HP-UX IPv6 Porting Guide

HP-UX 11i v2 September 2004

Manufacturing Part Number : 5990-8371

August 2004

United States

Legal Notices

The information in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty

A copy of the specific warranty terms applicable to your Hewlett- Packard product and replacement parts can be obtained from your local Sales and Service Office.

U.S. Government License

Proprietary computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

Trademark Notices

UNIX® is a registered trademark in the United States and other countries, licensed exclusively through The Open Group.

Intel® and Itanium® are registered trademarks of Intel Corporation.

Contents

Contents

Contents

About This Document

This document is intended to help HP-UX BSD Sockets Application Programmers port IPv4 network applications to IPv6.

The document printing date and part number indicate the document's current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. The document part number will change when extensive changes are made.

Document updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service. See your HP sales representative for details.

The latest version of this document can be found on line at: `docs.hp.com/hpux/netcom/index.html#IPv6`.

Intended Audience

This document is intended for HP-UX BSD Sockets Application Programmers porting IPv4 network applications to IPv6.

This document is not a tutorial.

What Is in This Docuent

The guide is organized as follows:

1 Introduction

2 IPv6 Addressing

3 Data Structure Changes

4 Migrating Applications from IPv4 to IPv6

5 Overview of IPv6 and IPv4 Call Set-up

6 Function Calls Converting Names to Addresses

7 Function Calls Converting IP addresses to Names

8 Reading Error Messages

9 Freeing Memory

10 Converting Binary and Text Addresses

11 Testing for Scope and Type of IPv6 addresses using Macros

12 Identifying Local Interface Names and Indexes

13 Configuring or Querying an Interface using IPv6 ioctl() Function Calls

14 Verifying IPv6 Installation

15 Sample Client/Server Programs

Appendix A IPv4 to IPv6 Quick-Reference Guide

HP-UX Release Names and Release Identifiers

Each HP-UX 11i release has an associated release name and release identifier. The *uname* (1) command with the -r option returns the release identifier. This table shows the releases available for HP-UX 11i.

Table 1 HP-UX 11i Releases

Release Identifier	Release Name	Supported Processor Architecture
B.11.23	HP-UX 11i v2 September 2004	PA-RISC and Intel® Itanium®
B.11.23	HP-UX 11i v2	Intel® Itanium®

Table 1 HP-UX 11i Releases (Continued)

Release Identifier	Release Name	Supported Processor Architecture
B.11.22	HP-UX 11i v1.6	Intel® Itanium®
B.11.20	HP-UX 11i v1.5	Intel® Itanium®
B.11.11	HP-UX 11i v1	PA-RISC

Related Documents

HP Documentation

Additional information about HP-UX IPv6 transport can be found within *docs.hp.com* in the *networking and communications* collection under *IPv6* at:

`http://www.docs.hp.com/hpux/netcom/index.html#IPv6`

Other documents in this collection (besides this guide) include:

HP-UX IPv6 Transport Administrator's Guide (TOUR 1.0)

HP-UX IPv6 Transport Administrator's Guide (HP-UX 11i v2)

Other Documentation

For more information, refer to RFC 2533 “Basic Socket Interface Extensions for IPv6”. The IETF (Internet Engineering Task Force) RFCs can be located at: `http://www.ietf.org/rfc.html`.

HP Welcomes Your Comments

HP welcomes your comments concerning this document. HP is committed to providing documentation that meets your needs.

Please send comments to: `netinfo_feedback@cup.hp.com`

Please include document title, manufacturing part number, and any comment, error found, or suggestion for improvement you have concerning this document. Also, please tell us what you like, so we can incorporate it into other documents.

1 Introduction

This chapter provides a brief introduction, including comments about existing IPv4 applications, transitioning to IPv6, and some general terminology.

Why IPv6 Now?

In the last five years, the Internet has transformed the way people live. The Internet's tremendous growth rate greatly exceeded any futurist's predictions, including the Internet Protocol (IP) architect's plans from twenty years ago. IP version 4 (IPv4) provided ample addresses for network growth throughout the 1980s, but the address-supply is now low outside the United States. If current Internet growth rates continue, the prediction is that the supply of unassigned IPv4 addresses will be depleted within ten years. Internet Protocol Version 6 (IPv6) overcomes many limitations of IPv4.

For additional information on using HP-UX IPv6 transport, refer to the following documentation as needed:

HP-UX IPv6 Transport Administrator's Guide (TOUR 1.0)

HP-UX IPv6 Transport Administrator's Guide (HP-UX 11i v2)

Who Should Read This Guide

HP-UX BSD Sockets Application Programmers porting IPv4 network applications to IPv6.

Do Existing IPv4 Applications Require Changes?

No. Current IPv4 applications can remain unchanged. Modify applications only to take advantage of new IPv6 features.

Does implementing IPv6 require a complete transition from IPv4?

No. Networks can migrate to IPv6 gradually, using transition mechanisms defined by IPv6 Protocol Specifications. IPv4 and IPv6 will coexist for a long time. IPv6 Protocol Specifications provide two major transition mechanisms:

Dual Stack: Dual-stack hosts have both IPv4 and IPv6 interfaces configured and can communicate with both IPv4 and IPv6 hosts.

Tunneling: Tunneling is a mechanism that has been defined to allow IPv6 packets to be encapsulated in IPv4 packets. A Dual-Stack host can send IPv6 packets through an IPv4 tunnel to a remote IPv6 host, without requiring an IPv6 infrastructure.

Terminology

This section provides brief definitions of some common general IP and IPv6 terms.

General IP Terminology

Node: A device that implements IP (either IPv4 or IPv6 or both).

Router: A node that forwards IP packets not explicitly addressed to itself.

Host: Any node that is not a router.

Link: A logical connection between two nodes. Here, a link is the layer below IP such as Ethernet, PPP, or ATM networks. A link also includes IPv6 traffic encapsulated within IPv4 packets, also known as tunneling.

Name Service: A database that maps host names to IP addresses. Common Name Services are Domain Name System (DNS) or the `/etc/hosts` file.

Site: An organization's Intranet, perhaps geographically disbursed.

IPv6 Terminology

IPv4 Address: A 32-bit IPv4 address

IPv6 Address: An 128-bit IPv6 address

IPv4-only node: A node that implements only IPv4. An IPv4-only node does not understand IPv6.

IPv6-only node: A node configured for IPv6 only. An IPv6-only node does not understand IPv4.

IPv4/IPv6 node: A node that implements both IPv4 and IPv6.

IPv6 node: A node that implements IPv6. IPv4/IPv6 and IPv6-only nodes are both IPv6 nodes.

IPv4 node: A host that implements IPv4. IPv4/IPv6 and IPv4-only nodes are both IPv4 nodes.

2 IPv6 Addressing

This chapter describes basic IPv6 addressing information.

Types of IPv6 addresses

IPv6 supports both single-destination (unicast) and multiple-destination (multicast) addresses. Addresses comprise three different scopes.

IPv4 to IPv6 Transition Addresses

To ease the transition from IPv4 to IPv6, the IPv6 Protocol Specifications define two global IPv6 addresses containing unique IPv4 address in the low-order 32-bits of the IPv6 address.

IPv6 Address scope

Link-local: An IPv6 address used over one local link; assigned during autoconfiguration.

Site-local: An IPv6 address used inside a local Intranet site only; not renumbered by an ISP.

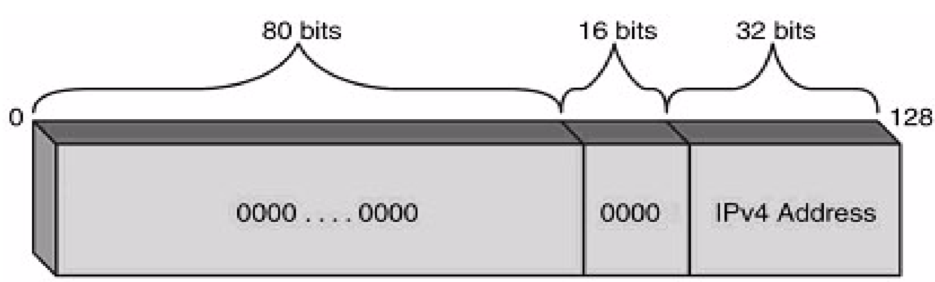
Global: An IPv6 address used throughout the Internet.

An IPv6 node always has a link-local address. It may have a site-local address or one or more global addresses.

