODE returns the second longword as 0.

p2=address

OpenVMS usage:	**vector_word_unsigned**
type:	word (unsigned)
access:	write only
mechanism:	by descriptor

Address of the descriptor for the extended characteristics buffer to receive the characteristics. The information returned in the buffer is formatted the same as the extended characteristics buffer used to set the connection characteristics. See the IO$_SETMODE function description for more information.

If bit 12 (mask 4096) is set in the parameter identifier (PID), the PID is followed by a counted string. If bit 12 is clear, the PID is followed by a longword value. While TCPware currently never returns a counted string for a parameter, this may change in the future.

Table 4-2 lists the port characteristics that the function returns.

Table 4-2 P2 Port Characteristics

PID	Meaning
0	**Local internet or multicast address.** Internet or multicast address is returned as the longword value. Last specified value or 0 is returned.
1	**Local port number.** Port number is returned in the low order word of the longword value. Local port number is either the default port number or the last specified local port number.
2	**Peer internet address.** Internet address is returned as the longword value. Last specified value or 0 is returned.
3	**Peer port number.** Port number is returned in the low order word of the longword value. Last specified value or 0 is returned.
5	**Port datagram size.** Maximum datagram size (in bytes) is returned as the longword value.
6	**Port state.** Longword value is 0 if the port is not open and 1 if the port is open.
7	**Shared access control.** Longword value is 0 if the port is set for exclusive access and 1 if the port is set for shared access.
10	**Socket options.** For the values of this option, see the socket options entry in P2 Set Port Characteristics .
14	**IP TTL.** Longword. Low byte contains the time-to-live (TTL).
15	**IP TOS. Longword.** Low byte of the longword contains the type of service (TOS).
18	**Multicast interface.** Longword. IP address of the local interface to use in sending multicast datagrams. If 0, the default interface is used.
19	**Multicast TTL.** Longword. Low byte contains the time-to-live (TTL) to be used in multicast datagrams to be transmitted.
20	**Multicast loopback.** Longword. Low byte is 0 to disable and 1 to enable local loopback of multicast datagrams.

Note! The order in which IO$_SENSEMODE | IO$M_CTRL returns the parameters can change. It is recommended that you search the buffer for the desired parameter.

Status

SS$_BUFFEROVF	Buffer too small for all connections Truncated characteristics buffer returned
SS$_DEVINACT	Device not active Contact your system manager to determine why TCPware (or UPDRIVER) was not started
SS$_NORMAL	Success Characteristics returned

The byte count for the characteristics buffer is returned in the high-order word of the first longword of the I/O status block. This may be less than the bytes requested.

The number of bytes in the receive queue is returned in the low order word of the second longword in the I/O status block.

Note! You can use the SYS$GETDVI system service to obtain the local port number, peer port number, and peer internet address. The DEVDEPEND field stores the local port number (low order word) and peer port number (high-order word). The DEVDEPEND2 field stores the peer internet address.

IO$_SENSEMODE | IO$M_RD_COUNT

Reads the UDP counters. You can add the IO$M_CLR_COUNT modifier to this function to zero the counters after they are read. However, you must have the OPER privilege to use this modifier.

Format

status=SYS$QIO(***efn, chan,***IO$_SENSEMODE | IO$M_RD_COUNT, *iosb, astadr, astprm,* **0, *address,* 0, 0, 0, 0)**

Argument

p2=address

OpenVMS usage:	**vector_longword_unsigned**
type:	longword (unsigned)
access:	write only
mechanism:	by descriptor

Address of the descriptor for the buffer that receives the counters. These counters relate only to the UDP level and are for all ports since the counters were last zeroed. The counters in the order they are returned:

1 Number of seconds since last zeroed

2 Number of datagrams transmitted

3 Number of datagrams transmitted with transmission errors

4 Number of datagrams received

5 Number of datagrams received in error (invalid UDP header or checksum)

6 Number of datagrams that were discarded because they could not be delivered to a receiver

All counters are longword values. All counters stop at their maximum value instead of overflowing.

Status

SS$_BUFFEROVF	Buffer too small for all connection Truncated buffer returned
SS$_DEVINACT	Device not active Contact your system manager to determine why TCPware (or UPDRIVER) was not started
SS$_NORMAL	Success, Counters returned

IO$_SETMODE | IO$M_CTRL

Sets the port characteristics in the extended characteristics buffer.

The buffer consists of a series of six-byte or counted string entries. The first word of each entry contains the parameter identifier (PID) of a characteristic, followed by either a longword that contains the value of that characteristic, or a counted string. Counted strings consist of a word with the size of the character string, followed by the character string.

Format

status=SYS$QIO(*efn*, *chan*, **IO$_SETMODE | IO$M_CTRL,***iosb*, *astadr*, *astprm*, **0, *address*, 0, 0, 0, 0)**

Argument

p2=address

OpenVMS usage:	**vector_word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by descriptor

The *address* argumentis the address of the extended characteristics buffer's descriptor. Figure 4-5 shows the format of the extended characteristics buffer and its descriptor.

Figure 4-5 P2 Set Characteristics Buffer

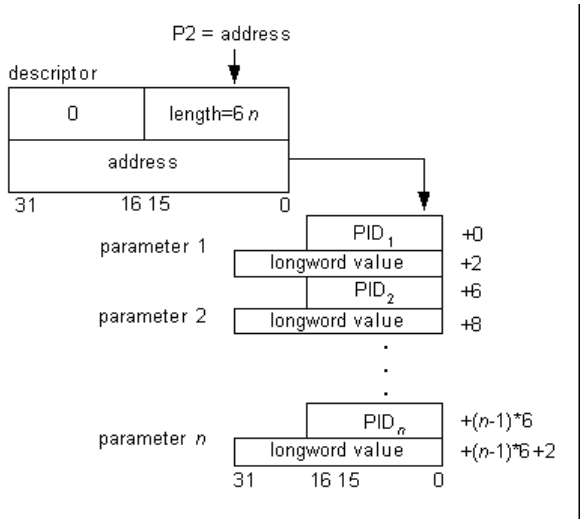


Table 4-3 lists the port characteristics you can set. The characteristics remain set, even after you close the port, until you change them with another IO$_SETMODE operation.

Table 4-3 P2 Set Port Characteristics

PID	Meaning
0	**Local internet address.** Longword with the local host's internet or multicast address. If used, must be a valid local internet or multicast address. Set this parameter only if the port is not open. Specify in internet byte order, reversed from the normal VMS byte order. For example, internet address 2.3.4.5 is stored as ^X05040302. To determine if an internet address is local, issue an IO$_SETMODE \| IO$M_CTRL function specifying the address in this characteristic. If the address is not local, the function returns an `SS$_BADPARAM` status.
1	**Local port number.** Low order word of the longword containing the port number. If omitted, the function uses the unique port number generated when you assign the UDP device unit. Set this parameter only if the port is not open. If you use `0`, the function generates a new, unique number. Specify in normal VMS byte order.
2	**Peer internet address.** Longword with the peer host's internet address. If you specify both the peer's internet address and port number after opening the port (or when the port is opened) the port is considered fully specified (or opened as fully specified) and all further receive requests receive datagrams from that address and port only. Transmitted requests that do not provide a source/destination buffer are sent to that address and port. Specify in internet byte order, as the local internet address characteristic above. Using `0` functionally omits the address.
3	**Peer port number.** Low order word of the longword containing the port number. (See PID 2 above.) Specify in normal VMS byte order. Using `0` functionally omits the port number.
7	**Shared access control.** Longword that specifies whether others can (shared access) or cannot (exclusive access) use the same local port number. For shared access (used primarily by servers), the value must be non-zero. You can open any number of ports on the local port number. Received datagrams are placed in a common queue so that the first receive on any of these opened ports returns the next datagram. For exclusive access (the default), the longword value must be `0`.
9	**Change ownership.** Longword that allows the current owner of the device unit to prepare to pass ownership to another process. Removes the owner's process ID from the connection. If the value field is non-zero, specifies the new owner UIC for the connection and changes the connection protection so that only system and owner have access (S:RWLP,O:RWLP). Changes the following fields for the connection: owner process ID, owner UIC, and device protection. Primarily the NETCP master server but also other programs use it.

10	**Set socket options.** Low order word of the longword value field containing the mask value of the socket options to set. Specify the following options, as defined in the TCPWARE_INCLUDE:SOCKET.H header file (the values are in hexadecimal): SO_DEBUG, with a value of 0001, enables debugging (ignored) SO_REUSEADDR, with a value of 0004, allows reuse of local address (controllable through the passive open access control parameter SO_KEEPALIVE, with a value of 0008, keeps connections alive SO_DONTROUTE, with a value of 0010, prevents routing (ignored) SO_BROADCAST, with a value of 0020, enables use of broadcasts (ignored) SO_USELOOPBACK, with a value of 0040, bypasses hardware when possible (ignored)
11	**Clear socket options.** Low order word of the longword value field containing the mask of the socket options to clear. (See the options under PID 10 above.)
13	**IP options.** Counted string containing the IP options to be included in datagrams.
14	**IP TTL.** Low byte of the longword containing the time-to-live (TTL) for datagrams.
15	**IP TOS.** Low byte of the longword containing the type of service (TOS) for datagrams.
16	**Join multicast group.** Counted string of eight bytes of which the first longword contains the multicast group IP address to join, and the last longword contains the IP address of the local interface on which to join the group. If `0`, the default interface is used.
17	**Leave multicast group.** Counted string of eight bytes of which the first longword contains the multicast group IP address to leave, and the last longword contains the IP address of the local interface from which to leave the group. If `0`, the default interface is used.
18	**Multicast interface.** Longword with the IP address of the local interface to use in sending multicast datagrams. If `0`, the default interface is used.
19	**Multicast TTL.** Longword of which the low byte contains the time-to-live (TTL) for multicast datagrams transmitted.
20	**Multicast loopback.** Longword of which the low byte is `0` to disable, or `1` to enable local loopback of multicast datagrams.

Status

SS$_BADPARAM	Bad parameter or value specified PID returned in low order word of second longword
SS$_DEVACTIVE	Port is open, and you made a request to change a parameter that cannot be changed when the port is open
SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_DUPUNIT	Multicast group address already joined
SS$_INSFMEM	Insufficient memory to complete request
SS$_IVBUFLEN	Extended characteristics buffer length invalid
SS$_NOPRIV	Setting IP layer options (PIDs 13 through 20) requires privileges Application must either be running under SYSTEM UIC or have SYSPRV, BYPASS, or OPER privilege
SS$_NORMAL	Success Characteristics set
SS$_TOOMUCHDATA	Too many multicast group addresses specified
SS$_UNREACHABLE	Default interface for multicast group address could not be found Specify local interface's IP address when joining multicast group

IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN

Closes a previously opened port.

After you issue this modifier any received datagrams are discarded.

Format

status=SYS$QIO(*efn*, *chan*, **IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN,** *iosb*, *astadr*, *astprm*, **0, 0, 0, 0, 0, 0***)*

Arguments

None.

Status

SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_NORMAL	Success Port is now (or was) closed Port can no longer receive datagrams; however, datagrams can still be sent

IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP

Opens a new receive port. When opening a new receive port, you have a choice of either fully or partially specifying the port characteristics.

To open a fully specified port, you must specify both the peer's internet address and port number. UDPDRIVER then delivers only datagrams received from that address and port and destined for the local port number.

To open a partially specified port, do not specify the peer's internet address and port number. UDPDRIVER then delivers all datagrams destined for the local port number, except those to be delivered to fully specified ports for the same local port number. You can convert a partially specified port into a fully specified port by issuing a IO$_SETMODE | IO$M_CTRL function and specifying both the peer's internet address and port number.

Format

status=SYS$QIO(*efn*, *chan*, **IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP,** *iosb*, *astadr*, *astprm*, **0,** ***address*, 0, 0, 0, 0)**

Argument

p2=address

OpenVMS usage:	**vector_word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by descriptor

Address of the descriptor for the extended characteristics buffer. See the previous IO$_SENSEMODE | IO$M_CTRL function description for details on the extended characteristics buffer.

Note! You only need to open ports if you are to receive datagrams. You can send datagrams whether or not you open the port.

Status

SS$_BADPARAM	Bad parameter or value specified PID returned in low order word of second longword
SS$_DEVACTIVE	Port is already open
SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_DUPUNIT	Specified port number is already in use by another user, and that user or you did not request shared access
SS$_NOPRIV	Insufficient privilege Program tried to open local port below 1024, but program does not have BYPASS or SYSPRV privilege
SS$_NORMAL	Success Port is open and ready for receives
SS$_THIRDPARTY	Port closed by third party Usually indicates that TCPware was shut down.

IO$_WRITEVBLK

Sends a datagram stored in a single user buffer.

The IO$M_NOFORMAT modifier suppresses checksum generation at the UDP level.

Format

status=SYS$QIO(*efn, chan*,**IO$_WRITEVBLK,** *iosb, astadr, astprm*,***buffer, size, address,* 0, 0, 0*)**

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	read only
mechanism:	by reference

Address of the buffer that contains the data to be sent.

p2=size

OpenVMS usage:	**word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by value

User's buffer size in bytes (the byte count), which is the amount of data that the user is sending. The value should not be greater than 64000 bytes.

p3=address

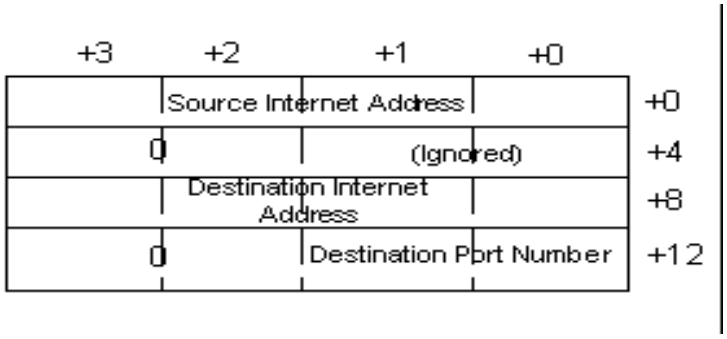
OpenVMS usage:	**longword_unsigned**
type:	longword (unsigned)
access:	read only

mechanism:	by value

Address of the optional 4-longword source/destination buffer. The source and destination internet address and port number specified in this buffer are used to transmit the datagram.

Figure 4-6 shows the buffer format.

Figure 4-6 Write Function Address Buffer Format



You can use any destination port. You can specify the source internet address, which must be a valid local address, or specify it as 0. When you specify the address as 0, UDPDRIVER supplies the appropriate local internet address. You must specify internet addresses in internet byte order, which is reversed from the normal VMS byte order. For example, internet address 2.3.4.5 is stored as ^X05040302.

If you do not specify the address buffer or if fields in this buffer are 0, UDPDRIVER uses the port's characteristic values set using the IO$_SETMODE | IO$M_CTRL or IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP functions for the unspecified fields.

To broadcast a datagram, you must either specify the source address and a destination address of 255.255.255.255, or specify a local network's broadcast address. The local network's broadcast address is an address with all bits set (or cleared) for the host number portion of the internet address.

Status

SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_IVBUFLEN	User's buffer too large
SS$_NORMAL	Success Datagram was transmitted
SS$_OPINCOMPL	Datagram not transmitted because no destination internet address or port number was specified by source/destination buffer or port's characteristics

SS$_THIRDPARTY	TCPware shut down
SS$_UNREACHABLE	No route to specified destination internet address

The number of bytes sent are returned in the high-order word of the first longword of the I/O status block.

IO$_WRITEVBLK | IO$M_EXTEND

Sends a datagram stored in an array of buffers.

The IO$M_NOFORMAT modifier suppresses checksum generation at the UDP level.

Format

status=SYS$QIO(***efn, chan,* **IO$_WRITEVBLK | IO$M_EXTEND,*iosb, astadr, astprm,***msghdr, size, 0, 0, 0, 0***)**

Arguments

p1=msghdr

OpenVMS usage:	**vector_longword_unsigned**
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of the `msghdr` structure. Points to the address of the structure storing the array of additional structures. Each of these array structures contains the address and size of one of the user buffers.

p2=size

OpenVMS usage:	**longword_unsigned**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Size of the `msghdr` structure. The size must be 24 bytes.

For details on the `msghdr` structure and an example of using the scatter-gather write function, see Chapter 3, *TCPDRIVER Services*.

Status

SS$_DEVINACT	Device not active Contact system manager why TCPware (or TCPDRIVER) not started
SS$_IVBUFLEN	User's buffer too large
SS$_NORMAL	Success Datagram was transmitted
SS$_OPINCOMPL	Datagram not transmitted because no destination internet address or port number was specified by source/destination buffer or port's characteristics
SS$_THIRDPARTY	TCPware shut down
SS$_UNREACHABLE	No route to specified destination internet address

The number of bytes sent are returned in the high-order word of the first longword of the I/O status block.

SYS$ASSIGN

Assigns a channel to a device.

Format

status = SYS$ASSIGN(*devnam*, *chan*, *[acmode]*, *[mbxnam]*)

Arguments

devnam

OpenVMS usage:	**device_name**
type:	character_coded text string
access:	read only
mechanism:	by descriptor-fixed length string descriptor

Address of a character string descriptor pointing to the device name string (UDP0:).

chan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	write only
mechanism:	by reference

Address of a word into which SYS$ASSIGN writes the channel number.

acmode

OpenVMS usage:	**access_mode**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Optional access mode associated with the channel. The most privileged access mode used is that of the caller.

mbxnam

OpenVMS usage:	**device_name**
type:	character-coded text string
access:	read only
mechanism:	by descriptor-fixed length string descriptor

Optional logical mailbox associated with the device.

Status

See Computer's *VMS System Services Reference Manual* for a complete list of status messages.

SYS$CANCEL

Cancels any I/O that is pending on a channel. The I/O will be completed with an *iosb* status of `SS$_CANCEL`.

Outstanding I/O operations are automatically cancelled at image exit.

Format

status = SYS$CANCEL(*chan*)

Argument

chan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	read only
mechanism:	by value

Number of the channel to be canceled.

Status

See HP's *VMS System Services Reference Manual* for a complete list of status messages.

SYS$DASSGN

Releases a channel.

When you deassign a channel, any outstanding I/O is completed with an *iosb* status of SS$_CANCEL.

I/O channels are automatically deassigned at image exit.

Format

status = SYS$DASSGN(*chan*)

Argument

chan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	read only
mechanism:	by value

Number of the channel to be deassigned.

Status

See HP's *VMS System Services Reference Manual* for a complete list of status messages.

Sample Programs

C Programs

The following sample programs, which show the use of SYS$QIO system service calls to the UDPDRIVER to set up a DISCARD client and server, are included in the TCPWARE_COMMON:[TCPWARE.EXAMPLES] directory:

- UDPDRIVER_CLIENT.C
- UDPDRIVER_SERVER.C

The client sequence of operations is as follows:

1 Assigns an I/O channel to UDP0, using SYS$ASSIGN.

2 Sends data using $QIO with IO$_WRITEVBLK.

3 Deassigns the channel, using SYS$DASSGN.

The server sequence of operations is as follows:

1 Assigns an I/O channel to UDP0, using SYS$ASSIGN.

2 Opens a receive port, using $QIO(IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP).

3 Exchanges data, using $QIO(IO$_WRITEVBLK) and $QIO(IO$_READVBLK).

4 Closes the port, using $QIO(IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN).

5 Deassigns the channel, using SYS$DASSGN.

To build any one of these applications using DEC C, enter:

```
$ CC/DECC/PREFIX=ALL filename
$ LINK filename
Ctrl/Z
```

To build any one of these applications using VAX C, enter:

```
$ CC/VAXC filename
$ LINK filename, TCPWARE:UCX$IPC/LIB, SYS$INPUT/OPTIONS-
_$  SYS$SHARE:VAXCRTL/SHARE
Ctrl/Z
```

FORTRAN Program

The TCPWARE_COMMON:[TCPWARE.EXAMPLES] directory also includes a UDPSAMPLES.FOR FORTRAN language file that transmits or receives data. This program links to the TCPware Socket Library as follows:

VAX

```
$ FORTRAN TCPWARE_ROOT:[TCPWARE.EXAMPLES]UDPSAMPLE
$ LINK UDPSAMPLE, SYS$INPUT/OPTIONS
    SYS$SHARE:TCPWARE_SOCKLIB_SHR/SHARE
Ctrl/Z
```

Alpha and I64

```
$ FORTRAN/NOALIGN TCPWARE_ROOT:[TCPWARE.EXAMPLES]UDPSAMPLE
$ LINK UDPSAMPLE, SYS$INPUT/OPTIONS
    SYS$SHARE:TCPWARE_SOCKLIB_SHR/SHARE
Ctrl/Z
```

Chapter 5 IPDRIVER Services

Introduction

This chapter describes the IP device driver (IPDRIVER) services. It describes both the user and external interfaces of this IP implementation:

- The user interface exists above the IP layer, de-multiplexing received datagrams, as described below.
- The external interface provides a method by which you can support four types of network interfaces. The program you create is responsible for sending and receiving raw IP datagrams over the network interface.

IP does not include end-to-end data reliability, flow control, sequencing, or other services usually found in host-to-host protocols. Since TCP does include these services, you should use TCP unless the applications programs intend to implement or do not require these services.

IP services are available through the OpenVMS Queue I/O (SYS$QIO and SYS$QIOW) system services. Functions are provided to open and close a port, and to transmit and receive datagrams.

SYS$QIOW is the synchronous and SYS$QIO the asynchronous form of VMS system services. Use each form depending on your application's requirements.

The IPDRIVER uses ports to demultiplex received datagrams. When an IP datagram is received, IPDRIVER validates the header and searches for a port opened on the protocol number indicated within the datagram's internet header. If no port is opened for that protocol or that port has no outstanding receive, IPDRIVER discards the datagram.

See Figure 5-1 for an illustration of the user interface and external interface working with IPDRIVER. The IP specifications are in RFC 791.

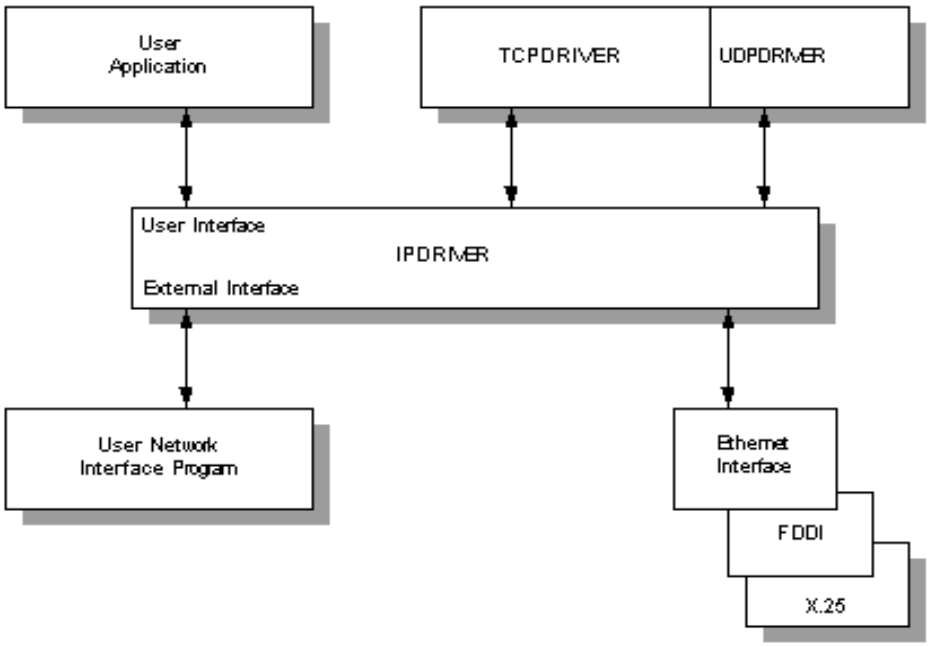
The external interface is intended primarily for sites that use proprietary network hardware. This interface lets you write programs that support the use of IPDRIVER with various network controllers. This chapter refers to these programs as Network Interface Programs.

Basically, the Network Interface Programs perform two functions:

- Delivery of received datagrams to IPDRIVER for processing.
- Receiving of datagrams from IPDRIVER for transmission.

These programs do not handle IP activities such as routing, fragmentation and reassembly, or datagram validation. IPDRIVER performs these activities.

Figure 5-1 IPDRIVER User and External Interfaces



Sequence of Operations

Perform the following sequence of operations to open a port:

1. Assign an I/O channel to IPA0: with the SYS$ASSIGN system service. SYS$ASSIGN creates a new device unit and assigns to it the channel for the port.
2. Open the port with the IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP function of the SYS$QIO or SYS$QIOW system service.
3. Perform read requests with the IO$_READVBLK function and write requests with the IO$_WRITEVBLK function to receive and transmit datagrams as desired.
4. Close the port with the IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN function.
5. Deassign the I/O channel with the SYS$DASSGN system service.

   See the appropriate OpenVMS documentation for information on the OpenVMS I/O system services (SYS$ASSIGN, SYS$CANCEL, SYS$DASSGN, SYS$QIO, and SYS$QIOW) and on related system services, such as asynchronous system trap (AST) and event flag services.

Other Operations

In addition to the sequence of operations described above, IPDRIVER includes other operations that:

Read the network device information, read the ARP table, or read the routing table with	IO$_SENSEMODE \| IO$M_CTRL
Read the IP counters with	IO$_SENSEMODE \| IO$M_RD_COUNT

Internet Protocol Implementation Notes

The material presented here does not explain or describe the Internet Protocol (IP).

TCPware for OpenVMS implements Version 4 of the IP, and the following restrictions apply:

- The type of service field is ignored.
- All options are ignored (options may be present in incoming datagrams). Options are copied to fragmented datagrams as required by the Internet Protocol specification.
- The TCPware host may send the following Internet Control Message Protocol (ICMP) messages:

`Destination unreachable`	When the datagram cannot be forwarded or the destination port is not active
`Echo`	In response to a echo request
`Parameter problem`	When you enable forwarding and the datagram contains a parameter error
`Redirect`	When you enable forwarding and the forwarded datagram exits by the same interface on which it was received (see the NETCU ENABLE FORWARDING command)
`Timestamp reply`	In response to a timestamp request

IPDRIVER System Service Call Format

The format for the IPDRIVER SYS$QIO system service call is as follows:

status = SYS$QIO*[W]*(*efn, chan, func, iosb, astadr, astprm, p1,p2,p3,p4,p5,p6*)

SYS$QIO or SYS$QIOW issues IP functions.

SYS$QIO is for asynchronous service completion, specifying that the service return to the caller immediately after queuing the I/O request.

SYS$QIOW is for synchronous service completion, specifying that the service place the calling process in a wait state and only return to the caller after completing the I/O operation.

The OpenVMS IODEF module provides definitions for the SYS$QIO function codes.

Note! The vertical bar (|) used in some of the functions described in this chapter is the C bitwise inclusive OR operator.

IPDRIVER System Service Call Arguments

You invoke IPDRIVER system service calls with the standard OpenVMS QIO mechanism.

See the appropriate OpenVMS documentation (for example, the Introduction to VMS System Services volume) for details on the QIO mechanism.

The following sections describe each system call argument.

efn

OpenVMS usage:	**ef_number**
type:	longword (unsigned)
access:	read only
mechanism	by value

Number of the event flag set by completion of the I/O operation. The argument is optional.

chan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	read only
mechanism:	by value

I/O channel assigned to the IP device to which you are directing the request.

func

OpenVMS usage:	**function_code**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Device-specific function code and the modifier, if appropriate, for each operation.

Note! The IPDRIVER User Interface System Service Call Function Codes and IPDRIVER External Interface sections describe each func in detail.

iosb

OpenVMS usage:	**function_code**
type:	longword (unsigned)
access:	read only
mechanism:	by value

I/O status block that receives the final completion status of the I/O operation, structured as in Figure 5-2.

Figure 5-2 I/O Status Block

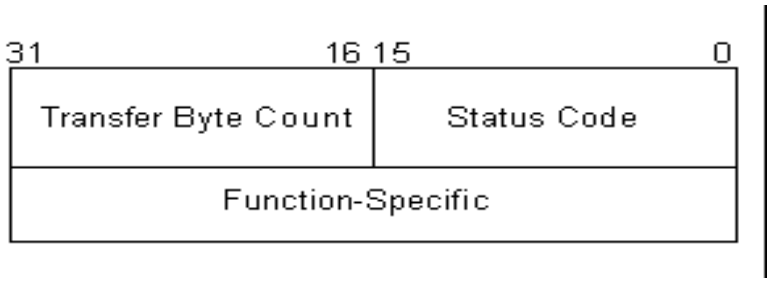


Table 5-1 describes the status block fields in more detail.

Table 5-1 I/O Status Block Fields

Field Name	**Description**
Function-Specific	Varies for each function code.
Status Code	SS$ status code or special error status code. If the low bit (0) of the OpenVMS error code is clear, the network returned an error.
Transfer Byte Count	Number of bytes of data transferred in the I/O operation.

astadr

OpenVMS usage:	**ast_procedure**
type:	procedure entry mask

access:	call without stack unwinding
mechanism:	by reference

Address of the asynchronous system trap (AST) routine executed when the I/O is completed.

astprm

OpenVMS usage:	**user_arg**
type:	longword (unsigned)
access:	read only
mechanism:	by value

AST parameter to be passed to the AST routine.

p1 to p6

OpenVMS usage:	**varying_arg**
type:	longword (unsigned)
access:	read only or write only
mechanism:	by reference or by value

Function-specific parameters, as described for each function.

IPDRIVER User Interface System Service Call Function Codes

System service call function codes specify what action the QIO performs. This section describes the following function codes for the User Interface:

IO$_READVBLK	IO$_SETMODE \| IO$M_CTRL

IO$_SENSEMODE \| IO$M_RD_COUNT	IO$_SETMODE \| IO$M_CTRL | IO$M_STARTUP
IO$_SENSEMODE	IO$_SETMODE \| IO$M_CTRL | IO$M_SHUTDOWN
IO$_SENSEMODE \| IO$M_CTRL	IO$_WRITEVBLK

IO$_READVBLK

Receives a datagram. The data received is written to the specified user buffer.

Note! IPDRIVER delivers ICMP messages received for the active protocol. It is up to the application to check the "protocol" field in the received datagram to determine if the received datagram is an ICMP message. IPDRIVER can deliver the following ICMP messages: destination unreachable, time exceeded, parameter problem, source quench, and redirect.

Note! A copy of a broadcast or multicast datagram is looped back by default. This means that your application receives its own broadcast and multicast datagrams.

Format

status = SYS$QIO(***efn, chan,***IO$_READVBLK,*iosb, astadr, astprm, **buffer, size,0, 0, 0, 0)**

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	write only
mechanism:	by reference

Address of the user's buffer that receives the datagram. This buffer must be large enough to store the internet header as well as the expected data. The maximum size of an internet header is 60 bytes.

p2=size

OpenVMS usage:	**word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by value

User's buffer size in bytes (the byte count). This is the amount of data the user is willing to receive (including the internet header). The value should not be greater than 65000 bytes.

The internet header contains the following fields:

Destination internet address (i.e., the local internet address)	Longword at an offset of 16 bytes into the header. Stored in internet byte order.
Identification	Word at an offset of 4 bytes into the header. Stored in internet byte order.
Options	Variable length buffer (internet header size - 20) at an offset of 20 bytes into the header.
Source internet address	Longword at an offset of 12 bytes into the header. Stored in internet byte order.
Time to Live	Byte at an offset of 8 bytes into the header.
Type of Service	Byte at an offset of 1 byte into the header.

See RFC 791 for details on the internet header.

Status

SS$ABORT	Request aborted due to closed connection
SS$_BUFFEROVF	User's buffer was too small for entire datagram Truncated datagram is returned Remainder of datagram is lost
SS$_CANCEL	Request cancelled
SS$_DEVINACT	Device not active or port was not opened
SS$_NODATA	No datagram is available and IO$M_NOW was specified
SS$_NORMAL	Success Datagram received

The number of bytes of the entire datagram (including the internet header) is returned in the high-order word of the first longword of the I/O status block. The size (in bytes) of the internet header is returned in the low-order word of the second longword. The size (in bytes) of the data within the datagram is returned in the high-order word of the second longword.

IO$_SENSEMODE | IO$M_CTRL

Performs the following functions:

- Reads extended characteristics
- Reads network device information
- Reads the ARP table
- Reads the routing table

Format

status = SYS$QIO(***efn, chan,* **IO$_SENSEMODE | IO$M_CTRL,*iosb, astadr, astprm,* ***buffer, address, function, line-id,*** **0, 0)**

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	write only
mechanism:	by reference

Optional address of the 8-byte device characteristics buffer. The data returned is the device class (DC$_SCOM) in the first byte, the device type (0) in the second byte, and the default buffer size (0) in the high-order word of the first longword. The second longword is returned as 0.

p2=address

OpenVMS usage:	**vector_word_unsigned**
type:	word (unsigned)
access:	write only
mechanism:	by descriptor

Address of the descriptor for the buffer to receive the information. The format of the buffer depends on the information requested. Each buffer format is described separately in the section that follows.

If bit 12 (mask 4096) is set in the parameter identifier (PID), the PID is followed by a counted string. If bit 12 is clear, the PID is followed by a longword value. While TCPware currently never returns a counted string for a parameter, this may change in the future.

p3=function

OpenVMS usage:	**longword-unsigned**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Code that designates the function. The function codes are shown in Table 5-2.

Table 5-2 P3 Function Codes

Code	**Function**
0	Read extended characteristics
1	Read network device information
2	Read routing table
3	Read ARP table
8	Read routing table (includes CIDR-related mask information)
9	Returns a record for each interface in the following format: Line Id Bytes Transmitted Bytes Received All values are longwords.

p3 function code 8 is a superset of function code 2. Both read the routing table, although the output is different. For details, see the Reading the R section.

p4=line-id

OpenVMS usage:	**longword-unsigned**
type:	longword (unsigned)
access:	read only

mechanism:	by value

Specify this argument only if you are reading a network device's ARP table function.

Reading Extended Characteristics

Use IO$_SENSEMODE | IO$M_CTRL with function=0 to read the extended characteristics. The information returned consists of one or more records, each record containing a word parameter identifier (PID) value followed by either a longword or a counted string. A counted string consists of a word length followed by the specified number of bytes. The parameters that can be returned are listed in Table 5-3.

Table 5-3 Extended Characteristics

PID	Meaning
14	**IP TTL.** Longword. Low byte of the longword contains the time-to-live (TTL).
15	**IP TOS.** Longword. Low byte of the longword contains the type of service (TOS).
18	**Multicast interface.** Longword. IP address of the local interface to use in sending multicast datagrams. If o, the default interface is used.
19	**Multicast TTL.** Longword. Low byte contains the time-to-live (TTL) to be used in sending multicast datagrams.
20	**Multicast loopback.** Longword. Low byte is 0 to disable and 1 to enable local loopback of multicast datagrams.

Reading Network Device Information

Use IO$_SENSEMODE | IO$M_CTRL with p3=1 to read network device information. The information returned in the buffer (specified by p2=address) can consist of multiple records. Each record consists of nine longwords, and one record is returned for each device.

The SHOW NETWORKS command in NETCU uses this function.

When you read network device information, the data in each record is returned in the order presented below. All are longword values.

1 Line id (see the description of the line-id argument)

2 Line's local internet address

3 Line's internet address network mask

4 Line's maximum transmission unit (MTU) in the low-order word, and the line flags in the high-order word

5 Number of packets transmitted (includes ARP packets for Ethernet lines)

6 Number of transmit errors

7 Number of packets received (includes ARP and trailer packets for Ethernet lines)

8 Number of receive errors

9 Number of received packets discarded due to insufficient buffer space

Reading the Routing Table

Use IO$_SENSEMODE | IO$M_CTRL with p3=2 or p3=8 to read the routing table. The information returned in the buffer (specified by p2=address) can consist of multiple records. Each record consists of five longwords, and one record is returned for each table entry.

The SHOW ROUTES command in NETCU uses this function.

The *p3*=8 function returns full routing information and is a superset of *p3*=2, which was retained for backwards compatibility with existing programs. *p3*=2 and *p3*=8 return the same table of routing entries, in the following order, except that *p3*=2 does not return items 7 and 8 (address mask and Path MTU):

1	**Destination internet address.**	Destination host or network to which the datagram is bound. Returned as a longword value.
2	**Gateway internet address.**	Internet address to which the datagram for this route is transmitted. Returned as a longword value.
3	**Flags.**	Routing table entry's flag bits. Returned as a word value: Mask 1, name GATEWAY, if set, the route is to a gateway (the datagram is sent to the gateway internet address). If clear, the route is a direct route. Mask 2, name HOST, if set, the route is for a host. If clear, the route is for a network. Mask 4, name DYNAMIC, if set, the route was created by a received ICMP redirect message. Mask 8, name AUTOMATIC, if set, this route was added by TCPWARE_RAPD process and will be modified or remoted by that process as appropriate. Mask 16, name LOCKED, if set, the route cannot be changed by an ICMP redirect message. Mask 32, name INTERFACE, if set, the route is for a network interface. Mask 64, name DELETED, if set, the route is marked for deletion (it is deleted when the reference count reaches 0). Mask 128, name POSSDOWN, if set, the route is marked as possibly down.
4	**Reference count.**	Number of connections currently using the route. Returned as a word value.
5	**Use count.**	Number of times the route has been used for outgoing traffic. Returned as a longword value.

6	**Line ID.**	Line identification for the network device used to transmit the datagram to the destination. See the description of the line-id argument later in this section for the line ID codes. Table 5-4 shows the line identification values
7	**Address mask.**	Address mask for the destination address. Returned as a longword value.
8	**Path MTU.**	Path maximum transmission unit. Returned as a longword value.

Table 5-4 Line ID Values

Line ID	**Line ID Value**
LPB-0	^X00000001
PRO-*n*	^X00*nn*0003
HYP-*n*	^X00*nn*0004
X25-*n*	^X00*nn*0006
UNA-*n*	^X00*nn*0102
DSV-*n*	^X00*nn*0105
SLIP-*n*	^X00*nn*0141
QNA-*n*	^X00*nn*0202
DSB-*n*	^X00*nn*0205
DECNET-*n*	^X00*nn*0241
BNA-*n*	^X00*nn*0302
DST-*n*	^X00*nn*0305
PPP-*n*	^X00*nn*0341

SVA-*n*	^X00*nn*0402
MNA-*n*	^X00*nn*0502
ISA-*n*	^X00*nn*0602
KFE-*n*	^X00*nn*0702
MXE-*n*	^X00*nn*0802
ERA-*n*	^X00*nn*0902
EWA-*n*	^X00*nn*0A02
CLIP-*n*	^X00*nn*2002
ELA-*n*	^X00*nn*2102
MFA-*n*	^X00*nn*4102
FZA-*n*	^X00*nn*4202
FAA-*n*	^X00*nn*4302
FEA-*n*	^X00*nn*4402
FQA-*n*	^X00*nn*4502
TRA-*n*	^X00*nn*6102
TRE-*n*	^X00*nn*6202

Note! The I/O status block (iosb) returns routing table entry size information for the *p3*=8 function to assist in diagnosing buffer overflow situations. See the *Status* section for details.

Reading the ARP Table Function

Use IO$_SENSEMODE | IO$M_CTRL with *function*=3 to read a network device's ARP table function. The information returned in the buffer (specified by *p2=address*) depends on the line id specified in *line-id*. The SHOW ARP command in NETCU uses this function.

The *line-id* argument is the line id and is a longword value. The least significant byte of the line id is the major device type code. The next byte is the device type subcode. The next byte is the controller unit number. The most significant byte is ignored.

The information returned in the buffer can consist of multiple records. Each record consists of 12 bytes, and one

record is returned for each ARP table entry.

When reading a table function, the data in each record is returned in the following order:

1	**Internet address.**	Returned as a longword value.
2	**Physical address.**	Returned as a 6 byte value.
3	**Flags.**	Returned as a word value. The ARP table entry's flag bits are shown in Table 5-5.

Table 5-5 ARP Table Entry Flag Bits

Mask	Name	Description
1	PERMANENT	If set, the entry can only be removed by a NETCU REMOVE ARP command and if RARP is enabled, the local host responds if a RARP request is received for this address. If clear, the entry can be removed if not used within a short period.
2	PUBLISH	If set, the local host responds to ARP requests for the internet address (this bit is usually only set for the local hosts's entry). If clear, the local host does not respond to received ARP requests for this address.
4	LOCKED	If set, the physical address cannot be changed by received ARP requests/replies.
4096	LASTUSED	If set, last reference to entry was a use rather than an update.
8192	CONFNEED	If set, confirmation needed on next use.
16384	CONFPEND	If set, confirmation pending.
32768	RESOLVED	If set, the physical address is valid.

Status

SS$_BADPARAM	Code specified in *function* argument invalid
SS$_BUFFEROVF	Buffer too small for all information Truncated buffer returned.

SS$_DEVINACT	Device not active Contact your system manager to determine why TCPware was not started
SS$_NORMAL	Success Requested information returned
SS$NOSUCHDEV	Line identification specified in *arp* argument does not exist

The byte count for the information or counters buffer is returned in the high-order word of the first longword of the I/O status block. This can be less than the bytes requested.

- For the p3=2 routing table function, in the second longword of the I/O status block, bit 0 is always set, bit 1 is set if the forwarding capability is enabled, and bit 2 is set if ARP replies for non-local internet addresses are enabled.
- For the p3=8 routing table function, the IOSB contains the following:

Status Code	SS$_NORMAL or SS$_BUFFEROVF
Transfer Byte Count	Number of bytes of returned information
Entry Size	Number of bytes in each entry
Number of Entries	Number of entries in the routing table

If the status is SS$_BUFFEROVF, you can determine the number of routing entries actually returned by calculating (Transfer Byte Count) DIV (Entry Size) and comparing that with the Number of Entries value. Be sure to check the Entry Size in the IO status block. Later versions of TCPware may return more information for each entry, which will return a larger Entry Size. Any additional information to be returned in the future will be added to the end of the returned entry.

IO$_SENSEMODE | IO$M_RD_COUNT

Reads the IP counters.

The IO$M_CLR_COUNT modifier with this function zeros the counters after they are read. Using IO$M_CLR_COUNT requires the OPER privilege.

Format

status = SYS$QIO(***efn, chan,***IO$_SENSEMODE | IO$M_RD_COUNT, *iosb, astadr, astprm,* ***buffer, address,*** **0, 0, 0, 0)**

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	write only
mechanism:	by reference

Optional address of the 8-byte device characteristics buffer. The data returned is the device class (DC$_SCOM) in the first byte, the device type (0) in the second byte, and the default buffer size (0) in the high-order word of the first longword. The second longword is returned as 0.

p2=address

OpenVMS usage:	**vector_longword_unsigned**
type:	longword (unsigned)
access:	write only
mechanism:	by descriptor

Address of the descriptor for the buffer to receive the information. The information returned in the buffer is a record that consists of eleven longwords. The data in the record is returned in the order below. Each value is a longword.

1 Number of seconds since last zeroed

2 Number of IP datagrams transmitted

3 Number of IP datagrams fragmented

4 Number of IP datagrams forwarded (if gateway capability is enabled)

5 Number of ICMP messages transmitted
6 Number of IP datagrams and fragments received
7 Number of IP fragments received
8 Number of ICMP messages received
9 Number of datagrams delivered to receivers
10 Number of IGMP messages transmitted
11 Number of IGMP messages received

Status

SS$_BUFFEROVF	Buffer too small for all characteristics Truncated characteristics buffer returned
SS$_NOPRIV	Insufficient privilege OPER privilege required to zero counters

The byte count for the counters buffer is returned in the high-order word of the first longword of the I/O status block. This is the actual number of bytes received, which may be less than what was requested.

IO$_SETMODE | IO$M_CTRL

Sets the port characteristics.

Format

status = SYS$QIO(***efn, chan,***IO$_SETMODE | IO$M_CTRL, *iosb, astadr, astprm,* ***0, address,*** **0, 0, 0, 0)**

Argument

p2=address

OpenVMS usage:	**vector_word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by descriptor

IO$_SETMODE | IO$M_CTRL sets the port characteristics in the extended characteristics buffer. The buffer consists of a series of six-byte or counted string entries. The first word of each entry contains the parameter identifier (PID) of a characteristic, followed by either a longword that contains the value for that characteristic or a counted string. Counted strings consist of a word with the size of the character string followed by the character string.

The address argument is the address of the descriptor for the extended characteristics buffer. Figure 5-3 shows the format of the descriptor for the extended characteristics buffer and the extended characteristics buffer.

Figure 5-3 P2 Set Characteristics Buffer

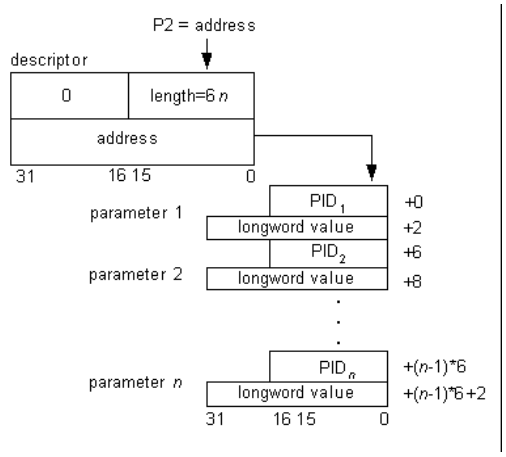


Table 5-6 lists the port characteristics you can set.

Table 5-6 P2 Set Port Characteristics

PID	Meaning
13	**IP options.** Counted string. Internet Protocol options to be included in datagrams to be transmitted.
14	**IP TTL.** Longword. Low byte of the longword contains the time-to-live (TTL) to be used in datagrams to be transmitted.
15	**IP TOS.** Longword. Low byte of the longword contains the type of service (TOS) to be used in datagrams to be transmitted.
16	**Join multicast group.** Counted string of eight bytes. First longword contains the multicast group IP address to join. Last longword contains the IP address of the local interface on which to join the group (if 0, the default interface is used).
17	**Leave multicast group.** (See the previous PID.)
18	**Multicast Interface.** Longword. IP address of the local interface to use in sending multicast datagrams. If 0, the default interface is used.
19	**Multicast TTL.** Longword. Low byte contains the time-to-live (TTL) to be used in multicast datagrams to be transmitted.
20	**Multicast loopback.** Longword. Low byte is 0 to disable and 1 to enable local loopback of multicast datagrams.

Note! The above parameters remain set even after you close the port until you change them with another IO$_SETMODE operation

Status.

SS$_BADPARAM	Specified bad parameter or parameter value Returns PID in low order word of second longword
SS$_DEVACTIVE	Port is open, and you made request to change parameter that cannot be changed when port is open
SS$_DEVINACT	Device is not active Contact your system manager to determine why TCPware (or IPDRIVER) was not started
SS$_DUPINIT	Multicast group address already joined

SS$_INSFMEM	Insufficient memory to complete request
SS$_IVBUFLEN	Extended characteristics buffer length invalid
SS$_NOPRIV	Setting IP layer options (PIDs 13 through 20) requires privileges Application must either be running under SYSTEM UIC or have SYSPRV, BYPASS, or OPER privilege
SS$_NORMAL	Success Characteristics set
SS$_TOOMUCHDATA	Too many multicast group addresses specified
SS$_UNREACHABLE	No default interface for multicast group address could not be found Specify local interface's IP address when joining multicast group

IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN

Closes a port.

Closing a port aborts all pending receives for the port, prevents further receives and transmissions on the port, and lets another user use the protocol number.

Format

status = SYS$QIO(***efn, chan,***IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN,*iosb, astadr, astprm,* **0, 0, 0, 0, 0, 0)**

Arguments

None.

Status

SS$_DEVINACT	Device not active Contact your system manager to determine why TCPware (or IPDRIVER) was not started
SS$_NORMAL	Success Port now (or was) closed Port can no longer receive any datagrams; however, datagrams can still be sent
SS$_NOPRIV	Insufficient privileges BYPASS or SYSPRV privilege required to open port

IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP

Opens a port. Opening a port lets you receive internet datagrams.

Requires BYPASS or SYSPRV privileges.

Format

status = SYS$QIO(***efn, chan,***IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP, *iosb, astadr, astprm,* **0, 0, protocol, num, 0, 0)**

Arguments

p3=protocol

Internet protocol number in the low byte. Datagrams are transmitted using this protocol number, and you can only receive datagrams with it.

See the latest Assigned Numbers RFC for a list of the currently assigned protocol numbers.

p4=num

OpenVMS usage:	**longword_unsigned**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Number of unsolicited receives to be queued for delivery. A value of 0 to 5 is valid. 0 results in all unsolicited receives being discarded.

Status

SS$_DEVACTIVE	Port already open
SS$_DEVINACT	Device not active Contact your system manager to determine why TCPware was not started
SS$_DUPUNIT	Specified port number already in use by another port
SS$_NOPRIV	Insufficient privileges You need BYPASS or SYSPRV privilege to open port

SS$NORMAL	Success Port open and write and read functions can be used to transmit or receive datagrams

IO$_WRITEVBLK

Sends data. This builds and transmits the datagram using the protocol number for the port.

The IO$M_NOFORMAT modifier prevents fragmentation. This sets the "don't fragment" bit in the IP datagram.

The IO$M_FORCEPATH modifier prevents routing on the datagram. This limits the datagram to be sent to a locally connected address.

The IO$M_LPBEXT modifier inhibits a copy of a broadcast or multicast datagram from being looped back.

Format

status = SYS$QIO(***efn, chan,***IO$_WRITEVBLK,*iosb, astadr, astprm,* ***buffer, size, address, dest, source,*** **0)**

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	read only
mechanism:	by reference

User's buffer address. This is the buffer that contains the data to be sent. The buffer does not include the internet header.

p2=size

OpenVMS Usage:	**word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by value

User's buffer size in bytes (the byte count). This is the amount of data that the user has to send. The value should not be greater than 65000 bytes.

If the resulting datagram is larger than the maximum transmission unit (MTU) for the network, the datagram is fragmented, unless you specified IO$M_NOFORMAT. In that case, IO$_WRITEVBLK returns an error.

p3=address

OpenVMS usage:	**vector_word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by descriptor

Address of the descriptor for the optional transmit characteristics buffer.

The transmit characteristics buffer consists of a series of 6-byte and counted string entries. The first word of each entry contains the parameter identifier (PID) followed either by a longword or by a counted string.

The PID determines whether a longword value or counted string follows. The longword contains one of the values that can be associated with that parameter. The counted string consists of a word byte count field followed by the specified number of bytes of data.

Figure 5-4 shows the format of the descriptor for the extended characteristics buffer and the extended characteristics buffer.

Table 5-7 lists the characteristics you can specify.

