VSI OpenVMS

VSI TCP/IP Version 10.6 Release Notes

August 2019

This document describes features and release notes for VSI TCP/IP for OpenVMS Version 10.6.

Preface

1. Intended Audience

This document is intended for all users of VSI TCP/IP V10.5 or V10.6. Read this document, as well as those in the following list, before you install this product on your VSI OpenVMS system:

- *VSI TCP/IP Version 10.6 Installation, Upgrade, and Quick Start Guide*
- *VSI TCP/IP for OpenVMS User's Guide*
- *VSI TCP/IP for OpenVMS Software Product Description (SPD)*

2. Document Structure

This document contains the following chapters:

- *Prerequisites*: Products and configuration conditions required by VSI TCP/IP V10.6.
- *New* Features, *Improvements, and Behaviors:* New features and improvements added to VSI TCP/IP Version 10.6.
- *Deprecated Commands and Services:* A listing of items removed or deprecated in VSI TCP/IP V10.6
- *Future Features:* A list of features planned for future versions of VSI TCP/IP
- *VSI TCP/IP V10.6 Issues*: Release notes introduced in the current release. A subheading for each release note indicates the version of origin (for example, *Version 10.5*).
- *Layered Products and Tools Release Notes*: Release notes connected to VSI TCP/IP layered products and tools.
- *Documentation Errata*: Errors inadvertently included in the TCP/IP V10.5 documentation.
- *Using the $QIO System Service*: This chapter describes how to use the $QIO system service and its data structures with VSI TCP/IP. This information is included in the *VSI TCP/IP Programmer Reference Manual*.

3. VSI TCP/IP V10.6 Documentation

The complete set of VSI TCP/IP V10.6 documentation is available on the web at: http://www.vmssoftware.com/documents_list.html.

In the main window, navigate to **VSI OpenVMS Compilers and Layered Products Documentation** and then scroll to the **VSI TCP/IP Version 10.6** section. The following documents are available in PDF format:

VSI TCP/IP Version 10.6 Release Notes
VSI TCP/IP V10.6 Installation, Upgrade, and Quick Start Guide
VSI TCP/IP User's Guide
VSI TCP/IP Administrator's Guide: Volume I
VSI TCP/IP Administrator's Guide: Volume II
VSI TCP/IP Administrator's Reference
VSI TCP/IP Messages, Logicals, & DECnet Applications
VSI TCP/IP Programmer's Reference
VSI TCP/IP Version 10.6 Cover Letter
VSI TCP/IP Version 10.6 Software Product Description

Chapter 1: Prerequisites

Please note the following prerequisites for VSI TCP/IP Version 10.5 or V10.6:

1. For VSI TCP/IP V10.6, you can perform a fresh installation, or you can upgrade from VSI TCP/IP V10.5.

2. You can run VSI TCP/IP on VSI OpenVMS Integrity Version 8.4-2L1 or higher.

3. VSI TCP/IP requires the use of ODS-5 system disks. If you attempt an installation on a system disk that is not ODS-5, you will see a message similar to the following:

   ```
   VSI TCP/IP requires installation on an ODS-5 system disk.
   The disk on which you are installing, $1$DGA150:, is not an ODS-5
   disk.
   ```

4. VSI TCP/IP V10.6 requires the installation of an OpenVMS patch kit, VMS842L1I_CLUCONFIG-V0100.PCSI$COMPRESSED, which is distributed in the same directory as the VSI TCP/IP V10.6 kit.

   If this patch kit is not installed first, or at the same time as VSI TCP/IP, the VSI TCP/IP V10.5 or V10.6 installation will fail.

Chapter 2: New Features, Improvements, and Behavior

This chapter contains the following information about VSI TCP/IP V10.6 including:

- New features
- Improvements and fixes that have been added to VSI TCP/IP V10.6 since VSI TCP/IP V10.5
- Compatibility and differences between VSI TCP/IP 10.6 and HPE TCP/IP Services

New Features

1. New MultiNet Configuration File Converter Added to VSI TCP/IP

If you have MultiNet installed on your system, you can import your existing MultiNet configuration files to be used by VSI TCP/IP. The conversion can be run during the installation procedure.

See the *VSI TCP/IP Version 10.6 Installation, Upgrade, and Quick Start Guide, Chapter 2* for detailed information about the configuration file conversion process.

2. Updated Programs in IP$EXAMPLES

The `IP$EXAMPLES` directory contains updated programs that demonstrate aspects of network programming that you may find helpful. Please read the `SOCKET-README.TXT` file in the `IP$EXAMPLES:` directory for a complete list and instructions for how to use these programs.

3. Updated BIND 9 support

VSI TCPIP V10.6 has been updated to the following BIND 9 version: 9.11.8. BIND 9.11.8 contains CVE fixes. Further information on the contents of BIND 9.11.8 may be found at:
https://ftp.isc.org/isc/bind9/9.11.8/RELEASE-NOTES-bind-9.11.8.html

4. SSH Port Forwarding and OpenVMS Captive Users

SSH implements a user group, known internally by SSH, which designates the users of captive accounts. This group, `IP$SSH_CAPTIVE_USERS`, gives the system administrator a method by which to specify captive users in various aspects of SSH configuration without requiring definition and management of OpenVMS rights identifiers. Additionally, the supplied SSH configuration template, `SSHD2_CONFIG.TEMPLATE`, disables port forwarding for captive users by default.

To enable the SSH port forwarding feature for captive users, remove the line `"DenyTcpForwardingForGroups"` from the `SSHD2_CONFIG.CONF`, which can be found in `SYS$SPECIFIC:[IP.CONFIG.SSH2]`.

5. New Key Exchange Algorithms Added to SSH

VSI TCP/IP V10.6 contains new Key Exchange algorithms. The following KEX algorithms are now supported:

```
diffie-hellman-group14-sha256
diffie-hellman-group-exchange-sha1
diffie-hellman-group-exchange-sha256
```

6. VSI TCP/IP SNMP Supports Interrogation of MIBs

In VSI TCP/IP V10.6, SNMP supports integration of the following MIBs. Support for additional MIBs will be added in a future release of VSI TCP/IP.

Supported MIBs for V10.6:

iso.org.dod.internet.mgmt.mib_2.system
iso.org.dod.internet.mgmt.mib_2.system.sysDescr
iso.org.dod.internet.mgmt.mib_2.system.sysOID
iso.org.dod.internet.mgmt.mib_2.system.sysUpTime
iso.org.dod.internet.mgmt.mib_2.system.sysContact
iso.org.dod.internet.mgmt.mib_2.system.sysName
iso.org.dod.internet.mgmt.mib_2.system.sysLocation
iso.org.dod.internet.mgmt.mib_2.system.sysServices
iso.org.dod.internet.mgmt.mib_2.system.sysORLastChange
iso.org.dod.internet.mgmt.mib_2.system.sysORTable
iso.org.dod.internet.mgmt.mib_2.system.sysORTable.sysOREntry
iso.org.dod.internet.mgmt.mib_2.system.sysORTable.sysOREntry.sysORIndex
iso.org.dod.internet.mgmt.mib_2.system.sysORTable.sysOREntry.sysORID
iso.org.dod.internet.mgmt.mib_2.system.sysORTable.sysOREntry.sysORDescr
iso.org.dod.internet.mgmt.mib_2.system.sysORTable.sysOREntry.sysORUpTime
iso.org.dod.internet.mgmt.mib_2.interfaces
iso.org.dod.internet.mgmt.mib_2.interfaces.ifNumber
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifIndex
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifDescr
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifType
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifMtu
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifSpeed
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifPhysAddress
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifAdminStatus
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifOperStatus
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifLastChange
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifInOctets
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifInUcastPkts
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifInNUcastPkts
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifInDiscards
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifInErrors
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifInUnknownProtos
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifOutOctets
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifOutUcastPkts
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifOutNUcastPkts
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifOutDiscards
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifOutErrors

iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifOutQLen
iso.org.dod.internet.mgmt.mib_2.interfaces.ifTable.ifEntry.ifSpecific
iso.org.dod.internet.mgmt.mib_2.at
iso.org.dod.internet.mgmt.mib_2.at.atable
iso.org.dod.internet.mgmt.mib_2.at.atable.atEntry
iso.org.dod.internet.mgmt.mib_2.at.atable.atEntry.atlFIndex
iso.org.dod.internet.mgmt.mib_2.at.atable.atEntry.atPhysAddress
iso.org.dod.internet.mgmt.mib_2.at.atable.atEntry.atNetAddress
iso.org.dod.internet.mgmt.mib_2.ip
iso.org.dod.internet.mgmt.mib_2.icmp
iso.org.dod.internet.mgmt.mib_2.tcp
iso.org.dod.internet.mgmt.mib_2.udp
iso.org.dod.internet.mgmt.mib_2.snmp

VSI TCP/IP V10.5 Problems Fixed

This section lists problems that were originally documented in the *VSI TCP/IP Version 10.5 Release Notes*. These problems have been fixed in VSI TCP/IP Version 10.6.

1. IP SHOW/REMOTE May Result in Forced Exit

When the target host specified by an IP SHOW/REMOTE command is not reachable, or the connection is refused because the target does not have the NETSTAT service enabled, a forced exit will occur indicating an improperly handled condition.

This problem has been fixed in VSI TCP/IP Version 10.6.

2. Node Hangs When Booting into a Cluster

Removal or installation of TCP/IP Services can result in the deletion of some IPCI files, causing a node to hang when it attempts to boot into a cluster. Please see the note "*Other Networking Products*" in the *Layered Products and Tools Release Notes* section for information on how you can restore the IPCI configuration information.

This problem has been fixed in VSI TCP/IP Version 10.6.

3. VSI T4 INET Collection Call Fails with VSI TCP/IP

The VSI T4 tool's INET collection fails when used with VSI TCP/IP V10.5. As a result, Internet data is not collected. This will be addressed in a future update to VSI T4.

This problem has been fixed in VSI TCP/IP Version 10.6.

4. VSI TDC (The Data Collector) INET Collection Fails with VSI TCP/IP

The Data Collector (TDC) INET collection fails when used with VSI TCP/IP V10.5. As a result, Internet data is not collected. This will be addressed in a future update to VSI TDC.

This problem has been fixed in VSI TCP/IP Version 10.6.

5. VSI TCP/IP and TCP/IP Services Compatibility Fixed

The following compatibility issues have been resolved in VSI TCP/IP V10.6.

1. The socket option SO_SHARE is implemented in VSI TCP/IP V10.6.
2. The TCP/IP Services TELNET API is implemented in VSI TCP/IP V10.6.

For existing compatibility and differences, see the following section.

VSI TCP/IP and TCP/IP Services Compatibility and Differences in Behavior

Compatibility

VSI is investigating compatibility issues between the current TCP/IP Services network stack offering and VSI's TCP/IP V10.6. We encourage customers to provide feedback on the impact of the issues noted below and notify us of any others that are encountered. Please return feedback by sending email to VSI at support@vmssoftware.com.

1. The API if_indexname(), which retrieves information about the network interfaces configured in the system, does not include lo0 in the list of interfaces it returns.
2. When features are available via both the standard socket interface and QIO, VSI recommends use of the standard socket interface. For example, use of $QIO in VSI TCP/IP V10.5, or V10.6 might result in status return values that indicate failure; these values may not match the values returned by TCP/IP Services.
3. An HPE TCP/IP Services FTP client issuing a get of an index file from a VSI TCP/IP FTP server hangs. This is due to HPE TCP/IP Services FTP client and VSI TCP/IP FTP server identifying each other as VMS through the command SITE VMS. To work around this issue, enter the following command prior to issuing the get command:

```
FTP> disable vms
```

Differences in Behavior

VSI has observed the following differences in behavior between VSI TCP/IP and TCP/IP Services:

1. **COPY/FTP Exhibits Incorrect Behavior When Remote Files are Copied Using Wildcard File Specification**.

   When you use COPY/FTP to copy files from systems running TCP/IP Services and use a wildcard in the source file specification, the output files on the local system will have exactly the same version number as the file on the remote system. Here is an example of a command that results in incorrect behavior:

   ```
   copy/ftp node"user password"::login.*;* []  ...
   ```

   If files with the same name exist on the local system, the output files might have lower version numbers than the files on the local system, or they could collide with the existing file version(s).

2. **COPY/FTP of Remote File to Wildcard Local File Specification Fails**.

   When you use COPY/FTP to copy a remote file to a local destination and use a wildcard file specification for the output file, the copy will fail. Here is an example of a command that will fail:

   ```
   copy/ftp node"user password"::login.com *
   ```

   To work around this issue, specify the target directory without the wildcard file specification, as shown in this example:

   ```
   copy/ftp node"user password"::login.com []
   ```

3. **COPY/FTP May Change File Attributes When Copying Between VSI TCP/IP V10.5 or V10.6 and TCP/IP Services**.

   When copying files between systems running VSI TCP/IP V10.5 or V10.6 and those running HPE TCP/IP Services, the following file attributes may change: file organization, record attributes, and record format. To work around this issue, use FTP directly, rather than COPY/FTP, and transfer a backup save set containing the files in image mode.

4. **COPY/FTP Source File Search List Logical Interaction**.

   When using COPY/FTP from a system running VSI TCP/IP, more than one file will be copied if the source file specification references a search list logical (like `SYS$SYSROOT`) and copies of the desired file exist in more than one place in the search list.

5. **COPY/FTP Syntax Restriction**

   COPY/FTP improperly handles several forms of directory specifications, such as unmerged rooted directory specifications and directory specifications delimited by angle brackets < >. These are shown in the following examples:

   ```
   COPY/FTP [ROOT.][DIRECTORY]FOO.TXT host"username password"::
   COPY/FTP <DIRECTORY>FOO.TXT hostname"username password"::
   ```

   To work around the issue in the first example, remove the characters .][from the directory specification. In the second example, replace the angle brackets < > with brackets [ ].

6. **COPY/FTP Wildcard Input Files Are Copied to The Output Default Directory**.

   When a wildcard (*) is used to specify the input files to the command COPY/FTP, the specified destination directory is ignored and the files are copied to the default directory instead. In the case of a local destination, this will be the current default directory for the user. In the case of a remote destination, this will be the login default for the target user account (unless changed by login.com on the target host).

7. SCP Command Failures Observed Between Networking Stacks

The following failures have been observed when using the SCP command between TCP/IP Services V5.7 and VSI TCP/IP V10.5 or V10.6. These failures will be addressed in a future release.

