

Network Working Group  
Request for Comments: 2553  
Obsoletes: 2133  
Category: Informational

R. Gilligan  
FreeGate  
S. Thomson  
Bellcore  
J. Bound  
Compaq  
W. Stevens  
Consultant  
March 1999

## Basic Socket Interface Extensions for IPv6

### Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

### Copyright Notice

Copyright (C) The Internet Society (1999). All Rights Reserved.

### Abstract

The de facto standard application program interface (API) for TCP/IP applications is the "sockets" interface. Although this API was developed for Unix in the early 1980s it has also been implemented on a wide variety of non-Unix systems. TCP/IP applications written using the sockets API have in the past enjoyed a high degree of portability and we would like the same portability with IPv6 applications. But changes are required to the sockets API to support IPv6 and this memo describes these changes. These include a new socket address structure to carry IPv6 addresses, new address conversion functions, and some new socket options. These extensions are designed to provide access to the basic IPv6 features required by TCP and UDP applications, including multicasting, while introducing a minimum of change into the system and providing complete compatibility for existing IPv4 applications. Additional extensions for advanced IPv6 features (raw sockets and access to the IPv6 extension headers) are defined in another document [4].

## Table of Contents

1. Introduction.....	3
2. Design Considerations.....	3
2.1 What Needs to be Changed.....	4
2.2 Data Types.....	5
2.3 Headers.....	5
2.4 Structures.....	5
3. Socket Interface.....	6
3.1 IPv6 Address Family and Protocol Family.....	6
3.2 IPv6 Address Structure.....	6
3.3 Socket Address Structure for 4.3BSD-Based Systems.....	7
3.4 Socket Address Structure for 4.4BSD-Based Systems.....	8
3.5 The Socket Functions.....	9
3.6 Compatibility with IPv4 Applications.....	10
3.7 Compatibility with IPv4 Nodes.....	10
3.8 IPv6 Wildcard Address.....	11
3.9 IPv6 Loopback Address.....	12
3.10 Portability Additions.....	13
4. Interface Identification.....	16
4.1 Name-to-Index.....	16
4.2 Index-to-Name.....	17
4.3 Return All Interface Names and Indexes.....	17
4.4 Free Memory.....	18
5. Socket Options.....	18
5.1 Unicast Hop Limit.....	18
5.2 Sending and Receiving Multicast Packets.....	19
6. Library Functions.....	21
6.1 Nodename-to-Address Translation.....	21
6.2 Address-To-Nodename Translation.....	24
6.3 Freeing memory for getipnodebyname and getipnodebyaddr.....	26
6.4 Protocol-Independent Nodename and Service Name Translation.....	26
6.5 Socket Address Structure to Nodename and Service Name.....	29
6.6 Address Conversion Functions.....	31
6.7 Address Testing Macros.....	32
7. Summary of New Definitions.....	33
8. Security Considerations.....	35
9. Year 2000 Considerations.....	35
Changes From RFC 2133.....	35
Acknowledgments.....	38
References.....	39
Authors' Addresses.....	40
Full Copyright Statement.....	41

## 1. Introduction

While IPv4 addresses are 32 bits long, IPv6 interfaces are identified by 128-bit addresses. The socket interface makes the size of an IP address quite visible to an application; virtually all TCP/IP applications for BSD-based systems have knowledge of the size of an IP address. Those parts of the API that expose the addresses must be changed to accommodate the larger IPv6 address size. IPv6 also introduces new features (e.g., traffic class and flowlabel), some of which must be made visible to applications via the API. This memo defines a set of extensions to the socket interface to support the larger address size and new features of IPv6.

## 2. Design Considerations

There are a number of important considerations in designing changes to this well-worn API:

- The API changes should provide both source and binary compatibility for programs written to the original API. That is, existing program binaries should continue to operate when run on a system supporting the new API. In addition, existing applications that are re-compiled and run on a system supporting the new API should continue to operate. Simply put, the API changes for IPv6 should not break existing programs. An additional mechanism for implementations to verify this is to verify the new symbols are protected by Feature Test Macros as described in IEEE Std 1003.1. (Such Feature Test Macros are not defined by this RFC.)
- The changes to the API should be as small as possible in order to simplify the task of converting existing IPv4 applications to IPv6.
- Where possible, applications should be able to use this API to interoperate with both IPv6 and IPv4 hosts. Applications should not need to know which type of host they are communicating with.
- IPv6 addresses carried in data structures should be 64-bit aligned. This is necessary in order to obtain optimum performance on 64-bit machine architectures.

