VMS Software

VSI OpenVMS

VSI TCP/IP Services for OpenVMS Guide to IPv6

Document Number: DO-TCPIPV-01A

Publication Date: September 2021

Revision Update Information: This is a new manual.

Operating System and Version: VSI OpenVMS Integrity Version 8.4-2
VSI OpenVMS Alpha Version 8.4-2L1

Software Version: VSI TCP/IP Services Version 5.7

VMS Software, Inc. (VSI)
Burlington, Massachusetts, USA

VSI TCP/IP Services for OpenVMS Guide to IPv6

VMS Software

Preface

The *VSI TCP/IP Services for OpenVMS* product is the VSI implementation of the TCP/IP networking protocol suite and internet services for OpenVMS systems.

TCP/IP Services provides a comprehensive suite of functions and applications that support industry-standard protocols for heterogeneous network communications and resource sharing.

This manual describes IPv6 features included in this version of TCP/IP Services. The manual covers installing and configuring your system for IPv6, changes to the socket API, and how to port your applications to run in an IPv6 environment.

See the *VSI TCP/IP Services for OpenVMS Installation and Configuration* manual for information about installing, configuring, and starting this product.

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

VSI seeks to continue the legendary development prowess and customer-first priorities that are so closely associated with the OpenVMS operating system and its original author, Digital Equipment Corporation.

2. Intended Audience

This manual is for experienced OpenVMS and UNIX system managers and assumes a working knowledge of OpenVMS system management, TCP/IP networking, and TCP/IP terminology.

3. Document Structure

This manual contains the following chapters and appendixes:

Chapter 1	Describes IPv6 terminology, the types and function of the different IPv6 addresses, and typical IPv6 configurations.
Chapter 2	Describes how to configure the IPv6 software.
Chapter 3	Provides guidelines for running BIND in an IPv6 environment.
Chapter 4	Describes how to manage and monitor an IPv6 network.
Chapter 5	Describes how to configure and use mobile IPv6.
Chapter 6	Describes how to solve IPv6 problems.
Chapter 7	Describes the IPv6 additions to the socket API.
Chapter 8	Describes how to port applications.
Appendix A	Describes the supported IPv6 RFCs.
Appendix B	Describes deprecated functions that have been replaced by new ones.

4. Related Documents

The table below lists the documents available with this version of TCP/IP Services.

Table 1. TCP/IP Services Documentation

Manual	Contents
VSI TCP/IP Services for OpenVMS Concepts and Planning	This manual provides conceptual information about TCP/IP networking on OpenVMS systems, including general planning issues to consider before configuring your system to use the TCP/IP Services software. This manual also describes the manuals in the TCP/IP Services documentation set and provides a glossary of terms and acronyms for the TCP/IP Services software product.
VSI TCP/IP Services for OpenVMS Installation and Configuration	This manual explains how to install and configure the TCP/IP Services product.
VSI TCP/IP Services for OpenVMS User's Guide	This manual describes how to use the applications available with TCP/IP Services such as remote file operations, email, TELNET, TN3270, and network printing.
VSI TCP/IP Services for OpenVMS Management	This manual describes how to configure and manage the TCP/IP Services product.
VSI TCP/IP Services for OpenVMS Management Command Reference	This manual describes the TCP/IP Services management commands.
VSI TCP/IP Services for OpenVMS ONC RPC Programming	This manual presents an overview of high-level programming using open network computing remote procedure calls (ONC RPCs). This manual also describes the RPC programming interface and how to use the RPCGEN protocol compiler to create applications.
VSI TCP/IP Services for OpenVMS Sockets API and System Services Programming	This manual describes how to use the Sockets API and OpenVMS system services to develop network applications.
VSI TCP/IP Services for OpenVMS SNMP Programming and Reference	This manual describes the Simple Network Management Protocol (SNMP) and the SNMP application programming interface (eSNMP). It describes the subagents provided with TCP/IP Services, utilities provided for managing subagents, and how to build your own subagents.
VSI TCP/IP Services for OpenVMS Guide to IPv6	This manual describes the IPv6 environment, the roles of systems in this environment, the types and function of the different IPv6 addresses, and how to configure TCP/IP Services to access the IPv6 network.

5. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: `<docinfo@vmssoftware.com>`. Users who have OpenVMS support contracts through VSI can contact `<support@vmssoftware.com>` for help with this product.

6. Conventions

The following conventions may be used in this manual:

Convention	Meaning
Ctrl/ *x*	A sequence such as **Ctrl**/ *x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
`...`	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
[|]	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are options; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold text	This typeface represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER= *name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace type`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.

Convention	**Meaning**
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Other conventions are:

- All numbers are decimal unless otherwise noted.
- All Ethernet addresses are hexadecimal.

Chapter 1. What Is IPv6?

In the early 1990s, members of the Internet community realized that the address space and certain aspects of the current TCP/IP architecture were not capable of sustaining the explosive growth of the Internet. The problems included the exhaustion of the Internet address space, the size of routing tables, and requirements for new technology features.

The Internet Engineering Task Force (IETF) made several efforts to study and improve the use of the 32-bit Internet Protocol (IPv4) addresses. They also tackled the longer-term goal of identifying and replacing protocols and services that would limit growth.

These efforts identified the 32-bit addressing architecture of IPv4 as the principal problem affecting router overhead and network administration. In addition, IPv4 addresses were often unevenly allocated in blocks that were too large or too small; therefore, these addresses were difficult to change within any existing network.

In July 1994, the Internet Protocol Next Generation (IPng) directorate announced Internet Protocol Version 6 (IPv6) as the replacement network layer protocol, and IETF working groups began to build specifications. (See RFC 1752, *The Recommendation for the IP Next Generation Protocol*, for additional information about the IPv6 protocol selection process.)

IPv6 is both a completely new network layer protocol and a major revision of the Internet architecture. As such, it builds upon and incorporates experience gained with IPv4. This chapter describes the following:

- Terminology
- IPv6 addressing
- Using tunnels
- IPv6 environment

1.1. Terminology

The following terms are used in this chapter:

- **Node**

  Any system that uses the IPv6 protocol to communicate.

- **Router**

  A node that forwards IPv6 packets addressed to other nodes. These systems typically have more than one network interface installed and configured.

- **Host**

  Any system that is not a router.

- **Link**

  A medium or facility over which nodes communicate with each other at the Link layer. Examples include Ethernet, FDDI links, or internet layer tunnels.

- **Interface**

  A node's attachment to a link, which is usually assigned an IPv6 address or addresses. This can be a physical NIC (for example, WE0) or virtual network interface (for example, IT0).

- **Tunnel**

  A link over which a packet of one protocol is encapsulated inside the packet of another protocol. In this manner, one protocol's packets an be carried over another protocol's infrastructure. The process for doing this is called tunneling. See Section 1.4 for more information on the types of tunnels that are available for you to use.

1.2. Introduction to IPv6 Addresses

The most noticeable feature of IPv6 is the address itself. The address size is increased from 32 bits to 128 bits. The following sections describe the components of the IPV6 address.

1.2.1. Address Text Representation

Use the following syntax to represent IPv6 addresses as text strings:

```
x:x:x:x:x:x:x:x
```

The *x* is a hexadecimal value of a 16-bit piece of the address. For example, the following addresses are IPv6 addresses:

```
FEDC:BA98:7654:3210:FEDC:BA98:7654:3210
```

```
1070:0:0:0:0:800:200C:417B
```

IPv6 addresses can contain long strings of zero (0) bits. To make it easier to write these addresses, you can use a double colon (::) once in an address to represent one or more 16-bit groups of zeros. For example, you can compress the second IPv6 address example in the following way:

```
1070::800:200C:417B
```

Alternately, you can use the following syntax to represent IPv6 addresses in an environment of both IPv4 and IPv6 nodes:

```
x:x:x:x:x:x:d.d.d.d
```

In this case, *x* is a hexadecimal value of a 16-bit piece of the address (six high-order pieces) and *d* is a decimal value of an 8-bit piece of address (four low-order pieces) in standard, dotted-quad IPv4 form. For example, the following are IPv6 addresses:

```
0:0:0:0:0:0:13.1.68.3
```

```
0:0:0:0:0:FFFF:129.144.52.38
```

When compressed, these addresses are as follows:

```
::13.1.68.3
```

```
::FFFF:129.144.52.38
```

Like IPv4 address prefixes, IPv6 address prefixes are represented using the Classless Inter-Domain Routing (CIDR) notation. This notation has the following format:

```
ipv6-address/prefix-length
```

For example, you can represent the 60-bit hexadecimal prefix 12AB00000000CD3 in any of the following ways:

```
12AB:0000:0000:CD30:0000:0000:0000:0000/60
```

```
12AB::CD30:0:0:0:0/60
```

```
12AB:0:0:CD30::/60
```

1.2.2. Types of Addresses

There are three types of IPv6 addresses:

- Unicast
- Anycast
- Multicast

Note

Unlike IPv4, IPv6 does not define a broadcast address. To get the function of a broadcast address, use a multicast address. (See Section 1.2.2.3.)

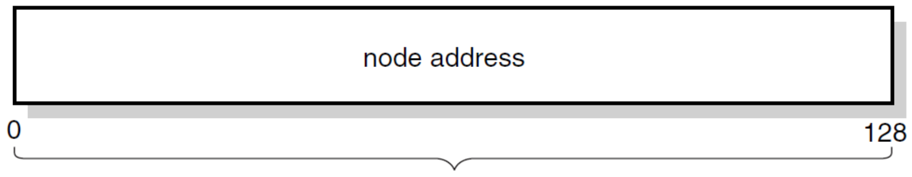
The following sections describe the unicast, anycast, and multicast address types.

1.2.2.1. Unicast Addresses

A unicast address is an identifier for an interface. Packets sent to a unicast address are delivered to the node containing the interface that is identified by the address.

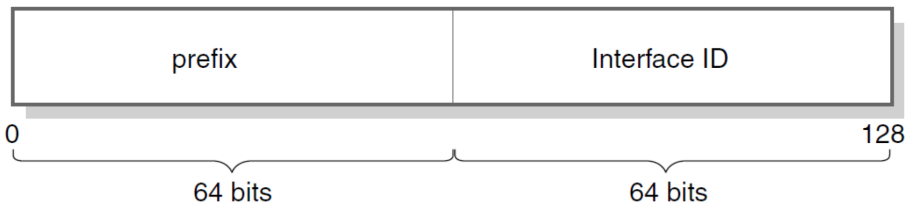
Figure 1.1 shows the format of unicast addresses.

Figure 1.1. Unicast Addresses



This address typically consists of a 64-bit prefix followed by a 64-bit interface ID, as shown in Figure 1.2.

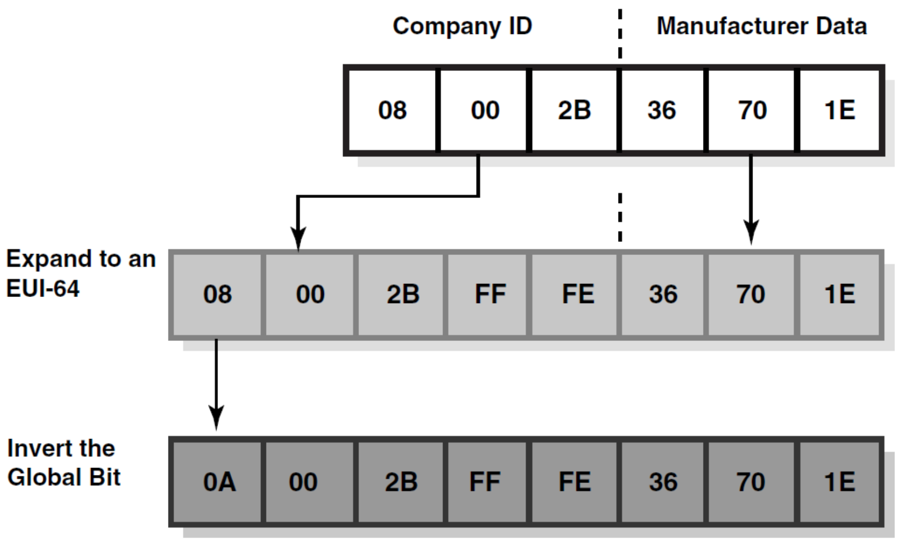
Figure 1.2. 64-Bit Prefix Plus 64-Bit Interface ID



An interface ID identifies an interface on a link. The interface ID is required to be unique on a link, but it may also be unique over a broader scope. In many cases, the interface ID is derived from its Link layer address. The same interface ID can be used on multiple interfaces on a single node.

According to RFC2373, most prefixes must have 64-bit interface identifiers. For a 48-bit MAC addresses, the interface identifier is created by inserting the hexadecimal values of 0xFF and 0xFE in the middle of the address and inverting the universal/local bit (bit 7) in the resulting 64-bit address. Figure 1.3 shows how this process works.

Figure 1.3. Creating an Interface ID from a MAC Address



The following list describes commonly used unicast addresses and their values:

- Unspecified address

  Indicates the absence of an address and is never assigned to an interface. The unspecified address has the following value:

  ```
  0:0:0:0:0:0:0:0 (normal form)
  ```

  ```
  :: (compressed form)
  ```

- Loopback address

  Used by a node to send IP datagrams to itself and is typically assigned to the loopback interface.

  The IPv6 loopback address has the following value:

  ```
  0:0:0:0:0:0:0:1 (normal form)
  ```

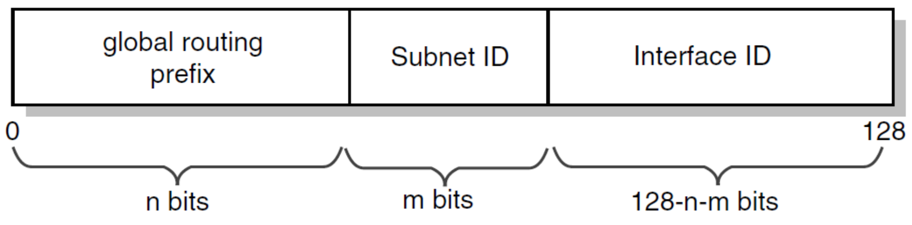
  ```
  ::1 (compressed form)
  ```

- Global unicast addresses

  These addresses are globally routable.

  Figure 1.4 shows the format of the global unicast address.

Figure 1.4. IPv6 Global Unicast Address



RFC 2374 defined aggregatable global unicast address formats that included Top Level Aggregator (TLA) and Next Level Aggregator (NLA). RFC 3587 replaces RFC 2374, and makes it and the TLS/NLA structure historic.

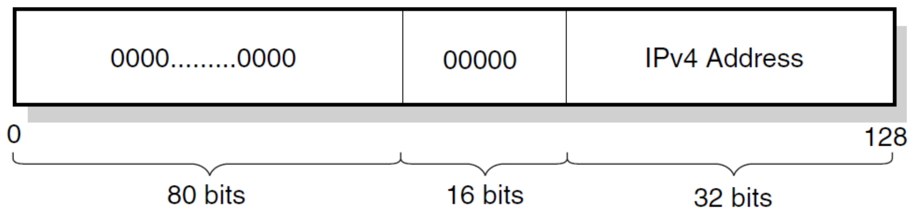
- IPv6 addresses with embedded IPv4 addresses

  Used in mixed IPv4 and IPv6 environments and can be either of the following:

  - IPv4-compatible IPv6 address

    Used by IPv6 nodes to tunnel IPv6 packets across an IPv4 routing infrastructure. The IPv4 address is carried in the low-order 32 bits. Figure 1.5 shows the format of the IPV4-compatible IPV6 address.

    Figure 1.5. IPv4-Compatible IPv6 Address



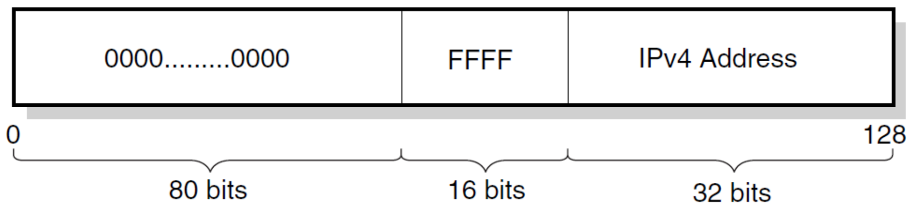
Note

Do not use IPv4-compatible IPv6 addresses in DNS or in TCPIP$ETC:IPNODES.

  - IPv4-mapped IPv6 address

    Used to represent an IPv4 address and to identify nodes that do not support IPv6. This address is not used in an IPv6 packet. Figure 1.6 shows the format of the IPv4-mapped IPv6 address.

    Figure 1.6. IPv4-Mapped IPv6 Address

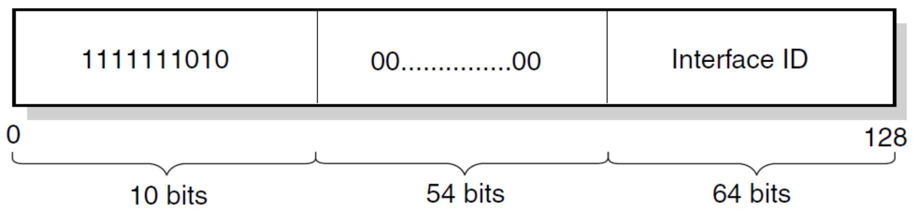


- Local-use IPv6 unicast addresses can be either of the following:

  - Link-local

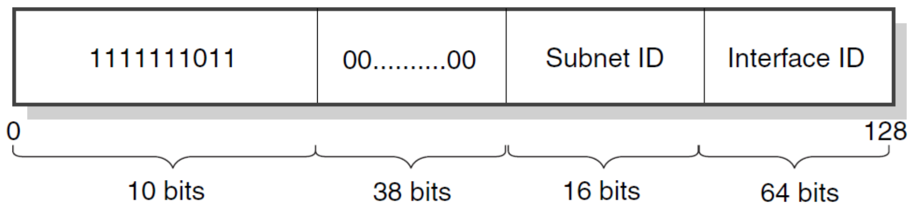
Used for addressing on a single link when performing address autoconfiguration or neighbor discovery or when no routers are present. Figure 1.7 shows the format of the link-local address.

Figure 1.7. IPv6 Link-Local Unicast Address



- Site-localUsed for sites or organizations that are not connected to the global Internet. Figure 1.8 shows the format of the site-local address.

Figure 1.8. IPv6 Site-Local Unicast Address



If you plan to use site-local addresses, be aware of the following guidelines:

- Do not connect a single node to multiple sites.
- Do not use site-local addresses in the global DNS (the addresses should not be visible outside the site).
- Dynamic DNS updates for site-local addresses are not supported.
- Do not advertise or propagate routes containing site-local prefixes outside the site.

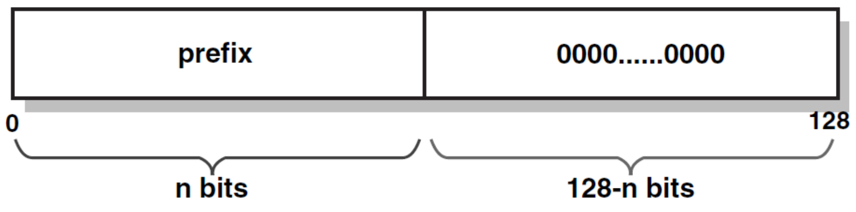
Interfaces typically have multiple IPv6 addresses. After IPv6 is configured and the system boots, the LAN and configured tunnel interfaces are automatically assigned a link-local address. If a router is on the link, the system also autoconfigures a global unicast address on the interfaces.

1.2.2.2. Anycast Addresses

An anycast address is an identifier for a set of interfaces typically belonging to different nodes. Packets sent to an anycast address are delivered to one of the interfaces identified as the "nearest" address, according to the routing protocol's measure of distance.

Anycast addresses are allocated from the unicast address space, and cannot be distinguished from unicast addresses. Only the subnet-router anycase address and addresses defined in RFC 2526 are easily identified. Packets sent to the subnet-router anycast address are delivered to the router closest to the originating host only. Figure 1.9 shows the format of anycast addresses.

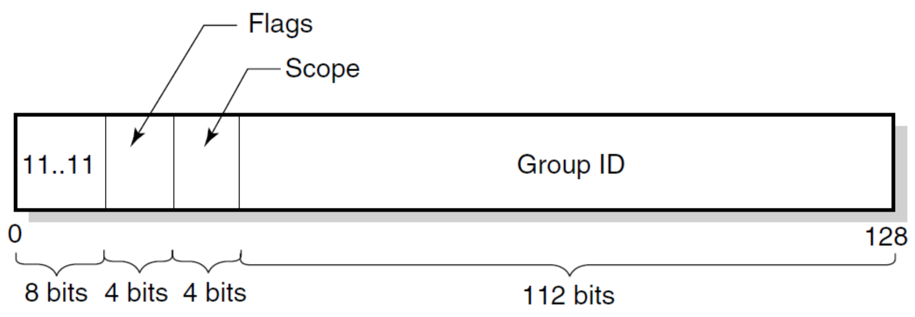
Figure 1.9. Anycast Address



1.2.2.3. Multicast Address

A multicast address is an identifier for a group of nodes. It is similar to an IPv4 multicast address. Figure 1.10 shows the format for multicast addresses.

Figure 1.10. IPv6 Multicast Address



In the multicast address format, the fields have the following definitions:

11..11	Identifies the address as multicast.
Flags	Can be either of the following values: • 0000, which indicates a permanently assigned (well-known) multicast address, • 0001, which indicates a nonpermanently assigned (transient) multicast address.
Scope	Indicates the scope of the multicast group. The following table lists the scope values:

Value (hex)	Scope
0	Reserved
1	Interface-local scope
2	Link-local scope
3	Reserved
4	Admin-local scope
5	Site-local scope
6	(unassigned)
7	(unassigned)
8	Organization-local scope
9	(unassigned)
A	(unassigned)

	B	(unassigned)
	C	(unassigned)
	D	(unassigned)
	E	Global scope
	F	Reserved
Group ID	Identifies the multicast group within the specified scope.	

Table 1.1 lists some well-known multicast addresses.

Table 1.1. Well-Known Multicast Addresses

Multicast Address	Meaning
FF01::1	All nodes (interface-local)
FF02::1	All nodes (link-local)
FF01::2	All routers (interface-local)
FF02::2	All routers (link-local)
FF05::2	All routers (site-local)
FF02::1:FFxx:xxxx	Solicited-node address

1.2.3. Address Prefixes

Each IPv6 address has a unique pattern of leading (high-order) bits that indicates its address type. Table 1.2 lists some IPv6 address types and their prefixes.

Table 1.2. IPv6 Address Types and Prefixes

Address Type	Binary Prefix	IPv6 Notation
Unspecified	00...0 (128 bits)	::/128
Loopback	00...1 (128 bits)	::1/128
Multicast	11111111	FF00::/8
Link-local unicast	1111111010	FE80::/10
Site-local unicast	1111111011	FEC0::/10
Global unicast	(everything else)	

1.2.4. Specifying IPv6 Nonglobal Addresses

The BIND resolver has been updated as described in the following RFC draft:

```
draft-ietf-ipngwg-scoping-arch-04.txt
```

This change allows the specification of an IPv6 nonglobal address without ambiguity by also specifying an intended scope zone. The format is as follows:

```
address%zone_id
```

The format of the nonglobal address includes the following:

- *address* is a literal IPv6 address.
- *zone_id* is a string to identify the zone of the address.

- % is a delimiter character to distinguish between the address and zone identifier.

For example, the following specifies a nonglobal address on interface WE0:

```
fe80::1234%WE0
```

1.2.5. Address Autoconfiguration

The IPv6 address changes have led to the following definitions for configuring addresses:

- Stateless address autoconfiguration
- Dynamic Host Configuration Protocol Version 6 (DHCPv6), which is stateful address autoconfiguration

In the stateless model, nodes learn address prefixes by listening for Router Advertisement packets. Addresses are formed by combining the prefix with a data link-specific interface token, which is typically derived from the data link address of the interface. This model is favored by administrators who do not need tight control over address configuration. See RFC 2462 for more information.

In DHCPv6, hosts may request addresses, configuration information and services from dedicated configuration servers. This model is favored by administrators who want to delegate addresses based on a client/server model.

Note

This version of TCP/IP Services for OpenVMS does not support DHCPv6.

In both cases, the resulting addresses have associated lifetimes, and systems must be able to acquire new addresses and release expired addresses. Combined with the ability to register updated address information with Domain Name System (DNS) servers, these mechanisms provide a path towards network renumbering and provide network administrators with control over the use of network addresses without manual intervention on each host on the network.

1.2.6. Address Resolution

The Domain Name System (DNS) provides support for mapping names to IP addresses and mapping IP addresses back to their corresponding names. Because of the increased size of the IPv6 address, the DNS has the following new features:

- AAAA resource record type

  This holds IPv6 addresses, encoded in network byte order. The version of BIND shipped with TCP/IP Services for OpenVMS supports AAAA records.

- AAAA query

  A query for a specified domain name in the Internet class returns all associated AAAA resource records in the response.

- IP6.ARPA domain for looking up a name for a specified address (address-to-name mapping)

  An IPv6 address is represented in reverse order as a sequence of 4-bit nibbles separated by dots with the suffix .IP6.ARPA appended. For example, the IPv6 address 4321:0:1:2:3:4:567:89ab has the following reverse lookup domain name:

```
b.a.9.8.7.6.5.0.4.0.0.0.3.0.0.0.2.0.0.0.1.0.0.0.0.0.0.0.1.2.3.4.IP6.ARPA
```

See Chapter 3 for guidelines on configuring BIND in an IPv6 environment.

1.3. Address Assignment

IPv6 addresses are now being deployed by the regional registries. See the IANA web page at the following location for more information:

`http://www.iana.org`

In addition, you can contact your Internet Service Provider (ISP) to obtain an IPv6 address.

Because of the need to test various implementations of the IPv6 RFCs, the IETF has defined a temporary IPv6 address allocation scheme. You can assign the addresses in this scheme to hosts and routers for testing IPv6 on the 6bone (a prototype IPv6 implementation that can be used for testing). See the 6bone home page at the following location for more information about 6bone address allocation and assignment:

`http://www.6bone.net`

The planning stage for a multiyear phaseout of the 6bone has begun.

At the present time, the 6bone test addresses are aggregatable global unicast addresses. Contact your 6bone service provider (for example, gw-6bone@pa.dec.com) for a 6bone address delegation.

1.4. Deploying IPv6 Using Tunnels

Because the Internet and most likely your network are based on IPv4, you need to know how to use this routing infrastructure to carry your IPv6 traffic while you gradually build up your IPv6 routing infrastructure. The best mechanism to employ for routing IPv6 traffic across IPv4 routing infrastructures is tunneling. The following types of tunnels are supported:

- Automatic
- 6to4
- Configured

The following sections describe each tunnel and its advantages and disadvantages. The more powerful the tunnel, the more configuration and administration it requires.

1.4.1. Automatic Tunnels

An IPv6 automatic tunnel is the simplest tunnel to configure and deploy. This mechanism enables hosts with a globally unique IPv4 address to automatically create a tunnel over an IPv4 network. The tunnel is created as a virtual interface (TN0) and is configured with an IPv4-compatible IPv6 address, which is derived from the IPv4 address. The destination address of the packet determines the tunnel destination endpoint. See Section 1.2.2.1 for more information about IPv4-compatible IPv6 addresses.

This mechanism is good for introducing hosts to IPv6 because it permits application porting, testing, and experimentation with the IPv6 protocol. However, an automatic tunnel has the following limitations:

- Requires a globally unique (not private) IPv4 address.

- Benefits hosts more than routers. You can neither run the RIPng protocol over the automatic tunnel nor can you forward packets over the tunnel.
- Communicates only with other nodes that are configured with IPv4-compatible IPv6 addresses. You cannot communicate with nodes that are configured with native IPv6 addresses only.
- Is quite possibly going to be deprecated by the IPv6 community. Therefore, do not deploy this in your production environment.

1.4.2. 6to4 Tunnels

A 6to4 tunnel is a type of automatic tunnel, but it offers greater connectivity. This mechanism enables a special IPv6 site, called a 6to4 site, with a single, globally unique IPv4 address to automatically create a tunnel over an IPv4 network to communicate with other 6to4 sites. The tunnel is created as a virtual interface (TNn) on a node at the IPv4 network attachment point. This node is either an individual host or a router called a border router. The tunnel is configured with a special 6to4 address that is derived from the IPv4 address. The destination address of the packet determines the tunnel destination endpoint.

Within the 6to4 site, the border router creates the 6to4 site prefix from its globally unique IPv4 address and advertises the prefix to all nodes in the 6to4 site. Each node automatically configures its 6to4 address based on the 6to4 prefix; no special configuration is necessary. Nodes within the 6to4 site communicate with each other using native IPv6. Any traffic that is addressed outside the site is forwarded to the border router.

This mechanism is easy to configure and can be deployed in a production environment. However, a 6to4 tunnel has the following limitations:

- Communicates only with other nodes that are configured with 6to4 addresses. However, if you use third-party 6to4 Relay Router services or 6to4 relay services on the Internet, you can communicate with nodes that are configured with native IPv6 addresses only.
- Relies on the underlying IPv4 network routing infrastructure. Therefore, routing might not be as efficient as native IPv6 connectivity or configured tunnels.

1.4.3. Configured Tunnels

A configured tunnel is the most complex tunnel to configure and deploy. There are two types of configured tunnels:

- IPv4 configured tunnel – encapsulates IPv4 or IPv6 packets in an IPv4 packet and carries those packets through an IPv4 network infrastructure. An IPv6 over IPv4 configured tunnel enables IPv6 sites and hosts to communicate with other IPv6 nodes across an IPv4 network.
- IPv6 configured tunnel – encapsulates IPv4 or IPv6 packets in an IPv6 packet and carries those packets through an IPv6 network infrastructure. An IPv6 over IPv6 configured tunnel is an enabling technology for mobile IPv6, and can also be used for traffic engineering (for example, IPv6 multihoming support).

A configured tunnel is created as a virtual interface (ITn) and uses IPv4 addresses (IPv4 configured tunnel) or IPv6 addresses (IPv6 configured tunnel) as the source and destination endpoints. If you want to send IPv6 traffic through any configured tunnel, you configure an IPv6 address on the tunnel interface. If you want to send IPv4 traffic through any configured tunnel, you configure an IPv4 address on the tunnel interface.

This mechanism is the most powerful tunneling mechanism, but has the following limitations:

- Requires a coordinated configuration of each tunnel endpoint.
- Relies on the expertise of the administrator to obtain efficient routing of traffic. If the endpoint is misconfigured, you might have inefficient routes, routing loops, or both.

1.5. IPv6 Environment

This section shows some example IPv6 configurations. Select a configuration that most closely matches the environment in which you want to configure IPv6 on your system.

Figure 1.11 shows a simple LAN configuration in which host A and host B communicate using IPv6 with no router.

Figure 1.11. Host-to-Host Configuration with No Router

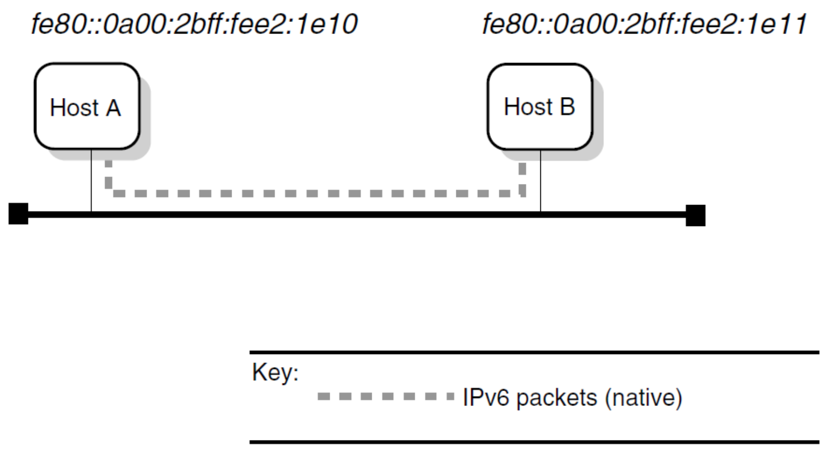


Figure 1.12 shows a simple LAN configuration in which host A, host B, and router A communicate using IPv6. Host A and host B obtain global addresses from router A.

Figure 1.12. Host-to-Host Configuration with Router

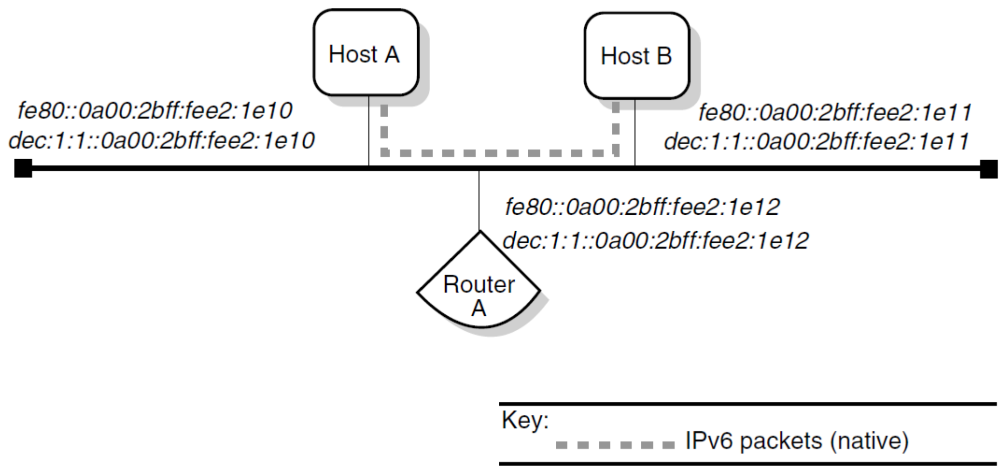

Host A
Host B
fe80::0a00:2bff:fee2:1e10
dec:1:1::0a00:2bff:fee2:1e10
fe80::0a00:2bff:fee2:1e11
dec:1:1::0a00:2bff:fee2:1e11
fe80::0a00:2bff:fee2:1e12
dec:1:1::0a00:2bff:fee2:1e12
Router A
Key:
IPv6 packets (native)

Figure 1.13 shows a configuration in which two IPv6 networks are connected through an IPv6 router (router A).

Figure 1.13. IPv6 Network to IPv6 Network with Router Configuration

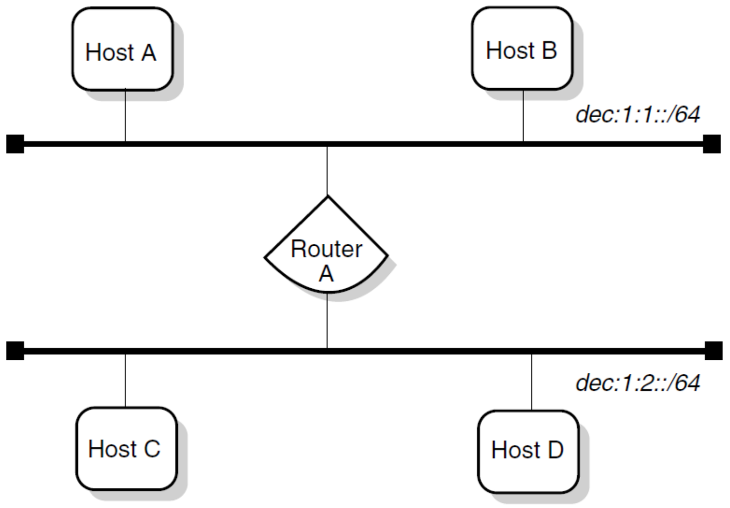


Figure 1.14 shows a configuration in which four IPv6 networks are connected using three routers. The three routers exchange routing information with each other using the RIPng protocol.