1. The TCP/IP Services V5.7 SCP command fails when attempting to transfer files to a VSI TCP/IP V10.5 or V10.6 server if the remote filename uses OpenVMS syntax.

   VSI TCP/IP V10.5 and V10.6 do not recognize TCP/IP Services V5.7 as an OpenVMS TCP/IP implementation. As a result, you must issue the SCP command using UNIX-style syntax for the remote file. OpenVMS syntax in remote filenames can be used when the server and the client both run VSI TCP/IP.

2. The VSI TCP/IP SCP command fails to transfer remote files if the filename includes wildcards.

   To successfully transfer files using SCP, surround the parts of the filename that are not wildcards and periods in double quotes. In the case of ODS2 files, the filename must be uppercase.

   Here are examples of commands that fail:

```
$ scp "system@blade1"::*.pl []
$ scp "system@blade1"::*.PL []
```

   Here are examples of commands that succeed:

```
$ scp "system@blade1"::*."PL" []
$ scp "system@blade1"::”D”*."PL" []
```

Chapter 3: Deprecated Commands and Services

This chapter lists commands and services that VSI has determined are either obsolete or are not used by modern TCP/IP stacks.

1. Interfaces No Longer Supported

VSI TCP/IP no longer supports Novell's EXOS programming interface. Additionally, VSI TCP/IP no longer supports configuring Ethernet interfaces with Novell raw IEEE 802.3, IEEE 802.2 LLC and IEEE 802.2 SNAP Ethernet frames.

2. Commands No Longer Supported

VSI TCP/IP no longer supports the `IP LPRM` command.

If the print job is still on your local system, you should use the VMS command DELETE /ENTRY to delete the job.

3. Commands and Services Not Present in VSI TCP/IP V10.5 or V10.6.

VSI TCP/IP V10.5 and V10.6 do not include the following commands and services found in MultiNet.

- Commands not present: DECODE, KERBEROS, MENU, RDATE
- Service names changed from those in MultiNet: NFS (in MultiNet, it was NFS3)
- Services not present: BOOTP, BWNFS, EKLOGIN, KADMIN, KLOGIN, KSHELL, NFS2, PCNFS, POP2, REMIND, TFTP

Note: BOOTP and TFTP will be added in a future release of VSI TCP/IP.

4. MAIL Facilities (SMTP, POP, IMAP) Lack Modern Capabilities

Mail support contained in VSI TCP/IP V10.5 lacks the security, virtualization, and spam avoidance features found in modern mail systems. If you require a production quality mail solution with VSI TCP/IP, we recommend that you use PMDF, a complete and up-to-date messaging product available from Process Software. PMDF is a high performance standards-based Internet product suite that has been tested on VSI TCP/IP V10.5 and V10.6.

Chapter 4: Future Features

VSI understands that there are important network stack features that need to be carefully implemented and fully tested before those features are deployed in critical customer environments. The following network stack features are not present in the VSI TCPIP V10.6 release.

These features will be added in a future release. Please contact VSI Support at if you have specific requirements for these features:

1. Clusters Over IP Not Recommended

VSI has observed problems in a cluster when SCS communications occur over the TCP/IP stack. This configuration is commonly known as OpenVMS Clusters over IP. Currently, VSI does not recommend using Clusters over IP with VSI TCP/IP in a production environment.

2. Issues with Jumbo Frames

VSI has encountered issues with jumbo frames enabled and recommends against enabling jumbo frames on any VSI TCP/IP V10.5 or V10.6 interfaces.

3. Satellite Cluster Nodes Cannot Be Booted

It is not possible to boot satellite cluster nodes from a system running VSI TCP/IP V10.5 or V10.6 because the BOOTP and TFTP services are not supported in these versions. This will be corrected in a future release of VSI TCP/IP.

Chapter 5: Current VSI TCP/IP V10.6 Issues

Current issues and release notes pertaining to both VSI TCP/IP Versions 10.5 and 10.6. A subheading for each release note indicates the version of origin (for example, *Version 10.5*). Refer to Chapter 2 for issues that VSI Engineering has fixed in Version 10.6.

1. NFSMOUNT with /TRANSPORT=TCP Results in Some UDP Packets

Version 10.6

Using tcpdump to monitor NFSMOUNT activity, the following has been observed:

```
10:45:23.80 16.1.1.203.601 > 16.1.1.133.111: udp 88
10:45:23.80 16.1.1.133.111 > 16.1.1.203.601: udp 28 (DF)
10:45:23.80 16.1.1.203.601 > 16.1.1.133.20048: udp 84
10:45:23.81 16.1.1.133.20048 > 16.1.1.203.601: udp 28 (DF)
```

However, the rest of NFS activity afterwards is done using TCP protocol.

This is due to limitations on MOUNT services on VSI TCP/IP only supporting UDP. In order to successfully mount remote shares, remote NFS server must allow UDP mounts.

2. NFSDISMOUNT /LOG Qualifier Not Functioning Correctly

Version 10.6

The /LOG qualifier cannot be used with the NFSDISMOUNT command. Issuing this command generates the following system message:

```
$ NFSDISMOUNT NFS3: /LOG
%DCL-W-IVQUAL, unrecognized qualifier - check validity, spelling,
and placement\LOG\
```

This will be corrected in a future release of VSI TCP/IP.

3. Branding

Version 10.5

Copyright notices have been modified from Process Software's MultiNet to VSI TCP/IP. However, text that appears in some help files, display screens, or other areas may still say MultiNet. These will be updated in a future release of VSI TCP/IP.

4. DHCLIENT Use and Network Topologies

Version 10.5

DHCLIENT provides centralized assignment and management of IP addresses in conjunction with simple network topologies. However, VSI has observed problems with topologies that require definition of static routes that are outside of routes that can be supplied by DHCP. Customers with such use cases should contact VSI support at Support@vmssoftware.com for possible workarounds.

5. IP CONFIG/NFS SHOW MOUNT Does Not Show Client Host Mounts

Version 10.5

The SHOW MOUNT command of the NFS configuration utility does not work, producing no output regardless of the state of any NFS clients mounting exports on the local server.

6. Changing the LPD Spool Directory

Version 10.5

By default, LPD print jobs (and SMTP mail messages) on the OpenVMS system are stored in the directory `IP$COMMON_ROOT:[IP.SPOOL]`. You can change the directory with the NET-CONFIG SET SPOOL- DIRECTORY command by entering:

```
$ IP CONFIGURE
NET-CONFIG>SET SPOOL-DIRECTORY DISK$TEMP:[IP]
```

You must redefine the logical that points to the spooling area unless you reboot the system after modifying the VSI TCP/IP configuration by entering:

```
$ DEFINE/SYSTEM/EXEC IP$SPOOL DISK$TEMP:[IP]
```

Make sure the directory protections are set to SYSTEM:RWED, OWNER:RWED, GROUP:RE, and WORLD:RE.

7. Network DCL Commands May No Longer Function

Version 10.5

Removal or installation of TCP/IP Services can result in the deletion of some network DCL commands. Please see the note "*Other Networking Products*" in the *Layered Products and Tools Release Notes* section for information on how you can restore the commands to your system.

8. SSH Configuration Must Be Performed From SYSTEM Account

Version 10.5

VSI recommends that you configure SSH from the SYSTEM account only. Non-system accounts do not configure SSH correctly.

9. SSH Connections Are Not Logged When SSHD.log Files Reach Maximum Version Number

Version 10.5

When the SSHD.LOG file version reaches the maximum number of 32767, new log files are not generated. SSH connections still function, however, VSI recommends that you rename the `SSHD.LOG` to fix the problem.

You can change the log file name by defining the logical name **`IP$SSH_LOG_FILE`** with one or more of the following tokens:

```
%D  date in yyyymmdd format
%N  system SCS node name
%C  value of childcount
```

For example, defining the following logical name:

```
$ DEFINE /SYSTEM IP$SSH_LOG_FILE "SSH_%N_%D"
```

results in the following SSH log file names:

```
IP$LOG:SSH_MYNODE_20190513.LOG
```

VSI also recommends that files be removed regularly to allow space for new file creation.

10. SSH UNIX-style Options Not Available

Version 10.5

SSH does not offer the Unix-style options -h and -v, which display help and version information respectively. Use the /HELP and /VERSION qualifiers instead.

```
$ SSH /HELP
$ SSH /VERSION
```

11. NTPTRACE Not Supported

Version 10.5

VSI TCP/IP does not support NTPTRACE. VSI recommends that you use NTPQ instead. NTPQ is fully documented in the *VSI TCP/IP Administrator's Guide: Volume I, Using NTPQ* and will produce similar but not exactly the same results as NTPTRACE.

Chapter 6: Layered Products and Tools Release Notes

This section contains release notes for layered products and tools that interact with VSI TCP/IP.

1. **Common Internet File System (CIFS)**

The currently released version of VSI CIFS does not recognize VSI TCP/IP V10.5 or V10.6 as a supported network stack. In order to run VSI CIFS with VSI TCP/IP V10.5 or V10.6, you must update the CIFS startup procedures. Download the kit VSI-I64VMS-SAMBA-V0102-ECO1C-2.ZIPEXE using SFTP (required) from this location:

```
SFTP Server: vsiftp.vmssoftware.com (104.207.199.163)
Username:    OPENKITS
Password:    VSI#14kits
Directory:   i64opensource
Filename:    VSI-I64VMS-SAMBA-V0102-ECO1C-2.ZIPEXE
```

Unpack the kit on an OpenVMS host using the following command:

```
$ RUN VSI-I64VMS-SAMBA-V0102-ECO1C-2.ZIPEXE
```

The kit `VSI-I64VMS-SAMBA-V0102-ECO1C-2.ZIPEXE` contains `VSI-I64VMS-SAMBA-V0102-ECO1C-2.RELEASE_NOTES` (containing installation instructions) and the new CIFS startup files that recognize VSI TCP/IP V10.5 or V10.6.

Note:

- If you previously installed CIFS patch PS2_14, you may apply `VSI-I64VMS-SAMBA-V0102-ECO1C-2.ZIPEXE` on top of PS2_14.
- If you ***have not yet*** installed CIFS patch PS2_14 but intend to do so, install PS2_14 ***before*** installing `VSI-I64VMS-SAMBA-V0102-ECO1C-2.ZIPEXE`.

2. **Other Networking Products**

The network state and environment used by OpenVMS has several components that may overlap if multiple network products are installed on your system. After installing VSI TCP/IP V10.5 or V10.6, removing or installing another network product can render certain portions of the VSI TCP/IP network environment unusable. For example, removing or installing the TCP/IP Services product or Process Software's MultiNet product can result in these symptoms:

- **Network commands in DCL may no longer function**: Various network commands that are shared between products may be deleted or changed within the DCLTABLES on the system. These commands include: FINGER, FTP, RCP, RLOGIN, RPCGEN, RSHELL, SCP2, SFTP2, SSH, SSH2, and TELNET.
- **Configuration information for Clusters over IP via IPCI may be deleted**: If you use IPCI to enable clustering over IP for your VMScluster, the cluster's network configuration can be lost when another product is installed or deleted. As a result, the node will be unable to boot into the cluster since it cannot find the other cluster members via IP.

To avoid these symptoms, VSI provides a procedure to restore the VSI TCP/IP state that is overlapped and potentially removed. After installing or removing another network product, use the following command to restore the environment so that it will continue to use VSI TCP/IP:

```
$ @SYS$MANAGER:IP$FIX_VSI_TCPIP
```

This will restore the VSI TCP/IP network commands and IPCI configuration information. VSI strongly recommends that you invoke this procedure immediately after installing or removing another network product if you want to continue using VSI TCP/IP.

Note: Failure to run this procedure can result in a boot hang at some later time if you use IPCI for clustering over IP.

3. **TCP/IP Services Fails to Start After Installation of VSI TCP/IP V10.5 or V10.6**

If you try to start TCP/IP Services by running TCPIP$STARTUP after you have installed VSI TCP/IP V10.5 or V10.6, you will see the following error message:

```
$ @SYS$STARTUP:TCPIP$STARTUP
%TCPIP-E-STARTFAIL, failed to start TCP/IP Services
-TCPIP-E-NOLICENSE, license check failed
```

To resume use of TCP/IP Services after you have installed VSI TCP/IP V10.5 or V10.6, run the procedure `@SYS$MANAGER:IP$SET_STACK` to switch stacks. You must also use `IP$STARTUP` at boot time instead of using TCPIP$STARTUP. For additional information, see the *VSI TCP/IP for OpenVMS Installation, Upgrade, and Quick Start Guide*.

4. **VSI DCE Compatibility with VSI TCP/IP V10.5 or V10.6**

In order to run VSI DCE with VSI TCP/IP V10.5 or V10.6, you must acquire VSI DCE Version 3.2D. Previous versions of VSI DCE V3.2 do not recognize VSI TCP/IP V10.5 or V10.6 as a valid network stack.

To request the kit, VSI direct support customers should send email to Support@vmssoftware.com and reference their VSI support contract number in the email. HPE support customers should access the HPE support site or contact their HPE support representative to request the kit.

5. **VSI WBEM Services (WBEMCIM) Installation Requirement**

Installation of VSI WBEM Services (WBEMCIM) V3.0-C180108 or later for VSI OpenVMS Integrity Servers is required for compatibility with VSI TCP/IP V10.5 or V10.6. VSI highly recommends that you install the WBEMCIM kit prior to installing VSI TCP/IP V10.5 or V10.6.

If WBEM Services V3.0-C180108 is not installed, the following messages are displayed by the WBEM services configuration, startup, and shutdown procedures.

```
LCKHVN>> @sys$startup:wbem_services$startup
%SYSTEM-F-ABORT, abort
%RUN-S-PROC_ID, identification of created process is 23800460
%SYSTEM-F-ABORT, abort
%WBEMCIM-I-SERVERWAIT, Waiting for CIMServer to start. 180 seconds
remaining...
%SYSTEM-F-ABORT, abort
%WBEMCIM-I-SERVERWAIT, Waiting for CIMServer to start. 170 seconds
remaining...
```

VSI direct support customers can send email to L1-support@vmssoftware.com to obtain access to the kit. Please reference your VSI support contract number in the email. HPE direct support customers should access the kit via the HPE support site.

Chapter 7: Documentation Errata

The following release notes pertain to errors inadvertently included in the VSI TCP/IP V10.5 documentation.