Because of the importance of providing IPv4 compatibility in the API, these extensions are explicitly designed to operate on machines that provide complete support for both IPv4 and IPv6. A subset of this API could probably be designed for operation on systems that support only IPv6. However, this is not addressed in this memo.

## 2.1 What Needs to be Changed

The socket interface API consists of a few distinct components:

- Core socket functions.
- Address data structures.
- Name-to-address translation functions.
- Address conversion functions.

The core socket functions -- those functions that deal with such things as setting up and tearing down TCP connections, and sending and receiving UDP packets -- were designed to be transport independent. Where protocol addresses are passed as function arguments, they are carried via opaque pointers. A protocol-specific address data structure is defined for each protocol that the socket functions support. Applications must cast pointers to these protocol-specific address structures into pointers to the generic "sockaddr" address structure when using the socket functions. These functions need not change for IPv6, but a new IPv6-specific address data structure is needed.

The "sockaddr\_in" structure is the protocol-specific data structure for IPv4. This data structure actually includes 8-octets of unused space, and it is tempting to try to use this space to adapt the sockaddr\_in structure to IPv6. Unfortunately, the sockaddr\_in structure is not large enough to hold the 16-octet IPv6 address as well as the other information (address family and port number) that is needed. So a new address data structure must be defined for IPv6.

IPv6 addresses are scoped [2] so they could be link-local, site, organization, global, or other scopes at this time undefined. To support applications that want to be able to identify a set of interfaces for a specific scope, the IPv6 sockaddr\_in structure must support a field that can be used by an implementation to identify a set of interfaces identifying the scope for an IPv6 address.

The name-to-address translation functions in the socket interface are gethostbyname() and gethostbyaddr(). These are left as is and new functions are defined to support IPv4 and IPv6. Additionally, the POSIX 1003.g draft [3] specifies a new nodename-to-address translation function which is protocol independent. This function can also be used with IPv4 and IPv6.

The address conversion functions -- `inet_ntoa()` and `inet_addr()` -- convert IPv4 addresses between binary and printable form. These functions are quite specific to 32-bit IPv4 addresses. We have designed two analogous functions that convert both IPv4 and IPv6 addresses, and carry an address type parameter so that they can be extended to other protocol families as well.

Finally, a few miscellaneous features are needed to support IPv6. New interfaces are needed to support the IPv6 traffic class, flow label, and hop limit header fields. New socket options are needed to control the sending and receiving of IPv6 multicast packets.

The socket interface will be enhanced in the future to provide access to other IPv6 features. These extensions are described in [4].

## 2.2 Data Types

The data types of the structure elements given in this memo are intended to be examples, not absolute requirements. Whenever possible, data types from Draft 6.6 (March 1997) of POSIX 1003.1g are used: `uintN_t` means an unsigned integer of exactly N bits (e.g., `uint16_t`). We also assume the argument data types from 1003.1g when possible (e.g., the final argument to `setsockopt()` is a `size_t` value). Whenever buffer sizes are specified, the POSIX 1003.1 `size_t` data type is used (e.g., the two length arguments to `getnameinfo()`).

## 2.3 Headers

When function prototypes and structures are shown we show the headers that must be `#included` to cause that item to be defined.

## 2.4 Structures

When structures are described the members shown are the ones that must appear in an implementation. Additional, nonstandard members may also be defined by an implementation. As an additional precaution nonstandard members could be verified by Feature Test Macros as described in IEEE Std 1003.1. (Such Feature Test Macros are not defined by this RFC.)

The ordering shown for the members of a structure is the recommended ordering, given alignment considerations of multibyte members, but an implementation may order the members differently.

### 3. Socket Interface

This section specifies the socket interface changes for IPv6.

#### 3.1 IPv6 Address Family and Protocol Family

A new address family name, `AF_INET6`, is defined in `<sys/socket.h>`. The `AF_INET6` definition distinguishes between the original `sockaddr_in` address data structure, and the new `sockaddr_in6` data structure.