Figure 5-4 P3 Transmit Characteristics Buffer

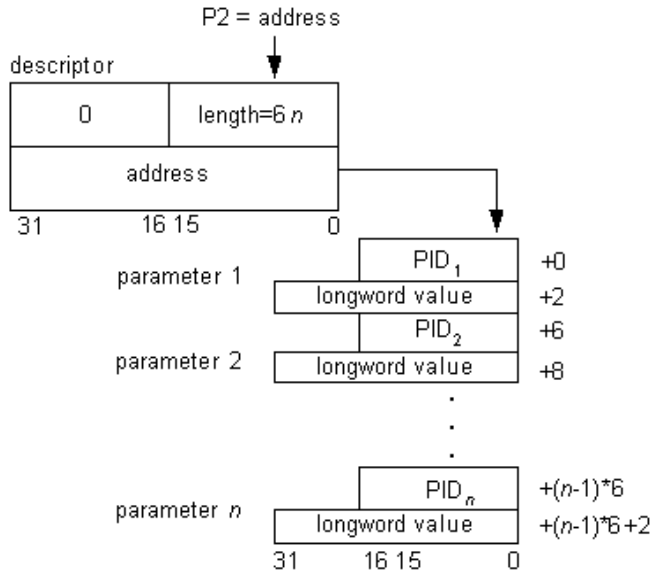


Table 5-7 P3 Transmit Characteristics

PID	Meaning
1	**Type of service.** Low-order byte of the longword value. Contains the type of service. If omitted, the previously specified value or the default of 0 is used.
2	**Time to live.** Low-order byte of the longword value field. Contains the time to live. If not specified, the previously specified value of the default is used. The default time to live is specified by the IPDEFAULTTTL parameter for non-multicast datagrams and is 1 for multicast datagrams.
3	**Identification.** Low-order word of the longword value field. Contains the identification number. IPDRIVER automatically increments this value for each transmission (the initial value starts at 0).
4	**Options.** Counted string. Contains the internet options buffer. This buffer is used as is and is not validated. The maximum size of this counted string is 40 bytes. Note that if the options are not a multiple of 4 bytes, IPDRIVER adds the required padding (using the End of Option List option).
5	**Protocol.** Lowest-order byte of the longword value field. Contains the IP protocol number to be used in sending the datagram. Can be used to override the protocol under which this unit was opened.

See the specifications for the Internet Protocol (RFC 791) for details on each of the internet header items described above.

p4=dest

OpenVMS usage:	**longword_unsigned**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Destination internet address. Internet addresses are specified in internet byte order (which is reversed from the normal VAX byte order). For example, internet address 2.3.4.5 is stored as ^X05040302. If the destination internet address is the loopback internet address, IPDRIVER loops-back the datagram.

p5=source

OpenVMS usage:	**longword_unsigned**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Optional source internet address. Must be a valid local internet address (if specified). Internet addresses are specified in internet byte order (which is reversed from the normal VAX byte order). For example, internet address 2.3.4.5 is stored as ^X05040302.

Status

SS$_BADPARAM	Bad parameter or parameter value specified PID returned in low order word of second longword If parameter ID is **0**, source internet address is not valid (it is not local address)
SS$_DEVINACT	Device not active Contact your system manager to determine why TCPware (or IPDRIVER) was not started
SS$_INSFMEM	Insufficient memory available to fragment datagram
SS$_INVSECLASS	Basic Security Option label incompatible with outgoing IPSO settings
SS$_IVBUFLEN	User's buffer too large
SS$_NORMAL	Success Datagram transmitted Failure of translation is not considered an error because datagram would most likely be retransmitted by higher level software at a later time Also allows some time for ARP reply to be received
SS$_OPINCOMPL	Datagram not transmitted because no destination internet address or port number was specified by source/destination bugger or port's characteristics
SS$_THIRDPARTY	TCPware shut down

SS$_UNREACHABLE	No route to the destination internet address exists Add route for destination or check internet address

The number of data bytes transmitted are returned in the high-order word of the first longword of the I/O status block.

SYS$ASSIGN

Assigns a channel to a device.

Format

status = SYS$ASSIGN (*devnam*, *chan*, *[acmode]*, *[mbxnam]*)

Arguments

devnam

OpenVMS usage:	**device_name**
type:	character_coded text string
access:	read only
mechanism:	by descriptor-fixed length string descriptor

Address of a character string descriptor pointing to the device name string (IPA0:).

chan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	write only
mechanism:	by reference

Address of a word into which SYS$ASSIGN writes the channel number.

acmode

OpenVMS usage:	**access_mode**
type:	longword (unsigned)
access:	read only
mechanism:	by value

Optional access mode associated with the channel. The most privileged access mode used is that of the caller.

mbxnam

OpenVMS usage:	**device_name**
type:	character-coded text string
access:	read only
mechanism:	by descriptor-fixed length string descriptor

Optional logical mailbox associated with the device. (Not supported by IPDRIVER)

Status

See HP's*VMS System Services Reference Manual* for a complete list of status messages.

SYS$CANCEL

Cancels any I/O that is pending on a channel.

The I/O will be completed with an *iosb* status of `SS$_CANCEL`.

Outstanding I/O operations are automatically canceled at image exit.

Format

status = SYS$CANCEL (*chan*)

Argument

chan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	read only
mechanism:	by value

Number of the channel to be canceled.

Status

See HP's *VMS System Services Reference Manual* for a complete list of status messages.

SYS$DASSGN

Releases a channel.

When you deassign a channel, any outstanding I/O is completed with an *iosb* status of `SS$_CANCEL`.

I/O channels are automatically deassigned at image exit.

Format

status = SYS$DASSGN (*chan*)

Argument

chan

OpenVMS usage:	**channel**
type:	word (unsigned)
access:	read only
mechanism:	by value

Number of the channel to be deassigned.

Status

See HP *VMS System Services Reference Manual* for a complete list of status messages.

IPDRIVER External Interface

This section describes the IPDRIVER External Interface.

This interface allows you to write Network Interface Programs that support the use of IPDRIVER with various network controllers.

I/O Functions for the External Interface

The Network Interface Program uses four functions to communicate with IPDRIVER. Table 5-8 lists these functions.

Table 5-8 I/O Functions for the External Interface

Function	Purpose
IO$_INITIALIZE	Initializes the IPDRIVER External Interface for the network device. (Informs IPDRIVER that the External Interface is on-line.)
IO$_READVBLK	Receives datagrams from IPDRIVER for transmission over the network.
IO$_SETMODE \| IO$M_CTRL | IO$M_SHUTDOWN	Shuts down the IPDRIVER External interface.
IO$_WRITEVBLK	Delivers datagrams that are received from the network to IPDRIVER. IPDRIVER processes and delivers them to higher-level applications (such as TCP and UDP).

Sequence of Operations for the Network Interface Program

This section provides the typical sequence of operations for the Network Interface Program. These operations apply only to the IPDRIVER External Interface. Any operations needed to set up and use the network device depend on the specific requirements of that device.

The Network Interface Program:

1. Uses the Assign I/O Channel (SYS$ASSIGN) system service to assign an I/O channel to IPA0:. SYS$ASSIGN creates a new unit for the channel.
2. Issues the IO$_INITIALIZE function on the assigned channel to initialize the network line.
3. Issues the IO$_READVBLK function to read IP datagrams from IPDRIVER, then sends the datagrams over the network device.
4. Issues the IO$_WRITEVBLK function for each datagram received from the network device.
5. Issues the IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN function to shut down the network device.
6. Uses the Deassign I/O Channel (SYS$DASSGN) system service to deassign the channel.

Your Network Interface programs must provide their own means of starting up and shutting down. You cannot use the NETCU START/IP and STOP/IP commands to start up or shut down these network lines.

IPDRIVER External Interface System Service Call Codes

The following pages describe each SYS$QIO[W] function code for the external interface. System service call function codes specify what action the QIO performs. See the format specified in the IPDRIVER System Service Call Format section before referencing each function.

You can use OpenVMS system services SYS$ASSIGN, SYS$DASSGN, and SYS$CANCEL for the External

Interface in the same way as described for the User Interface in the IPDRIVER User Interface System Service Call Function Co section.

This section describes the following function codes for the External Interface:

IO$_INITIALIZE	IO$_SETMODE	IO$_READVBLK	IO$_WRITEVBLK

IO$_INITIALIZE (External)

Informs IPDRIVER that the External Interface is on-line. IPDRIVER adds the External Interface to a list of existing network interfaces and creates a route for it.

Note! Once a Network Interface program issues an IO$_INITIALIZE on a channel, that channel becomes the External Interface. After that, the functions documented in the IPDRIVER User Interface System Service Call Function Co section for the User Interface (such as those used to open and close ports) are no longer valid: you must use only those functions documented in this part of the chapter, theIPDRIVER External Interface section.

Format

status = SYS$QIO(***efn, chan,***IO$_INITIALIZE,*iosb, astadr, astprm, **buffer, size, 0, 0, 0, 0)**

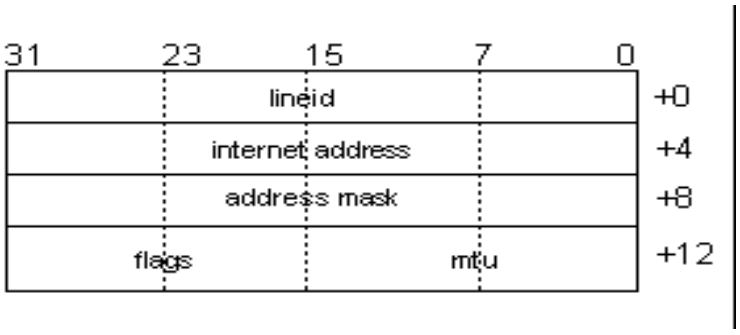
Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	read only
mechanism:	by reference

Address of the line information buffer. Figure 5-5 illustrates this buffer.

Figure 5-5 Line Information Buffer for IO$_INITIALIZE



In the figure:

address mask	is the address mask for the network. The bits set in the mask are the bits that specify the host number (the one's complement of the network mask).

flags	is the line's flag bits. Typically, this value is 0, but the following bits can be set as needed:
internet address	is the internet address of the host on the network or, for unnumbered interfaces, the internet address used as the source address when sending datagrams over the interface if a source address was not explicitly specified.
lineid	is the line identification. Its value should be 00*nnxx*41 (hex), where: *nn* is the unit number of the line. *xx* is a unique value for the specific network line. SLIP lines are 01. IP-over-DECnet lines are 02. PPP lines are 03. For Network Interface Programs, *nn* should start at 80 (hex) and a
mtu	is the maximum transmission unit for the network. It is the size of the largest IP datagram that can be transmitted over the network.

Bit	**Mask**	**Meaning**
0	1	Unnumbered interface. If this bit is set, the interface does not have a local address and the internet address specified is only used when originating datagrams over the interface when no source address was explicitly specified.

p2=size

OpenVMS usage:	**word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by value

Size of the line information buffer, in bytes.

Status

SS$_NORMAL	Initialization successful
SS$_BADPARAM	*lineid* parameter does not contain valid value, or already in use Lowest order byte must be 41 (hex) for External Interface
SS$_DEVACTIVE	Device already active
SS$_DEVREQERR	You are trying to start up new interface while IPDRIVER is in process of shutting down
SS$INSFMEM	Insufficient system memory Necessary control block cannot be allocated
SS$_IVADDR	Address already in use
SS$_NOPRIV	Insufficient privileges You need OPER privilege to start network interface

IO$_READVBLK (External)

Requests IPDRIVER to return the next IP datagram to the Network Interface program. The Network Interface program should then send the datagram over the network.

The Network Interface program is responsible for adding the network header to the datagram.

IPDRIVER provides "unsolicited" read support. This means that datagrams are held in a queue until the Network Interface Program issues an IO$_READVBLK function. IPDRIVER queues up to 100 datagrams.

Format

status = SYS$QIO(***efn, chan,* **IO$_READVBLK,*iosb, astadr, astprm,* ***buffer, size,*** **0, 0, 0, 0)**

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	byte (unsigned)
access:	write only
mechanism:	by reference

Address of the buffer that receives the next IP datagram.

p2=size

OpenVMS usage:	**word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by value

Size, in bytes, of the buffer. This buffer must be at least as large as the MTU of the interface and not greater than 65000 bytes. Otherwise, IPDRIVER might return a SS$_DATAOVERUN status code.

Status

SS$_NORMAL	Read successful In I/O status block, high order word of first longword contains size of IP datagram in bytes

SS$_ABORT	Requests aborted External Interface is down
SS$_DATAOVERUN	Receive buffer in Network Interface Program too small for IP datagram Truncated datagram returned to user-specified buffer In I/O status block high order word of first longword contains truncated size of IP datagram in bytes
SS$DEVINACT	Device not active Use IO$_INITIALIZE function first

The high-order word of the first longword contains the size of the IP datagram read.

The second longword of the I/O status block contains the internet address of the next destination. The next destination can be the final destination or a gateway. This longword is valid only for the SS$_NORMAL and SS$_DATAOVERUN status codes.

IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN (External)

Shuts down the External Interface for the channel.

Causes IPDRIVER to remove the network interface from the list of available interfaces. It also removes any routes for the network interface.

Format

status = SYS$QIO(***efn, chan,***IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN,*iosb, astadr, astprm,* **0, 0, 0, 0, 0, 0)**

Arguments

None.

Status

SS$_Normal	Success

IO$_WRITEVBLK (External)

Used after the Network Interface program receives a datagram from the network. Delivers the datagram to IPDRIVER for processing.

Format

status = SYS$QIO(***efn, chan,***IO$_WRITEVBLK,*iosb, astadr, astprm,* ***buffer, size,* 0, *flags*, 0, 0)**

Arguments

p1=buffer

OpenVMS usage:	**vector_byte_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by value

Address of the buffer containing the full IP datagram.

p2=size

OpenVMS usage:	**word_unsigned**
type:	word (unsigned)
access:	read only
mechanism:	by value

Size of the IP datagram in bytes. The value should not be greater than 65000 bytes.

Note! The buffer specified by these parameters must not include the network header.

p4=flags

OpenVMS usage:	**mask_longword**
type:	longword (unsigned)
access:	read only

mechanism:	by value

Flag bits providing information about the received datagram as follows:

Bit	Mask	Meaning
0	1	Datagram was sent to a broadcast/multicast address. This information is used to prevent sending ICMP datagrams to broadcast/multicast addresses. All other bits must be 0.

Status

SS$_NORMAL	Write successful
SS$_ABORT	Request aborted External Interface shut down
SS$_DEVINACT	Device not active Use IO$_INITIALIZE function first

Chapter 6 INETDRIVER Services

Introduction

This chapter describes the internet device driver (INETDRIVER) services.

The INETDRIVER services provide an asynchronous I/O implementation of the UNIX socket calls within the OpenVMS Queue I/O Request (SYS$QIO and SYS$QIOW) system services. These system services allow for efficient socket operations in that Asynchronous System Trap (AST) routines can be associated with I/O requests.

The INETDRIVER services provide the Stanford Research Institute (SRI) QIO interface. The interface is an international de facto standard that provides a one-to-one mapping between the UNIX socket functions and the OpenVMS SYS$QIO system services.

INETDRIVER interfaces directly with the TCP, UDP, and IP (for raw functions) protocols in the transport layers. It does not replace the TCPDRIVER or UDPDRIVER services, but provides another way to communicate with them.

For details, see the *TCPDRIVER Services* and *UDPDRIVER Services* chapters.

INETDRIVER supports QIOs to:

- Create a stream, datagram, or raw socket
- Bind a socket to a local port or address
- Set a socket to listen mode
- Accept a connection
- Create a connection
- Send and receive data
- Perform other socket and control functions

Note! The TCPware for OpenVMS INETDRIVER supports stream, datagram, and raw sockets, but only for the AF_INET address family.

Sequence of Operations

The INETDRIVER sequences of operations are divided into client operations and server operations. Each have distinct steps, as outlined in the following sections.

Client Operations

The INETDRIVER client operations for a stream socket connection are as follows:

1 Assign an INET channel to the INET0: device using the SYS$ASSIGN system service. This creates a new INET device unit and assigns a channel to it.

2 Create a stream socket using the SYS$QIO[W] system service with the IO$_SOCKET function.

3 If you want to assign a particular address to an unnamed socket, use the SYS$QIO[W] IO$_BIND function. This is optional and not necessary under most circumstances.

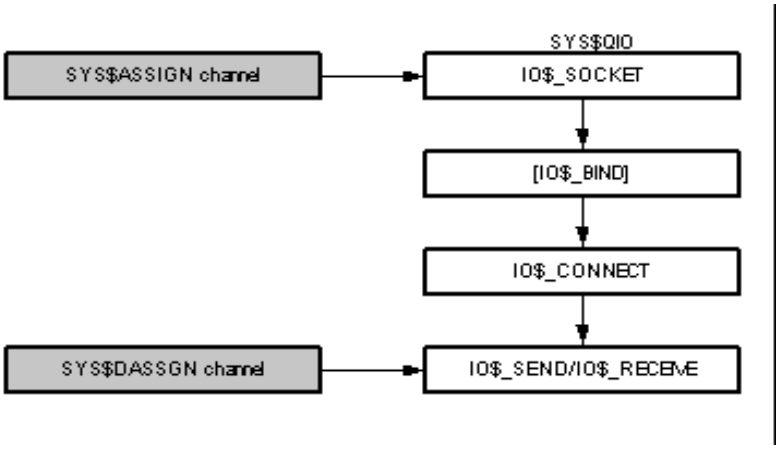
4 Connect to a peer on the server using the SYS$QIO[W] IO$_CONNECT function. Specify the server's internet address any "well-known" port number.

5 Exchange data with the peer using the SYS$QIO[W] IO$_SEND and SYS$QIO[W] IO$_RECEIVE functions.

6 After the data exchange is complete, deassign the channel using the SYS$DASSGN system service.

Figure 6-1 shows the client operation steps.

Figure 6-1 Client Operation Steps



Server Operations

The INETDRIVER server operations for a simple, single-thread stream socket connection are as follows:

1 Assign an INET listen channel to the INET0: device using the SYS$ASSIGN system service. This creates a new INET device unit and assigns a channel to it.

2 Create a stream socket using the SYS$QIO[W] IO$_SOCKET function.

3 Bind the socket to the server's "well-known" port number using the SYS$QIO[W] IO$_BIND function.

4 Enable listens and set up the listen queue length using the SYS$QIO[W] IO$_LISTEN function.

5 Assign an INET accept channel to the INET0: device using the SYS$ASSIGN system service again.

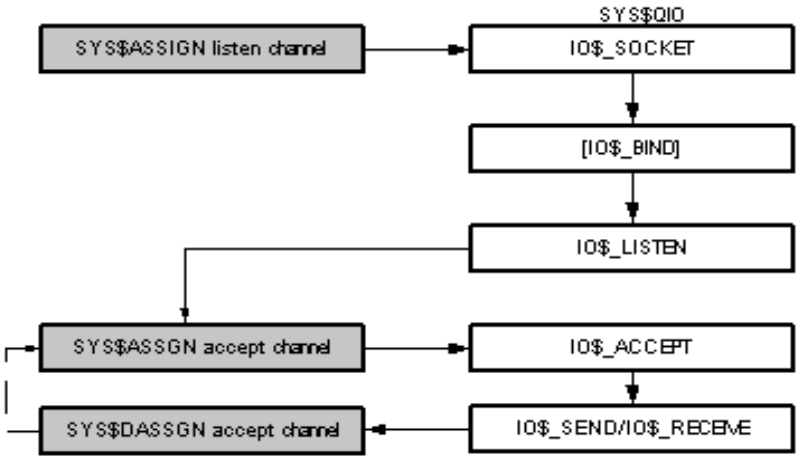
6 Accept incoming connect requests using the SYS$QIO[W] IO$_ACCEPT function on the accept channel, specifying the listen channel as a parameter. Then wait for the client to connect.

7 Exchange data with the client on the accept channel using the SYS$QIO[W] IO$_SEND and SYS$QIO[W] IO$_RECEIVE functions.

8 After the data exchange is complete, deassign the accept channel using the SYS$DASSGN system service. This closes the connection.

Repeat steps 5 through 8 to process additional server connections. Figure 6-2 shows the server operations steps.

Figure 6-2 Server Operation Steps



Multicasting

INETDRIVER Services includes the following `setsockopt` and `getsockopt` definitions at the IPPROTO_IP level for multicasting support:

```
#define   IP_MULTICAST_IF     2
#define   IP_MULTICAST_TTL    3
#define   IP_MULTICAST_LOOP   4
#define   IP_ADD_MEMBERSHIP   5
#define   IP_DROP_MEMBERSHIP  6
```

The IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP requests use the following structure:

```
struct ip_mreq {
  struct in_addr  imr_multiaddr; /* Multicast address of group */
  struct in_addr  imr_intrface;  /* local IP address of interface */
};
```

Other Operations

In addition to the routines associated with the sequence of operations above, INETDRIVER also includes other operations that:

- Set socket options (IO$_SETSOCKOPT)
- Get socket options (IO$_GETSOCKOPT)
- Shut down sockets (IO$_SHUTDOWN)
- Get socket names (IO$_GETSOCKNAME)
- Change the INET device characteristics (IO$_SETCHAR)
- Get the address of the remote end for a socket (IO$_GETPEERNAME)

- Change socket characteristics (IO$_IOCTL)
- Deliver an AST to the process if out-of-band data arrives (IO$_SETMODE | IO$_ATTNAST)

INETDRIVER Socket Library

The Socket Library routines described in the *Socket Library* chapter make use of the TCPDRIVER and UDPDRIVER programming interface. An alternate set of socket routines is available that makes use of the INETDRIVER programming interface and allows a mix of socket routines and INETDRIVER QIO calls. This alternate set of socket routines is presently only available in the TCPWARE:SOCKLIB.OLB socket library (the routines are not part of the TCPWARE_SOCKLIB_SHR.EXE shareable run-time library). The following routines are available:

inet_socket	inet_socket_recv	inet_sendto	inet_socket_ioctl
inet_bind	inet_recv	inet_shutdown	inet_getsockname
inet_connect	inet_recvfrom	inet_socket_close	inet_getpeername
inet_listen	inet_socket_write	inet_setsockopt	inet_socket_perror
inet_accept	inet_socket_send	inet_getsockopt	inet_tcpware_server
inet_socket_read	inet_send		

The socket number (descriptor) used by these routines is the VMS I/O channel of the INETDRIVER device being used. This allows for the mixing of these routines and direct QIOs to the driver.

INETDRIVER System Service Call Format

The format for the INETDRIVER SYS$QIO system service call is as follows:

status= SYS$QIO*[*W*]*(*efn*, *chan*, *func*, *iosb*, *astadr*, *astprm*, *p1*, *p2*, *p3*, *p4*, *p5*, *p6*)

The TCPWARE_INCLUDE:INETIODEF.H file provides definitions for the QIO function codes.

Note! The E-prefix status codes (such as `ECONNRESET`) listed under the STATUS headings are returned in the I/O status block (*iosb*) and are not the UNIX *errno* values. As described in I/O Status Block Fields under the *iosb* argument, the values returned for these codes are the UNIX *errno* values multiplied by 8 and logically ORed with 0x8000.
The TCPware Socket Library Runtime Library (RTL) contains the message pointer to the SYS$MESSAGE:TCPWARE_MSG.EXE file. The RTL also contains definitions for the status values. These are the UNIX *errno* names prefixed by TCPWARE$_ (for example, `TCPWARE$_ECONNRESET`).

Note! The vertical bar (|) used in some of the functions described in this chapter is the C bitwise inclusive OR operator.

INETDRIVER System Service Call Arguments

You invoke UDPDRIVER system service calls with the standard OpenVMS QIO mechanism.

See the appropriate OpenVMS documentation (for example, the *Introduction to VMS System Services* volume) for details on the QIO mechanism.

The following sections describe each system call argument.

efn

OpenVMS usage:	ef_number
type:	longword (unsigned)
access:	read only
mechanism:	by value

(Optional) Number of the event flag to be set when the I/O operation completes.

chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

I/O channel assigned to the INET device to which the request is directed.

func

OpenVMS usage:	function_code
type:	word (unsigned)
access:	read only
mechanism:	by value

Device-specific function codes and modifiers for each operation.

Note! ***Error! Reference source not found.*** describes each function code in detail.

iosb

OpenVMS usage:	io_status_block
type:	quadword (unsigned)
access:	write only
mechanism:	by reference

I/O status block that receives the final completion status of the I/O operation, structured as in Figure 6-3.

Figure 6-3 I/O Status Block

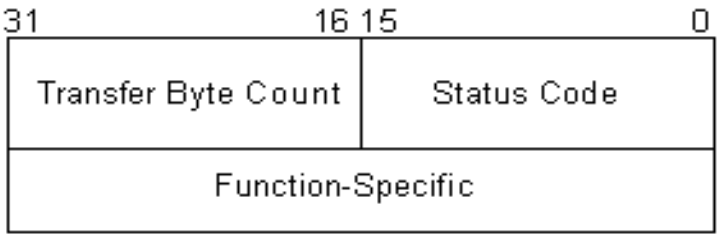


Table 6-1 describes the status block fields in more detail.-

Table 6-1 I/O Status Block Fields

Field Name	Description
Transfer Byte Count	Number of bytes of data transferred in the I/O operation.
Status Code	SS$ status code or special error status code. If the low bit (0) of the OpenVMS error code is clear, the network has returned an error. If the most significant bit of this word is set (mask of 0x8000), the status code is the UNIX errno code multiplied by 8 and logically ORed with 0x8000.
Function-Specific	Varies for each device and function code.

astadr

OpenVMS usage:	ast_procedure
type:	procedure entry mask
access:	call without stack unwinding
mechanism:	by reference

Asynchronous system trap (AST) routine to be executed when the I/O is completed.

astprm

OpenVMS usage:	user_arg
type:	longword (unsigned)
access:	read only
mechanism:	by value

AST parameter to be passed to the AST routine.

p1 to p6

OpenVMS usage:	varying_arg
type:	longword (unsigned)
access:	read only or write only
mechanism:	by reference or by value

Function-specific parameters, as described for each function.

INETDRIVER System Service Call Function Codes

System service call function codes specify what action the QIO performs.

IO$_ACCEPT

Waits for a connection to a listening socket and associates a OpenVMS channel to the socket for that new connection. The connection's socket has the same properties as the listening socket.

If no pending connections exist on the queue and the socket is not marked as nonblocking, IO$_ACCEPT blocks the caller until a connection is made to the listening socket. If the socket is marked as nonblocking and no pending connections exist on the queue, IO$_ACCEPT returns an `EWOULDBLOCK` error.

Multiple IO$_ACCEPT requests are supported on a single channel.

See IO$_ACCEPT_WAIT for an alternative to IO$_ACCEPT.

Format

status = SYS$QIO (*efn*, ***new-inet-chan*, IO$_ACCEPT,** *iosb*, *astadr*, *astprm*, ***address*, *addrlen*, *inet-chan*, 0, 0, 0)**

Arguments

chan=new-inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

OpenVMS channel to a newly-created INET: device. This channel should be created by using SYS$ASSIGN to assign a new channel to INET0: prior to issuing the IO$_ACCEPT function. The accepted connection uses this channel.

p1=address

OpenVMS usage:	special_structure
type:	structure defined below
access:	write only
mechanism:	by reference

Optional pointer to a structure that receives the peer's address for the connection following completion of the IO$_ACCEPT function. The structure is defined as follows:

```
struct {
  unsigned long Length;
  struct sockaddr_in Address;
}
```

p2=addrlen

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	read only
mechanism:	by value

Length (in bytes) of the buffer to which the *address* argument points. The value must be at least 20 bytes.

p3=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

OpenVMS channel for the INET: device on which the IO$_LISTEN was performed. After accepting the connection, this device remains available to accept additional connections.

Status

EALREADY	Operation is already in progress or the socket is in use (e.g. an IO$_ACCEPT is active on the channel).
ECONNABORTED	Listening channel was shut down or aborted.
EINVAL	Operation is not valid (for a nonstream socket), or an invalid *inet-chan* or *addrlen* argument was specified.
ENETDOWN	Network was shut down.
ENOTSOCK	No socket exists on the listening channel.

EOPNOTSUPP	Socket is not of type SOCK_STREAM.
EWOULDBLOCK	Connection is marked as nonblocking and no connection is waiting to be accepted.
SS$_IVCHAN	Invalid listening channel number was specified.
SS$_THIRDPARTY	Network is being shut down.

IO$_ACCEPT_WAIT

Waits for an incoming connection on a listening socket without accepting the connection. When the IO$_ACCEPT_WAIT is complete, a connection is available for accepting; IO$_ACCEPT is then used to accept it.

Multiple IO$_ACCEPT_WAIT requests are supported on a single channel.

IO$_ACCEPT_WAIT is useful to avoid holding an additional INET channel while waiting to service a connection.

Format

status=SYS$QIO(*efn*,***inet-chan*, IO$_ACCEPT_WAIT**, *iosb*, *astadr*, *astprm*, **0, 0, 0, 0, 0, 0)**

Argument

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

OpenVMS channel to the INET: device on which the IO$_LISTEN was performed.

Status

ECONNABORTED	Listening channel was shut down or aborted.
EINVAL	Operation is not valid (for a nonstream socket), or an invalid *inet-chan* or *addrlen* argument was specified.
ENETDOWN	Network was shut down.
ENOTSOCK	No socket exists on the listening channel.
EOPNOTSUPP	Socket is not of type SOCK_STREAM.
EWOULDBLOCK	Connection is marked as nonblocking and no connection can be accepted.
SS$_THIRDPARTY	Network is being shut down.

IO$_BIND

Assigns an address to an unbound socket. When a socket is created with IO$_SOCKET, it has no assigned address. IO$_BIND requests that the address be assigned to the socket.

Normally used by servers to bind to their "well-known" port number before issuing the IO$_LISTEN function. Clients typically do not use this function, since IO$_CONNECT normally will assign an unused port number.

Note! To bind to port numbers 1 through 1023, a process must be running under a privileged UIC, or with the SYSPRV or BYPASS privilege.

Format

status=SYS$QIO(*efn*, ***inet-chan*, IO$_BIND,** *iosb*, *astadr*, *astprm*, ***name*, *namelen*,0, 0, 0, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

OpenVMS channel to the INET: device on which the IO$_SOCKET was performed.

p1=name

OpenVMS usage:	socket_address
type:	structure `sockaddr_in`
access:	read only
mechanism:	by reference

Address to which the socket should be bound.

p2=namelen

OpenVMS usage:	socket_address_length

type:	longword (unsigned)
access:	read only
mechanism:	by value

Length of the *name* argument (in bytes).

Status

EACCES	Specified address is not available from the local machine.
EADDRNOTAVAIL	Requested address is protected; the user has no permission to have access to it.
EADDRINUSE	Specified address is already in use.
EINVAL	Operation is not valid (for a nonstream socket).
ENETDOWN	Network was shut down.
SS$_THIRDPARTY	Network is being shut down.

IO$_CONNECT

For stream sockets, attempts a connection to a peer socket. For datagram sockets, permanently specifies the peer to which the datagrams are to be sent or from which they are to be received.

Format

status=SYS$QIO(***efn,* ***inet-chan,*** **IO$_CONNECT,*iosb, astadr, astprm,***name, namelen,0, 0, 0, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=name

OpenVMS usage:	socket_address
type:	structure `sockaddr_in`
access:	read only
mechanism:	by reference

Address of the peer to which the socket should be bound.

p2=namelen

OpenVMS usage:	socket_address_length
type:	longword (unsigned)
access:	read only

mechanism:	by value

Length of the *name* argument (in bytes).

Status

EADDRNOTAVAIL	Requested address is protected; the user has no permission to have access to it.
EADDRINUSE	Specified address is already in use.
EALREADY	Operation is already in progress.
ECONNABORTED	Operation was aborted.
ECONNREFUSED	Connection was refused by the peer.
EHOSTUNREACH	Host was unreachable.
EINVAL	Socket has already been used for a connection, or the *namelen* argument is invalid.
EISCONN	Socket is already connected.
ENETDOWN	Network has been shut down.
ENETUNREACH	Destination network is unreachable.
EOPNOTSUPP	Operation is not supported (e.g., a connect on a listening socket).
ETIMEDOUT	Connection timed out.
EWOULDBLOCK	Socket is marked as nonblocking; connection attempt is in progress.
SS$_THIRDPARTY	Network is being shut down.

IO$_GETPEERNAME

Returns the address of the peer connected to the specified socket.

Format

status=SYS$QIO(*efn*,***inet-chan*, IO$_GETPEERNAME,** *iosb*, *astadr*, *astprm*, ***address, addrlen*,0, 0, 0, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=address

OpenVMS usage:	socket_address
type:	structure `sockaddr_in`
access:	write only
mechanism:	by reference

Receives the peer's address.

p2=addrlen

OpenVMS usage:	socket_address_length
type:	longword (unsigned)
access:	modify
mechanism:	by reference

On entry, contains the byte length of the space pointed to by *address*. On return, it contains the byte length of the data returned.

Status

EINVAL	Invalid *addrlen* was specified.
ENETDOWN	Network has been shut down.
ENOTCONN	Socket is not connected.
SS$_THIRDPARTY	Network is being shut down.

IO$_GETSOCKNAME

Returns the local address of a socket.

Format

status=SYS$QIO(*efn*,***inet-chan*, IO$_GETSOCKNAME,***iosb*, *astadr*, *astprm*, ***address, addrlen*,0, 0, 0, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=address

OpenVMS usage:	socket_address
type:	structure `sockaddr_in`
access:	write only
mechanism:	by reference

Receives the local or multicast address.

p2=addrlen

OpenVMS usage:	socket_address_length
type:	longword (unsigned)
access:	modify
mechanism:	by reference

On entry, contains the byte length of the space pointed to by *address*. On return, it contains the byte length of the data returned.

Status

EINVAL	*addrlen* argument is invalid.
ENETDOWN	Network was shut down.
SS$_THIRDPARTY	Network is being shut down.

IO$_GETSOCKOPT

Retrieves the value of options associated with a socket.

The TCPWARE_INCLUDE:SOCKET.H file contains definitions for the socket-level options. The IPPROTO_IP level options defined in the TCPWARE_INCLUDE:IN.H file are not compatible with INETDRIVER.

Format

status=SYS$QIO(*efn*,***inet-chan*, IO$_GETSOCKOPT,***iosb*, *astadr*, *astprm*, *level*, ***optname*, *optval*, *optlen*, 0, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=level

OpenVMS usage:	option_level
type:	longword (unsigned)
access:	read only
mechanism:	by value

SOL_SOCKET to change socket options, or **IPPROTO_IP** to change IP options (which requires system UIC, SYSPRV, or BYPASS privilege).

p2=optname

OpenVMS usage:	option_name
type:	longword (unsigned)

access:	read only
mechanism:	by value

TCPware ignores this option where indicated. However, you should include it if you created other routines that set this option to that you do not get an error message.

See *Optnam* and *Optnam* for the allowable values.

p3=optval

OpenVMS usage:	(dependent on *optname*)
type:	byte buffer
access:	write only
mechanism:	by reference

Pointer to a buffer that is to receive the current value of the option. The format of this buffer is dependent on the option requested. Options other than SO_LINGER return a longword value. For those options that are enabled or disabled, 0 is returned if disabled, 1 is returned if enabled.

SO_LINGER uses the following structure for the linger value:

```
structlinger {
  intl_onoff   ;/* option on/off */
  intl_linger  ;/* linger time */
};
```

p4=optlen

OpenVMS usage:	option_length
type:	longword (unsigned)
access:	modify
mechanism:	by reference

On entry, contains the byte length of the space pointed to by *optval*. On return, contains the byte length of the option returned.

Status

EINVAL	*level* specified is invalid.
ENOPROTOOPT	Option is unknown.
ENETDOWN	Network was shut down.
SS$_THIRDPARTY	Network is being shut down.

IO$_IOCTL

Manipulates socket characteristics. The TCPWARE_INCLUDE:IOCTL.H file contains definitions for the IO$_IOCTL request codes.

Note! This function does not support operations to set network interface information.

Format

status=SYS$QIO(*efn*,***inet-chan*, IO$_IOCTL,** *iosb, astadr, astprm,* ***request, argp*, 0, 0, 0, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=request

OpenVMS usage:	ioctl_request
type:	longword (unsigned)
access:	read only
mechanism:	by value

IO$_IOCTL function to perform. The IO$_IOCTL functions are listed in Table 6-2.

p2=argp

OpenVMS usage:	arbitrary
type:	byte buffer
access:	read, write, or modify, depending on *request*

mechanism:	by reference

Pointer to a buffer whose format and function depends on the *request* specified.

Table 6-2 *Request* Argument Values

Value	Description
FIONBIO	Sets/clears nonblocking mode
FIONREAD	Returns bytes available for reading
SIOCADDRT	Adds a routing entry
SIOCATMARK	Returns whether the read point is at the out-of-band (TCP urgent) mark
SIOCDARP	Deletes an address resolution (ARP) entry
SIOCDELRT	Deletes a routing entry
SIOCGARP	Gets an address resolution (ARP) entry
SIOCGIFCONF	Returns list of network interfaces
SIOCGIFADDR	Returns the internet address for an interface
SIOCGIFBRDADDR	Returns the broadcast address for an interface
SIOCGIFDSTADDR	Returns the destination address for a point-to-point interface
SIOCGIFFLAGS	Returns the flags for an interface
SIOCGIFMETRIC	Returns the metric for an interface
SIOCGIFMTU	Returns the minimum transmission unit for an interface
SIOCGIFNETMASK	Returns the network mask for an interface
SIOCSARP	Adds an address resolution (ARP) entry

Status

ENOPROTOOPT	*request* specified is invalid.
ENETDOWN	Network was shut down.
SS$_THIRDPARTY	Network is being shut down.

IO$_LISTEN

Places the stream in a listen state and specifies the maximum number of incoming connections that can be queued waiting to be accepted. This backlog must be specified before accepting a connection on a socket.

Applies to sockets of type SOCK_STREAM only.

Format

status=SYS$QIO(*efn*,***inet-chan*****, IO$_LISTEN,** *iosb, astadr, astprm*,***backlog*****, 0, 0, 0, 0, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=backlog

OpenVMS usage:	connection_backlog
type:	longword (unsigned)
access:	read only
mechanism:	by value

Maximum length of the queue of pending connections. If a connection request arrives when the queue is full, the request is ignored. The backlog queue is limited by the TCPware BACKLOG_LIMIT parameter. If the specified value is greater than BACKLOG_LIMIT, BACKLOG_LIMIT is used.

See the ADD SERVICE command description in Chapter 2, *NETCU Commands*, in the *NETCU Command Reference* for details on setting the BACKLOG_LIMIT parameter.

Status

EISCONN	Socket is already connected.
EOPNOTSUPP	Operation is not supported (such as when the connection on a listening socket).
ENETDOWN	Network was shut down.
SS$_THIRDPARTY	Network is being shut down.

IO$_RECEIVE

Receives data from a socket.

The length of the data received is returned in the second word of the I/O Status Block (*iosb*). A count of 0 indicates an end-of-file condition, or the connection was closed.

For SOCK_DGRAM and SOCK_RAW messages, if the message is too long to fit in the supplied buffer, excess bytes are discarded. For SOCK_STREAM data, no bytes are discarded, even though the amount of data processed on a request may be different than the amount sent.

If no data is available for the socket, the IO$_RECEIVE function waits for data to arrive, unless the socket is in nonblocking mode.

Format

status=SYS$QIO(*efn*,***inet-chan*, IO$_RECEIVE,***iosb*, *astadr*, *astprm*, ***buffer*, *size*, *flags*, *from*, *fromlen*, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=buffer

OpenVMS usage:	arbitrary
type:	write buffer
access:	write only
mechanism:	by reference

Address of the user's buffer.

p2=size

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Byte size of the user's buffer. The value should not be greater than 64000 bytes. The actual number of bytes read is returned in the *iosb*.

p3=flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by value

Control information that affects the IO$_RECEIVE function and is formed by logically OR-ing one or more of the following values:

```
#define  MSG_OOB           0x1/*send out-of-band data*/
#define  MSG_PEEK          0x2*peek at incoming message*/
#define  MSG_NONBLOCKING   0x10/*override blocking state*/
#define  MSG_TIME          0x100/*limit receive wait time*/
```

The MSG_OOB flag causes IO$_RECEIVE to read any out-of-band data arriving on the socket.

The MSG_PEEK flag causes IO$_RECEIVE to read the data present in the socket without removing the data. This allows the caller to view the data, but leave it in the socket for future IO$_RECEIVE functions.

The MSG_NONBLOCKING flag causes IO$_RECEIVE to be a nonblocking request. An `EWOULDBLOCK` status would be returned if the request cannot be completed immediately.

The MSG_TIME flag (for datagram sockets only) causes IO$_RECEIVE to complete within SO_RCVTIMEO seconds with the `ETIMEDOUT` status (if no message is received). Use the IO$_SETSOCKET function to set SO_RCVTIMEO.

p4=from

OpenVMS usage:	special_structure

type:	structure defined below
access:	write only
mechanism:	by reference

For SOCK_DGRAM and SOCK_RAW sockets, the optional *from* argument is a pointer to a structure that contains the address of the socket that sent the packet following completion of the IO$_RECEIVE function. The structure is defined as follows:

```
struct {
  unsigned short Length;
  struct sockaddr_inAddress;
} ;
```

Note! The *from* argument is ignored for SOCK_STREAM sockets.

p5=fromlen

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	read only
mechanism:	by value

For SOCK_DGRAM sockets, byte length (at least 18 bytes) of the buffer pointed to by the *from* argument.

Note! The *fromlen* argument is ignored for SOCK_STREAM sockets.

Status

ECONNRESET	Connection was reset by the peer.
EHOSTUNREACH	Host was unreachable.
EINVAL	*from* argument is invalid, or no out-of-band data was available.
ENETDOWN	Network was shut down.
ENOTCONN	Socket is not connected.

EOPNOTSUPP	Out-of-band request exists on a nonstream socket.
ETIMEDOUT	Connection timed out.
EWOULDBLOCK	Request would block.
SS$_CANCEL	Request was cancelled.
SS$_THIRDPARTY	Network is being shut down.

IO$_SEND

Sends stream data (for stream sockets), messages (for datagram sockets), or raw data to a socket.

IO$_SEND blocks if no buffer space is available at the socket to hold the data to be transmitted, unless the socket was placed in nonblocking mode. For datagram sockets, if the message is too long to pass through the underlying protocol in a single unit, an `EMSGSIZE` status is returned and the message is not transmitted.

Format

status=SYS$QIO(*efn*,***inet-chan*, IO$_SEND,** *iosb*, *astadr*, *astprm*, ***buffer, size, flags, to, tolen*, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=buffer

OpenVMS usage:	arbitrary
type:	write buffer
access:	read only
mechanism:	by reference

Address of the user's buffer.

p2=size

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)

access:	read only
mechanism:	by value

Byte size of the user's buffer. The actual number of bytes sent is returned in the *iosb*. The value should not be greater than 64000 bytes.

p3=flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by value

Control information that affects the IO$_SEND function and is formed by logically ORing one or more of the following values:

```
#define    MSG_OOB             0x1/*send out-of-band data*/
#define    MSG_NONBLOCKING     0x10/*override blocking state*/
```

The MSG_OOB flag causes IO$_SEND to send out-of-band data on sockets that support this operation (e.g., SOCK_STREAM sockets).

The MSG_NONBLOCKING flag causes IO$_SEND to be a nonblocking request. An `EWOULDBLOCK` status would be returned if the request cannot be completed immediately.

Note! For stream sockets, some of the data may have been sent: the amount is returned in the second word of the *iosb*.

p4=to

OpenVMS usage:	socket_address
type:	struct socket address
access:	read only
mechanism:	by reference

For datagram sockets, the optional *to* argument is a pointer to the address to which the packet should be transmitted.

p5=tolen

OpenVMS usage:	socket_address_length
type:	longword (unsigned)
access:	read only
mechanism:	by value

For datagram sockets, the optional *tolen* argument contains the byte length of the address pointed to by the *to* argument.

Note! The *to* and *tolen* arguments are ignored for SOCK_STREAM sockets.

Status

ECONNRESET	Connection was reset by the peer.
EDESTADDRREQ	Destination address was not specified.
EHOSTUNREACH	Host was unreachable.
EINVAL	*to* argument is invalid, or no out-of-band data was available.
EISCONN	Socket is already connected.
EMSGSIZE	Datagram was too large.
ENETDOWN	Network was shut down.
ENETUNREACH	Destination network is unreachable.
ENOTCONN	Socket is not connected.
EOPNOTSUPP	Out-of-band request exists on a nonstream socket.
EPIPE	Connection was broken.

ETIMEDOUT	Connection timed out.
EWOULDBLOCK	Request would block.
SS$_THIRDPARTY	Network is being shut down.

IO$_SETCHAR

Sets special device characteristics for the INET device rather than for the socket attached to it. These operations are normally used by the TCPware master server (NETCP) process only.

Note! IO$_SETCHAR requires LOG_IO privileges.

Format

s**tatus=SYS$QIO(***efn*,***inet-chan*, **IO$_SETCHAR,** *iosb*, *astadr*, *astprm*, ***flags*, 0, 0, 0, 0, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=flags

OpenVMS usage:	mask_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Address of a longword bit mask of one or more of the values that appear in Table 6-3. If IO$_SETCHAR is not called, all options are set to OFF.

Table 6-3 *Flags* Argument Values

Bit	Description
0	If set, makes the device permanent. If clear, makes the device temporary (default).

1	If set, makes the device shareable. If clear, makes the device nonshareable (default).
2	If set, allow the device to be handed off. (The device is not deleted when the last channel is deassigned, but is deleted the next time the last channel is deassigned.)

Status

SS$_NOPRIV	Argument requires LOG_IO privileges.

IO$_SETMODE | IO$M_ATTNAST

Enables an AST to be delivered to the process when out-of-band (TCP urgent) data is received on the socket. This function is similar to the UNIX 4.3BSD SIGURG signal being delivered.

This is a once-only AST. After the AST is delivered, you must explicitly re-enable it using this function if you want the AST to be delivered when future out-of-band data is received.

Format

status=SYS$QIO(*efn*,***inet-chan*, IO$_SETMODE | IO$M_ATTNAST,***iosb*, *astadr*, *astprm*, ***routine, parameter, acmode*, 0, 0, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=routine

OpenVMS usage:	ast_procedure
type:	procedure entry mask
access:	call without stack unwinding
mechanism:	by reference

Address of the AST routine to call when out-of-band data arrives on the socket. To cancel AST delivery, specify *routine* as 0.

p2=parameter

OpenVMS usage:	user_arg
type:	longword (unsigned)

access:	read only
mechanism:	by value

Argument to call the AST routine.

p3=acmode

OpenVMS usage:	access_mode
type:	longword (unsigned)
access:	read only
mechanism:	by value

Access mode for the AST.

IO$_SETSOCKOPT

Manipulates options associated with a socket.

The TCPWARE_INCLUDE:SOCKET.H file contains definitions for the socket-level options. The IPPROTO_IP level options defined in the TCPWARE_INCLUDE:IN.H file are not compatible with INETDRIVER.

Format

status=SYS$QIO(*efn*, ***inet-chan*, IO$_SETSOCKOPT,***iosb*, *astadr*, *astprm*, ***level*, *optname*, *optval*, *optlen*, 0, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=level

OpenVMS usage:	option_level
type:	longword (unsigned)
access:	read only
mechanism:	by value

SOL_SOCKET to change socket options, **IPPROTO_TCP** to change TCP options, or **IPPROTO_IP** to change IP options (which requires a SYSTEM UIC, or the SYSPRV or BYPASS privilege).

p2=optname

OpenVMS usage:	option_name
type:	longword (unsigned)

access:	read only
mechanism:	by value

Option to be manipulated. See Table 6-4 to Table 6-6.

p3=optval

OpenVMS usage:	(dependent on *optname*)
type:	byte buffer
access:	read only
mechanism:	by reference

Pointer to a buffer that contains the value to which the option is to be set. The format of this buffer is dependent on the option requested.

p4=optlen

OpenVMS usage:	option_length
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Byte length of the *optval* buffer

Table 6-4 *Optname* Argument Values for SOL_SOCKET Level

SOL_SOCKET Option	Description
SO_BROADCAST	Enables or disables broadcasting on the socket (ignored)
SO_DEBUG	Enables debugging in the protocol modules (ignored)
SO_DONTROUTE	Prevents routing applied to outgoing messages (ignored)
SO_ERROR	Returns current socket error (if any)
SO_KEEPALIVE	Keeps connections alive

SO_LINGER	Delay (in seconds) before closing a socket (ignored)
SO_OOBINLINE	Leaves received out-of-band data in line
SO_REUSEADDR	Allows local address reuse
SO_RCVBUF	Size of the internal receive buffer
SO_RCVLOWAT	(Ignored)
SO_RCVTIMEO	Receive timeout time (ignored for stream sockets)
SO_SNDBUF	Size of the internal send buffer
SO_SNDLOWAT	(Ignored)
SO_SNDTIMEO	Send timeout time (ignored for datagram sockets)
SO_TYPE	Type of socket (stream or datagram)
SO_USELOOPBACK	Bypasses hardware when possible (ignored)

Table 6-5 *Optname* Argument Values for IPPROTO_TCP Level

IPPROTO_TCP Option	**Description**
TCP_KEEPALIVE	Determines how long an idle socket should remain open

Table 6-6 *Optname* Argument Values for IPPROTO_IP Level

IPPROTO_IP Option	**Description**
IP_OPTIONS (1)	Gets or sets IP options to be sent in subsequent datagrams
IP_MULTICAST_IF (2)	Gets or sets the interface used for sending multicast datagrams
IP_MULTICAST_TTL (3)	Gets or sets the IP time-to-live (TTL) to sent in subsequent datagrams
IP_MULTICAST_LOOP (4)	Gets or sets whether sent multicast datagrams should be looped back locally
IP_ADD_MEMBERSHIP (5)	Adds a multicast group membership for an interface
IP_DROP_MEMBERSHIP (6)	Drops a multicast group membership from an interface

Status

EADDRNOTAVAIL	Address not available for use.
EADDRINUSE	Address already in use.
EINVAL	*level* specified is invalid.

ENETDOWN	Network was shut down.
ENOBUFS	Insufficient memory for requests.
ENOPROTOOPT	Option is unknown.
ETOOMANYREFS	Too many multicast memberships requested.
SS$_THIRDPARTY	Network is being shut down.

IO$_SHUTDOWN

Causes all or part of a full-duplex connection on a socket to be shut down. Can be used to signal an end-of-file to the peer without closing the socket itself, which would also prevent further data from being received.

Format

status=SYS$QIO(*efn*, ***inet-chan,* IO$_SHUTDOWN,** *iosb*, *astadr*, *astprm*, ***how,* 0, 0, 0, 0, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=how

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by value

Part of the full-duplex connection to shut down, as shown in Table 6-7.