1. Corrected SERVER-CONFIG Commands

Version 10.6

In *VSI TCP/IP Administrator's Guide: Volume I*, *Table 2.10 SERVER-CONFIG Commands*, the following commands have been corrected:

Incorrect Command	Correct Command
`SET RECEIVE-BUFFER-SIZE`	`SET RECEIVE-BUFFER-SPACE`
`SET SEND-BUFFER-SIZE`	`SET SEND-BUFFER-SPACE`

2. Corrected DOMAIN Commands Syntax

Version 10.6

In *VSI TCP/IP Administrator's Guide: Volume I*, the syntax of the following commands has been corrected:

Incorrect Syntax	Correct Syntax
`$ IP NETCONTROL DOMAIN RELOAD`	`$ IP NETCONTROL DOMAINNAME RELOAD`
`$ IP NETCONTROL DOMAIN RESTART`	`$ IP NETCONTROL DOMAINNAME RESTART`
`SERVER-CONFIG>ENABLE DNS`	`SERVER-CONFIG>ENABLE DOMAINNAME`

3. Corrected Command Syntax in Copying SSH2 Template File To The Target SSH2 Subdirectory

Version 10.6

In the *VSI TCP/IP Administrator's Guide: Volume II*, *Starting the SSH Server for the First Time* section, the following commands related to copying the SSH2 template file to the target SSH2 subdirectory syntax have been corrected:

Command	Syntax
`Copy`	**Incorrect:** `COPY IP$:SSHD_CONFIG.TEMPLATE IP$:SSHD_CONFIG.;`
	Correct: `$ copy ip$specific_root:[ip.ssh2]sshd2_config.template- _$ ip$specific_root:[ip.ssh2]sshd2_config`
`IP$SYSTARTUP`	**Incorrect**: `$ IP$SYSTARTUP.COM RESTART`
	Correct: `$ @IP$:IP$SYSTARTUP.COM RESTART`
`SHOW SYSTEM`	**Incorrect**: `$ SHOW=SYSTEM/PROCESS="IP$SSH_SERVER"`
	Correct: `$ SHOW SYSTEM/PROCESS="IP$SSH_SERVER"`
`SET FILE`	**Incorrect**: `$ set file /version_limit=X IP$ROOT:[IP.SSH]SSHD.LOG`

	Correct: `$ set file/version=X ip$log:sshd.log`

4. BOOTP "to" Option Values Swapped

Version 10.6

In *VSI TCP/IP Administrator's Guide: Volume II, Table 9.2: BOOTP "to" Option Values* have been corrected.

The corrected values are:

Value	Time Offset	DST Time Offset
CET/CET-DST	3600	7200
MET/MET-DST	3600	7200

5. DHCP Filename Corrected

Version 10.6

In *VSI TCP/IP Administrator's Guide: Volume II, Section 9.8 Checking the DHCP Configuration*, the DHCP configuration file name has been corrected. The corrected value is dhcpd4.exe. Calling DHCPD with an invalid parameter attempts to start the DHCPD server in this process.

6. Cluster Alias Logical Name Corrected

Version 10.6

In the *VSI TCP/IP Administrator's Guide: Volume I, Section 2.2.10.15. Configuring OpenVMScluster Aliasing*, the alias logical name has been corrected.

The corrected value is `ip$ip_cluster_aliases`.

7. IP FONT MKFONTDIR Command Syntax Corrected

Version 10.6

In the *VSI TCP/IP Administrator's Reference Guide, Chapter 1 DCL Command Reference*, the IP FONT MKFONTDIR command syntax has been corrected.

The corrected value is `IP FONT MKFONTDIR [directory_name]`.

8. Queue Groups Configuration Corrected

Version 10.6

In the *VSI TCP/IP Administrator's Guide: Volume I, Section 4.3.11.5 Configuring Queue Groups*, the conditions of implementation of the modified configurations have been corrected.

The modified configuration takes effect after restarting SMTP.

9. Customer Zone Files Privileges Corrected

Version 10.6

In the *VSI TCP/IP Administrator's Guide: Volume I, Table 1.4: Zone Statements*, the zone file command has been corrected to show W:RE privileges with the following syntax:

```
file filename (RWED,RWED,RE,RE)
```

10. SMTP ADD GATEWAY Command Description Corrected

Version 10.6

In the *VSI TCP/IP Administrator's Guide: Volume I, Section 4.3.11.12 Configuring Mail Gateways* and *VSI TCP/IP Administrator's Reference Guide, Chapter 3 MAIL-CONFIG Command Reference,* the ADD GATEWAY command description has been corrected.

In VSI TCP/IP, only one gateway definition is allowed per domain. Preference numbers are not allowed.

11. NFS Version 2 Commands Inadvertently Included

Version 10.5

The VSI TCP/IP V10.5 documentation inadvertently included NFS Version 2 commands and documents obsolete functionality not included in the current product. The following qualifiers are not supported:

```
DEFAULT
FID_CACHE_SIZE
INTERFACE
LABEL=(PORT|READ_SIZE|WRITE_SIZE)
LOCKING
PAGEFILE
PORT
READ_SIZE
RELOAD
SEMANTICS
SOFT
SYNTAX=(NFSMOUNT_RELOAD|NFSMOUNT_DECWINDOWS)
UNIQUE_FILENO
VALUE=(DEFAULT=DECWINDOWS|LIST|REQUIRED|TYPE=[DEFAULT=n|
   LOCKING_TYPES|SEMANTICS_TYPES])
WRITE_SIZE
WSEXTENT
WSQUOTA
```

12. Corrected SSH and SSH2 Directory Specifications

Version 10.5

The following SSH and SSH2 directory citations have been corrected in the *VSI OpenVMS TCP/IP Administrator's Guide: Volume II.*

In *Sections 15.6 and 15.8,* the correct SSH directory is `IP$SPECIFIC:[IP.SSH]`.

In *Section 15.14 SSH Logicals,* the following text has been changed:

`SSH_DIR`

Points to the directory where the SSH1 configuration, master server log file, and hostkey files are kept. Normally, this is `IP$SPECIFIC_ROOT:[CONFIG]`. It is defined in `START_SSH.COM`.

In *Section 15.15.5.1. HostSpecificConfig Notes:*

`SSH_DIR:SSHD2_CONFIG` has been corrected to `SSH2_DIR:SSHD2_CONFIG`

Chapter 8: Using the $QIO System Service

Note: When features are available via both the standard socket interface and QIO$, VSI recommends use of the standard socket interface. For example, use of $QIO in VSI TCP/IP V10.5 or V10.6 might result in status return values that indicate failure. These values may not match the values returned by TCP/IP Services.

This chapter describes how to use the $QIO system service and its data structures with VSI TCP/IP. After you create a network pseudodevice (BG:) and assign a channel to it, use the $QIO system. This information will be included in the *VSI TCP/IP Programmer Reference Manual* in a future release of the documentation.

$QIO System Service Variations

The two variations of the $QIO system service are:

- **Queue I/O Request ($QIO)** — Completes asynchronously. It returns to the caller immediately after queuing the I/O request, without waiting for the I/O operation to complete.
- **Queue I/O Request and Wait ($QIOW)** — Completes synchronously. It returns to the caller after the I/O operation completes. The only difference between the $QIO and $QIOW calling sequences is the service name. The system service arguments are the same.

$QIO Format

The $QIO calling sequence has the following format:

```
SYS$QIO [efn],chan,func,[iosb],[astadr],[astprm],[p1],[p2],[p3],[p4],
[p5],[p6]
```

The following table describes each argument.

$QIO Arguments

Argument	Description
astadr	AST (asynchronous system trap) service routine
astprm	AST parameter to be passed
chan	I/O channel
efn	Event flag number
func	Network pseudodevice function code and/or function modifier
iosb	I/O status block
p1, p2, p3, p4, p5, p6	Function-specific I/O request parameters

Symbol Definition Files

The following table lists the symbol definition files for the $QIO arguments p1 through p6. Use the standard mechanism for the programming language you are using to include the appropriate symbol definition files in your program.

Network Symbol Definition Files

File Name	File Name
TCPIP$INETDEF.H	C
TCPIP$INETDEF.FOR	Fortran
TCPIP$INETDEF.PAS	PASCAL
TCPIP$INETDEF.MAR	MACRO-32
TCPIP$INETDEF.R32	BLISS-32
TCPIP$INETDEF.ADA	Ada

TCPIP$INETDEF.BAS	BASIC

$QIO Functions

The following table lists the $QIO function codes commonly used in a network application.

Note: The IO$_SETMODE and IO$_SETCHAR function codes are identical. All references to the IO _SETMODE function code, its arguments, options, function modifiers, and condition values returned also apply to the IO$_SETCHAR function code, which is not explicitly described in this manual.

The IO$_SENSEMODE and IO$_SENSECHAR function codes are identical. All references to the IO$_SENSEMODE function code, its arguments, options, function modifiers, and condition values returned also apply to the IO$_SENSECHAR function code, which is not explicitly described in this manual.

Table 8.1 $QIO Function Codes

$QIO Function	Codes Description
$QIO(IO$_SETMODE) $QIO(IO$_SETCHAR)	Creates the socket by setting the internet domain, protocol (socket) type, and protocol of the socket. Binds a name (local address and port) to the socket. Defines a network pseudodevice as a listener on a TCP/IP server. Specifies socket options.
$QIO(IO$_ACCESS)	Initiates a connection request from a client to a remote host using TCP. Specifies the peer where you can send datagrams. Accepts a connection request from a TCP/IP client when used with the IO$M_ACCEPT function modifier.
$QIO(IO$_WRITEVBLK)	Writes data (virtual block) from the local host to the remote host for stream sockets, datagrams, and raw IP.
$QIO(IO$_READVBLK)	Reads data (virtual block) from the remote host to the local host for stream sockets, datagrams, and raw IP.
$QIO(IO$_DEACCESS)	Disconnects the link established between two communication agents through an IO$_DEACCESS function. Shuts down the communication link when used with the IO$M_SHUTDOWN function modifier. You can shut down the receive or transmit portion of the link, or both.
$QIO(IO$_SENSECHAR) $QIO(IO$_SENSEMODE)	Obtains socket information.

$QIO Arguments

You pass two types of arguments with the $QIO system service: function-independent arguments and function-dependent arguments. The following sections provide information about $QIO system service arguments.

$QIO Function-Independent Arguments

The following table describes the $QIO function-independent arguments.

Table 8.2 $QIO Function-Independent Arguments

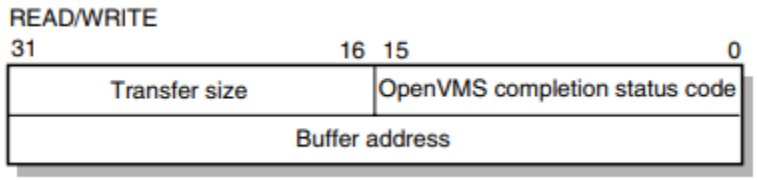
Argument	Description
astadr	Address of the asynchronous system trap (AST) routine to be executed when the I/O operation is completed.
astprm	A quadword (Alpha) containing the value to be passed to the AST routine.

chan	A longword value that contains the number of the I/O channel. The $QIO system service uses only the low-order word.
efn	A longword value of the event flag number that the $QIO system service sets when the I/O operation completes. The $QIO system service uses only the low-order byte.
func	A longword value that specifies the network pseudodevice function code and function modifiers that specify the operation to be performed. Function modifiers affect the operation of a specified function code. In MACRO-32, you use the exclamation point (!) to logically OR the function code and its modifier. In Compaq C, you use the vertical bar (|). This manual uses the vertical bar (|) in text.
iosb	The I/O status block that receives the final status message for the I/O operation. The iosb argument is the address of the quadword I/O status block. (For the format of the I/O status block, see Section 5.2.

I/O Status Block

The system returns the status of a $QIO operation in the I/O status block (IOSB) supplied as an argument to the $QIO call. In the case of a successful IO$_READVBLK or IO$_WRITEVBLK operation, the second word of the I/O status block contains the number of bytes transferred during the operation (see Figure 8.1).

Figure 8.1 I/O Status Block for a Successful READ or WRITE Operation



With an unsuccessful IO$_READVBLK or IO$_WRITEVBLK operation, in most cases, the system returns a UNIX error code in the second word of the I/O status block.

For C programs, the OpenVMS completion codes are defined in the SSDEF.H header file. The UNIX error codes are defined in the ERRNO.H header file and in the TCPIP$INETDEF.H header file. For other language variants, see **Error! Reference source not found.**.

$QIO Function-Dependent Arguments

Arguments **p1, p2, p3, p4, p5,** and **p6** to the $QIO system service are used to pass function-dependent arguments. Table 8.3 lists arguments p1 through p6 for the $QIO system service and indicates whether the parameter is passed by value, by reference, or by descriptor.

Table 8.3 $QIO Function-Dependent Arguments

$QIO	p1	p2	p3	p4	p5	p6
IO$_ACCESS	Not used	Not used	Remote socket name[1]	Not used	Not used	Not used
IO$_ACCESS \| IO$M_ACCEPT	Not used	Not used	Remote socket name[2]	Channel number[3]	Not used	Not used

[1] By item_list_2 descriptor.

[2] By item_list_3 descriptor.

[3] By reference.

IO$_ACPCONTROL	Subfunction code[4]	Input parameter[4]	Buffer length[3]	Buffer[4]	Not used	Not used
IO$_DEACCESS	Not used	Not used	Not used	Not used	Not used	Not used
IO$_DEACCESS \| IO$M_SHUTDOWN	Not used	Not used	Not used	Shutdown flags[5]	Not used	Not used
IO$_READVBLK	Buffer[3]	Buffer size[5]	Remote socket name[2]	Flags[5]	Not used	Output buffer list[4]
IO$_READVBLK \| IO$M_INTERRUPT	Buffer[3]	Buffer size[5]	Not used	Not used	Not used	Not used
IO$_WRITEVBLK	Buffer[3]	Buffer size[5]	Remote socket name[1]	Flags[5]	Input buffer list[2]	Not used
IO$_WRITEVBLK \| IO$M_INTERRUPT	Buffer[3]	Buffer size[5]	Not used	Not used	Not used	Not used
IO$_SETMODE	Socket char[3]	Not used	Local socket name	Backlog limit[5]	Input parameter list[1]	Not used
IO$_SETMODE \| IO$_OUTBAND	AST procedure[3]	User argument[5]	Access mode[5]	Not used	Not used	Not used
IO$_SETMODE \| IO$_READATTN	AST procedure[3]	User argument[5]	Access mode[5]	Not used	Not used	Not used
IO$_SETMODE \| IO$WRTATTN	AST procedure[3]	User argument[5]	Access mode[5]	Not used	Not used	Not used
IO$_SENSEMODE	Not used	Not used	Local socket name[2]	Remote socket name[2]	Not used	Output parameter list[1]

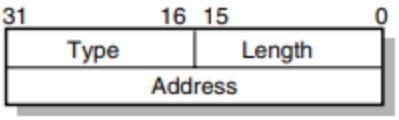
Passing Arguments by Descriptor

In addition to OpenVMS argument descriptors, I/O functions specific to TCP/IP Services also pass arguments by using item_list_2 and item_list_3 argument descriptors. The format of these argument descriptors is unique to TCP/IP Services, and they supplement argument descriptors defined in the *OpenVMS Calling Standard*.