A new protocol family name, `PF_INET6`, is defined in `<sys/socket.h>`. Like most of the other protocol family names, this will usually be defined to have the same value as the corresponding address family name:

```
#define PF_INET6          AF_INET6
```

The `PF_INET6` is used in the first argument to the `socket()` function to indicate that an IPv6 socket is being created.

#### 3.2 IPv6 Address Structure

A new `in6_addr` structure holds a single IPv6 address and is defined as a result of including `<netinet/in.h>`:

```
struct in6_addr {
    uint8_t  s6_addr[16];      /* IPv6 address */
};
```

This data structure contains an array of sixteen 8-bit elements, which make up one 128-bit IPv6 address. The IPv6 address is stored in network byte order.

The structure `in6_addr` above is usually implemented with an embedded union with extra fields that force the desired alignment level in a manner similar to BSD implementations of "struct in\_addr". Those additional implementation details are omitted here for simplicity.

An example is as follows:

```

struct in6_addr {
    union {
        uint8_t  _S6_u8[16];
        uint32_t _S6_u32[4];
        uint64_t _S6_u64[2];
    } _S6_un;
};
#define s6_addr _S6_un._S6_u8

```

### 3.3 Socket Address Structure for 4.3BSD-Based Systems

In the socket interface, a different protocol-specific data structure is defined to carry the addresses for each protocol suite. Each protocol-specific data structure is designed so it can be cast into a protocol-independent data structure -- the "sockaddr" structure. Each has a "family" field that overlays the "sa\_family" of the sockaddr data structure. This field identifies the type of the data structure.

The sockaddr\_in structure is the protocol-specific address data structure for IPv4. It is used to pass addresses between applications and the system in the socket functions. The following sockaddr\_in6 structure holds IPv6 addresses and is defined as a result of including the <netinet/in.h> header:

```

struct sockaddr_in6 {
    sa_family_t    sin6_family;    /* AF_INET6 */
    in_port_t      sin6_port;      /* transport layer port # */
    uint32_t        sin6_flowinfo; /* IPv6 traffic class & flow info */
    struct in6_addr sin6_addr;      /* IPv6 address */
    uint32_t        sin6_scope_id; /* set of interfaces for a scope */
};

```

This structure is designed to be compatible with the sockaddr data structure used in the 4.3BSD release.

The sin6\_family field identifies this as a sockaddr\_in6 structure. This field overlays the sa\_family field when the buffer is cast to a sockaddr data structure. The value of this field must be AF\_INET6.

The sin6\_port field contains the 16-bit UDP or TCP port number. This field is used in the same way as the sin\_port field of the sockaddr\_in structure. The port number is stored in network byte order.

The `sin6_flowinfo` field is a 32-bit field that contains two pieces of information: the traffic class and the flow label. The contents and interpretation of this member is specified in [1]. The `sin6_flowinfo` field SHOULD be set to zero by an implementation prior to using the `sockaddr_in6` structure by an application on receive operations.

The `sin6_addr` field is a single `in6_addr` structure (defined in the previous section). This field holds one 128-bit IPv6 address. The address is stored in network byte order.

The ordering of elements in this structure is specifically designed so that when `sin6_addr` field is aligned on a 64-bit boundary, the start of the structure will also be aligned on a 64-bit boundary. This is done for optimum performance on 64-bit architectures.

The `sin6_scope_id` field is a 32-bit integer that identifies a set of interfaces as appropriate for the scope of the address carried in the `sin6_addr` field. For a link scope `sin6_addr` `sin6_scope_id` would be an interface index. For a site scope `sin6_addr`, `sin6_scope_id` would be a site identifier. The mapping of `sin6_scope_id` to an interface or set of interfaces is left to implementation and future specifications on the subject of site identifiers.

Notice that the `sockaddr_in6` structure will normally be larger than the generic `sockaddr` structure. On many existing implementations the `sizeof(struct sockaddr_in)` equals `sizeof(struct sockaddr)`, with both being 16 bytes. Any existing code that makes this assumption needs to be examined carefully when converting to IPv6.