Figure 1.14. Multiple IPv6 Networks and Multiple Routers Configuration

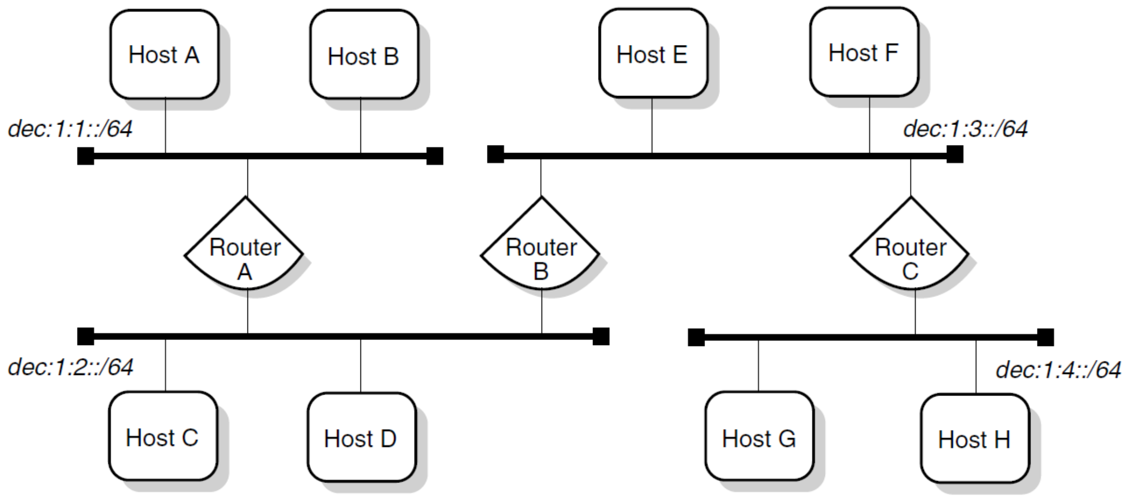


Figure 1.15 shows a configuration in which host A and host B, connected to an IPv4 network, communicate using IPv6 through an IPv4 tunnel.

Figure 1.15. Host-to-Host Configuration over Tunnel

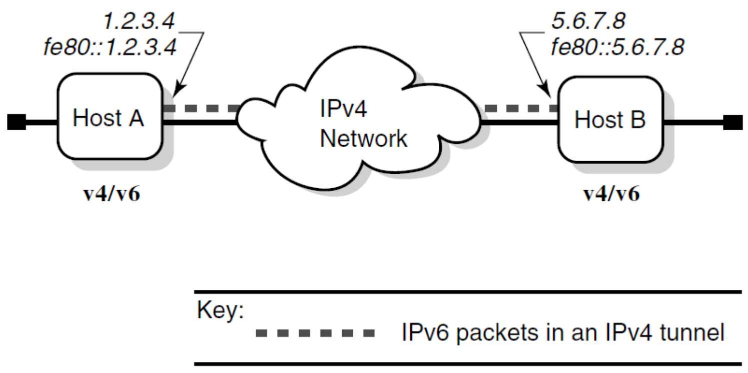


Figure 1.16 shows a configuration in which host X is connected to an IPv4 network. Router A, an IPv6 router, is connected to the same IPv4 network and is also connected to two IPv6 networks. Host X communicates with host B using IPv6 through an IPv4 tunnel between host X and router A.

Figure 1.16. Host-to-Router Configuration over Tunnel

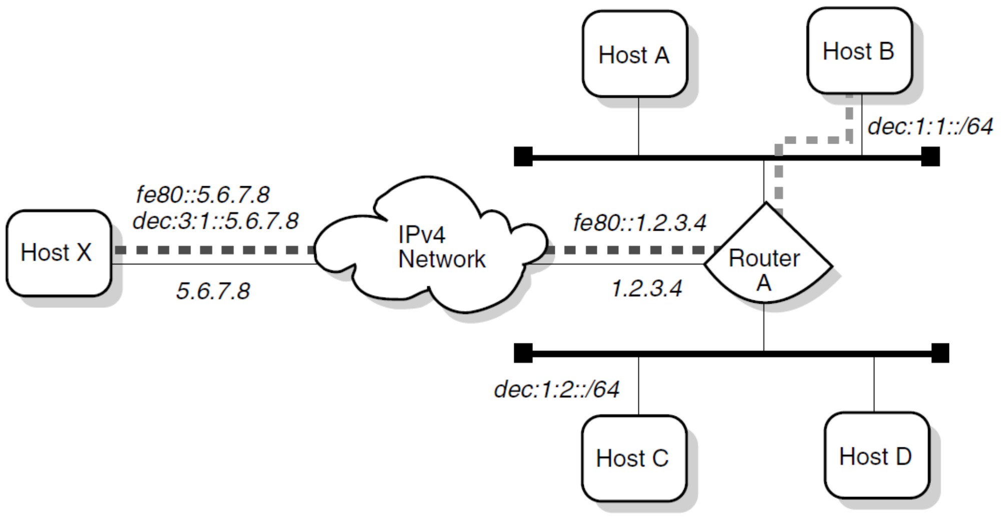


Figure 1.17 shows a configuration in which four IPv6 networks are connected through two routers and an IPv4 network. Host A communicates with host F through an IPv4 tunnel between router A and router B.

Figure 1.17. IPv6 Network-to-IPv6 Network Configuration over Tunnel

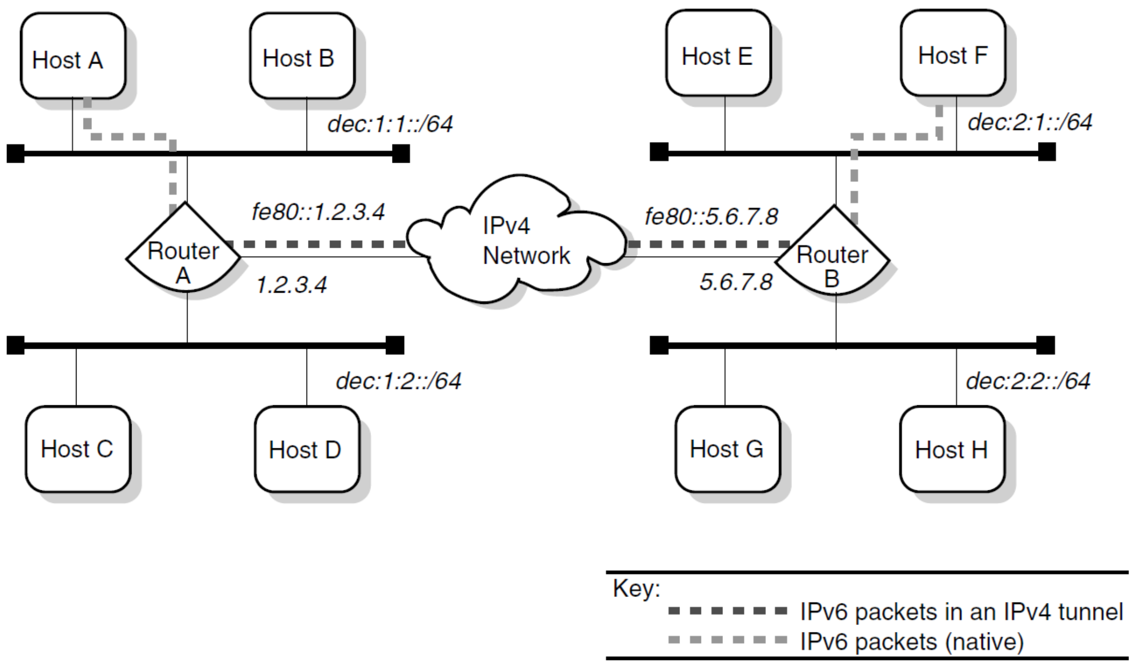
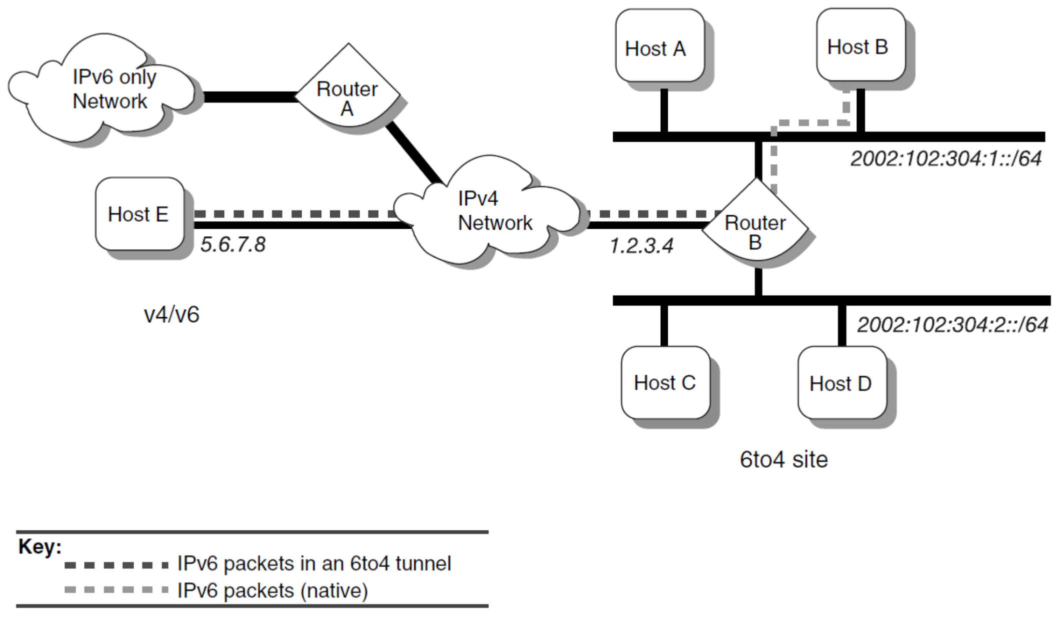


Figure 1.18 shows a configuration in which host E is connected to an IPv4 network. Router B, an IPv6 router, is connected to the same IPv4 network and also is connected to two IPv6 networks. Host E communicates with host B using a 6to4 tunnel between host E and router B.

Figure 1.18. IPv6 Network-to-IPv6 Network Configuration over Tunnel



Chapter 2. Configuring IPv6

After installing TCP/IP Services, you can configure your system to communicate in an IPv6 network environment by performing the tasks described in this chapter.

You can configure your node as either an IPv6 host or IPv6 router. You make this choice while running the TCPIP$IP6_SETUP configuration utility. After you run the configuration utility and restart TCP/IP Services, IPv6 processes associated with your choices are started on your system.

2.1. IPv6 Processes

Your selection of configuring your node as either a host or a router starts one of the following processes during TCP/IP startup:

- TCPIP$ND6HOST (host process)
- TCPIP$IP6RTRD (router process)

Caution

Do not run the TCPIP$ND6HOST and TCPIP$IP6RTRD processes on the same node, since doing so might produce unpredictable results.

2.1.1. TCPIP$ND6HOST

The TCPIP$ND6HOST process receives and processes IPv6 router advertisement (RA) packets of the neighbor discovery protocol. This enables a system to autoconfigure itself without manual intervention.

The TCPIP$ND6HOST process performs the following functions, based on the contents of IPv6 router advertisements it receives:

- Router discovery – Learns the IPv6 address of default routers and installs default routes in the kernel routing table.
- On-link prefix discovery – Learns IPv6 on-link prefixes (ranges of IPv6 addresses that are directly reachable on a given link).
- Stateless address configuration – Automatically creates and deletes interface addresses.
- Interface attribute configuration – Automatically configures datalink attributes, such as hop limit, reachable time, retransmit time, and link MTU.

2.1.2. TCPIP$IP6RTRD Process

After you configure the system as an IPv6 router, the TCPIP$IP6RTRD process sends out periodic router advertisements for the following reasons:

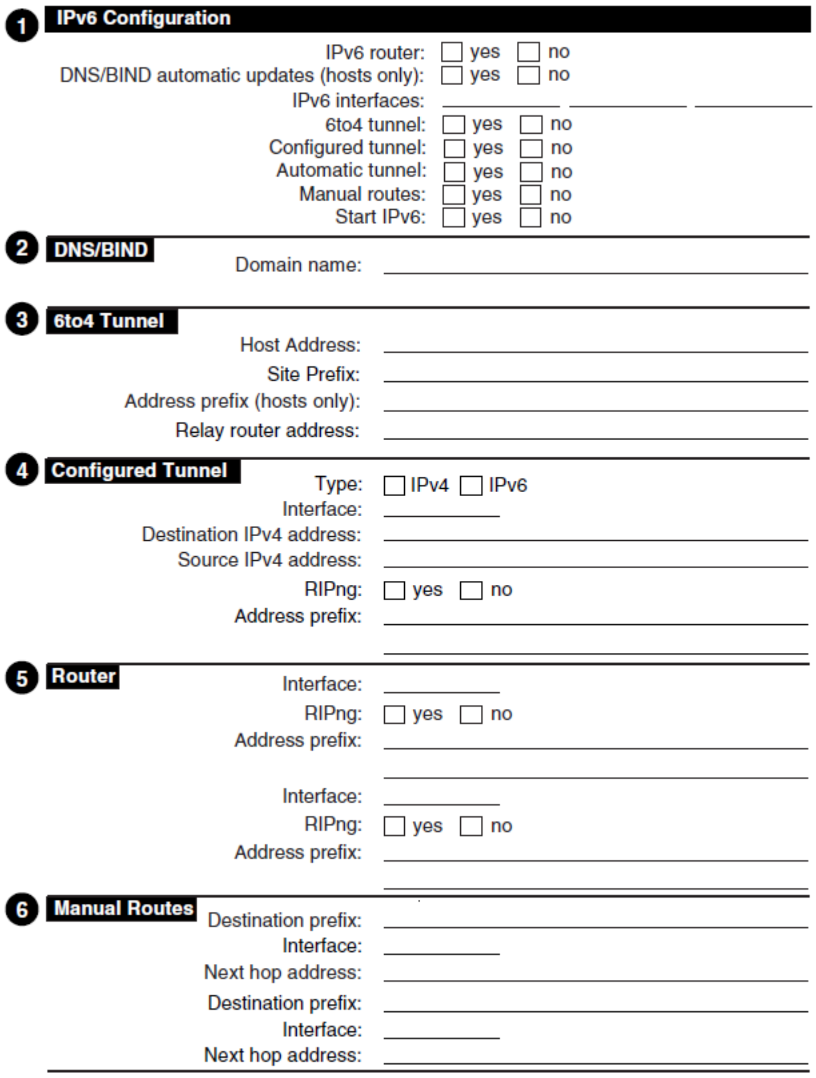
- To advertise itself as a potential default router for IPv6 traffic. The IPv6 hosts on the link receive these advertisements as part of their neighbor discovery processing.
- To advertise an IPv6 address prefix, in which case hosts on the link perform address autoconfiguration.

At startup, the TCPIP$IP6RTRD process reads its configuration file for startup information. See Section 2.6.2 for more information on the router configuration file.

2.2. Preparing for Configuration

Before you configure the network software, you must gather information about your system and network environment. The Configuration Worksheet shown in Figure 2.1 can help you assemble this information in an orderly fashion. The following sections describe the information that you need to record on the worksheet.

Figure 2.1. Configuration Worksheet



1 **IPv6 Configuration**

IPv6 router:	☐ yes ☐ no
DNS/BIND automatic updates (hosts only):	☐ yes ☐ no
IPv6 interfaces:	________ ________ ________
6to4 tunnel:	☐ yes ☐ no
Configured tunnel:	☐ yes ☐ no
Automatic tunnel:	☐ yes ☐ no
Manual routes:	☐ yes ☐ no
Start IPv6:	☐ yes ☐ no

2 **DNS/BIND**

Domain name:	________

3 **6to4 Tunnel**

Host Address:	________
Site Prefix:	________
Address prefix (hosts only):	________
Relay router address:	________

4 **Configured Tunnel**

Type:	☐ IPv4 ☐ IPv6
Interface:	________
Destination IPv4 address:	________
Source IPv4 address:	________
RIPng:	☐ yes ☐ no
Address prefix:	________
	________

5 **Router**

Interface:	________
RIPng:	☐ yes ☐ no
Address prefix:	________
	________
Interface:	________
RIPng:	☐ yes ☐ no
Address prefix:	________
	________

6 **Manual Routes**

Destination prefix:	________
Interface:	________
Next hop address:	________
Destination prefix:	________
Interface:	________
Next hop address:	________

1. IPv6 Configuration

   - IPv6 router

     If you want this system to function as an IPv6 router, check Yes; otherwise, check No. If you check No, the system is configured as an IPv6 host.

     An IPv6 router can advertise address prefixes to all hosts on connected links (for example, a LAN and a configured tunnel) and can forward packets to their destinations. Packets can be forwarded directly on link or over IPv4 tunnels.

- DNS/BIND automatic updates (hosts only)

  If you want this system to record its addresses in the DNS/BIND database automatically, check Yes; otherwise, check No. If you check Yes, you must configure your system as a DNS/BIND client and your DNS/BIND server must support dynamic updates to the DNS database. See Chapter 3 for information on configuring your DNS/BIND server.

- IPv6 interfaces

  Enter the device names of the network interface to the IPv6 network. For example, WE0 and WF0. If you are creating a configured tunnel only on your system, enter None.

- 6to4 tunnel

  If you want IPv6 to run over a 6to4 tunnel, check Yes; otherwise, check No. A 6to4 tunnel has one source and one destination in an IPv4 network.

- Configured tunnel

  If you want IPv6 to run over a configured IPv4 tunnel, check Yes; otherwise, check No. A configured tunnel has one source and one destination in an IPv4 network. You should use configured tunnels instead of automatic tunnels. You can configure multiple configured tunnels.

- Automatic tunnel

  If you want to configure IPv6 to run over IPv4 automatic tunnels, check Yes; otherwise, check No.

Note

Do not use automatic tunnels in production environments. Their use might be deprecated in the future.

- Manual routes

  If you want to configure manual routes to other systems, check Yes; otherwise, check No.

  On a router, you might want to configure manual routes if one of the following conditions is true:

  - You want a configured tunnel and you are not advertising an address prefix on the tunnel link.
  - You want a configured tunnel and the router at the other end of the tunnel is not running the RIPng protocol.
  - Your system is not running the RIPng protocol.

  On a host, you might want to configure manual routes if you want a configured tunnel to a router and the router is not advertising itself as a default router on the tunnel link.

- Start IPv6

If you want the IPv6 initialization script executed from the configuration utility, check Yes. If you want the initialization script executed during the next system boot, check No.

2. DNS/BIND

   - Domain name

     The fully qualified domain name for your node. This consists of the host name and the DNS/BIND domain name (for example, host1.subdomain.example).

3. 6to4 Tunnel

   - Host address

     Your node's name or IP address (this end of the tunnel).

   - Site prefix

     The TCPIP$IP6_SETUP command procedure automatically generates a 48-bit 6to4 site prefix.

   - Address prefix (hosts only)

     If your system is an IPv6 host, enter a 64-bit 6to4 prefix to be configured on the 6to4 tunnel interface. The upper 48 bits of the address prefix must be identical to the site prefix generated by the TCPIP$IP6_SETUP command procedure.

   - Relay router address

     If you want to communicate with an IPv6-only network, enter the 6to4 address of the relay router.

4. Configured Tunnel

   - Type

     The type of configured tunnel. Valid types are IPv4 and IPv6.

   - Interface

     The name of the configured tunnel interface. For example, IT0.

   - Destination address

     The remote node's address (the remote end of the tunnel).

   - Source address

     Your node's address (this end of the tunnel).

   - RIPng

     If your system is a router and you want the router to run the RIPng protocol on the tunnel link to exchange IPv6 routing information with a router at the remote end of the tunnel, check Yes; otherwise, check No.

   - Address prefix

If your system is a router and you want to advertise address prefixes to the node at the remote end of the tunnel, enter a 64-bit prefix; otherwise, write Done. If your system is an IPv6 host and the router at the remote end of the tunnel is not advertising an address prefix, enter a 64-bit prefix to be configured on the tunnel interface.

5. Router

   - Interface

     The name of the interface (LAN or configured tunnel) on which you want to run the RIPng protocol or advertise an address prefix.

   - RIPng

     If you want the router to run the RIPng protocol on the specified interface and to exchange IPv6 routing information with other routers on the link (LAN or configured tunnel), check Yes; otherwise, check No.

   - Address prefix

     If you want to advertise address prefixes to all hosts on the link, enter a 64-bit prefix; otherwise, write Done. If you do not specify a 64-bit prefix, the router will not advertise an address prefix. All hosts must obtain their prefix information from another source.

     Prefixes in IPv6 define a subnet and are typically configured on a router for a specific link by the network administrator. The router advertises this prefix to all nodes connected to that link, along with the length of the prefix, whether the prefix is on link (that is, a neighbor), whether the prefix can also be used for stateless address configuration, and the length of time the prefix is valid.

6. Manual Routes

   - Destination prefix

     The address prefix of a remote IPv6 network. The address prefix contains a Classless Inter-Domain Routing (CIDR) style bit length, for example, 5F00::/8. If you want to use the default route, write Default.

   - Interface

     The name of the interface through which you are sending traffic to the remote IPv6 network.

   - Next hop address

     The IPv6 address of the first router in the path to the destination prefix. Write the link local address of the router. If the connection to the router is over an IPv4 tunnel, write the link local IPv6 address of the remote tunnel endpoint.

When you run the TCPIP$IP6_SETUP command procedure, it gathers information from the system and prompts you for additional configuration information. See Section 2.4 for more information on running the TCPIP$IP6_SETUP command procedure.

2.3. IPv6 System Configuration Examples

This section shows how to use the configuration worksheet to assemble information for selected configurations. Each example shows how individual systems are configured. In some cases, additional options for you to consider are provided.

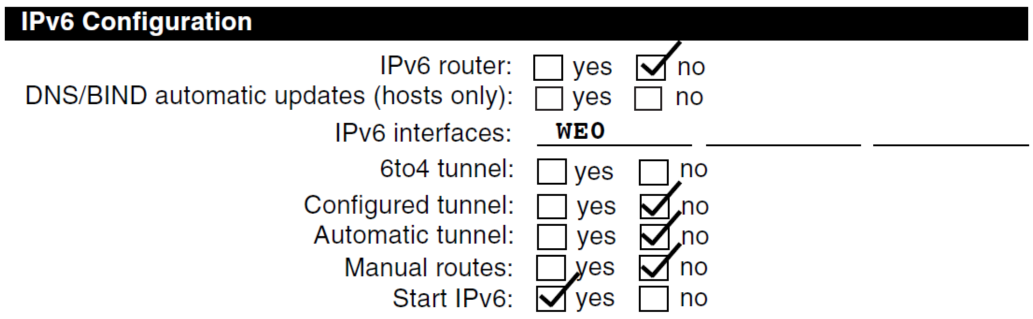
Note

OpenVMS interface names must be in uppercase.

2.3.1. Simple Host-to-Host Configuration

In a simple host-to-host configuration (shown in Figure 1.11), host A and host B use IPv6 link-local addresses. By default, the TCPIP$IP6_SETUP command procedure configures the hosts automatically with a link-local address for your system. Figure 2.2 shows the completed worksheet for host A.

Figure 2.2. Simple Host-to-Host Configuration



IPv6 Configuration

IPv6 router:	yes [✓ no]
DNS/BIND automatic updates (hosts only):	yes no
IPv6 interfaces:	WE0
6to4 tunnel:	yes no
Configured tunnel:	yes [✓ no]
Automatic tunnel:	yes [✓ no]
Manual routes:	yes [✓ no]
Start IPv6:	[✓ yes] no

After you configure IPv6 on host A, add a link-local address for host B to the TCPIP $ETC:IPNODES.DAT file. The configuration process for host B in this configuration is similar to that for host A.

In this configuration, no global address prefix is advertised on the LAN. If you want to advertise a global address prefix, you can either configure one of the hosts as a router by using TCPIP $IP6_SETUP or add an IPv6 router to the LAN configuration. An IPv6 router advertises a global prefix on the link.

You can use the netstat -in command to view a local node's link-local and global addresses.

The following TELNET command connects host A to host B using host B's link-local address:

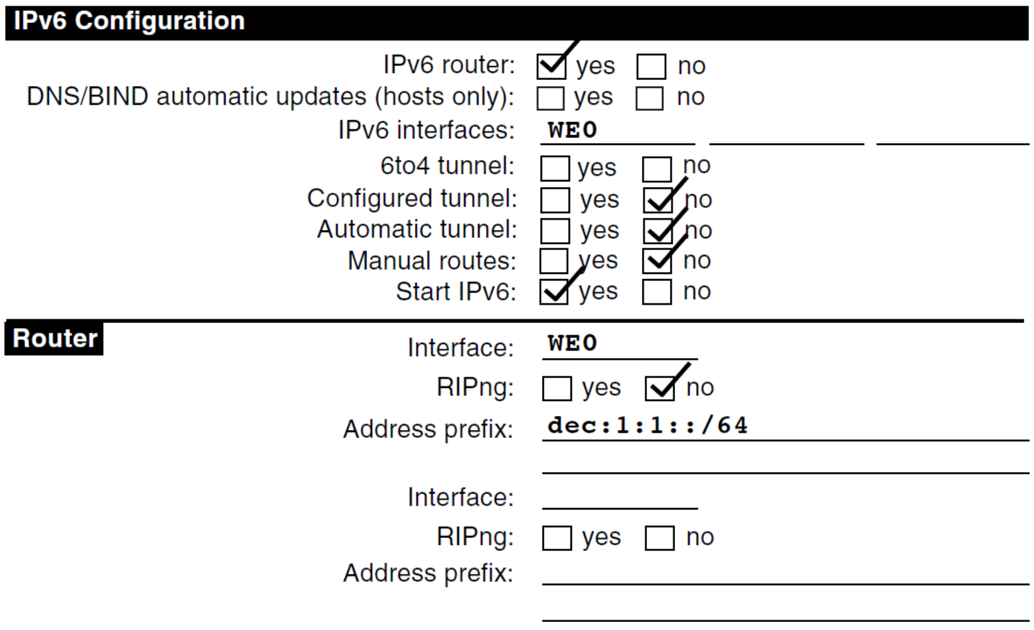
```
$ TELNET fe80::0a00:2bff:fee2:1e11
```

Alternately, you can place the address and node name in the TCPIP$ETC:IPNODES.DAT file. Then use the node name as the argument to the TELNET command.

2.3.2. Host-to-Host with Router Configuration

In a host-to-host with router configuration (shown in Figure 1.12), host A and host B are on a LAN with router A. In this case, router A advertises the global address prefix `dec:1:1::/64` on the LAN. Host A and host B use this address prefix to create global IPv6 addresses. (See Chapter 1 for information about obtaining experimental testing addresses.) Figure 2.3 shows the completed worksheet for router A.

Figure 2.3. Host-to-Host with Router Configuration



IPv6 Configuration

IPv6 router:	[✓ yes] no
DNS/BIND automatic updates (hosts only):	yes no
IPv6 interfaces:	WE0
6to4 tunnel:	yes no
Configured tunnel:	yes [✓ no]
Automatic tunnel:	yes [✓ no]
Manual routes:	yes [✓ no]
Start IPv6:	[✓ yes] no

Router

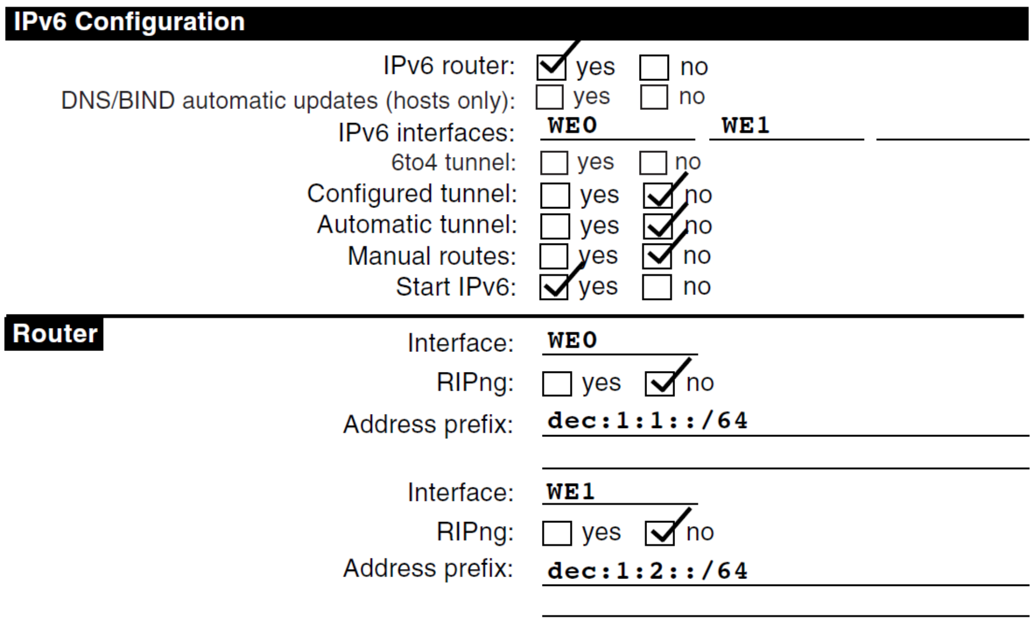
Interface:	WE0
RIPng:	yes [✓ no]
Address prefix:	dec:1:1::/64
Interface:	
RIPng:	yes no
Address prefix:	

After you configure IPv6 on router A, add the global addresses for the other hosts to the TCPIP $ETC:IPNODES.DAT file. Repeat this step on host A and host B. Alternatively, you could establish DNS/BIND in your network using the global addresses.

2.3.3. IPv6 Network-to-IPv6 Network with Router Configuration

In an IPv6 network-to-IPv6 network with router configuration (shown in Figure 1.13), two IPv6 networks are connected to each other through router A and its two interfaces. Figure 2.4 shows the completed worksheet for router A.

Figure 2.4. IPv6 Network-to-IPv6 Network with Router Configuration



IPv6 Configuration

IPv6 router:	[✓ yes] no
DNS/BIND automatic updates (hosts only):	yes no
IPv6 interfaces:	WE0 WE1
6to4 tunnel:	yes no
Configured tunnel:	yes [✓ no]
Automatic tunnel:	yes [✓ no]
Manual routes:	yes [✓ no]
Start IPv6:	[✓ yes] no

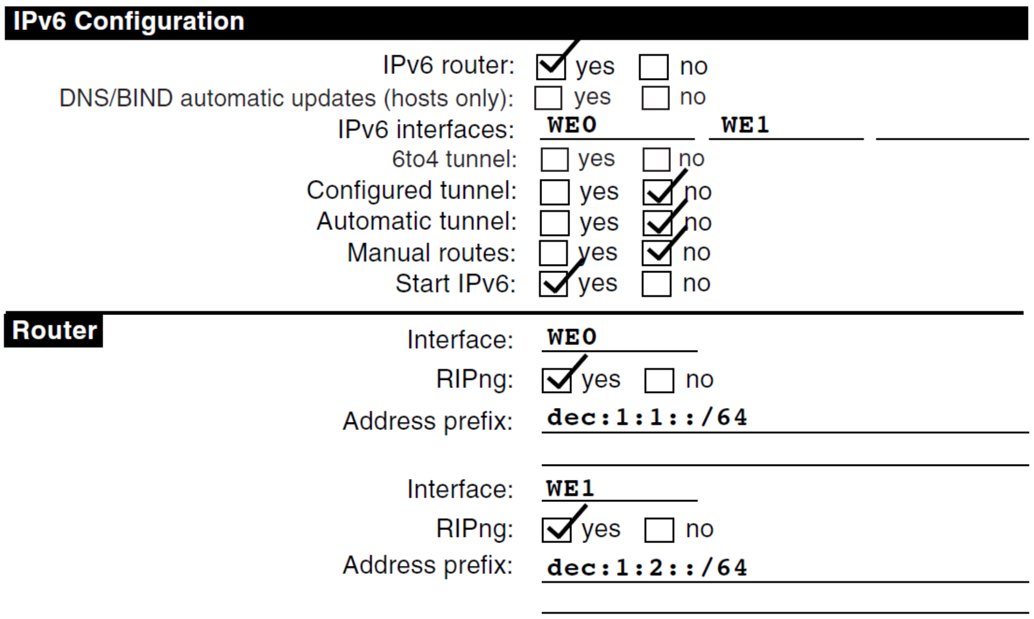
Router

Interface:	WE0
RIPng:	yes [✓ no]
Address prefix:	dec:1:1::/64
Interface:	WE1
RIPng:	yes [✓ no]
Address prefix:	dec:1:2::/64

2.3.4. Multiple IPv6 Networks and Multiple Routers Configuration

In this example configuration (shown in Figure 1.14), four IPv6 networks are connected to each other using three routers. In this configuration, the routers must exchange routing information in order to learn the routes to other subnets in the network. To accomplish this, each router must run the RIPng protocol. Figure 2.5 shows the completed worksheet for router A.

Figure 2.5. Multiple IPv6 Networks and Multiple Routers Configuration



IPv6 Configuration

IPv6 router:	[✓ yes] no
DNS/BIND automatic updates (hosts only):	yes no
IPv6 interfaces:	WE0 WE1
6to4 tunnel:	yes no
Configured tunnel:	yes [✓ no]
Automatic tunnel:	yes [✓ no]
Manual routes:	yes [✓ no]
Start IPv6:	[✓ yes] no

Router

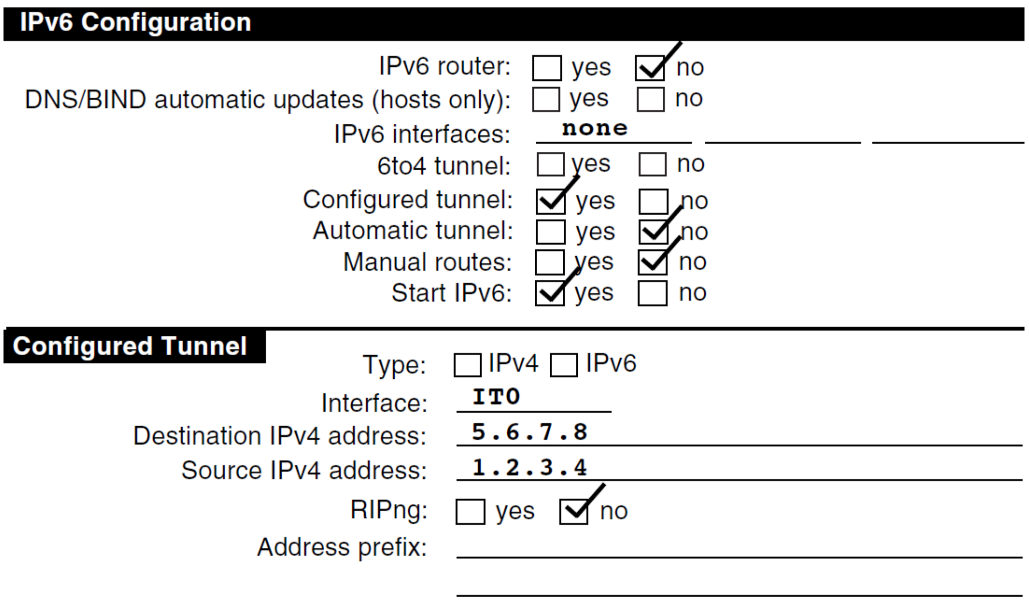
Interface:	WE0
RIPng:	[✓ yes] no
Address prefix:	dec:1:1::/64
Interface:	WE1
RIPng:	[✓ yes] no
Address prefix:	dec:1:2::/64

The completed worksheets for router B and C would be similar.