Table 6-7 *How* Argument Values

Value	Description
0	Further receive operations are not allowed
1	Further send operations are not allowed
2	Further receive and send operations are not allowed

Status

EINVAL	*how* specified is invalid.
ENETDOWN	Network was shut down.
SS$_THIRDPARTY	Network is being shut down.

IO$_SOCKET

Creates the desired socket type. The three currently supported types are stream sockets (SOCK_STREAM), datagram sockets (SOCK_DGRAM), and raw sockets (SOCK_RAW).

Note! Before issuing the IO$_SOCKET function, an INET channel must first be assigned to the INET0: device, using SYS$ASSIGN.

Format

status=SYS$QIO(*efn*, ***inet-chan,* IO$_SOCKET,** *iosb*, *astadr*, *astprm*, **AF_INET(2),** ***type, protocol*, 0, 0, 0)**

Arguments

chan=inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Channel to the socket.

p1=addressfam=AF_INET (2)

OpenVMS usage:	address_family
type:	longword (unsigned)
access:	read only
mechanism:	by value

Must be **AF_INET (2)**.

p2=type

OpenVMS usage:	socket_type
type:	longword (unsigned)

access:	read only
mechanism:	by value

Semantics of communication using the created socket. The three currently supported types are: **SOCK_STREAM** (for a TCP socket), **SOCK_DGRAM** (for a UDP socket), and **SOCK_RAW** (for a raw socket).

p3=protocol

OpenVMS usage:	protocol_number
type:	longword (unsigned)
access:	read only
mechanism:	by value

For TCP and UDP sockets, the *protocol* argument value must be 0. For raw sockets, a non-zero protocol can be specified.

Status

ENETDOWN	Network was shut down.
EPROTONOSUPPORT	Requested *addressfam*, *type*, or *protocol* is not supported.
SS$_THIRDPARTY	Network is being shut down.

SYS$ASSIGN

Assigns a channel to a device.

Format

status = SYS$ASSIGN(*devnam*, *inet-chan*, *[acmode]*, *[mbxnam]*)

Arguments

devnam

OpenVMS usage:	device_name
type:	character coded text string
access:	read only
mechanism:	by descriptor – fixed length string descriptor

Address of a character string descriptor pointing to the device name string.

inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	write only
mechanism:	by reference

Address of a word into which SYS$ASSIGN writes the channel number.

acmode

OpenVMS usage:	access_mode
type:	longword (unsigned)
access:	read only
mechanism:	by value

Optional access mode associated with the channel. The most privileged access mode used is that of the caller.

mbxnam

OpenVMS usage:	device_name
type:	character coded text string
access:	read only
mechanism:	by descriptor fixed length string descriptor

Optional logical mailbox associated with the device. (Not supported by INETDRIVER.)

Status

See HP's *VMS System Services Reference Manual* for a complete list of status messages.

SYS$CANCEL

Cancels any I/O that is pending on a socket.

The I/O will be completed with an *iosb* status of `SS$CANCEL`.

Outstanding I/O operations are automatically cancelled at image exit.

Format

status = SYS$CANCEL(*inet-chan*)

Argument

inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Number of the channel to be cancelled.

Status

See HP's *VMS System Services Reference Manual* for a complete list of status messages.

SYS$DASSGN

Closes a socket.

When you deassign a channel, any outstanding I/O is completed with an *iosb* status of `SS$CANCEL`.

I/O channels are automatically deassigned at image exit.

Format

status = SYS$DASSGN(*inet-chan*)

Argument

inet-chan

OpenVMS usage:	channel
type:	word (unsigned)
access:	read only
mechanism:	by value

Number of the channel to be deassigned.

Status

See HP's *VMS System Services Reference Manual* for a complete list of status messages.

Sample Programs

The following pair of sample programs, which show the use of stream sockets with SYS$QIO system service calls to the INETDRIVER, are included in the TCPWARE_COMMON:[TCPWARE.EXAMPLES] directory:

- INETDRIVER_CLIENT.C
- INETDRIVER_SERVER.C

They are functionally the same as the TCP_SOCKET_CLIENT.C and TCP_SOCKET_SERVER.C sample programs.

The client calling sequence is as follows:

1 Assign a channel for the connection ($ASSIGN)

2 Create a stream socket (IO$_SOCKET)

3 Connect to the server (IO$_CONNECT)

4 Exchange data (IO$_SEND, IO$_RECEIVE)

5 Close the connection ($DASSGN)

The server calling sequence is as follows:

1 Assign a channel to listen on ($ASSIGN)

2 Create a stream socket to listen on (IO$_SOCKET)

3 Bind the socket to an address (IO$_BIND)

4 Listen for incoming connections (IO$_LISTEN)

5 Assign a channel to accept on ($ASSIGN)

6 Accept the incoming connection (IO$_ACCEPT)

7 Exchange data (IO$_SEND, IO$_RECEIVE)

8 Close the connection ($DASSGN)

To build any one of these applications using DEC C, enter:

```
$ CC/DECC/PREFIX=ALL filename
$ LINK filename

Ctrl/Z
```

To build any one of these applications using VAX C, enter:

```
$ CC/VAXC filename
$ LINK filename, TCPWARE:UCX$IPC/LIB, SYS$INPUT/OPTIONS-
_$  SYS$SHARE:VAXCRTL/SHARE

Ctrl/Z
```

Chapter 7 FTP Library

Introduction

This chapter is for application programmers. It describes the FTP-OpenVMS library routines.

The FTP-OpenVMS library routines provide a programming interface to the FTP protocol. Use the FTP-OpenVMS library routines in your own applications to provide FTP capabilities.

The following routines are included in the library:

FTP_ACCOUNT	Specifies the user account on the remote host
FTP_ALLOCATE_CCB	Allocates a connection control block (CCB)
FTP_APPEND_FILE	Appends a file to a remote file
FTP_AUTH	Requests protected authentication.
FTP_CCC	Requests that the command channel be clear text.
FTP_CHECK_FEATURES	Sends the FEAT command to the FTP server and builds a bit mask of supported FTP optional features.
FTP_CLOSE_CONNECTION	Closes the connection to the remote FTP server
FTP_CREATE_DIRECTORY	Creates a directory on the remote host
FTP_DEALLOCATE_CCB	Deallocates a CCB
FTP_DELETE_DIRECTORY	Deletes a directory on the remote host
FTP_DELETE_FILE	Deletes a file on the remote host
FTP_GET_CCB	Gets information from a CCB field
FTP_GET_FILE	Copies a file from the remote host

FTP_GET_LIST	Gets file listing on the remote host
FTP_GET_NAME_LIST	Gets a filename listing on the remote host
FTP_LOGIN_USER	Authorizes the user on the remote host
FTP_OPEN_CONNECTION	Opens an FTP connection
FTP_PASSWORD	Specifies the user password on the remote host
FTP_PBSZ	Specifies the protection buffer size.
FTP_PRINT_DIRECTORY	Returns the current directory on the remote host
FTP_PROT	Specifies the protection level to use for transfers
FTP_PUT_FILE	Copies a file to the remote host
FTP_QUOTE	Sends an FTP command to the remote host
FTP_RENAME_FILE	Renames a file on the remote host
FTP_SET_DIRECTORY	Sets the remote directory
FTP_SET_DEBUG	Sets the debugging mode
FTP_SET_PASV	Sets passive mode transfers
FTP_SET_STRU	Specifies the file structure
FTP_SET_TYPE	Specifies the data representation type
FTP_USER	Specifies the user on the remote host

These routines allow you to establish and maintain FTP connections with remote hosts that support the FTP protocol.

The FTP-OpenVMS library routines follow the standard OpenVMS conventions for modular library routines. See the OpenVMS documentation on calling modular libraries for more information.

Because FTP-OpenVMS library routines use asynchronous system traps (ASTs), application programs must not run with ASTs disabled for long periods of time. Also, you should not call FTP-OpenVMS library routines at the AST level.

The FTP-OpenVMS library routines are in the TCPWARE_FTPLIB_SHR.EXE shareable library. The symbolic definitions for the various CCB fields and other parameters are in the TCPWARE_INCLUDE:FTPDEF.H

header file. While this file is written for C language, it is very easy to convert to other OpenVMS languages.

Building an FTP Client

The FTP Client sample program is provided in TCPWARE_COMMON:[TCPWARE.EXAMPLES]FTPSAMPLE.C.

To build using DEC C, enter:

```
$ CC/DECC/PREFIX=ALL FTPSAMPLE.C
$ LINK FTPSAMPLE, SYS$INPUT/OPTIONS
 SYS$SHARE:TCPWARE_FTPLIB_SHR/SHARE
SYS$SHARE:TCPWARE_SOCKLIB_SHR/SHARE

Ctrl/Z
```

To build using VAX C, enter:

```
$ CC/VAXC/PREFIX=ALL FTPSAMPLE.C
$ LINK FTPSAMPLE, SYS$INPUT/OPTIONS
SYS$SHARE:TCPWARE_FTPLIB_SHR/SHARE
                  SYS$SHARE:TCPWARE_SOCKLIB_SHR/SHARE
                  SYS$SHARE:VAXCRTL.EXE/SHARE

Ctrl/Z
```

Connection Control Block

The connection control block (CCB) contains all the information required to establish and maintain an FTP connection. Each open connection requires a CCB.

The storage space for the CCB is allocated dynamically. Therefore, the number of simultaneous connections is limited only by your process resources. You can reuse a CCB. You can close one connection and open a new one using the same CCB.

Gain access to the CCB fields using the FTP_GET_CCB library routine as described in *Library Routines*.

Table 7-1 lists CCB fields and their uses:

Table 7-1 CCB Fields

CCB Fields	CCB Uses
FTP_CCB_DC_CHAN	Channel for the FTP data connection (unsigned word)
FTP_CCB_DC_PN	Port number of the data connection (unsigned word)
FTP_CCB_FEATURE_MASK	Longword bit mask of server features reported by the response to the FTP FEAT command sent by the CHECK_FEATURES subroutine

FTP_CCB_FILENAME	Name of the file causing the transfer error (character string descriptor)
FTP_CCB_FOR_IA	IP address of the remote host (unsigned longword)
FTP_CCB_FOR_PN	Remote port number of the control connection (unsigned word)
FTP_CCB_FX_TYPE	Data representation type (character string descriptor with a maximum length of 2); see theFTP_SET_TYPE routine for the possible values
FTP_CCB_LOC_IA	IP address of the local host (unsigned longword)
FTP_CCB_LOC_PN	Local port number of the control connection (unsigned word)
FTP_CCB_LOGGED_IN	Login status (Boolean); true if the user is successfully logged in on the remote host
FTP_CCB_NETCHAN	Channel for the FTP control connection (unsigned word)
FTP_CCB_NODENAME	Node name of the remote host (character string descriptor with a maximum length of 128)
FTP_CCB_OPENED	Connection status (Boolean); true if control connection is successfully opened to the remote host
FTP_CCB_PASSWORD	User password on the remote host (character string descriptor with a maximum length of 64 characters)
FTP_CCB_PASV	Boolean that indicates whether passive mode is enabled (TRUE) or disabled (FALSE).
FTP_CCB_RPLCOD	Server reply code (longword)
FTP_CCB_RPLLEN	Length of the server reply message (longword)
FTP_CCB_RPLTXT	Reply text received from the server (character string descriptor with a maximum length of 512 characters)
FTP_CCB_STAT_BYTES	Number of bytes transferred (longword)
FTP_CCB_STAT_SECONDS	Seconds required for the transfer (double floating point)

FTP_CCB_STATUS	Completion status (unsigned longword)
FTP_CCB_STRU	File structure (character string descriptor); see the FTP_SET_STRU routine for the possible values
FTP_CCB_USERNAME	Username on the remote host (character string descriptor with a maximum length of 64 characters)

Transferring Files

The FTP_APPEND_FILE, FTP_GET_FILE, and FTP_PUT_FILE library routines described in *Library Routines* have two optional arguments: *mode* and *record-size*. These options provide finer control in transferring OpenVMS files.

Using the *mode* argument causes the transfer mode set by FTP_SET_TYPE and FTP_SET_STRU to be ignored. The lower word of the *mode* longword specifies the primary transfer mode. The following symbols define the primary transfer mode:

MODE_C_ASCII	Transfers the file in ASCII format
MODE_C_IMAGE	Transfers the file in IMAGE mode
MODE_C_BINARY	Transfers .OBJ, .STB, .BIN, and .LDA files in BINARY format
MODE_C_BLOCK	Transfers STREAM, STREAM_LF, STREAM_CR and UNDEFINED files in BLOCK mode

The higher word of the *mode* longword defines optional transfer modes. The following symbols define the optional transfer modes:

MODE_M_FCC	Combined with MODE_C_ASCII, transfers the ASCII file with FORTRAN carriage control.
MODE_M_APPEND	Appends the source to a destination file. (Not supported by all servers.)
MODE_M_VARIABLE	Combined with MODE_C_IMAGE, transfers an image file in variable length recode mode, except that all records are the same length. (Applies to local output image files only.)
MODE_M_RECORD	Transfers the file using STRU R so as to communicate the record structure during the copy. (Not supported by all servers.)

MODE_M_RESTART	When transferring the file in STREAM mode performs a restart from where the transfer stopped based upon file size. Requires support of the feature and the CHECK_FEATURES routine to be called beforehand.
MODE_M_VMS	Transfers the file in VMS file mode. Using MODE_M_VMS allows you to transfer any type of RMS file between OpenVMS systems. Note that if specifying this flag, all other flags are ignored.

Error Status Codes

To access TCPware error status codes, such as `TCPWARE_REJECTED`, define them in the code as follows:

```
globalvalue TCPWARE_REJECTED;
```

Then LINK with the definitions of TCPware error messages using the link option:

TCPWARE:SOCKLIB.OLB/INCLUDE=TCPWARE_MSGPTR

For example, the following code checks if the status returned by FTP_PUT_FILE is TCPWARE_REJECTED and takes the appropriate action:

```
globalvalue TCPWARE_REJECTED;
...
status = FTP_PUT_FILE(...);
if (status == TCPWARE_REJECTED)
{
do something;
return status;
}
```

Library Routines

This section describes each of the FTP-OpenVMS library routines. Each function argument is described using standard OpenVMS notation for procedure arguments.

All scalar and buffer arguments are passed by reference. Strings are passed by descriptor. All function arguments must be specified, although some optional arguments may be omitted by passing a 0 (by value).

Each FTP-OpenVMS library routine returns an unsigned longword condition value in R0. The standard system service return codes defined in the $SSDEF macro are used. Gain access to the TCPware error statuses (those beginning with `TCPWARE`) by global value.

These routines are included in the TCPWARE_FTPLIB_SHR.EXE shareable library.

FTP_ACCOUNT

Specifies the user account on the remote host.

Format

FTP_ACCOUNT (*ccb*, *account)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active connection.

account

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Name of the user account on the remote host.

Description

This routine sends the FTP ACCT command to specify the name of the account on the remote host.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_NONODE	Connection is not open

TCPWARE_NEEDACCT	Server requires an account for login
TCPWARE_LOGINFAIL	Login attempt failed
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_ALLOCATE_CCB

Allocates a connection control block (CCB).

Format

FTP_ALLOCATE_CCB (*ccb)*

Argument

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Returns a pointer to the new CCB.

Description

This routine allocates a new CCB, initializes the resources, and returns a pointer to it. A CCB must be allocated before all the subsequent calls to the FTP-OpenVMS library routines.

Condition Values Returned

Any condition value returned by LIB$GET_VM and LIB$GET_EF can be returned.

Example

```
#include <descrip.h>
#include <ssdef.h>
#include "ftpdef.h"
main ()
{
    $DESCRIPTOR (host, "ds.internic.net");
    $DESCRIPTOR (user, "anonymous");
    $DESCRIPTOR (pswd, "your-name@your-site");
    $DESCRIPTOR (dir, "rfc");
    $DESCRIPTOR (file, "rfc-index.txt");
    long ccb, status;
    long debug = DEBUG_REPLY;

    status = SS$_NORMAL;

    ftp_allocate_ccb (&ccb);
.
.
.

    return (status);
}
```

FTP_APPEND_FILE

Appends a file to a remote file.

Format

FTP_APPEND_FILE (*ccb*, *source*, *[destination]*, *[mode]*, *[record-size])*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection for the file transfer.

source

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Source file on the local host.

destination

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Destination file on the remote host. If omitted, the *source* filename and extensions are used.

mode

OpenVMS usage:	longword_signed
type:	longword (signed)
access:	read only
mechanism:	by reference

Transfer mode. See the *Transferring Files* section for the list of available transfer mode codes.

Note! Using *mode* causes the transfer mode set by FTP_SET_TYPE and FTP_SET_STRU to be ignored.

record-size

OpenVMS usage:	longword_signed
type:	longword (signed)
access:	read only
mechanism:	by reference

Record size of the local output file when transferring in IMAGE mode.

Description

This routine sends the APPE command to append the local file to the file on the remote host.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_NONODE	Connection is not open
TCPWARE_OPENIN	Error opening a file for input
TCPWARE_OPENDATA	Failed to open data connection

TCPWARE_SETPORT	Failed to set up data connection
TCPWARE_FILRDERR	Error reading from or sending a file
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_AUTH

Specifies the Authentication/Security mechanism to be used as per RFC 2228 and RFC 4217. Currently only TLS is supported.

Format

FTP_AUTH(*ccb, mechanism*)

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection for the file transfer.

mechanism

OpenVMS usage:	address
type:	string
access:	read only
mechanism:	by descriptor

Specifies the Authentication/Security mechanism to use. Currently only "TLS" is supported.

Description

This routine sends an FTP AUTH command to the FTP server. The FTP_AUTH procedure resets the data transfer parameters to the original state as specified in the RFCs.

Condition Values Returned

SS$_NORMAL	Normal successful completion

TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)
TCPWARE_IVREQUEST	Invalid request
TCPWARE_REQFAIL	Request failed.

FTP_CCC

Specifies that the command channel return to clear text transmission.

Format

FTP_CCC(*ccb*)

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection for the file transfer.

Description

This routine sends an FTP CCC command to the FTP server.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_NONODE	Connection is not open

FTP_CHECK_FEATURES

Sends the FEAT command to the server and builds the bit mask of supported features.

Format

FTP_CHECK_FEATURES(CCB)

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection for the file transfer.

Description

This routine sends an FTP FEAT command to the FTP server and interprets the response to build a bit mask of supported optional features.

Condition Values Returned

None.

FTP_CLOSE_CONNECTION

Closes the connection to the remote FTP server if one is open.

Format

FTP_CLOSE_CONNECTION (*ccb)*

Argument

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB whose active FTP connection is to be closed.

Description

This routine sends an FTP QUIT command to the remote FTP server to close the connection and free up the resources allocated to it.

Condition Value Returned

SS$_NORMAL	Normal successful completion

Example

```
    status = ftp_get_file (ccb, &file, 0, 0, 0);

    if (!(status & 1) )
    {
        printf ("failed to get file %s", file.dsc$a_pointer);
        goto error;
    }

 error:
    ftp_close_connection (ccb);

    return (status);
}
```

FTP_CREATE_DIRECTORY

Creates a directory on the remote host.

Format

FTP_CREATE_DIRECTORY (*ccb*, *directory)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection.

directory

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Directory specification on the remote host.

Description

This routine sends an FTP MKD command that creates a specified directory on the remote server. If the pathname is relative, a subdirectory is created under the current remote directory. If the MKD command is rejected, the XMKD command is used.

Condition Values Returned

SS$_NORMAL	Normal successful completion

TCPWARE_CREATEFAIL	Failed to create the directory
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_DEALLOCATE_CCB

Deallocates a CCB.

Format

FTP_DEALLOCATE_CCB (*ccb)*

Argument

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Identifies the CCB you want to deallocate.

Description

This routine deallocates a CCB that is no longer needed. The CCB is deallocated when it is fully closed. An implicit abort is performed on the connection if the CCB is still in use.

The virtual memory and event flag reserved for the CCB by the FTP_ALLOCATE_CCB routine are freed when this routine is completed.

Condition Value Returned

SS$_NORMAL	Normal successful completion

Example

```
    status = ftp_get_file (ccb, &file, 0, 0, 0);
    if (!(status & 1) )
    {
        printf ("failed to get file %s", file.dsc$a_pointer);
        goto error;
    }
error:
    ftp_close_connection (ccb);
error_nocon:
    ftp_deallocate_ccb (&ccb);
    return (status);
}
```

FTP_DELETE_DIRECTORY

Deletes a directory on the remote host.

Format

FTP_DELETE_DIRECTORY (*ccb*, *directory)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection.

directory

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Directory to be deleted on the remote host.

Description

This routine sends an FTP RMD command to delete a specified directory on the remote server. If the RMD command is rejected, the XRWD command is used.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_DELETEFAIL	Failed to delete the remote directory
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_DELETE_FILE

Deletes a file on the remote host.

Format

FTP_DELETE_FILE (*ccb*, *file)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection.

file

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

File specification of the file to be deleted on the remote host

Description

This routine sends an FTP DELE command to delete a specified file on the remote server.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_DELETEFAIL	Failed to delete the remote file
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_GET_CCB

Gets information from a CCB field.

Format

FTP_GET_CCB (*ccb, field-code, value, length)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB from which information is retrieved.

field-code

OpenVMS usage:	longword_signed
type:	longword (signed)
access:	read only
mechanism:	by reference

Code specifying the CCB field information you are requesting.

See *Connection Control Block* for the valid field codes and their data types. Symbols for these field codes are defined in the TCPWARE_INCLUDE:FTPDEF.H file.

value

OpenVMS usage:	address
type:	longword (unsigned)
access:	write only

mechanism:	by reference

Returned value of the specified CCB field. The value argument is the address of a variable that will receive the value of the CCB field. The data type of this variable depends on the field requested in *field-code*.

length

OpenVMS usage:	word_signed
type:	word (signed)
access:	write only
mechanism:	by reference

Returns the resulting length of the *value* field if the data type of the returned value is STRING.

Description

This routine returns the contents of the specified field from the CCB.

Condition Values Returned

SS$_NORMAL	Normal, successful completion
SS$_BADPARAM	Bad parameter specified

FTP_GET_FILE

Copies a file from a remote host.

Format

FTP_GET_FILE (*ccb*, *source*, *[destination]*, *[mode]*, *[record-size])*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection for the file transfer.

source

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Source file on the remote host.

destination

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Destination file on the local host. If omitted, the *source* filename and extensions are used.

mode

OpenVMS usage:	longword_signed
type:	longword (signed)
access:	read only
mechanism:	by reference

Transfer mode. See *Transferring Files* for a list of available transfer mode codes.

Note! Using *mode* causes the transfer mode set by FTP_SET_TYPE and FTP_SET_STRU to be ignored.

record-size

OpenVMS usage:	longword_signed
type:	longword (signed)
access:	read only
mechanism:	by reference

Record size of the local output file when transferring in IMAGE mode.

Description

This routine sends an FTP RETR command to transfer a remote file to the local host.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_NONODE	Connection is not open
TCPWARE_OPENOUT	Error opening a file for output
TCPWARE_OPENDATA	Failed to open data connection

TCPWARE_SETPORT	Failed to set up data connection
TCPWARE_FILWRTERR	Error writing to or receiving a file
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

Example

```
$DESCRIPTOR (file, "rfc-index.txt");
    long ccb, status;

    status = ftp_get_file (ccb, &file, 0, 0, 0);
    if (!(status & 1) )
    {
        printf ("failed to get file %s", file.dsc$a_pointer);
        goto error;
    }

 error:
    ftp_close_connection (ccb);

    return (status);
}
```

FTP_GET_LIST

Gets a file listing on the remote host.

Format

FTP_GET_LIST (*ccb*, *[directory]*, *output)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection.

directory

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Directory specification on the remote host. If omitted, the current remote directory is used.

output

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Specifies the local file where the output of the remote directory listing is stored.

Description

This routine sends the FTP LIST command to list a remote directory and stores the output to the local file. The output is in the format of the remote operating system.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_NONODE	Connection is not open
TCPWARE_OPENOUT	Error opening a file for output
TCPWARE_OPENDATA	Failed to open data connection
TCPWARE_SETPORT	Failed to set up data connection
TCPWARE_FILWRTERR	Error writing to or receiving a file
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_GET_NAME_LIST

Gets a filename listing on the remote host.

Format

FTP_GET_NAME_LIST (*ccb*, *[directory]*, *output)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection.

directory

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Directory specification on the remote host. If omitted, the current remote directory is used.

output

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Specifies the local file where the output of the remote directory listing is stored.

Description

This routine sends the FTP NAME LIST (NLIST) command to retrieve the name list of the specified remote directory and stores it in the local file. Output is a list of valid file pathnames with each name separated in separate lines. Usually this output is more suitable for machine processing than the list obtained using FTP_GET_LIST.

Condition Values Returned

See the FTP_GET_LIST routine.

FTP_LOGIN_USER

Authorizes the user on the remote host.

Format

FTP_LOGIN_USER (*ccb*, *username*, *password)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB of the active network connection.

username

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Username on the remote host.

password

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Password of the user on the remote host.

Description

This routine sends the FTP USER and PASS commands in sequence to the remote host to log in the specified user with the specified password. The connection must have been opened previously. If there is already a user logged in and the specified username and password are identical to the current values, this routine has no effect.

The user must be logged in on the remote host using this routine or a combination of FTP_USER and FTP_PASSWORD (optionally, FTP_ACCOUNT) performed before calling any other routines for data transfer.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_NONODE	Connection is not open
TCPWARE_NEEDACCT	Server requires an account for login
TCPWARE_LOGINFAIL	Login attempt failed

Example

```
#include <descrip.h>
#include <ssdef.h>
#include "ftpdef.h"

main ()
{
    $DESCRIPTOR (host, "ds.internic.net");
    $DESCRIPTOR (user, "anonymous");
    $DESCRIPTOR (pswd, "your-name@your-site");
    $DESCRIPTOR (dir, "rfc");
    $DESCRIPTOR (file, "rfc-index.txt");

    long ccb, status;
    long debug = DEBUG_REPLY;

    status = SS$_NORMAL;

    ftp_allocate_ccb (&ccb);

    status = ftp_open_connection (ccb, 0, &host, 0, 0);
    if (!(status & 1) )
    {
        printf ("failed to make connection to %s", host.dsc$a_pointer);
        goto error_nocon;
    }

    status = ftp_login_user (ccb, &user, &pswd);
```

```
    if (!(status & 1) )
    {
        printf ("failed to login as %s", user.dsc$a_pointer);
        goto error;
    }.

 error:
    ftp_close_connection (ccb);

    return (status);
}
```

FTP_OPEN_CONNECTION

Opens an FTP connection to a remote host.

Format

FTP_OPEN_CONNECTION (*ccb*, *[ia]*, *[host-name]*, *[port]*, *[timeout])*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB associated with the new connection.

ia

OpenVMS usage:	unsigned_long
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Internet address of the remote host to which you want to connect. The argument is the address of an unsigned longword containing the internet address in internet byte order (for example, internet address 1.2.3.4 is stored as 04030201 hex).

host-name

OpenVMS usage:	character_string
type:	character string
access:	read only

mechanism:	by descriptor

Host name of the remote host to which you want to connect. If the string is a valid internet address in a.b.c.d format, that address is used. Otherwise, the Socket Library's `gethostbyname` routine is used to determine the internet address from the host name.

port

OpenVMS usage:	unsigned_word
type:	word (unsigned)
access:	read only
mechanism:	by reference

Port number for the remote FTP server. If omitted, port number 21 is used.

timeout

OpenVMS usage:	unsigned_longword
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Timeout time in seconds for establishing the FTP control connection. If omitted, the timeout time is 120 seconds (2 minutes).

Description

This routine opens an active FTP connection to the specified remote host. Specify either the internet address or the host name. If you specify both, the internet address is used.

If you call this routine with a CCB with an active connection and the requested remote host is the same, SS$_WASSET is returned and connection is maintained. If you request a different remote host on an active CCB, the existing connection is closed and a new connection is opened.

Condition Values Returned

SS$_NORMAL	Normal successful completion
SS$_WASSET	Connection to specified node already established
SS$_BADPARAM	Missing internet address or host name
SS$_UNREACHABLE	Remote node is unreachable
TCPWARE_REJECTED	Remote server rejected the connection
TCPWARE_OPENCTRL	Failed to open control connection

Example

```
#include <descrip.h>
#include <ssdef.h>
#include "ftpdef.h"

main ()
{
    $DESCRIPTOR (host, "ds.internic.net");
    $DESCRIPTOR (user, "anonymous");
    $DESCRIPTOR (pswd, "your-name@your-site");
    $DESCRIPTOR (dir, "rfc");
    $DESCRIPTOR (file, "rfc-index.txt");
    long ccb, status;
    long debug = DEBUG_REPLY;

    status = SS$_NORMAL;

    ftp_allocate_ccb (&ccb);

    status = ftp_open_connection (ccb, 0, &host, 0, 0);

    if (!(status & 1) )

    {
        printf ("failed to make connection to %s", host.dsc$a_pointer);
        goto error_nocon;
    }

    return (status);
}
```

FTP_PASSWORD

Specifies the user password on the remote host.

Format

FTP_PASSWORD (*ccb*, *password)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active connection.

password

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Password of the user on the remote host.

Description

This routine sends the FTP PASS command to specify the password of the user on the remote host. You must call this routine immediately after the FTP_USER routine if the latter returns a `TCPWARE_NEEDPWD` status.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_NONODE	Connection is not open

TCPWARE_NEEDACCT	Server requires an account for login
TCPWARE_LOGINFAIL	Login attempt failed
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_PBSZ

Sets the protection buffer size as specified in RFC 2228 and RFC 4217. Only "0" (zero) is supported as per RFC 4217.

Format

FTP_PBSZ(*ccb*, *size_string*)

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection for the file transfer.

size_string

OpenVMS usage:	address
type:	string
access:	read only
mechanism:	by descriptor

Sets the size of the protection buffer. Currently only "0" (zero) is supported as per RFC 4217.

Description

This routine sends an FTP PBSZ command to the FTP server.

Condition Values Returned

SS$_NORMAL	Normal successful completion

TCPWARE_IVREQUEST	Invalid request
TCPWARE_NONODE	Connection is not open

FTP_PRINT_DIRECTORY

Returns the current directory on the remote host.

Format

FTP_PRINT_DIRECTORY (*ccb*, *directory*, *[length])*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection.

directory

OpenVMS usage:	char_string
type:	character string
access:	write only
mechanism:	by descriptor

Returns the current directory on the remote host.

length

OpenVMS usage:	word_signed
type:	word (signed)
access:	write only
mechanism:	by reference

Returns the resulting length of the *directory* field.

Description

This routine sends the FTP PWD command to retrieve the current working directory on the remote server. If the PWD command is rejected, the XPWD command is used.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_BADREPLY	Unexpected reply from the server
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_PROT

Sets the protection level specified in RFC 2228 and RFC 4217. Only “C” (clear) and “P” (private) are supported as per RFC 4217

Format

FTP_PROT(*ccb*, *prot_level*)

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection for the file transfer.

prot_level

OpenVMS usage:	address
type:	character
access:	read only
mechanism:	by value

Specifies the protection level to use for data transfers. Only “C” (clear) and “P” (private) are supported as per RFC 4217.

Description

This routine sends an FTP PROT command to the FTP server.

Condition Values Returned

SS$_NORMAL	Normal successful completion

TCPWARE_IVREQUEST	Invalid request
TCPWARE_NONODE	Connection is not open

FTP_PUT_FILE

Copies a file to a remote host.

Format

FTP_PUT_FILE (*ccb*, *source*, *[destination]*, *[mode]*, *[record-size])*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection for the file transfer.

source

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Source file on the local host.

destination

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Destination file on the remote host. If omitted, the *source* filename and extensions are used.

mode

OpenVMS usage:	longword_signed
type:	longword (signed)
access:	read only
mechanism:	by reference

Transfer mode. See *Transferring Files* section for a list of available transfer mode codes.

Note! Using *mode* causes the transfer mode set by FTP_SET_TYPE and FTP_SET_STRU to be ignored.

record-size

OpenVMS usage:	longword_signed
type:	longword (signed)
access:	read only
mechanism:	by reference

Record size of the local output file when transferring in IMAGE mode.

Description

This routine sends the STOR command to copy the local file to the remote host.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_NONODE	Connection is not open
TCPWARE_OPENIN	Error opening a file for input
TCPWARE_OPENDATA	Failed to open data connection

TCPWARE_SETPORT	Failed to set up data connection
TCPWARE_FILRDERR	Error reading from or sending a file
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_QUOTE

Sends an FTP command to the remote server.

Format

FTP_QUOTE (*ccb*, *command)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active connection.

command

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Command to be sent to the remote FTP server.

Description

This routine sends the specified literal command string to the remote FTP server. For example, sending the HELP command with this routine obtains the help information from the remote server (as reply messages), if the remote server supports this command.

Note that you can use this routine only for simple commands, not for commands involving data transfer.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_RENAME_FILE

Renames a file on the remote host.

Format

FTP_RENAME_FILE (*ccb*, *old-file*, *new-file)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection.

old-file

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

File to rename on the remote host.

new-file

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

New filename on the remote host.

Description

This routine sends the FTP RNFR and RNTO commands in sequence to rename a file on the remote server.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_RENAMEFAIL	Failed to rename the remote file
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_SET_DEBUG

Sets the debugging mode for the FTP library.

Format

FTP_SET_DEBUG (*ccb*, *flag*, *[output-routine])*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection.

flag

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Debugging flags to be set. Specify any of the following flag bits:

DEBUG_REPLY	Output the replies received from the remote server
DEBUG_COMMAND	Output the FTP commands sent to the remote server

The symbols for these flags are defined in the TCPWARE_INCLUDE:FTPDEF.H file.

output-routine

OpenVMS usage:	address of pointer to procedure

type:	address of address of procedure value
access:	read only
mechanism:	passing by reference a pointer to the address of a routine

Address of a pointer to a routine that writes debugging text one line at a time. If omitted, LIB$PUT_OUTPUT is used. If you specify the routine of your own, it must have the same calling format as LIB$PUT_OUTPUT.

Description

This routine sets the debugging mode for the FTP Library. When the debugging flags are set, the FTP library outputs the debugging messages by LIB$PUT_OUTPUT, or a user-specified routine.

Condition Value Returned

SS$_NORMAL	Normal successful completion

FTP_SET_DIRECTORY

Sets the remote directory.

Format

FTP_SET_DIRECTORY (*ccb*, *directory)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection.

directory

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Directory specification on the remote host. If omitted, the parent directory is used.

Description

This routine sends the FTP CWD command to change the default directory on the remote server, and the CUP command to change the directory to the parent directory. If the directory specification is null, it changes the directory to the parent directory on the remote server. If the CWD and CUP commands are not accepted, the XCWD and XCUP commands are used.

Condition Values Returned

SS$_NORMAL	Normal successful completion

TCPWARE_SETDEFAIL	Failed to set default directory
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

Example

```
    status = ftp_set_directory (ccb, &dir);

    if (!(status & 1) )
    {
        printf ("failed to change directory to %s", dir.dsc$a_pointer);
        goto error;
    }

error:
    ftp_close_connection (ccb);
```

FTP_SET_PASV

Sets passive mode.

Format

FTP_SET_PASV (*ccb*, *flag)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active and logged-in connection.

flag

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Value 0 disables passive mode; value 1 enables passive mode.

Description

This routine sets PASV (passive) mode. If enabled, the client sends the server the PASV command and the client initiates data connections to the server. If disabled, the client sends the server the PORT command and the server initiates the data connections.

Condition Values Returned

SS$_NORMAL	Normal successful completion

FTP_SET_STRU

Specifies the file structure.

Note! The *mode* arguments with FTP_GET_FILE and FTP_PUT_FILE cause the transfer mode set by FTP_SET_STRU to be ignored.

Format

FTP_SET_STRU (*ccb*, *stru)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active connection for setting the file structure.

stru

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Contains a character code that specifies the file structure. The valid types are:

'F'	File (no record structure)
'R'	Record structure
'V'	VMS structure
'O'	O VMS structure (for backward compatibility)

Description

This routine sends the FTP STRU command to specify the file structure for the subsequent file transfer.

Condition Values Returned

SS$_NORMAL	Normal successful completion
SS$_BADPARAM	Bad parameter
TCPWARE_NONODE	Connection is not open
TCPWARE_UNSXFRFORM	Unsupported transfer format
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_SET_TYPE

Specifies the data representation type.

Note! The *mode* arguments with FTP_GET_FILE and FTP_PUT_FILE cause the transfer mode set by FTP_SET_TYPE to be ignored.

Format

FTP_SET_TYPE (*ccb*, *type)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with the active connection.

type

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Character string code that specifies the file representation type. The first character is the primary transfer mode, as follows:

A	ASCII
I	Image

EBCDIC type is not supported.

Do not use an additional character for transfer mode **I**. For transfer mode **A**, the second character must specify

one of the following format control options:

N	Non-print (no vertical format information)
C	FORTRAN carriage control

TELNET format control is not supported.

Description

This routine sends an FTP TYPE command to specify the file representation for the subsequent file transmission. The default representation type is **`'AN'`** (ASCII transfer in Non-print mode).

Condition Values Returned

SS$_NORMAL	Normal successful completion
SS$_BADPARAM	Bad parameter
TCPWARE_NONODE	Connection is not open
TCPWARE_UNSTYPE	Unsupported data representation type
TCPWARE_CTRLERR	Unexpected error processing the control connection (the connection is closed)

FTP_USER

Specifies the user on the remote host.

Format

FTP_USER (*ccb*, *username)*

Arguments

ccb

OpenVMS usage:	address
type:	longword (unsigned)
access:	read only
mechanism:	by value

Identifies the CCB with an active connection for specifying the username.

username

OpenVMS usage:	char_string
type:	character string
access:	read only
mechanism:	by descriptor

Username on the remote host.

Description

This routine sends an FTP USER command to the remote host to specify the user on the remote host. It may require a subsequent call to FTP_PASSWORD to complete the login. If there is already a user logged in on the specified connection, that user is logged out.

Condition Values Returned

SS$_NORMAL	Normal successful completion
TCPWARE_NEEDPWD	Server requires a password for login
TCPWARE_LOGINFAIL	Login attempt failed

Chapter 8 Socket Library

Introduction

This chapter describes the Socket Library to use for your particular programming application.

TCPware provides a Socket Library if you are running a version of VMS earlier than 5.3 or are using the Remote Procedure Call (RPC) routines. However, Process Software does not recommend that you use the TCPware Socket Library for later versions of OpenVMS.

HP provides a C Socket Library you should use for VMS Version 5.3 and later. HP provides a collection of VAX C, DEC C, and DEC C++ subroutines that closely emulates the UNIX socket functions.

Note! For OpenVMS V5.3 and later, whether the compiler is VAX C, DEC C or DEC C++, network programmers should use HP's C Socket Library routines and header files. In the case of the compiler being DEC C or C++, programmers MUST use HP's library and header files. This applies to VAX, Alpha and I64 systems.

See HP's *VAX C Run Time Library Manual* or *DEC C Language Reference Manual* for information on these socket library routines.

See *Appendix A*, for details on the TCPware Socket Library routines.

Transitioning to the C Socket Library: Include (Header) Files

HP provides header files for its C Socket Library that are similar to those provided by TCPware.

The header files TCPware provides in its Socket Library are described in *Appendix A*.

To use the HP C Socket Library header files:

- If you are transitioning an existing VAX C socket application, change any `#include` statements for TCPware's header files to reference HP's header files. For example:

```
#include "tcpware_include:netdb.h"
```

becomes:

```
#include <netdb.h>
```

Then compile as follows:

– On the VAX C command line:

```
$ cc prog.c
```

– On the DEC C command line for VAX:

```
$ cc /stand=vaxcprog.c
```

– On the DEC C command line for Alpha and I64:

```
$ cc /stand=vaxc/nomember_align/assume=noalignedprog.c
```

- If you are porting or developing an ANSI C or C++ application, use the HP header files as shown above. Then compile the application (in the case of DEC C, omit the **/stand=vaxc** option).

Note! If you are developing a new program using DEC C, compile using:

```
$ cc /prefix_library_entries=all_entriesprog.c
```

Transitioning to the C Socket Library: Linking Applications

You can then link against HP's C Socket Library as follows:

- For VAX C:

```
$ link prog,sys$input/options
tcpware:ucx$ipc/lib
sys$share:vaxcrtl/share
```

- For DEC C on the VAX, Alpha and I64, and for DEC C++:

```
$ link prog
```

The link procedure for the TCPware Socket Library is described in Appendix A.

Sample Programs

The following sample programs are included in the TCPWARE_COMMON:[TCPWARE.EXAMPLES] directory:

- TCP_SOCKET_CLIENT.C
- TCP_SOCKET_SERVER.C
- UDP_SOCKET_CLIENT.C
- UDP_SOCKET_SERVER.C

The TCP_SOCKET_CLIENT.C and TCP_SOCKET_SERVER.C pair of programs provide a self-declared ECHO server that sequentially accepts client connections and echoes back the client messages. The UDP_SOCKET_CLIENT.C and UDP_SOCKET_SERVER.C pair of programs provide a self-declared DISCARD server that can receive (and discard) datagrams from multiple clients.

These programs are functionally equivalent to the BGDRIVER sample programs in Chapter 2, *UCX Compatibility Services*.

To build any one of these applications using DEC C, enter:

```
$ CC/DECC/PREFIX=ALL filename
$ LINK filename
Ctrl/Z
```

To build any one of these applications using VAX C, enter:

```
$ CC/VAXC filename
$ LINK filename, TCPWARE:UCX$IPC/LIB, SYS$INPUT/OPTIONS-
_$  SYS$SHARE:VAXCRTL/SHARE
Ctrl/Z
```

Appendix A, TCPware Socket Library, includes sample programs for the TCPware Socket Library. These require minor modification to be used in C Socket Library applications.

Debugging programs that use the C socket library

The logical TCPWARE_SOCKET_TRACE can be defined to get a trace of socket library operations and status values returned from TCPware to aid in debugging of applications. The details of the interpretation of the logical are in the table below:

Table 3-1 Interpretation of TCPWARE_SOCKET_TRACE logical

number	**1 - control operations** **2 - read operations** **4 - write operations** These values can be ORed together to get any combination of operations.
anything else	Interpreted as a (partial) file specification with the default specification being **SYS$SCRATCH:TCPWARE_SOCKET_**<process_name>**.LOG** This can be useful for services that are started up by NETCU or another listening service and that create separate processes. The logical can be defined at the SYSTEM level and the socket library routines will generate a separate log file for each process. When this format is used the log consists of all three classes (**control, read** and **write**) operations.

Chapter 9 TELNET Library

Introduction

This chapter is for application programmers. It describes the TELNET library routines.

The TELNET library routines provide a programming interface to the TELNET protocol. Use the TELNET library routines in your own applications to provide TELNET capabilities.

This chapter does not describe the TELNET protocol.

The following routines are included in the library:

TEL_ABORT_CONNECTION	Aborts a TELNET connection
TEL_ALLOCATE_CCB	Allocates a connection control block (CCB)
TEL_CLOSE_CONNECTION	Closes a TELNET connection
TEL_CREATE_TERMINAL	Opens and allocates a TELNET device
TEL_DEALLOCATE_CCB	Deallocates a CCB
TEL_GET_CCB	Gets the value of a CCB field
TEL_OPEN_CONNECTION	Opens a TELNET connection
TEL_RECEIVE_DATA	Receives data
TEL_SEND_COMMAND	Sends TELNET commands
TEL_SEND_DATA	Sends data
TEL_SET_CCB	Sets the value of a CCB field

These routines allow you to establish and maintain TELNET connections with remote hosts that support the TELNET protocol. You can establish multiple simultaneous TELNET connections; the number is limited only by the resources available to your process.

The TELNET library routines follow the standard OpenVMS conventions for modular library routines. See the OpenVMS documentation on calling modular libraries for more information.

Because TELNET library routines use asynchronous system traps (ASTs), application programs must not run with ASTs disabled for long periods of time.

The TELNET library routines are in the TCPWARE:TELLIB.OLB object library. The symbolic definitions for the various connection control block (CCB) fields are in the TCPWARE_INCLUDE:_CCBFLD.H header file. While this file is written for VAX C, it is easy to convert to other VAX languages.

The TELNET Client sample program is provided in TCPWARE_COMMON:[TCPWARE.EXAMPLES]TELNET_SAMPLE.C.

To build using DEC C, enter:

```
$ CC/DECC/PREFIX=ALL TELNET_SAMPLE.C
$ LINK FTPSAMPLE, SYS$INPUT/OPTIONS
TCPWARE:TELLIB/LIB
SYS$SHARE:TCPWARE_SOCKLIB_SHR/SHARE
Ctrl/Z
```

To build using VAX C, enter:

```
$ CC/VAXC/PREFIX=ALL TELNET_SAMPLE.C
$ LINK FTPSAMPLE, SYS$INPUT/OPTIONS
TCPWARE:TELLIB/LIB
SYS$SHARE:TCPWARE_SOCKLIB_SHR/SHARE
                SYS$SHARE:VAXCRTL.EXE/SHARE
Ctrl/Z
```

Connection Control Block

This section describes the connection control block (CCB) that contains all the information required to establish and maintain a TELNET connection. Each open connection requires a CCB.

The storage space for the CCB is allocated dynamically. Therefore, the number of simultaneous connections is limited only by your process resources. A CCB can be reused; you can close one connection and then open a new one using the same CCB.

Note! The *ccb-ptr* argument in the library routines must remain in scope at a fixed address. Also, the *ccb-ptr* is to be used with a single CCB from the time of TEL_ALLOCATE_CCB to TEL_DEALLOCATE_CCB.

You can access the CCB fields with the TEL_GET_CCB and TEL_SET_CCB library routines as described below. Table 9-1 lists the CCB fields and their uses:

Table 9-1 CCB Fields

CCB Fields	CCB Uses
CCB_ASTRTN	Address of the AST routine to be called when data is received (unsigned longword). Cleared once the AST is declared. You must explicitly reset this field each time you want the AST to be used. Use with TEL_GET_CCB or TEL_SET_CCB, or with TEL_OPEN_CONNECTION.

CCB_CHAN	Channel number for the TCP0: device (unsigned word), defined only when a connection is open (otherwise 0). (Use the CCB_ISOPEN field described below to determine if a connection is open instead of checking for a non-zero channel number.) Use with TEL_GET_CCB.
CCB_CMDRTN	Address of your TELNET command processing routine that you must use with TEL_OPEN_CONNECTION (unsigned longword). Also use with TEL_GET_CCB or TEL_SET_CCB.
CCB_EF	Event flag number to set when there is received data (unsigned longword). Can also be used as an argument to TEL_OPEN_CONNECTION. Use event flag 0 to disable the use of an event flag. Use with TEL_GET_CCB.
CCB_ISOPEN	Connection status (unsigned byte). Value is 1 if the connection is open, 0 if not. Use with TEL_GET_CCB.
CCB_LIA	Connection's local internet address (unsigned longword), in internet byte order (for example, internet address 1.2.3.4 is stored as 04030201 hex). Valid only when a connection is open. Use with TEL_GET_CCB.
CCB_LPORT	Local port number for the connection (unsigned word). Valid only when a connection is open. Use with TEL_GET_CCB.
CCB_RCVBCT	Total byte count for the connection (unsigned longword). The counter wraps with an overflow. Use with TEL_GET_CCB.
CCB_RCVBCNT	Number of unread bytes in the internal receive buffer that reflects the amount of raw data in the internal buffer (unsigned word). May be larger than the actual number of user data bytes you read due to requirements of the TELNET protocol. Use with TEL_GET_CCB.
CCB_RCVIOC	Total count of internal network read operations performed for the connection (unsigned longword). The counter wraps with an overflow. Use with TEL_GET_CCB.
CCB_RIA	Connection's remote internet address (unsigned longword), in internet byte order (for example, internet address 1.2.3.4 is stored as 04030201 hex). Can be set only when used as an argument to TEL_OPEN_CONNECTION. Use with TEL_GET_CCB.
CCB_RPORT	Remote port number for the connection (unsigned word), the default being 23 (for the TELNET server). You can only set this field when specified as an argument to TEL_OPEN_CONNECTION. Valid only when a connection is open. Use with TEL_GET_CCB.

CCB_SNDBCNT	Number of untransmitted bytes in the internal send buffer (unsigned word) that reflects the amount of raw data in the internal buffer. May be larger than the actual number of user data bytes you send due to requirements of the TELNET protocol. Use with TEL_GET_CCB.
CCB_SNDBCT	Total count of bytes sent for the connection (unsigned longword). The counter wraps with an overflow. Use with TEL_GET_CCB.
CCB_SNDIOC	Total count of internal write operations for the connection (unsigned longword). The counter wraps with an overflow. Use with TEL_GET_CCB.
CCB_TIMO	Time-out value (in seconds) for the connection (unsigned longword), the default being 120 seconds (2 minutes). Can also be used as an argument to TEL_OPEN_CONNECTION. Use with TEL_GET_CCB.
CCB_UFLAGS	Communicates state information between the user's application and the library routines (unsigned longword). Table 9-2 lists the only defined bits. All other bits are reserved and must be zero. Your program should set and clear bits 0 and 1 in this field when TELNET negotiates the TRANSMIT-BINARY option. Use with TEL_GET_CCB or TEL_SET_CCB.

If you want to enable SYNCH signal processing mode, set bit 31 before opening a connection. The TELNET library routines enter this mode when they receive urgent data. They remain in this mode until they encounter the TELNET DATA MARK command and no more urgent data exists. In this mode, TEL_RECEIVE_DATA discards all data and the EC and EL commands. All other TELNET commands are passed to your command processing routine.

Table 9-2 CCB_UFLAGS Bits

Bit...	**If set, means...**
0	Enables local binary mode
1	Enables remote binary mode
31	Enables SYNCH signal processing

CCB_USER1	User-definable (unsigned longword). Use with TEL_GET_CCB or TEL_SET_CCB.
CCB_USER2	User-definable (unsigned longword). Use with TEL_GET_CCB or TEL_SET_CCB.

The TELNET library routines use additional fields internally that are not accessible to application programs.

Library Routines Reference

This section describes each of the TELNET library routines.

All scalar and buffer arguments are passed by reference. Strings are passed by descriptor. You must use all function arguments, although you can omit some optional ones by passing a 0 (by value).

Each TELNET library routine returns an unsigned longword condition value in R0 and use the standard system service return codes defined in the $SSDEF macro.

The function arguments are described using the following "dot" notation:

argument-name.access-data-type.passing-mechanism parameter-form

For example, the following argument specification indicates the CCB address pointer, having modify access and a passing mechanism by reference:

ccb-ptr.ma.r

See the OpenVMS Run-Time Library documentation for a complete description of this procedure argument notation.

These routines are included in the TCPWARE:TELLIB.OLB object library. You must also include the Socket Library when linking.

See the beginning of this chapter for details on linking.

TEL_ABORT_CONNECTION

Aborts a connection.