IPv4-Compatible IPv6 Address

An administrator assigns an IPv4-compatible IPv6 address and host name entry to the Name Service for an IPv4/IPv6 host where no IPv6 router is available.The IPv4-compatible IPv6 address is an IPv6 address in the format:

Figure 2-1 IPv4-Compatible Address



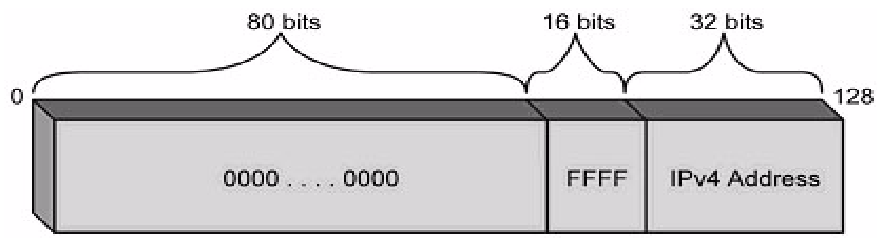
IPv4-Compatible Addresses help the migration process by enabling IPv6 features without requiring IPv6 Routers.

IPv4-Mapped Address

An IPv4-mapped IPv6 address enables an IPv6 application on an IPv4/IPv6 host to communicate with an IPv4-only node. IPv4-mapped IPv6 addresses are created internally by the Name Service resolver when an IPv6 application requests the host name for a node with an IPv4 address only.

The IPv6 module encodes the IPv4 address in the low-order 32 bits of the IPv6 address.

Figure 2-2 IPv4-Mapped Address



Comparing IPv4 and IPv6 Addresses

IPv4 addresses are 32-bit addresses represented as four dotted-decimal octets

Example: `10.1.3.7`

IPv6 Addresses are 128-bit records represented as eight fields of up to four hexadecimal digits. A colon separates each field (:).

Example: `8888:7777:6666:5555:4444:3333:2222:1111`

Leading Zeros Suppressed

Example: `0008:0007:0006:0005:0004:0003:0002:0001`

Is also valid in the format:

`8:7:6:5:4:3:2:1`

Comparing IPv4 and IPv6 Addresses

Contiguous Fields Containing only the Digits Zero can be collapsed

Example: `0008:0000:0000:0000:0000:0003:0002:0001`

Is also valid in the format:

`8::3:2:1`

NOTE Only one set of contiguous fields of zeros per IP address can be collapsed.

IPv4-Compatible and IPv4-Mapped IPv6 Addresses can display IPv4 Addresses in Dotted-Decimal Format

IPv4-compatible and IPv4-mapped addresses contain the IPv4 address in the low-order 32-bits. Mixing hexadecimal format and dotted-decimal format is valid. For example, the IPv4-compatible IPv6 address `::10.9.8.7` is valid in the following formats:

Table 2-1

`0::0a09:0807`	IPv4-compatible address
`::0a09:0807`	First zero removed
`::10.9.8.7`	Combined hex and decimal format

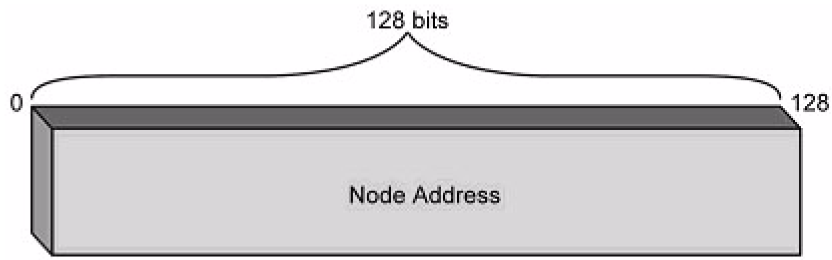
IPv6 addresses are classless, using Classless Internet Domain Registry CIDR format. The prefix follows the IPv6 address (<IPv6 addr>"/"<prefix>) and denotes the size of a subnet.

Example: `8:7:6:5:4:3:2:1/16`

IPv6 Address Types

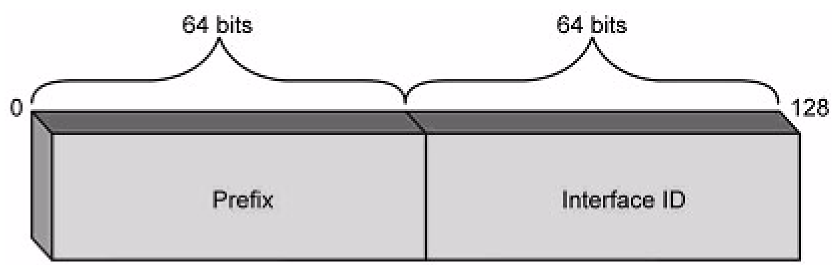
Unicast Address

Figure 2-3 Unicast Address



Unicast addresses usually comprise a 64-bit prefix and a 64-bit interface ID.

Figure 2-4 Unicast Prefix



The 64-bit interface ID must be unique on the link. An interface ID often includes the interface Link-Layer Address.

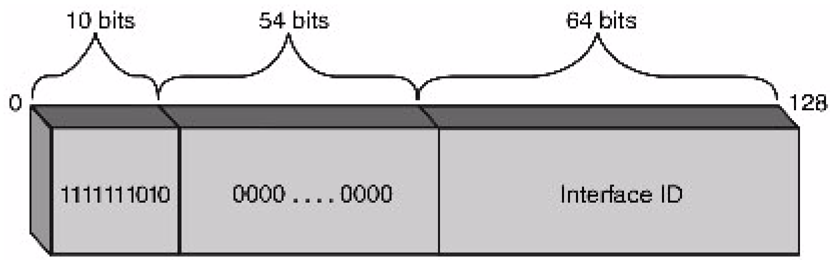
IPv6 Loopback Address

The loopback interface uses the IPv6 loopback address for self-testing, by sending IP datagrams to itself. The IPv6 loopback address is: `0:0:0:0:0:0:0:1` (or more simply, `::1`).

Link-local Unicast Address

The LAN segment is the scope of a Link-local Address, and is used for address autoconfiguration and neighbor discovery.

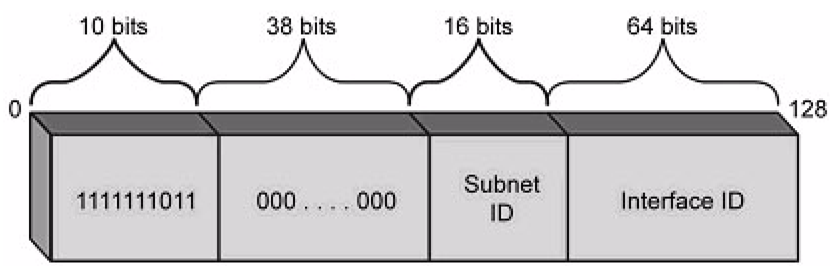
Figure 2-5 Link-Local Unicast Address



Site-local Addresses

Use site-local addresses on networks not connecting to the Internet.

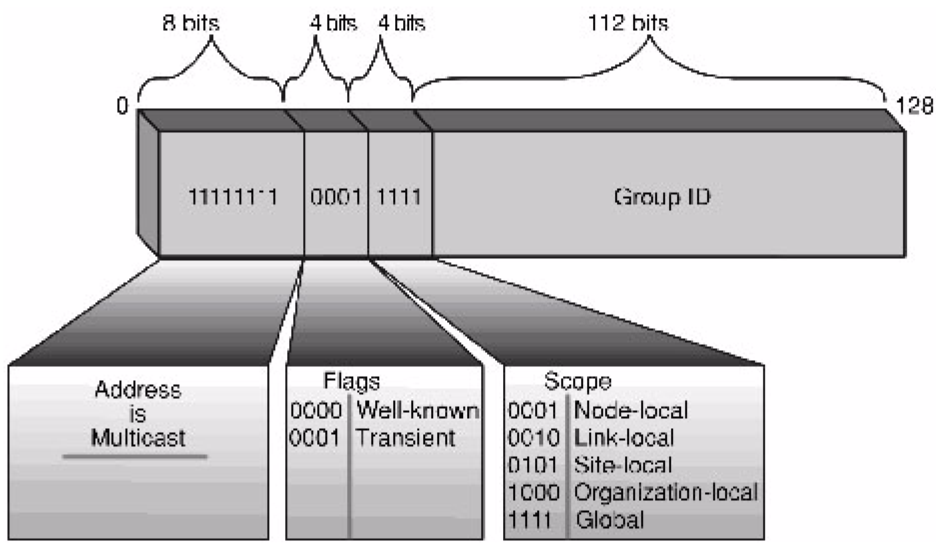
Figure 2-6 Site-local Unicast Address



IPv6 Multicast Addresses

IPv6 multicast addresses resemble IPv4 multicast addresses, but have an explicit field for address-scope.