2.3.5. Host-to-Host over IPv4 Configured Tunnel Configuration

In a host-to-host over tunnel configuration (shown in Figure 1.15), two IPv6 systems communicate with each other over a configured tunnel through an IPv4 network and use IPv6 link-local addresses. Figure 2.6 shows the completed worksheet for host A.

Figure 2.6. Host-to-Host over IPv4 Configured Tunnel Configuration



IPv6 Configuration

IPv6 router:	yes [✓ no]
DNS/BIND automatic updates (hosts only):	yes no
IPv6 interfaces:	none
6to4 tunnel:	yes no
Configured tunnel:	[✓ yes] no
Automatic tunnel:	yes [✓ no]
Manual routes:	yes [✓ no]
Start IPv6:	[✓ yes] no

Configured Tunnel

Type:	IPv4 IPv6
Interface:	ITO
Destination IPv4 address:	5.6.7.8
Source IPv4 address:	1.2.3.4
RIPng:	yes [✓ no]
Address prefix:	

After you configure IPv6 on host A, add the link-local address for host B to the TCPIP $ETC:IPNODES.DAT file. The configuration process for host B in this configuration is similar to that for host A.

With this configuration, no global address prefix is advertised on the tunnel. If you want to advertise a global address prefix, you can configure one of the hosts as a router by using TCPIP$IP6_SETUP. An IPv6 router advertises a global prefix on the link.

To view a local node's link-local and global addresses, use the netstat -in command.

The following TELNET command connects host A to host B:

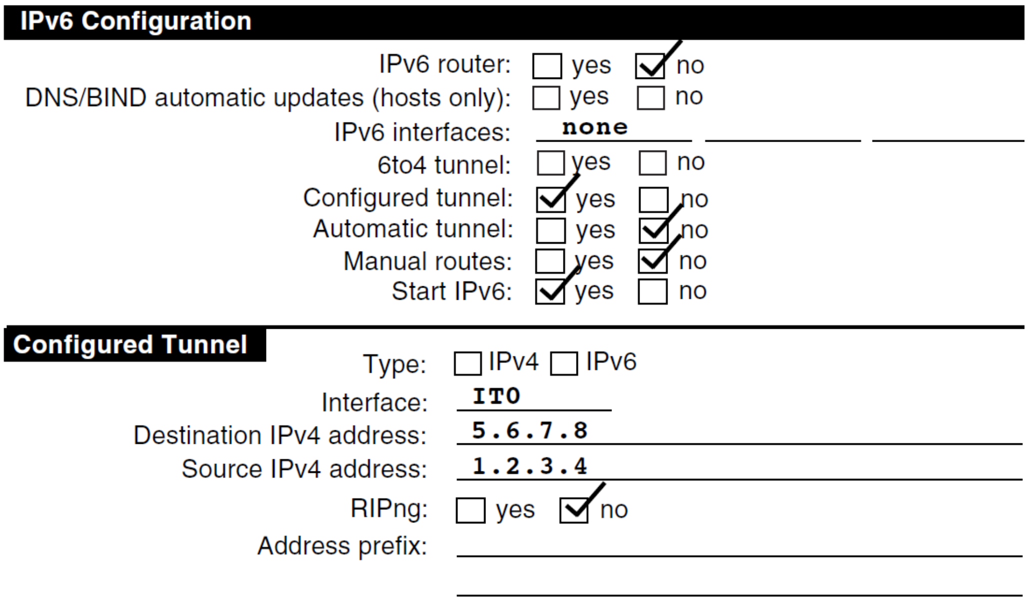
```
$ telnet fe80::5.6.7.8
```

Alternately, you can place the address and node name in the TCPIP$ETC:IPNODES.DAT file. Then use the Node name as the argument to the TELNET command.

2.3.6. Host-to-Router over IPv4 Configured Tunnel Configuration

In a host-to-router over tunnel configuration (shown in Figure 1.16), host X communicates with host B over a configured tunnel through an IPv4 network; both nodes use IPv6 addresses. The tunnel in this case is between host X and router A. Figure 2.7 shows the completed worksheet for host X when router A is advertising itself as the default router for the tunnel link and is advertising a global address prefix on the tunnel link.

Figure 2.7. Host-to-Router over IPv4 Configured Tunnel Configuration



IPv6 Configuration

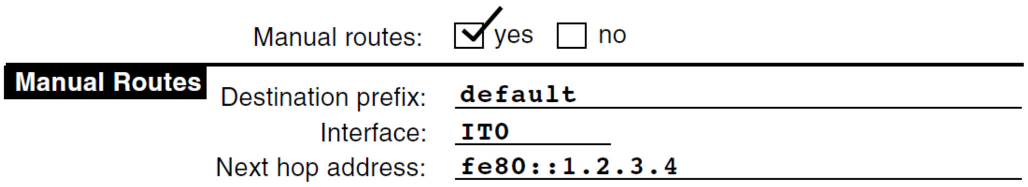
IPv6 router:	☐ yes [✓ no]
DNS/BIND automatic updates (hosts only):	☐ yes ☐ no
IPv6 interfaces:	none
6to4 tunnel:	☐ yes ☐ no
Configured tunnel:	[✓ yes] ☐ no
Automatic tunnel:	☐ yes [✓ no]
Manual routes:	☐ yes [✓ no]
Start IPv6:	[✓ yes] ☐ no

Configured Tunnel

Type:	☐ IPv4 ☐ IPv6
Interface:	ITO
Destination IPv4 address:	5.6.7.8
Source IPv4 address:	1.2.3.4
RIPng:	☐ yes [✓ no]
Address prefix:	

If router A is not advertising a global address prefix on the tunnel link, the value `dec:3:1::/64` would be in the Address prefix field in the Configured Tunnel section of the host X worksheet. If router A is not advertising itself as the default router for the tunnel link, the information shown in Figure 2.8 would also be on the host X worksheet:

Figure 2.8. Router Not Advertising a Global Address Prefix



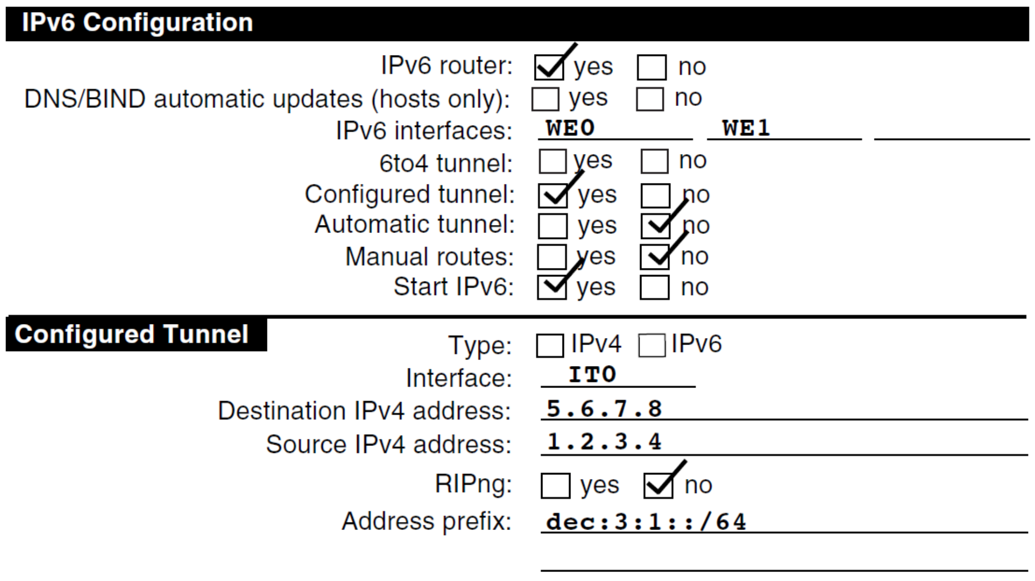
Manual routes:	[✓ yes] ☐ no

Manual Routes

Destination prefix:	default
Interface:	ITO
Next hop address:	fe80::1.2.3.4

Figure 2.9 shows the completed worksheet for router A when router A is advertising a global address prefix on the tunnel link.

Figure 2.9. Router Advertising a Global Address Prefix



IPv6 Configuration

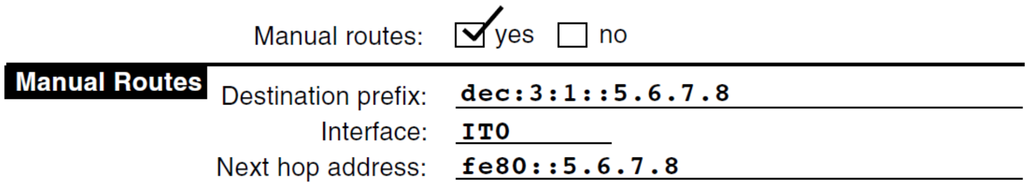
IPv6 router:	[✓ yes] ☐ no
DNS/BIND automatic updates (hosts only):	☐ yes ☐ no
IPv6 interfaces:	WE0 WE1
6to4 tunnel:	☐ yes ☐ no
Configured tunnel:	[✓ yes] ☐ no
Automatic tunnel:	☐ yes [✓ no]
Manual routes:	☐ yes [✓ no]
Start IPv6:	[✓ yes] ☐ no

Configured Tunnel

Type:	☐ IPv4 ☐ IPv6
Interface:	ITO
Destination IPv4 address:	5.6.7.8
Source IPv4 address:	1.2.3.4
RIPng:	☐ yes [✓ no]
Address prefix:	dec:3:1::/64

If router A is not advertising a global prefix on the tunnel link, the information shown in Figure 2.10 would be on the router A worksheet. Note the manual route to host X. Instead of specifying a destination network prefix, you specify the host route, `dec:3:1::5.6.7.8`, to host X. The next hop is the link-local IPv6 address of host X's tunnel interface, `fe80::5.6.7.8`.

Figure 2.10. Router A Not Advertising a Global Prefix on the Tunnel Link

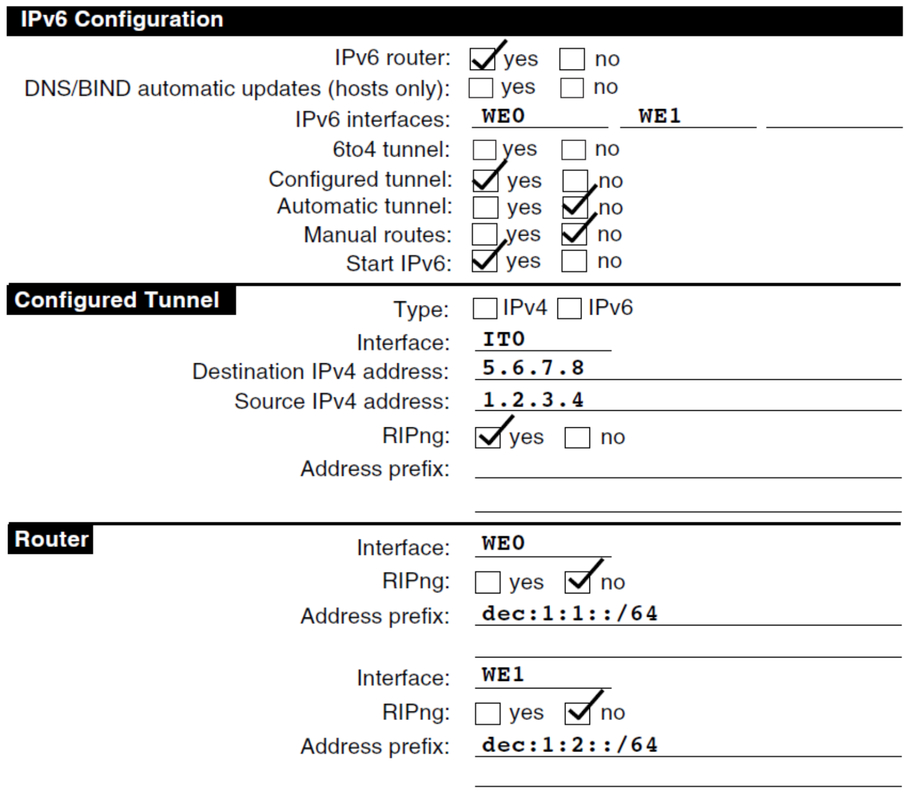


Manual routes:	[✓ yes] no
Manual Routes	
Destination prefix:	dec:3:1::5.6.7.8
Interface:	IT0
Next hop address:	fe80::5.6.7.8

2.3.7. IPv6 Network to IPv6 Network over IPv4 Configured Tunnel Configuration

In an IPv6 to IPv6 network over tunnel configuration (shown in Figure 1.17), host A communicates with host F over a configured tunnel through an IPv4 network. The host configuration is similar to that of host A (Section 2.3.1). All hosts automatically use their default router in order to communicate with hosts on other networks. Figure 2.11 shows the worksheet for router A.

Figure 2.11. IPv6 Network to IPv6 Network over IPv4 Configured Tunnel Configuration



IPv6 Configuration	
IPv6 router:	[✓ yes] no
DNS/BIND automatic updates (hosts only):	yes no
IPv6 interfaces:	WE0 WE1
6to4 tunnel:	yes no
Configured tunnel:	[✓ yes] no
Automatic tunnel:	yes [✓ no]
Manual routes:	yes [✓ no]
Start IPv6:	[✓ yes] no
Configured Tunnel	
Type:	IPv4 IPv6
Interface:	IT0
Destination IPv4 address:	5.6.7.8
Source IPv4 address:	1.2.3.4
RIPng:	[✓ yes] no
Address prefix:	
Router	
Interface:	WE0
RIPng:	yes [✓ no]
Address prefix:	dec:1:1::/64
Interface:	WE1
RIPng:	yes [✓ no]
Address prefix:	dec:1:2::/64

You do not have to run RIPng on the WE0 and WE1 interfaces because no routers are attached to the interfaces.

The configuration of router B is similar, except that the source and destination addresses for the configured tunnel would be switched and the address prefixes advertised on WE0 and WE1 would be `dec:2:1::/64` and `dec:2:2::/64`, respectively.

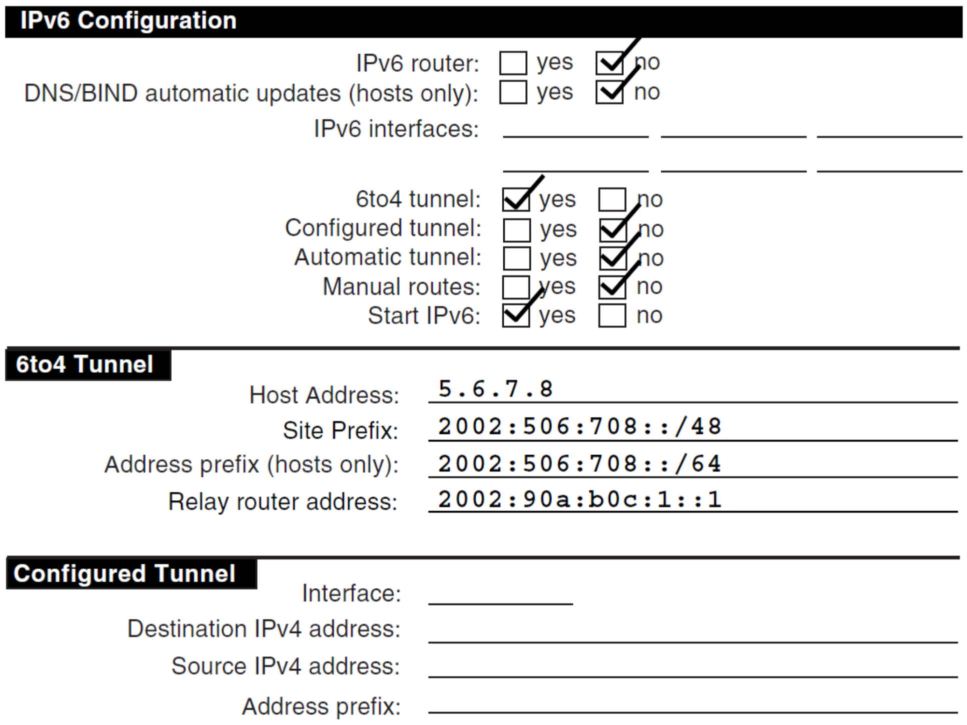
Note

If the routers were not configured to use RIPng over the tunnel interface, each router would need to specify a manual route to the other.

2.3.8. 6to4 Tunnel Configuration

In a 6to4 tunnel configuration (shown in Figure 1.18), host E is the only node in a 6to4 site. It communicates with host B over a 6to4 tunnel through an IPv4 network: both nodes use IPv6 6to4 addresses. The tunnel in this case is betwen host E and router B. IPv6 is not configured on the host E physical interface because it is connected to an IPv4 network. IPv6, however, is configured on the 6to4 tunnel. Figure 2.12 shows the worksheet for host E.

Figure 2.12. 6to4 Tunnel Host E Configuration



IPv6 Configuration

IPv6 router:	yes [✓ no]
DNS/BIND automatic updates (hosts only):	yes [✓ no]
IPv6 interfaces:	
6to4 tunnel:	[✓ yes] no
Configured tunnel:	yes [✓ no]
Automatic tunnel:	yes [✓ no]
Manual routes:	yes [✓ no]
Start IPv6:	[✓ yes] no

6to4 Tunnel

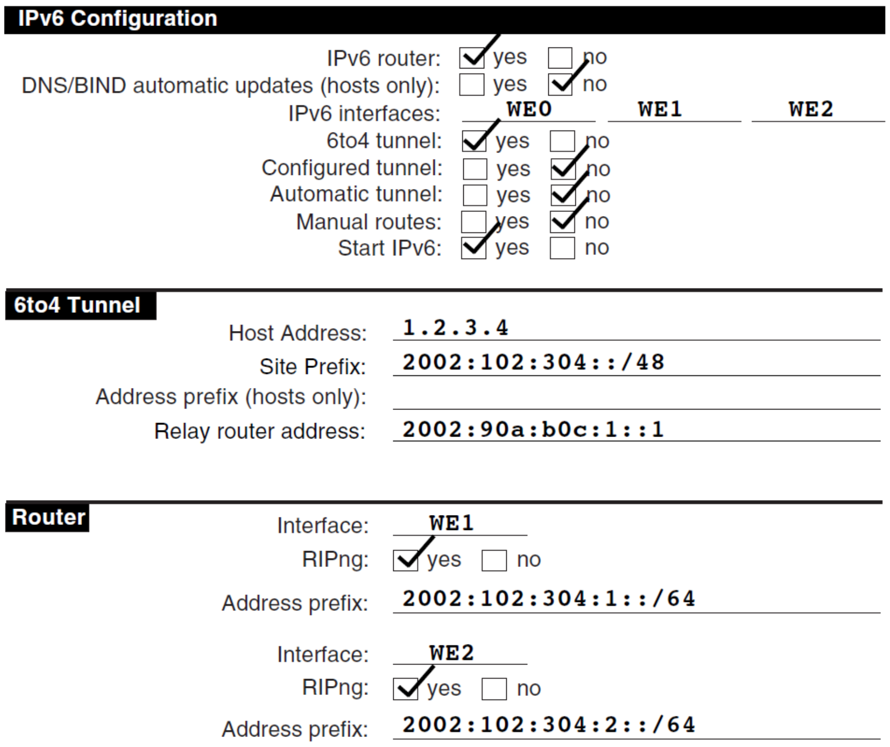
Host Address:	5.6.7.8
Site Prefix:	2002:506:708::/48
Address prefix (hosts only):	2002:506:708::/64
Relay router address:	2002:90a:b0c:1::1

Configured Tunnel

Interface:	
Destination IPv4 address:	
Source IPv4 address:	
Address prefix:	

Router B is the border router for another 6to4 site, and is also the IPv6 router for that site. Router B is advertising a 6to4 prefix on each subnet. The upper 48 bits of each 6to4 prefix are identical to the 6to4 site prefix. Figure 2.13 shows the worksheet for router B.

Figure 2.13. 6to4 Tunnel Host E Configuration



IPv6 Configuration

IPv6 router: [✓ yes] no
DNS/BIND automatic updates (hosts only): yes [✓ no]
IPv6 interfaces: WE0 WE1 WE2
6to4 tunnel: [✓ yes] no
Configured tunnel: yes [✓ no]
Automatic tunnel: yes [✓ no]
Manual routes: yes [✓ no]
Start IPv6: [✓ yes] no

6to4 Tunnel

Host Address: 1.2.3.4
Site Prefix: 2002:102:304::/48
Address prefix (hosts only):
Relay router address: 2002:90a:b0c:1::1

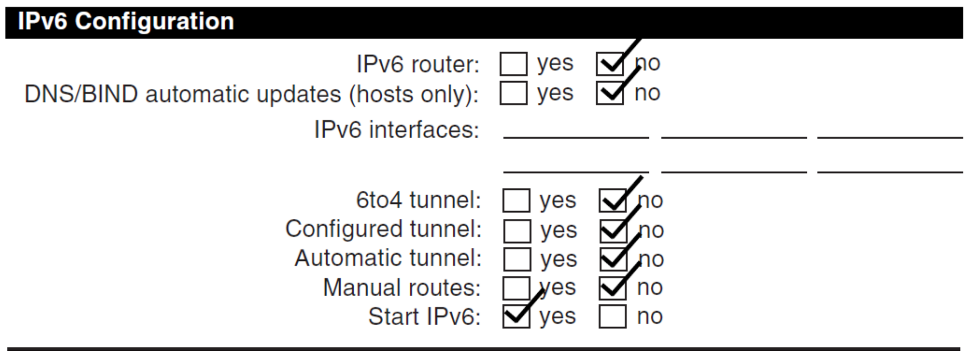
Router

Interface: WE1
RIPng: [✓ yes] no
Address prefix: 2002:102:304:1::/64

Interface: WE2
RIPng: [✓ yes] no
Address prefix: 2002:102:304:2::/64

Figure 2.14 shows the worksheet for host B. Because router B is advertising a 6to4 address prefix on the subnet, host B autoconfigures its own 6to4 address as part of its participation in the site; it does not need to configure any 6to4 tunnel interfaces.

Figure 2.14. 6to4 Tunnel Host E Configuration



IPv6 Configuration

IPv6 router: yes [✓ no]
DNS/BIND automatic updates (hosts only): yes [✓ no]
IPv6 interfaces:
6to4 tunnel: yes [✓ no]
Configured tunnel: yes [✓ no]
Automatic tunnel: yes [✓ no]
Manual routes: yes [✓ no]
Start IPv6: [✓ yes] no

2.4. Configuring IPv6

You can configure your system as either an IPv6 host or an IPv6 router. In either case, you perform the following steps:

1. Configure your IPv4 stack through the menu-driven TCPIP$CONFIG configuration procedure. This procedure is described in the *VSI TCP/IP Services for OpenVMS Installation and Configuration* manual.

Note

Add the following line to your LOGIN.COM file:

```
$ @SYS$MANAGER:TCPIP$DEFINE_COMMANDS.COM
```

This command procedure defines the UNIX management utilities as foreign commands. Rerun your LOGIN.COM to make the definitions effective for the current process.

2. Run the TCPIP$IP6_SETUP command procedure, described in Section 2.5 and Section 2.6, to configure your system as an IPv6 host or router.
3. Shut down TCP/IP Services and then restart it to enable IPv6.
4. Perform additional configuration tasks depending on whether you've configured your system as a host or router.

Once you have completed configuring your system as either an IPv6 host or router, you can optionally configure your system as a BIND server (described in Chapter 3) or as a correspondent node to support mobile IPv6, described in Chapter 5.

You may want to make other changes to your IPv6 configuration after initial setup. Chapter 4 describes how to make further changes.

2.5. Configuring an IPv6 Host

Before running the TCPIP$IP6_SETUP command procedure, make sure that you have configured your system for IPv4 by running TCPIP$CONFIG.

2.5.1. Run TCPIP$IP6_SETUP to Configure Host

To configure your system as an IPv6 host, do the following:

1. Invoke the TCPIP$IP6_SETUP command procedure by entering the following command:

   ```
   $ @SYS$MANAGER:TCPIP$IP6_SETUP
   ```

   The utility displays information about the IPv6 network configuration procedure and tells you that you can configure the system as either an IPv6 host or an IPv6 router.

2. Choose to configure the system as an IPv6 host by responding to the following prompt:

   ```
   Configure this system as an IPv6 router? [NO]:
   ```

   Press Return to configure the system as an IPv6 host.

3. Indicate whether you want to configure a 6to4 interface:

   ```
   Configure a 6to4 interface? [NO]:
   ```

   A 6to4 interface is needed only if the node is an isolated host with no connection to an IPv6 network.

   Press Return if you do not want to configure a 6to4 interface. The procedure goes to step 8.

   Enter Y and press Return, if you want to configure a 6to4 interface. You'll be prompted to enter information about the interface in subsequent steps.

4. Enter the node's IPv4 address in response to the following prompt:

```
Enter this node's IPv4 address:
```

When you enter your node's IPv4 address and press Return, an IPv6 address prefix is automatically generated and displayed.

5. You are prompted to enter the address prefix for the 6to4 tunnel:

   ```
   Enter the address prefix to use on TNn:
   ```

   Press Return to enter the IPv6 address prefix generated in the previous step.

6. Indicate whether you want this node to connect to native IPv6 sites:

   ```
   Connectivity to native IPv6 sites? [NO]:
   ```

   A relay router is needed to connect your system to native IPv6 sites. If a relay router is not specified, your system can connect to other 6to4 sites but not to native IPv6 sites.

   If you do not want your system to connect to native IPv6 sites, press Return. The procedure goes to step 8.

   If you want your system to connect to native IPv6 sites, enter Y and press Return.

7. Indicate the address of a relay router:

   ```
   Enter 6to4 address of a 6to4 Relay Router [2002:C058:6301::]:
   ```

   The address of the default relay router is displayed. Press Return to use the default, or enter another 6to4 relay router address, and then press Return. The procedure goes to step 8.

8. Answer the prompts about configuring each interface on your system. The procedure displays the following questions:

   ```
        Do you want to enable IPv6 on this interface?
   ```

   ```
   Enable IPv6 on interface WE0? [YES]:
   ```

   Press Return if you want to enable IPv6 on this interface; enter N if you do not.

   If your system has multiple interfaces, the procedure repeats this questions for each interface.

9. Indicate whether you want to configure an automatic tunnel by responding to the following prompt:

   ```
   Configure an IPv6 over IPv4 automatic tunnel interface? [NO]:
   ```

   If you want to configure an automatic tunnel, enter Y and press Return; if not, press Return.

10. Indicate whether you want to create a configured tunnel or additional configured tunnels by responding to the following prompt:

    ```
    Create a configured tunnel? [NO]:
    ```

    If you want to create a configured tunnel, enter Y and press Return. You will be prompted for source and destination addresses in steps 11 and 12.

    If you do not want to create a configured tunnel or if you have finished adding a series of configured tunnels, press Return. The procedure goes to step 14.

11. If you chose to create a configured tunnel, enter the tunnel's source IPv4 address in response to the following prompt:

    ```
    Source IPv4 address of tunnel IT0?:
    ```

    Enter an IPv4 address in the standard format (*xx.xx.xx.xx*) and press Return.

12. Enter the tunnel's destination IPv4 address in response to the following prompt:

    ```
    Destination IPv4 address of tunnel IT0?:
    ```

    Enter an IPv4 address in the following format (*xx.xx.xx.xx*) and press Return.

    - Indicate whether you want to create another configured tunnel by responding to the following prompt:

    ```
    Create another configured tunnel? [NO]
    ```

    If you want to create another configured tunnel, enter Y and press Return. The procedure takes you back to steps 10 through 12 for each additional configured tunnel you choose to create. If you do not want to create another configured tunnel, press Return.

    - The procedure asks whether you want to create a host configuration file based on the choices you have made.

    ```
         Create IPv6 Host configuration file?
    ```

    ```
    Please enter YES or NO [YES]:
    ```

    If you are not satisfied with the configuration, enter N and press Return. The utility ends immediately without changing any of the current configuration files.

    If you are satisfied with the configuration, enter Y and press Return. The TCPIP $IP6_SETUP command procedure creates a configuration file called SYS$SYSTEM:TCPIP $INET6_CONFIG.DAT. When you restart TCP/IP Services, a process called TCPIP$ND6HOST will be started automatically.

2.5.2. DNS Domain Name and Address Registration

After you shut down TCP/IP Services and before you restart it, you can use the TCPIP$ND6HOST process to register the host's domain name and address in the DNS.

The TCPIP$ND6HOST process receives and processes IPv6 router advertisement (RA) packets of the neighbor discovery protocol. This enables a system to autoconfigure itself without manual intervention. With this version of TCP/IP Services, you can also enable DNS registration.

To enable host name and address registration, enter the following command:

```
$ DEFINE /SYSTEM TCPIP$ND6D_ENABLE_DDNS 1
```

The domain name to be registered is obtained using the `gethostname()` call.

To update the zone, TCPIP$ND6HOST sends dynamic updates to the primary master name server. To determine the master name server, a query for the zone's SOA record is sent to the name server specified in the DNS resolver configuration. To display this information, use the TCP/IP management command SHOW NAME. The name of the primary master server is stored in the SOA MNAME field.

To make use of this feature, you must enable dynamic updates. By default, dynamic updates are rejected by DNS servers. For information about allowing dynamic updates, see the BIND Chapter of the *VSI TCP/IP Services for OpenVMS Management* guide.

2.6. Configuring an IPv6 Router

Before running the TCPIP$IP6_SETUP command procedure, make sure that you have configured your system for IPv4 by running TCPIP$CONFIG.

You must also enable forwarding by setting the **ipv6forwarding** and **ipv6router** attributes of the kernel `inet` subsystem to 1. You set these attributes temporarily by entering the following `sysconfig` commands:

```
$ sysconfig -r inet ipv6forwarding=1
$ sysconfig -r inet ipv6router=1
```

See the *VSI TCP/IP Services for OpenVMS Tuning and Troubleshooting* manual to modify these attributes permanently.

2.6.1. Running TCPIP$IP6_SETUP to Configure Router

To configure your system as an IPv6 router, do the following:

1. Invoke the TCPIP$IP6_SETUP command procedure by entering the following command:

   ```
   $ @SYS$MANAGER:TCPIP$IP6_SETUP
   ```

   The utility displays information about the IPv6 network configuration procedure and tells you that you can configure the system as either an IPv6 host or an IPv6 router.

2. Choose to configure the system as an IPv6 router by responding to the following prompt:

   ```
   Configure this system as an IPv6 router? [NO]:
   ```

   If you want to configure the system as an IPv6 router, enter Y and press Return.

3. Indicate whether you want this system to function as a 6to4 border router:

   ```
   Configure this system as a 6to4 Border IPv6 router? [NO]:
   ```

   To operate as a 6to4 border router, the IPv6 site to which this system is attached must have at least one valid, globally unique IPv4 address and must be configured on a network segment attached to the wide-area IPv4 network.

   If you do not want the system to function as a 6to4 border router, press Return. The procedure goes to step 8.

   If you want the system to function as a 6to4 border router, enter Y and press Return.

4. Indicate whether you want to configure a 6to4 interface on the border router system:

   ```
   Configure a 6to4 interface? [NO]:
   ```

   To communicate with other 6to4 sites over an IPv4 wide-area network without tunneling or to communicate with native IPv6 sites using 6to4 relay routers, you need to configure a 6to4 interface.

If you do not want to configure a 6to4 interface, press Return. The procedure goes to step 8. If you want to configure a 6to4 interface, enter Y and press Return. You'll be prompted for further information in subsequent steps.

5. Enter this node's IPv4 address:

   ```
   Enter this node's IPv4 address:
   ```

   Enter the IPv4 address for your system and press Return. A 6to4 site prefix is automatically generated and displayed.

6. Indicate whether you want the system to have connectivity to native IPv6 sites:

   ```
   Connectivity to native IPv6 sites? [NO]:
   ```

   A relay router is needed to connect your system to native IPv6 sites. If a relay router is not specified, your system can connect to other 6to4 sites but not to native IPv6 sites.

   If you do not want your system to connect to native IPv6 sites, press Return. The procedure goes to step 8.

   If you want your system to connect to native IPv6 sites, enter Y and press Return.

7. Indicate the address of a relay router:

   ```
   Enter 6to4 address of a 6to4 Relay Router [2002:C058:6301::]:
   ```

   Press Return to use the default relay router anycast address. Or enter another 6to4 relay router address, then press Return. The procedure goes to step 8.

8. Answer the prompts about configuring each interface on your system. The procedure displays the following questions:

   ```
        Do you want to enable IPv6 on this interface?
   ```

   ```
   Enable IPv6 on interface WE0? [YES]:
   ```

   Press Return if you want to enable IPv6 on this interface; enter N if you do not.

9. Answer the prompts about enabling IPv6 routing on each interface on your system. The procedure displays the following questions:

   ```
        Do you want to enable IPv6 routing on this interface?
   ```

   ```
   Enable IPv6 routing on interface WE0? [YES]:
   ```

   Press Return if you want to enable IPv6 routing on this interface; enter N if you do not.

10. Indicate whether you want the router to run the RIPng protocol on the designated interface by responding to the following prompt:

    ```
    Enable RIPng on interface WE0? [YES]:
    ```

    If you want the router to run the RIPng protocol, press Return; enter N and press Return if you do not.

11. Indicate whether you want the router to advertise an IPv6 address prefix for the LAN on the designated interface, by responding to the following prompt:

```
Address prefix to advertise on interface WE0?:
```

If you want the router to advertise an IPv6 address prefix, enter a 64-bit address prefix for the interface and press Return. The procedure repeats the same prompt. You can enter as many additional prefixes as you want for the interface. When you are finished, enter Done and press Return.

If you do not want the router to advertise an IPv6 address prefix on the designated interface, enter Done and press Return.

If there are additional interfaces on your system, the procedure returns to steps 8 through 11 for each interface. Once you have configured all interfaces, the procedure goes to step 12.

12. Indicate whether you want to configure an automatic tunnel by responding to the following prompt:

    ```
    Configure an IPv6 over IPv4 automatic tunnel interface? [NO]:
    ```

    If you want to configure an automatic tunnel, enter Y and press Return; if not, press Return.

13. Indicate whether you want to create a configured tunnel or additional configured tunnels by responding to the following prompt:

    ```
    Create a configured tunnel? [NO]:
    ```

    If you want to create a configured tunnel, enter Y and press Return. You will be prompted for source and destination addresses in steps 14 and 15.

    If you do not want to create a configured tunnel or if you have finished adding a series of configured tunnels, press Return. The procedure goes to step 20.

14. If you chose to create a configured tunnel, enter the tunnel's source IPv4 address in response to the following prompt:

    ```
    Source IPv4 address of tunnel IT0?:
    ```

    Enter an IPv4 address in the standard format (*xx.xx.xx.xx*) and press Return.

15. Enter the tunnel's destination IPv4 address in response to the following prompt:

    ```
    Destination IPv4 address of tunnel IT0?:
    ```

    Enter an IPv4 address in the following format (*xx.xx.xx.xx*) and press Return.

16. Indicate whether you want to enable IPv6 routing on the interface by responding to the following prompt:

    ```
    Enable IPv6 routing on interface IT0? [YES]:
    ```

    If you want to enable IPv6 routing on the interface, press Return; if not, enter N and press Return.

17. Indicate whether you want to enable RIPng on the interface by responding to the following prompt:

    ```
    Enable RIPng on interface IT0? [YES]:
    ```

Press Return if you want to enable RIPng protocol on this interface; enter N and press Return if you do not.

18. Indicate whether you want the host to use an IPv6 address prefix on the tunnel interface by responding to the following prompt:

    ```
    Address prefix to advertise on interface IT0?:
    ```

    If you want the host to use an IPv6 address prefix because a router is not advertising a global address prefix, enter the prefix and press Return. Enter as many prefixes as you want. When you are finished entering prefixes for the interface, enter Done and press Return.