Format

ret-status.wlc.v = TEL_ABORT_CONNECTION(*ccb-ptr*)

Argument

*ccb-ptr.*ma.r

Address of a pointer to the connection control block (CCB) used by all the TELNET library routines to identify a connection.

Description

Aborts the TELNET connection, cancels all outstanding sends and receives, and returns the SS$_VCBROKEN status.

Condition Values Returned

SS$_NORMAL	Connection aborted (connection closed)

TEL_ALLOCATE_CCB

Allocates a connection control block (CCB).

Format

ret-status.wlc.v = TEL_ALLOCATE_CCB(*ccb-ptr*,*rcv-buf-size*,*snd-buf-size*)

Arguments

*ccb-ptr.*wa.r

Address of an unsigned longword that will receive a pointer to the CCB used by all the TELNET library routines to identify a connection.

SeeConnection Control Block for a description of the CCB.

*rcv-buf-size.*rwu.r

Address of an unsigned word containing the number of bytes allocated to the internal receive buffer. There is no minimum buffer size by default. However, use at least 64 bytes as an absolute minimum, with 1024 bytes recommended to enhance performance.

*snd-buf-size.*rwu.r

Address of an unsigned word containing the number of bytes allocated to the internal send buffer. There is no minimum buffer size by default. However, use at least 64 bytes as an absolute minimum, with 1024 bytes recommended to enhance performance.

Description

Allocates memory for a CCB using the LIB$GET_VM routine. Memory is allocated from the default zone. The CCB and the internal buffers are contiguous in memory. An internal local event flag will also be allocated from the second local event flag cluster.

You should only access the CCB data fields using the TEL_GET_CCB and TEL_SET_CCB routines. You must allocate a CCB before calling any other TELNET library routine.

Condition Values Returned

Any condition value returned by LIB$GET_VM and LIB$GET_EF.

TEL_CLOSE_CONNECTION

Closes a TELNET connection.

Format

ret-status.wlc.v = TEL_CLOSE_CONNECTION(*ccb-ptr*)

Argument

*ccb-ptr.*ma.r

Address of a pointer to the CCB.

Description

Call this routine when you finished sending data to the remote host or when the remote host notifies you that it closed its end of the connection.

This routine initiates a close on the local end of an open connection. For a connection to be fully closed, both ends of the connection (local and remote) must be closed. No more data can be sent or received once a connection is fully closed.

Additional data can be received until the remote end closes its side of the connection. Therefore, you must call the TEL_RECEIVE_DATA routine until the SS$_VCCLOSED status is returned.

Here are the two typical close situations:

Remote end closes first	TEL_RECEIVE_DATA returns the SS$_VCCLOSED status, then you call TEL_CLOSE_CONNECTION to close your end of the connection. TEL_CLOSE_CONNECTION returns when the connection is fully closed.
Local end closes first	You call TEL_CLOSE_CONNECTION to close your end of the connection. TEL_CLOSE_CONNECTION returns, but the remote end of the connection is still open. You must call TEL_RECEIVE_DATA until it returns the SS$_VCCLOSED status.

In general TEL_RECEIVE_DATA must return SS$_VCCLOSED. This can happen before or after you call TEL_CLOSE_CONNECTION.

Condition Values Returned

SS$_ILLSEQOP	Connection not open
SS$_NORMAL	Success, close initiated or successful
SS$_THIRDPARTY	Software was shut down (connection closed)
SS$_TIMEOUT	Connection timed out (connection closed)
SS$_VCBROKEN	Connection broken (connection closed)

TEL_CREATE_TERMINAL

Opens and allocates a connection to an NTA device so that you can use a terminal device with TELNET.

Format

ret-status.wlc.v=TEL_CREATE_TERMINAL(*ccb-ptr*,*[ia]*,*[host]*,*cmd-rtn*,*[efn]*,
[ast-addr],*[port]*,
[timeout],*[flags]*)

Arguments

*ccb-ptr.*ma.r

Address of a pointer to the CCB. The CCB must be in a closed state.

*ia.*rlu.r

Address of an unsigned longword containing the remote host's internet address in internet byte order (for example, internet address 1.2.3.4 is stored as 04030201 hex). If omitted or 0, the *host* argument determines the remote host.

*host.*rt.ds

Address of a text string descriptor of the remote host's host name. If the string is a valid *a.b.c.d* type internet address, that address is used. Otherwise, the Socket Library's `gethostbyname` routine determines the internet address from the host name. If omitted or a null string, the *ia* argument determines the remote host.

*cmd-rtn.*szem.r

Address of the user's command processing routine called each time a TELNET command is received. Inputs to the command processing routine are the CCB of the connection, a buffer containing one complete command, and a buffer byte count.

SeeUser Command Processing .

*efn.*rlu.r

Address of an unsigned longword containing the event flag to use. The event flag is set whenever data is available in the receive buffer, and cleared when the last byte of data is removed. It is up to you to allocate the event flag if necessary. You must never set or clear the event flag. If omitted, the previous value of this field is used; if 0, the field is cleared. If no event flag is specified, you will have to use another method, such as an AST routine, to determine when data is received.

*ast-adr.*szem.r

Address of an AST routine to be called when data is received. This AST routine is declared when received data is available. You must set the *ast-adr* each time you want to be notified of received data. Use the TEL_SET_CCB routine to set the *ast-adr* after a connection is open.

A pointer to the CCB is passed to the AST routine by reference as the AST argument. You can use the CCB_USER1 and CCB_USER2 fields if you want to pass arguments to the AST routine.

*port.*rwu.r

Address of an unsigned word containing the 16-bit port number for the connection (normally 23). Supplied for use in special applications where the remote server is not a standard TELNET server. If omitted or 0, the default value 23 is used.

*timeout.*rlu.r

Address of an unsigned longword with the number of seconds to wait on the connection before timeout. If omitted or 0, the default value 120 (2 minutes) is used.

Description

Call this routine when you want to open and allocate a connection to an NTA device so that you can use a terminal device with TELNET.

TEL_DEALLOCATE_CCB

Deallocates a CCB.

Format

ret-status.wlc.v = TEL_DEALLOCATE_CCB(*ccb-ptr*)

Argument

*ccb-ptr.*ma.r

Address of a pointer to the CCB.

Description

Gets rid of a CCB you no longer need. The CCB is deallocated when it is fully closed. An implicit abort is performed on the connection if the CCB is still in use.

The virtual memory and event flag reserved for the CCB by the TEL_ALLOCATE_CCB routine is freed when this routine completes.

Condition Values Returned

SS$_NORMAL	Success

TEL_GET_CCB

Gets information from a CCB field.

Format

ret-status.wlc.v = TEL_GET_CCB(*ccb-ptr*,*field-code*,*value*)

Arguments

*ccb-ptr.*ma.r

Address of a pointer to the CCB. The CCB for the connection does not need to be open unless otherwise specified.

*field-code.*rwu.r

Address of an unsigned word containing the symbolic value for the CCB field. The symbolic definitions for the CCB fields are in the TCPWARE_INCLUDE:_CCBFLD.H file.

SeeConnection Control Block for a description of the CCB.

*value.*wz.r

Address of a variable that receives the value of the CCB field. The data type of this variable depends on the field requested in *field-code*.

Description

The value of the CCB field you requested is returned in the argument value. The data type of *value* should be the same as the field you requested.

Condition Values Returned

SS$_BADPARAM	Bad *field-code* specified
SS$_NORMAL	Normal, successful completion

TEL_OPEN_CONNECTION

Opens a TELNET connection to a remote host.

Format

ret-status.wlc.v = TEL_OPEN_CONNECTION(*ccb-ptr,[ia],[host],cmd-rtn,[efn],[ast-adr],[port], [timeout]*)

Arguments

*ccb-ptr.*ma.r

Address of a pointer to the CCB.

*ia.*rlu.r

Address of an unsigned longword containing the remote host's internet address in internet byte order (for example, internet address 1.2.3.4 is stored as 04030201 hex). If omitted or 0, the *host* argument determines the remote host name.

*host.*rt.ds

Address of a text string descriptor of the remote host's host name. If the string is a valid *a.b.c.d* type internet address, that address is used. Otherwise, the Socket Library's `gethostbyname` routine determines the internet address from the host name. If omitted or a null string, the *ia* argument determines the remote host.

*cmd-rtn.*szem.r

Address of the user's command processing routine called each time a TELNET command is received. Inputs to the command processing routine are the CCB of the connection, a buffer containing one complete command, and a buffer byte count.

SeeUser Command Processing .

*efn.*rlu.r

Address of an unsigned longword containing the event flag to use. The event flag is set whenever data is available in the receive buffer, and cleared when the last byte of data is removed. It is up to you to allocate the event flag if necessary. You must never set or clear the event flag. If omitted, the previous value of this field is used; if 0, the field is cleared. If no event flag is specified, you will have to use another method, such as an AST routine, to determine when data is received.

*ast-adr.*szem.r

Address of an AST routine to be called when data is received. This AST routine is declared when received data is available. You must set the *ast-adr* each time you want to be notified of received data. Use the TEL_SET_CCB routine to set the *ast-adr* after a connection is open.

A pointer to the CCB is passed to the AST routine by reference as the AST argument. You can use the CCB_USER1 and CCB_USER2 fields if you want to pass arguments to the AST routine.

*port.*rwu.r

Address of an unsigned word containing the 16-bit port number for the connection (normally 23). Supplied for use in special applications where the remote server is not a standard TELNET server. If omitted or 0, the default value 23 is used.

*timeout.*rlu.r

Address of an unsigned longword with the number of seconds to wait on the connection before timeout. If omitted or 0, the default value 120 (2 minutes) is used.

Description

Opens an active TELNET connection to the remote host specified in *ia* or *host*. A network read operation is automatically initiated.

Specify either the internet address or host name. If you specify both, the internet address is used.

See the *Management Guide* for details on internet addresses and hostnames.

A connection must be opened before you can send or receive data or commands.

Condition Values Returned

SS$_BADPARAM	Invalid I/O channel, missing internet address or host name, or no route exists to the specified internet address
SS$_NORMAL	Connection is open
SS$_NOSUCHNODE	*ia* was not specified, and *host* was not found by `gethostbyname`
SS$_THIRDPARTY	Software was shut down (connection closed)
SS$_TIMEOUT	Connection timed-out (connection closed)
SS$_VCBROKEN	Connection broken (connection closed)

Any condition value returned by the $ASSIGN and $QIO system services.

TEL_RECEIVE_DATA

Receives data from a remote host.

Format

ret-status.wlc.v = TEL_RECEIVE_DATA(*ccb-ptr*,*buffer-size*,*buffer*,*byte-count*)

Arguments

ccb-ptr.ma.r

Address of a pointer to the CCB.

The *ccb-ptr* variable must remain in scope at a fixed address. Also, the *ccb-ptr* is to be used with a single CCB from the time of TEL_ALLOCATE_CCB to TEL_DEALLOCATE_CCB.

*buffer-size.*rwu.r

Address of an unsigned word containing the number of bytes you are willing to receive.

*buffer.*wbu.ra

Address of the first byte of the buffer. The buffer must be at least *buffer-size* bytes in length.

*byte-count.*wwu.r

Address of an unsigned word variable that receives the length of the data in the buffer.

Description

Call this routine whenever data is available. You can use ASTs or event flags to determine when data is available. You must also call this routine after you initiated a close from the local end, as the remote host may still be open and sending data. When the remote host closes, TEL_RECEIVE_DATA returns a status of SS$_VCCLOSED.

If you are using ASTs, you must set the AST address each time you want to be notified by an AST that data is available. This is typically done before returning from your AST routine.

SeeTEL_SET_CCB on how to set the AST address or event flag number.

Do not use this routine to poll for data since that would waste CPU time.

Pairs of IAC characters (ASCII 255) are translated to a single IAC, and a NUL character (ASCII 0) following a carriage return is removed as dictated by the TELNET protocol. When in remote binary mode (bit 1 is set in CCB_UFLAGS) a NUL following a carriage return is not removed.

This routine ensures that TELNET commands and data are processed in the order in which they are received. It does this by copying TELNET data to your buffer until your buffer is full, there is no more data, or a TELNET command is encountered. If a TELNET command is encountered and data was copied to your buffer, it is not processed until the next call to TEL_RECEIVE_DATA, at which time any additional data following the TELNET command or commands is also returned.

When the *byte-count* argument is 0, all data (and commands) have been processed. If the remote end closes the connection, a SS$_VCCLOSED status is returned after the last byte is transferred. You must call TEL_CLOSE_CONNECTION to fully close the connection.

Condition Values Returned

SS$_ILLSEQOP	Connection not open
SS$_NORMAL	Success, data was copied to internal buffer
SS$_THIRDPARTY	Software was shut down (connection closed)
SS$_TIMEOUT	Connection timed-out (connection closed)
SS$_VCBROKEN	Connection broken (connection closed)
SS$_VCCLOSED	Connection closed by remote host (local end still open)

TEL_SEND_COMMAND

Sends TELNET commands to the remote host. Commands include option commands and control functions (such as BREAK and AYT).

Format

ret-status.wlc.v = TEL_SEND_COMMAND(*ccb-ptr*,*buffer*,*byte-count*)

Arguments

*ccb-ptr.*ma.r

Address of a pointer to the CCB.

*buffer.*rbu.ra

Address of the first byte of your data. The commands must be complete and correct. Command data *must* include the IAC character(s) where appropriate.

*byte-count.*rwu.r

Address of an unsigned word containing the number of bytes in the command buffer.

Description

This routine is similar to the TEL_SEND_DATA routine. The only difference is that no processing is done to the command data. The buffer is transmitted with IAC and CR characters left as is.

Condition Values Returned

SS$_ILLSEQOP	Connection not open
SS$_NORMAL	Success, data was copied to internal buffer
SS$_THIRDPARTY	Software was shut down (connection closed)
SS$_TIMEOUT	Connection timed-out (connection closed)
SS$_VCBROKEN	Connection broken (connection closed)

TEL_SEND_DATA

Sends data to the remote host.

Format

ret-status.wlc.v = TEL_SEND_DATA(*ccb-ptr*,*buffer*,*byte-count*)

Arguments

*ccb-ptr.*ma.r

Address of a pointer to the CCB.

*buffer.*rbu.ra

Address of the first byte of your data.

*byte-count.*rwu.r

Address of an unsigned word containing the number of bytes in the data buffer.

Description

Sends the data in your buffer to the remote host. This routine will not return until all of the data is copied to the internal transmit buffer.

Usually data is immediately copied to the internal buffer, and control is returned to your application program. However, if the internal transmit buffer is full, TEL_SEND_DATA waits until there is room in the internal buffer before copying the data and returning. Buffer space becomes available as data is delivered to the remote host.

IAC characters (ASCII 255) in your buffer are doubled and carriage returns not followed by line feeds have a NUL character (ASCII 0) inserted following the carriage return as dictated by the TELNET protocol. When in local binary mode (bit 0 is set in CCB_UFLAGS) a NUL character is not inserted after a carriage return.

Condition Values Returned

SS$_ILLSEQOP	Connection not open
SS$_NORMAL	Success, data was copied to internal buffer
SS$_THIRDPARTY	Software was shut down (connection closed)
SS$_TIMEOUT	Connection timed-out (connection closed)
SS$_VCBROKEN	Connection broken (connection closed)

TEL_SEND_URGENT

Sends TELNET commands to the remote host using TCP urgent notification. Commands include option commands and control functions (such as BREAK and AYT).

Format

ret-status.wlc.v = TEL_SEND_URGENT(*ccb-ptr*,*buffer*,*byte-count*)

Arguments

*ccb-ptr.*ma.r

Address of a pointer to the CCB.

*buffer.*rbu.ra

Address of the first byte of your data. The commands must be complete and correct. Command data *must* include the IAC character(s) where appropriate.

*byte-count.*rwu.r

Address of an unsigned word containing the number of bytes in the command buffer.

Description

This routine is similar to the TEL_SEND_COMMAND routine. The only difference is that TEL_SEND_URGENT sends data using TCP urgent notification.

Condition Values Returned

SS$_ILLSEQOP	Connection not open
SS$_NORMAL	Success, data was copied to internal buffer
SS$_THIRDPARTY	Software was shut down (connection closed)
SS$_TIMEOUT	Connection timed-out (connection closed)
SS$_VCBROKEN	Connection broken (connection closed)

TEL_SET_CCB

Sets the value of a CCB field.

Format

ret-status.wlc.v = TEL_SET_CCB(*ccb-ptr*,*field-code*,*value*)

Arguments

*ccb-ptr.*ma.r

Address of a pointer to the CCB. The CCB does not need to be open.

field-code.rwu.r

Address of an unsigned word containing the symbolic value for the CCB field. The symbolic definitions for the CCB fields are in the TCPWARE_INCLUDE:_CCBFLD.H file.

SeeConnection Control Block for a description of the CCB.

*value.*rz.r

Address of a variable that receives the value of the CCB field. The data type of this variable depends on the field requested in *field-code*.

Note! When you set the CCD_ASTRTN or the CCB_CMDRTN field, make sure that *value* is the address of the address of the routine.

Description

The CCB field you requested is set to the value specified in *value*. The data type of *value* should be the same as the field you specified.

Condition Values Returned

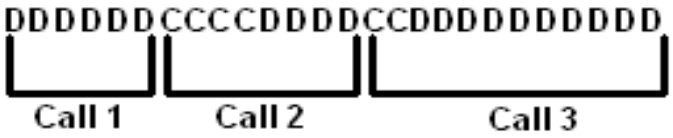
SS$_BADPARAM	Bad *field-code* specified
SS$_NORMAL	Normal, successful completion

User Command Processing

The user command processing routine is a routine that you write to process received TELNET commands. TELNET commands include option-negotiating commands (such as WILL, WONT, DO, and DONT) and control functions (such as AYT, EL, and BREAK).

The TEL_RECEIVE_DATA routine calls your processing routine to process TELNET commands. TELNET commands are processed before any data is returned in the TEL_RECEIVE_DATA call. For example, the data stream shown in Figure 9-1 is returned to you in three calls to TEL_RECEIVE_DATA.

Figure 9-1 TEL_RECEIVE_DATA Routine Returns



D represents data bytes and **C** represents TELNET commands. Your command processing routine is called four times at the beginning of call 2 and twice at the beginning of call 3. This ensures that any change in the way data following a TELNET command is to be interpreted can go into effect before the data is processed. Also if data needs to be included in the data stream in place of the TELNET command, it can be inserted before the beginning of the data buffer returned by the TEL_RECEIVE_DATA routine.

Your routine should have the following call format:

your-routine (*ccb-ptr.ma.r*,*buffer.rbu.ra*,*byte-count.rwu.r*)

The routine is called with the following inputs:

- CCB pointer, by reference
- Address of the command buffer, as an array of bytes
- Size of the command buffer, by reference to an unsigned word

There are no outputs or return values.

One complete command at a time is passed to your command processing routine. Commands are passed with the leading IAC character removed from them. For example, the command IAC *value* is passed as *value*; IAC DO ECHO is passed as DO ECHO, and IAC SB subnegotiation IAC SE is passed as SB. The subnegotiation command data is returned as is. No processing of IAC characters is done.

Chapter 10 SNMP Extendible Agent API Routines

Introduction

This chapter is for application programmers. It describes the Application Programming Interface (API) routines required for an application program to export private Management Information Bases (MIBs) using the TCPware SNMP agent.

To be able to use your private Management Information Base (MIB) with TCPware's SNMP agent, develop a shareable image that exports the following application programming interface routines, in addition to routines you may need to access the MIB variables:

SnmpExtensionInit	Called by the SNMPD agent after startup to initialize the MIB subagent
SnmpExtensionInitEx	Registers multiple subtrees with the subagent (called by the SNMPD agent at startup only if implemented)
SnmpExtensionQuery	Completes the MIB subagent query (called by the SNMPD agent to handle a `get`, `getnext`, or `set` request)
SnmpExtensionTrap	Sends an enterprise-specific trap (called by the SNMPD agent when the subagent alerts the agent that a trap needs to be sent)

Note! The routine names used in this API are taken from the Microsoft SNMP Extension Agent for Windows NT.

The SNMP shareable images need to be configured for the SNMP agent to interact with them.

See the *Configuration* chapter of the *Installation & Configuration Guide* for details on configuring the SNMP agent.

SNMP subagent developers should use the include file SNMP_COMMON.H found in the TCPWARE_INCLUDE directory. This file defines the data structures the API uses.

For details on TCPware's SNMP agent, see Chapter 10, in the *Management Guide*.

Requirements

You require the following before using the SNMP extendible agent API routines:

- Working knowledge of SNMP; specifically the following RFCs:

  - RFC 1155, *Structure and Identification of Management Information for TCP/IP-based Internets*
  - RFC 1157, *A Simple Network Management Protocol (SNMP)*
  - RFC 1213, *Management Information Base for Network Management of TCP/IP-based internets: MIB-II*
- Working knowledge of OpenVMS shareable images

Linking the Extension Agent Image

To link the Extension Agent Image you need to create an option file:

VAX

```
!Note: Exclude SnmpExtensionInitEx if it is not needed.
!See the definition of this routine.
!
UNIVERSAL=SnmpExtensionInit, -
SnmpExtensionQuery, -
SnmpExtensionTrap, -
SnmpExtensionInitEx
SYS$SHARE:VAXCRTL/SHARE
!
!List your object/library files here
```

Alpha and I64

```
!Note: Exclude SnmpExtensionInitEx if it is not needed.
!See the definition of this routine.
!
SYMBOL_VECTOR=( SnmpExtensionInit=PROCEDURE, -
SnmpExtensionQuery=PROCEDURE, -
SnmpExtensionTrap=PROCEDURE, -
SnmpExtensionInitEx=PROCEDURE)
!
!List your object/library files here
```

Your link statement should then look like this:

```
$ LINK /SHARE=image-name option-file/OPT
```

image-name is the name of the shareable image you want to build, and *option-file* is the option file mentioned above.

Installing the Extension Agent Image

You should copy the shareable image you build for your SNMP subagent to the SYS$SHARE directory.

CAUTION! Since the shareable image is loaded into the same process address space as the SNMPD server, an access violation by the subagent shareable image can crash the server application. Ensure the integrity of your shareable image by testing it thoroughly. Shareable image errors can also corrupt the server's memory space or may result in memory or resource leaks.

Sample Code and Data Structures

Sample code is provided in the SYS$COMMON:[TCPWARE.EXAMPLES] directory in the files SNMP_SUBAGENT.C and SNMP_SUBAGENT.H.

SNMP_SUBAGENT.H also defines the following data structures found in the subroutines:

- AsnOBJID
- RFC1157VarBindList

Debugging Code

SNMP subagent developers can use a debug logical, TCPWARE_SNMP_DEBUG to set certain debug masks. Define the logical as follows and use the *mask* values in Debugging Mask Values :

```
$ DEFINE SYSTEM TCPWARE_SNMP_DEBUG mask
```

Table 10-1 Debugging Mask Values

Mask Value	Description
0010	Raw SNMP input
0020	Raw SNMP output
0040	ASN.1 encoded message input
0080	ASN.1 encoded message output
1000	SNMP Subagent Developer debug mask (prints events and statuses)

Subroutine Reference

The following pages include the subroutine descriptions.

SnmpExtensionInit

Initializes the SNMP subagent and registers the subagent in the SNMPD agent. The subagent calls this routine at startup.

Format

status = SnmpExtensionInit (*trap-alert-routine*, *time-zero-reference*, *trap-event*, *supported-view*)

Return Values

TRUE	Subagent initialized successfully
FALSE	Subagent initialization failed

Arguments

trap-alert-routine

OpenVMS usage:	address
type:	integer
access:	read only
mechanism:	by value

Address of the routine the subagent should call when it is ready to send a trap.

time-zero-reference

OpenVMS usage:	unsigned long
type:	longword (unsigned)
access:	read only
mechanism:	by value

Time reference the SNMP agent provides, in hundredths of a second. Use C routines `time()` and `difftime()` to calculate MIB uptime (in hundredths of a second).

trap-event

OpenVMS usage:	unused
type:	longword (unsigned)
access:	write only
mechanism:	by reference

This parameter is reserved for future use.

supported-view

OpenVMS usage:	object identifier
type:	AsnOBJID (see the SNMP_SUBAGENT.H file)
access:	write only
mechanism:	by reference

Prefix of the MIB tree the subagent supports.

SnmpExtensionInitEx

Registers multiple MIB subtrees with agent.

This routine is called multiple times, once for each MIB subtree that needs to be registered. If the routine passes back the first or next MIB subtree, return with TRUE. If all the MIB subtrees were passed back, return with FALSE.

Note! Only implement this routine if you have multiple MIB subtrees in your extendible agent. The TCPware SNMP agent executes this routine if it exists and overwrites MIB information set by `SnmpExtensionInit`.

Format

status = SnmpExtentionInitEx (*supported-view*)

Return Values

TRUE	Returning first or next MIB subtree
FALSE	All MIB subtrees were passed back

Arguments

supported-view

OpenVMS usage:	object identifier
type:	AsnOBJID (see the SNMP_SUBAGENT.H file)
access:	write only
mechanism:	by reference

Prefix of the MIB tree the subagent supports.

Example

```
int SnmpExtensionInitEx (AsnOBJID            *supportedView)
  {
    int view1[] = {1, 3, 6, 1, 4, 1, 12, 2, 1 };
    int view2[] = {1, 3, 6, 1, 4, 1, 12, 2, 2 };
    int view3[] = {1, 3, 6, 1, 4, 1, 12, 2, 5 };
    static int whichView = 0;
    switch ( whichView++) {
    case 0:
      supportedView->idLength = 9;
      memcpy (supportedView->ids, view1, 9* sizeof (int));
```

```
    break;
  case 1:
    supportedView->idLength = 9;
    memcpy (supportedView->ids, view2, 9* sizeof (int));
    break;
  case 2:
    supportedView->idLength = 9;
    memcpy (supportedView->ids, view3, 9* sizeof (int));
    break;
  default:
    return (0);
  }
  return (1);
}
```

SnmpExtensionQuery

Queries the SNMP subagent to get or set a variable in the MIB tree served by the subagent. This routine is called by the SNMPD agent to handle a `get`, `getnext`, or `set` request.

Format

status = SnmpExtensionQuery (*request-type*, *var-bind-list*, *error-status*, *error-index*)

Return Values

TRUE	Operation successfully completed
FALSE	Operation could not be carried out by the subagent; use *error-status* and *error-index* to provide more information

Arguments

request-type

OpenVMS usage:	byte
type:	unsigned char
access:	read only
mechanism:	by value

Identifies the type of request GET, SET, or GET NEXT. These values are defined in TCPWARE_ROOT:[TCPWARE.INCLUDE]SNMP_COMMON.H

var-bind-list

OpenVMS usage:	user defined
type:	RFC1157VarBindList (see the SNMP_SUBAGENT.H file)
access:	read-write
mechanism:	by value

The list of name-value pairs used in the request. For a GET request the value is filled by the subagent and for a SET request, the value is be used to change the current variable value in the subagent.

error-status

OpenVMS usage:	integer
type:	integer
access:	write only
mechanism:	by reference

Status of a failed operation. See SNMP_COMMON.H for error value names.

error-index

OpenVMS usage:	integer
type:	integer
access:	write only
mechanism:	by reference

The index of the variable in the variable binding list for which the operation failed.

SnmpExtensionTrap

Sends a trap from the subagent. If the subagent wants to send a trap, it must first call the `trap-alert-routine` (see the `SnmpExtensionInit` routine). The trap-alert-routine should be called with two parameters (oid ids, oid idlength). For example:

If the Process Software's DNS process wants to send trap information to all the communities that are interested then the DNS server must be running and the objectids passed are 1, 3, 6, 1, 4, 1, 105, 1, 2, 1, 1, 1, 3, 1, and the length of 14.

- 1,3,6,1,4,1 is the default prefix
- 105 is the enterprise id for Process Software
- 1,2,1,1,1 are the Mib object ids for the DNS process
- 3,1 are the objectids for DNSUpTrap

The SNMP agent trap-alert-routine creates a table of all received trap mibs. For each of these entries, the agent then calls the subagent's `SnmpExtensionTrap` routine when it is ready to send the trap. Note that the SNMP agent calls the subagent from inside the `trap-alert-routine`.

Format

status = SnmpExtensionTrap (*enterprise*, *generic-trap*, *specific-trap*, *time-stamp*, *var-bind-list*)

Return Values

TRUE	More traps to be generated
FALSE	No more traps to be generated

Arguments

enterprise

OpenVMS usage:	array of object identifiers
type:	AsnOBJID (see the SNMP_SUBAGENT.H file)
access:	write only
mechanism:	by reference

The prefix of the MIB for the enterprise sending the trap.

generic-trap

OpenVMS usage:	integer
type:	integer
access:	write only
mechanism:	by reference

The generic enterprise trap id(6).

specific-trap

OpenVMS usage:	integer
type:	integer
access:	write only
mechanism:	by reference

The enterprise-specific trap number.

Note! Since an enterprise can have many traps, the combination of enterprise id, generic trap, and specific trap should give a unique identification for a trap.

time-stamp

OpenVMS usage:	integer
type:	integer (timeticks)
access:	write only
mechanism:	by reference

The time at which the trap was generated.

var-bind-list

OpenVMS usage:	user defined
type:	RFC1157VarBindList (see the SNMP_SUBAGENT.H file)
access:	read-write
mechanism:	by value

The list of name-value pairs. This list contains name and value of the MIB variable for which the trap is generated.

Chapter 11 Token Authentication API Functions

Introduction

This chapter is for application programmers. It describes the Application Programming Interface (API) to enable Token Authentication between the TCPware ACE/Client and the Security Dynamics ACE/Server.

See Chapter 25, *Token Authentication Management*, in the *Management Guide* for details on the TCPware ACE/Client, PASSCODES, and tokens.

The Token Authentication Application Programming Interface (API) on the TCPware ACE/Client includes the following functions:

creadcfg	Loads the Security Dynamics ACE/Server configuration file into memory
sd_auth	Combines the functions of `sd_check`, `sd_next`, and `sd_pin`
sd_check	Provides the PASSCODE collected from the user and the username for authentication
sd_close	Destroys the socket initialized using `sd_init`
sd_init	Initializes the socket and makes a verification call to the Security Dynamics ACE/Server
sd_next	Handles the processing of a second tokencode when one is required to complete an authentication
sd_pin	Handles the case of a new PIN required at login

Supported Languages

The TCPware ACE/Client API routines support the following programming languages:

BASIC	C	MACRO	PL1
BLISS	FORTRAN	PASCAL	

How to Use Functions

Use the **creadcfg** routine first. This routine loads the configuration file into memory. The **sd_init** routine initializes the socket and makes a call to the server to verify communication. Use this call after a successful call to **creadcfg** and before calling any other API function.

If your program performs the user input/output, use the three functions **sd_check**, **sd_pin**, and **sd_next**. Do not use **sd_check** without the other two functions. Use the **sd_auth** routine instead of these previous three routines only if **stdin** or **stdout** are available for user interaction. If using **sd_auth**, do not also use **sd_check**, **sd_pin**, and **sd_next**.

Header Files

You must include the following header files in all programs:

- **sdi_athd.h**
- **sdconf.h**

Programs that use **sd_check**, **sd_pin**, and **sd_next** must also include these header files:

- **sdi_size.h**
- **sdi_type.h**
- **sdi_defs.h**
- **sdacmvls.h**

You must include a variable named **configure** in your code. Make the following global variable declaration:

```
union config_record configure;
```

The definition for **config_record** is included in the header files. The header files are located in the TCPWARE_COMMON:[TCPWARE.INCLUDE] directory.

Activating Program Shareable Image

You can use one of two methods to activate the SYS$SHARE:TCPWARE_ACECLIENT_SHR.EXE shareable image for your application:

- Link the SYS$SHARE:TCPWARE_ACECLIENT_SHR.EXE file so that it is activated at image creation time. Add the following line in an .OPT file to link in the SYS$SHARE:TCPWARE_ACECLIENT_SHR.EXE shareable image:

  SYS$SHARE:TCPWARE_ACECLIENT_SHR.EXE/SHAREABLE

See the *OpenVMS Linker Utility Manual* for details.

- Use the LIB$FIND_IMAGE_SYMBOL to activate the shareable image at runtime. Use the OpenVMS LIB$FIND_IMAGE_SYMBOL runtime library to activate the shareable image at runtime.

  See the *OpenVMS RTL Library (LIB$) Manual* for details.

Function Reference

The following pages include the function descriptions.

creadcfg

Reads the ACE/Server configuration file stored in the TCPWARE_ACECLIENT_DATA_DIRECTORY data directory and loads it into memory. The configuration file was created on the ACE/Server during installation and should be copied to the ACE/Client data directory when installing the ACE/Client. The name of the configuration file is SDCONF.REC.

Call this function before any other API function.

Format

int creadcfg(void)

Return Values

0	Successful
-1	Failure to load the configuration file (it may be missing or corrupted; do not call any further function

Arguments

None

sd_init

Initializes client/server communication by initializing the socket and making a call to the server to verify communication.

Call this function after a successful call to **`creadcfg`** and before calling any other API function. Use **`sd_close`** after the authentication process to close the socket initialized by **`sd_init`**.

Format

int sd_init(struct SD_CLIENT *_sd_)

Return Values

0	Successful
1	Client unable to communicate with the ACE/server (general communication or configuration problem or the server is not running)

Argument

sd

Pointer to an SD_CLIENT structure. The structure must be set to **`0`** by the caller before calling **`sd_init`**.

sd_auth

Performs all user input/output. This function performs SecurID© authentication dialogues, including all authentication prompts and responses (such as **`Enter PASSCODE`** or **`PASSCODE accepted`**).

Call this function only after **`sd_init`** was called successfully.

Format

int sd_auth(struct SD_CLIENT *_sd_)

Return Values

ACM_OK	User successfully authenticated
ACM_ACCESS_DENIED	User failed authentication

Argument

sd

Pointer to the SD_CLIENT structure. Specify the login ID by setting the username field in the data structure. If not specified, the API tries to obtain it from the environment.

sd_check

Performs user authentication by checking the validity of the PASSCODE entered by a user. The integrating application must do all input/output because **`sd_check`** does not display the authentication prompts and messages (unlike **`sd_auth`**).

Call this function only after **`sd_init`** was called successfully. Use **`sd_pin`** to complete the transaction.

Format

int sd_check(char **passcode*, char **username*, struct SD_CLIENT **sd*)

Return Values

ACM_OK	User successfully authenticated
ACM_ACCESS_DENIED	User failed authentication
ACM_NEXT_CODE_REQUIRED	Next tokencode required; the following field is set in the SD_CLIENT structure: **`timeout`** (the number of seconds the server waits for a user to respond to the **`next-code`** prompt)
ACM_NEW_PIN_REQUIRED	New PIN required; the following fields are set in the SD_CLIENT structure: **system_pin**—random PIN generated by the system **min_pin_length**—minimum PIN length **max_pin_length**—maximum PIN length **user_selectable**—can have one of three values: • CANNOT_CHOOSE_PIN • MUST_CHOOSE_PIN • USER_SELECTABLE **alphanumerics**—PIN can contain letters

Arguments

passcode

Pointer to the NULL-terminated PASSCODE string. The PASSCODE must contain from 4 to 16 characters.

username

Pointer to the NULL-terminated username string. The username must contain fewer than 32 characters.

sd

Pointer to the SD_CLIENT structure.

sd_next

Function used in response to an **ACM_NEXT_CODE_REQUIRED** return from **sd_check**. Performs the Next Code operation that takes a second successive tokencode from a user and checks it validity. The integrating application must do all input/output because **sd_next** does not display the **Next Code** prompt.

Call this function only in response to an **ACM_NEXT_CODE_REQUIRED** return from **sd_check**.

Format

int sd_next(char **nextcode*, struct SD_CLIENT **sd*)

Return Values

ACM_OK	User successfully authenticated
ACM_ACCESS_DENIED	User failed authentication

Arguments

nextcode

Pointer to the NULL-terminated PASSCODE string. The PASSCODE must contain from 4 to 8 characters.

sd

Pointer to the SD_CLIENT structure.

sd_pin

Function used in response to an **ACM_NEW_PIN_REQUIRED** return from **sd_check**. Performs the New PIN operation in which a new PIN is stored in a token record. The integrating application must do all input/output because **sd_pin** does not display the **New PIN** prompts and messages.

Call this function only in response to an **ACM_NEW_PIN_REQUIRED** return from **sd_check**.

Note! Do not treat users as authenticated as soon as they complete the New PIN operation. Require them to authenticate using the new PIN

Format

int sd_pin(char **pin*, char *canceled*, struct SD_CLIENT **sd*)

Return Values

ACM_NEW_PIN_ACCEPTED	New PIN was accepted by the ACE/Server; the user should now be required to authenticate with it
ACM_NEW_PIN_REJECTED	New PIN was rejected by the ACE/Server; the PIN may not have matched the parameters set in the return from **sd_check**

Arguments

pin

Pointer to the NULL-terminated PIN string. The PIN must contain from 4 to 8 characters.

cancelled

Should equal **0** if a PIN is selected. If a token is in New PIN mode but you do not want to select the PIN at that time, the value should be set to **1**.

sd

Pointer to the SD_CLIENT structure.

sd_close

Closes the socket opened by **`sd_init`**.

Call this function after attempting to authenticate the user regardless whether or not the authentication was successful.

Format

void sd_close(void)

Return Values

None

Arguments

None

Chapter 12 ONC RPC Fundamentals

Introduction

TCPware provides two sets of ONC RPC Services for network programming:

- ONC RPC Services to be used with the DEC C Socket Library for DEC C
- ONC RPC Services to be used with TCPware's Socket Library for VAX C or DEC C

This chapter is for RPC programmers. It provides basic information you need to know before using ONC RPC Services to write distributed applications, including:

- What ONC RPC Services are
- What components are in ONC RPC Services
- How ONC RPC clients and servers communicate
- Important ONC RPC concepts and terms

What Are ONC RPC Services?

ONC RPC Services are a set of software development tools that allow you to build distributed applications on OpenVMS systems. These services are part of TCP-OpenVMS.

TCPware provides two types of ONC RPC Services:

– **ONC** RPC used with VAX C and the TCPware Socket Library
– RPC **XDR** used with DEC C and the DEC C Socket Library

The bold letters in the above list show the conventions this book uses to distinguish between the two services.

TCPware Implementation

ONC RPC Services are based on the Open Network Computing Remote Procedure Call (RPC) protocols developed by Sun Microsystems, Inc. These protocols are defined in the following Requests for Comments (RFCs):

- *RPC: Remote Procedure Call Protocol Specification, Version 2* (RFC 1057)
- *XDR: External Data Representation Standard* (RFC 1014)

Distributed Applications

A distributed application executes different parts of its programs on different hosts in a network. Computers on the network share the processing workload, with each computer performing the tasks for which it is best equipped.

For example, a distributed database application might consist of a central database running on a VAX server and numerous client workstations. The workstations send requests to the server. The server carries out the requests and sends the results back to the workstations. The workstations use the results in other modules of the application.

RPCs allow programs to invoke procedures on remote hosts as if the procedures were local. ONC RPC Services hides the networking details from the application.

ONC RPC Services facilitates distributed processing because it relieves the application programmer of performing low-level network tasks such as establishing connections, addressing sockets, and converting data from one machine's format to another.

Components of ONC RPC Services

ONC RPC Services comprises the following components:

Run-time libraries (RTLs)	RPCGEN compiler	Port Mapper	RPCINFO command

Run-Time Libraries (RTLs)

XDR: ONC RPC Services provides a single shareable RTL. The library contains:

- ONC RPC client and server routines
- XDR routines

ONC: ONC RPC Services provides two shareable RTLs, one for D_float numbers and one for G_float numbers. Both libraries contain:

- ONC RPC client and server routines
- XDR routines
- Additional management routines that are unique to ONC RPC Services

The *ONC RPC RTL Management Routines*, Chapter 16, and the chapters that follow it describe the RTLs in detail.

RPCGEN Compiler

RPCGEN is a compiler that creates the network interface portion of a distributed application. It effectively hides from the programmer the details of writing and debugging low-level network interface code. The *RPCGEN Compiler*, Chapter 14 , describes how to use RPCGEN.

Port Mapper

The Port Mapper helps ONC RPC client programs connect to ports that are being used by ONC RPC servers. A Port Mapper runs on each host that implements ONC RPC Services. These steps summarize how the Port Mapper works:

1. ONC RPC servers register with the Port Mapper by telling it which ports they are using.
2. When an ONC RPC client needs to reach a particular server, it supplies the Port Mapper with the numbers of the remote program and program version it wants to reach. The client also specifies a transport protocol (UDP or TCP). *(Identifying Remote Programs and Procedures* provides details on these numbers.)
3. The Port Mapper provides the correct port number for the requested service. This process is called binding.

Once binding has taken place, the client does not have to call the Port Mapper for subsequent calls to the same server. A service can register for different ports on different hosts. For example, a server can register for port 800 on Host A and port 1000 on Host B. The Port Mapper is itself an ONC RPC server and uses the ONC RPC RTL. It uses the UDPA and TCPA protocols for transports. The Port Mapper plays an important role in disseminating messages for broadcast RPC. The Port Mapper is part of the Network Control Process (NETCP). See the *Broadcast RPC* section for details.

RPCINFO Command

Use the RPCINFO command to:

- Request a listing of all programs that are registered with the Port Mapper
- Call the null routine of any program

You enter this command at the DCL prompt. (See *RPCINFO Utility* in Chapter 13, *Building Distributed Applications*, for details.)

Client-Server Relationship

In ONC RPC, the terms client and server do not describe particular hosts or software entities. Rather, they describe the roles of particular programs in a given transaction. Every ONC RPC transaction has a client and a server. The client is the program that calls a remote procedure; the server is the program that executes the procedure on behalf of the caller.

A program can be a client or a server at different times. The program's role merely depends on whether it is making the call or servicing the call.

External Data Representation (XDR)

External Data Representation (XDR) is a standard that solves the problem of converting data from one machine's format to another.

ONC RPC Services uses the XDR data description language to describe and encode data. Although similar to C language, XDR is not a programming language. It merely describes the format of data, using implicit typing. *XDR: External Data Representation Standard (*RFC 1014) defines the XDR language.

ONC RPC Processing Flow

Remote and local procedure calls share some similarities. In both cases, a calling process makes arguments available to a procedure. The procedure uses the arguments to compute a result, then returns the result to the caller. The caller uses the results of the procedure and resumes execution.

Figure 2-1 shows the underlying processing that makes a remote procedure call different from a local call.

The following steps describe the processing flow during a remote procedure call:

2 The client program passes arguments to the client stub procedure. (See Chapter 14, *RPCGEN Compiler*, for details on how to create stubs.)

2 The client stub marshals the data by:

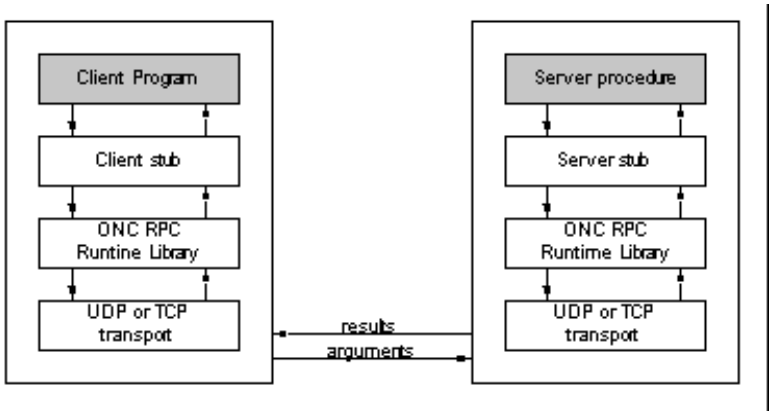
- Calling the XDR routines to convert the arguments from the local representation to XDR
- Placing the results in a packet

Using ONC RPC RTL calls, the client stub sends the packet to the UDP or TCP layer for transmission to the server.

3 The packet travels on the network to the server, up through the layers to the server stub.

4 The server stub un-marshals the packet by converting the arguments from XDR to the local representation. Then it passes the arguments to the server procedure.

Figure 12-1 ONC RPC Processing Flow



Local Calls versus Remote Calls

This section describes some of the ways in which local and remote procedure calls handle system crashes, errors, and call semantics.

Handling System Crashes

Local procedure calls involve programs that reside on the same host. Therefore, the called procedure cannot crash independently of the calling program.

Remote procedure calls involve programs that reside on different hosts. Therefore, the client program does not necessarily know when the remote host has crashed.

Handling Errors

If a local procedure call encounters a condition that prevents the call from executing, the local operating system usually tells the calling procedure what happened.

If a remote procedure call cannot be executed for some reason (e.g., errors occur on the network or remote host), the client might not be informed of what happened. You may want to build a signaling or condition-handling mechanism into the application to inform the client of such errors.

ONC RPC returns certain types of errors to the client, such as those that occur when it cannot decode arguments. The RPC server must be able to return processing-related errors, such as those that occur when arguments are invalid, to the client. However, the RPC server may not return errors during batch processing or broadcast RPC.

Call Semantics

Call semantics determine how many times a procedure executes.

Local procedures are guaranteed to execute once and only once.

Remote procedures have different guarantees, depending on which transport protocol is used.

XDR: The TCP transport guarantees execution once and only once as long as the server does not crash. The UDP transport guarantees execution at least once. It relies on the XID cache to prevent a remote procedure from executing multiple times.

ONC: The TCP and TCPA transports guarantee execution once and only once as long as the server does not crash. The UDP and UDPA transports guarantee execution at least once. They rely on the XID cache to prevent a remote procedure from executing multiple times.

See *XID Cache* for details on the XID cache.

Programming Interface

The ONC RPC RTL is the programming interface to RPC. You may think of this interface as containing multiple levels.

The ONC RPC RTL reference chapters describe each routine.

The sample programs listed in the *ONC RPC Sample Programs*, Chapter 20, show how ONC RPC routines are used at each level.

High-Level Routines

The higher-level RPC routines provide the simplest RPC programming interface. These routines call lower-level RPC routines using default arguments, effectively hiding the networking details from the application programmer.

When you use high-level routines, you sacrifice control over such tasks as client authentication, port registration, and socket manipulation, but you gain the benefits of using a simpler programming interface. Programmers using high-level routines can usually develop applications faster than they can using low-level RPC routines.

You can use the RPCGEN compiler only when you use the highest-level RPC programming interface.

Mid-Level Routines

The mid-level routines provide the most commonly used RPC interface. They give the programmer some control over networking tasks, but not as much control as the low-level routines permit.

For example, you can control memory allocation, authentication, ports, and sockets using mid-level routines.

The mid-level routines require you to know procedure, program, and version numbers, as well as input and output types. Output data is available for future use. You can use the `registerrpc` and `callrpc` routines.

Low-Level Routines

The low-level routines provide the most complicated RPC interface, but they also give you the most control over networking tasks such as client authentication, port registration, and socket manipulation. These routines are used for the most sophisticated distributed applications.

ONC: These routines also allow you to use the TCPA and UDPA transports, as described below.

Transport Protocols

XDR and ONC RPC Services use the transport protocols listed in Table 12-1. The RPC client and server must use the same transport protocol for a given transaction.

Table 12-1 XDR and ONC RPC Transport Protocols

Protocols	Characteristics
UDP (XDR and ONC) **UDPA** (ONC only)	Unreliable datagram service Connectionless Used for broadcast RPC Maximum broadcast message size in either direction on an Ethernet line: 1500 Execution is guaranteed at least once (see *XID Cache*) Calls cannot be processed in batch
TCP (XDR and ONC) **TCPA** (ONC only)	Reliable Connection-oriented Can send an unlimited number of bytes per RPC call Execution is guaranteed once and only once Calls can be processed in batch No broadcasting

Note! **XDR:** You must use the DEC C Socket Library with ONC RPC Services.

Note! **ONC:** You must use the TCPware Socket Library with ONC RPC Services.

XID Cache

The XID cache stores responses the server has sent. When the XID cache is enabled, the server does not have to recreate every response to every request. Instead, the server can use the responses in the cache. Thus, the XID cache saves computing resources and improves the performance of the server.

XDR: Only the UDP transports can use the XID cache. The reliability of the TCP transport generally makes the XID cache unnecessary. UDP is inherently unreliable.

ONC: Only the UDP, UDPA, and TCPA transports can use the XID cache. The reliability of the TCP and TCPA transports generally makes the XID cache unnecessary. UDP is inherently unreliable.

Table 12-2 shows how the XID caches differ for the UDP and UDPA/TCPA transports.

Table 12-2 XID Cache Differences

UDP Transport	UDPA/TCPA Transports
Places every response in the XID cache	Allows the server to specify which responses are to be cached, using the `svcudpa_enablecache` and `svctcpa_enablecache` routines
XID cache cannot be disabled	Requires you to disable the XID cache after use

Cache Entries

Each entry in the XID cache contains:

- The encoded response that was sent over the network
- The internet address of the client that sent the request

- The transaction ID that the client assigned to the request

Cache Size

You determine the size of the XID cache. Consider these factors:

- How many clients are using the server.
- Approximately how long the cache should save the responses.
- How much memory you can allocate. Each entry requires about 8Kbytes.

The more active the server is, the less time the responses remain in the cache.

Execution Guarantees

As explained earlier in *Local Calls Versus Remote Calls*, remote procedures have different execution guarantees, depending on which transport protocol is used. The XID cache affects the execution guarantee.

XDR: The TCP transport guarantees execution once and only once as long as the server does not crash. The UDP transport guarantees execution at least once. If the XID cache is enabled, a UDP procedure is unlikely to execute more than once.

ONC: The TCP and TCPA transports guarantee execution once and only once as long as the server does not crash. The UDP and UDPA transports guarantee execution at least once. If the XID cache is enabled, a UDP or UDPA procedure is unlikely to execute more than once.

Enabling XID Cache

XDR: Use the `svcudp_enablecache` routine to enable the XID cache. This routine is described in the ONC RPC RTL reference chapters.

ONC: Use the `svcudp_enablecache`, `svcudpa_enablecache` and `svctcpa_enablecache` routines to enable the XID cache. Use the `svcudpa_freecache` and `svctcpa_freecache` routines to disable the cache for a UDPA/TCPA server. These routines are described in the ONC RPC RTL reference chapters.

Not enabling the XID cache saves memory.

Active Cache

The active cache maintains a list of active requests that the server is working on. All UDPA servers use the active cache. A UDP server does not need an active cache because it can work on only one request at a time.

XDR: TCP servers do not need an active cache because they do not receive duplicate requests.

ONC: TCP and TCPA servers do not need an active cache because they do not receive duplicate requests. When a UDPA server receives a request, it searches the active cache for a match. If no match is found, the server places the request in the active cache and processes it. If a match is found, the server ignores the new request because it is already processing the request. When the server sends a response, it removes the request from the active cache and may add it to the XID cache.