Figure 2-7 Multicast Address Format



Some Well-know Multicast Addresses

`FF02::1` All nodes (link-local)

`FF02::2` All routers (link-local)

`FF02::9` All Routing Information Protocol next generation (RIPng) routers (link-local)

IPv6 Wildcard Addresses

In IPv4, an application can let the system choose which source IP address to bind to a socket by specifying a wildcard address: the symbolic constant `INADDR_ANY` in the `bind()` function call. In IPv6, because the IPv6 address type is a structure (`struct in6_addr`), a symbolic constant can initialize an IPv6 address structure variable, but cannot assign an IPv6 structure variable. Therefore, an IPv6 wildcard address requires two forms:

- For initialization, use the symbolic constant `IN6ADDR_ANY_INIT` of the type `struct in6_addr`. For example,
  `struct in6_addr anyaddr =` **`IN6ADDR_ANY_INIT;`**

NOTE Only use the constant during initialization.

- For assignment, use the global variable named in6addr_any, of the type in6_addr structure. For example:

Header file

```
<netinet/in.h>
extern const struct in6_addr in6addr_any;

struct sockaddr_in6 sin6;
…
sin6.sin6_addr = in6addr_any; /* structure assignment */
…
if (bind(s, (struct sockaddr *) &sin6, sizeof(sin6)) == -1)
```

IPv6 Loopback Addresses

The IPv4 loopback address is an integer type INADDR_LOOPBACK. The IPv6 loopback address is an in6_addr structure defined in <netinet/in.h>. For example:

Header file

```
<netinet/in.h>

sin6.sin6_addr = in6addr_loopback; /* structure assignment */
```

The symbolic constant named IN6ADDR_LOOPBACK_INIT is defined in <netinet/in.h>. Use it only when declaring a sockaddr_in6 struct. For example:

```
struct in6_addr loopbackaddr = IN6ADDR_LOOPBACK_INIT
```

NOTE IPv4 defines INADDR_* constants in IPv4 host byte order. However, IPv6 defines IN6ADDR_* and in6addr* constants in network byte order.

3 Data Structure Changes

IP Address Structure

Header file

```
<netinet/in.h>
```

IPv6 Structure

```
struct in6_addr {
uint8_t  s6_addr[16];
} /* array of 16 8-bit elements = one 128-bit IPv6 address */
```

IPv4 Structure

```
struct in_addr {
  unsigned  int s_addr ; /* 32-bit  IPv4*/
  };
```

Socket Address structure for 4.3BSD-based HP-UX

Header file

```
<netinet/in.h>
```

IPv4 Structure

```
struct sockaddr_in {
 short sin_family; /*AF_INET */
 u_short sin_port; /* transport layer port number */
struct in_addr  sin_addr; /* IPv4 */
char            sin_zero[8];         /* Unused */
};
```

IPv6 Structure

```
struct sockaddr_in6 {

  sa_family_t  sin6_family; /*AF_INET6 */
 in_port_t  sin6_port; /* transport layer port number.* /
  uint32_t sin6_flowinfo; /* traffic class */
  struct in6_addr  sin6_addr; /* IPv6*/
  uint32_t sin6_scope_id;/* Address scope */

};
```

Generic Socket Address Structure

Header file

`<netinet/in.h>`

`struct sockaddr_storage`

The `sockaddr_storage` data structure simplifies writing portable code across multiple address families and platforms. This data structure provides the following flexibility and consistency.

- One simple addition to the sockets API that can help application writers is the `struct sockaddr_storage` structure. The structure is large enough to accommodate all supported protocol-specific address structures.
- `sockaddr_storage` aligns at an appropriate boundary so that pointers to it - can be cast as pointers to protocol specific address structures and used to access the fields of those structures without alignment problems.

4 Migrating Applications from IPv4 to IPv6

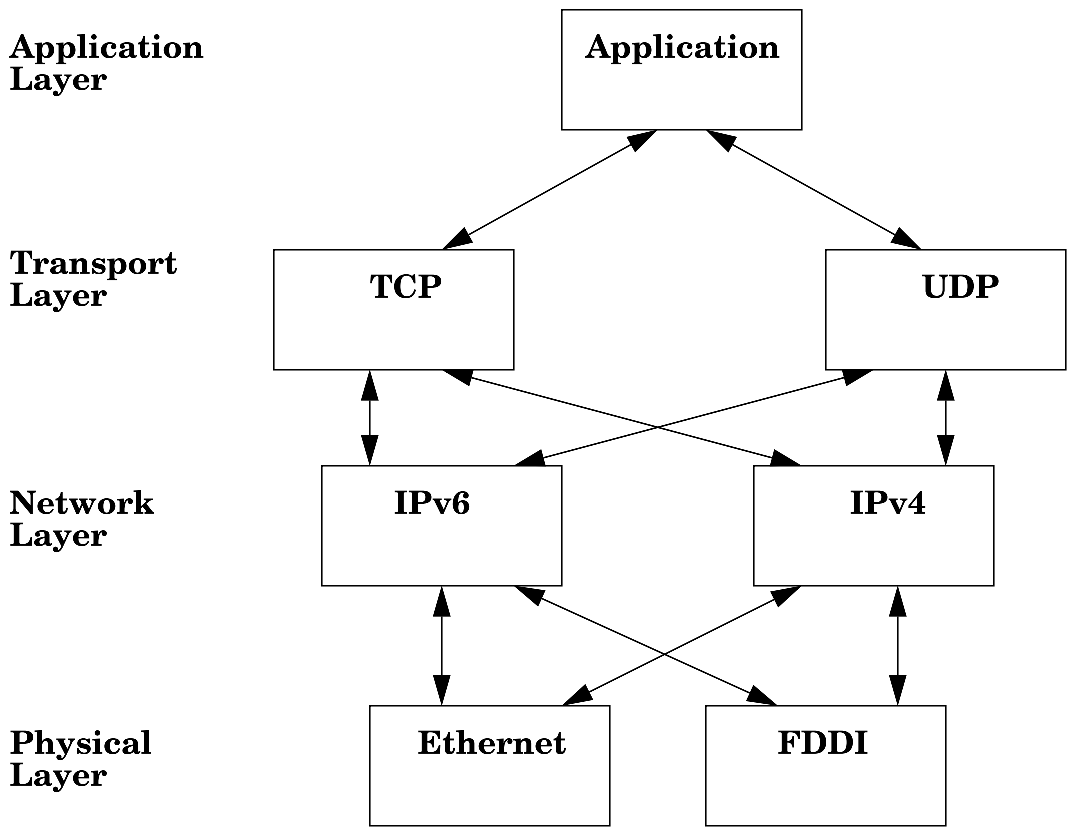
HP-UX supports two standard IPv4/IPv6 interoperability methods:

- IPv4/IPv6 Dual-Stack
- Tunneling: allows two IPv6 nodes to communicate by encapsulating IPv6 packets within IPv4 packets and routing them over an IPv4 network.

IPv4/IPv6 Dual Stack

HP-UX IPv6 supports a dual IPv4/IPv6 protocol stack. The Dual-Stack does not affect existing IPv4 source or binary files. Legacy IPv4-to-IPv4 applications follow existing code paths through the IPv4 module.

Figure 4-1 Dual IPv4 and IPv6 Stack

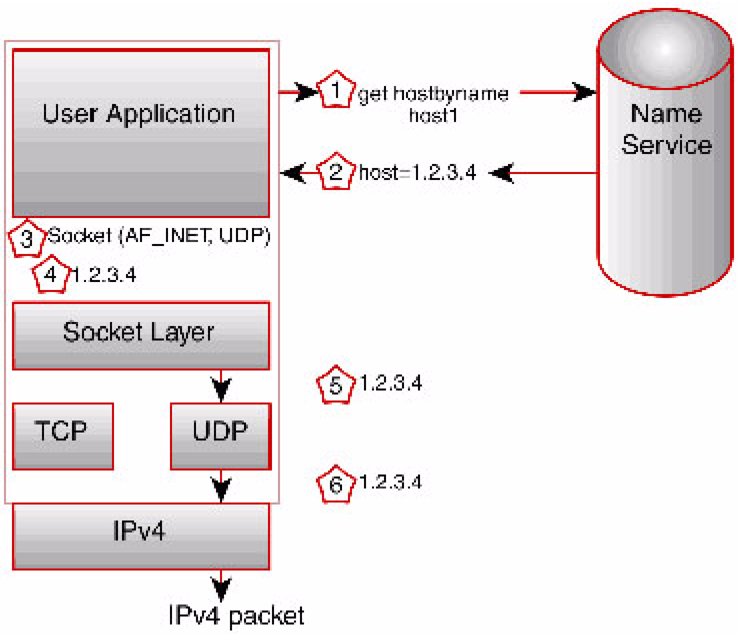


5 Overview of IPv4 and IPv6 Call Set-up

This chapter provides an overview of the call set-up process for IPv4 and IPv6.