    If you do not want the host to use an IPv6 address prefix on the tunnel interface, enter Done and press Return.

19. Indicate whether you want to create another configured tunnel by responding to the following prompt:

    ```
    Create another configured tunnel? [NO]:
    ```

    If you want to create another configured tunnel, enter Y and press Return. The procedure returns to step 13.

    If you do not want to create another configured tunnel, press Return.

20. The TCPIP$IP6_SETUP command procedure displays the configuration information and asks you to indicate whether you want to update the current startup procedures with the new configuration information.

    ```
         Create IPv6 Router configuration files?
    ```

    ```
    Please enter YES or NO [YES]:
    ```

    If you are not satisfied with the configuration, enter N and press Return. The utility ends immediately without changing any of the current configuration files.

    If you are satisfied with the configuration, enter Y and press Return. The TCPIP $IP6_SETUP command procedure creates a configuration file called SYS$SYSTEM:TCPIP $INET6_CONFIG.DAT and a router configuration file called SYS$SYSTEM:TCPIP $IP6RTRD.CONF, both with default values. When you restart VSI TCP/IP Services for OpenVMS, the TCIPI$IP6RTRD process starts automatically.

2.6.2. TCPIP$IP6RTRD.CONF Configuration File

At startup, the TCPIP$IP6RTRD process reads the TCPIP$IP6RTRD.CONF file to obtain data needed to send router advertisement and RIPng messages. The TCPIP$IP6RTRD.CONF file is created when TCPIP$IP6_SETUP is run, if the system is configured as a router. Initially, the link interface and advertised prefix are inserted, and other default values are used.

The TCPIP$IP6RTRD.CONF file consists of structured information for each interface in the following format:

```
interface interface-name {
   # interface keyword-value pairs, one per line
   Prefix prefix/length {
```

```
              # prefix keyword-value pairs, one per line
          }
       }
```

Comments begin with the pound sign (#) and continue to the end of the line. Accepted and default values for the interface keywords and prefix keywords are listed in Section 2.6.2.1 and Section 2.6.2.2. Section 2.6.2.3 contains a sample configuration file.

2.6.2.1. Interface Keyword Information for TCPIP$IP6RTRD.CONF

The following basic keywords are defined in RFC 2461 for IPv6 operation:

- AdvCurHopLimit

  Specifies the value to be placed in the Cur Hop Limit field in the Router Advertisement messages sent by the router. The value 0 (zero) means unspecified (by this router). Valid values are any nonnegative integer. The default is 64.

- AdvDefaultLifetime

  Specifies a time, in seconds, that is placed in the Router Lifetime field in the router advertisement. Valid values are between 0 or MaxRtrAdvInterval and 9000, inclusive. The default is 1800 seconds.

- AdvLinkMTU

  Specifies a nonnegative integer value to be placed in MTU options sent by the router. The default is 0.

- AdvManagedFlag

  Enables (1) or disables (0) the setting of a flag in the "Managed address configuration" flag field in the router advertisement. The default is 0.

- AdvOtherConfigFlag

  Enables (1) or disables (0) the setting of a flag in the "Other stateful configuration" flag field in the router advertisement. The default is 0.

- AdvReachableTime

  Specifies a time, in milliseconds, that is placed in the Reachable Time field in router advertisement messages. Valid values are between 0 and 3,600,000 (1 hour), inclusive. The default is 0 milliseconds.

- AdvRetransTimer

  Specifies a nonnegative integer value to be placed in the Retrans Timer field in the router advertisement. The default is 0 (zero).

- AdvSendAdvertisements

  Enables (yes) or disables (no) the sending of periodic Router Advertisements and responding to Router Solicitations. The default is yes.

- MaxRtrAdvInterval

Specifies the maximum time, in seconds, between sending unsolicited multicast router advertisements from the interface. Valid values are between 4 and 1800 seconds, inclusive. The default is 600 seconds.

- MinRtrAdvInterval

  Specifies the minimum time, in seconds, between sending unsolicited multicast router advertisements from the interface. Valid values are between 3 and .75 * MaxRtrAdvInterval. The default is 200 seconds.

The following additional interface keywords are accepted:

- AdvSendLinkLayerAddress

  Enables (yes) or disables (no) the sending of the interface link-layer address option in outgoing router advertisements. The default is yes.

- AdvSendSiteLocal

  Enables (yes) or disables (no) the sending of site local prefixes in outgoing router advertisements. The default is no.

- PoisonReverse

  Enables (1) or disables (0) the Poisoned Reverse algorithm as specified in RFC 2080. The default is 1.

- ripng

  Enables (yes) or disables (no) participation in RIPng on the interface. If enabled, RIPng updates are sent on the interface, and received RIPng updates are processed as defined in RFC 2080. You cannot specify yes for automatic tunnels (the tun0 interface). The default is yes (except for tun0).

- SplitHorizon

  Enables (1) or disables (0) the Split Horizon algorithm as specified in RFC 2080. The default is 1.

2.6.2.2. Address-Prefix Keyword Information for TCPIP $IP6RTRD.CONF

Each address prefix to be configured on the interface must be defined within a prefix block that begins with the keyword Prefix followed by the prefix and length (separated by a slash [/] and optionally followed by an additional address-prefix information block of keyword-value pairs).

The following address prefix keywords and values are defined in RFC 2461:

- AdvAutonomousFlag

  Enables (1) or disables (0) the setting of the Autonomous Flag field in the Prefix Information option. The default is 1.

- AdvOnLinkFlag

  Enables (1) or disables (0) the setting of the on-link flag field in outgoing router advertisements. The default is 1.

- AdvPreferredLifetime

  Specifies the preferred lifetime of the address prefix, in seconds, to be placed in outgoing router advertisements. The default is 604800 seconds, or 7 days.

- AdvValidLifetime

  Specifies the valid lifetime of the address prefix, in seconds, to be placed in outgoing router advertisements. The default is 2592000 seconds, or 30 days.

The following address prefix keywords and values are defined in RFC 2080:

- RouteMetric

  Specifies a value that represents the total cost of getting a datagram from the router to a destination. Valid values are between 1 and 16, inclusive. The default is 1.

- RouteTag

  Specifies a integer that is assigned to a route and must be preserved and readvertised with a route. The default is 0.

In addition, you can specify the following address-prefix keywords:

- ConfigureThisPrefix

  The TCPIP$IP6RTRD process will configure the advertised prefix on the interface if ConfigureThisPrefix is specified and set to 1, or if ConfigureThisPrefix is not specified and AdvAutonomousFlag is set to 1.

  The prefix is not autoconfigured in all other cases. Valid values are 0 and 1. The default value is the value of AdvAutonomousFlag.

- Gateway

  Specifies an IPv6 address to use as an off-link route to a gateway. You can use this mechanism to set up default routes.

- SendInAdvertisement

  Enables (yes) or disables (no) the sending of the address prefix in routine advertisements. The default is yes.

Each address to be configured on the interface must be defined within a address block that begins with the keyword Address followed by the IPv6 address and optionally followed by an additional address information block of keyword-value pairs. The address value is the 128-bit IPv6 address, as follows:

x:x:x:x:x:x:x:x

In this format, each x is the hexadecimal value of a 16-bit piece of the address. An IPv6 address typically consists of a 64-bit prefix followed by a 64-bit interface identifier.

You can specify the following address keywords and values:

- Anycast

  Configures (yes) or unconfigures (no) the specified address as an anycast address. The default is no.

- ConfigureThisAddress

  Configures (yes) or unconfigures (no) the specified address on the interface. The default is yes.

- Gateway

  Specifies an IPv6 address to use as an off-link route to a host. You can use this mechanism to set up host routes.

The following address keywords and values are defined in RFC 2080:

- RouteMetric

  Specifies a value that represents the total cost of getting a datagram from the router to a destination. Valid values are between 1 and 16, inclusive. The default is 1.

- RouteTag

  Specifies a integer that is assigned to a route and must be preserved and readvertised with a route. The default is 0.

For related information, see the following RFCs:

- RFC 2461, Neighbor Discovery for IP version 6 (IPv6), Narten, T.; Nordmark, E., Simpson W. A., December 1998
- RFC 2462, IPv6 Stateless Address Autoconfiguration, Thompson, S.; Narten, T., December 1998
- RFC 2080, RIPng for IPv6, Malkin, G., Minnear, R., January 1997

2.6.2.3. Editing the Router Configuration File

The SYS$SYSTEM:TCPIP$IP6RTRD.CONF file contains the configuration data needed to send router advertisement messages. This file is created when TCPIP$IP6_SETUP is run (if the system is configured as a router). The link interface and advertised prefix are inserted, and other default values are used.

You can modify this file as appropriate for your network, for example, when using multiple prefix values. Example 2.1 shows a sample configuration file.

Example 2.1. Sample TCPIP$IP6RTRD.CONF File

```
#
# Sample ip6rtrd configuration file
#
interface WE0 {
        MaxRtrAdvInterval 600
        MinRtrAdvInterval 200
        AdvManagedFlag 0
        AdvOtherConfigFlag 0
        AdvLinkMTU 1500
        AdvReachableTime 0
        AdvRetransTimer 0
        AdvMaxHopLimit 64
        AdvDefaultLifetime 1800
        Prefix dec:1::/64 {
                AdvValidLifetime 1200
                AdvPreferredLifetime 600
                AdvOnLinkFlag 1
                AdvAutonomousFlag1
        }
}
```

Chapter 3. Configuring BIND

The information in this chapter is for experienced DNS/BIND administrators. See the VSI TCP/IP Services for OpenVMS Management manual for more information on BIND.

3.1. IPv6 Support in BIND Version 9

BIND supports all forms of IPv6 name-to-address and address-to-name lookups. It can also accept queries over an IPv6 (AF_INET6) connection and use IPv6 addresses to make queries when running on an IPv6-capable system.

Note

The BIND resolver has not yet been ported to communicate over IPv6 connections. Using `getaddrinfo ()` and `getnameinfo ()` calls, IPv6 applications are able to retrieve IPv6 address information contained in AAAA and PTR records over an IPv4 transport until the BIND resolver is ported to IPv6.

3.1.1. Address lookups Using AAAA records

For name-to-address lookups, using AAAA records is recommended because A6 records have been moved to experimental status. Like most stub resolvers, the resolver in TCP/IP Services supports only AAAA lookups because of the difficulty in following A6 chains. The AAAA record for IPv6 is analogous to the A record for IPv4. It specifies an entire address in a single record. For example,

```
$ORIGIN ipv6.my.zone.

host1        IN       AAAA       5f00:0000:0102:0300:0203:0800:2b0a:0b0c
```

3.1.2. Name Lookups Using Nibble Format

For address-to-name lookups, the nibble format is recommended because use of the bitstring format has been moved to experimental status. Use of the ip6.arpa IPv6 reverse mapping zone defined in RFC 3152 is recommended because the ip6.int IPv6 address space defined in RFC 1886 has been deprecated and will likely be phased out in the future.

As in IPv4, when looking up an address in nibble format, the address components are simply reversed and ip6.arpa. is appended to the resulting name. For example, the following would provide reverse lookup for a host with the address 5f00:0000:0102:0300:0203:0800:2b0d:0e0f:

```
$ORIGIN 3.0.2.0.0.0.3.0.2.0.1.0.0.0.0.0.0.0.0.f.5.ip6.arpa.

f.0.e.0.d.0.b.2.0.0.8.0        IN        PTR        host2.ipv6.my.zone.
```

3.1.3. Using DNAME To Rename ip6.int

The deprecation of the ip6.int IPv6 reverse mapping zone has resulted in an issue for existing clients that will continue to search the ip6.int name space for PTR resource records. Administrators will need to continue to provide PTR data under both of these zones to be compatible with both old and new clients. There is a convenient method using DNAME resource records that can ease administration of this data. The DNAME resource record is used to substitute one suffix of a domain name with another.

In this case it will substitute your ip6.int zone suffix with the equivalent ip6.arpa zone suffix. For example, the following DNAME resource record accomplishes the substitution:

```
$ORIGIN 3.0.2.0.0.0.3.0.2.0.1.0.0.0.0.0.0.0.f.5.ip6.int.

        DNAME    3.0.2.0.0.0.3.0.2.0.1.0.0.0.0.0.0.0.f.5.ip6.arpa.
```

This approach will work for any point in the name space as long as all authoritative servers for the PTR zone fully implement DNAME resource record behavior as specified in RFC 2672. This includes BIND9 servers but excludes BIND8 servers.

3.1.4. Enabling IPv6 Interfaces

For IPv6, the BIND server does not bind a separate socket to each interface address as it does for IPv4. Instead, it listens on the IPv6 wildcard address, which is not enabled by default. You must use the listen-on-v6 option to specify the ports on which the server will listen for incoming queries sent using IPv6. To enable the BIND server to answer IPv6 queries, you must specify the port in the options statement of the BIND server configuration file. The only values allowed for the option are { any; } and { none; }. For example, to listen on the default port 53 specify the following:

```
    listen-on-v6 { any; };
```

To listen on port 1234, specify the following:

```
    listen-on-v6 port 1234 { any; };
```

If you do not specify the listen-on-v6 option, the BIND server will not listen on any IPv6 interfaces.

3.2. Sample BIND Configuration Files

The SYS$COMMON:[SYSHLP.EXAMPLES.TCPIP.IPV6.BIND] directory contains DNS configuration and data files that show sample IPv6 information for you to study and adapt to your environment.

Example 3.1 shows a sample BIND Server configuration file. This file is the mechanism used by BIND for pointing the server to its zone data files.

Example 3.1. Sample TCPIP$BIND.CONF_IPV6

```
#
# File name:      TCPIP$BIND.CONF_IPV6
# Product:        hp TCP/IP Services for OpenVMS
# Version:        V5.4-00
#
# © Copyright 1976, 2003 Hewlett-Packard Development Company, L.P.
#

#
# Example IPv6 BIND server configuration
#

options {
        directory "sys$specific:[tcpip$bind]";
        #
        # (listen-on-v6 is for BIND 9 and later)
        # Unless this option is specified, the server
        # does not listen on any IPv6 addresses.
        #      Use: listen-on-v6 { any; };
        #
};

zone "ipv6.my.zone" {
 type master;
 file "ipv6.db";
};

zone "3.0.2.0.0.0.3.0.2.0.1.0.0.0.0.0.0.0.f.5.IP6.ARPA" {
 type master;
 file "ipv6.arpa";
};

zone "3.0.2.0.0.0.3.0.2.0.1.0.0.0.0.0.0.0.f.5.IP6.INT" {
 type master;
 file "ipv6.int";
};

zone "0.0.127.in-addr.arpa" {
 type master;
 file "127_0_0.db";
};

zone "localhost" in {
        type master;
        file "localhost.db";
};

zone "." {
        type hint;
        file "root.hint";
};
```

Example 3.2 shows the forward mapping data file for the ipv6.my.zone zone. Note that both AAAA resource records (IPv6) and A resource records (IPv4) can be included in a zone. Administrators may wish to delegate a separate zone containing only IPv6 resource records for convenience.

Example 3.2. Sample IPV6.DB File

```
;
; File name:      IPV6.DB
; Product:        hp TCP/IP Services for OpenVMS
; Version:        V5.4-00
;
; © Copyright 1976, 2003 Hewlett-Packard Development Company, L.P.
;

;
; Example BIND data file for ipv6.my.zone
;

$TTL 1d
@       IN      SOA     ns.ipv6.my.zone. postmaster.ipv6.my.zone. (
                        1       ; Serial
                        3600    ; Refresh
                        300     ; Retry
                        3600000 ; Expire
                        3600 )  ; Minimum
;
; Nameservers
;

        IN      NS      ns.ipv6.my.zone.
        IN      NS      ns.ipv4.my.zone.
;
; IPv6 nodes
;

host1   IN      AAAA 5F00:0000:0102:0300:0203:0800:2B0A:0B0C
host2   IN      AAAA 5F00:0000:0102:0300:0203:0800:2B0D:0E0F

;
; IPv4 and IPv6 nodes
;

host3   IN      AAAA 5F00:0000:0102:0300:0203:0800:2B0C:0B0A
        IN      A       10.20.30.40
host4   IN      A       10.30.40.50
```

Example 3.3 shows the reverse mapping data file for the 3.0.2.0.0.0.3.0.2.0.1.0.0.0.0.0.0.0.f.5.ip6.arpa zone.

Example 3.3. Sample IPV6.ARPA File

```
;
; File name:      IPV6.ARPA
; Product:        hp TCP/IP Services for OpenVMS
; Version:        V5.4-00
;
; © Copyright 1976, 2003 Hewlett-Packard Development Company, L.P.
;

;
; Example BIND data file for
 3.0.2.0.0.0.3.0.2.0.1.0.0.0.0.0.0.0.f.5.IP6.ARPA
; (corresponds to the 5F00:0000:0102:0300:0203::/80 prefix)
;

$TTL 1d
@       IN      SOA     ns.ipv6.my.zone. postmaster.ipv6.my.zone. (
                        1       ; Serial
                        3600    ; Refresh
                        300     ; Retry
                        3600000 ; Expire
                        3600 )  ; Minimum

;
; Nameservers
;

        IN      NS      ns.ipv6.my.zone.
        IN      NS      ns.ipv4.my.zone.

;
```

```
; IPv6 nodes
;

c.0.b.0.a.0.b.2.0.0.8.0 IN PTR host1.ipv6.my.zone.
f.0.e.0.d.0.b.2.0.0.8.0 IN PTR host2.ipv6.my.zone.
a.0.b.0.c.0.b.2.0.0.8.0 IN PTR host3.ipv6.my.zone.
```

Example 3.4 shows a sample IPV6.INT data file containing the single DNAME resource record that accomplishes the ip6.int renaming as discussed in Section 3.1.3.

Any data added to the ip6.arpa name space in the IPV6.ARPA zone data file will now also be available in the ip6.int name space. No changes need to be made to the IPV6.INT zone data file. The IPV6.INT and IPV6.ARPA zone statements in the BIND server configuration file are the same as those in Example 3.1.

Example 3.4. Sample IPV6.INT File

```
;
; File name:      IPV6.INT
; Product:        hp TCP/IP Services for OpenVMS
; Version:        V5.4-00
;
; © Copyright 1976, 2003 Hewlett-Packard Development Company, L.P.
;

;
; Example BIND data file for
 3.0.2.0.0.0.3.0.2.0.1.0.0.0.0.0.0.0.0.f.5.IP6.INT
; (corresponds to the 5F00:0000:0102:0300:0203::/80 prefix)
;

$TTL 1d
@       IN      SOA     ns.ipv6.my.zone. postmaster.ipv6.my.zone. (
                        1       ; Serial
                        3600    ; Refresh
                        300     ; Retry
                        3600000 ; Expire
                        3600 )  ; Minimum

;
; Nameservers
;

        IN      NS      ns.ipv6.my.zone.
        IN      NS      ns.ipv4.my.zone.

;
; DNAME record
;

        DNAME   3.0.2.0.0.0.3.0.2.0.1.0.0.0.0.0.0.0.0.f.5.ip6.arpa.
```

Chapter 4. Managing and Monitoring the IPv6 Network

Once you have configured your system for IPv6, you may want to make changes to your configuration or monitor the network. TCP/IP Services for OpenVMS supplies commands to do both.

Extensions to existing management commands and a new IPv6 command allow you to perform typical management functions. Section 4.1 describes these commands.

Section 4.2 describes typical IPv6 management tasks, with examples.

Section 4.3 describes UNIX-style management tools to monitor the network.

Section 4.4 describes log files that you can use to monitor network performance.

4.1. IPv6 Extensions to Management Commands

The *VSI TCP/IP Services for OpenVMS Management Command Reference* manual describes the basic management commands, including the UNIX commands, you can use to manage the TCP/IP Services software. The *VSI TCP/IP Services for OpenVMS Tuning and Troubleshooting* manual contains more detailed information about the UNIX management commands. The following sections describe only IPv6 extensions to those management commands.

To use UNIX management commands at the DCL prompt, execute the following command procedure (or put it into your LOGIN.COM so that it executes each time you log in):

```
$ @SYS$MANAGER:TCPIP$DEFINE_COMMANDS
```

Note

UNIX flags and OpenVMS interface names are case sensitive. When entering UNIX management commands at the DCL prompt, you must enclose uppercase UNIX flags and OpenVMS interface names in quotation marks to preserve the case of the input.

4.1.1. ifconfig Command

For the AF_INET6 address family, use the following syntax:

```
ifconfig interface_id address_family [[ip6prefix]
  address[/bitmask] [dest_address]] [parameters]
```

For the AF_INET6 address family, the address argument is either a host name or the 128-bit IPv6 address, in the following format:

`x:x:x:x:x:x:x:x`

In this format, each *x* is the hexadecimal value of a 16-bit piece of the address.

The **ip6prefix** argument specifies that the interface identifier is to be appended to the **address** argument when configuring an address on the interface. The interface identifier uniquely identifies an interface on a subnet and is typically the interface's Link layer address. The following are the parameters for the `ifconfig` command.

Parameters [AF_INET6 only]:

- **ip6interfaceid** *id*

  Overrides the default interface ID, which depends on the underlying link type (for example, Ethernet, FDDI), and specifies an inet6 interface ID for the interface. For example, if your system has the Ethernet hardware address 08-00-2b-2a-1e-d3, the following command generates the `inet6` link-local address `fe80::a00:2bff:fe2a:1ed3` for the interface:

  ```
  $ ifconfig "WEO" ipv6
  ```

  On the same system, the following command generates the `inet6` interface ID `abcd:1234` for the interface:

  ```
  $ ifconfig "WE0" ip6interfaceid ::abcd:1234 ipv6
  ```

- **ipv6**

  Initializes IPv6-related data structures and assigns an IPv6 link-local address to the interface.

- **-ipv6**

  Removes any IPv6 configuration associated with the interface, including all IPv6 addresses and IPv6 routes through the interface. This command is equivalent to the `ifconfig interface inet6 delete` command.

- **ip6dadtries** *value*

  Specifies the number of consecutive neighbor solicitation messages that your system transmits as it performs duplicate address detection on a tentative address.

- **ip6hoplimit** *hops*

  Sets the default number of hops to be included in transmitted unicast IP packets.

- **ip6mtu** *mtu_value*

  Alters the maximum transmission unit (MTU) for messages that your system transmits on the link.

- **ip6nonud**

  Disables Neighbor Unreachability Detection (NUD) on the interface.

- **ip6reachabletime** *time*

  Sets the time, in milliseconds, that your system considers a neighbor is reachable after your system receives a reachability confirmation message.

- **ip6retranstimer** *value*

  Sets the time interval, in milliseconds, between neighbor solicitation messages to a neighbor.

Refer to the *VSI TCP/IP Services for OpenVMS Tuning and Troubleshooting* manual for more information on the `ifconfig` command.

4.1.2. iptunnel Command

The `iptunnel` command creates configured tunnels for sending and receiving IPv6 or IPv4 packets that are encapsulated as the payload of an IPv4 datagram.

The `iptunnel` command can perform the following operations:

- **create**

  Creates a tunnel interface, which you must subsequently configure by using the `ifconfig` command. The syntax of the create operation is as follows:

  ```
  iptunnel create [-I int-name] [v4-dest] [v4-src]
  ```

  Parameters

  - **-I int-***name*

    Specifies the interface unit of the tunnel to be created. This is an optional parameter. The **int-***name* parameter has the form it*x*, where *x* is the interface unit number. By default, the interface name selected for the tunnel is it*x*+1, or the value of the interface unit number of the last tunnel created plus 1.

  - **v4-***dest*

    Specifies the remote endpoint to which a tunnel is to be created.

  - **v4-***src*

    Sets the IPv4 source address in the encapsulating header. The tunnel is enabled (packets are sent and received on the tunnel) only if *v4-src* is a valid address on the system. This is an optional parameter.

- **delete**

  Deletes a tunnel interface. You must disable the tunnel before you can delete it by entering the following command:

  ```
  $ ifconfig tunnelname down  delete abort
  ```

  Then enter:

  ```
  $ iptunnel delete tunnel
  ```

- **show**

  Shows the tunnel attributes (name, tunnel endpoints, next hop for tunneled packets).

  ```
  $ iptunnel show tunnel
  ```

For related information, see RFC 2003.

4.2. Typical Management Tasks

After restarting the network with IPv6 enabled, you might want to do the following:

- Connect to the 6bone network

- Initialize a new interface for IPv6
- Create a configured tunnel
- Add addresses to or delete addresses from an interface
- Add or delete a default router
- Manually add a route for an onlink prefix

The following sections describe these tasks.

4.2.1. Connecting to the 6bone Network

The 6bone network provides a test environment for IPv6 networks. To connect to the 6bone, choose a 6bone point that is reasonably close to your normal IPv4 paths into the Internet. The 6bone web site at http://www.6bone.net contains information on how to join the 6bone and how to find an attachment point. If you want to connect to the 6bone through the Palo Alto site either before or after you configure IPv6 on your host or router, complete the following steps:

1. Register your IPv4 tunnel by sending your 6bone IPv6 address prefix and the IPv4 address of your router to the following address:

   ```
   gw-6bone@pa.dec.com
   ```

2. Wait for confirmation that support for your tunnel is configured at VSI.

   VSI will provide both an IPv6 global address prefix for you to use at your site and the IPv4 address of the Palo Alto router.

3. Configure your tunnel by running the TCPIP$IP6_SETUP utility.

4. Verify that your tunnel is operational by issuing the `ping` command to one of the following VSI IPv6 nodes:

   ```
   altavista.ipv6.digital.com
   ftp.ipv6.digital.com
   www.ipv6.hp.com
   ```

   For additional information about connecting to the 6bone, see the 6bone home page:

   `http://www.6bone.net`

4.2.2. Initializing a New Interface for IPv6

In some cases, you might want to either add a new interface card to your system or change an interface card from one type to another. After the new card is installed, you must initialize it for IPv6 operation. To initialize an interface, use the `ifconfig` command with the following syntax:

```
$ ifconfig device ipv6 up
```

Note

OpenVMS interface names must be in uppercase. When you enter them with UNIX management commands at the DCL prompt, you must enclose the name of the interface in double quotation marks.

For LAN interfaces, the `ifconfig` command creates the link-local address (FE80::) and starts detection of duplicate addresses.

For example, to initialize Ethernet interface WE0 for use with IPv6, enter the following:

```
$ ifconfig "WE0" ipv6 up
```

To initialize the loopback interface for use with IPv6, enter the following:

```
$ ifconfig "LO0" ipv6 up
```

To initialize the automatic tunnel interface, enter the following:

```
$ ifconfig "TN0" ipv6 up
```

This command designates one of the system's IPv4 addresses for use as the tunnel endpoint.

If you want the designated IPv4 address to be the permanent tunnel endpoint, you must use TCPIP $IP6_SETUP.

4.2.2.1. Setting the IPv6 Interface Identifier

You can set the IPv6 interface ID at the same time you initialize an interface by using the `ifconfig` command with the **ip6interfaceid** parameter. For example, to initialize Ethernet interface WE0 for use with IPv6 and to set its interface ID to the 64-bit value `0x0123456789abcdef`, enter the following:

```
$ ifconfig "WE0" ip6interfaceid ::0123:4567:89ab:cdef ipv6 up
```

Although the interface ID is expressed in standard IPv6 address format, only the low-order 64 bits are used.

4.2.2.2. Removing IPv6 from an Interface

Removing IPv6 from an interface removes the IPv6 configuration associated with the interface, including all IPv6 addresses and IPv6 routes through the interface. To remove IPv6 from an interface, use the `ifconfig` command with the following syntax:

```
$ ifconfig interface -ipv6
```

For example, to remove IPv6 from Ethernet interface WE0, enter the following:

```
$ ifconfig "WE0" -ipv6
```

4.2.3. Creating a Configured Tunnel

To create a configured tunnel, use the `iptunnel` command in the following format:

```
iptunnel create remote-tunnel-IPv4address
```

For example, to create a tunnel to remote system 16.20.136.47, enter the following command:

```
$ iptunnel create 16.20.136.47
```

To initialize the tunnel for IPv6 operation, enter the following command:

```
$ ifconfig "IT0" ipv6 up
```

Note

OpenVMS interface names must be in uppercase. When you enter them with UNIX management commands at the DCL prompt, you must enclose the name of the interface in quotation marks.

4.2.4. Adding an Address to an Interface

To add or assign an IPv6 prefix to an interface and to direct the kernel to automatically append the interface identifier, use the `ifconfig` command with the following syntax:

```
ifconfig interface inet6 ip6prefix prefix
```

The following example assigns the address `dec:2::0a00:2bff:fe12:3456` to interface WE0 (the interface ID is `0a00:2bff:fe12:3456`):

```
$ ifconfig "WE0" inet6 ip6prefix dec:2::/64
```

The **ip6prefix** parameter directs the kernel to automatically append the interface identifier to the address prefix.

To add or assign a full IPv6 address to an interface manually, use the `ifconfig` command with the following syntax:

```
ifconfig interface inet6 ipv6address
```

The following example assigns the address dec:2::1 to interface WE0:

```
$ ifconfig "WE0" inet6 dec:2::1
```

Note

For IPv6 hosts, the TCPIP$ND6HOST process configures interface prefixes automatically, depending on the contents of router advertisements.

For IPv6 routers, the TCPIP$IP6RTRD process configures interface prefixes automatically, depending on the contents of the SYS$SYSTEM:TCPIP$IP6RTRD.CONF file.

4.2.5. Deleting an Address from an Interface

To delete an IPv6 address from an interface manually, use the `ifconfig` command with the following syntax:

```
ifconfig interface inet6 delete ipv6address
```

For example:

```
$ ifconfig "WE0" inet6 delete dec:2::1
```

Note

OpenVMS interface names must be in uppercase. When you enter them with UNIX management commands at the DCL prompt, you must enclose the name of the interface in quotation marks.

4.2.6. Adding or Deleting a Default Router

To add a default router, use the route command with the following syntax:

```
route add -inet6 default ipv6address -I interface
```

For example:

```
$ route add -inet6 default fe80::0a00:2bff:fe12:3456 -"I" "WE0"
```

Note

UNIX flags and OpenVMS interface names are case sensitive. When entering UNIX management commands at the DCL prompt, you must enclose uppercase UNIX flags and OpenVMS interface names in quotation marks.

To delete a default router, use the route command with the following syntax:

```
route delete -inet6 default ipv6address -I interface
```

For example:

```
$ route delete -inet6 default fe80::0a00:2bff:fe12:3456 -"I" "WE0"
```

Note

For IPv6 hosts, the TCPIP$ND6HOST process performs the add and delete route operations automatically, depending on the contents of router advertisements. See Section 2.1.1 for more information about TCPIP$ND6HOST.

4.2.7. Manually Adding a Route for an On-Link Prefix

After you manually add an address prefix to an interface, you also can add a static route so that traffic to other hosts with the same prefix is sent directly to the destination rather than through a router. For example, if the prefix `DEC:5::/64` has been added to the Ethernet interface WE0, which has been initialized with the link-local address `fe80::0a00:2bff:fe12:3456`, the following command adds a route to neighboring hosts with the same prefix:

```
$ route add -inet6 dec:5::/64 fe80::0a00:2bff:fe12:3456 -interface
```

This command specifies that destinations with prefix `dec:5::0/64` are reachable through the interface with address `fe80::0a00:2bff:fe12:3456`. That is, `dec:5::0/64` is an on-link prefix.

Note

For IPv6 hosts, the TCPIP$ND6HOST process automatically adds on-link prefixes based on the contents of router advertisements. See Section 2.1.1 for more information about TCPIP$ND6HOST.

4.3. UNIX-Style Commands to Monitor the Network

To monitor your network, use the following UNIX-style commands:

- `ping` command
- `netstat` command
- `traceroute` command
- `tcpdump` command

Note

UNIX flags are case sensitive. When using an uppercase flag you must enclose it with quotation marks to get the expected behavior. OpenVMS interface names are case sensitive. The name of the interface must be enclosed in quotation marks.

The following sections describe each command.

4.3.1. ping Command

You can test access to internet network hosts with the `ping` command. The `ping` command accepts an IPv4 address, an IPv6 address, or a node name on the command line. The following sample command specifies an IPv6 address:

```
$ ping -c 2 5F00:2100:108C:4000:8C40:800:2B2D:2B2

PING (5F00:2100:108C:4000:8C40:800:2B2D:2B2): 56 data bytes
64 bytes from 5F00:2100:108C:4000:8C40:800:2B2D:2B2: icmp6_seq=0
     hlim=58 time=17 ms
64 bytes from 5F00:2100:108C:4000:8C40:800:2B2D:2B2: icmp6_seq=1
     hlim=58 time=17 ms
----5F00:2100:108C:4000:8C40:800:2B2D:2B2 PING Statistics----
2 packets transmitted, 2 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 17/17/17 ms
```

The `ping` command accepts a `-V4` or `-V6` flag to send an IPv4 ECHO_REQUEST to a node with an IPv4 address, or to send an IPv6 ECHO_REQUEST to a node with an IPv6 address, respectively. If you do not specify either flag, the `ping` command sends an appropriate ECHO_REQUEST based on the address family being used.

You can also use the `-I` flag to force the use of a specific interface. For example:

```
$ ping -"I" "WE0" FE80::800:2B2D:2B2
```

4.3.2. netstat Command

The `netstat` command displays network-related data in various formats. You can display network statistics for sockets, interfaces, and routing tables.

The parameters **-f** *address_family* limit reports to the specified address family. For example,

- `netstat -f inet` limits data to the IPv4 address family (AF_INET).
- `netstat -f inet6` limits data to the IPv6 address family (AF_INET6).

To display IPv6 routing entries, enter this command:

```
$ netstat -rnf inet6
```

To display active IPv6 connections, enter this command:

```
$ netstat -af inet6
```

To display statistics for all protocols including IPv6 and ICMPv6, enter this command:

```
$ netstat -s
```

4.3.3. traceroute Command

The `traceroute` command with the **host** argument prints the route that packets take to both IPv4 and IPv6 hosts.

The **-G** *@addr1@addr2...* parameters (IPv6 only) specify the source route for packets to travel. The route consists of one or more IPv6 node names or addresses. Use the at character (@) to separate multiple addresses. You can specify up to 10 addresses.

The **-V** *version* parameter specifies the Internet Protocol (IP) version number to enable the resolver to return the correct address. Use the **-V 4** option if you want to issue a `traceroute` command to a host name (not an IP address) that has both IPv4 and IPv6 addresses, and you want to trace the route to the IPv4 address.

Note

By default, `traceroute` tries to resolve destination host names as an IPv6 address. If that fails, it resolves the host name as an IPv4 address. You can override this behavior with the **-V** option.

In the following examples, the backslash (\) and the continuation of output onto a second line is for display purposes only. In actual output, the information appears on a single line.

```
$ traceroute -n  -"V" 6 v6host1
 traceroute to v6host1.corp.com (3ffe:1200:4110:3:a00:2bff:feb4:89c5), \
30 hops max, 24 byte packets
1  fe80::a00:2bff:fe2a:1ed3  130.86 ms 119.141 ms  119.14 ms
2  3ffe:1200:4110:1:a00:2bff:fe2d:2b2  126.014 ms  117.308 ms  116.33 ms
3  3ffe:1200:4110:3:a00:2bff:feb4:89c5 122.195 ms  135.882 ms 119.263 ms

$ traceroute 3ffe:1200:4110:3:a00:2bff:feb4:89c5
traceroute to 3ffe:1200:4110:3:a00:2bff:feb4:89c5 \
(3ffe:1200:4110:3:a00:2bff:feb4:89c5), 30 hops max, 24 byte packets
1  fe80::a00:2bff:fe2a:1ed3 (fe80::a00:2bff:fe2a:1ed3) 123.046 ms \
114.258 ms  117.188 ms
2  v6host2.corp.com (3ffe:1200:4110:1:a00:2bff:fe2d:2b2)  115.234 ms \
117.188 ms  116.287 ms
3  v6host1.corp.com (3ffe:1200:4110:3:a00:2bff:feb4:89c5)  120.241 ms \
113.398 ms  120.24 ms
```

When the route has an IPv6-over-IPv4 tunnel, `traceroute` views this as a single hop. It prints only the IPv6 addresses of the nodes at each end of a tunnel, and none of the intermediate IPv4 routers between the tunnel source and destination. If a `traceroute` command over a tunnel interface fails, run the command again and specify the tunnel's IPv4 destination address.