Broadcast RPC

Broadcast RPC allows the client to send a broadcast call to all Port Mappers on the network and wait for multiple replies from ONC RPC servers.

For example, a host might use a broadcast RPC message to inform all hosts on a network of a system shutdown.

Table 12-3 shows the differences between normal RPC and broadcast RPC.

Table 12-3 Normal RPC vs Broadcast RPC

Normal RPC	Broadcast RPC
Client expects one answer	Client expects many answers
Can use TCP or UDP	Requires UDP
Server always responds to errors	Server does not respond to errors; Client does not know when errors occur
Port Mapper is desirable, but not required if you use fixed port numbers	Requires Port Mapper services

Broadcast RPC sends messages to only one port–the Port Mapper port–on every host in the network. On each host, the Port Mappers pass the messages to the target ONC RPC server. The servers compute the results and send them back to the client. See *Broadcast RPC Sample Programs* in Chapter 20, *ONC RPC Sample Programs*, for a list of sample programs using broadcast RPC.

ONC RPC Batch Facilities

Normally, an ONC RPC client sends a remote procedure call, stops processing, and waits for the server to reply. Some remote procedure calls, however, do not require a reply or acknowledgment. In these cases, the client wastes time waiting for the server to respond.

ONC RPC batch facilities solve this problem by allowing the client to send many remote procedure calls without waiting for replies.

ONC RPC Services performs batching by placing the client's messages to a particular server sequentially in a pipeline. The server knows it does not have to respond to these messages. The client sends a normal RPC call at the end of the batch sequence to "flush" the pipeline and let the server know there are no more batch calls.

Batching decreases the amount of communication overhead used by the client and server. The client can place many call messages in a buffer and send them to the server in one system call. The client can generate new calls while the server processes previous calls.

Batch Requirements

All ONC RPC batch calls must meet these requirements (see *Batch RPC Sample Programs* in Chapter 20, *ONC RPC Sample Programs*, for an example of batching):

- Specify that the result of the call is to be decoded by the XDR routine at address zero
- Have a timeout of zero
- Use the TCP transport

Identifying Remote Programs and Procedures

The ONC RPC client must uniquely identify the remote procedure it wants to reach. Therefore, all remote procedure calls must contain these three fields:

- A remote program number
- The version number of the remote program
- A remote procedure number

Remote Program Numbers

A remote program is a program that implements at least one remote procedure. Remote programs are identified by numbers that you assign during application development. Use Table 12-4 to determine which program numbers are available. The numbers are in groups of hexadecimal 20000000.

Table 12-4 Remote Program Numbers

Range	Purpose
0 to 1FFFFFFF	Defined and administered by Sun Microsystems. Should be identical for all sites. Use only for applications of general interest to the Internet community.
20000000 to 3FFFFFFF	Defined by the client application program. Site-specific. Use primarily for new programs.
40000000 to 5FFFFFFF	Use for applications that generate program numbers dynamically.
60000000 to FFFFFFFF	Reserved for the future. Do not use.

Remote Version Numbers

Multiple versions of the same program may exist on a host or network. Version numbers distinguish one version of a program from another. Each time you alter a program, remember to increment its version number.

Remote Procedure Numbers

A remote program may contain many remote procedures. Remote procedures are identified by numbers that you assign during application development. Follow these guidelines when assigning procedure numbers:

- Use 1 for the first procedure in a program. (Procedure 0 should do nothing and require no authentication to the server.)
- For each additional procedure in a program, increment the procedure number by one.

Additional Terms

Before writing RPC applications, you should be familiar with the terms in Table 12-5.

Table 12-5 Additional Terms

XDR and ONC	
Term	**Definition**
Channel	An OpenVMS term referring to a logical path that connects a process to a physical device, allowing the process to communicate with that device. A process requests OpenVMS to assign a channel to a device. Refer to HP's documentation for more information on channels.

Client handle	Information that uniquely identifies the server to which the client is sending the request. Consists of the server's host name, program number, program version number, and transport protocol. See the following routines in the *ONC RPC RTL Client Routines*, Chapter 16:

XDR		**ONC**	
Term	**Definition**	**Term**	**Definition**
Client handle	authnone_create authunix_create authunix_create_default clnt_create clnttcp_create clntudp_create clnt_perror	Client handle	authnone_create authunix_create authunix_create_default clnt_create clnttcp_create clntudp_create clnt_call clnt_control clnt_destroy clnt_freeres clnt_geterr clnt_perror

XDR and ONC	
Term	**Definition**
Port	An abstract point through which a datagram passes from the host layer to the application layer protocols.

XDR	
Term	**Definition**
Server handle	Information that uniquely identifies the server. Content varies according to the transport being used. See the following routines in Chapter 18, *ONC RPC RTL Server Routines*: svcudp_create svctcp_create svc_destroy svc_freeargs svc_getargs svc_getcaller svc_register svc_sendreply svcerr_ routines

ONC	
Term	**Definition**
Server handle	Information that uniquely identifies the server. Content varies according to the transport being used. See the following routines in Chapter 18, *ONC RPC RTL Server Routines*: svcudp_create svcudpa_create svctcp_create svctcpa_create svc_destroy svc_freeargs svc_getargs svc_register svc_sendreply svcerr_ routines

XDR and ONC	
Term	**Definition**
Socket	An abstract point through which a process gains access to the Internet. A process must open a socket and bind it to a specific destination. Note: The TCPware Socket Library must be used with ONC RPC Services.

Chapter 13
Building Distributed Applications with RPC

Introduction

This chapter is for RPC programmers. It explains:

- What components a distributed application contains
- How to use ONC RPC to develop a distributed application, step by step
- How to use the RPCINFO utility

Distributed Application Components

Table 13-1 lists the components of a distributed application.

Table 13-1 Application Components

Component	Description
Main program (client)	An ordinary main program that calls a remote procedure as if local
Network interface	Client and server stubs, header files, XDR routines for input arguments and results
Server procedure	Carries out the client's request (at least one is required)

These components may be written in any high-level language. The ONC RPC Run-Time Library (RTL) routines are written in C language.

What You Need to Do

The following steps summarize what you need to do to build a distributed application:

1 Design the application.

2 Write an RPC interface definition. Compile it using RPCGEN, then edit the output files as necessary. (This step is optional. An RPC interface definition is not required. If you do not write one, proceed to step 3.)

3 Write any necessary code that RPCGEN did not generate.

4 Compile the RPCGEN output files, server procedures, and main program using the appropriate language compiler(s).

5 Link the object code, making sure you link in the ONC RPC RTL.

6 Start the Port Mapper on the server host.

7 Execute the client and server programs.

Step 1: Design the Application

You must write a main (client) program and at least one server procedure. The network interface, however, may be hand-written or created by RPCGEN. The network interface files contain client and server stubs, header files, and XDR routines. You may edit any files that RPCGEN creates.

When deciding whether to write the network interface yourself, consider these factors:

Is execution time critical?	Your hand-written code may execute faster than code that RPCGEN creates.
Which RPC interface layer do you want to use?	RPCGEN permits you to use only the highest layer interface. If you want to use the lower layers, you must write original code. The *ONC RPC Fundamentals*, Chapter 12, describes the characteristics of each RPC interface layer.
Which transport protocol do you want to use?	

ONC: If you use an asynchronous transport (UDPA or TCPA), you must either write original code for the server stubs or edit the RPCGEN output.

You may write your own XDR programs, but it is usually best to let RPCGEN handle these.

Step 2: Write and Compile the Interface Definition

An interface definition is a program the RPCGEN compiler accepts as input. The *RPCGEN Compiler*, Chapter 14, explains exactly what interface definitions must contain.

Interface definitions are optional. If you write the all of the network interface code yourself, you do not need an interface definition.

You must write an interface definition if you want RPCGEN to generate network interface code.

After compiling the interface definition, edit the output file(s).

If you are not writing an interface definition, skip this step and proceed to step3.

Step 3: Write the Necessary Code

Write any necessary code that RPCGEN did not create for you. Table 13-2 lists the texts you may use as references.

Table 13-2 Coding References

Reference	Purpose
RFC 1057	Defines the RPC language. Use for writing interface definitions.
RFC 1014	Defines the XDR language. Use for writing XDR filter routines.
The *ONC RPC RTL Client Routines* chapter and those that follow	Defines each routine in the ONC RPC RTL. Use for writing stub procedures and XDR filter routines.

Building a Structure

If an input argument or result is not an integer, you need to create a structure for it in the RPCGEN input file. RPCGEN converts the structure to C format and creates XDR code to encode and decode it.

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI.X file provides a sample interface definition containing a structure.

Step 4: Compile All Files

Compile the RPCGEN output files, server procedures, and main program separately using the appropriate language compiler(s):

XDR: DEC C (VAX, Alpha and I64):

```
$ CC /STANDARD=RELAXED /WARNING=DISABLE=(IMPLICITFUNC) filename.C
```

ONC: VAX C:

```
$ CC filename.C
```

ONC: DEC C (VAX):

```
$ CC /STANDARD=VAXC filename.C
```

ONC: DEC C (Alpha and I64):

```
$ CC /STANDARD=VAXC /NOMEMBER_ALIGN /ASSUME=NOALIGN filename.C
```

FORTRAN (VAX):

```
$ FORTRAN filename.FOR
```

FORTRAN (Alpha and I64):

```
$ FORTRAN /NOALIGN /WARNING=(DECLARATIONS,ALIGN) filename.FOR
```

Step 5: Link the Object Code

Link the object code files. Make sure you link in the ONC RPC RTL. Use the following command:

XDR: DEC C (VAX, Alpha and I64):

```
$ LINK filenames, SYS$INPUT /OPTIONS
UCX$RPCXDR_SHR /SHARE
SYS$SHARE:DECC$SHR /SHARE
Ctrl/Z
```

ONC: VAX:

```
$ LINK filenames, SYS$INPUT /OPTIONS
SYS$SHARE:VAXCRTL/SHARE
SYS$SHARE:TCPWARE_RPCLIB_SHR/SHARE
Ctrl/Z
```

ONC: Alpha and I64:

```
$ LINK filenames, SYS$INPUT /OPTIONS
UCX$RPCXDR_SHR /SHARE
SYS$SHARE:DECC$SHR /SHARE
Ctrl/Z
```

After entering the command, press **Ctrl/Z**.

To avoid repetitive data entry, you may create an OpenVMS command procedure to execute these commands.

Step 6: Start the Port Mapper

The Port Mapper must be running on the server host. If it is not running, use the NETCU ADD SERVICE command to start it. See Chapter 2, *NETCU Commands* of the *NETCU Command Reference* manual for details on this command.

Step 7: Execute the Client and Server Programs

Perform these steps:

1. Run the server program interactively to debug it, or using the /DETACHED qualifier. Refer to HP's documentation for details.
2. Run the client main program.

Using Asynchronous Transports

ONC: ONC RPC provides two asynchronous transport protocols: TCPA and UDPA. These protocols allow you to write multi-threaded servers that can process multiple requests in parallel.

Only an asynchronous service can benefit from using an asynchronous transport.

For example, you would use an asynchronous transport for a service that performs a lot of asynchronous file I/O by issuing QIOs. You would not use an asynchronous transport for a service that performs a lot of synchronous file I/O.

Client processes cannot use TCPA and UDPA.

The asynchronous transport protocols are specific to TCPware.

The *ONC RPC Fundamentals*, Chapter 12, explains the differences between each transport protocol.

Writing an Asynchronous Server

ONC: This section explains how to write an asynchronous server.

Before You Begin

1. Decide how many threads the server will support. The number of threads determines the number of requests the server can handle in parallel. The server can parallel process as many requests as it has threads.
2. Choose the TCPA and/or UDPA transport protocol. (A server can support both.)
3. Determine:
   - The size of the XID cache. Generally, this is at least the number of UDPA or TCPA threads.
   - Which procedures, if any, the XID cache will contain responses for.

Writing the Code

ONC: Write a program that performs the following steps:

1. To set up the server:
   a. Call `svcudpa_create` and/or `svctcpa_create` for each thread. Save the return values for later use. Use the `svc_getchan` routine after the first call and specify the returned value in subsequent calls.
   b. If you are enabling the XID cache, call `svcudpa_enablecache` and/or `svctcpa_enablecache` for each thread. Use the result of the first call to `svcudpa_enablecache/svctcpa_enablecache` as input to successive calls to that routine, so that all threads use the same cache.
   c. Call the `svc_register` routine once for each transport to register the service.
2. Put the main code in "hibernation" by calling SYS$HIBER instead of `svc_run`. SYS$HIBER is an OpenVMS system service that accepts no input arguments.
3. To shut down the server:
   a. Perform any shutdown steps that are specific to your server.
   b. Call `svctcpa_shutdown` once for all TCPA threads.
   c. Call `svcudpa_shutdown` once for all UDPA threads.
   d. Call `svc_destroy` once for each thread.

How Asynchronous Transports Affect Memory

ONC: Asynchronous transports use more memory than synchronous transports because each service has more than one thread.

When TCPA receives requests greater than 4Kbytes, it uses significantly more memory than TCP uses for the same size requests. TCP reads in 4Kbytes of data, processes the data, then reads more data as necessary. TCPA reads all of the data before calling the server dispatch routine.

Asynchronous System Traps

ONC: All UDPA and TCPA servers use asynchronous system traps (ASTs). Refer to the *Guide to VMS Programming Resources* for complete information on ASTs.

When using ASTs, follow these guidelines:

- Do not permit the main program to disable ASTs for long periods of time. If the program disables ASTs before it calls SYS$HIBER, the server does not respond to requests.

- The AST quota for the server process must allow at least one AST for each thread.
- The `dispatch` routine is called as a user-mode AST. This means a program cannot do synchronous waiting when an AST is required to wake up the program.
- Only one AST can be active at a time.

RPCINFO Utility

RPCINFO is an ONC RPC utility that allows you to:

- Request a listing of all programs registered with a Port Mapper.
- Call the NULL routine of any program

RPCINFO supports both the UDP and TCP protocols.

Requesting a Program Listing

To request a listing of all programs that are registered with the Port Mapper, enter the RPCINFO command in the following format at the DCL prompt:

RPCINFO -p *[***-u** | **-t***][**host-name**]*

-p	Calls the Port Mapper.
-u	Uses the UDP transport to send the request. This is the default if you do not specify a transport.
-t	Uses the TCP transport to send the request.
host-name	Specifies the domain name of the host on which the Port Mapper resides. If you omit this parameter, RPCINFO uses the name of the local host.

Example 13-1 shows an example.

Example 13-1 Sample RPCINFO Program Listing

```
$ rpcinfo -p hermes

   Program  Version  Protocol   Port
   -------  -------  --------   ----
    100000        2       udp    111  rpcbind
    100000        2       tcp    111  rpcbind
    100005        1       udp   2049 mountd
```

Calling a NULL Routine

To call the NULL routine of a program, enter the RPCINFO command in the following format at the DCL prompt:

RPCINFO *[***-u** | **-t***]**host-nameprogram**[**version**]*

-u	Uses the UDP transport to send the request. If you do not specify a trans port, this is the default.
-t	Uses the TCP transport to send the request.
host-name	Specifies the domain name of the target host.
program	Specifies the program number of the target program.
version	Specifies the version number of the target program. If you omit the version, the default is 1.

For example, suppose you enter the following command:

```
RPCINFO ETA 100000 2
```

The following message displays if the call completes successfully:

```
Version 2 of program 100000 successfully called
```

Chapter 14 RPCGEN Compiler

Introduction

This chapter is for RPC programmers.

What Is RPCGEN?

RPCGEN is the RPC Protocol Compiler. This compiler creates the network interface portion of a distributed application, effectively hiding from the programmer the details of writing and debugging low-level network interface code.

You are not required to use RPCGEN when developing a distributed application. If speed and flexibility are critical to your application, you can write the network interface code yourself, using ONC RPC Run-Time Library (RTL) calls where they are needed.

Compiling with RPCGEN is one step in developing distributed applications. See Chapter 13, *Building Distributed Applications*, for a complete description of the application development process.

RPCGEN allows you to use the highest layer of the RPC programming interface. The *ONC RPC Fundamentals*, Chapter 12, provides details on these layers.

Software Requirements

XDR: The following software must be installed on your system before you can use RPCGEN:

VMS Version 5.0 or later	DEC C compiler Version 3.2 or later

ONC: The following software must be installed on your system before you can use RPCGEN:

VMS Version 5.0 or later	VAX C compiler Version 3.0 or later

Input Files

The RPCGEN compiler accepts as input programs called *interface definitions*, written in RPC Language(RPCL), an extension of XDR language. RFC 1057 and RFC 1014 describe these languages in detail.

An interface definition must always contain the following information:

- Remote program number
- Version number of the remote program
- Remote procedure number(s)
- Input and output arguments

Example 14-1 shows a sample interface definition from the TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]PRINT.X file.

Example 14-1 Interface Definition

```
/*
** RPCGEN input file for the print file RPC batching example.
**
** This file is used by RPCGEN to create the files PRINT.H and PRINT_XDR.C
** The client and server files were developed from scratch.
*/

const MAX_STRING_LEN = 1024;    /* maximum string length */

/*
** This is the information that the client sends to the server

*/
struct a_record
{
        string  ar_buffer< MAX_STRING_LEN>;
};

program PRINT_FILE_PROG
{        version PRINT_FILE_VERS_1
        {
          void   PRINT_RECORD( a_record) = 1;
          u_long SHOW_COUNT( void) = 2;
        } = 1;
} = 0x20000003;


/* end file PRINT.X */
```

The default extension for RPCGEN input files is .X.

You do not need to call the ONC RPC RTL directly when writing an interface definition. RPCGEN inserts the necessary library calls in the output file.

Output Files

RPCGEN output files contain code in C language. Table 14-1 lists the RPCGEN output files and summarizes their purpose. You can edit RPCGEN output files during application development. The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC] directory provides sample RPCGEN output files.

Table 14-1 RPCGEN Output Files

File	Purpose
Client and server stub calls	Interface between the network and the client and server programs. Stubs use RPC RTL to communicate with the network.
XDR routines	Convert data from a machine's local data format to XDR for mat, and vice versa.
Header	Contains common definitions, such as those needed for any structures being passed.

Invoking RPC explains how to request specific output files.

Table 14-2 shows the conventions you should use to name output files.

Table 14-2 RPCGEN File Naming Conventions

File	Output Filename
Client stub	*inputname*_CLNT.C
Server stub	*inputname*_SVC.C
Header file	*inputname*.H
XDR filter routines	*inputname*_XDR.C

– *inputname* is the name of the input file. For example, if the input file is TEST.X, the server stub is TEST_SVC.C.

When you use the RPCGEN command to create all output files at once, RPCGEN creates the output filenames listed in RPCGEN File Naming Conventions by default. When you want to create specific kinds of output files, you must specify the names of the output files in the command line.

Preprocessor Directives

RPCGEN runs the input files through the C preprocessor before compiling. You can use the macros listed in Table 14-3 with the `#ifdef` preprocessor directive to indicate that specific lines of code in the input file are to be used only for specific RPCGEN output files.

Table 14-3 Macros

File	Macro
Client stub	RPC_CLNT
Server stub	RPC_SVC
Header file	RPC_HDR
XDR filter routines	RPC_XDR

Invoking RPCGEN

This section explains how to invoke RPCGEN to create:

- All output files at once
- Specific output files
- Server stubs for either the TCP or UDP transport

Creating All Output Files at Once

This command creates all four RPCGEN output files at once:

RPCGEN *input*

where*input* is the name of the file containing the interface definition.

In the following example, RPCGEN creates the output files PROGRAM.H, PROGRAM_CLNT.C, PROGRAM_SVC.C, and PROGRAM_XDR.C:

```
RPCGEN PROGRAM.X
```

Creating Specific Output Files

This command creates only the RPCGEN output file that you specify:

RPCGEN {**-c** |**-h** | **-l** | **-m**}*[***-o** *output]input*

-c	Creates an XDR filter file (_XDR.C)
-h	Creates a header file (.H)
-l	Creates a client stub (_CLNT.C)
-m	Creates a server stub (_SVC.C) that uses both the UDP and TCP transports
-o	Specifies an output file (or the terminal if no output file is given)
output	Name of the output file
input	Name of an interface definition file with a .X extension

Follow these guidelines:

- Specify just one output file (`-c`, `-h`, `-l`, or `-m`) in a command line
- If you omit the output file, RPCGEN sends output to the terminal screen

Examples:

1 RPCGEN -h PROGRAM

RPCGEN accepts the file PROGRAM.X as input and sends the header file output to the screen, because no output file is specified.

2 RPCGEN -l -o PROGRAM_CLNT.C PROGRAM.X

RPCGEN accepts the PROGRAM.X file as input and creates the PROGRAM_CLNT.C client stub file.

3 RPCGEN -m -o PROGRAM_SVC.C PROGRAM.X

RPCGEN accepts the PROGRAM.X file as input and creates the PROGRAM_SVC.C server stub file. The server can use both the UDP and TCP transports.

Creating Server Stubs for TCP or UDP Transports

This command creates a server stub file for either the TCP or UDP transport:

RPCGEN -s {**udp** | **tcp**}*[***-o** *output]input*

-s	Creates a server (_SVC.C) that uses either the UDP or TCP transport (with `-s`, you must specify either **udp** or **tcp**; do not also use `-m`)
udp	Creates a UDP server

tcp	Creates a TCP server
-o	Specifies an output file (or the terminal if no output file is given)
output	Name of the output file
input	Name of an interface definition file with a .X extension

If you omit the output file, RPCGEN sends output to the terminal screen.

In this example, RPCGEN accepts the PROGRAM.X file as input and creates the PROGRAM_SVC.C output file, containing a TCP server stub:

```
RPCGEN -s tcp -o PROGRAM_SVC.C PROGRAM.X
```

Error Handling

RPCGEN stops processing when it encounters an error. It indicates which line the error is on.

Restrictions

RPCGEN does not support the following:

- The syntax `int x, y;` . You must write this as **`int x; int y;`**
- Asynchronous transports

Chapter 15 RPC RTL Management Routines

Introduction

This chapter is for RPC programmers. It introduces RPC Run-Time Library (RTL) conventions and documents the management routines in the RPC RTL. These routines are the programming interface to RPC.

Management Routines

There are two types of services with RTL:

– ONCRPC used with **VAX C** and the TCPware Socket Library
– RPCXDR used with **DEC C** and the HP Socket Library

The bold letters in the above list show the conventions this book uses to distinguish between the two services.

The RPC RTL contains:

- RPC management routines
- **DEC C:** RPC client and server routines for the UDP and TCP transport layers
- **VAX C:** RPC client and server routines for the UDP, UDPA, TCP, and TCPA transport layers
- **DEC C:** On VAX, Alpha and I64 systems, RPC provides a single shareable image accessed via the UCX$RPCXDR_SHR logical. This shareable image contains routines for all of the DEC C floating-point types. The correct routines will automatically be called based on the compiler options used to compile the RPC application. See the DEC C documentation for how to use the floating-point compiler options.
- **VAX C:** On VAX systems, RPC provides two shareable RTLs:
  – A D_float library, for standard double-precision real numbers between 10^{-38} and 10^{+38}. This library is in the SYS$SHARE:TCPWARE_RPCLIB_SHR.EXE file.
  – A G_float library, for double-precision real numbers between 10^{-308} and $10^{+308.}$ This library is in the SYS$SHARE:TCPWARE_RPCLIBG_SHR.EXE file.
- **DEC C:** On Alpha and I64 systems, ONC RPC provides three shareable RTLs:
  – A D_float library, for standard double-precision real numbers between 10^{-38} and 10^{+38}. This library is in the SYS$SHARE:TCPWARE_RPCLIBD_SHR.EXE file.
  – A G_float library, for double-precision real numbers between 10^{-308} and $10^{+308.}$ This library is in the SYS$SHARE:TCPWARE_RPCLIB_SHR.EXE file.
  – A T_float library, for IEEE double-precision real numbers between 10^{-308} and $10^{+308.}$ This library is in the SYS$SHARE:TCPWARE_RPCLIBT_SHR.EXE file.
- **VAX C:** Invoke these libraries by using the appropriate qualifiers when you compile your application programs. See your HP VAX documentation for instructions on compiling.

Chapter 13, *Building Distributed Applications with RPC*, explains how to link in the RPC RTL.

Routine Name Conventions

In this chapter, all routines are documented according to their standard UNIX names. Routines that are unique to TCPware have UNIX-style names.

If you are writing code in C language, you may use the routine names used in this chapter. If you are writing code in a different language, however, you must use the TCPware names defined in the SYS$COMMON:[TCPWARE.INCLUDE.RPCOLD]:ONCRPC_FUNC.H file. These names all begin with the letters `ONCRPC`.

Header Files

All RPC programs include the file named RPC.H. Locations for this file are:

– **VAX C**: SYS$COMMON:[TCPWARE.INCLUDE.RPCOLD]: RPC.H
– **DEC C**: UCX$RPC:RPC.H

The RPC.H file includes the files listed in Table 15-1.

Table 15-1 Header Files Included In RPC.H

Filename	Purpose
Pertains to DEC C and VAX C	
AUTH.H	Used for authentication.
AUTH_UNIX.H	Contains XDR definitions for UNIX-style authentication.
CLNT.H	Contains various RPC client definitions.
IN.H	Defines structures for the internet and socket addresses (*`in_addr`* and *`sockaddr_in`*). This file is part of the C Socket Library.
RPC_MSG.H	Defines the RPC message format.
SVC.H	Contains various RPC server definitions.
SVC_AUTH.H	Used for server authentication.
TYPES.H	Defines UNIX C data types.
XDR.H	Contains various XDR definitions.
Pertains to VAX C only[1]	
ONCRPC_CONST.H	Defines RPC characteristics and other constants.
ONCRPC_FUNC.H	Maps ONCRPC routine names with their UNIX counterparts.
ONCRPC_STRUCT.H	Defines structures for the RPC client and server counters.
Pertains to DEC C only	
NETDB.H	Defines structures and routines to parse /etc/rpc.

There is an additional header file not included by RPC.H that is used by xdr_pmap and xdr_pmaplist routines. The file name is pmap_prot.h, and the location is:

[1] Specific to the ONCRPC library for TCPware.

```
VAX C: SYS$COMMON:[TCPWARE.INCLUDE.RPCOLD]:PMAP_PROT.H
DEC C: UCX$RPC:PMAP_PROT.H
```

Boolean Values

Many ONC RPC routines return TRUE or FALSE values. In C, FALSE is zero, and TRUE is any other value.

TCPware/Sun Implementation Differences

ONC RPC Services are based on the Open Network Computing Remote Procedure Call protocols developed by Sun Microsystems, Inc. This section lists the ways in which the TCPware's implementation of ONC RPC Services differs from Sun's UNIX implementation of RPC.

- **DEC C:** TCPware provides the following management routines:

get_myaddress	getrpcbynumber	getrpcport

- **VAX C:** TCPware provides the following management routines:

ONCRPC_GET_CHAR	ONCRPC_SET_CHAR	ONCRPC_GET_STATS

  These routines allow you to retrieve and maintain information that describes how a process is using ONC RPC. Sun does not provide these routines.

- **VAX C:** TCPware does not support the following Sun routines:

svcstdio_create	svcraw_create	clntraw_create

  These routines deal with transports that the OpenVMS environment does not support.

- The `svcfd_create` routine provided by TCPware supports only TCP sockets. The Sun version supports file descriptors, including `stdin` and `stdout`.
- In TCPware, the global variable `svc_fdset` contains an array of structures, where each element is a socket pointer and a service handle. Sun implements this variable as a bit mask with an associated array of service handles.
- The `authunix_create_default` routine provided by TCPware creates credentials based on the VAX C `geteuid` and `getegid` routines.
- TCPware provides the `pmap_freemaps` routine to free the memory that was allocated by the `pmap_getmaps` routine. Sun requires the programmer to know the internals of the `pmaplist` structure and free the memory himself.
- TCPware provides the `xdr_netobj` routine. This routine encodes and decodes `netobj`, an aggregate data structure that is opaque and contains a counted array of 1024 bytes.
- TCPware provides the `xdr_netobj` routine. This routine encodes and decodes `netobj`, an aggregate data structure that is opaque and contains a counted array of 1024 bytes.
- **ONC:** TCPware provides asynchronous transports, with the following routines:

svctcpa_create	svctcpa_getxdrs	svcudpa_enablecache

svctcpa_shutdown	svcudpa_bufcreate	svcudpa_freecache
svctcpa_enablecache	svcudpa_create	svcudpa_getxdrs
svctcpa_freecache	svcudpa_shutdown	

Sun does not provide asynchronous transports.

- **VAX C:** TCPware provides the following macros:

svc_getchan	svc_getport

Sun does not provide these macros.

- The TCPware RPCGEN compiler gives the input file a default extension of .X if no extension is specified. The Sun RPCGEN compiler requires you to enter the .X extension.
- The TCPware RPCGEN compiler accepts input only from a file, not from the terminal. The Sun RPCGEN compiler accepts both types of input.
- The TCPware RPCINFO command supports the TCP and UDP protocol for any request. The Sun version supports only the UDP protocol when you use RPCINFO to request a Port Mapper listing.

Management Routines

RPC management routines retrieve and maintain information that describes how a process is using RPC. This section describes each management routine and function in detail. The following information is provided for each routine:

- Format
- Arguments
- Description
- Diagnostics, or status codes returned, if any

get_myaddress

DEC C and **VAX C** Returns the internet address of the local host.

Format

```
#include
void get_myaddress (struct sockaddr_in *addr);
```

Argument

addr

Address of a `sockaddr_in` structure that will be loaded with the host internet address. The port number is always set to `htons(PMAPPORT)`.

Description

The `get_myaddress` routine returns the internet address of the local host without doing any name translation or DNS lookups.

getrpcbynumber

DEC C Gets an RPC entry.

Format

```
#include
struct rpcent *getrpcbynumber(number)
int number;
```

Argument

number

Program name or number.

Description

The `getrpcbynumber` routine returns a pointer to an object with the following structure containing the broken-out fields of a line in the RPC program number database, `/etc/rpc`.

```
struct rpcent {
     char *r_name;       /* name of server for this RPC program */
     char **r_aliases;   /* alias list */
     long r_number;      /* RPC program number */
    };
```

The members of this structure are:

`r_name`	Name of the server for this RPC program
`r_aliases`	Zero-terminated list of alternate names for the RPC program
`r_number`	RPC program number for this service

The `getrpcbynumber` routine sequentially searches from the beginning of the file until a matching RPC program name or program number is found, or until an EOF is encountered.

Diagnostics

A NULL pointer is returned on EOF or error.

getrpcport

DEC C Gets an RPC port number.

Format

```
int getrpcport(host, prognum, versnum, proto)

char *host;

int prognum, versnum, proto;
```

Arguments

host

Host running the RPC program.

prognum

Program number.

proto

Protocol name. Must be IPPROTO_TCP or IPPROTO_UDP.

Description

The `getrpcport` routine returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*.

It returns 0 if it cannot contact the portmapper, or if *prognum* is not registered. If *prognum* is registered but not with *versnum*, it still returns a port number (for some version of the program), indicating that the program is indeed registered. The version mismatch is detected on the first call to the service.

ONCRPC_GET_CHAR

VAX C Returns the characteristic values of an RPC client or server process.

Format

```
u_long ONCRPC_GET_CHAR(code, value)

u_long code;
void *value;
```

Arguments

code

Characteristic being returned. The following text describes each code.

RPCCHAR_ _AC_SIZE

Alters the number of entries in the active cache. This code is for UDPA transports only. If you try to set RPCCHAR__AC_SIZE when a UDPA transport is active, then RPC_SET_CHAR returns the SS$_DEVACTIVE error code. The type is u_short *value. The default is 20.

RPCCHAR_ _CHECKSUM

Enables or disables checksums for outgoing packets. The receiver checks packets if the sender generates a checksum. Disabling checksums improves performance, but reduces data integrity. This code is for UDPA servers only. Valid values are RPCCKSUM__ENABLE and RPCCKSUM__DISABLE. The type is u_long *value. The default is RPCCKSUM__ENABLE.

RPCCHAR_ _DEBUG

Enables the printing of messages that indicate what RPC is doing. All logging messages go to SYS$OUTPUT. The type is u_long *value. The default is 0.

These are the mask values for RPCCHAR__DEBUG:

RPCDBG_ _XDR	Prints a message indicating which XDR routines are being used.
RPCDBG_ _XID	Prints a message each time the XID cache is referenced. The message indicates the type of operation being per formed on the XID cache. For UDP and UDPA servers only.
RPCDBG_ _ACTIVE_CA	Prints a message each time the active cache is referenced. The message indicates the type of operation being per formed on the active cache. For UDPA servers only.
RPCDBG_ _GENERAL	Prints general messages about the activities of RPC RTL routines.
RPCDBG_ _RAW_RCV	Prints, in hexadecimal and ASCII, the contents of packets that were received.
RPCDBG_ _RAW_SEND	Prints, in hexadecimal and ASCII, the contents of packets that were sent.

RPCDBG_ _AUTH	Prints authentication processing information.
RPCDBG_ _MEMORY	Tracks memory allocation for the RPC RTL.

rRPCCHAR_ _DEFPORTS

If an RPC program does not specify a port to use, RPC assigns it a port. The RPCCHAR_ _DEFPORTS code determines whether RPC tries to use privileged ports (600 to 1023) or non-privileged ports (1024 and greater). The type is u_long**value*. Values are RPCPORTS_ _PRIVILEGE and RPCPORTS_NORMAL. RPCPORTS_PRIVILEGE is the default, which means the application uses a privileged port if it has sufficient privileges, otherwise it uses a non-privileged port.

RPCCHAR_ _FATALRTN

Calls an error-handling routine when the TCPA or UDPA transports detect a fatal error. The OpenVMS error status code is passed to the error-handling routine. If *value is zero, the default value is used. The type is void (***value*)(u_long status). The default is SYS$EXIT.

value

Address where the routine places the characteristic value.

Description

The characteristic values are defined as constants in the RPC_CONST.H file. Each process using the RPC library has its own copy of these values. The ONCRPC_GET_CHAR routine retrieves these values.

Example

```
u_long debug;

oncrpc_get_char( RPCCHAR__DEBUG, &debug);
printf( "The debugging value is %08X", debug);
```

This example prints a message that shows the present debugging value for RPC. The message goes to SYS$OUTPUT.

Diagnostics

SS$_NORMAL	Routine executed successfully.
SS$_BADPARAM	*code* is invalid.
SS$_ACCVIO	*value* is a null address.

See Also

ONCRPC_SET_CHAR

ONCRPC_GET_STATS

VAX C Returns counters for memory usage, the RPC server, or the RPC client.

Format

```
u_long ONCRPC_GET_STATS(code, buffer)

u_long code;
u_char *buffer;
```

Arguments

code

These codes are valid:

RPCSTAT_ _ZERO	Zeros out the client or server counters.
RPCSTAT_ _CLIENT	Retrieves client counters.
RPCSTAT_ _SERVER	Retrieves server counters.
RPCSTAT_ _MEMORY	Retrieves memory counters.

The Example shows how to use the `RPCSTAT_ _ZERO` code in combination with the other codes to zero-out the client and server counters after they are retrieved. You cannot zero-out memory counters.

buffer

Address of the structure to receive the requested counters. These structures are defined in the RPC_STRUCT.H file. The length of the buffer equals the size of the structures. These are the structures:

```
/*
** Statistics structure (server)
*/
typedef struct
{
time_tzero_time;                /* # seconds since zeroed*/
u_longrecvs,                    /* # receives*/
recv_errs,                      /* # bad receives*/
xmits,                          /* # transmits*/
xmit_errs,                      /* # transmit errors*/
xerr_badlen                     /* # calls w/body too small*/
xerr_nullrecv,                  /* # empty calls*/
xerr_auth_weak,                 /* # weak auth errors*/
xerr_auth_other,                /* # other auth errors*/
xerr_decode,                    /* # decode errors
xerr_noproc,                        /* # no procedure errors*/
xerr_noprog,                        /* # no program errors*/
xerr_novers,                        /* # no matching version errors*/
xerr_systemerr,                     /* # other system errors*/
xid_hits,                           /* # replies sent from xid cache*/
xid_saves,                          /* # replies cached*/
```

```
duprcvs;                              /* # calls found in active cache*/
} svc_counters;
/*
** Statistics structure (client)
*/
typedef struct
{
time_t             zero_time;         /* # seconds since zeroed*/
u_long             recvs,             /* # receives */
                   recv_errs,         /* # receive errors*/
                   badxid,            /* # xid mismatches*/
                   xmits,             /* # transmits*/
                   xmit_errs          /* # transmit errors*/
                   newcred,           /* # new credentials (always 0)*/
                   retrans,           /* # retransmissions*/
                   timeout,           /* # timeouts*/
                   wait;              /* (always 0)*/
} clnt_counters;
/*
** Memory statistics structure
*/
typedef struct
{
u_long             no_mallocs,         /* # times memory was allocated*/
                   no_frees,           /* # times memory was freed*/
                   mem_in_use;         /* # bytes of memory in use*/
} mem_counters;
```

Description

The ONCRPC_GET_STATS routine returns counters for this process only.

Example

```
clnt_counters cstats;

oncrpc_get_stats( RPCSTAT__CLIENT | RPCSTAT__ZERO, &cstats);
/* cstats now contains the client statistics */
```

This example retrieves client counters, then zeros them out.

Diagnostics

SS$_NORMAL	Routine successfully returned the counters.
SS$_BADPARAM	*code* is invalid.
SS$_ACCVIO	*buffer* is a null address.

ONCRPC_SET_CHAR

VAX C Defines the values of characteristics for each RPC client and server process.

Format

```
u_long ONCRPC_SET_CHAR(code, value)

u_long code;
void *value;
```

Arguments

code

Characteristic being set. See theONCRPC_GET_CHAR routine for a description of each code.

value

Address of the characteristic value.

Description

Use the `ONCRPC_SET_CHAR` routine to define characteristic values for RPC processes. Each RPC process has its own values. Codes and values are defined as constants in the RPC_CONST.H file.

Example

```
void exit_routine( status)
  u_long        status;
{
        exit( status);
}

main()
{
        void    (*fatal_routine)();

        fatal_routine = exit_routine;
        /* fatal_routine is the address of exit_routine */
        oncrpc_set_char( RPCCHAR__FATALRTN, &fatal_routine);
}
```

Sets the exit routine to be called when the TCPA or UDPA transport detects a fatal error.

Diagnostics

SS$_NORMAL	Routine executed successfully.
SS$_ACCVIO	*value* is a null address.
SS$_BADPARAM	*code* is invalid.

SS$_DEVACTIVE	Process tried to change the size of the active cache while the UDPA transport was active.

See Also

ONCRPC_GET_CHAR

Chapter 16 ONC RPC RTL Client Routines

Introduction

This chapter is for RPC programmers. It documents the client routines in the ONC RPC Run-Time Library (RTL). These routines are the programming interface to ONC RPC.

Common Arguments

Many client, Port Mapper, and server routines use the same arguments.

Table 16-1 lists these arguments and defines their purpose. Arguments that are unique to each routine are documented together with their respective routines in this and the following chapters

Table 16-1 Common Arguments

Argument	Purpose
args_ptr	Address of the buffer to contain the decoded RPC arguments.
auth	RPC authentication client handle created by the `authnone_create`, `authunix_create`, or `authunix_create_default` routine.
clnt	Client handle returned by any of the client create routines.
in	Input arguments for the service procedure.
inproc	XDR routine that encodes input arguments.
out	Results of the remote procedure call.
outproc	XDR routine that decodes output arguments.
procnum	Number of the service procedure.
prognum	Program number of the service program.
protocol	Transport protocol for the service. Must be IPPROTO_UDP or IPPROTO_TCP.
s	String containing the message of your choice. The routines append an error message to this string.

sockp	Socket to be used for this remote procedure call. If *sockp* is RPC_ANYSOCK, the routine creates a new socket and defines *sockp*. The `clnt_destroy` routine closes the socket. If *sockp* is a value other than RPC_ANYSOCK, the routine uses this socket and ignores the internet address of the server.
versnum	Version number of the service program.
xdr_args	XDR procedure that describes the RPC arguments.
xdrs	Structure containing XDR encoding and decoding information.
xprt	RPC server handle.

Client Routines

The client routines are called by the client main program or the client stub procedures.

The following sections describe each client routine in detail.

auth_destroy

ONC A macro that destroys authentication information associated with an authentication handle.

Format

```
void auth_destroy (AUTH *auth)
```

Argument

auth

RPC authentication client handle created by the `authnone_create`, `authunix_create`, or `authunix_create_default` routine.

Description

Use `auth_destroy` to free memory that was allocated for authentication handles. This routine undefines the value of *auth* by deallocating private data structures.

Do not use this memory space after `auth_destroy` has completed. You no longer own it.

See Also

`authnone_create, authunix_create, authunix_create_default`

authnone_create

XDR ONC Creates and returns a null RPC authentication handle for the client process.

Format

```
#include

AUTH *authnone_create();
```

Arguments

None.

Description

This routine is for client processes that require no authentication. RPC uses it as a default when it creates a client handle.

See Also

auth_destroy, authnone_create, authunix_create_default, clnt_create, clntraw_create, clnttcp_create, clntudp_create / c

authunix_create

XDR ONC Creates and returns an RPC authentication handle for the client process. Use this routine when the server requires UNIX-style authentication.

Format

```
#include

AUTH *authunix_create (char *host, int uid,int gid,int len,int gids);
```

Arguments

host

Address of the name of the host that created the authentication information. This is usually the local host running the client process.

uid

User ID of the person who is executing this process.

gid

User's group ID.

len

Number of elements in the **gids* array.

gids

Address of the array of groups to which the user belongs.

Description

Since the client does not validate the *uid* and *gid*, it is easy to impersonate an unauthorized user. Choose values the server expects to receive. The application must provide OpenVMS-to-UNIX authorization mapping.

You can use a Socket Library lookup routine to get the host name.

See Also

auth_destroy, authnone_create, authunix_create_default

authunix_create_default

XDR ONC Calls the `authunix_create` routine and provides default values as arguments.

Format

```
#include
AUTH *authunix_create_default()
```

Arguments

See below.

Description

Like the `authunix_create` routine, `authunix_create_default` provides UNIX-style authentication for the client process. However, `authunix_create_default` does not require you to enter any arguments. Instead, this routine provides default values for the arguments used by `authunix_create`, listed in Table 16-2.

Table 16-2 Default Arguments

Argument	Default Value
host	local host domain name
uid	getuid ( )
gid	getgid ( )
len	0
gids	0

This routine is provided to ensure compatibility with Sun Microsystems' ONC RPC. You can replace this call with `authunix_create` and provide appropriate values.

Example

```
auth_destroy(client->cl_auth);
client->cl_auth = authunix_create_default();
```

This example overrides the `authnone_create` routine, where *`client`* is the value returned by the `clnt_create`, `clntraw_create`, `clnttcp_create`, or `clntudp_create` routine.

See Also

`auth_destroy, authnone_create, authunix_create`

callrpc

XDR **ONC** Calls the remote procedure identified by the routine's arguments.

Format

```
#include

int callrpc (char *host,u_long prognum,u_long versnum,u_long procnum,xdrproc_t
inproc,u_char *in,
xdrproc_t outproc,u_char *out);
```

Arguments

host

Host where the procedure resides.

prognum, versnum, procnum, inproc, in, outproc, out

See Common Arguments for a description of the above arguments.

Description

The `callrpc` routine performs the same functions as the `clnt_create`, `clnt_call`, and `clnt_destroy` routines.

Since the `callrpc` routine uses the UDP transport protocol, messages can be no larger than 8Kbytes. This routine does not allow you to control timeouts or authentication.

If you want to use the TCP transport, use the `clnt_create` or `clnttcp_create` routine.

Diagnostics

The `callrpc` routine returns zero if it succeeds, and the value of `enum clnt_stat` cast to an integer if it fails.

You can use the `clnt_perrno` routine to translate failure status codes into messages.

See Also

`clnt_broadcast, clnt_call, clnt_create, clnt_destroy,`
`clnt_perrno / c, clnttcp_create`

clnt_broadcast

XDR ONC Broadcasts a remote procedure call to all local networks, using the broadcast address.

Format

```
#include

enum clnt_stat clnt_broadcast (u_long prognum, u_long versnum, u_long procnum,
xdrproc_t inproc, u_char *in,
xdrproc_t outproc, u_char *out, resultproc_t eachresult);
```

Arguments

prognum, versnum, procnum, inproc, in, outproc, out

See Common Arguments for a description of the above arguments.

eachresult

Each time `clnt_broadcast` receives a response, it calls the *eachresult* routine. If *eachresult* returns zero, `clnt_broadcast` waits for more replies. If *eachresult* returns a nonzero value, `clnt_broadcast` stops waiting for replies. The *eachresult* routine uses this form:

int eachresult(*out, addr*)

u_char **out*;

struct sockaddr_in **addr*;

out	Contains the results of the remote procedure call, in the local data format.
*addr	Is the address of the host that sent the results.

Description

The `clnt_broadcast` routine performs the same functions as the `callrpc` routine. However, `clnt_broadcast` sends a message to all local networks, using the broadcast address. The `clnt_broadcast` routine uses the UDP protocol.

Table 16-3 indicates how large a broadcast message can be.

Table 16-3 Maximum Message Size

Line	Maximum Size
Ethernet	1500 bytes
proNet	2044 bytes

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]SYSINFO.C file provides a sample program using `clnt_broadcast`.

Diagnostics

This routine returns diagnostic values defined in the CLNT.H file for `enumclnt_stat`.

See Also

`callrpc, clnt_perrno / c`

clnt_call

ONC A macro that calls a remote procedure.

Format

```
enum clnt_stat clnt_call (CLIENT *clnt, u_long procnum,
xdrproc_t inproc, u_char *in, xdrproc_t outproc, u_char *out,
struct timeval tout);
```

Arguments

clnt, procnum, inproc, in, outproc, out

See Common Arguments for a description of the above arguments.

tout

Time allowed for the results to return to the client, in seconds and microseconds. If you use the `clnt_control` routine to change the `CLSET_TIMEOUT` code, this argument is ignored.

Description

Use the `clnt_call routine` after using `clnt_create`. After you have finished with the client handle, use the `clnt_destroy` routine. You can use the `clnt_perror` routine to print messages for any errors that occurred.

Diagnostics

This routine returns diagnostic values defined in the CLNT.H file for `enumclnt_stat`.

See Also

`clnt_control, clnt_create, clnt_destroy,`
`clnt_perrno / c`

clnt_control

ONC A macro that changes or retrieves information about an RPC client process.

Format

```
bool_t clnt_control (CLIENT *clnt, u_long code, void *info);
```

Arguments

clnt

Client handle returned by any of the client create routines.

code

Code listed in Table 16-4.

Table 16-4 Valid Codes

Code	Type	Purpose
CLSET_TIMEOUT	struct timeval	Set total timeout
CLGET_TIMEOUT	struct timeval	Get total timeout
CLSET_RETRY_TIMEOUT*	struct timeval	Set retry timeout
CLGET_RETRY_TIMEOUT*	struct timeval	Get retry timeout
CLGET_SERVER_ADDR	struct sockaddr_in	Get server address
* Valid only for the UDP transport protocol.		

The `timeval` is specified in seconds and microseconds. The total timeout is the length of time that the client waits for a reply. The default total timeout is 25 seconds.

The retry time is the length of time that UDP waits for the server to reply before transmitting the request. The default retry timeout is 5 seconds. You might want to increase the retry time if your network is slow.

For example, suppose the total timeout is 10 seconds and the retry time is five seconds. The client sends the request and waits five seconds. If the client does not receive a reply, it sends the request again. If the client does not receive a reply within five seconds, it does not send the request again.

If you use `CLSET_TIMEOUT` to set the timeout, the `clnt_call` routine ignores the timeout parameter it receives for all future calls.

info

Address of the information being changed or retrieved.

Diagnostics

This routine returns TRUE if it succeeds, and FALSE if it fails.

See Also

```
clnt_call, clnt_create, clnt_destroy, clntraw_create,
clnttcp_create, clntudp_create / c
```

clnt_create

XDR ONC Creates an RPC client handle.

Format

```
#include

CLIENT *clnt_create (char *host, u_long prognum, u_long versnum, char *proto);
```

Arguments

host

Address of the string containing the name of the remote host where the server is located.

prognum, versnum

See Common Arguments for a description of the above arguments.

proto

Address of a string containing the name of the transport protocol. Valid values are **UDP** and **TCP**. The *ONC RPC Fundamentals* chapter explains the advantages and disadvantages of each transport protocol.

Description

The `clnt_create` routine creates an RPC client handle for *`prognum`*. An RPC client handle is a structure containing information about the RPC client. The client can use the UDP or TCP transport protocol.

This routine uses thePort Mapper. You cannot control the local port.

The default sizes of the send and receive buffers are 8800 bytes for the UDP transport, and 4000 bytes for the TCP transport.

The retry time for the UDP transport is five seconds.

Use the `clnt_create` routine instead of the `callrpc` or `clnt_broadcast` routines if you want to use one of the following:

- The TCP transport
- An authentication other than null
- More than one active client at the same time

You can also use `clntraw_create` to use the IP protocol, `clnttcp_create` to use the TCP protocol, or `clntudp_create` to use the UDP protocol.

The `clnt_create` routine uses the global variable `rpc_createerr`. `rpc_createerr` is a structure that contains the most recent service creation error. Use `rpc_createerr` if you want the client program to handle the error. The value of `rpc_createerr` is set by any RPC client creation routine that does not succeed.

The `rpc_createerr` variable is defined in the CLNT.H file.

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI_CLNT_CALL.C file provides a sample program that uses `clnt_create`.

Diagnostics

The `clnt_create` routine returns the address of the client handle, or zero (if it could not create the client handle).

If the `clnt_create` routine fails, you can use the `clnt_pcreateerror` or `clnt_spcreateerror` routines to obtain diagnostic information.

See Also

```
clnt_call, clnt_control, clnt_destroy,
clnt_pcreateerror / c, clntraw_create,
clnttcp_create, clntudp_create / c
```

clnt_destroy

ONC A macro that destroys an RPC client handle.

Format

```
void clnt_destroy (CLIENT *clnt);
```

Argument

clnt

Client handle returned by any of the client create routines.

Description

The `clnt_destroy` routine destroys the client's RPC handle by deallocating all memory related to the handle. The client is undefined after the `clnt_destroy` call.

If the `clnt_create` routine had previously opened a socket, this routine closes the socket. Otherwise, the socket remains open.

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI_CLNT_CALL.C file provides a sample program that uses `clnt_destroy`.

See Also

`clnt_create, clntraw_create, clnttcp_create, clntudp_create / c`

clnt_freeres

ONC A macro that frees the memory that was allocated when the RPC results were decoded.