Using AF_INET Socket for IPv4 UDP Communications

Figure 5-1

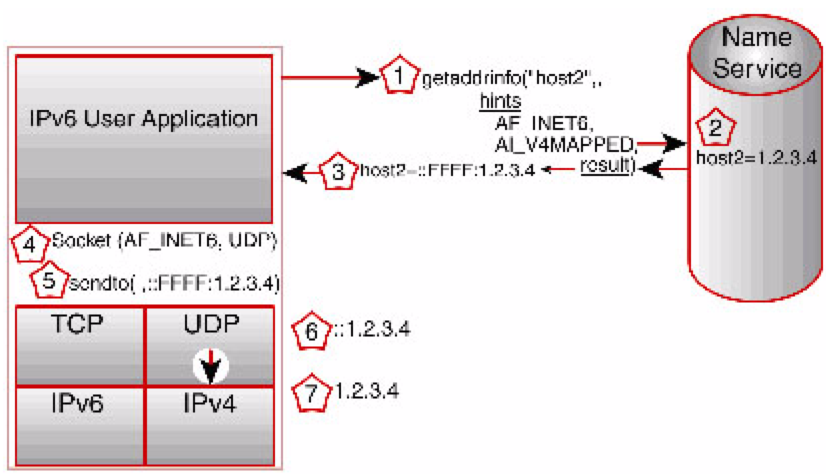


1. Application calls `gethostbyname()` and passes the host name, `host1`.
2. The search finds `host1` in the Name Service database and `gethostbyname()` returns the IPv4 address `1.2.3.4`.
3. The application calls the `socket()` function to open an IPv4 `AF_INET` socket.
4. The application calls the `send ()` function to the `1.2.3.4` address.
5. The socket layer passes the send request, socket information and address to the UDP/IP module.
6. The UDP/IP module puts the `1.2.3.4` address into the IPv4 packet header and passes the information to the IPv4 module for transmission.

Using AF_INET6 Socket to Send IPv4 UDP Communications

You can use the AF_INET6 socket for both IPv6 and IPv4 communications; IPv6 uses the POSIX function call getaddrinfo() rather than the IPv4 gethostbyname() function call. 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). The figure below shows the sequence of events for an application that uses an AF_INET6 socket to send IPv4 packets.

Figure 5-2



1. Application calls getaddrinfo() and passes:

- the host name (host2).
- the AF_INET6 address family *hint*, which asks the Name Service for an IPv6 address corresponding to the host name.
- The AI_V4MAPPED *flag hint*, which tells the function that if the Name Service finds no IPv6 address but finds an IPv4 address for host2, return the IPv4 address within an IPv4-mapped IPv6 address. See getaddrinfo(3N) later in this document for a description of *hints* and *flags* values.

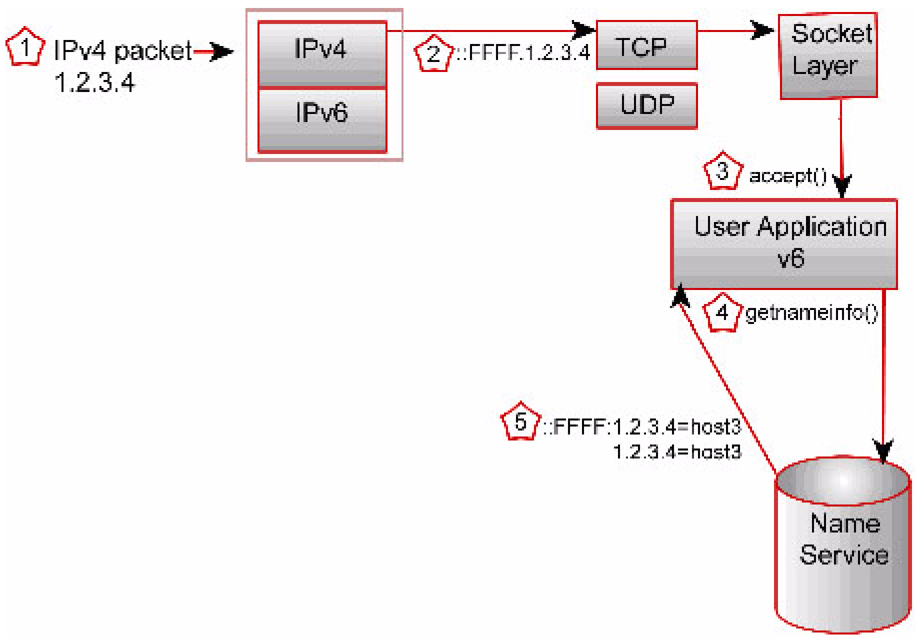
2. The search finds the IPv4 address `1.2.3.4` for `host2` in the Name Service database.
3. Because `getaddrinfo()` had the `AI_V4MAPPED` flag set, the function returns the IPv4 –mapped address `::FFFF:1.2.3.4`.
4. The application calls the `socket()` function to open an IPv6 `AF_INET6` socket.
5. The application calls the `sendto()` function toward the `::FFFF:1.2.3.4` address.
6. The socket layer passes the sendto request, socket information and IPv4-mapped IPv6 address to the UDP/IP module.
7. The UDP/IP module:
   a. identifies the IPv4-mapped IPv6 address.
   b. puts the `1.2.3.4` address into an IPv4 packet header.
   c. passes the packet to the IPv4 module for transmission.

Using AF_INET6 Socket to Receive IPv4 Communications

An IPv6 application using an `AF_INET6` socket can accept TCP connection requests from a remote IPv4 application. The example below is contrived to demonstrate an incoming IPv4 packet destined for an application's IPv6 socket.

In this overview diagram, an incoming IPv4 packet requests connection to an IPv6 socket. IPv6 internally creates an IPv4-mapped IPv6 address and accepts the connection.

Figure 5-3



1. An IPv4 packet arrives at an Ethernet port.
2. The Ethernet driver examines the *type* field in the Ethernet packet.

   `86DD` *type* is an IPv6 packet

   `0800` *type* is an IPv4 packet

   Here *type* is `0800`, so the Ethernet driver strips-off the Ethernet header and passes the IPv4 packet to the IPv4/IP module.

The IPv4/IP protocol stack passes the information and the IPv4-mapped IPv6 address (`::FFFF:1.2.3.4`) to the socket layer.