The following command shows a trace across the 6bone network to destination `tw4.es.net`. Note that the intermediate routers appear to drop every other message. The probable reason for this is that the routers rate-limit IPv6 ICMP error messages to one per second. Rate-limiting ICMP error messages is valid behavior.

In the following examples, the backslash (\) and the continuation of output onto a second line is for display purposes only. In actual output, the information appears on a single line.

```
$ traceroute tw4.es.net
traceroute to tw4.es.net (3ffe:780:40:1:a00:2bff:febc:e96c), 30 hops max,
 24 byte packets
1  gw1.ipv6.pa-x.dec.com (3ffe:1280:1000:1::f842:1428)  83.985 ms * 83.000
 ms
2  3ffe:700:20:1::21 (3ffe:700:20:1::21)  108.399 ms *  112.305 ms
3  3ffe:780:40:1:a00:2bff:febc:e96c(3ffe:780:40:1:a00:2bff:febc:e96c) \
124.023 ms  134.766 ms  116.211 ms
```

The following example shows a trace to destination `yogi-gbl` using 2000-byte messages. It also shows the effect of path MTU discovery on `traceroute` results.

```
$ traceroute yogi-gbl 2000
traceroute to yogi-gbl (fec0:10:60:0:200:f8ff:fe40:d8e6), 30 hops max, 2024
 byte packets
1  a30rtr-gbl (fec0:10:30:0:200:f8ff:fe45:cfb2)  5.859 ms  3.906 ms  3.907
 ms
2  fec0:10:20:0:a00:2bff:feb0:972d (fec0:10:20:0:a00:2bff:feb0:972d) \
4.882 ms  3.906 ms  3.906 ms
3  * fec0:10:40:1::a0a:283c (fec0:10:40:1::a0a:283c)  6.836 ms  6.836 ms
4  yogi-gbl (fec0:10:60:0:200:f8ff:fe40:d8e6)  8.789 ms  8.789 ms  7.812 ms
```

Hops 1 and 2 occur across Ethernet links that have a link MTU of 1500 bytes. Hop 3 occurs across a configured tunnel with an MTU of 1280 bytes.

The 1500-byte message fragments were transmitted without error until they hit the tunnel. The first fragment across hop 3 triggered a "message too big" error, which in turn caused the sender to record a reduced Path MTU for `yogi-gbl`. The sender sent all subsequent messages with smaller fragments. The `traceroute` display shows that the first probe to the tunnel was dropped but that all others succeeded.

4.3.4. tcpdump Command

The `tcpdump` command captures, parses, and prints IPv6 and ICMPv6 packets.

To see IPv6 packets, enter this command:

```
$ tcpdump interface -s 1500 [-x] [ipv6 | icmpv6]
```

The tunneling interface is not visible to the packet filter routines so you cannot trace the tunneled packets directly. To view these packets, use the actual IPv4 interface and filter for encapsulated IPv6 packets, which use the IP Protocol value 41, as follows:

```
$ tcpdump interface -s 1500 [-x] ip proto 41
```

4.4. IPv6 Process Log Files

The TCPIP$ND6HOST and TCPIP$IP6RTRD processes log informational and severe events in the TCPIP$ND6HOST.LOG and TCPIP$IP6RTRD.LOG files, which are located in the SYS $MANAGER directory.

Logging is enabled by default.

Chapter 5. Mobile IPv6

The Internet Protocol Version 6 (IPv6) was designed to support mobility through features such as extensible header structure, address autoconfiguration, security (IPsec), and tunneling. Mobile IPv6 builds on these features and defines operations that enable a mobile node to move from one link to another without changing the node's IP address. In this way, packets can be routed to and from mobile nodes transparently when they are on another network.

The Mobile IPv6 implementation has the following restrictions:

- Not supported on OpenVMS Clusters
- Does not support Binding Update authentication as specified in the IETF Internet Draft for Mobility Support in IPv6 (draft-ietf-mobileip-ipv6-15.txt). For this reason, limit the use of this implementation to test environments that are not subject to attack, since system integrity might be compromised by accepting unauthenticated bindings.

This chapter describes the following:

- Mobile IPv6 history (Section 5.1)
- Mobile IPv6 environment (Section 5.2)
- Mobile IPv6 operation (Section 5.3)
- Mobile IPv6 planning (Section 5.4)
- Mobile IPv6 configuration (Section 5.5)
- Monitoring the Mobile IPv6 environment (Section 5.6)

5.1. Mobile IPv6 History

In communications the trend is toward mobility. Mobile telephones have already transformed business and personal interactions. Computers, especially laptop computers and handhelds, are also mobile, but they currently do not enjoy the continuous connectivity that mobile telephones do.

Today, there are very basic data services that use the Wireless Application Protocol (WAP) and General Packet Radio Service (GPRS). But the demand for full voice and data mobile communications is being driven by the following trends:

- Development of third-generation (3G) networks
- Large amounts and types of content available on the Internet, including video, voice, and images
- Increasing numbers of wireless subscribers and Internet users
- Development of convergent devices that offer voice and data

5.2. Mobile IPv6 Environment

In a Mobile IPv6 environment, nodes can have the following roles:

- **mobile node**

  An IPv6 node, host or router, that can change its point of attachment from one link to another while still being reachable through its home address.

- **correspondent node**

  A peer IPv6 node with which a mobile node communicates. The correspondent node, whether a host or a router, can be either mobile or stationary. The TCP/IP Services for OpenVMS implementation of Mobile IPv6 allows a system to be a correspondent node.

- **home agent**

  A router on a mobile node's home link with which the mobile node registers its current care-of address.

To completely understand the relationship among these nodes, you should be familiar with the following terms:

- **home address**

  The IPv6 address of the mobile node when it is on its home link, or at home. The subnet prefix of this address is the home network's subnet prefix. The mobile node is always addressable by its home address; it does not change.

- **care-of address**

  The IPv6 address of the mobile node when it is on a foreign link, or away from home. The subnet prefix of this address is the foreign network's subnet prefix. A mobile node can have multiple care-of addresses, but the care-of address registered with the mobile node's home agent is called its **primary care-of address**.

- **binding**

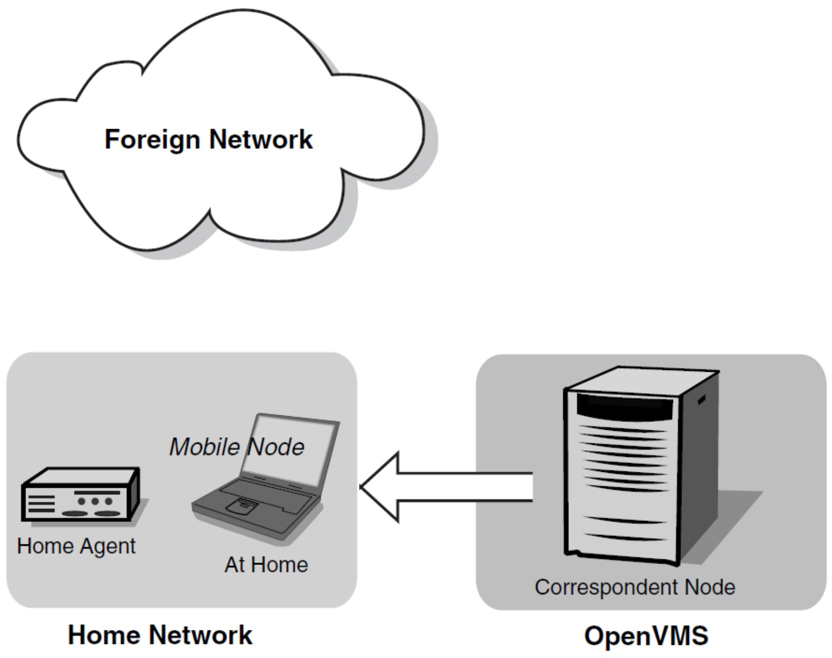
  An association of the mobile node's home address with its care-of address. This association also has a lifetime. Each node maintains a cache of all bindings. For information about viewing the contents of the binding cache, see Section 5.6.2.

5.3. Mobile IPv6 Operation

Figure 5.1, Figure 5.2, and Figure 5.3 show three scenarios that illustrate interactions among a correspondent node, home agent, and mobile node.

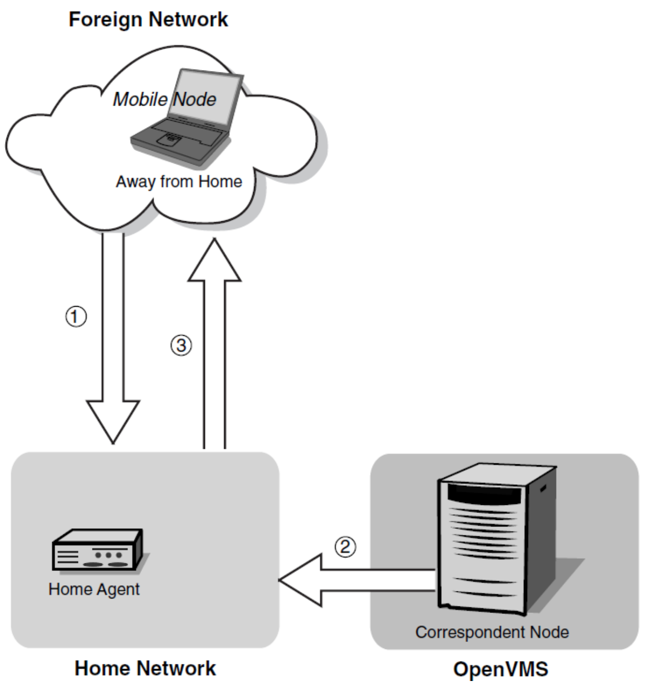
In Figure 5.1, the mobile node is on its home link (at home). Packets from the correspondent node that are addressed to the mobile node's home address are delivered through standard IP routing mechanisms.

Figure 5.1. Communication with Mobile Node at Home



In Figure 5.2, the mobile node has moved to a foreign link (away from home).

Figure 5.2. Communication with Mobile Node Away from Home – Part 1



On the foreign link, the following events occur:

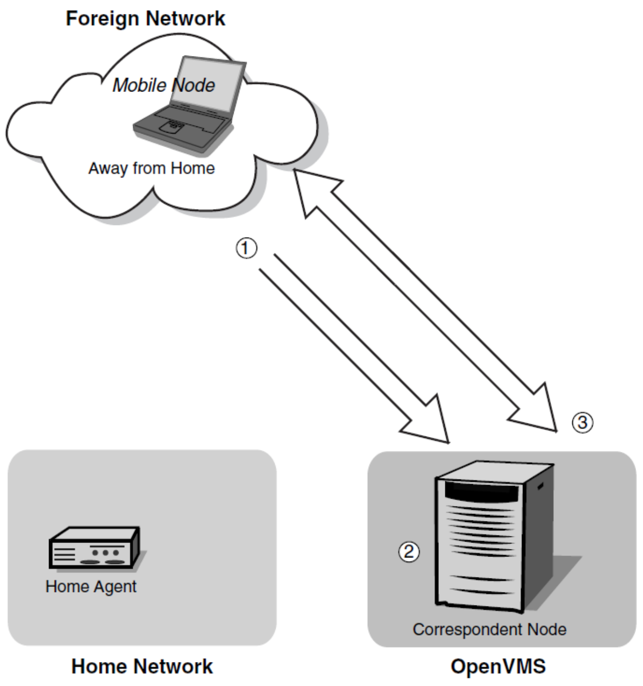
1. The mobile node configures a care-of address and registers it with its home agent by sending the home agent a binding update. This new address is the mobile node's primary care-of address.

   The home agent acknowledges the binding update by returning a binding acknowledgment to the mobile node.

2. Packets sent by a correspondent node to the mobile node's home address arrive at its home link.
3. The home agent intercepts the packets, encapsulates them, and tunnels them to the mobile node's registered care-of address.

In Figure 5.3, the mobile node has received the tunneled packets from the home agent.

Figure 5.3. Communication with Mobile Node Away from Home – Part 2



After the mobile node receives the tunneled packets, the following events occur:

1. The mobile node recognizes its primary care-of address in the tunneled packet's header. The mobile node assumes that the original sending correspondent node has no binding cache entry for the mobile node; otherwise, the correspondent node would have sent the packet directly to the mobile node using a routing header. It then sends a binding update to the correspondent node.
2. The correspondent node creates a binding between the home address and care-of address.
3. Packets flow directly between the correspondent node and mobile node. This route optimization does the following:
   - Eliminates what is commonly known as triangle routing.
   - Eliminates congestion at the mobile node's home agent and home link.
   - Reduces the impact of any possible failure of the home agent, the home link, or intervening networks leading to or from the home link, since these nodes and links are not involved in the delivery of most packets to the mobile node.

   When the mobile node is away from home, it always sends a home address option to inform the receiver of its home address. That way, the receiver can correctly identify the connection to which the packet belongs.

When the mobile node back on its home link, the mobile node sends a binding update to the home agent and to the correspondent node to clear the bindings.

5.4. Planning Mobile IPv6

This section describes tasks required before you configure Mobile IPv6.

Before you can use Mobile IPv6, you must configure your system as an IPv6 host node or a router. See Section 2.4 for more information.

You must verify that Mobile IPv6 support is enabled. You can verify this by issuing the following command:

```
$ sysconfig -q ipv6 mobileipv6_enabled
```

If the mobileipv6_enabled attribute is not set to 1, reconfigure it with the following command:

```
$ sysconfig -r ipv6 mobileipv6_enabled=1 mobileipv6_enabled: reconfigured
```

The system is now ready to function as a correspondent node. The correspondent node can also forward packets as a router. If you want your system to also function as a router, see Section 5.5.

5.5. Configuring Mobile IPv6

This section describes how to configure your IPv6 node both as a correspondent node and as a correspondent node that acts as an IPv6 router.

5.5.1. Configuring a Correspondent Node

After you verify that IPv6 mobile support is enabled, your system is ready to function as a correspondent node and to communicate with mobile nodes both through the home agent and, after the receiving a binding update from a mobile node, directly with the mobile node. No further configuration is necessary.

5.5.2. Configuring a Home Agent

Please see the *VSI TCP/IP Services for OpenVMS Release Notes* for the latest information on configuring a mobile node as a home agent.

5.6. Monitoring the Mobile IPv6 Environment

To monitor the Mobile IPv6 environment, use the following:

- `tcpdump` command
- `netstat` command
- TCPIP$IP6RTRD log file

5.6.1. Using tcpdump

The `tcpdump` command captures, parses, and prints IPv6 packets. The binding update and acknowledgment options are contained in IPv6 Destination Option headers in IPv6 packets.

To see IPv6 packets, issue the `tcpdump` command as follows:

```
$ tcpdump -s 1500 -x ipv6
```

See the *VSI TCP/IP Services for OpenVMS Tuning and Troubleshooting* manual for more information about using `tcpdump`.

5.6.2. Using netstat

The `netstat -b` command allows you to monitor current mobility bindings and their attributes. The following example shows the command output:

```
$ netstat -b

Mobile IPv6 Binding Cache

Home Address     Care-of Address    Flags     Refs  Sequence#   Lifetime
testhome         testcoa            A            1     1          43
  [1]               [2]             [3]          [4]        [5]        [6]
```

This example shows that:

1. The mobile node has a home address of `testhome`.
2. The mobile node is currently reachable at care-of address `testcoa`.
3. The mobile node has asked for the binding update to be acknowledged (A flag).
4. There is currently one reference on this binding data structure.
5. The sequence number is set to 1 in the binding update.
6. There are 43 seconds remaining on this binding's lifetime. When the lifetime expires, the entry is removed from the cache.

The `netstat -bs` command enables you to monitor mobility binding statistics. The following example shows the command output:

```
$ netstat -bs
Mobile IPv6:
       1 entry in binding cache
       1 add
       0 deletes
       0 changes
       0 frees
       3 lookups
```

5.6.3. TCPIP$IP6RTRD Log File

The TCPIP$IP6RTRD process logs informational and severity events in the SYS$MANAGER:TCPIP $IP6RTRD.LOG file.

Chapter 6. Solving IPv6 Problems

This chapter contains a diagnostic map to help you solve problems that might occur when you use an IPv6 network and network services. Use this chapter along with the appropriate documentation to solve problems that you encounter.

6.1. Using the Diagnostic Suggestions

IPv6 network and network service problems can occur for a number of reasons. This chapter should help you isolate the problem.

After you isolate the problem, you may be referred to other TCP/IP Services for OpenVMS documentation for more information about problem-solving tools and utilities.

If you use other products along with the IPv6 networking software described in this manual, you may need to consult the documentation associated with those products for additional information.

6.2. Getting Started

Before you start problem solving, ensure that communications hardware is ready for use. Verify the following:

- The system's physical connections are properly installed. See the documentation for your system and communications hardware device.
- Event logging is enabled to monitor network events. See the system administration manual for information about starting event logging and for descriptions of event messages.

Also check the product release notes for up-to-date information on known problems.

You should be familiar with the following terms:

- On-link node

  An on-link node is attached to the same subnetwork as your system. This subnetwork can be a LAN or an IPv6-over-IPv4 configured tunnel. There are no IPv6 routers between your system and the on-link node.

  For a configured tunnel, the on-link node is the node at the destination end of the tunnel.

- Off-link node

  An off-link node is not attached to the same subnetwork as your system. There is at least one IPv6 router between your system and the off-link node.

6.3. Solving IPv6 Network Problems

This section describes the most basic causes of IPv6 network problems. Before investigating further, make sure you perform the following checks:

1. Make sure the system is on and has completed all startup procedures.

Check the power to your system. See the system management manual for your system's startup procedure and any problem solving information.

2. Verify IPv6 installation.

   To verify that the IPv6 components are installed, enter the following command:

   ```
   $ TCPIP SHOW VER/ALL
   ```

   TCP/IP Services files should be listed. If the components are not listed, install TCP/IP Services for OpenVMS by using the PCSI command. See the *VSI TCP/IP Services for OpenVMS Installation and Configuration* manual for information about installing the product.

3. Verify IPv6 configuration.

   To verify that IPv6 is configured, enter the following command:

   ```
   $ DIR SYS$MANAGER:TCPIP$INET6_CONFIG.DAT
   ```

   See Section 2.4 for information about setting up and configuring an IPv6 host or router.

4. Verify that IPv6 is started.

   To verify that IPv6 is started, enter the following commands:

   ```
   $ SHOW LOGICAL TCPIP$IPV6_STARTED
   $ ping ::1
   ```

   If the "host is unreachable" message appears, enable IPv6 by entering the following command:

   ```
   $ @SYS$STARTUP:TCPIP$STARTUP
   ```

   This creates the IPv6 interfaces, brings them up, and starts the IPv6 processes. See Section 6.4 for a description of IPv6 host problems; see Section 6.5 for a description of IPv6 router problems.

6.4. Solving IPv6 Host Problems

This section describes possible problems with IPv6 hosts and procedures for solving them.

6.4.1. IPv6 Process Is Not Started

Verify that the TCPIP$ND6HOST process is running by issuing the following command:

```
$ SHOW SYSTEM /PROCESS=TCPIP$ND6HOST
```

If the process is not running, enable IPv6 with the following command:

```
$ @SYS$STARTUP:TCPIP$STARTUP.COM
```

This creates the IPv6 interfaces, brings them up, and starts the TCPIP$ND6HOST process.

6.4.2. Host Is Unknown

If a remote host is not known, the following message may appear in application log files:

```
unknown host
```

Perform the following steps:

1. Check whether the user is specifying a valid host name to reach the remote host.
2. Check whether the remote host is in another domain and whether the user specified the fully qualified domain name.
3. If the remote host is in a domain that you control and your site implements a BIND server, make sure the zone file contains an entry for the remote host. If you do not implement a BIND server, you can add the host to the local host database by editing the file TCPIP$ETC:TCPIP $IPNODES.DAT.
4. If the remote host does not reside in a domain under your control and you are using a BIND server to search the BIND database for name-to-address translation, make sure the resolver is pointing to a valid BIND server. See the *VSI TCP/IP Services for OpenVMS Management* guide for additional information about setting up your BIND environment.

6.4.3. On-Link Node Is Not Reachable

If an on-link node is not reachable, one of the following messages may appear in an application log file:

```
no route to host
network is unreachable
connection timed out
```

Verify that an on-link node or router (if one exists) is reachable by using the ping command. If the command fails or if packets are frequently dropped, perform the following steps:

1. If the node is attached to a LAN, check the data link counters by using the LANCP SHOW DEVICE *device* /COUNTERS command. Problems with the counters and their possible causes are as follows:
   - Zero blocks sent or received can indicate a network hardware failure or a wiring problem.
   - High collision rates can indicate an improperly wired network or a node that is sending excessive message traffic.
   - Data overrun and buffer unavailable errors indicate that your system is not configured properly.
2. If there is no problem with the data link counters, check the IPv6 and ICMPv6 counters with the `netstat -p ipv6` and `netstat -p ipv6-icmp` commands, respectively. Problems with counters and their possible causes are:
   - Packets discarded because of errors or errors resulting from ICMP errors indicate that another node is generating invalid messages. Other counters show more specific information.
   - Allocation errors can indicate excessive message traffic, an improperly configured system, or a program that repeatedly allocates memory without freeing it.
3. Using the `ifconfig -a` command, verify that IPv6 network interfaces exist, are up, and have inet6 addresses. If the interfaces do not have inet6 addresses, check the startup file TCPIP $INET6_CONFIG.DAT. Run the TCPIP$IP6_SETUP utility to correct any errors.

If your interface does not have a global or site-local address, contact your network administrator to verify that your local router is advertising a prefix on the link. If there is no local router, you can define a prefix by using the `ifconfig` command.

4. Contact the system manager for the adjacent on-link node. Verify that the on-link node is up and running, that it is configured correctly for IPv6, and that the address you are using is enabled on the node's interface.

5. If IPv4 is configured on both systems, issue the ping command to the on-link node's IPv4 address, If the commands succeeds, verify the IPv6 configuration on both systems. If the command fails, see the *VSI TCP/IP Services for OpenVMS Tuning and Troubleshooting* manual for diagnostic procedures.

6. Issue the ping command to other nodes on the link to determine whether the failure is confined to one node or extends to multiple nodes. Partial connectivity might indicate a faulty network device or cable on the link.

7. If the link is a configured tunnel, do the following:

   a. Verify the tunnel source and destination addresses by using the `ifconfig -a` command. Contact the administrator for the tunnel destination node and verify that your source and destination addresses match the destination and source addresses on that node.

      Issue the `ping` command to the tunnel destination address. If the command fails, see the *VSI TCP/IP Services for OpenVMS Tuning and Troubleshooting* guide for diagnostic procedures.

6.4.4. Off-Link Node Is Not Reachable

If an off-link node is not reachable, one of the following messages may appear in an application log files:

```
no route to host
network is unreachable
connection timed out
```

Verify that an off-link node is reachable by issuing the ping command.

If there is 100% packet loss, perform the following steps:

1. Verify connectivity between your system and an on-link router by using the ping command.

   If the command fails or shows frequently dropped packets, follow the steps in Section 6.4.3.

   If you do not know the address to a router, issue the following command:

   ```
   $ ping -"I" interface ff02::2
   ```

2. Verify that the interface over which you are sending messages has a global or site-local unicast address enabled by using the `ifconfig -a` command.

   If it does not, contact the router's administrator to verify that the router is advertising a prefix on the link.

   If the link is a configured tunnel and the router is not advertising an address prefix, manually define one for the tunnel by using the TCPIP$IP6_SETUP utility.

3. Contact the administrator for the remote system to verify that the system is up and running, that it is configured correctly for IPv6, and that the IPv6 address on its interface is the same as the address you are using.

   If the address is different, check your system's TCPIP$ETC:TCPIP$IPNODES.DAT file, or have the administrator for the remote system check the DNS entry.

4. Verify that there is a default route (with U and G flags set) to a router on the network by issuing the netstat -rf inet6 command. If there is no default route, contact the router administrator to check whether the router is advertising itself as a default router.

   Also, check other routers to see whether your messages are being directed on the wrong path.

5. Trace the path to the off-link node by using the `traceroute` command.

Frequently dropped packets might indicate either network congestion or an intermittent routing problem. To determine the cause, do the following:

1. Verify connectivity between your system and an on-link router by using the `ping` command.

2. Trace the path to the off-link node by using the `traceroute` command.

6.4.5. Your Node Is Unreachable

If someone reports a problem reaching your node from another node, perform the following steps:

1. Verify that their node is reachable by issuing the ping command.

   If the command fails, follow the steps in Section 6.4.3 for an on-link node or Section 6.4.4 for an off-link node.

2. If they are using a name from the DNS database, verify that the address for your node in the DNS database matches one of the addresses configured on your system's interfaces.

   Use the dig AAAA nodename command to retrieve the address from DNS and the `ifconfig -a` command to display addresses for your system.

3. If they are using an address defined in their local host file TCPIP$ETC:TCPIP$IPNODES.DAT, use the `ifconfig -a` command to compare that address with the addresses configured on your system's interfaces.

6.4.6. Connection Is Not Accepted

If a remote node is not configured to accept a connection from your application, the following message might appear in an application log file:

```
connection refused
```

Verify that TCP/IP Services has been correctly configured on the remote node to accept connections.

Contact the administrator for the remote node and ask whether the correct socket-based service definitions are defined in the TCPIP$SERVICES.DAT file. Check whether the service has IPv6 enabled.

6.4.7. Connection Terminates

If the connection terminates abnormally or a network application appears to hang, perform the following steps:

1. Verify that there is network connectivity to the remote node by using the ping command immediately after the failure.

   If the ping command fails or shows a high rate of packet loss, follow the steps in either Section 6.4.3 for on-link nodes, or in Section 6.4.4 for off-link nodes.

2. If your application transfers a large amount of data over the network, verify whether large or fragmented messages are being handled correctly by using the `ping -s 2000 nodename` command.

   If the ping command fails, trace the path to the remote node with 1200-byte packets by using the `traceroute nodename 1200` command. All IPv6 links should support message sizes of at least 1280 bytes. This command might show the location of the problem in the network.

3. Run the application with different client and server nodes located on different links in the network.

6.5. Solving IPv6 Router Problems

This section describes problems with IPv6 routers.

6.5.1. IPv6 Process Is Not Running

Verify that the TCPIP$IP6RTRD process is running by issuing the following command:

```
$ SHOW SYSTEM /PROCESS=TCPIP$IP6RTRD
```

If the process is not running, start IPv6 with the following command:

```
$ @SYS$STARTUP:TCPIP$STARTUP.COM
```

This creates the IPv6 interfaces, brings them up, and starts the TCPIP$IP6RTRD process.

6.5.2. Host Is Unknown

If a remote host is not known, the following message may appear in an application log file:

```
unknown host
```

If you receive this message, perform these steps:

1. Check whether the user is specifying a valid host name to reach the remote host.

2. Check whether the remote host is in another domain and whether the user specified the fully qualified domain name.

3. If the remote host is in a domain that you control and if your site implements a BIND server, make sure the zone file contains an entry for the remote host. If you do not implement a BIND server, you can add the host to the local host database by editing the file TCPIP$ETC:TCPIP $IPNODES.DAT.

4. If the remote host does not reside in a domain under your control and you are using a BIND server to search the BIND database for name-to-address translation, make sure the resolver is pointing to a valid BIND server. See the *VSI TCP/IP Services for OpenVMS Management* guide for additional information about setting up your BIND environment.

6.5.3. On-Link Node Is Unreachable

If an on-link node is not reachable, one of the following messages may appear in an application log file:

```
no route to host
network is unreachable
connection timed out
```

Verify that an on-link node or router is reachable by using the ping command. If the command fails or if packets are frequently dropped, complete the following steps:

1. If the node is attached to a LAN, check the data link counters by using the LANCP SHOW DEVICE *device* /COUNTERS command. Problems with the counters and their possible causes are as follows:
   - Zero blocks sent or received can indicate a network hardware failure or a wiring problem.
   - High collision rates can indicate an improperly wired network or a node that is sending excessive message traffic.
   - Data overrun and buffer unavailable errors indicate your system is not configured properly.
2. If the data link counters are okay, check the IPv6 and ICMPv6 counters with the `netstat -p ipv6` and `netstat -p ipv6-icmp` commands, respectively. Problems with the counters and their possible causes are as follows:
   - Packets discarded because of errors, or errors resulting from ICMP errors, indicate that another node is generating invalid messages. Other counters show more specific information.
   - Allocation errors can indicate excessive message traffic, an improperly configured system, or a program that repeatedly allocates memory without freeing it.
3. Verify that IPv6 network interfaces exist, are up, and have `inet6` addresses by using the `ifconfig -a` command. If they do not have `inet6` addresses, check the configuration file TCPIP$INET6_CONFIG.DAT. Run the TCPIP$IP6_SETUP utility to correct any errors.
4. Contact the system administrator for the adjacent on-link node and verify that the on-link node is up and running, that it is configured correctly for IPv6, and that the address you are using is enabled on the node's interface.
5. If IPv4 is configured on both systems, issue the `ping` command to the on-link node's IPv4 address. If the command succeeds, verify the IPv6 configuration on both systems. If the command fails, see the *VSI TCP/IP Services for OpenVMS Tuning and Troubleshooting* manual.
6. Issue the `ping` command to other nodes on the link to determine whether the failure is confined to one node or whether it extends to multiple nodes. Partial connectivity might indicate a faulty network device or cable on the link.
7. If the link is a configured tunnel, do the following:

a. Verify the tunnel source and destination addresses by using the `ifconfig -a` command. Contact the administrator for the tunnel destination node and verify that your source and destination addresses match the destination and source addresses on that node.

b. Issue the `ping` command to the tunnel destination address. If the command fails, see the *VSI TCP/IP Services for OpenVMS Tuning and Troubleshooting* guide for diagnostic procedures.

6.5.4. Off-Link Node Is Unreachable

If an off-link node is not reachable, the following messages may appear in an application log file:

```
no route to host
network is unreachable
connection timed out
```

Verify that an off-link node is reachable by issuing the ping command.

If there is 100% packet loss, perform the following steps:

1. Verify connectivity between your system and an on-link router by using the ping command.

   If the command fails or shows frequently dropped packets, follow the steps in Section 6.5.3.

2. Verify that the interface over which you are sending messages has a global or site-local unicast address enabled by using the `ifconfig -a` command.

   If it does not, check the prefixes defined in the SYS$SYSTEM:TCPIP$IP6RTRD.CONF file. Run the TCPIP$IP6_SETUP utility to correct any errors.

3. Contact the administrator for the remote system to verify that the system is up and running, that it is configured correctly for IPv6, and that the IPv6 address on its interface is the same as the address you are using.

   If the address is different, check your system's TCPIP$ETC:TCPIP$IPNODES.DAT file, or have the remote system administrator check the DNS entry.

4. Verify that there is a default route (with U and G flags set) to a router on the network by issuing the `netstat -rf inet6` command.

   If the route is missing or incorrect, check the routes and the address prefixes in the SYS $SYSTEM:TCPIP$IP6RTRD.CONF file.

   If your site uses RIPng, verify that RIP is enabled in the SYS$SYSTEM:TCPIP$IP6RTRD.CONF file. If it is, contact the administrator of the next router to verify that RIP is enabled.

5. Trace the path to the off-link node by using the `traceroute` command.

Frequently dropped packets indicate either network congestion or an intermittent routing problem.

To determine the cause, do the following:

1. Verify connectivity between your system and an on-link router by using the `ping` command.

2. Trace the path to the off-link node by using the `traceroute` command.

6.5.5. On-Link Node Addresses Are Not Configured

IPv6 hosts generate their global and site-local unicast addresses automatically by using address prefixes provided by a router on the link. If an on-link node cannot autoconfigure its addresses, perform the following steps:

1. Verify that the host is reachable from your router by using the `ping` command and specifying the host's link-local address. If the command fails or shows a high rate of packet loss, follow the steps in Section 6.5.3.

2. Edit the SYS$SYSTEM:TCPIP$IP6RTRD.CONF file and verify that the router is configured to advertise the correct prefixes and that the timers are reasonable. See Section 2.6.2.3 for more information.

6.5.6. Router Does Not Forward Messages

If another network user reports that message transmission appears to be failing at your router, perform the following steps:

1. Obtain the source and destination addresses of the message that your router is not forwarding. Then verify that your router can reach each node by using the ping command.

   If the command fails or shows a high rate of packet loss, follow the steps in Section 6.5.3 for on-link nodes, or in Section 6.5.4 for off-link nodes.

2. If your router is running the RIPng protocol, verify that the IPv6 router process is running by issuing the following command:

   ```
   $ SHOW SYSTEM /PROCESS=TCPIP$IP6RTRD
   ```

   If the process is running, edit the SYS$SYSTEM:TCPIP$IP6RTRD.CONF file and verify that the RIPng protocol is enabled on each IPv6 link. If it is not, your node may not be propagating routes correctly.

3. Make sure that you are not using manual routes on some interfaces and RIPng routes on other interfaces. Manual routes defined in the TCPIP$ROUTE.DAT file do not get propagated to other routers with RIPng.

6.5.7. Your Node Is Unreachable

If someone reports a problem reaching your node from another node, perform the following steps:

1. Verify that their node is reachable by issuing the ping command.

   If the command fails, follow the steps in Section 6.5.3 for an on-link node, or Section 6.5.4 for an off-link nodes.

2. If they are using a name from the DNS database, verify that the address for your node in the DNS database matches one of the addresses configured on your system's interfaces.

   Use the dig AAAA nodename command to retrieve the address from DNS; use the `ifconfig -a` command to display addresses for your system.

3. If they are using an address defined in their local host file, compare that address with the addresses configured on your system's interfaces by using the `ifconfig -a` command.

6.5.8. Connection Is Not Accepted

If a remote node is not configured to accept a connection from your application, the following message might appear in an application log file:

```
connection refused
```

Verify that TCP/IP Services has been correctly configured on the remote node to accept connections.

Contact the administrator for the remote node and ask whether the correct socket-based service definitions are defined in the TCPIP$SERVICES.DAT file. Check whether the service has IPv6 enabled.

6.5.9. Connection Terminates

If the connection terminates abnormally or if a network application appears to hang, perform the following steps:

1. Verify that there is network connectivity to the remote node by using the ping command immediately after the failure.

   If the ping command fails or shows a high rate of packet loss, follow the steps in Section 6.5.3 for an on-link node, or in Section 6.5.4 for an off-link node.

2. If your application transfers a large amount of data over the network, verify that large or fragmented messages are being handled correctly by using the `ping -s 2000` nodename command.

   If the ping command fails, trace the path to the remote node with 1200-byte packets by using the `traceroute nodename 1200` command. All IPv6 links should support message sizes of at least 1280 bytes. This command might show the location of the problem in the network.

3. Run the application with different client and server nodes located on different links in the network.

Chapter 7. Application Interface to Sockets

The TCP/IP Services for OpenVMS programming interface supports the Berkeley Software Distribution (BSD) socket programming interface. It also supports the basic sockets interface extensions for Internet Protocol Version 6 (IPv6) as defined in RFC 3493 and the advanced sockets application programming interfaces as defined in draft RFC 3542. The basic syntax of socket functions remains the same. Existing IPv4 applications will continue to operate as before, and IPv6 applications can interoperate with IPv4 applications.