Format

```
bool_t clnt_freeres (CLIENT *clnt, xdrproc_t xdr_res, char *res_ptr);
```

Arguments

clnt

Client handle returned by any of the client create routines.

xdr_res

Address of the XDR procedure that describes the RPC results.

res_ptr

Address of the RPC results.

Description

The `clnt_freeres` routine calls the `xdr_free` routine.

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI_CLNT_CALL.C file provides a sample program that uses `clnt_freeres`.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_free`

clnt_geterr

ONC A macro that returns an error code indicating why an RPC call failed.

Format

```
void clnt_geterr (CLIENT *clnt, struct rpc_err *errp);
```

Arguments

clnt

Client handle returned by any of the client create routines.

errp

Address of the structure containing information that indicates why an RPC call failed. This information is the same as `clnt_stat` contains, plus one of the following: the C error number, the range of server versions supported, or authentication errors.

Description

This routine is primarily for internal diagnostic use.

Example

```
#define PROGRAM         1
#define VERSION         1

  CLIENT          *clnt;
  struct rpc_err  err;

  clnt = clnt_create( "server name", PROGRAM, VERSION, "udp");

  /* calls to RPC library */

  clnt_geterr( clnt, &err);
```

This example creates a UDP client handle and performs some additional RPC processing. If an RPC call fails, `clnt_geterr` returns the error code.

See Also

`clnt_perror / c`

clnt_pcreateerror / clnt_spcreateerror

XDR ONC Return a message indicating why RPC could not create a client handle.

Format

```
#include

void clnt_pcreateerror (char *s);
char *clnt_spcreateerror (char *s);
```

Argument

s

String containing the message of your choice. The routines append an error message to this string.

Description

The `clnt_pcreateerror` routine prints a message to SYS$OUTPUT.

The `clnt_spcreateerror` routine returns the address of a string. Use this routine if:

- You want to save the string.
- You do not want to use `printf` to print the message.
- The message format is different from the one that `clnt_perrno` supports.

The `clnt_spcreateerror` routine overwrites the string it returns, unless you save the results.

Use these routines when the `clnt_create`, `clntraw_create`, `clnttcp_create`, or `clntudp_create` routine fails.

See Also

`clnt_create, clntraw_create, clnttcp_create, clntudp_create / c`

clnt_perrno / clnt_sperrno

XDR ONC Return a message indicating why the `callrpc` or `clnt_broadcast` routine failed to create a client handle.

Format

```
#include

void clnt_perrno (enum clnt_stat stat);
char *clnt_sperrno (enum clnt_stat stat);
```

Argument

stat

Appropriate error condition. Values for *stat* are defined in the CLNT.H file.

Description

The `clnt_perrno` routine prints a message to SYS$OUTPUT.

The `clnt_sperrno` routine returns the address of a string. Use this routine instead if:

- You want to save the string.
- You do not want to use `printf` to print the message.
- The message format is different from the one that `clnt_perrno` supports.

To save the string, copy it into your own memory space.

See Also

`callrpc, clnt_broadcast`

clnt_perror / clnt_sperror

XDR ONC Return a message if the `clnt_call` routine fails.

Format

```
#include

void clnt_perror (CLIENT *clnt, char *s);
char *clnt_sperror (CLIENT *clnt, char *s);
```

Arguments

clnt, s

See Common Arguments for a description of the above arguments.

Description

Use these routines after `clnt_call`.

The `clnt_perror` routine prints an error message to SYS$OUTPUT.

The `clnt_sperror` routine returns a string. Use this routine if:

- You want to save the string.
- You do not want to use `printf` to print the message.
- The message format is different from the one that `clnt_perror` supports.

The `clnt_sperror` routine overwrites the string with each call. Copy the string into your own memory space if you want to save it.

See Also

`clnt_call, clnt_create, clntraw_create, clnttcp_create, clntudp_create / c`

clntraw_create

XDR Returns an RPC client handle. The remote procedure call uses the IP transport.

Format

```
#include

CLIENT *clntraw_create (struct sockaddr_in *addr,
u_long prognum, u_long versnum, int *sockp, u_long sendsize,
u_long recvsize);
```

Arguments

addr, prognum, versnum

See Common Arguments for a description of the above arguments.

sockp

Socket to be used for this remote procedure call. *sockp* can specify the local address and port number. If *sockp* is RPC_ANYSOCK, then a port number is assigned. The example shown for the `clntudp_create` routine shows how to set up *sockp* to specify a port. See Common Arguments for a description of *sockp* and `RPC_ANYSOCK`.

addr

Internet address of the host on which the server resides.

sendsize

Size of the send buffer. If you enter a value less than 100, then 4000 is used as the default.

recvsize

Size of the receive buffer. If you enter a value less than 100, then 4000 is used as the default.

Description

The `clntraw_create` routine creates an RPC client handle for *addr*, *prognum*, and *versnum*. The client uses the IP transport. The routine is similar to the `clnt_create` routine, except `clnttcp_create` allows you to specify a socket and buffer sizes. If you specify the port number as zero by using *addr*`->sin_port`, the Port Mapper provides the number of the port on which the remote program is listening.

The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space (see also `svcraw_create`). This allows simulation of RPC and getting RPC overheads, such as round trip times, without kernel interference.

The `clnttcp_create` routine uses the global variable `rpc_createerr`, which is a structure that contains the most recent service creation error. Use `rpc_createerr` if you want the client program to handle the error. The value of `rpc_createerr` is set by any RPC client creation routine that does not succeed. The `rpc_createerr` variable is defined in the CLNT.H file.

Diagnostics

The `clntraw_create` routine returns the address of the client handle, or zero (if it could not create the client handle). If the routine fails, use the `clnt_pcreateerror` or `clnt_spcreateerror` routine to obtain additional diagnostic information.

See Also

```
clnt_call, clnt_control, clnt_create, clnt_destroy,
clnt_pcreateerror / c, clnttcp_create,
clntudp_create / c
```

clnttcp_create

XDR ONC Returns an RPC client handle. The remote procedure call uses the TCP transport.

Format

```
#include

CLIENT *clnttcp_create (struct sockaddr_in *addr,
u_long prognum, u_long versnum, int *sockp, u_long sendsize,
u_long recvsize);
```

Arguments

addr, prognum, versnum

See Common Arguments for a description of the above arguments.

sockp

Socket to be used for this remote procedure call. *sockp* can specify the local address and port number. If *sockp* is RPC_ANYSOCK, then a port number is assigned. The example shown for the `clntudp_create` routine shows how to set up *sockp* to specify a port. See Common Arguments for a description of *sockp* and `RPC_ANYSOCK`.

addr

Internet address of the host on which the server resides.

sendsize

Size of the send buffer. If you enter a value less than 100, then 4000 is used as the default.

recvsize

Size of the receive buffer. If you enter a value less than 100, then 4000 is used as the default.

Description

The `clnttcp_create` routine creates an RPC client handle for *addr*, *prognum*, and *versnum*. The client uses the TCP transport. The routine is similar to the `clnt_create` routine, except `clnttcp_create` allows you to specify a socket and buffer sizes. If you specify the port number as zero by using *addr*`->sin_port`, the Port Mapper provides the number of the port on which the remote program is listening.

The `clnttcp_create` routine uses the global variable `rpc_createerr`. `rpc_createerr` is a structure that contains the most recent service creation error. Use `rpc_createerr` if you want the client program to handle the error. The value of `rpc_createerr` is set by any RPC client creation routine that does not succeed. The `rpc_createerr` variable is defined in the CLNT.H file.

Diagnostics

The `clnttcp_create` routine returns the address of the client handle, or zero (if it could not create the client handle). If the routine fails, use the `clnt_pcreateerror` or `clnt_spcreateerror` routine to obtain additional diagnostic information.

See Also

clnt_call, clnt_control, clnt_create, clnt_destroy,
clnt_pcreateerror / c,
clntudp_create / c

clntudp_create / clntudp_bufcreate

XDR ONC Returns an RPC client handle. The remote procedure call uses the UDP transport.

Format

```
#include

CLIENT *clntudp_create (struct sockaddr_in *addr,
u_long prognum, u_long versnum, struct timeval wait,
int *sockp);

CLIENT *clntudp_bufcreate (struct sockaddr_in *addr,
u_long prognum, u_long versnum, struct timeval wait,
int *sockp, u_long sendsize, u_long recvsize);
```

Arguments

addr

Internet address of the host on which the server resides.

prognum, versnum, sockp

See Common Arguments for a description of the above arguments.

wait

Time interval the client waits before resending the call message. This value changes the `CLSET_RETRY_TIMEOUT` code. The `clnt_call` routine uses this value.

sendsize

Size of the send buffer. If you enter a value less than 100, then 4000 is used as the default.

recvsize

Size of the receive buffer. If you enter a value less than 100, then 4000 is used as the default.

Description

These routines create an RPC client handle for *addr*, *prognum*, and *versnum*. The client uses the UDP transport protocol.

If you specify the port number as zero by using *addr*`->sin_port`, the Port Mapper provides the number of the port on which the remote program is listening.

Note! Use the `clntudp_create` routine only for procedures that handle messages shorter than 8K bytes. Use the `clntudp_bufcreate` routine for procedures that handle messages longer than 8K bytes.

The `clntudp_create` routine uses the global variable `rpc_createerr`. `rpc_createerr` is a structure that contains the most recent service creation error. Use `rpc_createerr` if you want the client program to handle the error. The value of `rpc_createerr` is set by any RPC client creation routine that does not succeed.

The `rpc_createerr` variable is defined in the CLNT.H file.

Example

```
main()
{
        int     sock;
        u_long  prog = PROGRAM, vers = VERSION;
        CLIENT  *clnt;
        struct sockaddr_in      local_addr, remote_addr;
        struct timeval  timeout = { 35, 0},
                        retry = { 5, 0};


        remote_addr.sin_family = AF_INET;
        remote_addr.sin_port = 0;  /* consult the remote port mapper */
        remote_addr.sin_addr.s_addr = 0x04030201; /* internet
         addr 1.2.3.4 */


        local_addr.sin_family = AF_INET;
        local_addr.sin_port = 12345;                /* use port 12345 */
        local_addr.sin_addr.s_addr = 0x05030201;    /* internet addr
                                                            1.2.3.5 */


        sock = socket( AF_INET, SOCK_DGRAM, 0);



        /* bind the socket to the local addr */
        bind( sock, &local_addr, sizeof( local_addr));



        /* create a client that uses the local IA and port given above */
        clnt = clntudp_create( &remote_addr, prog, vers, retry, &sock);



        /* use a connection timeout of 35 seconds, not the default */
        clnt_control( clnt, CLSET_TIMEOUT, &timeout);
        /*call the server here*/
}
```

This example defines a socket structure, binds the socket, and creates a UDP client handle.

Diagnostics

These routines return the address of the client handle, or zero (if they cannot create the client handle).

If these routines fail, you can obtain additional diagnostic information by using the `clnt_pcreateerrror` or `clnt_spcreateerror` routine.

See Also

`clnt_call, clnt_control, clnt_create, clnt_destroy, clnt_pcreateerror / c, clnttcp_create`

Chapter 17 ONC RPC RTL Port Mapper Routines

Introduction

This chapter is for RPC programmers. It documents the port mapper routines in the ONC RPC Run-Time Library (RTL). These routines are the programming interface to ONC RPC.

Port Mapper Routines

Port Mapper routines provide a simple callable interface to the Port Mapper. They allow you to request Port Mapper services and information about port mappings. Table 17-1 summarizes the purpose of each Port Mapper routine.

Table 17-1 Port Mapper Routines

Routine	Purpose
pmap_freemaps	Frees memory that was allocated by the `pmap_getmaps` routine.
pmap_getmaps	Returns a list of Port Mappings for the specified host.
pmap_getport	Returns the port number on which a specified service is waiting.
pmap_rmtcall	Requests the Port Mapper on a remote host to call a procedure on that host.
pmap_set	Registers a remote service with a remote port.
pmap_unset	Unregisters a service so it is no longer mapped to a port.

Port Mapper Arguments

Port Mapper routines use many of the same arguments as client routines.

See Table 16-1 in the *ONC RPC RTL Client Routines* chapter for a list of these arguments.

Routine Descriptions

The following sections describe each Port Mapper routine in detail.

pmap_freemaps

ONC Frees memory that was allocated by the `pmap_getmaps` routine.

Format

```
void pmap_freemaps (struct pmaplist *list);
```

Argument

list

Address of a structure containing the list returned by the `pmap_getmaps` routine.

Description

Call the `pmap_freemaps` routine when the list returned by `pmap_getmaps` is no longer needed. Do not call `pmap_freemaps` to free a list that you created.

See Also

`pmap_getmaps`

pmap_getmaps

XDR ONC Returns a list of Port Mappings for the specified host.

Format

```
struct pmaplist *pmap_getmaps (struct sockaddr_in *addr);
```

Argument

addr

Address of a structure containing the internet address of the host whose Port Mapper is being called.

Description

The `pmap_getmaps` routine returns a list of current RPC server-to-Port Mappings on the host at *addr*. The list structure is defined in the PMAP_PROT.H file.

The RPCINFO command uses this routine.

Diagnostics

If an error occurs (for example, `pmap_getmaps` cannot get a list of Port Mappings, the internet address is invalid, or the remote Port Mapper does not exist), the routine returns either NULL or the address of the list.

See Also

`pmap_freemaps, pmap_getport, pmap_set, pmap_unset`

pmap_getport

XDR ONC Returns the port number on which a specified service is waiting.

Format

```
u_short pmap_getport (struct sockaddr_in *addr,
u_long prognum, u_long versnum, u_long protocol);
```

Arguments

addr

Address of a structure containing the internet address of the remote host on which the server resides.

prognum, versnum, protocol

See Table 16-1 in the *ONC RPC RTL Client Routines* chapter for a list of these arguments.

Diagnostics

If the requested mapping does not exist or the routine fails to contact the remote Port Mapper, the routine returns either the port number or zero.

The `pmap_getport` routine uses the global variable `rpc_createerr`. `rpc_createerr` is a structure that contains the most recent service creation error. Use `rpc_createerr` if you want the service program to handle the error. The value of `rpc_createerr` is set by any RPC server creation routine that does not succeed.

The `rpc_createerr` variable is defined in the CLNT.H file.

See Also

`pmap_getmaps, pmap_set, pmap_unset`

pmap_rmtcall

XDR **ONC** Requests the Port Mapper on a remote host to call a procedure on that host.

Format

```
enum clnt_stat pmap_rmtcall (struct sockaddr_in *addr,
u_long prognum, u_long versnum, u_long procnum,
xdrproc_t inproc, u_char *in, xdrproc_t outproc, u_char *out, struct
timeval tout, u_long *portp);
```

Arguments

addr

Address of a structure containing the internet address of the remote host on which the server resides.

prognum, versnum, procnum, inproc, in, outproc, out

See Table 16-1 in the *ONC RPC RTL Client Routines* chapter for a list of these arguments.

tout

Time allowed for the results to return to the client, in seconds and microseconds.

portp

Address where `pmap_rmtcall` will write the port number of the remote service.

Description

The `pmap_rmtcall` routine allows you to get a port number and call a remote procedure in one call. The routine requests a remote Port Mapper to call a *prognum*, *versnum*, and *procnum* on the Port Mapper's host. The remote procedure call uses the UDP transport.

If `pmap_rmtcall` succeeds, it changes *portp* to contain the port number of the remote service.

After calling the `pmap_rmtcall` routine, you may call the `clnt_perrno` routine.

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI_CLNT_RMTCALL.C file provides a sample program using `pmap_rmtcall`.

Diagnostics

This routine returns diagnostic values defined in the CLNT.H file for `enumclnt_stat`.

See Also

`clnt_broadcast`, `clnt_perrno / clnt_sperrno`

pmap_set

XDR **ONC** Registers a remote service with a remote port.

Format

```
bool_t pmap_set (u_long prognum, u_long versnum,
u_long protocol, u_short port);
```

Arguments

prognum, versnum, protocol

See Table 16-1 in the *ONC RPC RTL Client Routines* chapter for a list of these arguments.

port

Remote port number.

Description

The `pmap_set` routine calls the local Port Mapper to tell it which *port* and *protocol* the*prognum*, *versnum* is using.

You are not likely to use `pmap_set`, because `svc_register` calls it.

Diagnostics

The `pmap_set` routine returns TRUE if it succeeds, and FALSE if it fails.

See Also

`pmap_getport, pmap_getmaps, pmap_unset, svc_register`

pmap_unset

XDR **ONC** Unregisters a service so it is no longer mapped it to a port.

Format

```
bool_t pmap_unset (u_long prognum, u_long versnum);
```

Arguments

prognum, versnum

See Table 16-1 in the *ONC RPC RTL Client Routines* chapter for a list of these arguments.

Description

The `pmap_unset` routine calls the local Port Mapper and, for all protocols, removes the *prognum* and *versnum* from the list that maps servers to ports.

You are not likely to use `pmap_unset`, because `svc_unregister` calls it.

Example

The GETSYI_PROC_A.C and GETSYI_SVC.C files provide sample programs using the `pmap_unset` routine.

These files are in the TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC] directory.

Diagnostics

The `pmap_unset` routine returns TRUE if it succeeds, FALSE if it fails.

See Also

`pmap_getport, pmap_getmaps, pmap_set, svc_unregister`

Chapter 18 ONC RPC RTL Server Routines

Introduction

This chapter is for RPC programmers. It documents the server routines in the ONC RPC Run-Time Library (RTL). These routines are the programming interface to ONC RPC.

Server Routines

The server routines are called by the server program or the server stub procedures. Table 18-1 lists each server routine and summarizes its purpose.

Table 18-1 Server Routines

XDR and ONC Routines	Purpose
registerrpc	Performs creation and registration tasks for server.
svc_destroy	Macro that destroys RPC server handle.
svc_freeargs	Macro that frees memory allocated when RPC arguments were decoded.
svc_getargs	Macro that decodes RPC arguments.
svc_getcaller	Macro that returns address of client that called server.
svc_getchan	Macro that returns channel of server handle.
svc_getport	Macro that returns port associated with server handle.
svc_getreqset	Reads data for each server connection.
svc_register	Adds specified server to list of active servers, and registers service program with Port Mapper.
svc_run	Waits for RPC requests and calls `svc_getreqset` routine to dispatch to appropriate RPC service program.
svc_sendreply	Sends results of remote procedure call to client.
svc_unregister	Calls Port Mapper to unregister specified program and version for all protocols.
svcerr_auth	Sends error code when server cannot authenticate client.
svcerr_decode	Sends error code to client if server cannot decode arguments.

svcerr_noproc	Sends error code to client if server cannot implement requested procedure.
svcerr_noprog	Sends error code to client when requested program is not registered with Port Mapper.
svcerr_progvers	Sends error code to client when requested program is registered with Port Mapper, but requested version is not registered.
svcerr_systemerr	Sends error code to client when server encounters error not handled by particular protocol.
svcerr_weakauth	Sends error code to client when server cannot perform remote procedure call because it received insufficient (but correct) authentication parameters.
svcfd_create	Returns address of structure containing server handle for specified TCP socket.
svctcp_create	Returns address of server handle that uses TCP transport.
ONC Routine	**Purpose**
svctcpa_create	Returns address of server handle that uses TCPA transport.
svctcpa_enablecache	Enables XID cache for specified TCPA transport server.
svctcpa_freecache	Deallocates TCPA XID cache.
svctcpa_getxdrs	Returns XDR structure associated with server handle.
svctcpa_shutdown	Cancels all outstanding I/O on channel associated with server handle.
XDR ONC Routine	**Purpose**
svcudp_bufcreate	Returns address of server handle that uses UDP transport. For procedures that pass messages longer than 8Kbytes.
svcudp_create	Returns address of server handle that uses UDP transport. For procedures that pass messages shorter than 8Kbytes.
svcudp_enablecache	Enables XID cache for specified UDP transport server.
ONC Routine	**Purpose**
svcudpa_bufcreate	Returns address of server handle that uses UDPA transport. For procedures that pass messages longer than 8Kbytes.
svcudpa_create	Returns address of server handle that uses UDPA transport. For procedures that pass messages shorter than 8Kbytes.
svcudpa_enablecache	Enables XID cache for specified UDPA transport server.
svcudpa_freecache	Deallocates UDPA XID cache.
svcudpa_getxdrs	Returns XDR structure associated with server handle.
svcudpa_shutdown	Cancels all outstanding I/O on channel associated with the server handle.

XDR ONC Routine	Purpose
xprt_register	Adds UDP or TCP server socket to list of sockets.
xprt_unregister	Removes UDP or TCP server socket from list of sockets.

Routine Descriptions

The following sections describe each server routine in detail.

registerrpc

XDR **ONC** Performs creation and registration tasks for the server.

Format

```
#include
int registerrpc (u_long prognum, u_long versnum, u_long procnum,
u_char *(*procname) (), xdrproc_t inproc, xdrproc_t outproc);
```

Arguments

prognum, versnum, procnum, inproc, outproc

See Table 16-1 in the *ONC RPC RTL Client Routines* chapter for a list of these arguments.

procname

Address of the routine that implements the service procedure. The routine uses the following format:

```
u_char *procname(out);
u_char *out;
```

where *out* is the address of the data decoded by `outproc`.

Description

The `registerrpc` routine performs the following tasks for a server:

- Creates a UDP server handle.
- Calls the `svc_register` routine to register the program with the Port Mapper.
- Adds `prognum`, `versnum`, and `procnum` to an internal list of registered procedures. When the server receives a request, it uses this list to determine which routine to call.

A server should call `registerrpc` for every procedure it implements, except for the NULL procedure.

Example

The GETSYI_SVC_REG.C file provide a sample program using the `registerrpc` routine. This file is in the TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC] directory.

Diagnostics

The `registerrpc` routine returns zero if it succeeds, and -1 if it fails.

See Also

`svc_register`

svc_destroy

ONC Macro that destroys the RPC server handle.

Format

```
void svc_destroy (SVCXPRT *xprt);
```

Argument

xprt

RPC server handle.

Description

The `svc_destroy` routine destroys *xprt* by deallocating private data structures. After this call, *xprt* is undefined.

If the server creation routine received RPC_ANYSOCK as the socket, `svc_destroy` closes the socket. Otherwise, you must close the socket.

Example

The TCPWARE_ROOT[TCPWARE.EXAMPLES.RPC]GETSYI_PROC_A.C file provides a sample program using the `svc_destroy` routine.

See Also

`svcfd_create, svctcp_create, svcudp_create`

svc_freeargs

ONC Macro that frees the memory that was allocated when the RPC arguments were decoded.

Format

```
bool_t svc_freeargs (SVCXPRT *xprt, xdrproc_t xdr_args,
char *args_ptr);
```

Arguments

xprt, xdr_args, args_ptr

See Table 16-1 in the *ONC RPC RTL Client Routines* chapter for a list of these arguments.

Description

The `svc_freeargs` routine calls the `xdr_free` routine.

Example

The GETSYI_PROC_A.C and GETSYI_SVC.C files provide sample programs that use the `svc_freeargs` routine. These files are in the TCPWARE:[TCPWARE.EXAMPLES.RPC] directory.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`svc_getargs, xdr_free`

svc_getargs

ONC Macro that decodes the RPC arguments.

Format

```
bool_t svc_getargs (SVCXPRT *xprt, xdrproc_t xdr_args,
u_char *args_ptr);
```

Arguments

xprt, xdr_args, args_ptr

See Table 16-1 in the *ONC RPC RTL Client Routines* chapter for a list of these arguments.

Example

The GETSYI_PROC_A.C and GETSYI_SVC.C files provide sample programs that use the `svc_getargs` routine.

These files are in the TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC] directory.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`svc_freeargs`

svc_getcaller

ONC Macro that returns the address of the client that called the server.

Format

```
struct sockaddr_in *svc_getcaller (SVCXPRT *xprt);
```

Argument

xprt

RPC server handle.

svc_getchan

ONC Macro that returns the channel associated with the server handle.

Format

```
u_short svc_getchan (SVCXPRT *xprt);
```

Argument

xprt

RPC server handle.

Description

Use `svc_getchan` when multiple servers are listening on the same channel and port.

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI_PROC_A.C file provides a sample program that uses `svc_getchan`.

svc_getport

ONC Macro that returns the port associated with the server handle.

Format

```
u_short svc_getport (SVCXPRT *xprt);
```

Argument

xprt

RPC server handle.

Description

Use `svc_getport` when you want to know what port the server is listening on. You can use this macro with synchronous and asynchronous transports.

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI_PROC_A.C file provides a sample program that uses `svc_getport`.

svc_getreqset

XDR **ONC** Reads data for each server connection.

Format

```
#include

void svc_getreqset (int rdfds);
```

Argument

rdfds

Address of the read socket descriptor array. This array is returned by the `select` routine.

Description

The server calls `svc_getreqset` when it receives an RPC request. The `svc_getreqset` routine reads in data for each server connection, then calls the server program to handle the data.

The `svc_getreqset` routine does not return a value. It finishes executing after all *rdfds* sockets have been serviced.

You are unlikely to call this routine directly, because the `svc_run` routine calls it. However, there are times when you cannot call `svc_run`. For example, suppose a program services RPC requests and reads or writes to another socket at the same time. The program cannot call `svc_run`. It must call `select` and `svc_getreqset`.

The `svc_getreqset` routine is for servers that implement custom asynchronous event processing, do not use the `svc_run` routine.

You may use the global variable `svc_fdset` with `svc_getreqset`. The `svc_fdset` variable lists all sockets the server is using. It contains an array of structures, where each element is a socket pointer and a service handle. It uses the following format:

```
struct sockarr svc_fdset [MAXSOCK +1];
```

This is how to use `svc_fdset`: first, copy the socket handles from `svc_fdset` into a temporary array that ends with a zero. Pass the array to the `select` routine. The `select` routine overwrites the array and returns it. Pass this array to the `svc_getreqset` routine.

You may use `svc_fdset` when the server does not use `svc_run`.

The `svc_fdset` variable is not compatible with UNIX.

Example

```
#define MAXSOCK         10

        int     readfds[ MAXSOCK+1],    /* sockets to select from */
                i, j;

        for( i = 0, j = 0; i < MAXSOCK; i++)
          if( (svc_fdset[i].sockname != 0) && (svc_fdset[i].sockname !=
1))
            readfds[j++] = svc_fdset[i].sockname;
        readfds[j] = 0;                 /* list of sockets ends w/ a zero */
```

```
switch( select( 0, readfds, 0, 0, 0))
{
  case -1:      /* an error happened */
  case 0:       /* time out */
        break;
  default:      /* 1 or more sockets ready for reading */
        errno = 0;
        ONCRPC_SVC_GET_REQSET( readfds);
        if( errno == ENETDOWN || errno == ENOTCONN)
          sys$exit( SS$_THIRDPARTY);
}
```

See Also

svc_run

svc_register

XDR ONC Adds the specified server to a list of active servers, and registers the service program with the Port Mapper.

Format

```
#include

bool_t svc_register (SVCXPRT *xprt, u_long prognum,
u_long versnum, void (*dispatch) (), u_long protocol);
```

Arguments

xprt, prognum, versnum

See Table 16-1 in the *ONC RPC RTL Client Routines* chapter for a list of these arguments.

dispatch

Routine that `svc_register` calls when the server receives a request for *`prognum`*, *`versnum`*. This routine determines which routine to call for each server procedure. This routine uses the following form:

void dispatch(*request, xprt*)

struct svc_req **request*;

SVCXPRT **xprt*;

The `svc_getreqset` and `svc_run` routines call *dispatch*.

protocol

Must be IPPROTO_UDP, IPPROTO_TCP, or zero. Zero indicates that you do not want to register the server with the Port Mapper.

Example

The GETSYI_PROC_A.C and GETSYI_SVC.C files provide sample programs that use the `svc_register` routine.

These files are in the TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC] directory.

Diagnostics

The `svc_register` routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`pmap_set, svc_getreqset, svc_unregister`

svc_run

XDR ONC Waits for RPC requests and calls the `svc_getreqset` routine to dispatch to the appropriate RPC service program.

Format

```
#include

void svc_run()
```

Arguments

None.

Description

The `svc_run` routine calls the `select` routine to wait for RPC requests. When a request arrives, `svc_run` calls the `svc_getreqset` routine. Then `svc_run` calls `select` again.

The `svc_run` routine never returns.

You may use the global variable `svc_fdset` with `svc_run`. See the `svc_getreqset` routine for more information on `svc_fdset`.

Examples

These files contain sample programs that use `svc_run`:

- GETSYI_SVC.C
- GETSYI_SVC_REG.C
- PRINT_SVC.C
- SYSINFO_SVC.C

These files are in the TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC] directory.

See Also

`svc_getreqset`

svc_sendreply / svc_sendreply_dq

Sends the results of a remote procedure call to the client.

Format

XDR ONC

```
#include
bool_t svc_sendreply (SVCXPRT *xprt, xdrproc_t outproc, u_char *out);
```

ONC

```
bool_t svc_sendreply_dq (SVCXPRT *xprt, xdrproc_t outproc, u_char *out);
```

Arguments

XDR ONC

xprt, outproc, out

See Table 16-1 in the *ONC RPC RTL Client Routines* chapter for a list of these arguments.

Description

Both routines send the results of a remote procedure call to the client.

The `svc_sendreply_dq` routine, however, does not queue a read and is for UDPA and TCPA servers only.

Examples

These files contain sample programs that use `svc_sendreply`:

- SYSINFO_SVC.C
- PRINT_SVC.C
- GETSYI_PROC_A.C
- GETSYI_SVC.C

These files are in the TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC] directory.

Diagnostics

These routines returns TRUE if they succeed and FALSE if they fail.

svc_unregister

XDR ONC Calls the Port Mapper to unregister the specified program and version for all protocols. The program and version are removed from the list of active servers.

Format

```
#include

void svc_unregister (u_long prognum, u_long versnum);
```

Arguments

prognum, versnum

See Table 16-1 in the *ONC RPC RTL Client Routines* chapter for a list of these arguments.

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI_PROC_A.C file contains a sample program that uses the `svc_unregister` routine.

See Also

`pmap_unset, svc_register`

svcerr_auth
svcerr_decode
svcerr_noproc
svcerr_noprog
svcerr_progvers
svcerr_systemerr
svcerr_weakauth

XDR **ONC** Sends various error codes to the client process.

Format

```
#include

void svcerr_auth (SVCXPRT *xprt, enum auth_stat why);
void svcerr_decode (SVCXPRT *xprt);
void svcerr_noproc (SVCXPRT *xprt);
void svcerr_noprog (SVCXPRT *xprt);
void svcerr_progvers (SVCXPRT *xprt, u_long low-vers, u_long high-vers);
void svcerr_systemerr (SVCXPRT *xprt);
void svcerr_weakauth (SVCXPRT *xprt);
```

Arguments

xprt

RPC server handle.

why

Error code defined in the AUTH.H file.

low-vers

Lowest version number in the range of versions that the server supports.

high-vers

Highest version in the range of versions that the server supports.

Description

svcerr_auth

See `svc_getreqset`. Calls `svcerr_auth` when it cannot authenticate a client. The `svcerr_auth` routine returns an error code (*why*) to the caller.

svcerr_decode

Sends an error code to the client if the server cannot decode the arguments.

svcerr_noproc

Sends an error code to the client if the server does not implement the requested procedure.

svcerr_noprog

Sends an error code to the client when the requested program is not registered with the Port Mapper. Generally, the Port Mapper informs the client when a server is not registered. Therefore, the server is unlikely to use this routine.

svcerr_progvers

Sends an error code to the client when the requested program is registered with the Port Mapper, but the requested version is not registered.

svcerr_systemerr

Sends an error code to the client when the server encounters an error that is not handled by a particular protocol.

svcerr_weakauth

Sends an error code to the client when the server cannot perform a remote procedure call because it received insufficient (but correct) authentication parameters. This routine calls the `svcerr_auth` routine. The value of *`why`* is AUTH_TOOWEAK, which means "access permission denied."

svcfd_create

XDR **ONC** Returns the address of a structure containing a server handle for the specified TCP socket.

Format

```
#include

SVCXPRT *svcfd_create (int sock, u_long sendsize, u_long recvsize);
```

Arguments

sock

Socket number. Do not specify a file descriptor.

sendsize

Size of the send buffer. If you enter a value less than 100, then 4000 is used as the default.

recvsize

Size of the receive buffer. If you enter a value less than 100, then 4000 is used as the default.

Description

The `svcfd_create` routine returns the address of a server handle for the specified TCP socket. This handle cannot use a file. The server calls the `svcfd_create` routine after it accepts a TCP connection.

Diagnostics

This routine returns zero if it fails.

See Also

`svctcp_create`

svcraw_create

XDR Creates a server handle for memory-based Sun RPC for simple testing and timing.

Format

```
#include

SVCXPRT svcraw_create ();
```

Argument

None.

Description

The `svcraw_create` routine creates a toy Sun RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding client should live in the same address space.

This routine allows simulation of and acquisition of Sun RPC overheads (such as round trip times) without any kernel interference.

Diagnostics

This routine returns NULL if it fails.

See Also

`clntraw_create`

svctcp_create

XDR **ONC** Returns the address of a server handle that uses the TCP transport.

Format

```
#include

SVCXPRT *svctcp_create (int sock, u_long sendsize, u_long recvsize);
```

Arguments

sock

Socket for this service. The `svctcp_create` routine creates a new socket if you enter RPC_ANYSOCK. If the socket is not bound to a TCP port, `svctcp_create` binds it to an arbitrary port.

sendsize

Size of the send buffer. If you enter a value less than 100, then 4000 bytes is used as the default.

recvsize

Size of the receive buffer. If you enter a value less than 100, then 4000 bytes is used as the default.

Examples

The PRINT_SVC.C and GETSYI_SVC.C files provides sample programs that use `svctcp_create`. These files are in the TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC] directory.

Diagnostics

The `svctcp_create` routine returns either the address of the server handle, or zero (if it could not create the server handle).

See Also

`svcfd_create, svc_destroy`

svctcpa_create

ONC Returns the address of a server handle that uses the TCPA transport.

Format

```
SVCXPRT *svctcpa_create (u_short channel, u_short port, u_long sendsize, u_long
recvsize);
```

Arguments

channel

If you enter RPC_ANYCHAN (defined in the SVC.H file), the `svctcpa_create` routine assigns a channel and sets the local port to the *port* value. If you enter any other value, `svctcpa_create` ignores the *port* value.

Multiple server handles may use the same channel if you specify RPC_ANYCHAN and a port number on the first call to `svctcpa_create`, and if you specify the assigned channel with any port number on subsequent calls to `svctcpa_create`. (Use `svc_getchan` to obtain the channel. Note that for TCPA, this is really not a channel, just a unique identifier.)

All server handles that use the same channel should use the same XID cache.

port

Number of the TCP port on which the server will listen. If you enter RPCANYPORT, then RPC assigns a port.

sendsize

Size of the send buffer. If you enter a value less than 100, then 4000 bytes is used as the default.

recvsize

Size of the receive buffer. If you enter a value less than 100, then 4000 bytes is used as the default.

Examples

The GETSYI_PROC_A.C file contains a sample program that uses `svctcpa_create`. This file is in the TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC] directory.

Diagnostics

The `svctcpa_create` routine returns either the address of the server handle, or zero (if it could not create the server handle).

See Also

`svc_destroy, svctcp_create, svctcpa_shutdown, svcudpa_create`

svctcpa_enablecache

ONC Enables the XID cache for the specified TCPA transport server.

Format

```
void *svctcpa_enablecache (SVCXPRT *xprt, void *cacheaddr, u_long size, reply_id
*reqlst);
```

Arguments

xprt

RPC server handle.

cacheaddr

Address of the XID cache. If *cacheaddr* is zero, this routine allocates a cache with *size* number of entries. All TCPA transports that use this cache must have the same size buffers.The first time you call svctcpa_enablecache, specify zero as the *cacheaddr*. The second time you call svctcpa_enablecache, specify the address that svctcpa_enablecache returned on the previous call as the *cacheaddr*. All server handles that use a channel should use the same XID cache.

size

Number of entries in the cache. You may estimate this number based on how active the server is, and on how long you want to retain old replies.

reqlst

Address of an array of structures containing a list of procedures for which replies are to be cached. The array is terminated by prognum==0. This is the structure:

```
typedef struct
{
   u_long prognum,
          versnum,
          procnum;
} reply_id
```

If this address is zero, the server saves all replies in the XID cache.

Description

Call the svctcpa_enablecache routine after each TCPA server handle is created. The server places all appropriate outgoing responses in the XID cache. The cache can be used to improve the performance of the server, for example, by preventing the server from recalculating the results or sending incorrect results. You can disable the cache by calling the svctcpa_freecache routine. The *ONC RPC Fundamentals*, Chapter 12, provides more information on the XID cache.

Diagnostics

The svctcpa_enablecache routine returns either the address of the cache, or zero if an error occurs.

See Also

svctcpa_create, svctcpa_freecache

svctcpa_freecache

ONC Deallocates the TCPA XID cache.

Format

```
void svctcpa_freecache (void *cacheaddr);
```

Argument

cacheaddr

Address of the TCPA XID cache.

Description

The `svc_destroy` routine calls the `svctcpa_freecache` routine for every server handle, after all servers that reference the cache have been destroyed.

The *ONC RPC Fundamentals*, Chapter 12, provides more information on the XID cache.

See Also

`svc_destroy, svctcpa_enablecache`

svctcpa_getxdrs

ONC Returns the XDR structure associated with the server handle.

Format

```
XDRS *svctcpa_getxdrs (SVCXPRT *xprt);
```

Argument

xprt

RPC server handle.

svctcpa_shutdown

ONC Cancels all outstanding I/O on the channel associated with the server handle.

Format

```
void svctcpa_shutdown (SVCXPRT *xprt);
```

Argument

xprt

RPC server handle.

Description

The `svctcpa_shutdown` routine cancels all I/O on the channel associated with *`xprt`* and flags the server as shutting down. The server then begins the shutdown process. Call this routine only once for a channel before calling `svc_destroy` to destroy individual server handles.

This routine affects all TCPA handles that are using the same channel.

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI_PROC_A.C file contains a sample program that uses `svctcpa_shutdown`.

See Also

`svc_destroy, svctcpa_create`

svcudp_create / svcudp_bufcreate

XDR ONC Returns the address of a server handle that uses the UDP transport.

Format

```
#include

SVCXPRT *svcudp_create (int sock);

SVCXPRT *svcudp_bufcreate (int sock, u_long sendsize,
u_long recvsize);
```

Arguments

sock

Socket for this service. The `svcudp_create` routine creates a new socket if you enter RPC_ANYSOCK. If the socket is not bound to a UDP port, the `svcudp_create` routine binds it to an arbitrary port.

sendsize

Size of the send buffer. The minimum size is 100 bytes. The maximum size is 65468, the maximum UDP packet size. If you enter a value less than 100, then 4000 is used as the default.

recvsize

Size of the receive buffer. The minimum size is 100 bytes. The maximum size is 65000, the maximum UDP packet size. If you enter a value less than 100, then 4000 is used as the default.

Description

Use the `svc_create` routine only for procedures that pass messages shorter than 8Kbytes long. Use the `svcudp_bufcreate` routine for procedures that pass messages longer than 8Kbytes.

Examples

The SYSINFO_SVC.C and GETSYI_SVC.C files contain sample programs that use `svcudp_create`. These files are in the TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC] directory.

Diagnostics

These routines return either a server handle, or zero (if they could not create the server handle).

See Also

`svc_destroy, svcudp_enablecache`

svcudp_enablecache

XDR ONC Enables the XID cache for the specified UDP transport server.

Format

```
bool_t svcudp_enablecache (SVCXPRT *xprt, u_long size);
```

Arguments

xprt

RPC server handle.

size

Number of entries permitted in the XID cache. You may estimate this number based on how active the server is, and on how long you want to retain old replies.

Description

Use the `svcudp_enablecache` routine after a UDP server handle is created. The server places all outgoing responses in the XID cache. The cache can be used to improve the performance of the server, for example, by preventing the server from recalculating the results or sending incorrect results.

You cannot disable the XID cache for UDP servers.

The *ONC RPC Fundamentals*, Chapter 12, provides more information on the XID cache.

Example

```
#define FALSE   0
#define UDP_CACHE_SIZE  10

        SVCXPRT *udp_xprt;

        udp_xprt = svcudp_create( RPC_ANYSOCK);
        if( svcudp_enablecache( udp_xprts, UDP_CACHE_SIZE) == FALSE)
          printf( "XID cache was not enabled");
        else
          printf( "XID cache was enabled");
```

Diagnostics

This routine returns TRUE if it enables the XID cache, and FALSE if the cache was previously enabled or an error occurs.

svcudpa_create / svcudpa_bufcreate

ONC Returns the address of a server handle that uses the UDPA transport.

Format

```
SVCXPRT *svcudpa_create (u_short channel, u_short port);

SVCXPRT *svcudpa_bufcreate (u_short channel, u_short port, u_long sendsize, u_long
recvsize);
```

Arguments

channel

If you enter RPC_ANYCHAN (defined in the SVC.H file), then the `svcudpa_bufcreate` routine assigns the channel and sets the local port to the *port* value. If you enter any other value, then `svcudpa_bufcreate` ignores the *port* value. Multiple server handles may use the same channel if you specify RPC_ANYCHAN and a port number on the first call to `svcudpa_create`, and if you specify the assigned channel with any port number on subsequent calls to `svcudpa_create`. All server handles that use the same channel should use the same XID cache.

port

Number of the UDP port on which the server will listen. All servers that use the same port should use the same channel. If you enter RPCANYPORT, then RPC assigns the port.

sendsize

Size of the send buffer. If you enter a value less than 100, then 4000 is used as the default.

recvsize

Size of the receive buffer. If you enter a value less than 100, then 4000 is used as the default.

Description

Both routines return the address of a structure containing a UDPA server handle. The `svcudpa_create` routine limits the call to 8Kbytes of data. The `svcudpa_bufcreate` routine allows you to define the buffer sizes.

See *Using Asynchronous Transports* in Chapter 13, *Building Distributed Applications with RPC*, for more information on writing asynchronous transports.

Example

TCPWARE_ROOT:[TCPWARE.EXAMPLE].RPC]GETSYI_PROC_A.C provides a sample program and procedure that use `svcudpa_create`.

Diagnostics

These routines return the address of the server handle, or zero (if they could not create the server handle).

See Also

`svc_destroy, svcudpa_enablecache, svcudpa_shutdown`

svcudpa_enablecache

ONC Enables the XID cache for the specified UDPA transport server.

Format

```
void *svcudpa_enablecache (SVCXPRT *xprt, void *cacheaddr, u_long size, reply_id
*reqlst);
```

Arguments

xprt

RPC server handle.

cacheaddr

Address of the XID cache. If *cacheaddr* is zero, this routine allocates a cache with *size* number of entries. All UDPA transports that use this cache must have the same size buffers.

The first time you call svcudpa_enablecache, specify zero as the *cacheaddr*. The second time you call svcudpa_enablecache, specify the address that svcudpa_enablecache returned on the previous call as the *cacheaddr*.

All server handles that use a channel should use the same XID cache.

size

Number of entries in the cache. You may estimate this number based on how active the server is, and on how long you want to retain old replies.

reqlst

Address of an array of structures containing a list of procedures for which replies are to be cached. The array is terminated by prognum==0. This is the structure:

```
typedef struct
{
   u_long prognum,
          versnum,
          procnum;
} reply_id
```

If this address is zero, the server saves all replies in the XID cache.

Description

Call the svcudpa_enablecache routine after each UDPA server handle is created. The server places all appropriate outgoing responses in the XID cache.

The cache can be used to improve the performance of the server, for example, by preventing the server from recalculating the results or sending incorrect results. You can disable the cache by calling the svcudpa_freecache routine.

The *ONC RPC Fundamentals*, Chapter 12, provides more information on the XID cache.

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI_PROC_A.C file contains a sample program that uses the `svcudpa_enablecache` routine.

Diagnostics

The `svcudpa_enablecache` routine returns either the address of the cache, or zero if an error occurs.

See Also

`svcudpa_create, svcudpa_freecache`

svcudpa_freecache

ONC Deallocates the UDPA XID cache.

Format

```
void svcudpa_freecache (void *cacheaddr);
```

Argument

cacheaddr

Address of the UDPA XID cache.

Description

The `svc_destroy` routine calls the `svcudpa_freecache` routine for every server handle, after all servers that reference the cache have been destroyed.

The *ONC RPC Fundamentals*, Chapter 12, provides more information on the XID cache.

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI_PROC_A.C file contains a sample program that uses `svcudpa_freecache`.

See Also

`svc_destroy, svcudpa_enablecache`

svcudpa_getxdrs

ONC Returns the XDR structure associated with the server handle.

Format

```
XDRS *svcudpa_getxdrs (SVCXPRT *xprt);
```

Argument

xprt

RPC server handle.

svcudpa_shutdown

ONC Cancels all outstanding I/O on the channel that is associated with the server handle.

Format

```
void svcudpa_shutdown (SVCXPRT *xprt);
```

Argument

xprt

RPC server handle.

Description

The `svcudpa_shutdown` routine cancels all I/O on the channel associated with *`xprt`* and flags the server as shutting down. The server then begins the shutdown process. Call this routine only once for a channel, before calling `svc_destroy` to destroy individual servers.

This routine affects all UDPA handles that are using the same channel.

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI_PROC_A.C file contains a sample program that uses `svcudpa_shutdown`.

See Also

`svc_destroy, svcudpa_create`

xprt_register

XDR ONC Adds a TCP or UDP server socket to a list of sockets.

Format

```
#include

void xprt_register (SVCXPRT *xprt);
```

Argument

xprt
RPC server handle.

Description

The `xprt_register` and `xprt_unregister` routines maintain a list of sockets. This list ensures that the correct server is called to process the request. The `xprt_register` routine adds the server socket to the `svc_fdset` variable, which also stores the server handle that is associated with the socket. The `svc_run` routine passes the list of sockets to the `select` routine. The `select` routine returns to `svc_run`a list of sockets that have outstanding requests.

You are unlikely to call this routine directly because `svc_register` calls it.

See Also

`svc_register, xprt_unregister`

xprt_unregister

XDR ONC Removes a TCP or UDP server socket from a list of sockets.

Format

```
#include

void xprt_unregister (SVCXPRT *xprt);
```

Argument

xprt

RPC server handle.

Description

This list of sockets ensures that the correct server is called to process the request. See the xprt_register routine for a description of how this list is maintained.

You are unlikely to call this routine directly because `svc_unregister` calls it.

See Also

`svc_unregister, xprt_register`

Chapter 19 ONC RPC RTL XDR Routines

Introduction

This chapter is for RPC programmers. It documents the XDR routines in the ONC RPC Run-Time Library (RTL). These routines are the programming interface to ONC RPC.

XDR Routines

This section explains what XDR routines do and when you would call them. It also provides quick reference and detailed reference sections describing each XDR routine.

What XDR Routines Do

Most XDR routines share these characteristics:

- They convert data in two directions: from the host's local data format to XDR format (called encoding or marshalling), or the other way around (called decoding or unmarshalling).
- They use `xdrs`, a structure containing instructions for encoding, decoding, and deallocating memory.
- They return a boolean value to indicate success or failure.

Some XDR routines allocate memory while decoding an argument. To free this memory, call the `xdr_free` routine after the program is done with the decoded value.

Table 19-1 shows the order in which XDR routines perform encoding and decoding.

Table 19-1 XDR Actions

Client	Server
1. Encodes arguments	1. Decodes arguments
2. Decodes results	2. Encodes results
3. Frees results from memory	3. Frees arguments from memory

When to Call XDR Routines

Under most circumstances, you are not likely to call any XDR routines directly. The `clnt_call` and `svc_sendreply` routines call the XDR routines.

You would call the XDR routines directly only when you write your own routines to convert data to or from XDR format.

Quick Reference

Table 19-2 lists the XDR routines that encode and decode data.

Table 19-2 XDR Encoding and Decoding Routines

This routine...	Encodes and decodes...
xdr_array	Variable-length array
xdr_bool	Boolean value
xdr_bytes	Bytes
xdr_char	Character
xdr_double	Double-precision floating point number
xdr_enum	Enumerated type
xdr_float	Floating point value
xdr_hyper	VAX quad word to an XDR hyper-integer, or the other way
xdr_int	Four-byte integer
xdr_long	Longword
xdr_opaque	Contents of a buffer (treats the data as a fixed length of bytes and does not attempt to interpret them)
xdr_pointer	Pointer to a data structure
xdr_reference	Pointer to a data structure (the address must be non-zero)
xdr_short	Two-byte unsigned integer
xdr_string	Null-terminated string
xdr_u_char	Unsigned character
xdr_u_hyper	VAX quad word to an XDR unsigned hyper-integer
xdr_u_int	Four-byte unsigned integer
xdr_u_long	Unsigned longword
xdr_u_short	Two-byte unsigned integer
xdr_union	Union
xdr_vector	Vector (fixed length array)
xdr_void	Nothing
xdr_wrapstring	Null-terminated string

Table 19-3 lists the XDR routines that perform various support functions.

Table 19-3 XDR Support Routines

This routine...	Does this...
xdr_free	Deallocates a data structure from memory
xdrmem_create	Creates a memory buffer XDR stream
xdrrec_create	Creates a record-oriented XDR stream

xdrrec_endofrecord	Marks the end of a record
xdrrec_eof	Goes to the end of the current record, then verifies whether any more data can be read
xdrrec_skiprecord	Goes to the end of the current record
xdrstdio_create	Initializes an `stdio` stream

Table 19-4 lists the upper layer XDR routines that support RPC.

Table 19-4 Upper Layer XDR Routines

This routine...	Encodes and decodes...
xdr_accepted_reply	Part of an RPC reply message after the reply is accepted
xdr_authunix_parms	UNIX-style authentication information
xdr_callhdr	Static part of an RPC request message header (encoding only)
xdr_callmsg	RPC request message
xdr_netobj	Data in the `netobj` structure
xdr_opaque_auth	Authentication information
xdr_pmap	Port Mapper parameters
xdr_pmaplist	List of Port Mapping data
xdr_rejected_reply	Part of an RPC reply message after the reply is rejected
xdr_replymsg	RPC reply header; it then calls the appropriate routine to convert the rest of the message

Routine Descriptions

The following sections describe each XDR routine in detail.

xdr_accepted_reply

XDR ONC Converts an RPC reply message from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_accepted_reply (XDR *xdrs, struct accepted_reply *ar);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

ar

Address of the structure containing the RPC reply message.