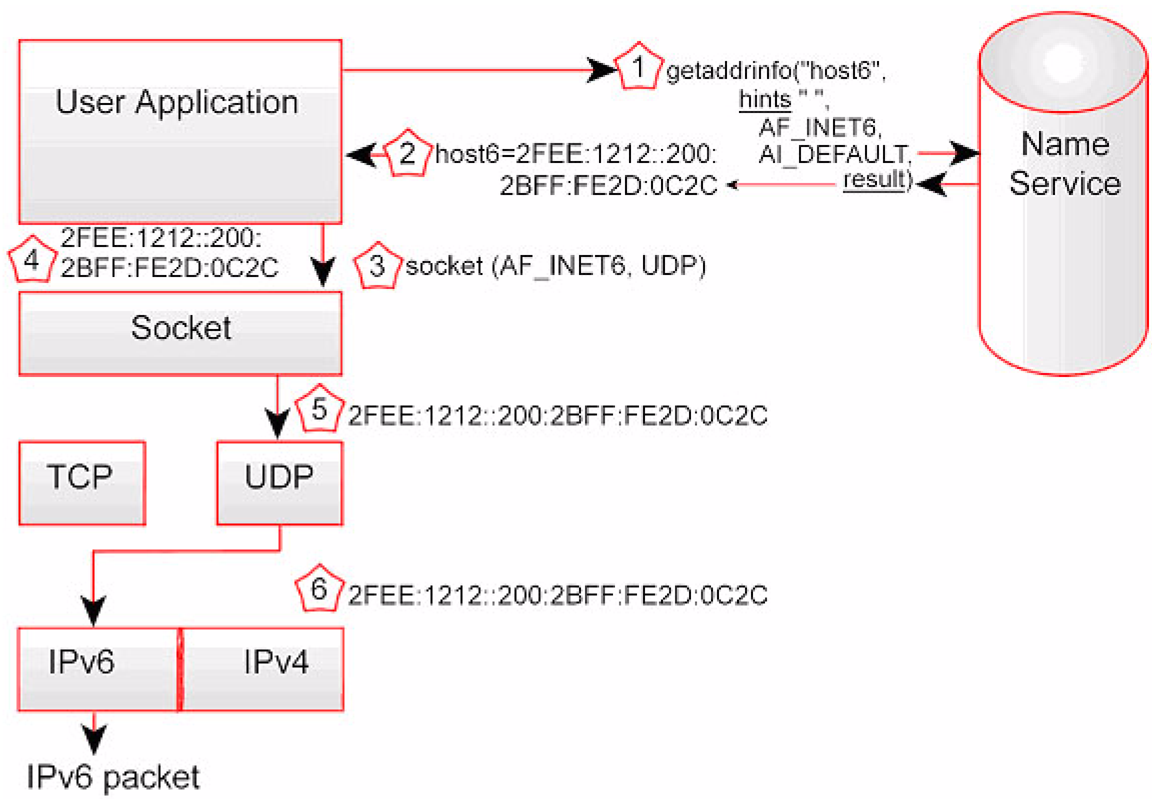
3. The application calls `accept()` to accept the remote connection request. The application was already listening on an established IPv6 socket.
4. The application calls `getnameinfo()` to lookup the host name for IP address `::FFFF:1.2.3.4`. See `getnameinfo(3N)` later in the guide for more information.
5. The search finds the host name for the `1.2.3.4` address in the `hosts` database and `getnameinfo()` returns the host name.

NOTE In the case where `getnameinfo()` cannot locate the node name corresponding to an IPv4 address, `getnameinfo()` returns the numeric form of the IPv4-mapped IPv6 address instead of the node name. Displaying the IPv4-mapped IPv6 address may cause compatibility issues for certain applications. In this case, HP recommends a pure IPv4 address is displayed.

Using AF_INET6 Socket for IPv6 Communications

For IPv6 communications, create an `AF_INET6` socket and pass it a `sockaddr_in6` structure that contains an IPv6 address that is not an IPv4-mapped IPv6 address (for example, `2fee:1212::200:2bff:fe2d:0c2c`). The diagram below shows the sequence of events for an application that uses an `AF_INET6` socket to send IPv6 packets.

Figure 5-4



1. Application calls `getaddrinfo()` and passes the host name (`host6`), the IPv6 `AF_INET6` address family *hint*, and the `AI_DEFAULT` flag *hint*. The flag *hint* tells the function to find an IPv6 address for `host6`, then return it if found. See `getaddrinfo(3)` for a description of *hints* fields and values.

2. The search finds an IPv6 address for `host6` in the hosts database, then `getaddrinfo` returns the IPv6 address `2fee:1212::200:2bff:fe2d:0c2c`.
3. The application opens an `AF_INET6` socket.
4. The application sends information to the `2fee:1212::200:2bff:fe2d:0c2c` 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 `2fee:1212::200:2bff:fe2d:0c2c` address into the packet header and passes the information to the IPv6 module for transmission.

6 Function Calls Converting Names to Addresses

The existing gethostbyname() function still looks up IPv4 addresses for particular host names. However, this library call function cannot specify address types such as IPv6 or

IPv4-mapped. Two new IPv6 function calls for IP address lookup are:

- `getaddrinfo()`
- `getipnodebyname()`

getaddrinfo(3N)

`getaddrinfo()` is a nodename-to-address and servicename-to-port-number function call. The protocol-independent function call complies with POSIX 1003.1g Draft 6.6 (1997). For more information refer to the `getaddrinfo(3N)` man page.

Syntax

```
getaddrinfo(const char *nodename, const char *servname, const struct
addrinfo *hints, struct addrinfo **res);
```

Parameters

**nodename*: A pointer to a node name or numeric string, such as an IPv4 dotted-decimal address or an IPv6 hexadecimal address. *nodename* can also point to a NULL string.

servname: A pointer to a service name (such as `ftp`) or port number (such as `21`). **servname* can also point to a NULL string. Either **nodename* or **servname* must point to a name or numeric string.

**hints*: A pointer to an `addrinfo` structure containing filters for socket-type, address family, or protocol-type. *hints* can also point to a NULL string. `addrinfo` and *hints* are described below.

***res*: A pointer to a linked list of `addrinfo` structures each containing a socket address and information regarding the socket.

addrinfo Data Structure pointed-to by hints

```
struct addrinfo {

     int     ai_flags; /* AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST,
                        * See RFC 2533 for more details*/
     int     ai_family;    /* PF_xxx */
     int     ai_socktype;  /* SOCK_xxx */
     int     ai_protocol;  /* 0 or IPPROTO_xxx for IPv4 and IPv6 */

     size_t  ai_addrlen;   /* length of ai_addr */
     char   *ai_canonname; /* canonical name for nodename */
     struct sockaddr  *ai_addr; /* binary address */
     struct addrinfo  *ai_next; /* next structure in linked list */
   };
```

NOTE Initialize the entire addrinfo data structure to zero before assigning hint values to `ai_flags`, `ai_family`, `ai_socktype`, or `ai_protocol`.

getipnodebyname(3N)

An application program calls the getipnodebyname() function to performs lookups for IPv4/IPv6 hosts.

NOTE Starting with HP-UX 11i v2, the `getipnodebyname()` function is entering OBSOLESCENCE, and will be OBSOLETED in a future HP-UX release. Therefore, it is recommended the `getnameinfo()` function be used instead.

Syntax

Host_ptr=getipnodebyname(const char *name, int addr_family, int flags, int *error_num);

Parameters

**name*: A pointer to a node name or numeric string, such as an IPv4 dotted-decimal address or an IPv6 hexadecimal address.

Addr_family: An integer that sets the address-type searched-for and returned-by the function. *Addr_family* is either `AF_INET` (IPv4) or `AF_INET6` (IPv6).

flags: An integer that specifies the conditions for returning an address, such as IPv6-only, IPv4-mapped if no IPv6 address found, or return an address only if the remote node name has at least one IP address configured.

**error_num*: A pointer to the error code returned by the `getipnodebyname()` function.

Host_ptr: The struct *hostent* returned by the *getipnodebyname()* function, containing one or more IP address for *name*.

The *hostent* structure comprises the following fields:

char **h_name*: A pointer to the canonical name (Fully Qualified Name) of host name.

char ***h_alias*: A pointer to an array of pointers-to-aliases for the host name.

int *h_addrtype*: The type of address returned within the `hostent` structure: either `AF_INET` for IPv4 addresses or `AF_INET6` for IPv6 addresses.

int *h_length*: The length of the IP address pointed-to by *name*, either 4 octets (IPv4) or 16 octets (IPv6)*.

char ***h_addr_list[0]*: Pointer to an array of pointers-to-IPv4-or-IPv6-addresses for the host name.

7 Function Calls Converting IP addresses to Names

The existing gethostbyaddr() function still looks up IPv4 host names for particular addresses. However, this library call function cannot specify address types such as IPv6 or

IPv4-mapped. Two new name lookup functions are:

- `getnameinfo(3N)`
- `getipnodebyaddr(3N)`

getnameinfo(3N)

The `getnameinfo()` function takes a socket-address structure and returns a node name or service name.

Header Files

```
#include <sys/socket.h>

#include <netdb.h>
```

Syntax

```
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
    char *host, size_t hostlen, char *serv, size_t servlen, int flags);
```

The `getnameinfo()` function translates a socket address to a node name and service location. The definitions for `getaddrinfo()` apply to `getnameinfo()`.

Parameters

**sa*: A pointer to a socket-address structure awaiting translation.

sockelen_t: The integer size of the socket address structure pointed to by *sa*.

**host*: A pointer to the host name returned by `getnameinfo()`.If the function finds no host name, it returns the host's IP address If *host* points to `NULL` or *hostlen* equals `zero`, then *host* does not return a host name or IP address. Both *host* and *serv* cannot point to `NULL`.

hostlen: The length of the character string *host*.

**serv*: A pointer to the service name returned by `getnameinfo()`. If it finds no service name, it returns the service's port number. If *serv* points to `NULL` or *servlen* equals `zero`, then *serv* does not return a service name or port number.

servlen: The length of the character string *serv*.

flags: *flags* change the default actions of the function.