Use of an item_list_2 or item_list_3 argument descriptor is indicated when the argument's passing mechanism is specified as an item_list_2 descriptor or an item_list_3 descriptor.

The item_list_2 argument descriptors describe the size, data type, and starting address of a service parameter.

An item_list_2 argument descriptor contains three fields, as depicted in the following diagram:

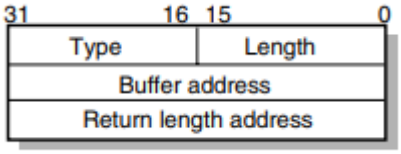


The first field is a word containing the length (in bytes) of the parameter being described. The second field is a word containing a symbolic code specifying the data type of the parameter. The third field is a longword containing the starting address of the parameter.

[4] By descriptor.

[5] By value.

The item_list_3 argument descriptors describe the size, data type, and address of a buffer in which a service writes parameter information returned from a get operation. An item_list_3 argument descriptor contains four fields, as depicted in the following diagram:



The first field is a word containing the length (in bytes) of the buffer in which a service writes information.

The length of the buffer needed depends on the data type specified in the type field. If the value of buffer length is too small, the service truncates the data. The second field is a word containing a symbolic code specifying the type of information that a service is to return. The third field is a longword containing the address of the buffer in which a service writes the information. The fourth field is a longword containing the address of a longword in which a service writes the length (in bytes) of the information it actually returned.

Note: When a parameter specified as a descriptor is described as ‘‘read-only’’, the descriptor itself is only read, and TCP/IP Services does not modify the memory described. However, system service postprocessing requires that the described memory must be both readable and writable.

Specifying an Input Parameter List

Use the **p5** argument with the IO$_SETMODE function to specify input parameter lists. The **p5** argument specifies the address of a item_list_2 descriptor that points to and identifies the type of input parameter list.

To initialize an item_list_2 descriptor, you need to:

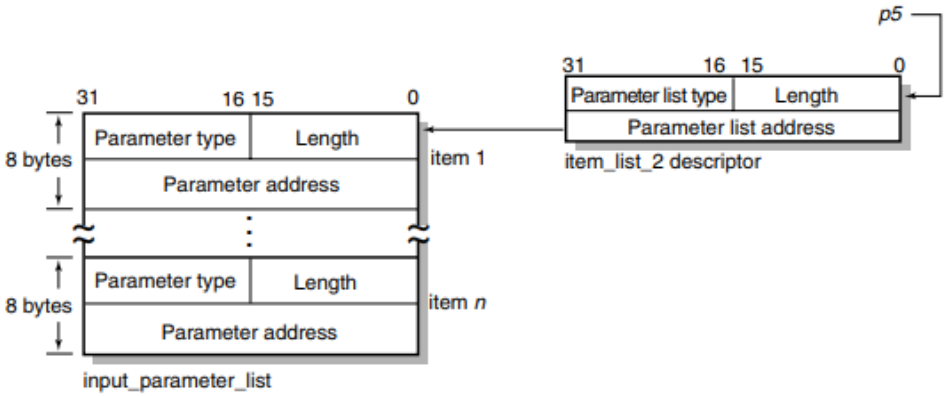
1. Set the descriptor’s type field to one of the following symbolic codes to specify the type of input parameter list:

Symbolic Name	**Input Parameter List Type**
TCPIP$C_SOCKOPT	Socket options
TCPIP$C_TCPOPT TCP	protocol options
TCPIP$C_IPOPT IP	protocol options
TCPIP$C_IOCTL I/O	control commands

2. Set the descriptor’s length field to specify the length of the input parameter list.
3. Set the descriptor’s address field to specify the starting address of the input parameter list.

The following figure illustrates how the **p5** argument specifies an input parameter list.

Figure 8.2 Specifying an Input Parameter List



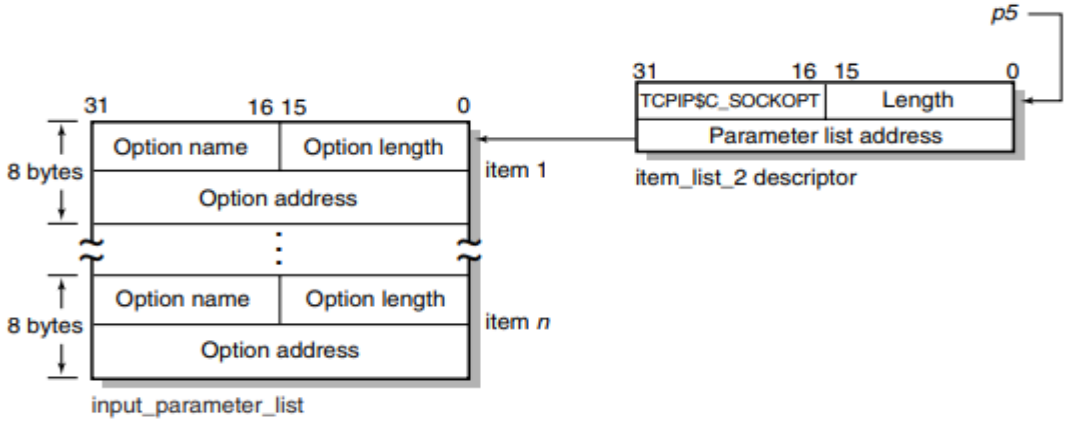
As the name implies, input parameter lists consist of one or more contiguous item_list_2 or ioctl_comm structures. The length of a input parameter list is determined solely from the length field of its associated argument descriptor. Input parameter lists are never terminated by a longword containing a zero.

Each item_list_2 structure that appears in an input parameter list describes an individual parameter or item to set. Such items include socket or protocol options as identified by the item's type field. To initialize an item_list_2 descriptor, you need to:

1. Set the item's type field to one of the symbolic codes in Section 5.7.
2. Set the item's length field to specify the length of the item.
3. Set the item's address field to specify the starting address of its data.

The following figure illustrates how to specify setting socket options.

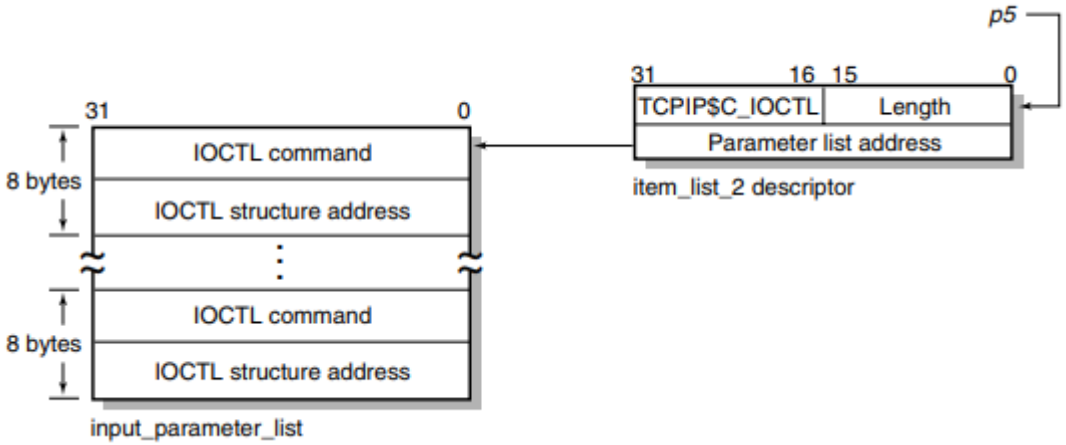
Figure 8.3 Setting Socket Options



Each ioctl_comm structure appearing in an input parameter list contains an I/O control command the IOCTL request code (as defined by $SIOCDEF) and its associated IOCTL structure address.

The following figure illustrates how to specify (set) I/O control (IOCTL) commands.

Figure 8.4 Setting IOCTL Parameters



Specifying an Output Parameter List

Use the **p6** argument with the IO$_SENSEMODE function to specify output parameter lists. The **p6** argument specifies the address of an item_list_2 descriptor that points to and identifies the type of output parameter list.

To initialize an item_list_2 descriptor, you need to:

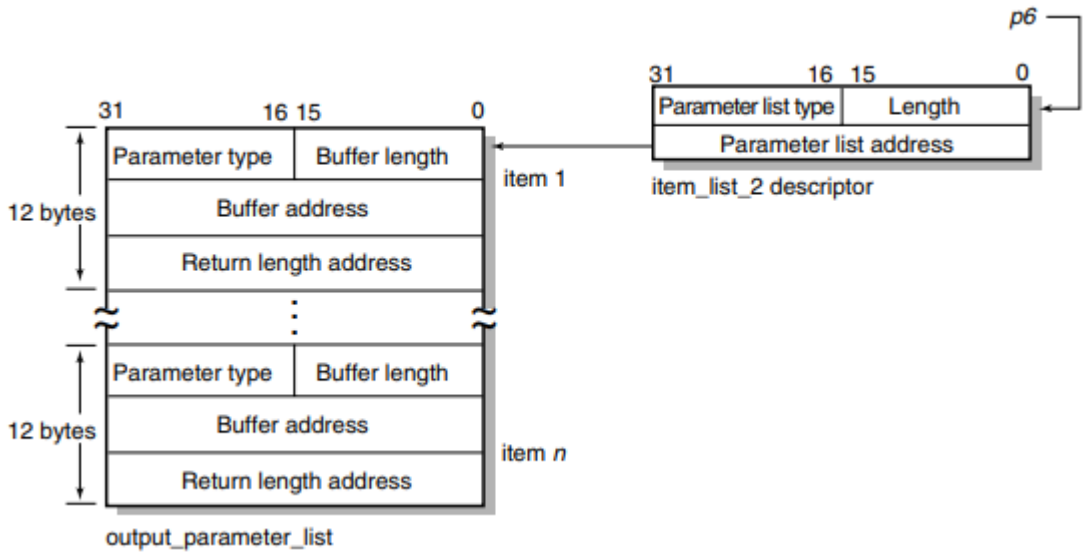
1. Set the descriptor's type field to one of the following symbolic codes to specify the type of output parameter list:

Symbolic Name	Output Parameter List Type
TCPIP$C_SOCKOPT	Socket options
TCPIP$C_TCPOPT	TCP protocol options
TCPIP$C_IPOPT	IP protocol options
TCPIP$C_IOCTL	I/O control commands

2. Set the descriptor's length field to specify the length of the output parameter list.
3. Set the descriptor's address field to specify the starting address of the output parameter list.

The following figure illustrates how the **p6** argument specifies an output parameter list:

Figure 8.5 Specifying an Output Parameter List



As the name implies, output parameter lists consist of one or more contiguous item_list_3 or ioctl_comm structures. The length of an output parameter list is determined solely from the length field of its associated argument descriptor. Output parameter lists are never terminated by a longword containing a zero.

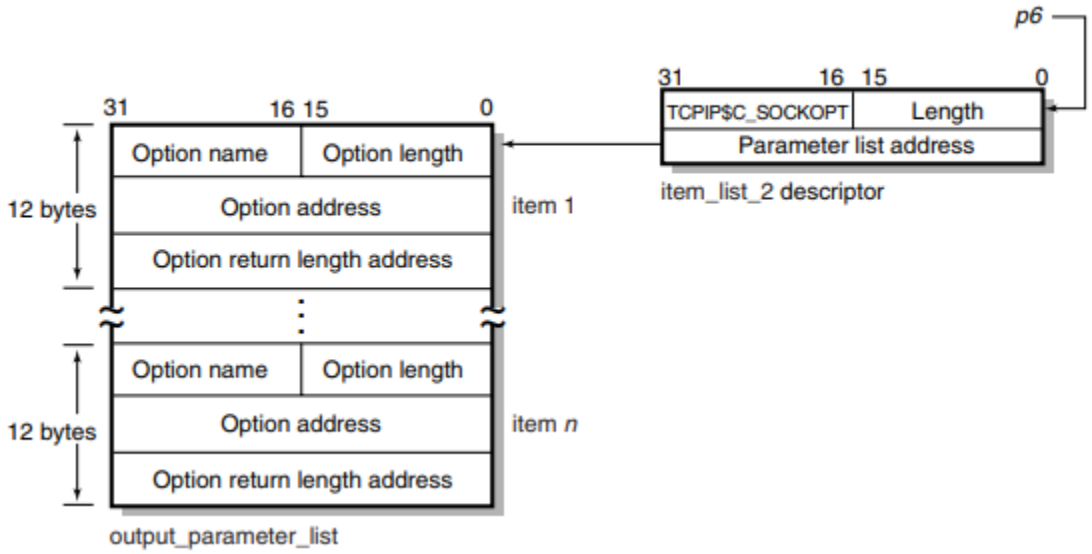
Each item_list_3 structure that appears in an output parameter list describes an individual parameter or item to return. Such items include socket or protocol options as identified by the item's type field.

To initialize an item_list_3 structure, you need to:

1. Set the item's type field to one of symbolic codes found in Section 5.7.
2. Set the item's buffer length field to specify the length of its buffer.
3. Set the item's buffer address field to specify the starting address of its buffer.
4. Set the item's returned length address field to specify the address of a longword to receive the length in bytes of the information actually returned for this item.

The following figure illustrates how to specify getting socket options.