Description

The `xdr_replymsg` routine calls the `xdr_accepted_reply` routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_replymsg`

xdr_array

XDR ONC Converts a variable-length array from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_array (XDR *xdrs, u_char **addrp, u_long *sizep, u_long maxsize, u_long
elsize, xdrproc_t elproc);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

addrp

Address of the address containing the array being converted. If *addrp* is zero, then `xdr_array` allocates ((*`*sizep)*elsize`) number of bytes when it decodes.

sizep

Address of the number of elements in the array.

maxsize

Maximum number of elements the array can hold.

elsize

Size of each element, in bytes.

elproc

XDR routine that handles each array element.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_authunix_parms

XDR ONC Converts UNIX-style authentication information from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_authunix_parms (XDR *xdrs, struct authunix_parms *aupp);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

aupp

UNIX-style authentication information being converted.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_bool

XDR ONC Converts a boolean value from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_bool (XDR *xdrs, bool_t *bp);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

bp

Address of the boolean value.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_bytes

XDR ONC Converts bytes from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_bytes (XDR *xdrs, u_char **cpp, u_long *sizep, u_long maxsize);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

cpp

Address of the address of the buffer containing the bytes being converted. If **cpp* is zero, `xdr_bytes` allocates *maxsize* bytes when it decodes.

sizep

Address of the actual number of bytes being converted.

maxsize

Maximum number of bytes that can be used. The server protocol determines this number.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_callhdr

XDR ONC Encodes the static part of an RPC request message header.

Format

```
#include

bool_t xdr_callhdr (XDR *xdrs, struct rpc_msg *chdr);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

chdr

Address of the data being converted.

Description

The `xdr_callhdr` routine converts the following fields: transaction ID, direction, RPC version, server program number, and server version. It converts the last four fields once, when the client handle is created.

The `clnttcp_create` and `clntudp_create` routines call the `xdr_callhdr` routine.

Diagnostics

This routine always returns TRUE.

See Also

`clnt_call`, `clnttcp_create`, `clntudp_create`, `xdr_callmsg`

xdr_callmsg

XDR ONC Converts an RPC request message from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_callmsg (XDR *xdrs, struct rpc_msg *cmsg);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

cmsg

Address of the message being converted.

Description

The `xdr_callmsg` routine converts the following fields: transaction ID, RPC direction, RPC version, program number, version number, procedure number, client authentication.

The `pmap_rmtcall`, `svc_sendreply`, and `svc_sendreply_dq` routines call `xdr_callmsg`.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_callhdr

xdr_char

XDR ONC Converts a character from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_char (XDR *xdrs, char *cp);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

cp

Address of the character being converted.

Description

This routine provides the same functionality as the `xdr_u_char` routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdr_u_char

xdr_double

XDR ONC Converts a double-precision floating point number between local and XDR format.

Format

```
#include

bool_t xdr_double (XDR *xdrs, double *dp);
```

Arguments

xdrs

Pointer to an XDR stream handle created by one of the XDR stream handle creation routines.

dp

Pointer to the double-precision floating point number.

Description

NEW

This routine provides a filter primitive that translates between double-precision numbers and their external representations. It is actually implemented by four XDR routines:

xdr_double_D	Converts VAX D format floating point numbers
xdr_double_G	Converts VAX G format floating point numbers
xdr_double_T	Converts IEEE T format floating point numbers
xdr_double_X	Converts IEEE X format floating point numbers

You can reference these routines explicitly or you can use compiler settings to control which routine is used when you reference the `xdr_double` routine.

ONC

VAX

If you use...	**link your program with...**
VAX C D_float RTL	TCPWARE_RPCLIB_SHR.EXE
VAX C G_float RTL	TCPWARE_RPCLIBG_SHR.EXE.

ALPHA and I64

If you use...	link your program with...
DEC C D_float RTL	TCPWARE_RPCLIBD_SHR.EXE.
DEC C G_float RTL	TCPWARE_RPCLIB_SHR.EXE.
DEC C T_float RTL	TCPWARE_RPCLIBT_SHR.EXE.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_enum

XDR ONC Converts an enumerated type from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_enum (XDR *xdrs, enum_t *ep);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

ep

Address containing the enumerated type.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_float

XDR ONC Converts a floating point value from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_float (XDR *xdrs, float *fp);
```

Arguments

xdrs

Pointer to an XDR stream handle created by one of the XDR stream handle creation routines.

fp

Pointer to a single-precision floating point number.

Description

NEW

This routine provides a filter primitive that translates between double-precision numbers and their external representations. It is actually implemented by four XDR routines:

xdr_float_F	Converts VAX F format floating point numbers
xdr_float_S	Converts IEEE T format floating point numbers

You can reference these routines explicitly or you can use compiler settings to control which routine is used when you reference the `xdr_float` routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_free

XDR ONC Deallocates a data structure from memory.

Format

```
#include

void xdr_free (xdrproc_t proc, u_char *objp);
```

Arguments

proc

XDR routine that describes the data structure.

objp

Address of the data structure.

Description

Call this routine after decoded data is no longer needed. Do not call it for encoded data.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_hyper

XDR ONC Converts a VAX quad word to an XDR hyper-integer, or the other way around.

Format

```
bool_t xdr_hyper (XDR *xdrs, quad *ptr);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

ptr

Address of the structure containing the quad word. The quad word is stored in standard VAX quad word format, with the low-order longword first in memory.

Description

This routine provided the same functionality as the `xdr_u_hyper` routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_u_hyper`

xdr_int

XDR ONC Converts one four-byte integer from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_int (XDR *xdrs, int *ip);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

ip

Address containing the integer.

Description

This routine provides the same functionality as the `xdr_u_int`, `xdr_long`, and `xdr_u_long` routines.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_u_int, xdr_long, xdr_u_long`

xdr_long

XDR ONC Converts one longword from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_long (XDR *xdrs, u_long *lp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

lp

Address containing the longword.

Description

This routine provides the same functionality as the `xdr_u_long`, `xdr_int`, and `xdr_u_int` routines.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_u_long, xdr_int, xdr_u_int`

xdr_netobj

XDR ONC Converts data in the `netobj` structure from the local data format to XDR format, or the other way around.

Format

```
bool_t xdr_netobj (XDR *xdrs, netobj *ptr);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

ptr

Address of the following structure:

```
typedef struct
{
   u_long n_len;
   byte *n_bytes;
} netobj;
```

This structure defines the data being converted.

Description

The `netobj` structure is an aggregate data structure that is opaque and contains a counted array of 1024 bytes.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_opaque

XDR ONC Converts the contents of a buffer from the local data format to XDR format, or the other way around. This routine treats the data as a fixed length of bytes and does not attempt to interpret them.

Format

```
#include

bool_t xdr_opaque (XDR *xdrs, char *cp, u_long cnt);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

cp

Address of the buffer containing opaque data.

cnt

Byte length.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_opaque_auth

XDR ONC Converts authentication information from the local data format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_opaque_auth (XDR *xdrs, struct opaque_auth *ap);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

ap

Address of the authentication information. This data was created by the `authnone_create`, `authunix_create`, or `authunix_create_default` routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_pmap

XDR ONC Converts Port Mapper parameters from the local data format to XDR format, or the other way around.

Format

```
#include "TCPWARE_INCLUDE:PMAP_PROT.H"

bool_t xdr_pmap (XDR *xdrs, struct pmap *regs);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

regs

Address of a structure containing the program number, version number, protocol number, and port number. This is the data being converted.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_pmaplist

XDR ONC Converts a list of Port Mapping data from the local data format to XDR format, or the other way around.

Format

```
#include "TCPWARE_INCLUDE:PMAP_PROT.H"

bool_t xdr_pmaplist (XDR *xdrs, struct pmaplist **rpp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

rpp

Address of the address of the structure containing Port Mapper data. If this routine is used to decode a Port Mapper listing, *rpp* is set to the address of the newly allocated linked list of structures.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_pointer

XDR ONC Converts a recursive data structure from the local data format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_pointer (XDR *xdrs, u_char **objpp, u_long obj_size,
xdrproc_t xdr_obj);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

objpp

Address of the address containing the data being converted. May be zero.

obj_size

Size of the data structure in bytes.

xdr_obj

XDR routine that describes the object being pointed to. This routine can describe complex data structures, and these structures may contain pointers.

Description

An XDR routine for a data structure that contains pointers to other structures, such as a linked list, would call the `xdr_pointer` routine. The `xdr_pointer` routine encodes a pointer from an address into a boolean. If the boolean is TRUE, the data follows the boolean.

Example

```
bool_t xdr_pointer( xdrs, objpp, obj_size, xdr_obj)
  XDR           *xdrs;
  char          **objpp;
  longw         obj_size;
  xdrproc_t     xdr_obj;
{
        bool_t  more_data;

/*
** determine if the pointer is a valid address (0 is invalid)
*/
        if( *objpp != NULL)
          more_data = TRUE;
        else
          more_data = FALSE;
/*
** XDR the flag

** If we are decoding, then more_data is overwritten.
*/
```

```
        if( !xdr_bool( xdrs, &more_data))
          return( FALSE);
/*
** If there is no more data, set the pointer to 0 (No effect if we
** were encoding) and return TRUE
*/
        if( !more_data)
        {
          *objpp = NULL;
          return( TRUE);
        }
/*
** Otherwise, call xdr_reference.  The result is that xdr_pointer is
** the same as xdr_reference, except that xdr_pointer adds a Boolean
** to the encoded data and will properly handle NULL pointers.
*/
        return( xdr_reference( xdrs, objpp, obj_size, xdr_obj));
} /* end function xdr_pointer() */
```

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_reference

XDR ONC This routine recursively converts a structure that is referenced by a pointer inside the structure.

Format

```
#include

bool_t xdr_reference (XDR *xdrs, u_char **objpp, u_long obj_size, xdrproc_t
xdr_obj);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

objpp

Address of the address of a structure containing the data being converted. If *objpp* is zero, the `xdr_reference` routine allocates the necessary storage when decoding. This argument must be non-zero when encoding.

When `xdr_reference` encodes data, it passes `*objpp` to `xdr_obj`. When decoding, `xdr_reference` allocates memory if `*objpp` equals zero.

obj_size

Size of the referenced structure.

xdr_obj

XDR routine that describes the object being pointed to. This routine can describe complex data structures, and these structures may contain pointers.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_rejected_reply

XDR ONC Converts the remainder of an RPC reply message after the header indicates that the reply is rejected.

Format

```
#include

bool_t xdr_rejected_reply (XDR *xdrs, struct rejected_reply *rr);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

rr

Address of the structure containing the reply message.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_replymsg

XDR ONC Converts the RPC reply header, then calls the appropriate routine to convert the rest of the message.

Format

```
#include

bool_t xdr_replymsg (XDR *xdrs, struct rpc_msg *rmsg);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

rmsg

Address of the structure containing the reply message.

Description

The `xdr_replymsg` routine calls the `xdr_rejected_reply` or `xdr_accepted_reply` routine to convert the body of the RPC reply message from the local data format to XDR format, or the other way around.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_accepted_reply, xdr_rejected_reply`

xdr_short

XDR ONC Converts a two-byte integer from the local data format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_short (XDR *xdrs, short *sp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

sp

Address of the integer being converted.

Description

This routine provides the same functionality as `xdr_u_short`.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_u_short`

xdr_string

XDR ONC Converts a null-terminated string from the local data format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_string (XDR *xdrs, char **cpp, u_long maxsize);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

cpp

Address of the address of the first byte in the string.

maxsize

Maximum length of the string. The service protocol determines this value.

Description

The `xdr_string` routine is the same as the `xdr_wrapstring` routine, except `xdr_string` allows you to specify the *`maxsize`*.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_wrapstring`

xdr_u_char

XDR ONC Converts an unsigned character from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_u_char (XDR *xdrs, u_char bp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

bp

Address of the character being converted.

Description

This routine provides the same functionality as `xdr_char`.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_char`

xdr_u_hyper

XDR ONC Converts an VAX quad word to an XDR unsigned hyper-integer, or the other way around.

Format

```
bool_t xdr_u_hyper (XDR *xdrs, quad *ptr);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

ptr

Address of the structure containing the quad word. The quad word is stored in standard VAX format, with the low-order longword first in memory.

Description

This routine provides the same functionality as the `xdr_hyper` routine.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_hyper`

xdr_u_int

XDR ONC Converts a four-byte unsigned integer from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_u_int (XDR *xdrs, int *ip);
```

Arguments

xdrs

Address of a structure containing XDR encoding and decoding information.

ip

Address of the integer.

Description

This routine provides the same functionality as `xdr_int`, `xdr_long`, and `xdr_u_long`.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_int`

xdr_u_long

XDR ONC Converts an unsigned longword from local format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_u_long (XDR *xdrs, u_long *lp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

lp

Address of the longword.

Description

This routine provides the same functionality as `xdr_long`, `xdr_int`, and `xdr_u_int`.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_long, xdr_int, xdr_u_int`

xdr_u_short

XDR ONC Converts a two-byte unsigned integer from the local data format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_u_short (XDR *xdrs, u_short *sp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

sp

Address of the integer being converted.

Description

This routine provides the same functionality as `xdr_short`.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_short`

xdr_union

XDR ONC Converts a union from the local data format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_union (XDR *xdrs, enum_t *dscmp, u_char *unp, xdr_discrim *choices,
xdrproc_t dfault);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

dscmp

Integer from the *choices* array.

unp

Address of the union.

choices

Address of an array. This array maps integers to XDR routines.

dfault

XDR routine that is called if the `dscmp` integer is not in the *choices* array.

Description

The `xdr_union` routine searches the array choices for the value of *dscmp*. If it finds the value, it calls the corresponding XDR routine to process the remaining data.If `xdr_union` doesn't find the value, it calls the `dfault` routine.

Example

The TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI_XDR_2.C file contains a sample routine that calls `xdr_union`.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_vector

XDR ONC Converts a vector (fixed length array) from the local data format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_vector (XDR *xdrs, u_char *basep, u_long nelem, u_long elmsize,
xdrproc_t xdr_elem);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

basep

Address of the array.

nelem

Number of elements in the array.

elmsize

Size of each element.

xdr_elem

Converts each element from the local data format to XDR format, or the other way around.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

xdr_void

XDR ONC Converts nothing.

Format

```
#include

bool_t xdr_void (XDR *xdrs, u_char *ptr);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

ptr

Ignored.

Description

Use this routine as a place-holder for a program that passes no data. The server and client expect an XDR routine to be called, even when there is no data to pass.

Diagnostics

This routine always returns TRUE.

xdr_wrapstring

XDR ONC Converts a null-terminated string from the local data format to XDR format, or the other way around.

Format

```
#include

bool_t xdr_wrapstring (XDR *xdrs, char **cpp);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

cpp

Address of the address of the first byte in the string.

Description

The `xdr_wrapstring` routine calls the `xdr_string` routine. The `xdr_wrapstring` routine hides the *`maxsize`* argument from the programmer. Instead, the maximum size of the string is assumed to be 2^{32} - 1.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

`xdr_string`

xdrmem_create

XDR ONC Creates a memory buffer XDR stream.

Format

```
#include

void xdrmem_create (XDR *xdrs, u_char *addr, u_long size, enum xdr_op op);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

addr

Address of the buffer containing the encoded data.

size

Size of the *addr* buffer.

op

Operations you will perform on the buffer. Valid values are `XDR_ENCODE`, `XDR_DECODE`, and `XDR_FREE`. You may change this value.

Description

The `xdrmem_create` routine initializes a structure so that other XDR routines can write to a buffer. The UDP and UDPA transports use this routine.

xdrrec_create

XDR ONC Creates a record-oriented XDR stream.

Format

```
#include

void xdrrec_create (XDR *xdrs, u_long sendsize, u_long recvsize,
u_char *tcp_handle, int (*readit)(), int (*writeit)();
```

Arguments

xdrs

Address of the structure being created. The `xdrrec_create` routine will write XDR encoding and decoding information to this structure.

sendsize

Size of the send buffer in bytes. The minimum size is 100 bytes. If you specify fewer than 100 bytes, 4000 bytes is used as the default.

recvsize

Size of the receive buffer in bytes. The minimum size is 100 bytes. If you specify fewer than 100 bytes, 4000 bytes is used as the default.

tcp_handle

Address of the client or server handle.

readit

Address of a user-written routine that reads data from the stream transport. This routine must use the following format:

int readit(tcp_handle, buffer, len)

u_char **tcp_handle*;

u_char **buffer*;

u_long *len*;

where `*tcp_handle` is the client or server handle, `*buffer` is the buffer to fill, and `len` is the number of bytes to read. The `readit` routine returns either the number of bytes read, or -1 if an error occurs.

writeit

Address of a user-written routine that writes data to the stream transport. This routine must use the following format:

```
int writeit(tcp_handle, buffer, len)
u_char *tcp_handle;
u_char *buffer;
u_long len;
```

– `tcp_handle` is the client or server handle.

– `buffer` is the address of the buffer being written.

– *len* is the number of bytes to write.

The *writeit* routine returns either the number of bytes written, or -1 if an error occurs.

Description

The xdrrec_create routine requires one of the following:

- The TCP transport
- A stream-oriented interface (such as file I/O) not supported by TCPware. The stream consists of data organized into records. Each record is either an RPC request or reply.

The clnttcp_create and svcfd_create routines call the xdrrec_create routine.

See Also

clnttcp_create, svcfd_create, xdrrec_endofrecord, xdrrec_eof, xdrrec_skiprecord

xdrrec_endofrecord

XDR **ONC** Marks the end of a record.

Format

```
#include

bool_t xdrrec_endofrecord (XDR *xdrs, bool_t sendnow);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

sendnow

Indicates when the calling program will send the record to the *writeit* routine (see xdrrec_create).

If *sendnow* is TRUE, xdrrec_endofrecord sends the record now. If *sendnow* is FALSE, xdrrec_endofrecord writes the record to a buffer and sends the buffer when it runs out of buffer space.

Description

A client or server program calls the xdrrec_endofrecord routine when it reaches the end of a record it is writing. The program must call the xdrrec_create routine before calling xdrrec_endofrecord.

Diagnostics

This routine returns TRUE if it succeeds and FALSE if it fails.

See Also

xdrrec_create, xdrrec_eof, xdrrec_skiprecord

xdrrec_eof

XDR ONC Goes to the end of the current record, then verifies whether any more data can be read.

Format

```
#include

bool_t xdrrec_eof (XDR *xdrs);
```

Argument

xdrs

Address of the structure containing XDR encoding and decoding information.

Description

The client or server program must call the `xdrrec_create` routine before calling `xdrrec_eof`.

Diagnostics

This routine returns TRUE if it reaches the end of the data stream, and FALSE if it finds more data to read.

See Also

`xdrrec_create, xdrrec_endofrecord, xdrrec_skiprecord`

xdrrec_skiprecord

XDR ONC Goes to the end of the current record.

Format

```
#include

bool_t xdrrec_skiprecord (XDR *xdrs);
```

Argument

xdrs

Address of the structure containing XDR encoding and decoding information.

Description

A client or server program calls the `xdrrec_skiprecord` routine before it reads data from a stream. This routine ensures that the program starts reading a record from the beginning.

The `xdrrec_skiprecord` routine is similar to the `xdrrec_eof` routine, except that `xdrrec_skiprecord` does not verify whether any more data can be read.

The client or server program must call the `xdrrec_create` routine before calling `xdrrec_skiprecord`.

Diagnostics

This routine returns TRUE if it has skipped to the start of a record. Otherwise, it returns FALSE.

See Also

`xdrrec_create, xdrrec_endofrecord, xdrrec_eof`

xdrstdio_create

XDR Initializes an `stdio` XDR stream.

Format

```
#include

void xdrstdio_create (XDR *xdrs, FILE *file, enum xdr_op op);
```

Arguments

xdrs

Address of the structure containing XDR encoding and decoding information.

file

File pointer `FILE *`, which is to be associated with the stream.

op

An XDR operation, one of: `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`.

Description

The `xdrstdio_create` routine initializes an stdio stream for the specified file.

Chapter 20 ONC RPC Sample Programs

Introduction

This chapter is for RPC programmers. It explains:

- How to run the sample programs.
- The purpose of each sample program file.

Introducing Sample Programs

The ONC RPC sample programs are divided into three groups:

- GETSYI
- PRINT
- SYSINFO

Each group has a command procedure to facilitate compiling and linking. All command procedures are in the TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC] directory.

For more information on each program, see the comments in the GETSYI.COM, PRINT.COM, and SYSINFO.COM files.

Running Sample Programs

To run an RPC sample program, follow these steps:

1 Move to the directory where you want the command procedure to place the client and server sample programs. It must be a directory in which you have write privilege.

2 Compile the sample client and server programs by entering one of the following commands at the DCL prompt:

$ **@TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]PRINT**
$ **@TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]SYSINFO**
$ **@TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI XDR1**
$ **@TCPWARE_ROOT:[TCPWARE.EXAMPLES.RPC]GETSYI XDR2**

3 Verify whether the Port Mapper is running. Use the RPCINFO command or the NETCU SHOW SERVICES command. If the Port Mapper is not running, ask the system manager to start it.

4 Log on to a second terminal. Run the server that you compiled in step 2. To stop the server, type CTRL/C.

5 Define symbols that point to the executable client program. A symbol can point to either a logical or a disk/directory specification, as follows:

symbol:==$*logical*:*prog*
symbol:==$*disk*:[*dir*]*prog*

- *symbol* specifies GETSYI, PRINTF, or SYSINFO.
- *logical* is the logical pointing to the directory that contains the .EXE client program. This logical must translate to a disk and directory specification.
- *prog* is the .EXE program for GETSYI, PRINT, or SYSINFO.
- *disk*:[*dir*] is the disk and directory where the .EXE client program is located.

These symbols remain valid until you log off.

6 Run the clients. Use the commands given in the following sections.

Running GETSYI Client

To run the GETSYI client, enter the following command at the DCL prompt:

```
$ GETSYI [-u | -t][-d#][-nnode][-hhost]code[code...]
```

GETSYI	Runs the GETSYI client.
-u	UDP transport. Supported only for the GETSYI_CLNT_CALL client. This is the default.
-t	TCP transport. Supported only for the GETSYI_CLNT_CALL client.
-d#	Debugging value passed to the ONCRPC_SET_CHAR routine. Debugging values are defined in the ONCRPC_CONST.H file. The pound sign (#) represents a hexadecimal number. The default is 0.
-n*node*	Name of the node about which the server is returning information. Can be used only if the server is part of a VMS cluster. The default is a NULL string (no node).
-h*host*	Domain name of the host running the remote server. If you omit *host*, *node* is used as the default. If *node* is not specified, the default is the local host.
code	String indicating what information the server needs to return. You can enter up to 64 codes. The codes are – BOOTTIME NODE_SWVERS – CLUSTER_NODES NODENAME – CPU PAGEFILE_FREE – NODE_HWTYPE PAGEFILE_PAGE – NODE_HWVERS SWAPFILE_FREE – NODE_NUMBER SWAPFILE_PAGE – NODE_SWTYPE VERSION

	The GETSYI_SVC_SUBS.C file defines all possible codes. Three codes in the file are specific to VMS Version 5.*x* or OpenVMS. These are commented out. You can remove the comments in order to use them.
	The *OpenVMS System Services Reference Manual* describes each code under the $ GETSYI system service.

Running PRINT Client

To run the PRINT client, type the following command at the DCL prompt:

PRINTF *[***-d#***][***-h*host*]****filename***

-d#	Debugging value that is passed to the ONCRPC_SET_CHAR routine. Debugging values are defined in the ONCRPC_CONST.H file. The pound sign (#) represents a hexadecimal number. The default is 0.
-h*host*	Domain name of the host running the remote server. The default is the local host.
filename	Name of the file to be displayed on the screen. You must have the appropriate privilege to gain access to this file.

Running SYSINFO Client

To run the SYSINFO client, type the following command at the DCL prompt:

SYSINFO *[***-d#***][****limit****]*

-d#	Debugging value that is passed to the ONCRPC_SET_CHAR routine. Debugging values are defined in the ONCRPC_CONST.H file. The pound sign (#) represents a hexadecimal number. The default is 0.
limit	Maximum number of replies the client can receive before it exits. The client processes until it reaches this limit or it times out, whichever comes first.

Miscellaneous Clients and Servers

The sample programs listed in Table 20-1 include three client programs using the `client_call`, `callrpc`, and `pmap_rmtcall` routines; a synchronous server; and a server using the `registerrpc` routine.

In this group of programs, a client calls a server to request system information, and the server provides the information.

Table 20-1 GETSYI Sample Programs

File	Description
GETSYI.COM	Command procedure that compiles and links related RPC sample programs.
GETSYI.H	Header file used by the sample RPC GETSYI programs. Generated by RPCGEN.
GETSYI.X	Sample RPCGEN input file for the TCP and UDP transports.
GETSYI_CLNT.C	Sample client code generated by the RPCGEN compiler.
GETSYI_CLNT_CALL.C	Sample RPC client program using the `clnt_call` routine.
GETSYI_CLNT_CALLRPC.C	Sample RPC client program using the `callrpc` routine.
GETSYI_CLNT_RMTCALL.C	Sample RPC client program using the `pmap_rmtcall` routine.
GETSYI_CLNT_SUBS.C	Miscellaneous client support routines.
GETSYI_DEF.H	Contains various structure definitions used by the RPC sample programs.
GETSYI_PROC_S.C	Sample RPC procedure for a synchronous server.
GETSYI_SVC.C	Sample RPC program for a synchronous server. Generated by RPCGEN.
GETSYI_SVC_REG.C	Sample RPC program and procedure using the `registerrpc` routine.
GETSYI_SVC_SUBS.C	Miscellaneous server support routines for the sample GETSYI programs.
GETSYI_XDR.C	XDR routines generated by RPCGEN, used by sample RPC clients and servers.
GETSYI_XDR_2.C	Alternative XDR routines using `xdr_union`, used by sample RPC clients and servers.

Batch RPC Sample Programs

In the batch programs listed in Table 20-2, a client sends a file to a server one record at a time. After the last record is sent, the client flushes the pipeline and receives the total number of records received by the server.

Table 20-2 Sample RPC Batch Programs

File	Description
PRINT.C	Sample RPC batch client program.
PRINT.COM	Command procedure that compiles and links related RPC sample programs.
PRINT.H	Header file for the sample RPC batch programs. Generated by RPCGEN.

PRINT.X	RPCGEN input file used by the sample RPC batch programs.
PRINT_DEF.H	Contains various structure definitions used by the RPC sample batch programs.
PRINT_SVC.C	Sample RPC batch server program.
PRINT_XDR.C	XDR routines used by the sample RPC batch programs. Generated by RPCGEN.

Broadcast RPC Sample Programs

In the broadcast RPC programs listed in Table 20-3, the server returns the name of its host and operating system. The `each result` routine displays the address of the remote host and the values returned.

The `clnt_broadcast` routine waits for replies until it times out or reaches the specified limit. Then the client stops processing.

Table 20-3 Broadcast RPC Sample Programs

File	Description
SYSINFO.C	Sample RPC client program using broadcast RPC.
SYSINFO.COM	Command procedure that compiles and links related RPC sample programs.
SYSINFO.H	Header file used by the sample RPC broadcast programs. Generated by RPCGEN.
SYSINFO.X	RPCGEN input file used by the sample RPC broadcast programs.
SYSINFO_DEF.H	Contains various structure definitions used by the sample RPC broadcast programs.
SYSINFO_SVC.C	Sample RPC server program using broadcast RPC.
SYSINFO_XDR.C	XDR routines used by the sample RPC broadcast programs. Generated by RPCGEN.

Appendix A TCPware Socket Library

Introduction

This appendix describes the TCPware for OpenVMS Socket Library.

Note! This appendix is for use with versions of VMS earlier than 5.3. For later versions, use the VAX C or DEC C socket libraries described in *Chapter 8, Socket Library*.

The Socket Library is a collection of VAX C (on VAX machines) and DEC C (on Alpha and I64 machines) subroutines that closely emulates the UNIX socket functions. The Socket Library supports a subset of these functions, including stream and datagram sockets.

ALPHA and I64 When using DEC C, be sure to use the `/stand=vaxc` compiler option.

TCPware provides these subroutines so that UNIX C programs, which use the UNIX socket functions, can easily be migrated to the TCPware environment. Programs written in VAX C, DEC C, or other high level languages can call the Socket Library.

TCPware does not support all features of the UNIX socket functions. It provides only limited asynchronous support. Most differences are primarily due to the differences between the UNIX and OpenVMS operating systems.

When developing new applications, consider using a QIO programming interface. This interface is not difficult to use and provides full capabilities for event-driven programming.

For more information on the direct QIO programming interfaces, see Chapter 6, *INETDRIVER Services*, Chapter 3, *TCPDRIVER Services*, Chapter 4, *UDPDRIVER Services*, or Chapter 5, *IPDRIVER Services*.

The Socket Library and related VAX C header files are located in the TCPWARE_INCLUDE: directory.

Include (Header) Files

The following VAX C header files are provided with the Socket Library:

IF.H	Defines the interface data structures used with the `socket_ioctl` routine to obtain information about network interfaces.
IF_ARP.H	Defines the ARP (Address Resolution Protocol) data structures used with the `socket_ioctl` routine to set and get ARP entries.
IN.H	Defines the internet address structure (`in_addr`) and socket address structure (`sockaddr_in`). This file is required for almost all socket operations.

INET.H	Defines the `inet` subroutines. Use this file when you need to call them.

ALPHA and I64 Use the INET.H file when calling the TCPware `inet` subroutines. If you do not, the program uses HP subroutines instead. See Subroutines Redefined in Header Files for Use on Alpha and I64 Systems for a list of subroutines that are redefined in this file.

IOCTL.H	Defines the `socket_ioctl` subroutine.
NAMESER.H	Defines the constant needed for using the `resolver` routines for Domain Name Services (DNS) lookups.
NETDB.H	Defines the network database structures; in particular, the `hostent`, `protoent`, and `servent` structures. It also declares the subroutines. This file is required if using the `gethostbyname`, `gethostbyaddr`, `getprotobyname`, `getprotobynumber`, `getservbyname`, and `getservbyport` subroutines.

ALPHA and I64 Use the NETDB.H file when calling the TCPware subroutines above. If you do not, the program uses HP subroutines instead. See Subroutines Redefined in Header Files for Use on Alpha and I64 Systems for a list of subroutines that are redefined in this file.

RESOLV.H	Defines the options provided for the `resolver` routines for DNS lookups.
ROUTE.H	Defines the routing data structures used with the `socket_ioctl` routine to set and get routing table entries.
SOCKERR.H	Defines the socket error codes (these are usually defined in ERRNO.H). This file is required for all socket operations if you want to test for specific error status codes.
SOCKET.H	Defines the `sockaddr` structure and the SOCK_STREAM, SOCK_DGRAM, AF_INET, and various other symbols used when calling the Socket Library subroutines. This file is required for all socket operations.

ALPHA and I64 Use the SOCKET.H file when calling the TCPware subroutines. If you do not, the program uses HP subroutines instead. See Subroutines Redefined in Header Files for Use on Alpha and I64 Systems for a list of subroutines that are redefined in this file.

SOCKETVAR.H	Defines the socket structure used by the socket subroutines themselves. It is only needed if you want to get access to this internal structure for special purposes (such as reading the channel number). It is not recommended that you use the SOCKETVAR.H header file as it may be removed in a future TCPware release. Use the `getsockopt` subroutine, with SO_IOCHAN as *optname*, if you need the OpenVMS I/O channel a socket uses.

TYPES.H	Defines many UNIX C data type names. It is useful when adapting UNIX C programs to work under VAX C. The TYPES.H file also defines macros used in manipulating the socket descriptor sets used with `select`, such as FD_SET, FD_CLR, FD_ISSET, FD_ZERO, FD_SET, and FD_SETSIZE.

ALPHA and I64 Use the TYPES.H file when calling certain TCPware subroutines. If you do not, the program uses HP subroutines instead. See Subroutines Redefined in Header Files for Use on Alpha and I64 Systems for a list of subroutines that are redefined in this file.

The INET.H, NETDB.H, SOCKET.H, and TYPES.H header files are included with TCPware. These header files contain the subroutine and function name re-definitions in Table A-1.

Table A-1 Subroutines Redefined in Header Files for Use on Alpha and I64 Systems

This Subroutine/Function...	Is Redefined As...
File: SOCKET.H	
accept	tcpware_accept
bind	tcpware_bind
connect	tcpware_connect
getpeername	tcpware_getpeername
getsockname	tcpware_getsockname
getsockopt	tcpware_getsockopt
listen	tcpware_listen
recvfrom	tcpware_recvfrom
select	tcpware_select
sendto	tcpware_sendto
setsockopt	tcpware_setsockopt
socket	tcpware_socket
socket_close	tcpware_socket_close
socket_ioctl	tcpware_socket_ioctl
socket_read	tcpware_socket_read
socket_recv	tcpware_socket_recv
socket_send	tcpware_socket_send
socket_write	tcpware_socket_write
shutdown	tcpware_shutdown
getdomainname	tcpware_getdomainname
setdomainname	tcpware_setdomainname
gethostid	tcpware_gethostid

gethostaddr	tcpware_gethostaddr
gethostname	tcpware_gethostname
sethostname	tcpware_sethostname
pneterror	tcpware_pneterror
File: TYPES.H	
htonl	tcpware_htonl
htons	tcpware_htons
ntohl	tcpware_ntohl
ntohs	tcpware_ntohs
File: INET.H	
inet_addr	tcpware_inet_addr
inet_aton	tcpware_inet_aton
inet_lnaof	tcpware_inet_lnaof
inet_makeaddr	tcpware_inet_makeaddr
inet_ntoa	tcpware_inet_ntoa
inet_netof	tcpware_inet_netof
inet_network	tcpware_inet_network
File: NETDB.H	
gethostbyname	tcpware_gethostbyname
gethostbyaddr	tcpware_gethostbyaddr
getnetent	tcpware_getnetent
getnetbyaddr	tcpware_getnetbyaddr
getnetbyname	tcpware_getnetbyname
getprotobyname	tcpware_getprotobyname
getprotobynumber	tcpware_getprotobynumber
getprotoent	tcpware_getprotoent
getservbyname	tcpware_getservbyname
getservbyport	tcpware_getservbyport

Linking Applications

To use the Socket Library from your applications, you must link to the Socket Library shareable image and the VAX C Run-Time Library (RTL). On Alpha and I64 systems, HP supplied RTLs link in automatically.

VAX For example, the following command on the VAX links a program called TEST with the Socket Library shareable image and the VAX C RTL:

```
$ LINK TEST,SYS$INPUT/OPTIONS
           SYS$SHARE:TCPWARE_SOCKLIB_SHR/SHARE
           SYS$SHARE:VAXCTRL/SHARE
Ctrl/Z
```

Although it is recommended that you use TCPWARE_SOCKLIB_SHR.EXE or TCPWARE_SOCKLIBG_SHR.EXE, you can link with TCPWARE:SOCKLIB.OLB instead.

The following command links the TEST program with SOCKLIB.OLB and the VAX C RTL:

```
$ LINK TEST,TCPWARE:SOCKLIB/LIB,SYS$INPUT/OPTIONS
           SYS$SHARE:VAXCRTL/SHARE
Ctrl/Z
```

For details on linking C programs, see the VAX C documentation.

ALPHA and I64 The command is:

```
$ LINK TEST,SYS$INPUT/OPTIONS
           SYS$LIBRARY:TCPWARE_SOCKLIB_SHR/SHARE
Ctrl/Z
```

Although it is recommended that you use TCPWARE_SOCKLIB_SHR.EXE, TCPWARE_SOCKLIBD_SHR.EXE, or TCPWARE_SOCKLIBT_SHR.EXE, you can link with TCPWARE:SOCKLIB.OLB instead.

The following command links the TEST program with SOCKLIB.OLB and the DEC C RTL:

```
$ LINK TEST,TCPWARE:SOCKLIB/LIB
```

On Alpha and I64 systems, HP supplied RTL's link in automatically.

For details on linking C programs, see the HP C documentation.

Sample Programs

Table A-2 lists the sample Socket Library based client and server programs TCPware provides in the SYS$COMMON:[TCPWARE.EXAMPLES]:.

Table A-2 Sample Programs

Program	Purpose
DAYTIMED.C	Server for the DAYTIME protocol
DISCARD.C	Client for the DISCARD protocol
DISCARDD.C	Server for the DISCARD protocol
FINGER.C	Client for the FINGER protocol
FINGERD.C	Server for the FINGER protocol
WHOIS.C	Client for the NAME protocol

Subroutine Categories

This section lists the subroutines provided in the Socket Library. These subroutines are divided into the following categories:

Socket operations	Lookup operations
Byte-order conversion operations	Byte string operations
Internet address conversion	Server operation

See the *Socket Library Reference* section for details on each one of the subroutines and functions associated with these categories.

Socket Operations

The Socket Library contains the following socket operations:

accept	pneterror	socket_read
bind	select	socket_recv
connect	setsockopt	recvfrom
getpeername	shutdown	socket_send
getsockname	socket	socket_write
getsockopt	socket_close	sendto
listen	socket_ioctl	tcpware_server

Lookup Operations

The Socket Library contains the following lookup operations:

getdomainname	getnetbyaddr	HNS_LOOKUPHOST
gethostname	getnetbyname	HNS_LOOKUPIA
gethostbyaddr	getprotobyname	resolver
gethostbyname	getprotobynumber	setdomainname

gethostid	getservbyname/ getservbyport	sethostname

Byte Order Conversion Operations

The Socket Library contains the following byte order conversion functions, defined in the TYPES.H and IN.H header files:

htonl	htons	ntohl	ntohs

Byte String Operations

The Socket Library contains the following byte string operations:

bcopy	bcmp	bzero

Internet Address Conversion Subroutines

The Socket Library contains the following internet address conversion subroutines:

inet_addr	inet_makeaddr	inet_network	inet_aton
inet_lnaof	inet_netof	inet_ntoa	

Server Operation

The Socket Library contains one server operation subroutine called `tcpware_server`.

TheSample Discard Protocol Programs section shows how to use `tcpware_server`.

Subroutine Data Structures

The following sections contain information about the data structures included in various socket library subroutines. These structures include:

hostent	protoent	sockaddr_in	netent	servent

hostent

The `hostent` structure represents the internet-host-name-to-address mappings in the subroutines `gethostbyaddr` and `gethostbyname`. The structure is defined as follows in the NETDB.H header file in the TCPWARE_INCLUDE: directory:

```
struct  hostent         {
        char     *h_name;         /* official host name */
        char     **h_aliases;     /* alias list */
          int    h_addrtype;      /* host address type */
          int    h_length;        /* address length */
          char   **h_addr_list;   /* list of addresses (name server) */
#define   h_addr haddr_list[0];   /* address (backward compatibility) */
};
```

The `h_addr_list` list of addresses is null-terminated and is required because some hosts can have many addresses, each having the same name. The `h_addr` definition provides backward compatibility and is the first address (in network byte order) in the list of addresses in the `hostent` structure.

netent

The `netent` structure represents the network name/number mappings used with the subroutines `getnetbyname` and `getnetbyaddr`. The structure is defined as follows in the NETDB.H header file in the TCPWARE_INCLUDE: directory:

```
struct  netent   {
        char     *n_name;      /* official name of net */
        char     **n_aliases;  /* alias list */
        int      n_addrtype;   /* net address type */
        unsigned long   n_net; /* network #, host byte order */
};
```

protoent

The `protoent` structure represents the protocol-name mappings used with the subroutines `getprotobyname` and `getprotobynumber`. The structure is defined as follows in the NETDB.H header file in the TCPWARE_INCLUDE: directory:

```
struct  protoent   {
        char       *p_name;      /* official protocol name */
        char       **p_aliases;  /* alias list */
        int        p_proto;      /* protocol number */
};
```

servent

The `servent` structure represents the service mappings used with the subroutines `getservbyname` and `getservbynumber`. The structure is defined as follows in the NETDB.H header file in the TCPWARE_INCLUDE: directory:

```
struct  servent   {
        char      *s_name;       /* official service name */
        char      **s_aliases;   /* alias list */
        int       s_port;        /* port number, network byte order */
        char      *s_proto;      /* protocol to use */
};
```

sockaddr_in

The `sockaddr_in` structure represents the socket address used with the subroutines `bind`, `connect`, `getpeername`, `getsockname`, `getservbyname` and `getservbynumber`. The structure is defined as follows in the IN.H header file in the TCPWARE_INCLUDE: directory:

```
struct    sockaddr_in          {
          short                sin_family;     /* address family */
          unsigned short       sin_port;       /* port number */
          struct in_addr       sin_addr;       /* address */
          char                 sin_zero[8];
};
struct   in_addr {
         unsigned long         s_addr;
};
```

WIN/TCP Socket Library Support

You can use most WIN/TCP applications with TCPware. You can do so if you use the existing WIN/TCP socket library and invoke the `SETUP_TWG.COM` command procedure provided with TCPware. This command procedure defines some logicals and sets up some files to emulate the Wollongong environment so that the WIN/TCP socket library can operate.

Using WIN/TCP applications with TCPware assumes that the applications were:

- Built with the Wollongong header files.
- Linked against the TWGLIB.OLB socket library or TWG_RTL.EXE run-time library.

Note that the Run-Time Library must be present on your system in SYS$SHARE if the application is linked against it.

Using WIN/TCP Applications Under TCPware

When you install TCPware for OpenVMS, one of the files provided is the TCPWARE:SETUP_TWG.COM file. TCPware does not invoke this command procedure. Edit your system startup file to invoke this procedure.

If you use applications developed for Wollongong's WIN/TCP (or Pathways) under TCPware for OpenVMS, observe the following:

1 Start TCPware if it is not running.

2 Invoke the `SETUP_TWG.COM` procedure, for example, **`@TCPWARE:SETUP_TWG`**

This command procedure uses the definitions of several TCPware logicals.

3 Edit your system startup file to include the following line after TCPware is started if you want the command procedure to be permanent: `$` **`@TCPWARE:SETUP_TWG`**

The SETUP_TWG.COM procedure:

- Defines the logical needed to use the WIN/TCP socket library under TCPware:

  ARPANET_HOST_NAME
  INET_DOMAIN_NAME
  INET_NAMESERVER_LIST
  TWG$TCP

- Creates the `TWG$TCP:[NETDIST.ETC]` directory it does not already exist.
- Copies the following TCPware files to the `TWG$TCP:[NETDIST.ETC]` directory:

  `HOSTS.`
  `NETWORKS.`
  `PROTOCOLS.`
  `SERVICES.`

Recompiling and Linking WIN/TCP Applications

Applications written for the WIN/TCP socket library that you recompile and then link with the TCPware Socket Library (object or RTL) will probably not work without modification. The WIN/TCP and TCPware socket libraries have differences, such as:

- TCPware uses TCPDRIVER and UDPDRIVER, not INETDRIVER. This means that any mix of SYS$QIO*[W]* calls with socket library calls will not work.
- The TCPware socket number is not the VMS I/O channel; it is the address of an internal data structure.
- Not all routines in the WIN/TCP library are available in the TCPware socket library. Also, not all routines in the TCPware socket library are available in the WIN/TCP library.
- Some routines use different names.

Fortunately, modifying the applications written for the WIN/TCP socket library so that they can link against TCPware's socket library usually does not require a lot of work. It can typically be done using conditional compilations.

Socket Library Reference

This section describes each Socket Library subroutine and function in detail.

accept

Waits for and accepts the next listening connection. Usually used by servers. Valid only for stream TCP sockets.

Format

snew = accept(*s*, *name*, *namelen*)

Arguments

int snew

Accepted connection's socket descriptor, or `-1` for failure.

int s

Socket descriptor (as returned by `socket`). The `listen` subroutine must have been called on this socket.

struct sockaddr_in *name

Address of the `sockaddr_in` structure to receive peer's internet address and port number. (See the *sockaddr_in* subsection for the `sockaddr_in` structure definition.)

int *namelen

Address of the length of the `sockaddr_in` structure (passed to and returned by `accept`).

Description

Allocates a new socket structure and performs a passive open on the socket with the port number from the listen socket.

Call the `listen` subroutine before calling `accept`**.**

Calls to the `socket` and `bind` subroutines are made to create the new socket.

QIO Function Performed

IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP | 0x0800 issued on the channel for the socket to wait for the next passive connection on the port.

See Also

listen

select

Status

Returns `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EADDRINUSE	Port number is already in use.
EBADF	Socket structure is not valid.
ECONNRESET	Peer resets the connection.
EINVAL	*namelen* value is not valid or listen has not been called on socket *s*.
EIO	Unexpected system status returned during operation.
EOPNOTSUPP	Not a stream socket.
ENETDOWN	Network was shut down.
ETIMEDOUT	Connection timed-out.
EWOULDBLOCK	Socket was set to non-blocking mode and the operation would block.

Also, `accept` may return any status that the `socket` and `bind` subroutines return.

The `vaxc$errno` variable contains the system service or I/O status code for `EADDRINUSE`, `ECONNRESET`, `EIO`, `ENETDOWN`, or `ETIMEDOUT`.

bcmp

Compares two buffers to determine if they are identical.

Format

status = bcmp(*b1*, *b2*, *length*)

Arguments

char *b1, *b2

Address of the first and second strings.

int length

Number of bytes to compare.

Description

Provides a byte string operation that compares two buffers of a specified length, returning zero if they are identical and non-zero if they are not. The function does not check for null bytes.

Status

- Returns `0` if the byte strings are identical or if the length is zero
- Returns non-zero if the byte strings are not identical

bcopy

Copies a specified number of bytes from one buffer to another.

Format

(void) bcopy(*b1*, *b2*, *length*)

Arguments

char *b1, *b2

Address of the source and destination strings. The two strings can overlap.

int length

Number of bytes to be copied.

Description

Provides a byte string operation that copies a specified number of bytes from one buffer to another. Does not check for null bytes. Overlapping strings are handled correctly.

bind

Binds the local internet address information to the socket.

Format

status = bind(*s*, *name*, *namelen*)

Arguments

int s

Socket descriptor (as returned by `socket`).

struct sockaddr_in *name

Address of the `sockaddr_in` structure containing the local internet address and local port number. (See the *sockaddr_in* subsection for the `sockaddr_in` structure definition.)

int namelen

Length of the `sockaddr_in` structure.

Description

Binds the local address information to the socket. You must specify the local internet address (or `INADDR_ANY[=0]`) and local port number (or `0`).

QIO Function Performed

- IO$_SETMODE | IO$M_CTRL for a stream (TCP) socket
- IO$_SETMODE | IO$M_CTRL | IO$M_STARTUP for a datagram (UDP) socket

Status

Returns `0` for success or `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows. The `vaxc$errno` variable is valid for `EADDRINUSE`, `EADDRNOTAVAIL`, `EIO`, or `ENETDOWN`.

EACCES	Insufficient privilege.
EADDRINUSE	Port number is already in use.
EADDRNOTAVAIL	Local internet address is invalid.
EBADF	Socket structure is not valid.
EINVAL	Name structure is invalid, its length is wrong, or the socket is already in use.
EIO	Unexpected system status returned during operation.
ENETDOWN	Network was shut down.

bzero

Places a specified length of zero bytes into a buffer.

Format

(void) bzero(*b*, *length*)

Arguments

char *b

Address of the string.

int length

Number of bytes to be zeroed.

Description

Provides a byte string operation that places a specified length of zero bytes into a buffer.

connect

Initiates an active connection to a server. It is usually used by the client-end of applications.

Format

status = connect(*s*, *server*, *serverlen*)

Arguments

int s

Socket descriptor (as returned by `socket`).

struct sockaddr_in *server

Address of the `sockaddr_in` structure containing the peer's internet address and port number. (See the *sockaddr_in* subsection for the `sockaddr_in` structure definition.)

int serverlen

Length of the `sockaddr_in` structure.

Description

This subroutine opens an active connection for a stream (TCP) socket. For a datagram (UDP) socket, the port is configured as fully specified (meaning that only datagrams from the specified peer can be received)**.** For a raw (IP) socket, the destination internet address is set.

A `bind` call is optional before a `connect` call.

QIO Function Performed

- IO$_SENSEMODE | IO$M_CTRL|IO$M_STARTUP for a stream (TCP) socket
- IO$_SENSEMODE | IO$M_CTRL for a datagram (UDP) socket

Status

The `connect` subroutine returns `0` for success or `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EADDRINUSE	Port number is already in use.
EAFNOSUPPORT	Address family specified in the `server` structure is not AF_INET.
EBADF	Socket structure is not valid.
ECONNREFUSED	Connection was refused by the peer.
EINVAL	Server structure is invalid, its length is wrong, or the socket is al ready in use.
EIO	Unexpected system status returned during operation.

ENETDOWN	Network was shut down.
ENETUNREACH	There is no routing information to reach the peer.
ETIMEDOUT	Connection timed out.

The `vaxc$errno` variable is valid for `EADDRINUSE`, `ECONNREFUSED`, `EIO`, `ENETDOWN`, `ENETUNREACH`, or `ETIMEDOUT`.

getdomainname / gethostname

Gets the local host's domain name or hostname.

Format

len = getdomainname(*name*, *namelen*)

len = gethostname(*name*, *namelen*)

Arguments

int len

Length of the domain name string (or `0` if none available).

char *name

Address of an ASCII (null-terminated) string in which to return the domain name, or that contains the domain name.

int namelen

Maximum length of character string name (passed to the subroutines).

Description

The user process `getdomainname` or `gethostname` subroutine reads the system logical name TCPWARE_DOMAINNAME.

See Also

setdomainname

sethostname

Status

- If no name is available, returns `0`
- Otherwise, returns the length of the domain name string

gethostbyaddr

Returns the host name or alias for an internet address.

Format

he = gethostbyaddr(*addr*, *len*, *type*)

Arguments

struct hostent *he

Address of the returned `hostent` structure, or `0` (NULL) if no match is found. (See the *hostent* subsection for the `hostent` structure definition.)

The returned structure is in a static area, so you must copy it to save it.

char *addr

Address of the host's four-byte internet address, in network byte order.

int len

Length of an internet address (must be **4**).

int type

Address type (must be **AF_INET = 2**). AF_INET is defined in SOCKET.H.

Description

Uses the DNS Name Server or cached-in-memory version of your local Hosts database to locate the host name or alias of the given internet address. A process reloads the Hosts database into memory every 10 minutes by default if modifications exist. The database is in the TCPWARE:HOSTS. file.

See the `gethostbyname` subroutine for more details on the Hosts database.

See Also

gethostbyname

gethostbyname

Returns the internet address for a named host or alias.

Format

he = gethostbyname(*name*)

Arguments

struct hostent *he

Address of the returned `hostent` structure, or `0` (NULL) if no match is found. (See the *hostent* subsection for the `hostent` structure definition.)

The returned structure is in a static area, so you must copy it to save it.

char *name

Address of an ASCII (null-terminated) string containing the name or alias of the host. The string is not case-sensitive.

Description

Uses the DNS Name Server or cached version of your local Hosts database to locate the internet address of the host name or alias entry. A process reloads the Hosts database into memory every 10 minutes by default if modifications exist. The database is in the TCPWARE:HOSTS. file.

The TCPWARE_SVCORDER logical name contains the list of services used in the order specified. The valid values for the logical are:

local	Uses the HOSTS. file.
bind	Uses DNS (provided the TCPWARE_NAMESERVERS and TCPWARE_DOMAINNAME logicals are properly defined).