- `NI_NOFQDN`: If set, `getnameinfo()` returns only the host name of Fully Qualified Domain Name (FQDN).
- `NI_NUMERICHOST`: If set, `getnameinfo()` returns only the numeric form of host's address.
- `NI_NAMEREQD`: If set, `getnameinfo()` returns an error if it finds no host name.

- NI_NUMERICSERV: If set, getnameinfo() returns only service's port number.
- NI_NUMERICSCOPE: If set, getnameinfo() returns the numeric form of the scope-ID. It is ignored if the *sa* parameter is not an IPv6 address.
- NI_DGRAM: If set, service is a datagram service (SOCK_DGRAM). Default: service is a stream service (SOCK_STREAM). This distinguishes between services for TCP and UDP that share port numbers (for example, 512 to 514).

getipnodebyaddr(3N)

The IPv6 `getipnodebyaddr()` function call improves upon the IPv4 `gethostbyaddr()` by adding an error number parameter.

NOTE Starting with the HP-UX 11i v2 release, the `getipnodebyaddr()` function is entering OBSOLESCENCE, and will be OBSOLETED in a future HP-UX release. Therefore, it is recommended the `getaddrinfo()` function be used instead.

Header Files

```
#include <sys/socket.h>

#include <netdb.h>
```

Syntax

```
name_ptr =getipnodebyaddr(const void *src, size_t len,int af, int *error_num);
```

Parameters

**src*: A pointer to the structure containing the IP address searched.

len: The length of the IP address: four octets for `AF_INET` or sixteen octets for `AF_INET6`.

af: Address family `AF_INET` or `AF_INET6`.

**error_num*: **error_num* is a pointer to the integer containing an error code, if any.

name_ptr: A pointer to the struct *hostent* returned by the function, containing the host name.

Data Structures

```
struct hostent {
char  *h_name;    /* Canonical name of host name such as grace.hp.com*/
char **h_alias;  /* Pointer to an array of pointers to alias names */
int    h_addrtype; /* AF_INET (for IPv4 addresses)AF_INET6 (for IPv6)*/
int    h_length;   /* 4 octets (IPv6) or 16 octets (IPv6) */
char **h_addr_list[0]; /* Pointer to an array of pointers to IPv4 */
}     /* addresses or IPv6 addresses */
```

How getipnodebyaddr() processes IPv4-compatible IPv6 addresses

If *af* is AF_INET6, *len* equals 16, and the IPv6 address is an IPv4-mapped or an IPv4-compatible IPv6 address, then:

1. skip the first 12 bytes of the IPv6 address.
2. set *af* to AF_INET.
3. set *len* to 4.

If *af* is AF_INET, lookup the name for the given IPv4 address; that is, query for a PTR record in the in-addr.arpa domain.

If *af* is AF_INET6, lookup the name for the given IPv6 address; that is, query for a PTR record in the ip6.int domain.

A successful function call copies **src* and *af* into the returned hostent *name_ptr* structure. An unsuccessful function returns a nonzero *error_num*.

8 Reading Error Messages

The IPv6 functions getipnodebyaddr(), getipnodebyname(), getaddrinfo(), and getnameinfo() return errors in a thread-safe structure. The gai_strerror() function call returns a character string describing the error code passed into it.

Header Files

```
#include <netdb.h>
```

Syntax

```
char *gai_strerror(int ecode);
```

Parameters

ecode: One of the `EAI_xxx` values defined in RFC 25333, “Basic Socket Extensions for IPv6”. The return value points to a string describing the error. If *ecode* is not one of the `EAI_xxx` values, the function returns a pointer to a string indicating an unknown error.

9 Freeing Memory

The four IPv6 name and address conversion function calls all dynamically allocate memory. IPv6 provides two function calls to free memory.

Freeing Memory from getaddrinfo() and getnameinfo() Function Calls

The function call `freeaddrinfo()` frees the memory of one or more `addrinfo()` structures returned by the `getaddrinfo()` or `getnameinfo()` functions.

Header Files

```
#include <netdb.h>
```

Syntax

```
void freeaddrinfo(struct addrinfo *ai);
```

Parameters

`*ai`: pointer to the structure `addrinfo`.

Freeing Memory from getipnodebyaddr() and getipnodebyname() Function Calls

The function call `freehostent()` frees the memory of one or more `hostent()` structures returned by the `getipnodebyaddr()` or `getipnodebynameinfo()` functions.

Syntax

```
void freehostent(struct hostent *ptr);
```

Parameters

`*ptr`: A pointer to the structure `hostent`.

10 Converting Binary and Text Addresses

The IPv4 function calls convert IPv4 addresses as follows:

The `inet_aton()` or `inet_addr()` functions convert dotted-decimal string (such as `10.9.8.7`) to 32-bit binary in network byte order.

`inet_ntoa()` converts 32-bit network byte order binary into dotted-decimal string (such as `10.9.8.7`).

Two new IPv6 functions convert both IPv4 and IPv6 addresses.

Converting a Text Address to Binary

Syntax

```
void inet_pton(int addr_family, const char *strptr, void *addrptr)
```

The `inet_pton()` function call converts the IP address pointed to by *strptr*, from presentation (string) format to numeric (binary) format, in the buffer pointed to by *addrptr*.

Converting a Binary Address to Text

Syntax

```
inet_ntop(int family, const void *addrptr, char *strptr, site_t len)
```

The `inet_ntop()` function call converts an IP address from *numeric* format to *string* format. The *len* parameter specifies the calling function's buffer size to prevent overflow. Two definitions specify this buffer size for either IPv4 or IPv6 addresses in the `<netinet/in.h>` header file.

```
#defineINET_ADDRSTRLEN16 /* for IPv4 dotted-decimal */
#defineINET6_ADDRSTRLEN46 /* for IPv6 hex string */
```

11 Testing for Scope and Type of IPv6 addresses using Macros

Use the following macros to verify IPv6 address types. The first seven macros return true if the address is of the specified type, or false otherwise. The last five macros return true if the address is a multicast address of the specified scope, or return false if the address is either not a multicast address or not of the specified scope.

NOTE `IN6_IS_ADDR_LINKLOCAL` and `IN6_IS_ADDR_SITELOCAL` return true only for the link-local scope or site-local scope IPv6 unicast addresses. These two macros do not return true for IPv6 multicast addresses of either link-local scope or site-local scope.

```
int  IN6_IS_ADDR_UNSPECIFIED (const struct in6_addr *);
int  IN6_IS_ADDR_LOOPBACK    (const struct in6_addr *);
int  IN6_IS_ADDR_MULTICAST   (const struct in6_addr *);
int  IN6_IS_ADDR_LINKLOCAL   (const struct in6_addr *);
int  IN6_IS_ADDR_SITELOCAL   (const struct in6_addr *);
int  IN6_IS_ADDR_V4MAPPED    (const struct in6_addr *);
int  IN6_IS_ADDR_V4COMPAT    (const struct in6_addr *);
```

These macros test the scope of IPv6multicast addresses:

```
int  IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr *);
int  IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr *);
int  IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr *);
int  IN6_IS_ADDR_MC_ORGLOCAL (const struct in6_addr *);
int  IN6_IS_ADDR_MC_GLOBAL   (const struct in6_addr *);
```

12 Identifying Local Interface Names and Indexes

The IPv6 sockets API uses an interface index (a small positive integer) to identify the local interface joined to a multicast group. Interfaces are normally known by names such as "lan0”.

On HP-UX implementations, when the system configures an interface, the kernel assigns a unique positive integer value (called the interface index) to that interface. These small positive integers start at one. Interface numbering is not necessarily contiguous.

This API defines:

- two functions that map between an interface name and index:
  - `if_nametoindex()`
  - `if_indextoname()`
- a function that returns all interface names and indexes:
  - `if_nameindex()`
- a function to return the dynamic memory allocated by the previous function:
  - `if_freenameindex()`

Name-to-Index

The first function maps an interface name into its corresponding index.

Header Files

```
#include <net/if.h>
```

Syntax

```
unsigned int  if_nametoindex(const char *ifname);
```

If the specified interface name does not exist, the function returns a value of zero, and sets `errno` to `ENXIO`. If a system error occurred (such as running out of memory), the function returns a value of zero and sets `errno` to the proper value (such as `ENOMEM`).

Index-to-Name

The second function maps an interface index into its corresponding name.