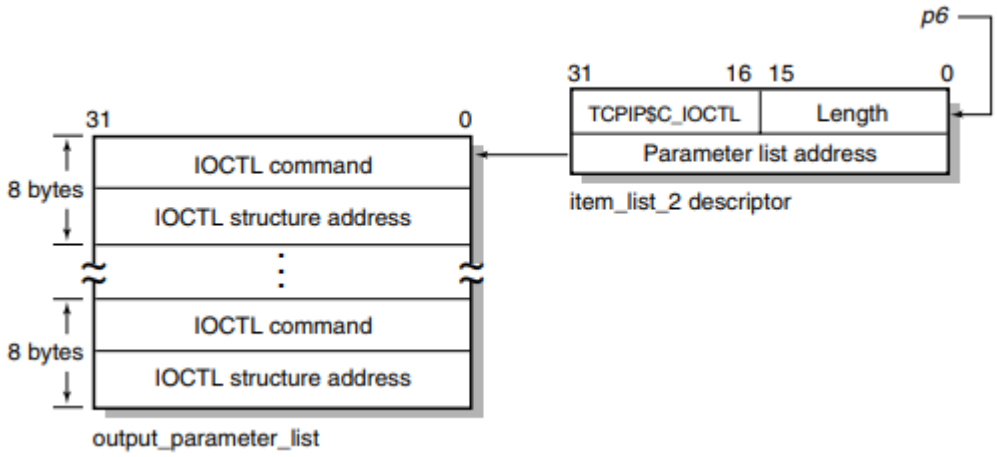
Figure 8.6 Getting Socket Options



Each ioctl_comm structure appearing in a output parameter list contains an I/O control command the IOCTL request code (as defined by $SIOCDEF) and its associated IOCTL structure address.

The following figure illustrates how to specify (get) I/O control (IOCTL) commands.

Figure 8.7 Getting IOCTL Parameters



Specifying a Socket Name

Use the **p3** or **p4** argument with the IO$_ACCESS, IO$_READVBLK, IO$_SENSEMODE, IO$_SETMODE, and IO$_WRITEVBLK functions to specify a socket name. The **p3** and **p4** arguments specify the address of an item_list_2 or item_list_3 descriptor that points to a socket name structure. The socket name structure contains address domain, port number, and host internet address.

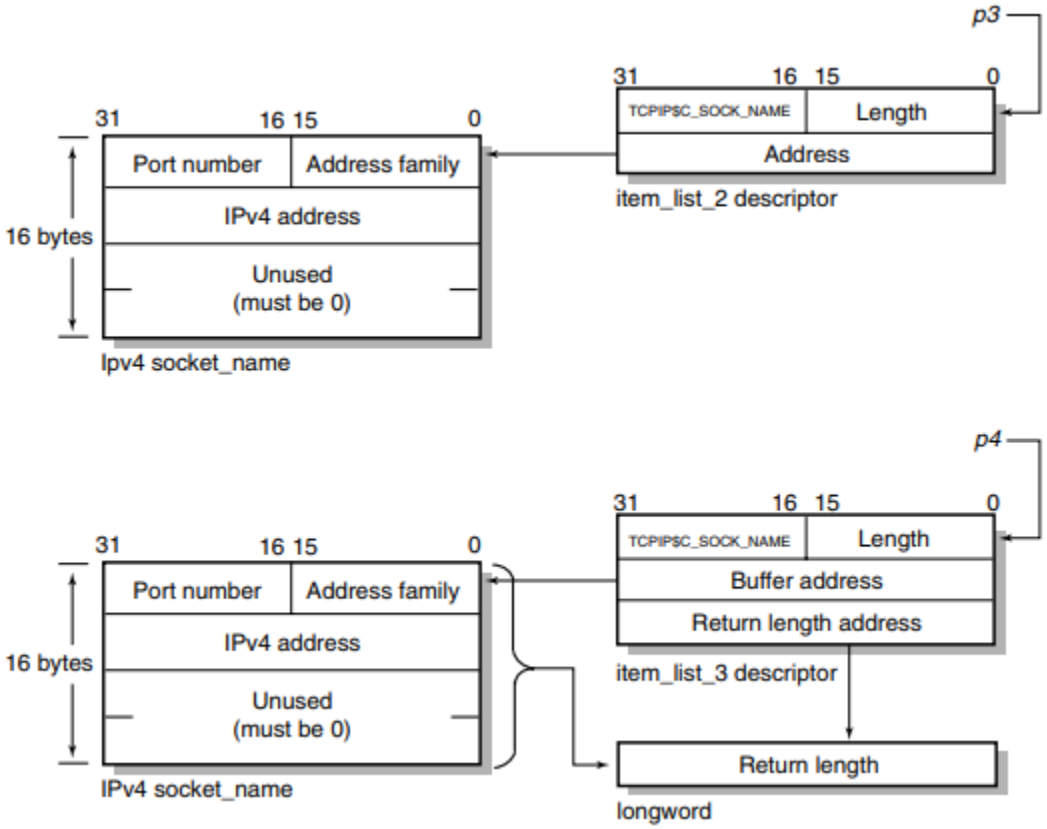
Note: Port numbers 1 to 1023 require a system UIC or a UIC with SYSPRV and BYPASS privileges when assigned. If you specify zero when binding a socket name, the system assigns an available port.

Use an item_list_2 argument descriptor with the IO$_ACCESS, IO$_WRITEVBLK, and IO$_SETMODE functions to specify (set) a socket name. The descriptor's parameter type is TCPIP$C_SOCK_NAME.

Use an item_list_3 argument descriptor with the IO$_ACCESS|IO$M_ACCEPT, IO$_READVBLK, and IO$_SENSEMODE functions to specify (get) a socket name. The descriptor's parameter type is TCPIP$C_SOCK_NAME.

With BSD Version 4.4, specify socket names as illustrated in the following figure:

Figure 8.8 Specifying a Socket Name



Specifying a Buffer List

Use the **p5** argument with the IO$_WRITEVBLK function to specify input buffer lists. The **p5** argument specifies the address of a 32- or 64-bit fixed-length descriptor (on Alpha systems) pointing to an input buffer list.

Use the **p6** argument with the IO$_READVBLK function to specify output buffer lists. The **p6** argument specifies the address of a 32- or 64-bit fixed-length descriptor (on Alpha systems) pointing to an output buffer list.

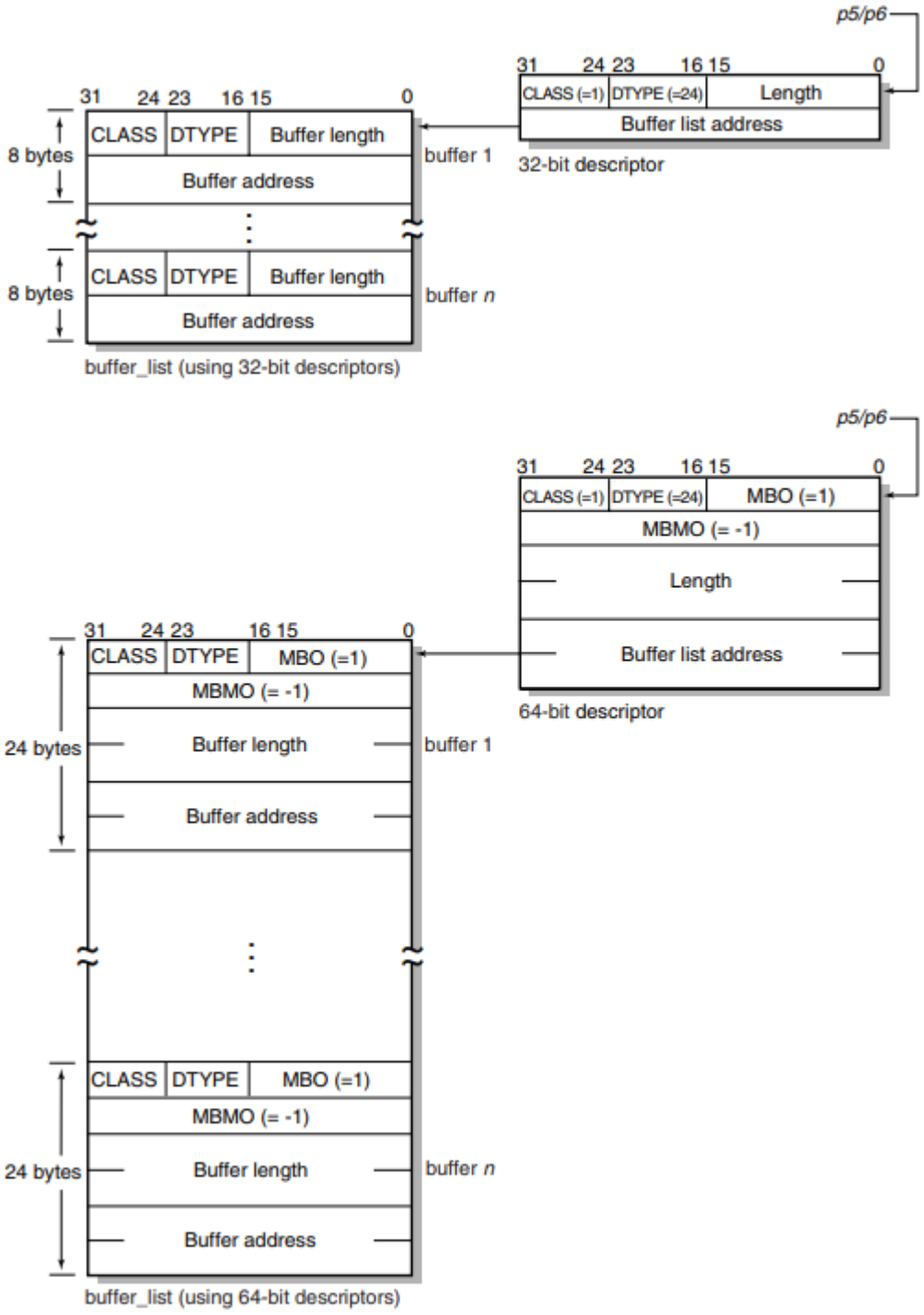
To initialize the **p5** or **p6** argument descriptor, you need to:

1. Set the descriptor's data-type code (the DTYPE field) to DSC$K_DTYPE_DSC to specify a buffer list containing one or more descriptors defining the length and starting address of user buffers.
2. Set the descriptor's class code (the CLASS field) to DSC$K_CLASS_S.
3. Set the descriptor's length field to specify the length of the buffer list.

4. Set the descriptor's MBO field to 1 and the MBMO field to all 1s if this is a 64-bit argument descriptor.

The following figure illustrates how to specify a buffer list:

Figure 8.9 Specifying a Buffer List



Buffer lists, as the name implies, consist of one or more contiguous 32- or 64-bit fixed-length descriptors (on Alpha systems).

Each 32- or 64-bit descriptor that appears in a buffer list describes one user buffer. Initialize each descriptor by setting its data type, class, length, and address fields as appropriate for 32- and 64-bit descriptors.

For more information about using 32-bit and 64-bit descriptors, refer to the *OpenVMS Calling Standard*.

$QIO Interface

The $QIO interface allows programmers to use more sophisticated programming techniques than available with the socket library. Using the $QIO interface, you can perform fully asynchronous I/O to the network and receive Asynchronous System Traps (ASTs) when out-of-band data arrives (similar to the UNIX `SIGURG` signal). In general, there is a one-to-one mapping between the socket library functions and $QIO calls.

The $QIO interface returns an OpenVMS error code in the first word of the Input/Output Status Block (IOSB). If the low bit of the OpenVMS error code is clear, an error has been returned by the network.

The OpenVMS error code is generated from the UNIX **errno** code by multiplying the UNIX code by 8 (eight) and logical ORing it with 0x8000.

You can mix and match the socket library function and the $QIO calls. For example, you can use **socket()** and **connect()** to establish a connection, then use IO$_SEND and IO$_RECEIVE to send and receive data on it.

Note: If more than one $QIO operation is pending on a socket at any one time, there is no guarantee that the $QIO calls will complete in the order they are queued. In particular, if more than one read or write operation is pending at any one time, the data may be interleaved. You do not need to use multiple read or write operations concurrently on the same socket to increase performance because of the network buffering.

The function codes for the VSI TCP/IP-specific $QIO functions are defined in the include file `SYS$SYSDEVICE:[VMS$COMMON.IP.EXAMPLES.VMS directory]IP_root:[IP.include.vms]inetiodef.h`.

If the compile time constant `USE_BSD44_ENTRIES` is defined, then the BSD 4.4 variant of the `IO$_ACCEPT, IO$_BIND, IO$_CONNECT, IO$_GETPEERNAME, IO$_GETSOCKNAME, IO$_RECEIVE, IO$_SEND` is selected.

The following are the interface functions:

IO$_ACCEPT	**IO$_SEND**
IO$_ACCEPT_WAIT	**IO$_SENSEMODE**
IO$_BIND	**IO$_SENSEMODE \| IO$M_CTRL**
IO$_CONNECT	**IO$_SETCHAR**
IO$_GETPEERNAME	**IO$_SETMODE \| IO$M_ATTNAST**
IO$_GETSOCKNAME	**IO$_SETSOCKOPT**
IO$_GETSOCKOPT	**IO$_SHUTDOWN**
IO$_IOCTL	**IO$_SOCKET**
IO$_LISTEN	**SYS$CANCEL**
IO$_RECEIVE (IO$_READVBLK)	**SYS$DASSGN**
IO$_SELECT	

IO$_ACCEPT

IO$_ACCEPT — Extracts the first connection from the queue of pending connections on a socket, creates a new socket with the same properties as the original socket, and associates an OpenVMS channel to the new socket. **IO$_ACCEPT** is equivalent to the **accept()** socket library function. Normally, instead of calling **IO$_ACCEPT** to wait for a connection to become available, **IO$_ACCEPT_WAIT** is used. This allows your process to wait for the connection without holding the extra network channel and tying up system resources. When the **IO$_ACCEPT_WAIT** completes, it indicates that a connection is available. **IO$_ACCEPT** is then called to accept it.

Format

Status = SYS$QIOW(Efn, New_VMS_Channel, **IO$_ACCEPT**, IOSB, AstAdr, AstPrm, Address, AddrLen, VMS_Channel, 0, 0, 0);

Arguments

New_VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

An OpenVMS channel to a newly-created INET device. Create this channel by using SYS$ASSIGN to assign a fresh channel to INET0: before issuing the **IO$_ACCEPT** call. The accepted connection is accessed using this channel.

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

The OpenVMS channel to the INET: device on which the **IO$_LISTEN** call was performed. After accepting the connection, this device remains available to accept new connections.

Address	
OpenVMS Usage:	**special_structure**
type:	**structure defined below**
access:	**write only**
mechanism:	**by reference**

An optional pointer to a structure that, following the completion of the **IO$_ACCEPT** call, contains the address of the socket that made the connection. This structure is defined as follows:

```
struct {
unsigned long Length;
struct sockaddr Address;
};
```

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

The length of the buffer pointed to by the **Address** argument, in bytes. It must be at least 20 bytes.