You can also use the values **`"bind,local"`** (the default if the logical is not defined) and **`"local,bind"`** (which uses DNS if the Hosts database lookup fails).

If you do not use DNS and want to read from the HOSTS. files, edit it in the proper format. Each entry must be ***address hostname [aliases]***, where *address* is the host internet address, *hostname* the official host name, and *aliases* a list of alias names separated by spaces. Comments can appear prefixed by a `#`. If you want to read the HOSTS. file, use the `sethostent`, `gethostent`, and `endhostent` subroutines:

`sethostent(int stayopen)`	opens (or rewinds) the database; if the `stayopen` value is non-zero, the database remains open
`struct hostent *gethostent()`	returns the next entry's `hostent` structure address, or zero if all entries were read
`endhostent()`	closes the database (unless `stayopen` is non-zero)

`gethostbyname` returns every entry satisfying the search criteria. Therefore, avoid equating a host with a standard alias, defining a hostname by an existing alias, and equating more than one host with the same alias.

gethostid

Returns the local host's internet address.

Format

hostid = gethostid()

Argument

int hostid

Returned internet address for the local host in byte reversed order. (Returns `-1` if no address is available.)

Description

Returns one of the local internet addresses for a network interface.

See Also

gethostname

sethostname

getnetbyaddr

Returns the `netent` structure for a network number.

Format

ne = getnetbyaddr(*net*, *type*)

Arguments

struct netent *ne

Address of the returned `netent` structure, or `0` (NULL) if no match is found. (See the *netent* subsection for the `netent` structure definition.)

The returned structure is in a static area, so you must copy it to save it.

long net

Network number, in host byte order (such as the value returned by `inet_network`).

int type

Address type (must be **`AF_INET = 2`**). AF_INET is defined in SOCKET.H.

Description

Uses the cached-in-memory version of your local Networks database to locate the entry for the given network address. A process reloads the Networks database into memory every 10 minutes by default if modifications exist. The database is in the TCPWARE:NETWORKS. file.

See thegetnetbyname subroutine for more details on the Networks database.

See Also

getnetbyname

getnetbyname

Returns the `netent` structure for a named network.

Format

ne = getnetbyname(*name*)

Arguments

struct netent *ne

Address of the returned `netent` structure, or `0` (NULL) if no match is found. (See the *netent* subsection for the `netent` structure definition.)

The returned structure is in a static area, so you must copy it to save it.

char *name

Address of an ASCII (null-terminated) string containing the name or alias of the network. The network name string is case-sensitive.

Description

Uses the cached-in-memory version of your local Networks database to locate the entry for the given network name or alias. A process reloads the Networks database into memory every 10 minutes by default if modifications exist. The database is in the TCPWARE:NETWORKS. file.

If you want to read from the NETWORKS. files, edit it in the proper format. Each entry must be ***name number [aliases]***, where *name* is the official name for the network, *number* the network number, and *aliases* a list of alias names separated by spaces. Comments can appear in the file prefixed by a `#`. If you want to read the NETWORKS. file, use the `setnetent`, `getnetent`, and `endnetent` subroutines:

`setnetent (int stayopen)`	opens (or rewinds) the database; if the `stayopen` value is non-zero, the database remains open
`struct netent *getnetent`	returns the next entry's `netent` structure address, or zero if all entries were read
`endnetent()`	closes the database (unless `stayopen` is non-zero)

getpeername

Returns the peer's internet address and port number for a socket.

Format

status= getpeername(*s*, *name*, *namelen*)

Arguments

int s

Socket descriptor (as returned by `socket`). (See the *sockaddr_in* subsection for the `sockaddr_in` structure definition.)

struct sockaddr_in *name

Address of the `sockaddr_in` structure in which to return the peer's internet address and port number.

int *namelen

Address of the length of the `sockaddr_in` structure (passed to and returned by getpeername).

Description

Returns the peer's internet address and port number stored in the socket structure. Note that this information is set by `connect` and `accept`.

Status

Returns `0` for success and `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EBADF	Socket structure is not valid.
EINVAL	*namelen* value is incorrect. It must be `sizeof(struct sockaddr_in)`.
ENOTCONN	Socket is not connected.

getprotobyname

Returns the protocol number for a named protocol or alias.

Format

pe = getprotobyname(*name*)

Arguments

struct protoent *pe

Address of the returned `protoent` structure, or `0` (NULL) if no match is found. (See the *protoent* subsection for the `protoent` structure definition.) The returned structure is in a static area, so you must copy it to save it.

char *name

Address of an ASCII (null-terminated) string containing the protocol name (such as `TCP` or `UDP`) or alias. The string is not case-sensitive.

Description

Scans the TCPWARE:PROTOCOLS. file and returns the entry for the given protocol name or alias. A process checks the file for modifications by default every 10 minutes and then reloads it into memory if modifications exist.

You must format the PROTOCOLS. file properly for the subroutine to work. Format each entry as ***protocol port [aliases]***, where *protocol* is the protocol name, *port* the decimal port number, and *aliases* a list of alias names separated by spaces. Comments may appear in the file prefixed by a `#`.

The PROTOCOLS. file, as supplied, contains standard Internet protocols. Changes to this file may affect the operation of your application and TCPware as well. To avoid conflict, be sure when adding a name or an alias that it is not already in use.

To read the PROTOCOLS. file, use the `setprotoent`, `getprotoent`, and `endprotoent` subroutines:

`setprotoent(int stayopen)`	opens (or rewinds) the database; if the `stayopen` value is non-zero, the database remains open
`struct protoent *getprotoent`	returns the next entry's `protoent` structure address, or zero if all entries were read
`endprotoent()`	closes the database (unless `stayopen` is non-zero)

See Also

getprotobynumber

getprotobynumber

Returns the protocol name or alias for a protocol number.

Format

pe = getprotobynumber(*proto*)

Arguments

struct protoent *pe

Address of the returned `protoent` structure, or `0` (NULL) if no match found. (See the *protoent* subsection for the `protoent` structure definition.)

The returned structure is in a static area, so you must copy it to save it.

int proto

Protocol number of the desired protocol name or alias.

Description

Scans the TCPWARE:PROTOCOLS. file and returns the entry for the given protocol name. A process checks the file for modifications by default every 10 minutes and then reloads it into memory if modifications exist.

You must format the PROTOCOLS. file properly for the subroutine to work. Format each entry as ***protocol port [aliases]***, where *protocol* is the protocol name, *port* the decimal port number, and *aliases* a list of alias names separated by spaces. Comments may appear in the file prefixed by a `#`.

The PROTOCOLS. file, as supplied, contains standard Internet protocols. Changes to this file may affect the operation of your application and TCPware as well. To avoid conflict, be sure when adding a name or an alias that it is not already in use. To read from the PROTOCOLS. file, use the `setprotoent`, `getprotoent`, and `endprotoent` subroutines:

`setprotoent(int stayopen)`	opens (or rewinds) the database; if the `stayopen` value is non-zero, the database remains open
`struct protoent *getprotoent`	returns the next entry's `protoent` structure address or zero if all entries were read
`endprotoent()`	closes the database (unless `stayopen` is non-zero)

See Also

getprotobyname

getservbyname

Returns the port number for a named service.

Format

se = getservbyname(*name*, *proto*)

Arguments

struct servent *se

Address of the returned `servent` structure, or `0` (NULL) if no match found. (See the *servent* subsection for the `servent` structure definition.) The returned structure is in a static area, so you must copy it to save it. Only the Internet services and protocols are understood.

char *name

Address of an ASCII (null-terminated) string containing the service name (such as `FTP`) or alias. The string is not case-sensitive.

char *proto

Address of an ASCII (null-terminated) string containing the protocol name (such as `TCP` or `UDP`) or alias. The string is not case-sensitive.

Description

Uses the cached-in-memory version of your local Services database to locate the entry for the given service name. A process reloads the Services database into memory every 10 minutes by default if modifications exist. The database is in the TCPWARE:SERVICES. file. The database is reloaded into memory by default every 10 minutes.

If you want to read from the SERVICES. files, edit it in the proper format. Each entry must be ***service port*/*protocol [aliases]***, where: *service* is the service name, *port* the decimal port number followed after a slash (`/`) by *protocol*, the protocol (TCP or UDP), and *aliases* a list of alias names separated by spaces. Comments can appear in the file prefixed by a `#`. If you want to read the SERVICES. file, use the `setservent`, `getservent`, and `endservent` subroutines:

`setservent(int stayopen)`	opens (or rewinds) the database; if the `stayopen` value is non-zero, the database remains open
`struct servent *getservent`	returns the next entry's `servent` structure address, or zero if all entries were read
`endservent()`	closes the database (unless `stayopen` is non-zero)

See Also

getservbyport

getservbyport

Returns the service name or alias for a port number.

Format

se = getservbyport(*port*, *proto*)

Arguments

struct servent *se

Address of the returned `servent` structure, or `0` (NULL) if no match found. (See the *servent* subsection for the `servent` structure definition.)

The returned structure is in a static area, so you must copy it to save it. Only the Internet services and protocols are understood.

int port

Port number of the desired service name. Passed in byte reversed order (use the `htons` subroutine).

char *proto

Address of an ASCII (null-terminated) string containing the protocol name (such as `TCP` or `UDP`) or alias. The string is not case-sensitive.

Description

Uses the cached-in-memory version of your local Services database to locate the entry for the given port number. A process reloads the Services database into memory every 10 minutes by default if modifications exist. The database is in the TCPWARE:SERVICES. file. The database is reloaded into memory by default every 10 minutes.

If you want to read from the SERVICES. files, edit it in the proper format. Each entry must be ***service port*/*protocol* [*aliases*]**, where:*service* is the service name, *port* the decimal port number followed after a slash (`/`) by *protocol*, the protocol (TCP or UDP), and *aliases* a list of alias names separated by spaces. Comments can appear in the file prefixed by a `#`. If you want to read the SERVICES. file, use the `setservent`, `getservent`, and `endservent` subroutines:

`setservent(int stayopen)`	opens (or rewinds) the database; if the `stayopen` value is non-zero, the database remains open
`struct servent *getservent`	returns the next entry's `servent` structure address, or zero if all entries were read
`endservent()`	closes the database (unless `stayopen` is non-zero)

See Also

getservbyname

getsockname

Returns the local internet address and port number for a socket.

Format

status= getsockname(*s*, *name*, *namelen*)

Arguments

int s

Socket descriptor (as returned by `socket`).

struct sockaddr_in *name

Address of the `sockaddr_in` structure in which to return the peer's internet address and port number.

(See the *sockaddr_in* subsection for the `sockaddr_in` structure definition.)

int *namelen

Address of the length of the `sockaddr_in` structure (passed to and returned by `getsockname`).

Description

Returns the local internet address and port number stored in the socket structure. This information is set by `bind`.

Status

Returns `0` for success and `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EBADF	Socket structure is not valid.
EINVAL	*namelen* value is incorrect. It must be a pointer to an integer whose value is the size of the sockaddr_in subroutine.

getsockopt

Returns option information regarding a socket.

The SOCKET.H file contains definitions for the socket-level options. The IN.H file contains definitions for the IPPROTO_IP level options.

Format

status= getsockopt(*s*, *level*, *optname*, *optval*, *optlen*)

Arguments

int s

Socket descriptor (as returned by the socket).

int level

SOL_SOCKET to return socket options, or **IPPROTO_IP** to return IP options (which requires a SYSTEM UIC, or the SYSPRV or BYPASS privilege).

int optname

Option code to return.

See thesetsockopt description for the currently supported options.

char *optval

Address of the value for the option (if the option requires a value).

int *optlen

On input, length of the `optval` buffer. On return, amount of data returned to the *optval* buffer.

Description

The specified socket option value is returned.

Status

Returns `0` for success and `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EBADF	Invalid socket structure.
EINVAL	Invalid *optname*, *optval*, or *optlen*.

HNS_LOOKUPHOST

Designed for use by FORTRAN programs to return the internet address for a host name.

C programmers should use the `gethostbyname` subroutine.

Format

status = HNS_LOOKUPHOST(*host-name*, *internet-address*)

Arguments

CHARACTER*(*) host-name

Address of the string descriptor for the host name (or an ASCII internet address).

INTEGER*4 internet-address

Address of an INTEGER*4 to which the internet address is returned in network byte order.

Description

Calls the `gethostbyname` subroutine to obtain the internet address for the host.

Status

Returns the following status codes:

SS$_NORMAL	Success. Internet address for the host name has been returned.
SS$_NOSUCHNODE	No translation for the host name to an internet address could be found.

HNS_LOOKUPIA

Designed for use by FORTRAN programs to return the host name for an internet address.

C programmers should use the `gethostbyaddr` subroutine.

Format

status = HNS_LOOKUPIA(*host-name*, *internet-address*)

Arguments

CHARACTER*(*) host-name

Address of the string descriptor to which the host name is returned.

INTEGER*4 internet-address

Address of an INTEGER*4 containing the internet address for the host, in network byte order.

Description

Calls the `gethostbyaddr` subroutine to obtain the host name for the internet address.

Status

Returns the following status codes:

SS$_NORMAL	Success. Host name has been returned.
SS$_NOSUCHNODE	No host name could be found for the internet address.

htonl

Swaps the byte order of a four-byte integer from VAX byte order to network byte order.

Programmers can use this function to develop programs independent of the hardware architectures.

Format

retval = htonl(*val*)

Arguments

int retval

Byte-swapped integer corresponding to *val*.

int val

Four-byte integer to convert to network byte order.

Description

Converts between 32-bit (long) host byte order and network byte order.

Requires the TYPES.H and IN.H header files.

Status

Returns the byte-swapped integer that corresponds to *val*. For example, if *val* is `0xc029e401`, the returned value is `0x01e429c0`.

htons

Swaps the byte order of a two-byte integer from VAX byte order to network byte order.

Programmers can use this function to develop programs independent of the hardware architectures.

Format

retval = htons(*val*)

Arguments

int retval

Byte-swapped integer corresponding to *val*.

int val

Two-byte integer to convert to network byte order.

Description

Converts between 16-bit (short) host byte order and network byte order.

Requires the TYPES.H and IN.H header files.

Status

Returns the byte-swapped integer that corresponds to *val*. For example, if *val* is `0x0017`, the returned value is `0x1700`.

inet_

The `inet_` subroutines:

- Convert internet addresses from text to binary form and vice versa
- Build internet addresses when given a network number and local address
- Return the network or local portions of the internet address when given a complete address

The `inet_` subroutines are defined in the INET.H header file.

Format

int = inet_addr(cp)

net = inet_network(cp)

cp = inet_ntoa(in)

flg = inet_aton(cp, &in)

in = inet_makeaddr(net, lna)

lna = inet_lnaof(in)

net = inet_netof(in)

ALPHA and I64

The subroutine names are prefixed with **tcpware_** to prevent name conflicts with the C language RTL.

Arguments

struct in_addr in

Internet address in binary form in network byte order. This structure is defined as follows:

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

char *cp

ASCII character string containing an internet address in standard *a.b.c.d* format. A program calling these subroutines can specify all ASCII numbers as decimal, octal, or hexadecimal (as specified in C language).

int flg

Flag that returns `1` if *cp* is a valid ASCII representation of an IP address, and `0` if it is not.

long net

Network number in VAX byte order.

int lna

Host address in VAX byte order.

Description

inet_addr	Converts an internet address from an ASCII string to binary form. Returns -1 if the address is invalid. Supports decimal, hexadecimal, and octal values for the internet address components using standard C notation (that is, the 0x prefix before hexadecimal values and a leading 0 before octal values).
inet_network	Converts an ASCII network number (*a*, *a*.*b*, or *a*.*b*.*c*) to a binary value in VAX byte order. Returns -1 if the address is invalid. For example, a network number of 192.9.200 would be returned as 0X00C009C8.
inet_ntoa	Converts an internet address in network order from a binary string to ASCII text format. Returns a pointer to the string in *a*.*b*.*c*.*d* format.
inet_aton	Checks an ASCII internet address for validity and converts it to a binary address. Returns 1 if valid, 0 if invalid. Replaces `inet_addr` since `inet_addr` cannot distinguish between a failure and a local address.
inet_makeaddr	Returns an internet address when given a network number and a local address.
inet_lnaof	Returns the host address portion of an internet address. The returned value is in VAX byte order.
inet_netof	Returns the network portion of an internet address. The returned value is in VAX byte order.

You must use the INET.H header file to define these subroutines. The `in_addr` structure is defined in the IN.H header file.

Status

The `inet_addr` and `inet_network` subroutines return `-1` if the address is invalid. The program cannot distinguish between an error condition and the 255.255.255.255 internet address.

ipso_getauthbyname

Returns the bit mask value corresponding to an IPSO authority name.

Format

mask = ipso_getauthbyname (*name*)

Arguments

unsigned long mask

Mask corresponding to the name provided, or 0 if no match if found.

char *name

Address of the ASCII string containing the name of the IPSO authority field. See Table A-3 for a list of valid IPSO protection authorities.

Description

This subroutine uses the TCPWARE:IPSO_AUTHORITIES. file to match the name against a bit mask. The IPSO_AUTHORITIES. file must be properly formatted for the subroutine to work. See Example A-1 for sample contents of this file.

Requires the IPSO.H header file.

Table A-3 IPSO Protection Authorities

Protection Authority	Hexadecimal Value	Point of Contact
GENSER	%X80	Designated Approving Authority per DOD 5200.28
SIOP-ESI	%X40	DoD Joint Chiefs of Staff
SCI	%X20	Director of Central Intelligence
NSA	%X10	National Security Agency
DOE	%X08	Department of Energy

Example A-1 Sample IPSO_AUTHORITIES. File

```
!GENSER         0x80
!SIOP-ESI       0x40
!SCI            0x20
!NSA            0x10
!DOE            0x08
!
!SITE-SPECIFIC:
ALPHA                0x30          !SCI+NSA
BETA                 0x50          !SIOP-ESI+NSA
GAMMA                0x18          !NSA+DOE
DELTA                0x58          !SIOP-ESI+NSA+DOE
```

ipso_getauthbynumber

Returns the IPSO authority name corresponding to a bit mask value.

Format

char **name* = ipso_getauthbynumber (*mask*)

Arguments

char *name

Address of the ASCII string to contain the name of the IPSO authority field. See Sample IPSO_AUTHORITIES. File for a list of valid IPSO protection authorities.

unsigned long mask

Mask for the name.

Description

This subroutine uses the TCPWARE:IPSO_AUTHORITIES. file to match the name against a bit mask. The IPSO_AUTHORITIES. file must be properly formatted for the subroutine to work. See Sample IPSO_AUTHORITIES. File for sample contents of this file.

Requires the IPSO.H header file.

ipso_getlevelbyname

Returns the bit mask value corresponding to an IPSO security level.

Format

mask = ipso_getlevelbyname (*name*)

Arguments

unsigned long mask

Mask corresponding to the name provided, or 0 if no match if found.

char *name

Address of the ASCII string containing the name of the IPSO security level. See IPSO Security Levels for a list of valid IPSO security levels.

Description

This subroutine searches a list of the IPSO classifications for a name and returns the corresponding bit mask value. See Table A-4 for a list of valid IPSO security levels.

Requires the IPSO.H header file.

Table A-4 IPSO Security Levels

Security Level	Hexadecimal Value
Top_Secret	%X3D
Secret	%X5A
Confidential	%X96
Unclassified	%XAB

ipso_getlevelbynumber

Returns the IPSO security level corresponding to a bit mask value.

Format

char **name* = ipso_getlevelbynumber (*mask*)

Arguments

char *name

Address of the ASCII string to contain the name of the IPSO security level. See IPSO Security Levels for a list of valid IPSO security levels.

unsigned long mask

Mask for the name.

Description

This subroutine searches a list of the IPSO classifications for a name and returns the corresponding bit mask value. See IPSO Security Levels for a list of valid IPSO security levels.

Requires the IPSO.H header file.

listen

Makes it possible to listen for connections on a port number. It is only valid for stream sockets and is usually used by the server end of an application.

Format

status = listen(*s*, *backlog*)

Arguments

int s

Socket descriptor (as returned by the socket).

int backlog

Maximum number of outstanding connections that can be queued.

Description

The value of *backlog* determines the value of the TCPDRIVER passive open access control parameter. If *backlog* is greater than 1, non-exclusive access is requested. The value of *backlog* must be between 1 and 16, inclusive. Call the `accept` subroutine to accept a connection.

The channel associated with the socket is deassigned by listen since the channel will not be used (only the local internet address, port number, and *backlog* values from the socket structure are used by `accept`).

See Also

accept

select

Status

Returns `0` for success and `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EACCES	Insufficient privilege.
EBADF	Socket structure is not valid.
EINVAL	Value of *backlog* is not valid or the socket is already in use.
EOPNOTSUPP	Not a stream socket.

ntohl

Swaps the byte order of a four-byte integer from network byte order to VAX byte order.

Programmers can use this function to develop programs independent of the hardware architectures.

Format

retval = ntohl(*val*)

Arguments

int retval

Byte-swapped integer corresponding to *val*.

int val

Four-byte integer to convert to network byte order.

Description

Converts between 32-bit (long) network byte order and host byte order.

Requires the TYPES.H and IN.H header files.

Status

Returns the byte-swapped integer that corresponds to *val*. For example, if *val* is `0x01e429c0`, the returned value is `0xc029e401`.

ntohs

Swaps the byte order of a two-byte integer from network byte order to VAX byte order.

Programmers can use this function to develop programs independent of the hardware architectures.

Format

retval = ntohs(*val*)

Arguments

int retval

Byte-swapped integer corresponding to *val*.

int val

Two-byte integer to convert to network byte order.

Description

Converts between 16-bit (short) network byte order and host byte order.

Requires the TYPES.H and IN.H header files.

Status

Returns the byte-swapped integer that corresponds to *val*. For example, if *val* is `0x1700`, the returned value is `0x0017`.

pneterror

Displays the error message for the error status returned by a Socket Library subroutine.

This subroutine has the same inputs and format as `perror`; however, it will print the text for network errors (see SOCKERR.H).

Format

status = pneterror(*s*)

Argument

char *s

Address of an ASCII (null-terminated) character string to be displayed before the error text.

Description

Displays the error message for the error code in the `errno` variable.

Status

Displays the error message for the error status returned by a Socket Library subroutine. If the error code is not a known network error code, the standard VAX C error subroutine is called.

recvfrom

Reads data from an unconnected or connected socket.

Format

msglen = recvfrom(*s*, *buffer*, *buflen*, *flags*, *from*, *fromlen*)

Arguments

int msglen

-1 for error, 0 for connection closed by peer and no more data available, >0 for the number of bytes of data read.

int s

Socket descriptor (as returned by `socket`).

char *buffer

Address of the buffer into which the data is to be received.

int buflen

Length of the buffer into which data is to be received.

int flags

Flag bits for operation. The supported flag bits are:

Flag Bit	Description
MSG_NONBLOCKING	returns immediately if no data is available (EWOULDBLOCK is returned in `errno` if no data is available).
MSG_PEEK	lets you peek at the data without removing it from the data stream.
MSG_TRUNCATE	returns truncated datagrams (otherwise ENOMEM is returned if the datagram is larger than the buffer). This flag is only used by datagram (UDP) sockets.
MSG_TIME	limits the time to wait for a datagram to be received. The time limit for the receive is set using the `setsockopt` subroutine. This flag is only used by datagram (UDP) sockets.

All other flags are ignored.

char *from

Address of the `sockaddr_in` structure (or `0`) to be filled with peer's internet address and port number. (See the *sockaddr_in* subsection for the `sockaddr_in` structure definition.)

int *fromlen

Address of the length of the *from* argument.

QIO Function Performed

- IO$_READVBLK on the socket's channel
- IO$M_NOW modifier if specifying MSG_NONBLOCKING
- IO$M_DATACHECK modifier if specifying MSG_PEEK (see *flags*)

See Also

`select`, if your application handles multiple connections simultaneously

Status

Returns `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EBADF	Socket structure is not valid.
ECONNRESET	Connection is reset by the peer.
EINVAL	*fromlen* value is invalid (for recvfrom only).
EIO	Unexpected system status returned during operation.
ENETDOWN	Network was shut down.
ENOMEM	Buffer was too small for the datagram and MSG_TRUNCATE was not specified. Note that the truncated datagram is in the buffer.
ENOTCONN	Socket is not connected.
ETIMEDOUT	Connection timed-out or the request timed-out (if MSG_TIME set for a datagram socket).
EWOULDBLOCK	MSG_NONBLOCKING flag is set and no data is available.

The `vaxc$errno` variable contains the system service or I/O status code for `ECONNRESET`, `EIO`, `ENETDOWN`, `ETIMEDOUT`, or `EWOULDBLOCK`.

resolver

The `resolver` subroutines create, send, and interpret packets to DNS servers. DNS is primarily a host name and address lookup service for the Internet, allowing client systems to obtain host names and addresses from DNS servers.

For more information on DNS, see Chapter 6, *Domain Name Services*, in the *Management Guide.*

The `resolver` subroutines:

- Initialize the routines
- Store a standard query message in a buffer
- Send a query to the DNS servers and return an answer
- Compress and expand the domain name

Format

res_init()

res_mkquery*(op, dn, class, type, data, datal, newrr, buf, bufl)*

res_send*(msg, msglen, answer, anslen)*

dn_comp*(exp-dn, comp-dn, length, dnptrs, lastdnptr)*

dn_exp*(msg, eomorig, comp-dn, exp-dn, length)*

Arguments

int op

Opcode. Usually QUERY, but can be any of the query types defined in the NAMESER.H header file.

char *dn

Pointer to the domain name. If *dn* consists of a single label and the RES_DEFNAMES flag is enabled (the default), *dn* is appended with the current domain name, which is defined by the TCPWARE_DOMAINNAME logical name.

int class

Address class, which is typically C_IN (for Internet).

int type

Query type, which is typically T_A for a host name lookup.

char *data

Pointer to the resource record data.

int datal

Length in bytes of the resource record data.

struct rrec *newrr

Address of the new resource record data structure for modify or append operations.

char buf

Buffer in which the routine places the standard query message data.

int bufl

Length in bytes of the buffer in which the routine places the standard query message data.

char *msg

Pointer to the beginning of the message.

int msglen

Length in bytes of the message.

char *answer

Pointer to the answer to the standard query.

int anslen

Length in bytes of the answer to the standard query.

char *exp_dn

Pointer to a buffer of the expanded domain name.

char *comp-dn

Buffer for the compressed domain name for the expand routine (`dn_comp`).

int length

Size in bytes of the array to which *comp-dn* points. In the case of `dn_expand`, *length* refers to the length in bytes of the resulting expansion buffer.

char **dnptrs

List of pointers to previously compressed names in the current message. The first pointer points to the beginning of the message and the list ends with NULL.

char **lastdnptr

Pointer to the end of the array to which *dnptrs* points. Also updates the list of pointers for labels inserted into the message by `dn_comp` as the name is compressed. If *dnptrs* is NULL, the names are not compressed; if *lastdnptr* is NULL, the list is not updated.

char *eomorig

Pointer to the first byte after the message.

Description

res_init	Initializes the routines by getting the default domain name and internet address of the initial host running the name server. The domain name is defined by the TCPWARE_DOMAINNAME logical. The name servers are defined by the TCPWARE_NAMESERVERS logical.
res_mkquery	Makes a standard query message and places it in *buf*. The routine returns the size of the query or -1 if the query is larger than *bufl*.
res_send	Sends a query to the DNS servers and returns an answer. The routine calls the res_init routine. The length of the message is returned or -1 if there were errors.

dn_comp	Compresses the domain name and stores it in a buffer (comp_dn). The size of the compressed name is returned, or -1 if there were errors.
dn_expand	Expands the compressed domain name (dn_comp) to a full domain name. Expanded names are converted to uppercase. The size of the compressed name is returned, or -1 if there was an error.

The TYPES.H, IN.H, NAMESER.H, and RESOLV.H header files define the structures and constants needed by the `resolver` routines.

Global information used by the `resolver` routines is stored in the `_res` structure. Most of the values have reasonable defaults. The options are a simple bit mask and are OR-ed in to enable. The options are stored in `_res.options` and the mask values are defined in TCPWARE_INCLUDE:RESOLV.H. The options are as follows:

RES_INIT	Initial name server address and default domain names are initialized.
RES_DEBUG	Print debugging messages.
RES_AAONLY	Accept authoritative messages only.
RES_USEVC	Use TCP connections (virtual circuits) instead of UDP connections for queries.
RES_STAYOPEN	Used with RES_USEVC to keep the TCP connection open.
RES_RECURSE	Set the recursion desired bit in queries (the default). The res_send routine does not do iterative queries and expects the BIND server to handle recursion.
RES_DEFNAMES	Append the default domain name to single-label queries (the default).

select

Allows for synchronous I/O multiplexing.

Format

nfound = select(*nchan*, *readds*, *writeds*, *exceptds*, *timeout*)

Arguments

int nfound

Contains **-1** for error, **0** for timeout, or the number of sockets ready for reading (>0).

int nchan

If not using the FD_SET, FD_CLR, FD_ZERO macros to build the list of descriptors (see below), *nchan* must be 0. If using FD_SET, FD_CLR, FD_ZERO, specify **FD_SETSIZE** for *nchan*.

int *readds

Address of the descriptor set for the sockets to be checked if ready for reading (if connected) or ready to be accepted (if listening).

The following macros are provided for manipulating the descriptor set:

FD_ZERO (&*fdset*)	initializes the *fdset* descriptor set to the null set
FD_SET (*s*, &*fdset*)	includes the socket *s* in the set
FD_CLR (*s*,&*fdset*)	removes the socket *s* from the set
FD_ISSET (*s*,&*fdset*)	is non-zero if *s* is a member of the set

Use the *fd_set* typedef to declare a descriptor set. The FD_SETSIZE symbol defines the maximum number of sockets that may be specified in a descriptor set. If not explicitly defined by the user before including the TYPES.H header file, a default value of 64 is used.

Note! It is recommended that you use FD_SET, FD_CLR, FD_ZERO, and FD_ISSET macros as they improve portability and support future revisions to the socket library. Code that directly builds the currently used zero-terminated array of integers (one socket descriptor per integer) should be modified to make use of these macros as soon as possible.

int *writeds

Ignored. For future use (to determine which sockets are ready for writing).

int *exceptds

Ignored. For future use (to determine which sockets have had exceptions).

struct {long tv_sec, tv_usec;} *timeout

Maximum time (in seconds and microseconds) to wait for one or more sockets to be ready.

Description

Checks whether any socket in the *readds* list is ready to be read or accepted, have timed out, or were reset or closed by the peer. The number of sockets ready is returned in *nfound*.

If timeout is a nonzero pointer and the time values are 0, a poll is affected and `select` returns immediately. If timeout is a zero pointer, `select` returns only after at least one socket is ready. If one or more sockets are ready, the *readds* list is updated to reflect those sockets that are ready.

A socket may be ready for reading. However, this does not mean a read will complete immediately, because a push may not have been received and more bytes may have been requested than were actually received. We recommend that you do a non-blocking read on the socket.

See the `socket_recv` routine.

User-written AST routines can call the `select_wake()` subroutine to wake up a `select`. `select` will return 0 (the timeout status) in this case. Calling `select_wake` when no `select` is active will cause the next `select` to return 0 (the timeout status). The subroutine also allows for a sleep state (`select(0,0,0,0,&time)`) and an infinite wait state (`select(0,0,0,0,0)`).

Requirements

Uses the following system resources:

- One buffered-I/O request for each socket (the BUFIO quota must be sufficient for the application)
- One event flag (allocated and freed using LIB$GET_EF and LIB$FREE_EF, respectively)
- One timer (if a timeout points to a nonzero value)

Also, you must enable ASTs if a timer is required.

Status

Returns `0` in *nfound* for a timeout, `-1` for a failure, or the number of sockets ready (>0). For a failure status, the `errno` variable contains the reason for the error as follows:

EBADF	Invalid or duplicate socket is specified.
EINVAL	Socket list is not specified or the time out value is negative.
EIO	Unexpected system status returned during operation.
ENOTCONN	Socket is not in a valid state. Only connected or listening sockets may be specified in the *readds* list.

The `vaxc$errno` variable contains the system service status code for `EIO`.

sendto

Sends data over an unconnected or connected socket.

Format

cc = **sendto(**s, buffer, buflen, flags, to, tolen**)**

Arguments

int cc

Number of bytes sent or `-1` for failure.

int s

Socket descriptor (as returned by socket).

char *buffer

Address of the buffer which contains the data to be sent.

int buflen

Length of the buffer which contains the data to be sent.

int flags

Flag bits for operation. The supported flag bits are:

MSG_OOB	to send urgent data (for stream sockets only).

All other flags are ignored.

char *to

Address of `sockaddr_in` structure (defined in IN.H) containing peer's internet address and port number or 0 if none. (See the *sockaddr_in* subsection for the `sockaddr_in` structure definition.)

int tolen

Length of the *to* buffer.

Description

An IO$_WRITEVBLK QIO function is issued on the socket's channel. The IO$M_URGENT modifier is used if MSG_OOB is specified (see *flags*).

If `sendto` is called on an unconnected stream (TCP) socket and you specify a *to* structure, an implicit `connect` is done before sending the data.

Status

Returns `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EAFNOSUPPORT	Address family specified in the *to* argument of the `socket_send/socket_write` routine is not AF_INET.
EBADF	Socket structure is not valid.
ECONNRESET	Connection is reset by the peer.
EINVAL	*tolen* value is invalid (for sendto only).
EIO	Unexpected system status returned during operation.
EMSGSIZE	Buffer size is too large or invalid.
ENETDOWN	Network was shut down.
ENOTCONN	Socket is not connected and no to structure was provided to connect the socket.
EPIPE	Connection was closed or reset.
ETIMEDOUT	Connection timed out.

The `vaxc$errno` variable contains the system service or I/O status code for `ECONNRESET`, `EIO`, `EMSGSIZE`, `ENETDOWN`, `EPIPE`, or `ETIMEDOUT`.

setdomainname / sethostname

Sets the local host's domain name.

Format

status= setdomainname(*name*, *namelen*)

status= sethostname(*name*, *namelen*)

Arguments

int len

Length of the domain name string (or `0` if none available).

char *name

Address of an ASCII (null-terminated) string that contains the domain name.

int namelen

Length of the character string.

Description

These routines require sufficient privileges to define a system logical name because they set the TCPWARE_DOMAINNAME system logical name.

See Also

getdomainname

gethostname

Status

Returns `0` for success and `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EPERM	Returned if the call to SYS$CRELNM fails with the SS$_NOPRIV status.
EVMSERR	Returned if the call to SYS$CRELNM fails with any status other than SS$_NOPRIV.

The `vaxc$errno` variable contains the status code returned by SYS$CRELNM.

setsockopt

Sets socket processing options.

The SOCKET.H file contains definitions for the socket-level options. The IN.H file contains definitions for the IPPROTO_IP level options.

Format

status= setsockopt(*s*, *level*, *optname*, *optval*, *optlen*)

Arguments

int s

Socket descriptor (as returned by `socket`).

int level

SOL_SOCKET to change socket options, or **IPPROTO_IP** to change IP options (which requires system UIC, SYSPRV, or BYPASS privilege).

int optname

Option code to change. See Table A-5 for the SOL_SOCKET option values and Table A-6 for the IPPROTO_IP option values.

char *optval

Address of the value for the option (if the option requires a value).

int optlen

Length of *optval* (if the option requires a value).

Description

The specified socket option is changed.

Status

Returns `0` for success and `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EADDRNOTAVAIL	Address not available for use.
EADDRINUSE	Address already in use.
EBADF	Socket structure is not valid.
EINVAL	Invalid optname is specified or the *optlen* or *optval* arguments are invalid.
ENOBUFS	Insufficient memory for requests.
ETOOMANYREFS	Too many multicast memberships requested.

Table A-5 *Optname* Argument Values for SOL_SOCKET Level

SOL_SOCKET Option	Description
SO_IOCHAN	Obtains the socket's OpenVMS I/O channel number. *optval* is the address of a short to receive the I/O channel number.
SO_RCVTIMEO	Sets the timeout value for datagram (UDP) sockets when the MSG_TIME flag is used in `socket_recv` or `recvfrom` calls. The *optval* option is the address of a short containing the timeout time (in seconds); optlen = sizeof (short). The default value (set by `socket`) is 5 seconds.
SO_REUSEADDR	Enables shared access mode on the local port for datagram (UDP) sockets. The *optval* and *optlen* arguments are ignored.
SO_SNDTIMEO	Sets the timeout value for stream (TCP) sockets. The *optval* option is the address of a short containing the timeout time (in seconds); optlen = sizeof (short). The default value (set by `socket`) is 120 seconds. The minimum value is 20 seconds.

Table A-6 *Optname* Argument Values for IPPROTO_IP Level

IPPROTO_IP Option	Description
IP_OPTIONS (1)	Gets or sets IP options to be sent in subsequent datagrams
IP_TOS (3)	Gets or sets the IP type of service (TOS) to sent in subsequent datagrams
IP_TTL (4)	Gets or sets the IP time-to-live (TTL) to sent in subsequent datagrams
IP_MULTICAST_IF (16)	Gets or sets the interface used for sending multicast datagrams
IP_MULTICAST_TTL (17)	Gets or sets the IP time-to-live (TTL) to sent in subsequent datagrams
IP_MULTICAST_LOOP (18)	Gets or sets whether sent multicast datagrams should be looped back locally
IP_ADD_MEMBERSHIP (19)	Adds a multicast group membership for an interface
IP_DROP_MEMBERSHIP (20)	Drops a multicast group membership from an interface

shutdown

Closes or aborts the connection for a socket.

Format

status= shutdown(*s*, *how*)

Arguments

int s

Socket descriptor (as returned by `socket`).

int how

`0` to close the receive side (ignored for stream sockets), **`1`** to close the sending side (ignored for datagram sockets), and **`2`** to abort the connection.

Description

The operation specified by the how argument is performed.

To close a connection fully, use the `socket_close` subroutine.

Note! Always call the `socket_close` subroutine to clean up a socket.

QIO Function Performed

Depending on the value of the *how* argument, one of the following QIO functions is issued on the channel:

- IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN
- IO$_SETMODE | IO$M_CTRL | IO$M_SHUTDOWN | IO$M_ABORT

See Also

socket_close

Status

Returns `0` for success and `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EBADF	Socket structure is not valid.
EINVAL	Socket is not in a valid state or the value for how is not valid.

socket

Creates a socket structure.

Must be called to allocate a socket structure before calling `bind`, `connect`, `listen`, `accept`, `socket_send` or `sendto`, `socket_recv` or `recvfrom`, `shutdown`, or `socket_close`.

Format

s = socket(*af*, *type*, *protocol*)

Arguments

int s

Socket descriptor or `-1`. This socket descriptor is the address of the socket structure maintained by the Socket Library subroutines. This socket structure is defined in SOCKETVAR.H.

int af

Address family (must be **`AF_INET = 2`**). AF_INET is defined in SOCKET.H.

int type

`SOCK_STREAM` (for TCP), **`SOCK_DGRAM`** (for UDP), or **`SOCK_RAW`** (for IP). These constants are defined in SOCKET.H.

int protocol

Must be **`0`** for SOCK_STREAM and SOCK_DGRAM. For SOCK_RAW, *protocol* is the IP protocol number.

Description

A socket structure is allocated using LIB$GET_VM and added to the linked list of socket structures. The socket structure is initialized and a TCP, UDP, or IP channel is assigned.

The socket structure returned by socket is deallocated by calling the `socket_close` subroutine.

Status

Returns `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EAFNOSUPPORT EINVAL ESOCKTNOSUPPORT	Input parameter is invalid.
ENETDOWN	Call to SYS$ASSIGN failed. `vaxc$errno` contains the status code returned by SYS$ASSIGN.
ENOBUFS	Call to LIB$GET_VM failed. `vaxc$errno` contains the status code returned by LIB$GET_VM.

socket_close

Closes and deallocates a socket.

Note! Under UNIX, the standard `close` subroutine is used. The VAX C `close` subroutine cannot be used with sockets.

Format

status= socket_close(*s*)

Argument

int s

Socket descriptor (as returned by `socket`).

Description

This subroutine closes the connection and receives (and discards) any data not yet read. The channel for the socket is deassigned and the socket structure is then deallocated (using LIB$FREE_VM).

Status

Returns `0` for success and `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EBADF	Socket structure is not valid.

socket_ioctl

Performs several control functions on a socket.

The IOCTL.H file contains definitions for this subroutine.

Format

status= socket_ioctl(*s*, *request*, *argp*)

Arguments

int s

Socket descriptor (as returned by `socket`).

int request

Request code for the control function. Supported request codes are:

Supported Request Codes	Description
FIOASYNC	Sets or clears asynchronous I/O. If *argp*=1, descriptor is set for asynchronous I/O. If 0, descriptor is cleared for asynchronous I/O.
FIONREAD	Returns the number of bytes available for reading on the socket to the longword specified by argp.
SIOCATMARK	Returns whether the stream socket is at the out-of-band mark. If at the out-of-band mark, 1 is returned to the longword specified by argp. Otherwise, 0 is returned.
SIOCDARP **SIOCGARP** **SIOCSARP**	Used to Delete, Get, or Set an ARP table entry. The argp argument points to an *arpreq* structure (see the IF_ARP.H file for the definition of this structure).
SIOCGIFADDR **SIOCGIFBRDADDR** **SIOCGIFDSTADDR** **SIOCGIFFLAGS** **SIOCGIFMETRIC** **SIOCGIFNETMASK** **SIOCGIFMTU**	Used to obtain information about a network The *argp* argument points to a *ifreq* structure (see the IF.H file for the definition of this structure).
SIOCGIFCONF	Used to obtain a list of the available network interfaces. The argp argument points to an *ifconf* structure (see the IF.H file for the definition of this structure).

SIOCADDRT **SIOCDELRT**	Used to Add or Delete a routing table entry. The argp argument points to an *rtentry* structure (see the ROUTE.H file for the definition of this structure).

char *argp

Address of the buffer to receive the information or that contains information, depending on the request.

Status

Returns `0` for success and `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EBADF	Socket structure is not valid.
EINVAL	Input request or buffer address is invalid.

socket_read / socket_recv

Read data from a connected socket.

Note that under UNIX, `socket_read` is called `read` and `socket_recv` is called `recv`.

Format

msglen = socket_read(*s*, *buffer*, *buflen*)

msglen = socket_recv(*s*, *buffer*, *buflen*, *flags*)

Arguments

int msglen

`-1` for error, `0` for connection closed by peer and no more data available, `>0` for the number of bytes of data read.

int s

Socket descriptor (as returned by `socket`).

char *buffer

Address of the buffer into which the data is to be received.

int buflen

Length of the buffer into which data is to be received.

int flags

Flag bits for operation. The supported flag bits are:

Flag Bit	Description
MSG_NONBLOCKING	returns immediately if no data is available (EWOULDBLOCK is returned in `errno` if no data is available).
MSG_PEEK	lets you peek at the data without removing it from the data stream.
MSG_TRUNCATE	returns truncated datagrams (otherwise ENOMEM is returned if the datagram is larger than the buffer). This flag is only used by datagram (UDP) sockets.
MSG_TIME	limits the time to wait for a datagram to be received. The time limit for the receive is set using the `setsockopt` subroutine. This flag is only used by datagram (UDP) sockets.

All other flags are ignored.

QIO Function Performed

- IO$_READVBLK on the socket's channel
- IO$M_NOW modifier if specifying MSG_NONBLOCKING
- IO$M_DATACHECK modifier if specifying MSG_PEEK (see *flags*)

See Also

`select`, if your application handles multiple connections simultaneously

Status

Returns `-1` for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EBADF	Socket structure is not valid.
ECONNRESET	Connection is reset by the peer.
EIO	Unexpected system status returned during operation.
ENETDOWN	Network was shut down.
ENOMEM	Buffer was too small for the datagram and MSG_TRUNCATE was not specified. Note that the truncated datagram is in the buffer.
ENOTCONN	Socket is not connected.
ETIMEDOUT	Connection timed-out or the request timed-out (if MSG_TIME set for a datagram socket).
EWOULDBLOCK	MSG_NONBLOCKING flag is set and no data is available.

The `vaxc$errno` variable contains the system service or I/O status code for `ECONNRESET`, `EIO`, `ENETDOWN`, `ETIMEDOUT`, or `EWOULDBLOCK`.

socket_send / socket_write

Send data over a connected socket.

Note! Under UNIX, `socket_send` is called `send` and `socket_write` is called `write`.

Format

cc = socket_send(*s*, *buffer*, *buflen*, *flags*)

cc = socket_write(*s*, *buffer*, *buflen*)

Arguments

int cc

Number of bytes sent or `-1` for failure.

int s

Socket descriptor (as returned by socket).

char *buffer

Address of the buffer which contains the data to be sent.

int buflen

Length of the buffer which contains the data to be sent.

int flags

Flag bits for operation. The supported flag bits are **MSG_OOB** to send urgent data (for stream sockets only). All other flags are ignored.

QIO Function Performed

- IO$_WRITEVBLK is issued on the socket's channel
- IO$M_URGENT modifier is used if MSG_OOB is specified (see *flags*)

Status

Returns -1 for failure. For a failure status, the `errno` variable contains the reason for the error as follows:

EAFNOSUPPORT	Address family specified in the *to* argument of the `sendto` routine is not AF_INET.
EBADF	Socket structure is not valid.
ECONNRESET	Connection is reset by the peer.
EIO	Unexpected system status returned during operation.
EMSGSIZE	Buffer size is too large or invalid.
ENETDOWN	Network was shut down.

ENOTCONN	Socket is not connected and no to structure was provided to connect the socket.
EPIPE	Connection was closed or reset.
ETIMEDOUT	Connection timed out.

The `vaxc$errno` variable contains the system service or I/O status code for `ECONNRESET`, `EIO`, `EMSGSIZE`, `ENETDOWN`, `EPIPE`, or `ETIMEDOUT`.

tcpware_atolineid

Returns the numeric TCPware line ID for the ASCII (null-terminated) string.

Format

lineid =tcpware_atolineid(*name*)

Arguments

unsigned long lineid

Numeric line ID of the character string or 0 if the character string is not a valid line ID. For a description of the line ID values, see the description of IO$_SENSEMODE | IO$M_CTRL in the *IPDRIVER Services* chapter in this guide.

char *name

Line ID character string. For a description of valid TCPware line IDs, see the description of START/IP in the *NETCU Command Reference*.

tcpware_gettimezone

Returns the current timezone information.

Format

status = tcpware_gettimezone(*seconds*, *name*)

Arguments

int *seconds

Longword into which to return the offset, in seconds, from universal time.

char **name

Address to which to return the address of the time zone string.

See Also

tcpware_settimezone

Status

A standard OpenVMS status code is returned to the caller.

tcpware_lineidtoa

Converts the numeric line ID (*lineid*) into an ASCII (null-terminated) string.

Format

retval = tcpware_lineidtoa(*lineid*, *name*)

Arguments

char *retval

Address of ASCII string corresponding to *lineid* (address of *name* string).

unsigned long lineid

Numeric line ID to be converted into string format.

char *name

Address of a character string in which `tcpware_lineidtoa`returns the line ID. This string should be at least 16 characters in length.

tcpware_server

Gets the I/O channel number or creates a socket for a server connection (TCP or UDP) that was initiated by the master server (NETCP).

Format

status = tcpware_server(*option*, *argument*)

Arguments

int option

If *option* equals...	The subroutine returns a...
`1`	channel number in *argument*.
`2`	socket descriptor in *argument*.

int *argument

Address of an integer that receives a channel or socket number.

Description

Server processes that are created by the master server (NETCP) call this subroutine. Provides the server process with the socket descriptor or OpenVMS I/O channel number that is associated with the connection. Obtains these values from NETCP through mailbox communication.

Status

Returns `0` for success or `-1` for failure. For a failure status, the `errno` variable describes the reason for the error as follows:

EINVAL	Option argument is invalid.
EIO	Unexpected system status returned during operation.
ENODEV	NETCP did not create the server process.

tcpware_settimezone

Sets the current time zone information.

Requires SYSNAM and OPER privileges.

Format

status = tcpware_settimezone(*seconds*, *name*)

Arguments

int seconds

Offset, in seconds, from universal time.

char *name

Address of the time zone string. Specify `0` if no time zone name.

Description

This routine defines the TCPWARE_TIMEZONE system logical name. In addition, `tcpware_settimezone` also sets the IPDRIVER's universal time (UT) offset value.

The TCPWARE_TIMEZONE logical can have two equivalence strings:

4. *+hhmmss*	*hh* are the hours *mm* are the minutes *ss* are the seconds offset from the universal time (UT). ***Note!*** `+` is for east of the central meridian, `-` is for west. For example: `+04:00:00` is four hours east of the central meridian at Greenwich. Another example: eastern standard time (EST) is five hours west of UT, so the offset is `-0500`.
5. *name*	an optional name for the time zone. For example: EDT for Eastern Daylight time. Can be one of the following: Universal Time—UT, UTC or GMT North American Time—EST, EDT, CST, CDT, MST, MDT, PST, PDT Military Time—Any single uppercase letter A through Z except J (this format is not recommended) Any other character sequence The *name* is not validated and may be used by applications to report the local time zone.

Sample Discard Protocol Programs

These programs are in TCPWARE_COMMON:[TCPWARE.EXAMPLES]DISCARD.C and DISCARDD.C. The latter calls the `tcpware_server` subroutine to implement the Discard Protocol (DISCARDD). Under this protocol, the server listens for a connection on TCP port 9. Once the server establishes a connection, it throws away any data it receives. It does not send a response. This continues until the client closes the connection.

Example A-2 shows the NETCU command you can use to enable DISCARDD.

Example A-2 NETCU Command to Enable DISCARDD

```
ADD SERVICE DISCARD TCP TCPWARE:DISCARDD-
     /PROCESS_NAME=DISCARDD-
     /NOACCOUNTING-
     /NOAUTHORIZE-
     /INPUT=NLA0:-
     /OUTPUT=NLA0:-
     /ERROR=NLA0:-
     /UIC=[SYSTEM]-
     /AST_LIMIT=10-
     /BUFFER_LIMIT=10240-
     /ENQUEUE_LIMIT=6-
     /EXTENT=500-
     /FILE_LIMIT=20-
     /IO_BUFFERED=6-
     /IO_DIRECT=6-
     /MAXIMUM_WORKING_SET=300-
     /PAGE_FILE=10000-
     /PRIORITY=4-
     /PRIVILEGES=(NOSAME,NETMBX,TMPMBX)-
     /QUEUE_LIMIT=8-
     /WORKING_SET=200-
     /SUBPROCESS_LIMIT=0
```

Chapter 2 in the *NETCU Command Reference* explains how to use the ADD SERVICE command.

For an additional program using the `tcpware_server` subroutine, see the TCPWARE_COMMON:[TCPWARE.EXAMPLES]DAYTIMED.C file.