Header Files

```
#include <net/if.h>
```

Syntax

```
char  *if_indextoname(unsigned int ifindex, char *ifname);
```

The *ifname* parameter must point to a buffer at least `IF_NAMESIZE` bytes large. The function returns to *ifname* the interface name of the specified index. (`IF_NAMESIZE` is also defined in `<net/if.h>` and its value includes a terminating NULL byte at the end of the interface name.) The pointer to `if_indextoname` also returns the value of the function. If no interface corresponds to the specified index, the function returns NULL, and sets `errno` to `ENXIO`. If a system error occurred (such as running out of memory), `if_indextoname()` returns NULL and sets `errno` to the proper value (that is, `ENOMEM`).

Returning All Interface Names and Indexes

The `if_nameindex` structure holds the information about a single interface. The definition of the structure is in the `<net/if.h>` header file.

```
struct if_nameindex {
 unsigned int   if_index;  /* 1, 2, ... */
        char          *if_name;   /* null terminated name: "le0", .. */
      };
```

The final function returns an array of `if_nameindex` structures, returning one structure per interface.

```
struct if_nameindex  *if_nameindex(void);
```

The `if_nameindex` function signals the end of the array of structures by returning a structure with a `zero` *if_index* value and a `NULL` *if_name* value. If an error occurred, the function returns a `NULL` pointer, and sets `errno` to the appropriate value.

The if_nameindex() function acquires memory dynamically for the array of if_nameindex structures and for if_name's interface names. The if_freenameindex() function frees that memory.

Freeing Memory

The if_freenameindex() function frees the dynamic-memory allocated by if_nameindex().

Header Files

```
#include <net/if.h>
```

Syntax

```
void  if_freenameindex(struct if_nameindex *ptr);
```

The ptr parameter is the pointer returned by a previous if_nameindex() call.

13 Configuring or Querying an Interface using IPv6 ioctl() Function Calls

Certain IPv4 applications need detailed configuration information for a network interface of a node. They use the `SIOCGIFCONF`, `SIOCGIFADDR`, `SIOCGIFFLAGS`, and other `ioctl()` function calls, as defined in /usr/include/sys/ioctl.h, to determine the characteristics of the network interfaces and their attributes.

All of the IPv4 SIOC* ioctl() function calls use the struct ifreq data structure (defined in /usr/include/net/if.h) as one of the arguments for the SIOC* ioctl() function calls. However, the ifreq data structure defined for IPv4 is not large enough to hold an IPv6 address. Therefore, the existing IPv4 SIOC* and their associated data structures are not applicable for IPv6 applications.

New ioctl() function calls for IPv6-applications-follow the SIOCSL* and SIOCGL* ioctl() name format. IPv6 ioctl() function calls also use a larger data structure described below. They are otherwise identical to the IPv4 ioctl() function calls.

NOTE The IPv6 SIOCSL* and SIOCGL* ioctl() function calls are not supported for IPv4 applications.

Definitions for both IPv6 and IPv4 ioctl() function calls are in /usr/include/sys/ioctl.h.

NOTE Use a larger data structure for IPv6 addresses. IPv6 addresses cannot fit into the IPv4 struct ifreq data structure used by IPv4 SIOC* ioctl() function calls. IPv6 applications pass, as a parameter to IPv6 ioctl()function calls, the data structures struct if_laddrreq and struct if_laddrconf.

Both IPv6 and IPv4 ioctl() data structures are in /usr/include/net/if.h.

14 Verifying IPv6 Installation

The following code fragment shows how an application can determine programmatically whether IPv6 is implemented on HP-UX. An application can check the existence of the `/dev/ip6` device file at compile-time and/or run-time to determine whether IPv6 APIs and the IPv6 stack are on the system. If `/dev/ip6` does not exist, an application continues to use IPv4

APIs.

```
if ((fd = open("dev/ip6", O_RDWR)) == -1)
        /*
         * /dev/ip6 failed to open., Therefore  the IPv6 product
         *is not installed on the system. An application should use the
         * existing IPv4 code.
         */

     ...

else
        /*
         * dev/ip6 exists, so the IPv6 product is probably installed.
         * IPv6 APIs can handle both IPv4 and IPv6 traffic */
```

NOTE Starting with HP-UX 11i v2, IPv6 is automatically included in HP-UX.

15 Sample Client/Server Programs

The following code fragments are based on the same IPv4 client/server sample programs shipped in the HP-UX 11i v2 `/usr/lib/demos/networking/socket` directory.

The client requests a service called `example`. Add an entry to the client's `/etc/services` file for `example`. Assign any unused port number, such as 22375, to the service `example` for a port address. The host running the server must also have the same port number assigned to `example` in the server's `/etc/services` file.

IPv4 TCP Client Code Fragment

This code fragment is part of the same IPv4 client program that ships in the HP-UX 11i IPv6 `/usr/lib/demos/networking/socket` directory.

The client requests a service called “`example`.” Add an entry to the `/etc/services` for “`example`”. Assign any unused port number, such as 22375, to the service “`example`” for a port address. The host running the server must also have the same port number assigned to “`example`” in the `/etc/services` file.

```
struct sockaddr_in peeraddr_in; /* for peer socket address */

memset ((char *)&peeraddr_in, 0, sizeof(struct sockaddr_in));

hp = gethostbyname (argv[1]);

        if (hp == NULL) {
                fprintf(stderr, "%s: %s not found in /etc/hosts\n",
                                argv[0], argv[1]);
                exit(1);
        }

peeraddr_in.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;

                /* Find the information for the "example" server
                 * in order to get the needed port number.
                 */

sp = getservbyname ("example", "tcp");

if (sp == NULL) {
    fprintf(stderr, "%s: example not found in /etc/services\n argv[0]);
     exit(1);
}
peeraddr_in.sin_port = sp->s_port;

        /* Create the socket. */
s = socket (AF_INET, SOCK_STREAM, 0);
if (s == -1) {
    perror(argv[0]);
    fprintf(stderr, "%s: unable to create socket\n", argv[0]);
       exit(1);
}
/* Try to connect to the remote server at the address put in peeraddr.
 */
if (connect(s, &peeraddr_in, sizeof(struct sockaddr_in)) == -1{
```

```
    perror(argv[0]);
    fprintf(stderr, "%s: unable to connect to remote\n", argv[0]);
            exit(1);
        }
```

IPv6 TCP Client using getipnodebyname()

This code fragment is part of an example IPv6 client program that ships in the HP-UX 11i v2 /usr/lib/demos/networking/socket/af_inet6 directory, rewritten using the getipnodebyname() function call.

```
struct sockaddr_in6 peeraddr_in6;       /* for peer socket address */

memset ((char *)&peeraddr_in6, 0, sizeof(struct sockaddr_in6));

hp = getipnodebyname (argv[1], AF_INET6, AI_DEFAULT, &error);

  if (hp == NULL) {
    fprintf(stderr, "%s: %s not found in /etc/hosts\n",
        argv[0], argv[1]);
    exit(1);
  }
  peeraddr_in6.sin6_family = hp->h_addrtype;
  memcpy(&peeraddr_in6.sin6_addr, hp->h_addr, hp->h_length);
    /* Find the information for the "example" server
     * in order to get the needed port number.
     */

  sp = getservbyname ("example", "tcp");

  if (sp == NULL) {

    fprintf(stderr, "%s: example not found in /etc/services\n",
        argv[0]);
    exit(1);
  }
  peeraddr_in6.sin6_port = sp->s_port;

    /* Create the socket. */
  s = socket (AF_INET6, SOCK_STREAM, 0);
  if (s == -1) {
    perror(argv[0]);
    fprintf(stderr, "%s: unable to create socket\n", argv[0]);
    exit(1);
  }
    /* Try to connect to the remote server at the address
     * which was just built into peeraddr.
     */

  if (connect(s, &peeraddr_in6, sizeof(peeraddr_in6)) == -1) {

    perror(argv[0]);
    fprintf(stderr, "%s: unable to connect to remote\n", argv[0]);
    exit(1);
  }
```