To support IPv6, TCP/IP Services for OpenVMS provides the following:

- An address family AF_INET6, used by the `socket` function.
- A protocol level IPPROTO_IPV6, used by the `setsockopt` and `getsockopt` functions
- Socket options used by the `getsockopt` and `setsockopt` functions
- Header files that define new structures, constants, and variables.
- Library functions

This chapter describes the following aspects of the IPv6 API:

- Structures
- Header files
- Socket options
- Basic interface functionality
- Advanced interface functionality
- Guidelines for compiling and linking
- IPv6 library functions

7.1. Structures

The following structures support IPv6. The header file containing each structure is listed. Consult the appropriate header file for details on the definitions of each structure.

7.1.1. in6_addr Structure

The `in6_addr`, defined in the IN6.H header file, supports IPv6.

The address is stored in network byte order as an array of sixteen 8-bit elements.

A wildcard address, defined in network byte order, has the following forms:

- A global variable, `in6addr_any`, that is an `in6_addr` structure.

- A symbolic constant, IN6ADDR_ANY_INIT, that can be used to initialize an `in6_addr` structure only when it is declared.

A loopback address, defined in network byte order, has the following forms:

- A global variable, `in6addr_loopback`, that is an `in6_addr` structure.
- A symbolic constant, IN6ADDR_LOOPBACK_INIT, that can be used to initialize an `in6_addr` structure only when it is declared.

7.1.2. sockaddr_in6 Structure

The `sockaddr_in6` structure, defined in IN6.H header file, is the protocol-specific address data structure for IPv6.

7.1.3. msghdr Structure

This data structure enables applications to send and receive ancillary data using the sendmsg and recvmsg functions. This data structure, which is defined in the SOCKET.H header file, also allows AF_INET sockets and raw AF_INET6 sockets to receive certain data.

For IPv4, see the *VSI TCP/IP Services for OpenVMS Sockets API and System Services Programming* manual for the descriptions of the IP_RECVDSTADDR and IP_RECVOPTS socket options.

For IPv6, Section 7.3 describes the IPV6_RECVHOPOPTS , IPV6_RECVDSTOPTS , and IPV6_RECVRTHDR options.

7.1.4. cmsghdr Structure

The `cmsghdr` structure describes ancillary data objects transferred by the `sendmsg` and `recvmsg` functions.

The `msg_control` member of the `msghdr` data structure points to the ancillary data that are contained in a `cmsghdr` structure. Typically, only one data object is passed in a `cmsghdr` structure. However, the IPv6 advanced sockets API enables the `sendmsg` and `recvmsg` functions to pass multiple objects. See Section 7.5.1 for information about using raw IPv6 sockets.

The data structure is defined in the SOCKET.H header file.

7.2. Header Files

Header files are provided by the C Run-Time Library (DECC$RTLDEF.TLB). Updated header files may also appear in the TCPIP$EXAMPLES: directory. If you use the /INCLUDE=TCPIP $EXAMPLES: qualifier in your compile command, the header files in TCPIP$EXAMPLES: supersede those in DECC$RTLDEF.TLB.

7.3. Socket Options

The following socket options supporting IPv6 supplement those listed in the *VSI TCP/IP Services for OpenVMS Sockets API and System Services Programming* manual. The IPv6 socket options do not have system service symbols.

Table 7.1. Socket Options

Sockets API Symbol	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 `getsockopt` function and `setsockopt` function.
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.

7.4. Basic API

The basic IPv6 API focuses on interoperability between IPv4 and IPv6 applications. The API provides functions to identify interfaces, socket options to support IPv6 multicast datagrams, and functions to translate and convert addresses.

7.4.1. Interface Identification

To identify the interface on which a datagram is received, on which a datagram is to be sent, and on which a multicast group is joined, the API uses a small, positive integer called an **interface index**. The kernel assigns this integer to an interface when the interface is initialized.

The API defines the following new functions:

Function	Description
if_nametoindex()	Maps an interface name to its corresponding index.
if_indextoname()	Maps an interface index to its corresponding name.
if_nameindex()	Returns an array of all interface names and indexes.
if_freenameindex()	Frees dynamic memory allocated by `if_nameindex()` to the array of interface names and indexes.

7.4.2. IPv6 Multicast Datagrams

7.4.2.1. Sending IPv6 Multicast Datagrams

To send IPv6 multicast datagrams, an application indicates the multicast group to send to by specifying an IPv6 multicast address in a `sendto` function. The system maps the specified IPv6 destination address to the appropriate Ethernet or FDDI multicast address prior to transmitting the datagram.

An application can explicitly control multicast options with arguments to the `setsockopt` function. The following options can be set by an application using the `setsockopt` function:

- Hop limit (IPV6_MULTICAST_HOPS)
- Multicast interface (IPV6_MULTICAST_IF)
- Disabling loopback of local delivery (IPV6_MULTICAST_LOOP)

Note

The examples here illustrate how to use the `setsockopt` function options that apply to IPv6 multicast datagrams only.

The IPV6_MULTICAST_HOPS option to the `setsockopt` function allows an application to specify a value between 0 and 255 for the hop limit field.

Multicast datagrams with a hop limit value of 0 restrict distribution of the multicast datagram to applications running on the local host. Multicast datagrams with a hop limit value of 1 are forwarded only to hosts on the local link. If a multicast datagram has a hop limit value greater than 1 and a multicast router is attached to the sending host's network, multicast datagrams can be forwarded beyond the local link. Multicast routers forward the datagram to known networks that have hosts belonging to the specified multicast group. The hop limit value is decremented by each multicast router in the path. When the hop limit value is decremented to 0, the datagram is not forwarded further.

The following example shows how to use the IPV6_MULTICAST_HOPS option to the `setsockopt` function:

```
u_char hops;
hops=2;

if (setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_HOPS, &hops,
                     sizeof(hops)) < 0)
                     perror("setsockopt: IPV6_MULTICAST_HOPS error");
```

A datagram addressed to an IPv6 multicast address is transmitted from the default network interface unless the application specifies that an alternate network interface is associated with the socket. The default interface is determined by the interface associated with the default route in the kernel routing table or by the interface associated with an explicit route, if one exists. Using the IPV6_MULTICAST_IF option to the `setsockopt` function, an application can specify a network interface other than that specified by the route in the kernel routing table.

The following example shows how to use the IPV6_MULTICAST_IF option to the `setsockopt` function to specify an interface other than the default:

```
u_int if_index = 1;
.
.
.
if (setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_IF, &if_index,
                              sizeof(if_index)) < 0)
     perror ("setsockopt: IPV6_MULTICAST_IF error");
  else
     printf ("new interface set for sending multicast datagrams\n");
```

The **if_index** parameter specifies the interface index of the desired interface, or specifies 0 to select a default interface. You can use the `if_nametoindex()` function to find the interface index.

If a multicast datagram is sent to a group that has the sending node as a member, a
copy of the datagram is, by default, looped back by the IP layer for local delivery. The IPV6_MULTICAST_LOOP option to the `setsockopt()` function allows an application to disable this loopback delivery.

The following example shows how to use the IPV6_MULTICAST_LOOP option to the `setsockopt()` function:

```
u_char loop=0;
if (setsockopt( sock, IPPROTO_IPV6, IPV6_MULTICAST_LOOP, &loop,
         sizeof(loop)) < 0)
         perror("setsockopt: IPV6_MULTICAST_LOOP error");
```

If the value of **loop** is 0, loopback is disabled; if the value of **loop** is 1, loopback is enabled. For performance reasons, you should disable the default by setting loop to 0, unless applications on the same host must receive copies of the datagrams.

7.4.2.2. Receiving IPv6 Multicast Datagrams

Before a node can receive IPv6 multicast datagrams destined for a particular multicast group other than the All Nodes group, an application must direct the node to become a member of that multicast group.

This section describes how an application can direct a node to add itself to and remove itself from a multicast group.

An application can direct the node it is running on to join a multicast group by using the IPV6_JOIN_GROUP option to the `setsockopt()` function:

```
struct ipv6_mreq imr6;
.
.
.
imr6.ipv6mr_interface = if_index;
if (setsockopt( sock, IPPROTO_IPV6, IPV6_JOIN_GROUP,
     (char *)&imr6, sizeof(imr6)) < 0)
     perror("setsockopt: IPV6_JOIN_GROUP error");
```

The **imr6** parameter has the following structure:

```
structipv6_mreq {
        struct in6_addr ipv6mr_multiaddr; /* IP multicast address of
group */
        unsigned int ipv6mr_interface; /* local interface index*/
        };
```

Each multicast group membership is associated with a particular interface. It is possible to join the same group on multiple interfaces. The *ipv6mr_interface* variable can be specified with a value of 0, which allows an application to choose the default multicast interface. Alternatively, specifying one of the host's local interfaces allows an application to select a particular multicast-capable interface. The maximum number of memberships that can be added on a single socket is subject to the IPV6_MAX_MEMBERSHIPS value, which is defined in the `<in.h>` header file.

To drop membership from a particular multicast group, use the IPV6_LEAVE_GROUP option to the `setsockopt` function:

```
struct ipv6_mreq imr6;
if (setsockopt( sock, IPPROTO_IPV6, IPV6_LEAVE_GROUP, &imr6,
     sizeof(imr6)) < 0)
     perror("setsockopt: IPV6_LEAVE_GROUP error");
```

The **imr6** parameter contains the same structure values used for adding membership.

If multiple sockets request that a node join a particular multicast group, the node remains a member of that multicast group until the last of those sockets is closed.

To receive multicast datagrams sent to a specific UDP port, the receiving socket must have bound to that port using the bind system call. More than one process can receive UDP datagrams destined for the same port if the bind function is preceded by a `setsockopt` function that specifies the SO_REUSEPORT option.

Delivery of IP multicast datagrams to SOCK_RAW sockets is determined by the protocol type of the destination.

7.4.3. Address Translation and Conversion Functions

The following functions are available for node name to address translation:

Function	**Description**
`gethostbyname()`	Returns IPv4 addresses.
`getaddrinfo()`	Protocol-independent function for mapping names to addresses.
`freeaddrinfo()`	Returns `addrinfo()` structures and dynamic storage to the system.

The following functions are available for address to node name translation:

Function	**Description**
`gethostbyaddr()`	Returns a node name for an IPv4 address.
`getnameinfo()`	Protocol-independent function for mapping addresses to names.
`freeaddrinfo()`	Returns `addrinfo()` structures and dynamic storage to the system.

The following address conversion functions convert both IPv4 and IPv6 addresses.

Function	Description
`inet_pton()`	Converts an address in its standard text presentation form to its numeric binary form, in network byte order.
`inet_ntop()`	Converts a numeric address to a text string suitable for presentation.

7.4.4. Address-Testing Macros

Table 7.2 lists the currently defined address-testing macros and the return value for a valid test. To use these macros, include the following file in your application:

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

Table 7.2. Summary of Address-Testing Macros

Macro	Return
IN6_IS_ADDR_UNSPECIFIED	True, if specified type.
IN6_IS_ADDR_LOOPBACK	True, if specified type.
IN6_IS_ADDR_MULTICAST	True, if specified type.
IN6_IS_ADDR_LINKLOCAL	True, if specified type.
IN6_IS_ADDR_SITELOCAL	True, if specified type.
IN6_IS_ADDR_V4MAPPED	True, if specified type.
IN6_IS_ADDR_V4COMPAT	True, if specified type.
IN6_IS_ADDR_MC_NODELOCAL	True, if specified scope.
IN6_IS_ADDR_MC_LINKLOCAL	True, if specified scope.
IN6_IS_ADDR_MC_SITELOCAL	True, if specified scope.
IN6_IS_ADDR_MC_ORGLOCAL	True, if specified scope.
IN6_IS_ADDR_MC_GLOBAL	True, if specified scope.
IN6_ARE_ADDR_EQUAL	True, if addresses are equal.

7.5. Advanced API

The advanced API provides support for advanced applications that may need knowledge of IPv6 headers. These applications commonly use raw sockets to access IPv6 or ICMPv6 header fields. The advanced interface provides the following:

- Support for portable interfaces for applications that use raw sockets under IPv6
- Functions to access router headers
- Functions to access option headers

7.5.1. Using IPv6 Raw Sockets

Raw sockets are used in both IPv4 and IPv6 to bypass the TCP and UDP transport layers.

Table 7.3 describes the principal differences between IPv4 and IPv6 raw sockets.

Table 7.3. Differences Between IPv4 and IPv6 Raw Sockets

	IPv4	**IPv6**
Use	Access ICMPv4, IGMPv4, and to read and write IPv4 datagrams that contain a protocol field the kernel does not recognize.	Access ICMPv6 and to read and write IPv6 datagrams that contain a Next Header field the kernel does not recognize.
Byte order	Not specified.	Network byte order for all data sent and received.
Send and receive complete packets	Yes	No. Uses ancillary data objects to transfer extension headers and hop limit information.

For output, applications can modify all fields, except for the flow label field, by using ancillary data or socket options, or both.

For input, applications can access all fields, except for the flow label, version number, and Next Header fields, and all extension headers by using ancillary data.

For IPv6 raw sockets other than ICMPv6 raw sockets, the application must set the `IPV6_CHECKSUM` socket option. For example:

```
int offset = 2;
setsockopt (fd, IPPROTO_IPV6, IPV6_CHECKSUM, &offset, sizeof(offset));
```

This enables the kernel to compute and store a checksum for output and to verify the checksum on input. This relieves the application from having to perform source address selection on all outgoing packets. This socket option is disabled by default. You can explicitly disable this option by setting the offset variable to -1.

Using IPv6 raw sockets, an application can access the following information:

- ICMPv6 messages
- IPv6 header
- Routing header
- IPv6 options headers: hop-by-hop options header and destination options header

The following sections describe how to access this information.

7.5.1.1. Accessing ICMPv6 Messages

An ICMPv6 raw socket is a `socket` that is created by calling the socket function with the `PF_INET6, SOCK_RAW`, and `IPPROTO_ICMPV6` arguments.

The kernel calculates and inserts the ICMPv6 checksum for all outbound ICMPv6 packets and verifies the checksum for all received packets. If the received checksum is incorrect, the packet is discarded.

Because ICMPv6 is a superset of ICMPv4, an ICMPv6 raw socket can receive many more messages than an ICMPv4 raw socket. By default, when you create an ICMPv6 raw socket, it passes all ICMPv6 message types to an application. An application, however, does not need access to all messages. An application can specify the ICMPv6 message types it wants passed by creating an ICMPv6 filter.

The ICMPv6 filter has a datatype of `struct icmp6_filter`. Use `getsockopt()` to retrieve the current filter and `setsockopt()` to store the filter. For example, to enable filtering of ICMPv6 messages, use the `ICMP6_FILTER` option, as follows:

```
struct icmp6_filter myfilter;

setsockopt (fd, IPPROTO_ICMPV6, IPV6_FILTER, &(myfilter), (sizeof)
(myfilter));
```

The value of *myfilter* is an ICMPv6 message type between 0 and 255.

Table 7.4 describes the ICMPv6 filter macros.

Table 7.4. ICMPv6 Filtering Macros

Macro	Description
ICMP6_FILTER_SETPASSALL	Passes all ICMPv6 messages to an application.
ICMP6_FILTER_SETBLOCKALL	Blocks all ICMPv6 messages from being passed to an application.
ICMP6_FILTER_SETPASS	Passes ICMPv6 messages of a given type to an application.
ICMP6_FILTER_SETBLOCK	Blocks ICMPv6 messages of a given type from being passed to an application.
ICMP6_FILTER_WILLPASS	Returns true, if specified message type is passed to application.
ICMP6_FILTER_WILLBLOCK	True, if the specified message type is blocked from being passed to an application.

To clear an installed filter, call `setsockopt()` for the `ICMP_FILTER` option with a zero-length filter.

The kernel does not perform any validity checks on message type, message content, or packet structure. The application is responsible for checking them.

7.5.1.2. Accessing the IPv6 Header

When using IPv6 raw sockets, applications must be able to receive the IPv6 header content. To receive this optional information, use the `setsockopt()` function with the appropriate socket option.

Table 7.5 describes the socket options for receiving optional information.

Table 7.5. Optional Information and Socket Options

Optional Information	Socket Option	cmsg_type
Source and destination IPv6 address, and sending and receiving interface	IPV6_RECVPKTINFO	IPV6_PKTINFO
Hop limit	IPV6_RECVHOPLIMIT	IPV6_HOPLIMIT
Routing header	IPV6_RECVRTHDR	IPV6_RTHDR
Hop-by-Hop options	IPV6_RECVHOPOPTS	IPV6_HOPOPTS
Destination options	IPV6_RECVDSTOPTS	IPV6_DSTOPTS

The `recvmsg()` function returns the received data as one or more ancillary data objects in a `cmsghdr` data structure.

To determine the value of a socket option, use the `getsockopt()` function with the corresponding option. If the `IPV6_RECVPKTINFO` option is not set, the function returns an in6_pktinfo data structure with `ipi6_addr` set to `in6addr_any` and `ipi6_addr` set to zero. For other options, the function returns an `option_len` value of zero if there is no option value.

An application can receive the following IPv6 header information as ancillary data from incoming packets:

- Destination IPv6 address
- Interface index
- Hop limit

The IPv6 address and interface index are contained in a `in6_pktinfo` data structure that is received as ancillary data with the `recvmsg()` function. the `in6_pktinfo` data structure is defined in `in.h`. The tasks associated with the IPv6 header are:

- Receiving an IPv6 address

  If the `IPV6_RECVPKTINFO` option is enabled, the `recvmsg()` function returns a `in6_pktinfo` data structure as ancillary data. The `ipi6_addr` member contains the destination IPv6 address from the received packet. For TCP sockets, the destination address is the local address of the connection.

- Receiving an interface

  If the IPV6_RECVPKTINFO option is enabled, the `recvmsg()` function returns a `in6_pktinfo` data structure as ancillary data. The `ipi6_ifindex` member contains the interface index of the interface that received the packet.

- Receiving a hop limit

  If the IPV6_RECVHOPLIMIT option is enabled, the `recvmsg()` function returns a `cmsghdr` data structure as ancillary data. The `cmsg_type` member is `IPV6_HOPLIMIT` and the `cmsg_data[]` member contains the first byte of the integer hop limit.

7.5.1.3. Accessing the IPv6 Routing Header

The advanced sockets API enables you to access the IPv6 routing header. The routing header is an IPv6 extension header that enables an application to perform source routing. RFC 2460 defines the type 0 routing header, which supports up to 127 intermediate nodes, or 128 hops.

Table 7.6 describes the sockets calls that an application uses to build and examine routing headers.

Table 7.6. Socket Calls for Routing Header Name Description

Function	Description
`inet6_rth_space()`	Returns the number of bytes required for a routing header.
`inet6_rth_init()`	Initializes buffer data for a routing header.
`inet6_rth_add()`	Adds one address to a routing header.
`inet6_rth_reverse()`	Reverses the order of fields in a routing header.

Function	Description
`inet6_rth_segments()`	Returns the number of segments, or addresses, in a routing header.
`inet6_rth_getaddr()`	Fetches one address from a routing header.

The tasks associated with the routing header are:

- Receiving a routing header

  To receive a routing header, an application calls `setsockopt()` with the `IPV6_RECVRTHDR` option enabled.

  For each received routing header, the kernel passes one ancillary data object in a `cmsghdr` structure with the `cmsg_type` member set to `IPV6_RTHDR`. An application processes a routing header by calling `inet6_rth_reverse()`, `inet6_rth_segments()`, and `inet6_rth_getaddr()`.

- Sending a routing header

  To send a routing header, an application specifies the header either as ancillary data in a call to `sendmsg()` or by calling `setsockopt()`. An application can remove a sticky routing header by calling `setsockopt()` for the `IPV6_RTHDR` option and specifying a option length of zero.

  When using ancillary data, the application sets `cmsg_level` member to `IPPROTO_IPV6` and the `cmsg_type` member to `IPV6_RTHDR`. Use the `inet6_rth_space()`, `inet6_rth_init()`, and `inet6_rth_add()` calls to build the routing header.

  When an application specifies a routing header, the destination address specified in a call to the `connect()`, `sendto()`, or `sendmsg()` function is the final destination of the datagram. Therefore, the routing header contains the addresses of all intermediate nodes.

  Because of the order of extension headers specified in RFC 2460, an application can specify only one outgoing routing header.

7.5.1.4. Accessing the IPv6 Options Headers

The advanced sockets API enables applications to access the following IPv6 options headers:

- Hop-by-hop header

  A single hop-by-hop options header can contain a variable number of hop-by-hop options. Each option is encoded with a type, length, and value (TLV). The application uses sticky options or ancillary data to communicate this information with the kernel.

- Destination header

  One or more destination options headers can contain a variable number of destination options. A destination options header appearing before a routing header is processed by the first and subsequent destinations specified in the routing header. A destination option appearing after the routing header is processed only by the final destination. Each option is encoded with a type, length, and value (TLV). The application uses sticky options or ancillary data to communicate this information with the kernel.

See RFC 2460 for additional information about the alignment requirements of the headers and ordering of the extensions headers.

Table 7.7 lists the sockets calls that an application uses to build and examine hop-by-hop and destination headers.

Table 7.7. Socket Calls for Options Headers

Function	**Description**
`inet6_opt_init()`	Initializes buffer data for options.
`inet6_opt_append()`	Adds an option to the options header.
`inet6_opt_finish()`	Finishes adding options to the options header.
`inet6_opt_set_val()`	Adds one component of the option content to the options header.
`inet6_opt_next()`	Extracts the next option from the options header.
`inet6_opt_find()`	Extracts an option of a specified type from the options header.
`inet6_opt_get_val()`	Retrieves one component of the option content from the options header.

The tasks associate with options headers are:

- Receiving hop-by-hop options

  To receive a hop-by-hop options header, an application calls `setsockopt()` with the `IPV6_RECVHOPOPTS` option enabled.

  When using ancillary data, the kernel passes a hop-by-hop options header to the application and sets the `cmsg_level` member to `IPPROTO_IPV6` and the `cmsg_type` member to `IPV6_HOPOPTS`.

  An application retrieves these options by calling `inet6_opt_next()`, `inet6_opt_find()`, and `inet6_opt_get_val()`.

- Sending hop-by-hop options

  To send a hop-by-hop options header, an application specifies the header either as ancillary data in a call to `sendmsg()` or by calling `setsockopt()`. An application can remove a sticky hop-by-hop options header by calling `setsockopt()` for the `IPV6_HOPOPTS` option and specifying a option length of zero (0).

  When using ancillary data, all hop-by-hop options are specified by a single ancillary data object. The application sets `cmsg_level` member to `IPPROTO_IPV6` and the `cmsg_type` member to `IPV6_HOPOPTS`. Use the `inet6_opt_init()`, `inet6_opt_append()`, `inet6_opt_finish()`, and `inet6_opt_set_val()` calls to build the option header.

- Receiving destination options

  To receive a destination options header, an application calls `setsockopt()` with the `IPV6_RECVDSTOPTS` option enabled. The kernel passes each destination option to the application as one ancillary data object and sets the `cmsg_level` member to `IPPROTO_IPV6` and the `cmsg_type` member to `IPV6_DSTOPTS`.

  An application processes these options by calling `inet6_opt_next()`, `inet6_opt_find()`, and `inet6_opt_get_val()`.

- Sending destination options

  To send a destination options header, an application specifies the header either as ancillary data in a call to `sendmsg()` or by calling `setsockopt()`.

  An application can remove a sticky hop-by-hop options header by calling `setsockopt()` for either the `IPV6_RTHDRDSTOPTS` or the `IPV6_DSTOPTS` option and specifying a option length of zero (0).

  In accordance with RFC 2460, the API assumes that the extension headers are in order. Only one set of destination options can precede a routing header and only one set of destination options can follow a routing header.

  Each set can contain one or more options, but each set is considered a single extension header.

  When using ancillary data, the application passes a destination options header to the kernel in one of the following ways:

  - For destination options that precede a routing header, the application sets the `cmsg_level` member to `IPPROTO_IPV6` and the `cmsg_type` member to `IPV6_RTHDRDSTOPTS`. Any `setsockopt()` or ancillary data is ignored unless the application explicitly specifies its own routing header.

  - For destination options that follow a routing header or when no routing header is specified, the application sets the `cmsg_level` member to `IPPROTO_IPV6` and the `cmsg_type` member to `IPV6_DSTOPTS`.

  An application builds these options by calling `inet6_opt_init()`, `inet6_opt_append()`, `inet6_opt_finish()`, and `inet6_opt_set_val()`.

7.6. Guidelines for Compiling and Linking IPv6 Applications

To compile an IPv6 application that contains #includes of the following form

```
#include <path/file.h>
```

that is, an include file specification preceded by `"path/"`, you need to set up the following environment:

```
$ DEFINE DECC$SYSTEM_INCLUDE TCPIP$EXAMPLES:
$ DEFINE ARPA TCPIP$EXAMPLES:
$ DEFINE NET TCPIP$EXAMPLES:
$ DEFINE NETINET TCPIP$EXAMPLES:
$ DEFINE SYS TCPIP$EXAMPLES:
```

If your IPv6 application does not contain any include statements of that format, the DEFINE statements are not necessary.

To use certain features of the basic and advanced APIs, you must take special measures when compiling and linking.

Under either of these conditions:

- Using `getaddrinfo()` with OpenVMS 7.3-1 or earlier

- Using any of the advanced APIs with any version of OpenVMS

You should do both of the following:

- Add /INCLUDE_DIRECTORY=TCPIP$EXAMPLES: to the compile command line. This allows the compiler to use the updated header files that exist in the TCPIP$EXAMPLES directory which are provided by TCP/IP Services for OpenVMS. Otherwise, the compiler will use the header files from the C Run-Time Library, which may not contain the information needed or which may contain out-of-date information.
- Add TCPIP$LIBRARY:TCPIP$LIB/LIBRARY to the link command line. This allows the linker to resolve references to routines from TCPIP$LIB.OLB provided by TCP/IP Services for OpenVMS. This is necessary because routines have not yet been implemented in the C Run-Time library.

See Section 8.6 for examples of the compile and link command lines.

7.7. IPv6 Library Functions API

This section describes functions that comprise the Sockets API that support IPv6 and that are supported by TCP/IP Services.

freeaddrinfo()

freeaddrinfo() — This function frees system resources used by an address information structure.

Syntax

```
#include <netdb.h>
```

```
void freeaddrinfo ( struct addrinfo *ai );
```

Arguments

ai

Points to the `addrinfo` structure to be freed.

Description

This function frees an `addrinfo` structure and any dynamic storage pointed to by the structure. The process continues until the function encounters a NULL `ai_next` pointer.

gai_strerror()

gai_strerror() — Provides a descriptive text string that corresponds to an EAI_xxx error value.

Syntax

```
#include <netdb.h>
```

```
const char *gai_strerror ( int ecode );
```

Arguments

ecode

The *ecode* argument is one of the EAI_*xx* values defined for the `getaddrinfo` and `getnameinfo` functions.

Description

The `gai_strerror ()` function returns a descriptive text string that corresponds to an EAI_*xxx* error value. The return value points to a string that describes the error. If the argument is not one of the EAI_*xx* values, the function still returns a pointer to a string whose contents indicates an unknown error.

Return Values

X	text string
-1	Failure

Errors

EAI_AGAIN	The name could not be resolved at this time. Future attempts may succeed.
EAI_BADFLAGS	The flags parameter had an invalid value.
EAI_FAIL	A nonrecoverable error occurred when attempting to resolve the name.
EAI_FAMILY	The address family was not recognized.
EAI_MEMORY	There was a memory allocation failure when trying to allocate storage for the return value.
EAI_NONAME	The name does not resolve for the supplied parameters. Neither **nodename** nor **servname** were supplied. At least one of these must be supplied.
EAI_SERVICE	The service passed was not recognized for the specified socket type.
EAI_SOCKTYPE	The intended socket type was not recognized.
EAI_SYSTEM	A system error occurred; the error code can be found in `errno`.

getaddrinfo()

getaddrinfo() — Takes a service location (**nodename**) or a service name (**servname**), or both, and returns a pointer to a linked list of one or more structures of type `addrinfo`.

Syntax

```
#include <socket.h>

#include <netdb.h>

int getaddrinfo ( const char *nodename, const char *servname, const struct
 addrinfo *hints, struct addrinfo **res);
```

Arguments

nodename

Points to a network node name, alias, or numeric host address (for example, an IPv4 dotted-decimal address or an IPv6 hexadecimal address). An IPv6 nonglobal address with an intended scope zone may also be specified. See Section 1.2.4 for more information. This is a null-terminated string or NULL. NULL means the service location is local to the caller. The **nodename** and **servname** arguments cannot both be NULL.

servname

Points to a network service name or port number. This is a null-terminated string or NULL; NULL returns network-level addresses for the specified nodename. The **nodename** and **servname** arguments cannot both be NULL.

hints

Points to an `addrinfo` structure that contains information about the type of socket, address family, or protocol the caller supports. The `<netdb.h>` header file defines the `addrinfo` structure. If hints is a null pointer, the behavior is the same as if `addrinfo` contained the value 0 for the `ai_flags`, `ai_socktype` and `ai_protocol` members and AF_UNSPEC for the `ai_family` member.

res

Points to a linked list of one or more `addrinfo` structures.

Description

The `getaddrinfo()` routine takes a service location (**nodename**) or a service name (**servname**), or both, and returns a pointer to a linked list of one or more structures of type `addrinfo`. Its members specify data obtained from either the local hosts database TCPIP$ETC:IPNODES.DAT file, local TCPIP$HOSTS.DAT file, or one of the files distributed by DNS/BIND.

The `<netdb.h>` header file defines the `addrinfo` structure.

If the **hints** argument is non-NULL, all `addrinfo` structure members other than the following members must be zero or a NULL pointer:

- **ai_flags**

  Controls the processing behavior of `getaddrinfo()`. See Table 7.8 for a complete description of the flags.

- **ai_family**

  Specifies to return addresses for use with a specific protocol family.

  - If you specify a value of AF_UNSPEC, the routine returns addresses for any protocol family that can be used with **nodename** or **servname**.

  - If the value is not AF_UNSPEC and `ai_protocol` is not zero, the routine returns addresses for use only with the specified protocol family and protocol.

  - If the application handles only IPv4, set this member of the **hints** structure to PF_INET.

  - If `ai_family` is set to PF_INET6, the function looks only in the TCPIP $ETC:IPNODES.DAT file and the lookup fails in the BIND database.

- **ai_socktype**

Specifies a socket type for the given service. If you specify a value of 0, you will accept any socket type. This resolves the service name for all socket types and returns all successful results.

- **ai_protocol**

  Specifies a network protocol. If you specify a value of 0, you will accept any protocol. If the application handles only TCP, set this member to IPPROTO_TCP.

Table 7.8 describes the ai_flags member values.

Table 7.8. ai_flags Member Values

Flag Value	**Description**	
AI_V4MAPPED	**If af value is AF_INET:**	**If af value is AF_INET6:**
	Ignored.	Searches for AAAA records. The lookup sequence is LOCAL host database, TCPIP $ETC:IPNODES.DAT, BIND database. The lookup for a particular type of record, for example an AAAA record, will be performed in each database before moving on to perform a lookup for the next type of record. If AAAA records found, returns IPv6 addresses; no search for A records is performed. If no AAAA records found, searches for A records. If A records found, returns IPv4-mapped IPv6 addresses. If no A records found, returns a NULL pointer.
AI_ALL | AI_V4MAPPED	**If af value is AF_INET:**	**If af value is AF_INET6:**
	Ignored.	Searches for AAAA records. The lookup sequence is LOCAL host database, TCPIP $ETC:IPNODES.DAT, BIND database. The lookup for a particular type of record, for example an AAAA record, will be performed in each database before moving on to perform a lookup for the next type of record.

Flag Value	Description	
		If AAAA records found, IPv6 addresses will be included with the returned addresses. Searches for A records. If A records found, returns IPv4-mapped IPv6 addresses and also any IPv6 addresses that were found with the AAAA record search. If no A records found, returns a NULL pointer.
AI_CANONNAME	If the **nodename** argument is not NULL, the function searches f or the specified node's canonical name. Upon successful completion, the `ai_canonname` member of the first `addrinfo` structure in the linked list points to a null-terminated string containing the canonical name of the specified node name. If the **nodename** argument is an address literal, the `ai_cannonname` member will refer to the **nodename** argument that has been converted to its numeric binary form, in network byte order. If the canonical name is not available, the `ai_canonname` member refers to the **nodename** argument or to a string with the same contents. The `ai_flags` field contents are undefined.	
AI_NUMERICHOST	A non-NULL node name string must be a numeric host address string. Resolution of the service name is not performed.	
AI_NUMERICSERV	A non-NULL service name string must be a numeric port string. Resolution of the service name is not performed.	
AI_PASSIVE	Returns a socket address structure that your application can use in a call to `bind()`. If the **nodename** parameter is a NULL pointer, the IP address portion of the socket address structure is set to INADDR_ANY (for an IPv4 address) or IN6ADDR_ANY_INIT (for an IPv6 address). If not set, returns a socket address structure that your application can use to call connect() (for a connection-oriented protocol) or either `connect()`, `sendto()`, or `sendmsg()` (for a connectionless protocol). If the **nodename** argument is a NULL pointer, the IP address portion of the socket address structure is set to the loopback address.	

You can use the flags in any combination to achieve finer control of the translation process. The AI_ADDRCONFIG flag is typically used in combination with other flags to modify the search based on the source address or addresses configured on the system. The following table describes how the AI_ADDRCONFIG flags works by itself.

Flag Value	**If af Value is AF_INET**	**If af Value is AF_INET6**
AI_ADDRCONFIG	If an IPv4 source address is configured, searches for A records.	If an IPv6 source address is configured, searches for AAAA records.

Most applications will want to use the combination of the AI_ADDRCONFIG and AI_V4MAPPED flags to control their search. To simplify this for the programmer, the AI_DEFAULT symbol, which is a logical OR of AI_ADDRCONFIG and AI_V4MAPPED, is defined. The following table describes how AI_DEFAULT directs the search.

Flag Value	**If af Value is AF_INET**	**If af Value is AF_INET6**
AI_DEFAULT	Searches for A records only if an IPv4 source address is configured on the system. If found, returns IPv4 addresses. If not, returns a NULL pointer.	Searches for AAAA records only if an IPv6 source address is configured on the system. If found, returns IPv6 addresses. If not and if an IPv4 address is configured on the system, searches for A records. If found, returns IPv4-mapped IPv6 addresses. If not, returns a NULL pointer.

These flags are defined in `<netdb.h>`.

addrinfo Structure Processing

Upon successful return, `getaddrinfo()` returns a pointer to a linked list of one or more `addrinfo` structures. The application can process each `addrinfo` structure in the list by following the `ai_next` pointer until a NULL pointer is encountered. In each returned `addrinfo` structure, the `ai_family`, `ai_socktype`, and `ai_protocol` members are the corresponding arguments for a call to the `socket()` function. The `ai_addr` member points to a filled-in socket address structure whose length is specified by the `ai_addrlen` member.

Return Values

0 (zero)	Success
nonzero value	Failure

getnameinfo()

getnameinfo() — Maps addresses to names in a protocol-independent way.