IO$_ACCEPT_WAIT

IO$_ACCEPT_WAIT — Used to wait for an incoming connection without accepting it. This allows your process to wait for the connection without holding the extra network channel and tying up system resources. When the **IO$_ACCEPT_WAIT** call completes, it indicates that a connection is available. **IO$_ACCEPT** is then called to accept it. The **IO$_ACCEPT_WAIT** call takes no function-specific parameters.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_ACCEPT_WAIT, IOSB, AstAdr, AstPrm, 0, 0, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

The OpenVMS channel to the INET: device on which the **IO$_LISTEN** call was performed.

IO$_BIND

IO$_BIND — Assigns an address to an unnamed socket. When a socket is created with **IO$_SOCKET**, it exists in a name space (address family) but has no assigned address. IO$_BIND requests that the address be assigned to the socket. **IO$_BIND** is equivalent to the bind() socket library function.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_BIND, IOSB, AstAdr, AstPrm, Name, NameLen, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Name	
OpenVMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**read only**
mechanism:	**by reference**

The address to which the socket should be bound. The exact format of the **Address** argument is determined by the domain in which the socket was created.

NameLen	
OpenVMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the **Name** argument, in bytes.

IO$_CONNECT

IO$_CONNECT — When used on a **SOCK_STREAM** socket, this function attempts to make a connection to another socket. When used on a **SOCK_DGRAM** socket, this function permanently specifies the peer to which datagrams are sent to and received from. The peer socket is specified by name, which is an address in the communications domain of the socket. Each communications domain interprets the name parameter in its own way. **IO$_CONNECT** is equivalent to the **connect()** socket library function. If the address of the local socket has not yet been specified with **IO$_BIND**, the local address is also set to an unused port number when **IO$_CONNECT** is called.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_CONNECT, IOSB, AstAdr, AstPrm, Name, Name-Len, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Name	
OpenVMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**read only**
mechanism:	**by reference**

The address of the peer to which the socket should be connected. The exact format of the **Address** argument is determined by the domain in which the socket was created.

NameLen	
OpenVMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the **Name** argument, in bytes.

IO$_GETPEERNAME

IO$_GETPEERNAME — Returns the name of the peer connected to the specified socket. It is equivalent to the **getpeername()** socket library function.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_GETPEERNAME, IOSB, AstAdr, AstPrm, Address, AddrLen, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Address	

OpenVMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**write only**
mechanism:	**by reference**

A result parameter filled in with the address of the peer, as known to the communications layer. The exact format of the **Address** argument is determined by the domain in which the communication is occurring.

AddrLen	
OpenVMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

On entry, contains the length of the space pointed to by **Address**, in bytes. On return, it contains the actual length, in bytes, of the address returned.

IO$_GETSOCKNAME

IO$_GETSOCKNAME — Returns the current name of the specified socket. Equivalent to the **getsockname()** socket library function.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_GETSOCKNAME, IOSB, AstAdr, AstPrm, Address, AddrLen, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Address	
OpenVMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**write only**
mechanism:	**by reference**

A result parameter filled in with the address of the local socket, as known to the communications layer. The exact format of the **Address** argument is determined by the domain in which the communication is occurring.

AddrLen	
OpenVMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

On entry, contains the length of the space pointed to by **Address**, in bytes. On return, it contains the actual length, in bytes, of the address returned.

IO$_GETSOCKOPT

IO$_GETSOCKOPT — Retrieves options associated with a socket. It is equivalent to the **getsockopt()** library routine. Options can exist at multiple protocol levels; however, they are always present at the uppermost socket level. When manipulating socket options, you must specify the level at which the option

resides and the name of the option. To manipulate options at the socket level, specify level as **SOL_SOCKET**. To manipulate options at any other level, specify the protocol number of the appropriate protocol controlling the option. For example, to indicate that an option is to be interpreted by the TCP protocol, set **Level** to the protocol number of TCP, as determined by calling **getprotobyname()**. **OptName** and any specified options are passed without modification to the appropriate protocol module for interpretation. The include file `IP_root:[IP.include.sys]socket.h` contains definitions for socket-level options. Options at other protocol levels vary in format and name. For more information on what socket options may be retrieved with **IO$_GETSOCKOPT**, see **setsockopt()**.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_GETSOCKOPT, IOSB, AstAdr, AstPrm, Level, Opt-Name, OptVal, OptLen, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Level	
OpenVMS Usage:	**option_level**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The protocol level at which the option will be manipulated. Specify **Level** as **SOL_SOCKET** or a protocol number as returned by **getprotoent()**.

OptName	
OpenVMS Usage:	**option_name**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The option that is to be manipulated.

OptVal	
OpenVMS Usage:	**dependent on OptName**
type:	**byte buffer**
access:	**write only**
mechanism:	**by reference**

A 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.

OptLen	
OpenVMS Usage:	**option_length**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

On entry, contains the length of the space pointed to by **OptVal**, in bytes. On return, it contains the actual length, in bytes, of the option returned.

IO$_IOCTL

IO$_IOCTL — Performs a variety of functions on the network; in particular, it manipulates socket characteristics, routing tables, ARP tables, and interface characteristics. The **IO$_IOCTL** call is equivalent to the **socket_ioctl()** library routine. A **IO$_IOCTL** request has encoded in it whether the argument is an input or output parameter, and the size of the argument, in bytes. Macro and define statements used in specifying an **IO$_IOCTL** request are located in the file `IP_root:[IP.include. sys]ioctl.h.`

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_IOCTL, IOSB, AstAdr, AstPrm, Request, ArgP, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Request	
OpenVMS Usage:	**ioctl_request**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Which **IO$_IOCTL** function to perform. The available **IO$_IOCTL** functions are documented in the **socket_ioctl** sections.

ArgP	
OpenVMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**read, write, or modify depending on Request**
mechanism:	**by reference**

A pointer to a buffer whose format and function is dependent on the **Request** specified.

IO$_LISTEN

IO$_LISTEN — Specifies the number of incoming connections that may be queued while waiting to be accepted. This backlog must be specified before accepting a connection on a socket. The **IO$_LISTEN** function applies only to sockets of type **SOCK_STREAM**. The **IO$_LISTEN** call is equivalent to the **listen()** socket library function.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_LISTEN, IOSB, AstAdr, AstPrm, BackLog, 0, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Backlog	
OpenVMS Usage:	**connection_backlog**
type:	**longword (unsigned)**

access:	**read only**
mechanism:	**by value**

Defines the 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 length is limited to 5.

IO$_RECEIVE (IO$_READVBLK)

IO$_RECEIVE — Receives messages from a socket. This call is equivalent to the **recvfrom()**, **recv()**, and **socket_read()** socket library functions. The length of the message received is returned in the second and third word of the I/O Status Block (IOSB). A count of 0 indicates an end-of-file condition; that is, the connection has been closed. If a message is too long to fit in the supplied buffer and the socket is type **SOCK_DGRAM**, excess bytes are discarded. If no messages are available at the socket, the **IO$_RECEIVE** call waits for a message to arrive, unless the socket is nonblocking (see **socket_ioctl()**).

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_RECEIVE, IOSB, AstAdr, AstPrm, Buffer, Size, Flags, From, FromLen, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Buffer	
OpenVMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**write only**
mechanism:	**by reference**

The address of a buffer in which to place the data read.

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

The length of the buffer specified by **Buffer**. The actual number of bytes read is returned in the **Status**.

Flags	
OpenVMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Control information that affects the **IO$_RECEIVE** call. The **Flags** argument is formed by ORing one or more of the following values:

```
#define MSG_OOB 0x1 /* process out-of-band data */
#define MSG_PEEK 0x2 /* peek at incoming message */
```

The MSG_OOB flag causes **IO$_RECEIVE** to read any out-of-band data that has arrived on the socket.

The MSG_PEEK flag causes **IO$_RECEIVE** to read the data present in the socket without removing the data. This allows the caller to view the data, but leaves it in the socket for future **IO$_RECEIVE** calls.

From	
OpenVMS Usage:	**special_structure**
type:	**structure defined below**
access:	**write only**
mechanism:	**by reference**

An optional pointer to a structure that, following the completion of the **IO$_RECEIVE**, contains the address of the socket that sent the packet. This structure is defined as follows:

```
struct {
unsigned short Length;
struct sockaddr Address;
};
```

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

The length of the buffer pointed to by the **From** argument, in bytes. It must be at least 18 bytes.

IO$_SELECT

IO$_SELECT — Examines the specified channel to see if it is ready for reading, ready for writing, or has an exception condition pending (the presence of out-of-band data is an exception condition). The UNIX **select()** system call can be emulated by posting multiple **IO$_SELECT** calls on different channels. **IO$_SELECT** is only useful for channels assigned to the INET: device. It cannot be used for any other VMS I/O device.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SELECT, IOSB, AstAdr, AstPrm, Modes, 0, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Modes	
OpenVMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**Modify**
mechanism:	**by reference**

On input, the **Modes** argument is a bit mask of one or more of the following values:

```
#define SELECT_DONTWAIT (1<<0)
#define SELECT_READABLE (1<<1)
#define SELECT_WRITEABLE (1<<2)
#define SELECT_EXCEPTION (1<<3)
```

If the SELECT_DONTWAIT bit is set, the **IO$_SELECT** call will complete immediately, whether or not the socket is ready for any I/O operations. If this bit is not set, the **IO$_SELECT** call will wait until the socket is ready to perform one of the requested operations.

If the SELECT_READABLE bit is set, the **IO$_SELECT** call will check if the socket is ready for reading or a connection has been received and is ready to be accepted.

If the SELECT_WRITEABLE bit is set, the **IO$_SELECT** call will check if the socket is ready for writing or a connection request has been completed.

If the SELECT_EXCEPTION bit is set, the **IO$_SELECT** call will check if the socket has out-of band data ready to read.

On output, the **Modes** argument is a bit mask that indicates which operations the socket is ready to perform. If the SELECT_DONTWAIT operation was specified, the **Modes** value may be zero; if SELECT_DONTWAIT is not specified, then one or more of the SELECT_READABLE, SELECT_WRITABLE, or SELECT_EXCEPTION bits will be set.

IO$_SEND

IO$_SEND — Transmits a message to another socket. It is equivalent to the **sendto()**, **send()**, and **socket_write()** socket library functions. If no message space is available at the socket to hold the message to be transmitted, **IO$_SEND** blocks unless the socket has been placed in non-blocking I/O mode via **IO$_IOCTL**. If the message is too long to pass through the underlying protocol in a single unit, the error EMSGSIZE is returned and the message is not transmitted.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SEND, IOSB, AstAdr, AstPrm, Buffer, Size, Flags, To, ToLen, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Buffer

Buffer	
OpenVMS Usage:	**arbitrary**
type:	**byte buffer**
access:	**read only**
mechanism:	**by reference**

The address of a buffer containing the data to send.

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

The length of the buffer specified by **Buffer**.

Flags	
OpenVMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Control information that affects the **IO$_SEND** call. The **Flags** argument can be zero or the following:

```
#define MSG_OOB 0x1 /* process out-of-band data */
```

The MSG_OOB flag causes **IO$_SEND** to send out-of-band data on sockets that support this operation (such as **SOCK_STREAM**.

To	
OpenVMS Usage:	**socket_address**
type:	**struct sockaddr**
access:	**read only**
mechanism:	**by reference**

An optional pointer to the address to which the packet should be transmitted. The exact format of the **Address** argument is determined by the domain in which the communication is occurring.

ToLen	
OpenVMS Usage:	**socket_address_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

An optional argument that contains the length of the address pointed to by the **To** argument.

IO$_SENSEMODE

IO$_SENSEMODE — Reads the active connections status and returns status information for all of the active and listening connections.

Format

Status = SYS$QIO(efn, chan, IO$_SENSEMODE, iosb, astadr, astprm, buffer, address, conn_type, 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.

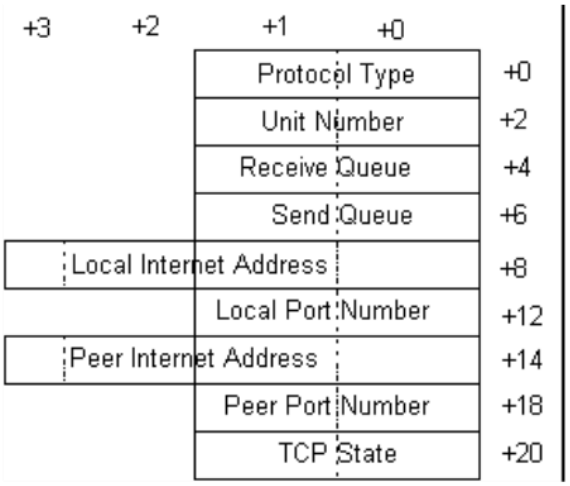
P3=value	
OpenVMS Usage:	**Longword_unsigned**
type:	**Longword (unsigned)**
access:	**Read only**
mechanism:	**by value**

0 to get information about TCP connections, non-zero to get information about UDP connections.

Figure 8.10 shows the 22 bytes of information returned for each connection.

Protocol type	Word value is 4 for INETDRIVER stream sockets, and 5 for BGDRIVER stream sockets.
Unit number	Word value is the 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 Table 8.4 for the mask value definitions.

Figure 8.10 Connection Status Information



Status

SS$_BUFFEROVF	Buffer too small for all connections Truncated buffer returned
SS$_DEVINACT	Device not active Contact system manager for why VSI TCP/IP (or INETDRIVER) 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 5.2 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 8.11 I/O Status Block

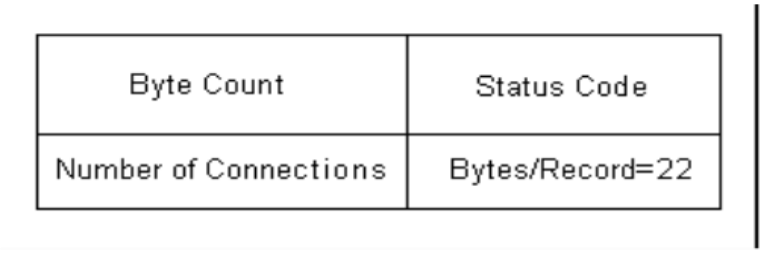


Table 8.4 TCP State Mask Values

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		

IO$_SENSEMODE | IO$M_CTRL

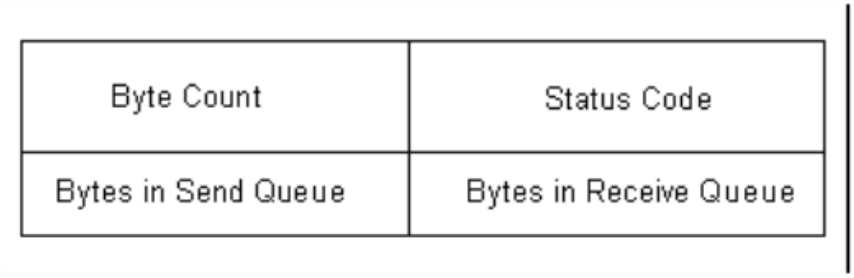
Description

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 8.12 shows the I/O Status Block.