IPv6 TCP Client Using getaddrinfo() for Name/Service Lookup

This fragment of an IPv6 TCP Client is a port of the preceding IPv6 client, using `getaddrinfo()` rather than `gethostbyname()`.

```
struct addrinfo *res, *ainfo;

struct addrinfo hints;

/* clear out hints */

memset ((char *)&hints, 0, sizeof(hints));

hints.ai_socktype = SOCK_STREAM;

error =  getaddrinfo(argv[1], "example", &hints, &res);

if (error != 0) {
fprintf(stderr, "%s: %s not found in name service database\n",
argv[0], argv[1]);
exit(1);
}
for (ainfo = res; ainfo != NULL; ainfo = ainfo->ai_next) {
/* Create the socket. */
s = socket (ainfo->ai_family,ainfo->ai_socktype,
ainfo->ai_protocol);
if (s == -1) {
perror(argv[0]);
fprintf(stderr, "%s: unable to create socket\n", argv[0]);
freeaddrinfo(res);
exit(1);
}
if (connect(s, ainfo->ai_addr, ainfo->ai_addrlen) == -1) {
 perror(argv[0]);
 fprintf(stderr, "%s: unable to connect to remote\n", argv[0]);
close(s);
continue;
}
else
break;
}
```

IPv4 TCP Server Code Fragment

This code fragment is part of the same example IPv4 server program that ships in the HP-UX 11i v2 `/usr/lib/demos/networking/socket` directory.

```
struct sockaddr_in6 peeraddr_in6;       /* for peer socket address */

sp = getservbyname ("example", "tcp");

        if (sp == NULL) {
   fprintf(stderr, "%s: example not found in /etc/services\n",argv[0]);
                exit(1);
        }
        myaddr_in.sin_port = sp->s_port;

                /* Create the listen socket. */
        ls = socket (AF_INET, SOCK_STREAM, 0);
        if (ls == -1) {
                perror(argv[0]);
            fprintf(stderr, "%s: unable to create socket\n", argv[0]);
                exit(1);
        }
                /* Bind the listen address to the socket. */
        if (bind(ls, &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
                perror(argv[0]);
             fprintf(stderr, "%s: unable to bind address\n", argv[0]);
                exit(1);
        }
                /* Initiate the listen on the socket so remote users
                 * can connect.  The listen backlog is set to 5, which
                 * is within the supported range of 1 to 20.
                 */
        if (listen(ls, 5) == -1) {
          perror(argv[0]);
          fprintf(stderr, "%s: unable to listen on socket\n", argv[0]);
                exit(1);
        }
```

IPv6 TCP Server using getaddrinfo() for Service Address Lookup

This code fragment is part of the example IPv6 server program that ships in the HP-UX 11i v2 `/usr/lib/demos/networking/socket/af_inet6` directory, rewritten using the `getaddrinfo()` function call.

```
struct addrinfo *ainfo, *res;

struct addrinfo hints;


/* zero-out the hints before assignment */

memset (&hints, 0, sizeof(hints));
.

hints.ai_family = AF_INET6;
 hints.ai_flags = AI_PASSIVE;
hints.ai_socktype = SOCK_STREAM;


 error = getaddrinfo(NULL, "example", &hints, &res);

 if (error != 0) {
 fprintf(stderr, "%s: %s for service 'example'\n",
 argv[0], gai_strerror(error));
 exit(1);
 }
 /* Create the listen socket. */
 ls = socket (res->ai_family, res->ai_socktype, res->ai_protocol);
 if (ls == -1) {
  perror(argv[0]);
 fprintf(stderr, "%s: unable to create socket\n", argv[0]);
 exit(1);
 }
 /* Bind the listen address to the socket. */
 if (bind(ls, res->ai_addr, res->ai_addrlen) == -1) {
 perror(argv[0]);
 fprintf(stderr, "%s: unable to bind address\n", argv[0]);
 close(ls);
 exit(1);
 }
 /* Initiate the listen on the socket so remote users
  * can connect.  The listen backlog is set to 5, which
  * is within the supported range of 1 to 20.
  */
 if (listen(ls, 5) == -1) {
 perror(argv[0]);
```

```
fprintf(stderr, "%s: unable to listen on socket\n", argv[0]);
close(ls);
exit(1);

}
```

A IPv4 to IPv6 Quick Reference Guide

This guide is for Socket Application programmers who primarily want to know which source code symbols and functions require alteration to support IPv6.

Do Existing IPv4-to-IPv4 Applications Require Changes?

No. Current IPv4 applications remain unchanged. Modify applications only to take advantage of new IPv6 features.

Summary: Source Code Symbols and Function Changes

The following tables cover changes in the source code symbols and functions that Socket Application programmers need to be aware of when porting code to support IPv6.

Changes to Symbols, Data Structures, and Function Calls

Table A-1 Changes to Symbols, Data Structures, and Function Calls

Search source code for:	Replace with:

Symbols
 	
`AF_INET`
`PF_INET`
	
	`AF_INET6`
	`PF_INET6`

Data Structures
 	
`sockaddr_in`
`u_short sin_family`
`in_port_t sin_port`
`sin_addr struct in_addr`
 	
`ifreq`
`ifconf`
	
	`sockaddr_in6`
	`shortsin6_family;`
	`u_shortsin6_port;`
	`uint32_tsin6_flowinfo;`
	`struct in6_addrsin6_addr;`
	`uint32_tsin6_scope_id`
	
	`struct if_laddrreq`
	`struct if_laddrconf`

Function Calls
 	
`gethostbyname()`
 	
`gethostbyaddr()`
 	
`inet_ntoa()`
`inet_addr() or inet_aton()`
	
	`getaddrinfo() or getipnodebyname(),`
	`freeaddrinfo()`
	
	`getipnodebyaddr(),getnameinfo(),`
	`freeaddrinfo()`
	
	`inet_ntop()`
	`inet_pton()`

Watch for hard-coded data structure sizes

Watch for `sizeof(struct sockaddr_in) = sizeof(struct sockaddr) = 16` in pre-ported applications. The IPv6 address data structure `sockaddr_in6` is larger than the traditional `sockaddr_in` data structure.

Multicast and IPv4 Options

Table A-2 Multicast and IPv4 Options

IPv4	IPv6	Comments
IN_CLASSA IN_CLASSB IN_CLASSC IN_CLASSD	None. IPv6 addressing is classless.	

Loopback Address

Table A-3 Loopback Address

IPv4	IPv6	Comments
INADDR_LOOPBACK	in6addr_loopback	in6adr_loopback is an in6_addr structure

Wildcard Address

Table A-4 Wildcard Address

IPv4	IPv6	Comments
INADDR_ANY	in6addr_any	in6addr_any is an in6_addr structure

Multicast Defaults

Table A-5 Multicast Defaults

IPv4	IPv6	Comments
IP_DEFAULT_MULTICAST_LOOP IP_DEFAULT_MULTICAST_TTL	IPV6_DEFAULT_MULTICAST_LOOP IPV6_DEFAULT_MULTICAST_HOPS	

IPv6 Multicast Options

Table A-6 IPv6 Multicast Options

IPv4	IPv6	Comments
IP_MULTICAST_IF IP_MULTICAST_TTL IP_MULTICAST_LOOP IP_ADD_MEMBERSHIP IP_DROP_MEMBERSHIP	IPV6_MULTICAST_IF IPV6_MULTICAST_HOPS IPV6_MULTICAST_LOOP IPV6_JOIN_GROUP IPV6_LEAVE_GROUP	

NOTE When setting the `getsockopt()` and `setsockopt()` *`level`* parameter, use `IPPROTO_IPV6` level for all `IPV6_*` options listed here.

IP Packet Options

Table A-7 IP Packet Options

IP_OPTIONS	IPV6_PKTOPTIONS	Comments
IP_RECVDSTADDR	IPV6_DESTOPTS	Receive Destination options
IP_RECVIF	IPV6_HOPLIMIT	Unicast hop limit for receiving packets
	IPV6_HOPOPTS	Receive hop-by-hop options
	IPV6_NEXTHOP	Set next-hop address
	IPV6_PKTINFO	Set packet information
	IPV6_PKTINFO	Return destination IP address
	IPV6_PKTINFO	Return received interface index
	IPV6_RTHDR	Send or receive routing header
IP_TTL	IPv6_UNICAST_HOPS	Default unicast hop limit
ip_mreq	ipv6_IP_OPTIONSmreq	

NOTE Bundle the seven options above into a single `setsockopt()` call using `IPV6_PKTOPTIONS`.

Types of Service Options

Table A-8 Types of Service Options

`IP_TOS`	Still under discussion by IETF IPng working group.

Multicast Group, IP Address, and IPv6 Interface Index

Table A-9 Multicast Group, IP Address, and IPv6 Interface Index

IPv4	IPv6	Comments
`struct in_addr imr_multicast`	`struct in6_addr ipv6mr_multiaddr`	Multicast address of group
`struct in_addrimr_interface`	`uint32 ipv6mr_interface`	IPv4: local IP address of interface IPv6: interface index