Syntax

```
#include <socket.h>

#include <netdb.h>

int getnameinfo ( const struct sockaddr *sa, size_t salen,
 char *node, size_t nodelen,
 char *service, size_t servicelen,
 int flags );
```

Arguments

sa

Points either to a `sockaddr_in` structure (for IPv4) or to a `sockaddr_in6` structure (for IPv6) that holds the IP address and port number.

salen

Specifies the length of either the `sockaddr_in` structure or the `sockaddr_in6` structure.

node

Points to a buffer in which to receive the null-terminated network node name or alias corresponding to the address contained in the **sa**. The node name may be suffixed with an intended scope zone as described in Section 1.2.4. A NULL pointer instructs the routine not to return a node name. The node parameter and service parameter cannot both be zero.

nodelen

Specifies the length of the node buffer. A value of zero instructs the routine not to return a node name.

service

Points to a buffer in which to receive the null-terminated network service name associated with the port number contained in **sa**. A NULL pointer instructs the routine not to return a service name. The node parameter and service parameter cannot both be 0.

servicelen

Specifies the length of the service buffer. A value of zero instructs the routine not to return a service name.

flags

Specifies changes to the routine's default actions. By default, the routine searches for the fully qualified domain name of the node in the host's database and returns it. See Table 7.9 for a list of flag bits and their meanings.

Description

The `getnameinfo()` routine looks up an IP address and port number in a `sockaddr` structure specified by **sa** and returns node name and service name text strings in the buffers pointed to by the **node** and **service** parameters, respectively.

If the node name is not found, the routine returns the numeric form of the node address, regardless of the value of the **flags** parameter. If the service's name is not found, the routine returns the numeric form of the service's address (port number) regardless of the value of the **flags** parameter.

The application must provide buffers large enough to hold the fully qualified domain name and the service name, including the terminating null characters.

Flag bits

Table 7.9 describes the flag bits and, if set, their meanings.

Table 7.9. Flag Bits

Flag Value	Description
NI_DGRAM	Specifies that the service is a datagram service (SOCK_DGRAM). The default assumes a stream service (SOCK_STREAM). This is required for the few ports (512-514) that have different services for UDP and TCP.
NI_NAMEREQD	Returns an error if the host name cannot be located in the host's database.
NI_NOFQDN	Searches the host's database and returns the node name portion of the fully qualified domain name for local hosts.
NI_NUMERICHOST	Returns the numeric form of the host's address instead of its name. Resolution of the host name is not performed.
NI_NUMERICSERV	Returns the numeric form (port number) of the service address instead of its name. Resolution of the host name is not performed.

The two NI_NUMERIC* flags are required to support the `-n` flag that many commands provide. All flags are defined in `<netdb.h>` header file.

Return Values

0 (zero)	Success
nonzero value	Failure

if_freenameindex()

if_freenameindex() — Frees dynamic memory allocated by `if_nameindex()` to the array of interface names and indexes.

Syntax

```
#include <if.h>

void if_freenameindex ( struct if_nameindex *ptr );
```

Arguments

ptr

Pointer that was returned by the `if_nameindex()` function.

Description

The `if_freenameindex()` function frees dynamic memory that was allocated by the `if_nameindex()` function.

if_indextoname()

if_indextoname() — Maps an interface index to its corresponding name.

Syntax

```
#include <if.h>
```

```
char *if_indextoname ( unsigned int ifindex, char *ifname );
```

Arguments

ifindex

The interface index.

ifname

The **ifname** argument points to a buffer that is IFNAMSIZ bytes in length (IFNAMSIZ is defined in `<if.h>`). If an interface name is found, it is returned in the buffer.

Description

The `if_indextoname()` function maps an interface index to its corresponding name.

Return Values

Interface name	If interface name is found, it is returned to the buffer.
NULL	If no interface name corresponds to the specified index, the function returns NULL and sets errno to ENXIO.

Errors

ENXIO	No interface name corresponds to the specified index.
System error	A system error.

if_nameindex()

if_nameindex() — Returns an array of all interface names and indexes.

Syntax

```
#include <if.h>
```

```
struct if_nameindex *if_nameindex ( void );
```

Description

The `if_nameindex()` function dynamically allocates memory for an array of `if_nameindex` structures, one structure for each interface. A structure with an `if_index` value of 0 and a NULL `if_name` value indicates the end of the array.

The following `if_nameindex` structure must also be defined (by including `<if.h>`) prior to the call to `if_nameindex()`:

```
struct if_nameindex {
   unsigned int   if_index;
   char           *if_name;
};
```

To free the memory allocated by this function, use the `if_freenameindex()` function. If an error occurs, the function returns a NULL pointer and sets `errno` to an appropriate value.

Return Values

NULL	Indicates an error; errno is set to an appropriate value.

if_nametoindex()

if_nametoindex() — Maps an interface name to its corresponding index.

Syntax

```
#include <if.h>

unsigned int if_nametoindex ( const char *ifname );
```

Arguments

ifname

The name of the interface.

Description

If **ifname** is the name of an interface, the `if_nametoindex()` function returns the interface index corresponding to the name.

Return Values

Interface index	Success
0	Failure

inet6_opt_append()

inet6_opt_append() — Returns the length of an IPv6 extension header with a new option and appends the option.

Syntax

```
#include <ip6.h>
int inet6_opt_append ( void *extbuf, size_t extlen, int offset,
 uint8_t type,
 size_t len, uint_t align, void **databufp);
```

Arguments

extbuf

Points to a buffer that contains an extension header. This is either a valid pointer

or a NULL pointer.

extlen

Specifies the length of the extension header to initialize. Valid values are 0 if

extbuf equals 0, a value returned by inet6_opt_finish( ), or any number that

is a multiple of 8.

offset

Specifies the length of the existing extension header. Obtain this value from a

prior call to inet6_opt_init( ) or inet6_opt_append( ).

type

Specifies the type of option. Specify a value from 2 to 255, inclusive, excluding

194.

len

Specifies the length of the option data, excluding the option type and option

length fields. Specify a value from 0 to 255, inclusive.

align

Specifies the alignment of the option. Specify one of the following values: 1, 2, 4,

or 8.

databufp

Points to a buffer that contains the option data.

Description

The `inet6_opt_append()` function, when called with **extbuf** as a NULL pointer and **extlen** as 0, returns the updated number of bytes in an extension header.

If you specify **extbuf** as a valid pointer and valid **extlen** and **align** parameters, the function returns the same information as in the previous case, but also inserts the pad option, initializes the **type** and **len** fields, and returns a pointer to the location for the option content.

After you call `inet6_opt_append()`, you can then use the data buffer directly or call `inet6_optt_set_val()` to specify the option contents.

Return Values

x	Upon successful completion, the `inet6_opt_append()` function returns the updated number of bytes in an extension header.
-1	Failure

inet6_opt_find()

inet6_opt_find() — Finds a specific option in an extension header.

Syntax

```
#include <ip6.h>

int inet6_opt_find ( void *extbuf, size_t extlen, int offset, uint8_t type,
 size_t *lenp, void **databufp );
```

Arguments

extbuf

Points to a buffer that contains an extension header.

extlen

Specifies the length, in bytes, of the extension header.

offset

Specifies the location in the extension header of an option. Valid values are either 0 (zero) for the first option or the length returned from a previous call to either `inet6_opt_next()` or `inet6_opt_find()`.

type

Specifies the type of option to find.

lenp

Points to the length of the option found.

databufp

Points to the option data.

Description

The `inet6_opt_find()` function searches a received option extension header for an option specified by **type**. If it finds the specified option, it returns the option length and a pointer to the option data. In addition, it returns an offset to the next option that you specify in the **offset** parameter to subsequent calls to `inet6_opt_next()` in order to search for additional occurrences of the same option type.

Return Values

x	Upon successful completion, the `inet6_opt_find()` function returns an offset from which you can begin the next search in the data buffer.
-1	Failure

inet6_opt_finish()

inet6_opt_finish() — Returns the total length of an IPv6 extension header, including padding, and initializes the option.

Syntax

```
#include <ip6.h>

int inet6_opt_finish ( void *extbuf, size_t extlen, int offset );
```

Arguments

extbuf

Points to a buffer that contains an extension header. This is either a valid pointer or a NULL pointer.

extlen

Specifies the length of the extension header to finish initializing. A valid value is any number greater than or equal to 0.

offset

Specifies the length of the existing extension header. Obtain this value from a prior call to `inet6_opt_init()` or `inet6_opt_append()`.

Description

The `inet6_opt_finish()` function when called with **extbuf** as a NULL pointer and **extlen** as 0, returns the total number of bytes in an extension header, including final padding.

If you specify **extbuf** as a valid pointer and a valid **extlen** parameter, the function returns the same information as in the previous case, increments the buffer pointer, and verifies that the buffer is large enough to hold the header.

Return Values

x	Upon successful completion, the `inet6_opt_finish()` function returns the total number of bytes in an extension header, including padding.
-1	Failure

inet6_opt_get_val()

inet6_opt_get_val() — Extracts data items from the data portion of an IPv6 option.

Syntax

```
#include <ip6.h>

int inet6_opt_get_val ( void *databuf, size_t offset, void *val, int vallen
 );
```

Arguments

databuf

Points to a buffer that contains an extension header. This is a pointer returned by a call to `inet6_opt_find( )` or `inet6_opt_next( )`.

offset

Specifies the location in the data portion of the option from which to extract the data. You can access the first byte after the option type and length by specifying the offset of 0.

val

Points to a destination for the extracted data.

vallen

Specifies the length of the data, in bytes, to be extracted.

Description

The `inet6_opt_get_val()` function copies data items from data buffer **databuf** beginning at **offset** to the location **val**. In addition, it returns the **offset** for the next data field to assist you in extracting option content that has multiple fields.

Make sure that each field is aligned on its natural boundaries.

Return Values

x	Upon successful completion, the `inet6_opt_get_val()` function returns the offset for the next field in the data buffer.
-1	Failure

inet6_opt_init()

inet6_opt_init() — Returns the length of an IPv6 extension header with no options and initializes the header.

Syntax

```
#include <ip6.h>
```

```
int inet6_opt_init ( void *extbuf, size_t extlen );
```

Arguments

extbuf

Points to a buffer that contains an extension header. This is either a valid pointer or a NULL pointer.

extlen

Specifies the length of the extension header to initialize. Valid values are 0 and any number that is a multiple of 8.

Description

The `inet6_opt_init()` function when called with **extbuf** as a NULL pointer and **extlen** as 0, returns the number of bytes in an extension header that has no options.

If you specify **extbuf** as a valid pointer and **extlen** as a number that is a multiple of 8, the function returns the same information as in the previous case, initializes the extension header, and sets the length field.

Return Values

x	Upon successful completion, the `inet6_opt_init()` function returns the number of bytes in an extension header with no options.
-1	Failure

inet6_opt_next()

inet6_opt_next() — Parses received option extension headers.

Syntax

```
#include <ip6.h>

int inet6_opt_next ( void *extbuf, size_t extlen, int offset, unit8_t
 *typep,
 size_t *lenp, void **databufp );
```

Arguments

extbuf

Points to a buffer that contains an extension header.

extlen

Specifies the length, in bytes, of the extension header.

offset

Specifies the location in the extension header of an option. Valid values are either 0 for the first option or the length returned from a previous call to either `inet6_opt_next( )` or `inet6_opt_find( )`.

typep

Points to the type of the option found.

lenp

Points to the length of the option found.

databufp

Points to the option data.

Description

The `inet6_opt_next()` function parses a received option extension header and returns the next option. In addition, it returns an offset to the next option that you specify in the **offset** parameter to subsequent calls to `inet6_opt_next()`.

This function does not return any PAD1 or PADN options.

Return Values

x	Upon successful completion, the `inet6_opt_next()` function returns the offset for the next option in the data buffer.
-1	Failure

inet6_opt_set_val()

inet6_opt_set_val() — Adds one component of the option content to the options header.

Syntax

```
#include <ip6.h>

int inet6_opt_set_val ( void *databuf, size_t offset, void *val int vallen
 );
```

Arguments

databuf

Points to a buffer that contains an extension header. This is a pointer returned by a call to `inet6_opt_append( )`.

offset

Specifies the location in the data portion of the option into which to insert the data. You can access the first byte after the option type and length by specifying the offset of 0 (zero).

val

Points to the data to be inserted.

vallen

Specifies the length of the data, in bytes, to be inserted.

Description

The `inet6_opt_set_val()` function copies data items at the location **val** into a data buffer **databuf** beginning at **offset**. In addition, it returns the offset for the next data field to assist you in composing content that has multiple fields.

Make sure that each field is aligned on its natural boundaries.

Return Values

x	Upon successful completion, the `inet6_opt_set_val()` function returns the offset for the next field in the data buffer.
-1	Failure

inet6_rth_add()

inet6_rth_add() — Adds an IPv6 address to the routing header under construction.

Syntax

```
#include <ip6.h>

int inet6_rth_add ( void *bp, const struct in6_addr *addr );
```

Arguments

bp

Points to a buffer that is to contain an IPv6 routing header.

addr

Points to an IPv6 address to add to the routing header.

Description

The `inet6_rth_add()` function adds IPv6 address to the end of the routing header under construction. The address pointed to by **addr** cannot be either an IPv6 V4-mapped address or an IPv6 multicast address.

The function increments the `ip60r_segleft` member in the `ip6_rthdr0` structure. The `ip6_rthdr0` structure is defined in `<ip6.h>`.

Only routing header type 0 is supported.

Return Values

x	Upon successful completion, the `inet6_rth_add()` function returns 0 (zero).
-1	Failure

inet6_rth_getaddr()

inet6_rth_getaddr() — Retrieves an address for an index from an IPv6 routing header.

Syntax

```
#include <ip6.h>
```

```
struct in6_addr *inet6_rth_getaddr ( const void *bp, int index );
```

Arguments

bp

Points to a buffer that contains an IPv6 routing header.

index

Specifies a value that identifies a position in a routing header for a specific address. Valid values range from 0 to the return value from `inet6_rth_segments()` minus 1.

Description

The `inet6_rth_getaddr()` function uses a specified **index** value and retrieves a pointer to an address in a Routing header specified by **bp**. Call `inet6_rth_segments()` before calling this function in order to determine the number of segments (addresses) in the routing header.

Return Values

x	Upon successful completion, the `inet6_rth_getaddr()` function returns a pointer to an address.
NULL pointer	Failure

inet6_rth_init()

inet6_rth_init() — Initializes an IPv6 routing header buffer.

Syntax

```
#include <ip6.h>
```

```
void *inet6_rth_init ( void *bp, int bp_len, int type, int segments );
```

Arguments

bp

Points to a buffer that is to contain an IPv6 routing header.

bp_len

Specifies the length, in bytes, of the buffer.

type

Specifies the type of routing header. The valid value is IPV6_RTHDR_TYPE_0 for IPv6 routing header type 0.

segments

Specifies the number of segments or addresses that are to be included in the routing header. The valid value is from 0 to 127, inclusive.

Description

The `inet6_rth_init()` function initializes a buffer and buffer data for an IPv6 routing header. The function sets the `ip6r0_segleft`, `ip6r0_nxt`, and `ip6r0_reserved` members in the ip6_rthdr0 structure to zero. In addition, it sets the `ip6r0_type` member to type and sets the `ip6r0_len` member based in the segments parameter. (See RFC 2460 for a description of the actual value.) The `ip6_rthdr0` structure is defined in `<ip6.h>`.

The application must allocate the buffer. Use the `inet6_rth_space()` function to determine the buffer size.

Use the returned pointer as the first argument to the `inet6_rth_add()` function.

Return Values

x	Upon successful completion, the `inet6_rth_init()` function returns a pointer to the buffer that is to contain the routing header.
NULL pointer	If the **type** is not supported, the **bp** is a null, or the number of **bp_len** is invalid.

inet6_rth_reverse()

inet6_rth_reverse() — Reverses the order of addresses in an IPv6 routing header.

Syntax

```
#include <ip6.h>

int inet6_rth_reverse ( const void *in, void *out );
```

Arguments

in

Points to a buffer that contains an IPv6 routing header.

out

Points to a buffer that is to contain the routing header with the reversed addresses. This parameter can point to the same buffer specified by the in parameter.

Description

The `inet6_rth_reverse()` function reads an IPv6 routing header and writes a new routing header, reversing the order of addresses in the new header. The **in** and **out** parameters can point to the same buffer.

The function sets the `ip6r0_segleft` member in the `ip6_rthdr0` structure to the number of segments (addresses) in the new header.

The `ip6_rthdr0` structure is defined in `<ip6.h>`.

Return Values

0 (zero)	Success
-1	Failure

inet6_rth_segments()

inet6_rth_segments() — Returns the number of segments (addresses) in an IPv6 routing header.

Syntax

```
#include <ip6.h>

int inet6_rth_segments ( const void *bp );
```

Arguments

bp

Points to a buffer that contains an IPv6 routing header.

Description

The `inet6_rth_segments()` function returns the number of segments (or addresses) in an IPv6 routing header.

Return Values

x	Upon successful completion, the `inet6_rth_segments()` function returns the number of segments, 0 (zero) or greater than 0.
-1	Failure

inet6_rth_space()

inet6_rth_space() — Returns the number of bytes required for an IPv6 routing header.

Syntax

```
#include <ip6.h>

size_t inet6_rth_space ( int type, int segments );
```

Arguments

type

Specifies the type of routing header. The valid value is `IPV6_RTHDR_TYPE_0` for IPv6 routing header type 0.

segments

Specifies the number of segments or addresses that are to be included in the routing header. The valid value is from 0 to 127, inclusive.

Description

The `inet6_rth_space()` function determines the amount of space, in bytes, required for a routing header. Although the function returns the amount of space required, it does not allocate buffer space. This enables the application to allocate a larger buffer.

If the application uses ancillary data, it must pass the returned length to `CMSG_LEN()` to determine the amount of memory required for the ancillary data object, including the `cmsghdr` structure.

Note

If an application wants to send other ancillary data objects, it must specify them to `sendmsg()` as a single `msg_control` buffer.

Return Values

x	Upon successful completion, the `inet6_rth_space()` function returns the length, in bytes, of the routing header and the specified number of segments.
0 (xero)	Failure, if the type is not supported or the number of segments is invalid for the type of routing header.

inet_ntop()

inet_ntop() — Converts a numeric address to a text string suitable for presentation.

Syntax

```
#include <inet.h>

const char *inet_ntop ( int af, const void *src, char *dst, size_t size );
```

Arguments

af

Specifies the address family. Valid values are AF_INET for an IPv4 address and AF_INET6 for an IPv6 address.

src

Points to a buffer that contains the numeric Internet address.

dst

Points to a buffer that is to contain the text string.

size

Specifies the size of the buffer pointed to by the **dst** parameter. For IPv4 addresses, the minimum buffer size is 16 octets; for IPv6 addresses, the minimum buffer size is 46 octets. The `<in.h>` header file defines the INET_ADDRSTRLEN and INET6_ADDRSTRLEN constants, respectively, for these values.

Description

The `inet_ntop()` function converts a numeric Internet address value to a text string.

Return Values

Pointer to the buffer containing the text string.	Success
Pointer to the buffer containing NULL.	Failure

inet_pton()

inet_pton() — Converts an address in its standard text presentation form its numeric binary form, in network byte order.

Syntax

```
#include <inet.h>

int inet_pton ( int af, const char *src, void *dst );
```

Arguments

af

Specifies the address family. Valid values are AF_INET for an IPv4 address and AF_INET6 for an IPv6 address.

src

Points to the address text string to be converted.

dst

Points to a buffer that is to contain the numeric address.

Description

The `inet_pton()` function converts a text string to a numeric value in Internet network byte order.

- If the **af** parameter is AF_INET, the function accepts a string in the standard IPv4 dotted-decimal format:

  ```
  ddd.ddd.ddd.ddd
  ```

  In this format, *ddd* is a one- to three-digit decimal number between 0 and 255.

- If the **af** parameter is AF_INET6, the function accepts a string in the following format:

  ```
  x:x:x:x:x:x:x:x
  ```

  In this format, *x* is the hexadecimal value of a 16-bit piece of the address.

  IPv6 addresses can contain long strings of zero (0) bits. To make it easier to write these addresses, you can use double-colon characters (::) one time in an address to represent 1 or more 16-bit groups of zeros.

- For mixed IPv4 and IPv6 environments, the following format is also accepted:

  ```
  x:x:x:x:x:x:ddd.ddd.ddd.ddd
  ```

  In this format, *x* is the hexadecimal value of a 16-bit piece of the address, and *ddd* is a one- to three-digit decimal value between 0 and 255 that represents the IPv4 address. See RFC 2373 for more information about IPv6 addressing formats.

The calling application is responsible for ensuring that the buffer referred to by the **dst** parameter is large enough to hold the numeric address. AF_INET addresses require 4 bytes and AF_INET6 addresses require 16 bytes.

Return Values

1	Success
0 (zero)	If the input string is neither a valid IPv4 dotted-decimal string nor a valid IPv6 address string, the function returns a 0.
-1	Failure. `errno` is set to the following value.

Errors

EAFNOSUPPORT	The address family specified in the **af** parameter is unknown.

Chapter 8. Porting Applications

This chapter describes the changes you must make in your application code to operate in an IPv6 networking environment.

- Name changes
- Structure changes
- Other changes

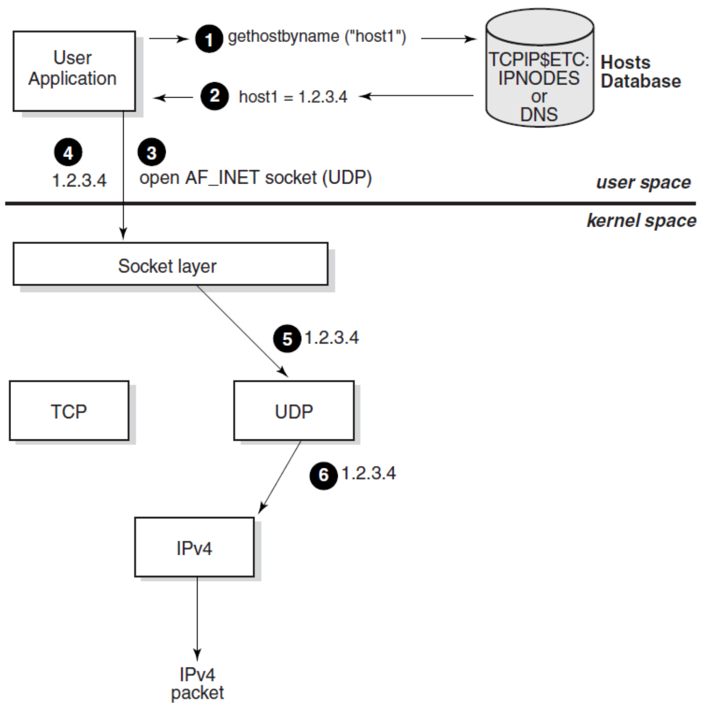
You can also use this information as guidelines for creating new IPv6-ready applications.

See RFC 3493, *Basic Socket Interface Extensions for IPv6*, for complete information on the changes to the BSD socket applications programming interface (API). See RFC 3542, *Advanced Sockets API for IPv6* for complete information on how to use raw sockets and header information in IPv6 applications.

8.1. Using AF_INET6 Sockets

At present, applications use AF_INET sockets for IPv4 communications. Figure 8.1 shows a sample sequence of events for an application that uses an AF_INET socket to send IPv4 packets.

Figure 8.1. Using AF_INET Socket for IPv4 Communications



1. Application calls `gethostbyname()` and passes the host name, host1.
2. The search finds host1 in the hosts database and `gethostbyname` returns the IPv4 address 1.2.3.4.
3. The application opens an AF_INET socket.

4. The application sends information to the 1.2.3.4 address.

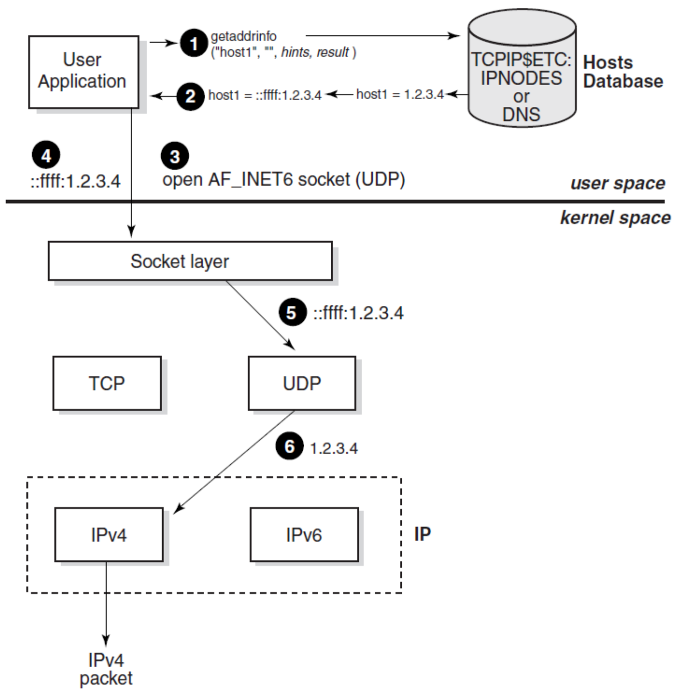
5. The socket layer passes the information and address to the UDP module.

6. The UDP module puts the 1.2.3.4 address into the packet header and passes the information to the IPv4 module for transmission.

Section 8.6.1.1 contains sample program code that demonstrates these steps.

You can use the AF_INET6 socket for both IPv6 and IPv4 communications. For IPv4 communications, create an AF_INET6 socket and pass it a `sockaddr_in6` structure that contains an IPv4-mapped IPv6 address (for example, `::ffff:1.2.3.4`). Figure 8.2 shows the sequence of events for an application that uses an AF_INET6 socket to send IPv4 packets.

Figure 8.2. Using AF_INET6 Socket to Send IPv4 Communications



1. Application calls `getaddrinfo()` and passes the host name (host1), the AF_INET6 address family hint, and the (AI_V4MAPPED | AI_ADDRCONFIG) flag hint. The flag tells the function that if an IPv4 address is found for host1, return the address as an IPv4-mapped IPv6 address.

2. The search finds an IPv4 address, 1.2.3.4, for host1 in the hosts database and `getaddrinfo()` returns the IPv4-mapped IPv6 address `::ffff:1.2.3.4`.

3. The application opens an AF_INET6 socket.

4. The application sends information to the `::ffff:1.2.3.4` address.

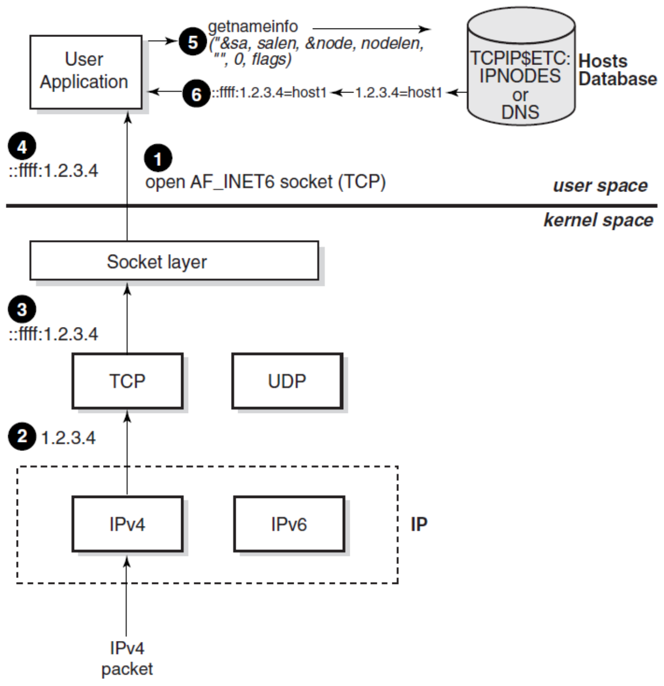
5. The socket layer passes the information and address to the UDP module.

6. The UDP module identifies the IPv4-mapped IPv6 address, puts the 1.2.3.4 address into the packet header, and passes the information to the IPv4 module for transmission.

AF_INET6 sockets can receive messages sent to either IPv4 or IPv6 addresses on the system. An AF_INET6 socket uses the IPv4-mapped IPv6 address format to represent IPv4 addresses. Figure 8.3

shows the sequence of events for an application that uses an AF_INET6 socket to receive IPv4 packets.

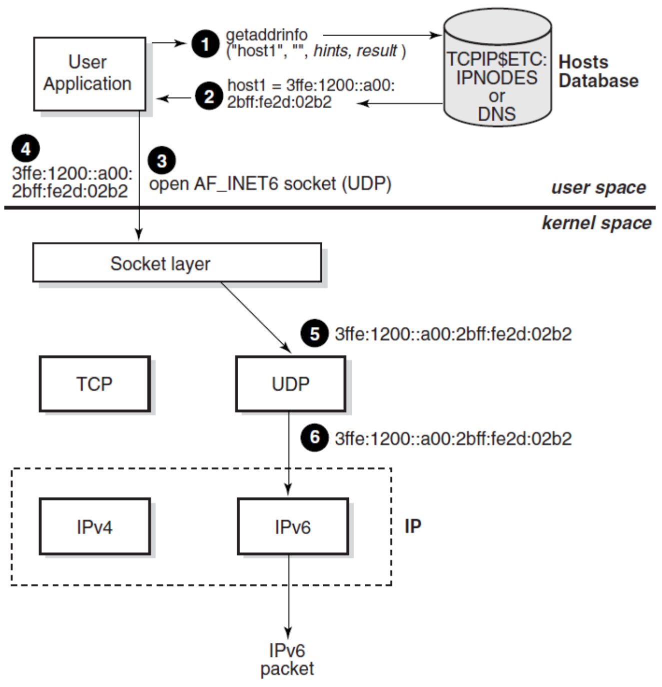
Figure 8.3. Using AF_INET6 Socket to Receive IPv4 Communications



1. The application opens an AF_INET6 socket, binds to it, and listens on it.
2. An IPv4 packet arrives and passes through the IPv4 module.
3. The TCP layer strips off the packet header and passes the information and the IPv4-mapped IPv6 address `::ffff:1.2.3.4` to the socket layer.
4. The application calls `accept()` and retrieves the information from the socket.
5. The application calls `getnameinfo()` and passes the `::ffff:1.2.3.4` address and the NI_NAMEREQD flag. The flag tells the function to return the host name for the address. See Table 7.9 for a description of the flag bits and their meanings.
6. The search finds the host name for the 1.2.3.4 address in the hosts database, and `getnameinfo()` returns the host name.

For IPv6 communications, create an AF_INET6 socket and pass it a sockaddr_in6 structure that contains an IPv6 address (for example, `3ffe:1200::a00:2bff:fe2d:02b2`). Figure 8.4 shows the sequence of events for an application that uses an AF_INET6 socket to send IPv6 packets.

Figure 8.4. Using AF_INET6 Socket for IPv6 Communications


User Application
1 getaddrinfo ("host1", "", *hints, result*)
TCPIP$ETC: IPNODES or DNS
Hosts Database
2 host1 = 3ffe:1200::a00: 2bff:fe2d:02b2
4 3ffe:1200::a00: 2bff:fe2d:02b2
3 open AF_INET6 socket (UDP)
user space
kernel space
Socket layer
5 3ffe:1200::a00:2bff:fe2d:02b2
TCP
UDP
6 3ffe:1200::a00:2bff:fe2d:02b2
IPv4
IPv6
IP
IPv6 packet

1. Application calls `getaddrinfo()` and passes the host name (host1), the AF_INET6 address family hint, and the (AI_V4MAPPED | AI_ADDRCONFIG) flag hint. The flag tells the function that if an IPv4 address is found for host1, to return it.

2. The search finds an IPv6 address for host1 in the hosts database, and `getaddrinfo()` returns the IPv6 address `3ffe:1200::a00:2bff:fe2d:02b2`.

3. The application opens an AF_INET6 socket.

4. The application sends information to the `3ffe:1200::a00:2bff:fe2d:02b2` address.

5. The socket layer passes the information and address to the UDP module.

6. The UDP module identifies the IPv6 address and puts the `3ffe:1200::a00:2bff:fe2d:02b2` address into the packet header and passes the information to the IPv6 module for transmission.

Section 8.6.2.1 contains sample program code that demonstrates these steps.

The following sections show how to convert an existing AF_INET application to an AF_INET6 application that is capable of communicating over both IPv4 and IPv6.

8.2. Name Changes

Most of the changes required are straightforward and mechanical, though some may require a bit of code restructuring. For example, a routine that returns an int data type holding an IPv4 address may need to be modified to take as an extra parameter a pointer to an in6_addr into which it writes the IPv6 address.

Table 8.1 summarizes the changes you must make to your application's code.

Table 8.1. Name Changes

Search file for	Replace with	Comments
AF_INET	AF_INET6	Replace with IPv6 address family macro.
PF_INET	PF_INET6	Replace with IPv6 protocol family macro.
INADDR_ANY	in6addr_any	Replace with IPv6 global variable.

8.3. Structure Changes

The structure names and field names have changed for the following structures:

- `in_addr`
- `sockaddr_in`
- `sockaddr`
- `hostent`

The following sections discuss these changes.

8.3.1. in_addr Structure

Applications that use the IPv4 `in_addr` structure must be changed to use the IPv6 `in6_addr` structure, as follows:

IPv4 Structure	IPv6 Structure
`struct in_addrunsigned int s_addr`	`struct in6_addruint8_t s6_addr`

Make the following changes to your application, as needed:

1. Change the structure name `in_addr` to `in6_addr`.
2. Change the data type from `unsigned int` to `uint8_t` and the field name `s_addr` to `s6_addr`.

8.3.2. sockaddr Structure

Applications that use the generic socket address structure (`sockaddr`) to hold an AF_INET socket address (`sockaddr_in`) must be changed to use the AF_INET6 sockaddr_in6 structure, as follows:

AF_INET Structure	AF_INET6 Structure
`struct sockaddr`	`struct sockaddr_in6`

Make the following change to your application, as needed:

1. Change structure name `sockaddr` to `sockaddr_in6`.

Note

A `sockaddr_in6` structure is larger than a `sockaddr` structure.

8.3.3. sockaddr_in Structure

Applications that use the BSD Version 4.4 IPv4 `sockaddr_in` structure must be changed to use the IPv6 `sockaddr_in6` structure, as follows:

IPv4 Structure	IPv6 Structure
`struct sockaddr_inunsigned char sin_lensa_family_t sin_familyin_port_t sin_portstruct addr sin_addr`	`struct sockaddr_in6uint8_t sin6_lensa_family_t sin6_familyint_port_t sin6_portstruct in6_addr sin6_addr`

Make the following changes to your application, as needed:

1. Change structure name `sockaddr_in` to `sockaddr_in6`. Initialize the entire `sockaddr_in6` structure to zero after your structure declarations.
2. Change the data type `unsigned char` to `uint8_t` and the field name `sin_len` to `sin6_len`.
3. Change the field name `sin_family` to `sin6_family`.
4. Change the field name `sin_port` to `sin6_port`.
5. Change the field name `sin_addr` to `sin6_addr`.

8.3.4. hostent Structure

Applications that use the `hostent` structure must be changed to use the `addrinfo` structure, as follows:

AF_INET Structure	AF_INET6 Structure
`struct hostent`	`struct addrinfo`

Make the following change to your application, as needed:

1. Change the structure name `hostent` to `addrinfo`.

See also Section 8.4.2 for related changes.

8.4. Function Changes

The names and parameters have changed for the following functions:

- `gethostbyaddr()`
- `gethostbyname()`
- `inet_ntoa()`
- `inet_addr()`

The following sections discuss these changes.

8.4.1. gethostbyaddr() Function

Applications that use the IPv4 `gethostbyaddr()` function must be changed to use the IPv6 `getnameinfo()` function, as follows:

AF_INET Call	**AF_INET6 Call**
gethostbyaddr(*xxx*,4,AF_INET)	err=getnameinfo(*&sa*, *salen*, *node*, *nodelen*, *service*, *servicelen*, *flags*);

Make the following change to your application, as needed:

1. Change the function name from `gethostbyaddr()` to `getnameinfo()` and provide a pointer to the socket address structure, a character string for the returned node name, an integer for the length of the returned node name, a character string to receive the returned service name, an integer for the length of the returned service name, and an integer that specifies the type of address processing to be performed.