SS$_BUFFEROVF	Buffer too small for all characteristics. Truncated characteristics buffer is returned.
SS$_DEVINACT	Device not active. Contact system manager for why VSI TCP/IP (or TCPDRIVER) not started
SS$_NORMAL	Success. Characteristics returned

Figure 8.12 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.

Performs the following functions:

- Reads network device information
- Reads the routing table

- Reads the ARP information
- Reads the IP SNMP information
- Reads the ICMP SNMP information
- Reads the TCP SNMP information
- Reads the UDP SNMP information

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 VSI TCP/IP 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 the Table 8.5.

Table 8.5 P3 Function Codes

Code	Function
1	P1 of the QIO is not used
2	VMS descriptor of the space in which to put the return information
3	10
4	Not used
5	Not used
6	Not used
7	Read UDP SNMP counters
8	Read routing table
10	Read interface throughput information

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 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.

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=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 **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 IP_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 8.6 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 8.6 Line ID Values

Line ID	Line ID Value	Line ID	Line ID Value	Line ID	Line ID Value
LO-0	^X00000001	DN-*n*	^X00*nn0241*	PD-n	^X00nn0042
PSI-*n*	^X00*nn0006*	PPP-n	^X00*nn0341*		
SL-*n*	^X00*nn0141*	SE-*n*	^X00*nn0402*		

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 Interface Throughput Information

Use **IO$_SENSEMODE | IO$M_CTRL** with p3=10 to read network device information. The information returned in the buffer (specified by p2=descriptor) can consist of multiple records. Each record consists of nine longwords, and one record is returned for each device.

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

Table 8.7 QIO Parameters

Code	Function
1	P1 of the QIO is not used
2	is a VMS descriptor of the space to put the return information
3	10
4	Not used
5	Not used
6	Not used

The returned data is in the following format (all values are integers):

1	Line ID
2	Average Out Bytes (for the last 6 seconds)
3	Average In Bytes
4	Average Out Packets
5	Average In Packets

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 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 8.8.

Table 8.8 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$_DEVINACT	Device not active Contact your system manager to determine why VSI TCP/IP was not started.
SS$_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.

Reading the IP SNMP Counters Function

Use **IO$_SENSEMODE | IO$M_CTRL** with function=4 to read the IP SNMP counters.

The data returned is an array of longwords in the following format:

- Indicates whether or not this entity is acting as an IP router.
- The default value inserted in the IP header's time-to-live field.
- The total number of input datagrams received.
- The number of input datagrams discarded due to errors in their IP headers.
- The number of input datagrams discarded because the IP address in their IP header's destination field was not a valid address to be received at this entity.
- The number of IP datagrams for which this entity was not their final destination, and for which forwarding to another entity was required.
- The number of datagrams received but discarded because of an unknown or unsupported protocol.
- The number of input datagrams received but discarded for reasons other than errors.
- The total number of input datagrams successfully delivered to IP user protocols, including ICMP.
- The total number of IP datagrams that local IP user protocols (including ICMP) supplied to IP in request for transmission.
- The number of output IP datagrams that were discarded for reasons other than errors.
- The number of IP datagrams discarded because no route could be found to transmit them to their destination.
- The maximum number of seconds that received fragments are held while they are awaiting reassembly at this entity.
- The number of IP fragments received that needed to be reassembled at this entity.
- The number of IP datagrams successfully reassembled.
- The number of failures detected by the IP reassembly algorithm.
- The number of IP datagrams that have been successfully fragmented at this entity.
- The number of IP datagrams that have been discarded at this entity because they could not be fragmented.
- The number of IP datagrams that have been created as a result of fragmentation at this entity.

Reading the ICMP SNMP Counters Function

Use **IO$_SENSEMODE | IO$M_CTRL** with function=5 to read the ICMP SNMP counters.

The data returned is an array of longwords in the following format:

- The total number of ICMP messages received.
- The number of ICMP messages received but determined as having ICMP-specific errors.
- The number of ICMP Destination Unreachable messages received.
- The number of ICMP Time Exceeded messages received.
- The number of ICMP Parameter Problem messages received.
- The number of ICMP Source Quench messages received.
- The number of ICMP Redirect messages received.
- The number of ICMP Echo (request) messages received.
- The number of ICMP Echo reply messages received.
- The number of ICMP Timestamp (request) messages received.
- The number of ICMP Timestamp Reply messages received.
- The number of ICMP Address Mask Request messages received.
- The number of ICMP Address Mask Reply messages received.
- The total number of ICMP messages that this entity attempted to send.
- The number of ICMP messages that this entity did not send because of ICMP-related problems.

- The number of ICMP Destination Unreachable messages sent.
- The number of ICMP Time Exceeded messages sent.
- The number of ICMP Parameter Problem messages sent.
- The number of ICMP Source Quench messages sent.
- The number of ICMP Redirect messages sent.
- The number of ICMP Echo (request) messages sent.
- The number of ICMP Echo reply messages sent.
- The number of ICMP Timestamp (request) messages sent.
- The number of ICMP Timestamp Reply messages sent.
- The number of ICMP Address Mask Request messages sent.
- The number of ICMP Address Mask Reply messages sent.

Reading the TCP SNMP Counters Function

Use **IO$_SENSEMODE | IO$M_CTRL** with function=6 to read TCP SNMP counters.

The data returned is an array of longwords in the following format:

- The algorithm used to determine the timeout value for retransmitting unacknowledged octets.
- The minimum value (measured in milliseconds) permitted by a TCP implementation for the retransmission timeout.
- The maximum value (measured in milliseconds) permitted by a TCP implementation for the retransmission timeout.
- The limit on the total number of TCP connections supported.
- The number of times TCP connections have made a transition to the SYN-SENT state from the CLOSED state.
- The number of times TCP connections have made a direct transition to the SYN-REVD state from the LISTEN state.
- The number of failed connection attempts.
- The number of resets that have occurred.
- The number of TCP connections having a current state of either ESTABLISHED or CLOSE-WAIT.
- The total number of segments received.
- The total number of segments sent.
- The total number of segments retransmitted.

Reading the UDP SNMP Counters Function

Use **IO$_SENSEMODE | IO$M_CTRL** with function=7 to read the UDP SNMP counters.

The data returned is an array of longwords in the following format:

- The total number of IDP datagrams delivered to UDP users.
- The total number of received UDP datagrams for which there was not an application at the destination port.
- The number of received UDP datagrams that could not be delivered for reasons other than the lack of an application at the destination port.
- The total number of UDP datagrams sent from this entity.

IO$_SETCHAR

IO$_SETCHAR — Sets special characteristics that control the operation of the INET: device, rather than the socket attached to it. These operations are normally used by only the `IP_SERVER` process to hand off a connection to a process that it creates to handle the connection.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SETCHAR, IOSB, AstAdr, AstPrm, Flags, 0, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Flags	
OpenVMS Usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

A bit mask of one or more of the following values. If **IO$_SETCHAR** is not called, all options are set to **OFF**.

```
#define SETCHAR_PERMANENT (1<<0)
#define SETCHAR_SHAREABLE (1<<1)
#define SETCHAR_HANDOFF (1<<2)
```

If the SETCHAR_PERMANENT bit is set when the last channel to the socket device is deassigned using the SYS$DASSGN system service, the socket is not closed and the socket device is not deleted.

Normally, the last deassign closes the socket. If this bit has been set, it must be explicitly cleared before the socket can be deleted.

If the SETCHAR_SHAREABLE bit is set, the socket becomes a shareable device and any process can assign a channel to it.

If the SETCHAR_HANDOFF bit is set, the socket is not closed and the socket device is not deleted when the last channel to the socket device is deassigned. After this occurs, the socket reverts to a normal socket, and if a new channel is assigned and deassigned, the socket is closed. The SETCHAR_HANDOFF bit is a safer version of the SETCHAR_PERMANENT bit because it allows a single hand-off to another process without the risk of a socket getting permanently stuck on your system.

IO$_SETMODE|IO$M_ATTNAST

IO$_SETMODE|IO$M_ATTNAST — Enables an AST to be delivered to your process when outof-band data arrives on a socket. This is similar to the UNIX 4.3BSD SIGURG signal being delivered. You cannot enable the delivery of the AST through the socket library functions. After the AST is delivered, you must explicitly reenable it using this call if you want the AST to be delivered when future out-of-band data arrives.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SETMODE|IO$M_ATTNAST, IOSB, AstAdr, Ast-Prm, Routine, Parameter, 0, 0, 0, 0);

Arguments

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

The address of the AST routine to call when out-of-band data arrives on the socket. To disable AST delivery, set **Routine** to 0.

Parameter	
OpenVMS Usage:	**user_arg**

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

The argument with which to call the AST routine.

IO$_SETSOCKOPT

IO$_SETSOCKOPT — Manipulates options associated with a socket. It is equivalent to the **setsockopt()** socket library function. Options may exist at multiple protocol levels; however, they are always present at the uppermost socket level. When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, specify **Level** as **SOL_SOCKET**. To manipulate options at any other level, specify the protocol number of the appropriate protocol controlling the option. For example, to indicate that an option is to be interpreted by the TCP protocol, set **Level** to the protocol number of TCP; see **getprotobyname()**.

OptName and any specified options are passed without modification to the appropriate protocol module for interpretation. The include file `IP_root:[IP.include.sys]socket.h` contains definitions for socket-level options. Options at other protocol levels vary in format and name.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SETSOCKOPT, IOSB, AstAdr, AstPrm, Level, Opt-Name, OptVal, OptLen, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

Level	
OpenVMS Usage:	**option_level**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The protocol level at which the option will be manipulated. Specify **Level** as **SOL_SOCKET**, or a protocol number as returned by **getprotobyname()**.

OptName	
OpenVMS Usage:	**option_name**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The option that is to be manipulated. For a description of each of the valid options for **IO$_SETSOCKOPT**, see the **socket option** sections.

OptVal	
OpenVMS Usage:	**dependent on OptName**
type:	**byte buffer**
access:	**read only**
mechanism:	**by reference**

A pointer to a buffer that contains the new value of the option. The format of this buffer depends on the option requested.

OptLen	
OpenVMS Usage:	**option_length**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The length of the buffer pointed to by **OptVal**.

IO$_SHUTDOWN

IO$_SHUTDOWN — Shuts down all or part of a full-duplex connection on the socket associated with VMS_Channel. This function is usually used to signal an end-of-file to the peer without closing the socket itself, which would prevent further data from being received. It is equivalent to the **shutdown()** socket library function.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SHUTDOWN, IOSB, AstAdr, AstPrm, How, 0, 0, 0, 0, 0);

Arguments

VMS_Channel	
OpenVMS Usage:	**channel**
type:	**word (signed)**
access:	**read only**
mechanism:	**by value**

A channel to the socket.

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

Controls which part of the full-duplex connection to shut down, as follows: if **How** is 0, further receive operations are disallowed; if **How** is 1, further send operations are disallowed; if **How** is 2, further send and receive operations are disallowed.

IO$_SOCKET

IO$_SOCKET — Creates an end point for communication and returns an OpenVMS channel that describes the end point. It is equivalent to the **socket()** socket library function. Before issuing the **IO$_SOCKET** call, an OpenVMS channel must first be assigned to the INET0: device to get a new channel to the network.

Format

Status = SYS$QIOW(Efn, VMS_Channel, IO$_SOCKET, IOSB, AstAdr, AstPrm, Address_Family, Type, Protocol, 0, 0, 0);

Arguments

Address_Family	
OpenVMS Usage:	**address_family**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

An address family with which addresses specified in later operations using the socket will be interpreted.

The following formats are currently supported; they are defined in the include file `IP_root:[IP.include.sys]socket.h`:

AF_INET	Internet (TCP/IP) addresses
AF_PUP	Xerox PUP addresses
AF_CHAOS	CHAOSnet addresses

Type	
OpenVMS Usage:	**socket_type**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

The semantics of communication using the created socket. The following types are currently defined:

SOCK_STREAM	SOCK_DGRAM	SOCK_RAW

A **SOCK_STREAM** socket provides a sequenced, reliable, two-way connection-oriented byte stream with an out-of-band data transmission mechanism.

A **SOCK_DGRAM** socket supports communication by connectionless, unreliable messages of a fixed (typically small) maximum length.

SOCK_RAW sockets provide access to internal network interfaces. The type **SOCK_RAW** is available only to users with **SYSPRV** privilege.

The **Type** argument, together with the **Address_Family** argument, specifies the protocol to be used.

For example, a socket created with **AF_INET** and **SOCK_STREAM** is a TCP socket, and a socket created with **AF_INET** and **SOCK_DGRAM** is a UDP socket.

Protocol	
OpenVMS Usage:	**protocol_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

A protocol to be used with the socket. Normally, only a single protocol exists to support a particular socket type using a given address format. However, many protocols may exist, in which case a particular protocol must be specified by **Protocol**. The protocol number to use depends on the communication domain in which communication will take place. For TCP and UDP sockets, the protocol number MUST be specified as 0. For **SOCK_RAW** sockets, the protocol number should be the value returned by **getprotobyname()**.

SYS$CANCEL

SYS$CANCEL — Cancels any I/O IOSB status of **SS$_CANCEL**. Outstanding I/O operations are automatically cancelled at image exit. For more information on **SYS$CANCEL**, see the *OpenVMS System Services Reference Manual*.

Format

Status = SYS$CANCEL(VMS_Channel);

SYS$DASSGN

SYS$DASSGN — Equivalent to the **socket_close()** function. When you deassign a channel, any outstanding I/O is completed with an **IOSB** status of **SS$_CANCEL**. Deassigning a channel closes the network connection. I/O channels are automatically deassigned at image exit. For more information on **SYS$DASSGN**, see the *OpenVMS System Services Reference Manual*.