8.4.2. gethostbyname() Function

Applications that use the `gethostbynam()e` function must be changed to use the `getaddrinfo()` function, as follows:

AF_INET Call	**AF_INET6 Call**
gethostbyname(*name*)	err=getaddrinfo(*nodename*, *servname*, *&hints*, *&res*);...freeaddrinfo(*&ai*);

Make the following changes to your application, as needed:

1. Change the function name from `gethostbyname()` to `getaddrinfo()` and provide a character string that contains the node name, a character string that contains the service name to use, a pointer to a hints structure that contains processing options, and a pointer to an `addrinfo` structure or structures for the returned address information.

2. Add a call to the `freeaddrinfo()` routine to free the `addrinfo` structure or structures when your application is finished using them.

8.4.3. inet_ntoa() Function

Applications that use the `inet_ntoa()` function must be changed to use the `getnameinfo()` function, as follows:

AF_INET Call	**AF_INET6 Call**
inet_ntoa(*addr*)	err=getnameinfo(*&sa*, *salen*, *node*, *nodelen*, *service*, *servicelen*, NI_NUMERICHOST);

Make the following change to your application, as needed:

1. Change the function name from `inet_ntoa()` to `getnameinfo()` and provide a pointer to the socket address structure, a character string for the returned node name, an integer for the length of the returned node name, a character string to receive the returned service name, an integer for the length of the returned service name, and the NI_NUMERICHOST flag.

8.4.4. inet_addr() Function

Applications that use the `inet_addr()` function must be changed to use the `getaddrinfo()` function, as follows:

AF_INET Call	**AF_INET6 Call**
result=inet_addr(*&string*)	err=getaddrinfo(*nodename*, *servname*, *&hints*, *&res*);...freeaddrinfo(*&ai*);

Make the following change to your application, as needed:

1. Change the function name from `inet_addr()` to `getaddrinfo()` and provide a character string that contains the node name, a character string that contains the service name to use, a pointer to a hints structure that contains the AI_NUMERICHOST option, and a pointer to an `addrinfo` structure or structures for the returned address information.

2. Add a call to the `freeaddrinfo()` routine to free the `addrinfo` structure or structures when your application is finished using them.

8.5. Other Application Changes

In addition to the name changes, you should review your code for specific uses of IP address information and variables.

8.5.1. Comparing IP Addresses

If your application compares IP addresses or tests IP addresses for equality, the in6_addr structure changes (see in Section 8.3.1) will change the comparison of *int* quantities to a comparison of structures. This will break the code and cause compiler errors.

Make either of the following changes to your application, as needed:

AF_INET Code	**AF_INET6 Code**
(addr1->s_addr == addr2->s_addr)	(memcmp(addr1, addr2, sizeof(struct in6_addr)) == 0)

1. Change the equality expression to one that uses the `memcmp` (memory comparison) function.

AF_INET Code	**AF_INET6 Code**
(addr1->s_addr == addr2->s_addr)	IN6_ARE_ADDR_EQUAL(addr1, addr2)

1. Change the equality expression to one that uses the `IN6_ARE_ADDR_EQUAL` macro.

8.5.2. Comparing an IP Address to the Wildcard Address

If your application compares an IP address to the wildcard address, the in6_addr structure changes (see Section 8.3.1) will change the comparison of int quantities to a comparison of structures. This will break the code and cause compiler errors.

Make either of the following changes to your application, as needed:

AF_INET Code	**AF_INET6 Code**
(addr->s_addr == INADDR_ANY)	IN6_IS_ADDR_UNSPECIFIED(addr)

1. Change the equality expression to one that uses the `IN6_IS_ADDR_UNSPECIFIED` macro.

AF_INET Code	**AF_INET6 Code**
(addr->s_addr == INADDR_ANY)	(memcmp(addr, in6addr_any, sizeof(struct in6_addr)) == 0)

1. Change the equality expression to one that uses the `memcmp` (memory comparison) function.

8.5.3. Using int Data Types to Hold IP Addresses

If your application uses int data types to hold IP addresses, the in6_addr structure changes (see Section 8.3.1) will change the assignment. This will break the code and cause compiler errors.

Make the following changes to your application, as needed:

AF_INET Code	**AF_INET6 Code**
struct in_addr foo;int bar;...bar = foo.s_addr;	struct in6_addr foo;struct in6_addr bar;...bar = foo;

1. Change the data type for `bar` from `int` to a struct `in6_addr`.
2. Change the assignment statement for `bar` to remove the `s_addr` field reference.

8.5.4. Using Functions that Return IP Addresses

If your application uses functions that return IP addresses as int data types, the in6_addr structure changes (see Section 8.3.1 will change the destination of the return value from an int to an array of char. This will break the code and cause compiler errors.

Make the following changes to your application, as needed:

AF_INET Code	**AF_INET6 Code**
struct in_addr *addr;addr->s_addr = foo(*xxx*);	struct in6_addr * *addr*;foo(*xxx, addr*);

1. Restructure the function to enable you to pass the address of the structure in the call. In addition, modify the function to write the return value into the structure pointed to by `addr`.

8.5.5. Changing Socket Options

If your application uses IPv4 IP-level socket options, change them to the corresponding IPv6 options.

8.6. Sample Client/Server Programs

This section contains sample client and server programs that demonstrate the differences between IPv4 and IPv6 coding conventions:

- Section 8.6.1 contains sample programs using IPv4 AF_INET sockets.
- Section 8.6.2 contains sample programs using IPv6 AF_INET6 sockets.

To build the examples, use the following commands:

```
$ CC/DEFINE=(_SOCKADDR_LEN)/INCLUDE=TCPIP$EXAMPLES: client.c
$ LINK client, TCPIP$LIBRARY:TCPIP$LIB/LIBRARY

$ CC/DEFINE=(_SOCKADDR_LEN)/INCLUDE=TCPIP$EXAMPLES: server.c
$ LINK server, TCPIP$LIBRARY:TCPIP$LIB/LIBRARY
```

8.6.1. Programs Using AF_INET Sockets

This section contains a client and a server program that use AF_INET sockets.

8.6.1.1. Client Program

The following is a sample client program that you can build, compile and run on your system. The program sends a request to and receives a response from the system specified on the command line.

```
#include <in.h>          /* define internet related constants,   */
                                    /* functions, and structures     */
#include <inet.h>        /* define network address info     */

#include <netdb.h>       /* define network database library info */

#include <socket.h>      /* define BSD 4.x socket api     */
#include <stdio.h>       /* define standard i/o functions     */
#include <stdlib.h>      /* define standard library functions    */
#include <string.h>      /* define string handling functions     */

#include <unixio.h>      /* define unix i/o       */


#define BUFSZ           1024           /* user input buffer size     */
#define SERV_PORTNUM    12345          /* server port number      */


int  main( void );                     /* client main    */
void get_serv_addr( void * );❶     /* get server host address    */


int
main( void )
{
    int sockfd;                     /* connection socket descriptor    */

    char buf[512];                  /* client data buffer    */

    struct sockaddr_in serv_addr;❷  /* server socket address structure */

    memset( &serv_addr, 0, sizeof(serv_addr) );❸
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port   = htons( SERV_PORTNUM );
    get_serv_addr( &serv_addr.sin_addr );❹

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )❺
 {
 perror( "Failed to create socket" );
 exit( EXIT_FAILURE );
 }
```

```
    printf( "Initiated connection to host: %s, port: %d\n",
     inet_ntoa(serv_addr.sin_addr), ntohs(serv_addr.sin_port)❻
   );

    if ( connect(sockfd,
   (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )❼
 {
 perror( "Failed to connect to server" );
 exit( EXIT_FAILURE );
 }

    if ( recv(sockfd, buf, sizeof(buf), 0) < 0 )
 {
 perror( "Failed to read data from server connection" );
 exit( EXIT_FAILURE );
 }

  printf( "Data received: %s\n", buf ); /* output client's data buffer */

    if ( shutdown(sockfd, 2) < 0 )
 {
 perror( "Failed to shutdown server connection" );
 exit( EXIT_FAILURE );
 }

    if ( close(sockfd) < 0 )
 {
 perror( "Failed to close socket" );
 exit( EXIT_FAILURE );
 }

    exit( EXIT_SUCCESS );
}

void
get_serv_addr( void *addrptr )❽
{
    char buf[BUFSZ];        /* input data buffer       */
    struct in_addr val;     /* remote host address structure  */
    struct hostent *host;   /* remote host hostent structure  */

    while ( TRUE )
 {
 printf( "Enter remote host: " );

 if ( fgets(buf, sizeof(buf), stdin) == NULL )
     {
     printf( "Failed to read User input\n" );
     exit( EXIT_FAILURE );
     }

 buf[strlen(buf)-1] = 0;

 val.s_addr = inet_addr( buf );

 if ( val.s_addr != INADDR_NONE )
     {
```

```
     memcpy( addrptr, &val, sizeof(struct in_addr) );
     break;
     }

 if ( (host = gethostbyname(buf)) )❾
     {
     memcpy( addrptr, host->h_addr, sizeof(struct in_addr) );
     break;
     }
 }
}
```

❶ Function code prototype for server host address/name translation function.
❷ Declares `sockaddr_in` structure.
❸ Clears the server `sockaddr_in` structure and sets values for fields of the structure
❹ Calls `get_serv_addr` passing a pointer to the socket address structure's `sin_addr` field.
❺ Creates an AF_INET socket
❻ Calls `inet_ntoa` to convert the server address to a text string.
❼ Calls `connect` passing a pointer to the `sockaddr_in` structure.
❽ Retrieves the server host's address from the user and then stores it in the server's socket address structure. The user can specify a server host by using either an IPv4 address in dotted decimal notation or a host domain name
❾ Calls `gethostbyname()` to retrieve the server host's address.

8.6.1.2. Server Program

The following is a sample server program that you can build, compile, and run on your system. The program receives requests from and sends responses to client programs on other systems.

```
#include <in.h>          /* define internet related constants,   */
                                    /* functions, and structures     */
#include <inet.h>        /* define network address info     */

#include <netdb.h>       /* define network database library info */

#include <socket.h>      /* define BSD 4.x socket api     */
#include <stdio.h>       /* define standard i/o functions     */
#include <stdlib.h>      /* define standard library functions    */
#include <string.h>      /* define string handling functions     */

#include <unixio.h>      /* define unix i/o       */


#define SERV_BACKLOG 1              /* server backlog      */
#define SERV_PORTNUM 12345          /* server port number      */


int  main( void );                     /* server main       */


int
main( void )
{
    int optval = 1;                 /* SO_REUSEADDR'S option value (on) */

    int conn_sockfd;                /* connection socket descriptor     */
    int listen_sockfd;              /* listen socket descriptor     */
    unsigned int client_addrlen; /* returned length of client socket */
                                 /* address structure */
  struct sockaddr_in client_addr;❶ /* client socket address structure */
    struct sockaddr_in serv_addr;    /* server socket address structure */

    struct hostent *host;❷           /* host name structure */

    char buf[] = "Hello, world!";       /* server data buffer */

    memset( &client_addr, 0, sizeof(client_addr) );

    memset( &serv_addr, 0, sizeof(serv_addr) );❸
    serv_addr.sin_family  = AF_INET;
    serv_addr.sin_port    = htons( SERV_PORTNUM );
    serv_addr.sin_addr.s_addr = INADDR_ANY;

    if ( (listen_sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )❹
        {
        perror( "Failed to create socket" );
        exit( EXIT_FAILURE );
        }

    if ( setsockopt(listen_sockfd,
      SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)) < 0 )
        {
        perror( "Failed to set socket option" );
        exit( EXIT_FAILURE );
        }

    if ( bind(listen_sockfd,
       (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
        {
        perror( "Failed to bind socket" );
        exit( EXIT_FAILURE );
        }

    if ( listen(listen_sockfd, SERV_BACKLOG) < 0 )
        {
        perror( "Failed to set socket passive" );
        exit( EXIT_FAILURE );
        }

    printf( "Waiting for a client connection on port: %d\n",
     ntohs(serv_addr.sin_port)
          );

    client_addrlen = sizeof(client_addr);

    conn_sockfd = accept( listen_sockfd,
                          (struct sockaddr *) &client_addr,
                          &client_addrlen
                        );
    if ( conn_sockfd < 0 )
        {
        perror( "Failed to accept client connection" );
        exit( EXIT_FAILURE );
        }

    host = gethostbyaddr( (char *)&client_addr.sin_addr.s_addr,
                          sizeof(client_addr.sin_addr.s_addr), AF_INET❺
                          );

    if ( host == NULL )
        {
        perror( "Failed to translate client address\n" );
        exit( EXIT_FAILURE );
        }

    printf( "Accepted connection from host: %s (%s), port: %d\n",
            host->h_name, inet_ntoa(client_addr.sin_addr),
            ntohs(client_addr.sin_port)
          );

    if ( send(conn_sockfd, buf, sizeof(buf), 0) < 0 )
        {
        perror( "Failed to write data to client connection" );
        exit( EXIT_FAILURE );
        }

    printf( "Data sent: %s\n", buf );    /* output server's data buffer  */

    if ( shutdown(conn_sockfd, 2) < 0 )
        {
        perror( "Failed to shutdown client connection" );
        exit( EXIT_FAILURE );
        }

    if ( close(conn_sockfd) < 0 )
        {
        perror( "Failed to close socket" );
        exit( EXIT_FAILURE );
        }

    if ( close(listen_sockfd) < 0 )
        {
        perror( "Failed to close socket" );
        exit( EXIT_FAILURE );
        }

    exit( EXIT_SUCCESS );
}
```

❶ Declares `sockaddr_in` structures.
❷ Declares `hostent` structure.
❸ Clears the server `sockaddr_in` structure and sets values for fields of the structure.
❹ Creates an AF_INET socket.
❺ Calls `gethostbyaddr()` to retrieve client name.

8.6.2. Programs Using AF_INET6 Sockets

This section contains a client and a server program that use AF_INET6 sockets.

8.6.2.1. Client Program

The following is a sample client program that you can build, compile and run on your system. The program sends a request to and receives a response from the system specified on the command line.

```
#include <in.h>             /* define internet related constants,   */
                                       /* functions, and structures     */
#include <inet.h>           /* define network address info     */

#include <netdb.h>          /* define network database library info */

#include <socket.h>         /* define BSD 4.x socket api     */
#include <stdio.h>          /* define standard i/o functions     */
#include <stdlib.h>         /* define standard library functions    */
#include <string.h>         /* define string handling functions     */

#include <unixio.h>         /* define unix i/o       */


#define BUFSZ           1024          /* user input buffer size     */
#define SERV_PORTNUM    "12345"       /* server port number string     */


int main( void );                     /* client main       */
void get_serv_addr( struct addrinfo *hints, struct addrinfo **res );❶
                                      /* get server host address */

int
main( void )
{
    int sockfd;                          /* connection socket descriptor */

    char buf[512];                       /* client data buffer */

    struct addrinfo hints;     /* input values to direct operation */
    struct addrinfo *res;❷     /* linked list of addrinfo structs */

    memset( &hints, 0, sizeof(hints) );❸
    hints.ai_family = AF_INET6;
    hints.ai_flags = AI_ADDRCONFIG | AI_V4MAPPED | AI_CANONNAME;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_socktype = SOCK_STREAM;
    get_serv_addr( &hints, &res );❹

    if ( (sockfd = socket(AF_INET6, SOCK_STREAM, 0)) < 0 )❺
 {
 perror( "Failed to create socket" );
 exit( EXIT_FAILURE );
 }

    printf( "Initiated connection to host: %s, port: %d\n",
     res->ai_canonname,
     htons(((struct sockaddr_in6 *)res->ai_addr)->sin6_port)❻
   );

    if ( connect(sockfd, res->ai_addr, res->ai_addrlen) < 0 )❼
 {
```

```
 perror( "Failed to connect to server" );
 exit( EXIT_FAILURE );
 }

    if ( recv(sockfd, buf, sizeof(buf), 0) < 0 )
 {
 perror( "Failed to read data from server connection" );
 exit( EXIT_FAILURE );
 }

  printf( "Data received: %s\n", buf ); /* output client's data buffer */

    if ( shutdown(sockfd, 2) < 0 )
 {
 perror( "Failed to shutdown server connection" );
 exit( EXIT_FAILURE );
 }

    if ( close(sockfd) < 0 )
 {
 perror( "Failed to close socket" );
 exit( EXIT_FAILURE );
 }

    exit( EXIT_SUCCESS );
}

void
get_serv_addr( struct addrinfo *hints, struct addrinfo **res )❽
{
    int gai_error;   /* return value of getaddrinfo()  */
    char buf[BUFSZ];         /* input data buffer       */
    const char *port = SERV_PORTNUM;    /* server port number  */

    while ( TRUE )
        {
 printf( "Enter remote host: " );

 if ( fgets(buf, sizeof(buf), stdin) == NULL )
     {
     printf( "Failed to read User input\n" );
     exit( EXIT_FAILURE );
     }

 buf[strlen(buf)-1] = 0;

 gai_error = getaddrinfo( buf, port, hints, res );❾
 if ( gai_error )
            printf( "Failed to resolve name or address: %s\n",
                    gai_strerror(gai_error)❿
                  );
 else
     break;
 }
}
```

❶ Function prototype for server host address/name translation function.

❷ Declares `addrinfo` structures.

❸ Clears the `addrinfo` structure and sets values for fields of the structure.
❹ Calls `get_serv_addr()` passing pointers to the input and output `addrinfo` structures.
❺ Creates an AF_INET6 socket.
❻ Uses values from the output `addrinfo` structure for host name and port.
❼ Calls `connect()` using values from the output `addrinfo` structure.
❽ Retrieves the server host's address from the user and stores it in the `addrinfo` structure. The user can specify a server host by using any of the following:

- An IPv4 address in dotted-decimal notation
- An IPv6 address in hexadecimal
- An Ipv4-mapped IPv6 address in hexadecimal
- A host domain name

❾ Calls `getaddrinfo()` to retrieve the server host's name or address.
❿ Calls `gai_strerror()` to convert one of the EAI_xx return values to a string describing the error.

8.6.2.2. Server Program

The following is a sample server program that you can build, compile, and run on your system. The program receives requests from and sends responses to client programs on other systems.

```
#include <in.h>             /* define internet related constants,
                            /* functions, and structures     */
#include <inet.h>           /* define network address info     */

#include <netdb.h>          /* define network database library info */

#include <socket.h>         /* define BSD 4.x socket api */
#include <stdio.h>          /* define standard i/o functions */
#include <stdlib.h>         /* define standard library functions */
#include <string.h>         /* define string handling functions */

#include <unixio.h>         /* define unix i/o */


#define SERV_BACKLOG    1             /* server backlog      */
#define SERV_PORTNUM    12345         /* server port number  */


int  main( void );                    /* server main         */

int
main( void )
{
    int optval = 1;       /* SO_REUSEADDR'S option value (on) */

    int conn_sockfd;              /* connection socket descriptor    */
    int listen_sockfd;            /* listen socket descriptor    */
    int gni_error;❶    /* return status for getnameinfo() */

    unsigned int client_addrlen;     /* returned length of client socket */
                                     /* address structure      */
    struct sockaddr_in6 client_addr; /* client socket address structure */
    struct sockaddr_in6 serv_addr;❷  /* server socket address structure */
```

```
   char buf[] = "Hello, world!";   /* server data buffer */
   char node[NI_MAXHOST]; ❸        /* buffer to receive node name */
   char port[NI_MAXHOST];          /* buffer to receive port number    */
   char addrbuf[INET6_ADDRSTRLEN]; /* buffer to receive host's address */

   memset( &client_addr, 0, sizeof(client_addr) );

   memset( &serv_addr, 0, sizeof(serv_addr) );❹
   serv_addr.sin6_family      = AF_INET6;
   serv_addr.sin6_port        = htons( SERV_PORTNUM );
   serv_addr.sin6_addr        = in6addr_any;

   if ( (listen_sockfd = socket(AF_INET6, SOCK_STREAM, 0)) < 0 )❺
{
perror( "Failed to create socket" );
exit( EXIT_FAILURE );
}

   if ( setsockopt(listen_sockfd,
     SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)) < 0 )
{
perror( "Failed to set socket option" );
exit( EXIT_FAILURE );
}

   if ( bind(listen_sockfd,
      (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
{
perror( "Failed to bind socket" );
exit( EXIT_FAILURE );
}

   if ( listen(listen_sockfd, SERV_BACKLOG) < 0 )
{
perror( "Failed to set socket passive" );
exit( EXIT_FAILURE );
}

   printf( "Waiting for a client connection on port: %d\n",
    ntohs(serv_addr.sin6_port)
  );

   client_addrlen = sizeof(client_addr);

   conn_sockfd = accept( listen_sockfd,
    (struct sockaddr *) &client_addr,
    &client_addrlen
  );
   if ( conn_sockfd < 0 )
{
perror( "Failed to accept client connection" );
exit( EXIT_FAILURE );
}

   gni_error = getnameinfo( (struct sockaddr *)&client_addr,
client_addrlen,
❻
        node, sizeof(node), NULL, 0, NI_NAMEREQD
      );
    if ( gni_error )
 {
 printf( "Failed to translate client address: %s\n",
  gai_strerror(gni_error) ❼
       );
 exit( EXIT_FAILURE );
 }

    gni_error = getnameinfo( (struct sockaddr *)&client_addr,
 client_addrlen,
        addrbuf, sizeof(addrbuf), port, sizeof(port),
        NI_NUMERICHOST | NI_NUMERICSERV ❽
      );
    if ( gni_error )
 {
 printf( "Failed to translate client address and/or port: %s\n",
  gai_strerror(gni_error)
       );
 exit( EXIT_FAILURE );
 }

    printf( "Accepted connection from host: %s (%s), port: %s\n",
     node, addrbuf, port
   );

    if ( send(conn_sockfd, buf, sizeof(buf), 0) < 0 )
 {
 perror( "Failed to write data to client connection" );
 exit( EXIT_FAILURE );
 }

    printf( "Data sent: %s\n", buf ); /* output server's data buffer */

    if ( shutdown(conn_sockfd, 2) < 0 )
 {
 perror( "Failed to shutdown client connection" );
 exit( EXIT_FAILURE );
 }

    if ( close(conn_sockfd) < 0 )
 {
 perror( "Failed to close socket" );
 exit( EXIT_FAILURE );
 }

    if ( close(listen_sockfd) < 0 )
 {
 perror( "Failed to close socket" );
 exit( EXIT_FAILURE );
 }

    exit( EXIT_SUCCESS );
}
```

❶ Declares variable for `getnameinfo()` return value.

❷ Declares `sockaddr_in6` structures

❸ Declares buffers to receive client's name, port number, and address for calls to `getnameinfo()`.

❹ Clears the server `sockaddr_in6` structure and sets values for fields of the structure.

❺ Creates an AF_INET6 socket.

❻ Calls `getnameinfo()` to retrieve client name. This is for message displaying purposes only and is not necessary for proper functioning of the server.

❼ Calls `gai_strerror()` to convert one of the EAI_xxx return values to a string describing the error.

❽ Calls `getnameinfo()` to retrieve client address and port number. This is for message displaying purposes only and is not necessary for proper functioning of the server.

8.6.3. Sample Program Output

This section contains sample output from the preceding server and client programs. The server program makes and receives all requests on an `AF_INET6` socket using `sockaddr_in6`. For requests received over IPv4, sockaddr_in6 contains an IPv4-mapped IPv6 address.

The following example shows a client program running on node `hostb6` and sending a request to node `hosta6`. The program uses an `AF_INET6` socket. The node `hosta6` has the IPv6 address `3ffe:1200::a00:2bff:fe97:7be0` in the Domain Name System (DNS).

```
$ run client.exe
Enter remote host: hosta6
Initiated connection to host: hosta6.ipv6.corp.example, port: 12345
Data received: Hello, world!
$
```

On the server node, the following example shows the server program invocation and the request received from the client node `hostb6`:

```
$ run server.exe
Waiting for a client connection on port: 12345
Accepted connection from host: hostb6.ipv6.corp.example
(3ffe:1200::a00:2bff:fe97:7be0), port: 49174
Data sent: Hello, world!
$
```

The following example shows the client program running on node `hostb` and sending a request to node `hosta`. The program uses an `AF_INET6` socket. The `hosta` node has only an IPv4 address in the DNS.

```
$ run client.exe
Enter remote host: hosta
Initiated connection to host: hosta.corp.example, port 12345
Data received: Hello, world!
$
```

On the server node, the following example shows the server program invocation and the request received from the client node `hostb`:

```
$ run server.exe
Waiting for a client connection on port: 12345
Accepted connection from host: hostb.corp.example (::ffff:10.10.10.251),
 port: 49175
Data sent: Hello, world!
```

```
$
```

The following example shows the client program running on node hostb6 and sending a request to node `hosta6` using its link-local address `fe80::a00:2bff:fe97:7be0`. The program uses an `AF_INET6` socket.

```
$ run client.exe
Enter remote host: fe80::a00:2bff:fe97:7be0
Initiated connection to host: fe80::a00:2bff:fe97:7be0, port: 12345
Data received: Hello, world!
$
```

On the server node, the following example shows the server program invocation and the request received from the client node `hostb6`.

```
$ run server.exe
Waiting for a client connection on port: 12345
Accepted connection from host: hosta6.ipv6.corp.example%WE0
(fe80::a00:2bff:fe97:7be0%WE0), port: 49177
Data sent: Hello, world!
$
```

Appendix A. Supported IPv6 RFCs

The following are supported IPV6 Request for Comments (RFCs):

- Internet Protocol Version 6 (IPv6) Specification, RFC 2460 (December 1998)
- Internet Control Message Protocol (ICMPv6) for Internet Protocol Version 6 (IPv6), RFC 2463 (December 1998)
- Neighbor Discovery for IP Version 6 (IPv6), RFC 2461 (December 1998)
- IPv6 Stateless Address Autoconfiguration, RFC 2462 (December 1998)
- Path MTU Discovery for IP Version 6, RFC 1981 (August 1996)
- Transition Mechanisms for IPv6 Hosts and Routers, RFC 1933 (April 1996)
- IP Version 6 Addressing Architecture, RFC 3513 (April 2003)
- An IPv6 Aggregatable Global Unicast Address Format, RFC 2374 (July 1998)
- IPv6 Testing Address Allocation, RFC 2471 (December 1998)
- Transmission of IPv6 Packets over Ethernet Networks, RFC 2464 (December 1998)
- Transmission of IPv6 Packets over FDDI Networks, RFC 2467 (December 1998)
- Basic Socket Interface Extensions for IPv6, RFC 3493 (February 2003)
- Advanced Sockets API for IPv6, RFC 3542 (May 2003)
- DNS Extensions to Support IP version 6, RFC 1886 (December 1995)
- Dynamic Updates in the Domain Name System (DNS UPDATE), RFC 2136 (April 1997)
- RIPng, RFC 2080 (January 1997)

Appendix B. Deprecated Library Functions

This appendix describes deprecated library functions that were provided in previous Early Adopter Kits (EAKs). Do not use these functions if you are developing new applications. If your existing applications use these functions, see Chapter 8 for changes you should make to your code.

The following table shows the deprecated functions and their replacements:

Deprecated Function	**Replacement Function**
getipnodebyname	getaddrinfo
getipnodebyaddr	getnameinfo
freehostent	freeaddrinfo

B.1. getipnodebyname Function

The `getipnodebyname` function has the following syntax:

```
#include <netdb.h>
struct hostent *getipnodebyname(
        const char *name,
        int addr_family,
        int flags,
        int *error_num );
```

Parameters:

- **name**

  Specifies the official network node name, alias, or numeric node address (for example, an IPv4 dotted-decimal address or an IPv6 hexadecimal address).

- **addr_family**

  Specifies the address family. This can be AF_INET for IPv4 addresses or AF_INET6 for IPv6 addresses.

- **flags**

  Specifies the type of addresses for which to search and the types of addresses that are returned. Table B.1 describes how the processing is affected by the values of the **af** parameter and commonly used flag values.

- **error_num**

  Specifies an error return code value if the function is not successful.

Description

The `getipnodebyname()` routine is an evolution of the `gethostbyname()` routine that enables name lookups in address families other than AF_INET.

The `getipnodebyname()` routine returns a pointer to a structure of type `hostent`. Its members specify data obtained from the local TCPIP$ETC:IPNODES.DAT file, TCPIP$HOSTS.DAT file or from one of the files distributed by DNS/BIND.

If multiple addresses are found, the `h_addr_list` field in the `hostent` structure contains the addresses.

The `<netdb.h>` header file defines the `hostent` structure.

If you are using DNS/BIND, the information is obtained from a name server as configured. When the name server is not running, the `getipnodebyname()` routine searches both the local TCPIP $ETC:IPNODES.DAT name file for IPv6 and IPv4 addresses and the hosts name file for IPv4 addresses, if the addresses not are found in the TCPIP$ETC:IPNODES.DAT file.

Table B.1 lists the flags parameters and how the processing is affected by the value of the *af* parameters.

Table B.1. Node Name to Address Processing

Flag Value	**af Value is AF_NET**	**af Value is AF_INET6**
0	Searches for A records. If found, returns IPv4 addresses (h_length=4). If not, returns a NULL pointer. Provides backward compatibility for existing IPv4 applications.	Searches for AAAA records. If found, returns IPv6 records (h_length=16). If not, returns a NULL pointer.
AI_V4MAPPED	Ignored.	Searches for AAAA records. If found, returns IPv6 records (h_length=16). If not, searches for A records. If A records are found, returns IPv4-mapped IPv6 addresses (h_length=16). If no A records are found, returns a NULL pointer.
AI_ALL | AI_V4MAPPED	Ignored.	Searches for AAAA records. If found, returns IPv6 addresses (h_length=16). Then searches for A records. If A records are found, returns IPv4-mapped IPv6 addresses (h_length=16). If no A records are found, returns a NULL pointer.

All flags can be used in any combination to achieve finer control of the translation process. The AI_ADDRCONFIG flag is typically used in combination with other flags to modify the search based on the source address or addresses configured on the system. Table B.2 describes how the AI_ADDRCONFIG flag works by itself.

Table B.2. AI_ADDRCONFIG Flag

Flag Value	af Value is AF_NET	af Value is AF_INET6
AI_ADDRCONFIG	Searches for A records only if an IPv4 source address is configured on the system.	Searches for AAAA records only if an IPv6 source address is configured on the system. Searches for A records only if an IPv4 source address is configured on the system.

Most applications will use a combination of the AI_ADDRCONFIG and AI_V4MAPPED flags to control their search. To simplify this for the programmer, the AI_DEFAULT symbol, which is a logical OR of AI_ADDRCONFIG and AI_V4MAPPED, is defined. Table B.3 describes how AI_DEFAULT directs the search.

Table B.3. AI_DEFAULT Flag

Flag Value	af Value is AF_NET	af Value is AF_INET6
AI_DEFAULT	Searches for A records only if an IPv4 source address is configured on the system. If found, returns IPv4 addresses (h_length=4). If not, returns a NULL pointer.	Searches for AAAA records only if an IPv6 source address is configured on the system. If found, returns IPv6 records (h_length=16). If not found and if an IPv4 address is configured on the system, searches for A records. If A records are found, returns IPv4-mapped IPv6 addresses (h_length=16). If no A records are found, returns a NULL pointer.

The `hostent` structure returned by the `getipnodebyname` function is dynamically allocated. You should free this structure and dynamic storage by using the `freehostent` function (see Section B.3).

Errors

If the `getipnodebyname()` routine call fails, **error_num** is set to one of the following values:

- HOST_NOT_FOUND

  The name you have used is not an official node name or alias; another type of name server request may be successful.

- NO_ADDRESS

The server recognized the request and the name, but no address is available for the name. Another type of name server request may be successful.

- NO_RECOVERY

  An unexpected server failure occurred. This is a nonrecoverable error.

- TRY_AGAIN

  A transient error occurred, for example, the server did not respond. A retry at some later time may be successful.

B.2. getipnodebyaddr Function

The `getipnodebyaddr` function has the following syntax:

```
#include <netdb.h>

struct hostent *getipnodebyaddr(
          const void *src,
          size_t len,
          int af,
          int *error_num);
```

Parameters

- **src**

  Specifies an Internet address in network order.

- **len**

  Specifies the number of bytes in an Internet address.

- **af**

  Specifies the Internet domain address format. Valid values are AF_INET and AF_INET6.

- **error_num**

  Specifies an error return code value if the function is not successful.

Description

The `getipnodebyaddr()` routine is an evolution of the `gethostbyaddr()` routine that enables address lookups in address families other than AF_INET.

The `getipnodebyaddr()` routine returns a pointer to a structure of type `hostent`. Its members specify data obtained from the local TCPIP$ETC:IPNODES.DAT file, the TCPIP$HOSTS.DAT file, or one of the files distributed by DNS/BIND.

The `getipnodebyaddr()` routine searches the network host database sequentially until a match with the **src** and **af** parameters occurs. The **len** parameter must specify the number of bytes in an Internet address. The **src** parameter must specify the address in network order. The **af** parameter can be either the constant AF_INET or AF_INET6, which specifies the IPv4 address format or the IPv6 address format, respectively. When EOF (end-of-file) is reached without a match, an error value is returned.

If the **src** parameter is either an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address, the routine performs the following steps:

1. If the **af** parameter is AF_INET6, the **len** parameter is 16, and the **src** parameter is either an IPV4-mapped IPv6 address or an IPv4-compatible IPv6 address, the routine skips the first 12 bytes of the address, sets **af** to AF_INET and **len** to 4.

2. If the **af** parameter is AF_INET, the routine queries for a PTR record in the `in-addr.arpa` domain.

3. If the **af** parameter is AF_INET6, the routine queries for a PTR record in the `ip6.int` domain.

4. If the routine returns success, the single address and address family returned in the `hostent` structure are copies of the **src** parameter and the **af** parameter, respectively, that were passed to the routine.

Note

The double colon (::) and ::1 IPv6 addresses are not considered IPv4-compatible addresses.

If you are using DNS/BIND, the address is obtained from a name server as configured. When the name server is not running, the `getipnodebyaddr()` routine searches the local TCPIP $ETC:IPNODES.DAT name file for IPv6 and IPv4 addresses and the hosts name file for IPv4 addresses, if the addresses are not found in the TCPIP$ETC:IPNODES.DAT file.

The `getipnodebyaddr()` routine dynamically allocates the `hostent` structure. Use the `freehostent()` routine to free the allocated memory. (See Section B.3.

Errors

If the `getipnodebyaddr()` routine call fails, **error_num** is set to one of the following the values:

- HOST_NOT_FOUND

  The name you have used is not an official node name or alias; another type of name server request may be successful.

- NO_ADDRESS

  The server recognized the request and the name, but no address is available for the name. Another type of name server request may be successful.

- NO_RECOVERY

  An unexpected server failure occurred. This is a nonrecoverable error.

- TRY_AGAIN

  A transient error occurred, for example, the server did not respond. A retry at some later time may be successful.

B.3. freehostent Function

The `freehostrent` function returns `hostent` structures and dynamic storage to the system. You should use this function to free `hostent` structures and storage that were returned by `getipnodebyname` and `getipnodebyaddr`.

This function has the following syntax:

```
void freehostent(
        struct hostent *ptr );
```

The **ptr** parameter is a pointer to the `hostent` structure to be freed.

Note

Do not use the `freehostent` function with `hostent` structures returned by `gethostbyname` and `gethostbyaddr`.