Format

Status = SYS$DASSGN(VMS_Channel);

Socket Options

This section describes the socket options that you can set with the Sockets API setsockopt() function and the $QIO system service IO$_SETMODE and IO$_SETCHAR I/O function codes. You can query the value of these socket options using the Sockets API getstockopt() function or the $QIO system service IO$_SENSEMODE or IO$_SENSECHAR I/O function code.

The following tables list:

- Socket Options
- TCP Protocol Options
- IP Protocol Options
- IPv6 Socket Options

The following table lists the socket options that are set at the SOL_SOCKET level and their Sockets API and system service symbol names.

Table 8.9 Socket Options

Sockets API Symbol	System Service Symbol	Description
SO_BROADCAST	TCPIP$C_BROADCAST	Permits the sending of broadcast messages. Takes an integer parameter and requires a system user identification code (UIC) or SYSPRV, BYPASS, or OPER privilege. Optional for a connectionless datagram.
SO_DONTROUTE	TCPIP$C_DONTROUTE	Indicates that outgoing messages should bypass the standard routing facilities. Instead, the messages are directed to the appropriate network interface according to the network portion of the destination address.
SO_ERROR	TCPIP$C_ERROR	Obtains the socket error status and clears the error on the socket.
SO_FULL_DUPLEX_CLOSE	TCPIP$C_FULL_DUPLEX_CLOSE	When set before a close operation, the receive and transmit sides of the communications are closed.
SO_KEEPALIVE	TCPIP$C_KEEPALIVE	Keeps connections active. Enables the periodic transmission of keepalive probes to the remote system. If the remote system fails to respond to the keepalive probes, the connection is broken. If the SO_KEEPALIVE option is enabled, the values of TCP_KEEPCNT, TCP_KEEPINTVL and TCP_KEEPIDLE affect TCP behavior on the socket.
SO_LINGER	TCPIP$C_LINGER	Lingers on a close() function if data is present. Controls the action taken when unsent messages queue on a socket and a close() function is performed. Uses a lingerstructure parameter defined in SOCKET.H to specify the state of the option and the linger interval. If SO_LINGER is specified, the system blocks the process during the close() function until it can transmit the data or

		until the time expires. If the option is not specified and aclose() function is issued, the system allows the process to resume as soon as possible.
SO_OOBINLINE	TCPIP$C_OOBINLINE	When this option is set, out-of-band data is placed in the normal input queue. When SO_OOBINLINE is set, the MSG_OOB flag to the receive functions cannot be used to read the out-of-band data. A value of 0 disables the option, and a nonzero value enables the option.
SO_RCVBUF	TCPIP$C_RCVBUF	Sets the receive buffer size, in bytes. Takes an integer parameter and requires a system UIC or SYSPRV, BYPASS, or OPER privilege.
SO_RCVTIMEO	TCPIP$C_RCVTIMEO	For VSI use only. Sets the timeout value for a recv() operation. The argument to the two sockopt functions is a pointer to a timeval structure containing an integer value specified in seconds.
SO_REUSEADDR	TCPIP$C_REUSEADDR	Specifies that the rules used in validating addresses supplied by a bind() function should allow reuse of local addresses. A value of 0 disables the option, and a non-zero value enables the option. The SO_REUSEPORT option is automatically set when an application sets SO_REUSEADDR
SO_REUSEPORT	TCPIP$C_REUSEPORT	Allows more than one process to receive UDP datagrams destined for the same port. The bind() call that binds a process to the port must be preceded by a setsockopt() call specifying this option. SO_REUSEPORT is automatically set when an application sets the SO_REUSEADDR option.
SO_SHARE	TCPIP$C_SHARE	Allows multiple processes to share the socket.
SO_SNDBUF	TCPIP$C_SNDBUF	Sets the send buffer size in bytes. Takes an integer parameter and requires a system UIC or SYSPRV, BYPASS, or OPER privilege. Optional for a connectionless datagram.
SO_SNDLOWAT	TCPIP$C_SNDLOWAT	Sets the low-water mark for a send() operation. The send low-water mark is the amount of space that must exist in the socket send buffer for select() to return writeable. Takes an integer value specified in bytes.
SO_SNDTIMEO	TCPIP$C_SNDTIMEO	For VSI use only. Sets the timeout value for a send() operation. The

		argument to the two sockopt() functions is a pointer to a timeval structure containing an integer value specified in seconds.
SO_TYPE	TCPIP$C_TYPE	Obtains the socket type.
SO_USELOOPBACK	TCPIP$C_USELOOPBACK	For VSI use only. This option applies only to sockets in the routing domain (AF_ROUTE), When you enable this option, the socket receives a copy of everything sent on the socket.

The following table lists the TCP protocol options that are set at the IPPROTO_TCP level and their Sockets API and system service symbol names.

Table 8.10 TCP Protocol Options

Sockets API Symbol	System Service Symbol	Description
TCP_KEEPCNT	TCPIP$C_TCP_KEEPCNT	When the SO_KEEPALIVE option is enabled, TCP sends a keepalive probe to the remote system of a connection that has been idle for a period of time. If the remote system does not respond to the keepalive probe, TCP retransmits a keepalive probe for a certain number of times before a connection is considered to be broken. The TCP_KEEPCNT option specifies the maximum number of keepalive probes to be sent. The value of TCP_KEEPCNT is an integer value between 1 and n, where n is the value of the systemwide tcp_keepcnt parameter. The default value for for the systemwide parameter, tcp_keepcnt , is To display the values of the systemwide parameters, enter the following command at the system prompt: `$ sysconfig -q inet` The default value for TCP_KEEPCNT is 8.
TCP_KEEPIDLE	TCPIP$C_TCP_KEEPIDLE	When the SO_KEEPALIVE option is enabled, TCP sends a keepalive probe to the remote system of a connection that has been idle for a period of time. If the remote system does not respond to the keepalive probe, TCP retransmits a keepalive probe for a certain number of times before a connection is considered to be broken. TCP_KEEPIDLE specifies the number of seconds before TCP will send the initial keepalive probe. The default value for TCP_KEEPIDLE is an integer value between 1 and n, where n is the value for the systemwide parameter tcp_keepidle . The default

		value for tcp_keepidle , specified in half-second units, is 150 (75 seconds). To display the values of the systemwide parameters, enter the following command at the system prompt: `$ sysconfig -q inet` The default value for TCP_KEEPIDLE is 75 seconds.
TCP_KEEPINIT	TCPIP$C_TCP_KEEPINIT	If a TCP connection cannot be established within a period of time, TCP will time out the connection attempt. The default timeout value for this initial connection establishment is 75 seconds. The TCP_KEEPINIT option specifies the number of seconds to wait before the connection attempt times out. For passive connections, the TCP_KEEPINIT option value is inherited from the listening socket. The value of TCP_KEEPINIT is an integer between 1 and n, where n is the value for the systemwide parameter tcp_keepinit . The default value of the systemwide parameter tcp_keepinit , specified in half-second units, is 150 (75 seconds). To display the values of the systemwide parameters, enter the following command at the system prompt: `$ sysconfig -q inet` The TCP_KEEPINIT option does not require the SO_KEEPALIVE option to be enabled.
TCP_KEEPINTVL	TCPIP$C_TCP_KEEPINTVL	When the SO_KEEPALIVE option is enabled, TCP sends a keepalive probe to the remote system on a connection that has been idle for a period of time. If the remote system does not respond to a keepalive probe, TCP retransmits the keepalive probe after a period of time. The default value for this retransmit interval is 75 seconds. The TCP_KEEPINTVL option specifies the number of seconds to wait before retransmitting a keepalive probe. The value of the TCP_KEEPINTVL option is an integer between 1 and n, where n is the value of the systemwide parameter tcp_keepintvl which is specified in half-second units. The default value for the

		systemwide parameter tcp_keepintvl is 150 (75 seconds). To display the values of the systemwide parameters, enter the following command at the system prompt: `$ sysconfig -q inet`
TCP_NODELAY	TCPIP$C_TCP_NODELAY	Specifies that the send() operation not be delayed to merge packets. Under most circumstances, TCP sends data when it is presented. When outstanding data has not yet been acknowledged, TCP gathers small amounts of the data into a single packet and sends it when an acknowledgment is received. This functionality can cause significant delays for some clients that do not expect replies (such as windowing systems that send a stream of events from the mouse). The TCP_NODELAY disables the Nagle algorithm, which reduces the number of small packets on a wide area network.
TCP_MAXSEG	TCPIP$C_TCP_MAXSEG	Sets the maximum transmission unit (MTU) of a TCP segment to a specified integer value from 1 to 65535. The default is 576 bytes. Can only be set before a listen() or connect() operation on the socket. For passive connections, the value is obtained from the listening socket. Note that TCP does not use an MTU value that is less than 32 or greater than the local network's MTU. Setting the option to zero results in the default behavior.
TCP_NODELACK	TCPIP$C_TCP_NODELACK	When specified, disables the algorithm that gathers outstanding data that has not been acknowledged and sends it in a single packet when acknowledgment is received. Takes an integer value.
The following TCP protocol options are obsolete but provided for backward compatibility:		
TCP_DROP_IDLE	TCPIP$C_TCP_DROP_IDLE	When the TCP_KEEPALIVE option is enabled, the TCP_DROP_IDLE option specifies the time interval after which a connection is dropped. The value of TCP_DROP_IDLE is an integer specified in seconds. The default value is 600 seconds. When the TCP_DROP_IDLE option is set, the value of the TCP_KEEPCNT

		option is calculated as the value of TCP_DROP_IDLE divided by the value of TCP_KEEPINTVL. A call to getsockopt() function specifying the TCP_DROP_IDLE option returns the result of multiplying the values of TCP_KEEPCNT and TCP_KEEPINTVL.
TCP_PROBE_IDLE	TCPIP$C_TCP_PROBE_IDLE	When the TCP_KEEPALIVE option is enabled, the TCP_PROBE_IDLE option specifies the time interval between the keepalive probes and for the connections establishing the timeout. The default value for TCP_PROBE_IDLE is 75 seconds. The value of TCP_PROBE_IDLE is an integer specified in seconds. When this option is set, TCP_KEEPINTVL, TCP_KEEPIDLE and TCP_KEEPINIT are set to the value specified for TCP_PROBE_IDLE. A call to the getsockopt() function specifying the TCP_PROBE_IDLE option returns the value of TCP_KEEPINTVL.

The following table lists options that are set at the IPPROTO_IP level and their Sockets API and system service symbol names.

Table 8.11 Protocol Options

Sockets API Symbol	System Service Symbol	Description
IP_ADD_MEMBERSHIP	TCPIP$C_IP_ADD_ MEMBERSHIP	Adds the host to the membership of a multicast group. A host must become a member of a multicast group before it can receive datagrams sent to the group. Membership is associated with a single interface; programs running on multihomed hosts may need to join the same group on more than one interface. Up to IP_MAX_MEMBERSHIPS (currently 20) memberships may be added on a single socket.
IP_DROP _MEMBERSHIP	TCPIP$ C_IP_DROP_MEMBERSHIP	Removes the host from the membership of a multicast group.
IP_HDRINCL	TCPIP$C_IP_HDRINCL	If specified for a raw IP socket, you must build the IP header for all datagrams sent on the raw socket.
IP_MULTICAST_IF	TCPIP$C_IP_MULTICAST_IF	Specifies the interface for outgoing multicast datagrams sent on this socket. The interface is specified as an in_addr structure.

<table>
<tr><td>IP_MULTICAST_LOOP</td><td>TCPIP$C_IP_MULTICAST_
LOOP</td><td colspan="2">Disables loopback of local delivery. If a multicast datagram is sent to a group which the sending host is a member, a copy of the datagram is looped back by the IP layer for local delivery (the default). To disable the loopback delivery, specify a value of 0.</td></tr>
<tr><td rowspan="4">IP_MULTICAST_TTL</td><td rowspan="4">TCPIP$C_IP_MULTICAST_
TTL</td><td colspan="2">Specifies the time-to-live (TTL) value for outgoing multicast datagrams. Takes an integer value between 0 and 255:</td></tr>
<tr><td>Value</td><td>Action</td></tr>
<tr><td>0</td><td>Restricts distribution to applications running on the local host.</td></tr>
<tr><td>1</td><td>Forwards the multicast datagram to hosts on the local subnet.</td></tr>
<tr><td>IP_OPTIONS</td><td>TCPIP$C_IP_OPTIONS</td><td colspan="2">Provides IP options to be transmitted in the IP header of each outgoing packet.</td></tr>
<tr><td>IP_RECVDSTADDR</td><td>TCPIP$C_IP_
RECVDSTADDR</td><td colspan="2">Enables a SOCK_DGRAM socket to receive the destination IP address for a UDP datagram.</td></tr>
<tr><td>IP_RECVOPTS</td><td>TCPIP$C_IP_RECVOPTS</td><td colspan="2">Enables a SOCK_DGRAM socket to receive IP options.</td></tr>
<tr><td>IP_TTL</td><td>TCPIP$C_IP_TTL</td><td colspan="2">Time to live (TTL) for a datagram.</td></tr>
<tr><td>IP_TOS</td><td>TCPIP$C_IP_TOS</td><td colspan="2">Type of service (1-byte value).</td></tr>
</table>

The following table describes the socket options supporting IPv6. The IPv6 socket options do not have system service symbols.

Table 8.12 IPv6 Socket Options

Option	Description
IPV6_RECVPKTINFO	Source and destination IPv6 address, and sending and receiving interface.
IPV6_RECVHOPLIMIT	Hop limit.
IPV6_RECVRTHDR	Routing header.
IPV6_RECVHOPOPTS	Hop-by-hop options.
IPV6_RECVDSTOPTS	Destination options.

IPV6_CHECKSUM	For raw IPv6 sockets other than ICMPv6 raw sockets, causes the kernel to compute and store checksum for output and to verify the received checksum on input. Discards the packet if the checksum is in error.
IPV6_ICMP6_FILTER	Fetches and stores the filter associated with the ICMPv6 raw socket using the getsockopt( ) function and setsockopt( ) functions.
IPV6_UNICAST_HOPS	Sets the hop limit for all subsequent unicast packets sent on a socket. You can also use this option with the getsockopt( ) function to determine the current hop limit for a socket.
IPV6_MULTICAST_	IF Sets the interface to use for outgoing multicast packets.
IPV6_MULTICAST_HOPS	Sets the hop limit for outgoing multicast packets.
IPV6_MULTICAST_LOOP	Controls whether to deliver outgoing multicast packets back to the local application.
IPV6_JOIN_GROUP	Joins a multicast group on the specified interface
IPV6_LEAVE_GROUP	Leaves a multicast group on the specified interface.

Legal Notice