### 3.4 Socket Address Structure for 4.4BSD-Based Systems

The 4.4BSD release includes a small, but incompatible change to the socket interface. The `"sa_family"` field of the `sockaddr` data structure was changed from a 16-bit value to an 8-bit value, and the space saved used to hold a length field, named `"sa_len"`. The `sockaddr_in6` data structure given in the previous section cannot be correctly cast into the newer `sockaddr` data structure. For this reason, the following alternative IPv6 address data structure is provided to be used on systems based on 4.4BSD. It is defined as a result of including the `<netinet/in.h>` header.



```
struct sockaddr_in6 {
    uint8_t      sin6_len;          /* length of this struct */
    sa_family_t  sin6_family;       /* AF_INET6 */
    in_port_t    sin6_port;         /* transport layer port # */
    uint32_t     sin6_flowinfo;      /* IPv6 flow information */
    struct in6_addr sin6_addr;       /* IPv6 address */
    uint32_t     sin6_scope_id;      /* set of interfaces for a scope */
};
```

The only differences between this data structure and the 4.3BSD variant are the inclusion of the length field, and the change of the family field to a 8-bit data type. The definitions of all the other fields are identical to the structure defined in the previous section.

Systems that provide this version of the `sockaddr_in6` data structure must also declare `SIN6_LEN` as a result of including the `<netinet/in.h>` header. This macro allows applications to determine whether they are being built on a system that supports the 4.3BSD or 4.4BSD variants of the data structure.

### 3.5 The Socket Functions

Applications call the `socket()` function to create a socket descriptor that represents a communication endpoint. The arguments to the `socket()` function tell the system which protocol to use, and what format address structure will be used in subsequent functions. For example, to create an IPv4/TCP socket, applications make the call:

```
s = socket(PF_INET, SOCK_STREAM, 0);
```

To create an IPv4/UDP socket, applications make the call:

```
s = socket(PF_INET, SOCK_DGRAM, 0);
```

Applications may create IPv6/TCP and IPv6/UDP sockets by simply using the constant `PF_INET6` instead of `PF_INET` in the first argument. For example, to create an IPv6/TCP socket, applications make the call:

```
s = socket(PF_INET6, SOCK_STREAM, 0);
```

To create an IPv6/UDP socket, applications make the call:

```
s = socket(PF_INET6, SOCK_DGRAM, 0);
```

Once the application has created a PF\_INET6 socket, it must use the `sockaddr_in6` address structure when passing addresses in to the system. The functions that the application uses to pass addresses into the system are:

```
bind()  
connect()  
sendmsg()  
sendto()
```

The system will use the `sockaddr_in6` address structure to return addresses to applications that are using PF\_INET6 sockets. The functions that return an address from the system to an application are:

```
accept()  
recvfrom()  
recvmsg()  
getpeername()  
getsockname()
```

No changes to the syntax of the socket functions are needed to support IPv6, since all of the "address carrying" functions use an opaque address pointer, and carry an address length as a function argument.

### 3.6 Compatibility with IPv4 Applications

In order to support the large base of applications using the original API, system implementations must provide complete source and binary compatibility with the original API. This means that systems must continue to support PF\_INET sockets and the `sockaddr_in` address structure. Applications must be able to create IPv4/TCP and IPv4/UDP sockets using the PF\_INET constant in the `socket()` function, as described in the previous section. Applications should be able to hold a combination of IPv4/TCP, IPv4/UDP, IPv6/TCP and IPv6/UDP sockets simultaneously within the same process.

Applications using the original API should continue to operate as they did on systems supporting only IPv4. That is, they should continue to interoperate with IPv4 nodes.

### 3.7 Compatibility with IPv4 Nodes

The API also provides a different type of compatibility: the ability for IPv6 applications to interoperate with IPv4 applications. This feature uses the IPv4-mapped IPv6 address format defined in the IPv6 addressing architecture specification [2]. This address format

allows the IPv4 address of an IPv4 node to be represented as an IPv6 address. The IPv4 address is encoded into the low-order 32 bits of the IPv6 address, and the high-order 96 bits hold the fixed prefix 0:0:0:0:0:FFFF. IPv4-mapped addresses are written as follows:

```
::FFFF:<IPv4-address>
```

These addresses can be generated automatically by the `getipnodebyname()` function when the specified host has only IPv4 addresses (as described in Section 6.1).

Applications may use `PF_INET6` sockets to open TCP connections to IPv4 nodes, or send UDP packets to IPv4 nodes, by simply encoding the destination's IPv4 address as an IPv4-mapped IPv6 address, and passing that address, within a `sockaddr_in6` structure, in the `connect()` or `sendto()` call. When applications use `PF_INET6` sockets to accept TCP connections from IPv4 nodes, or receive UDP packets from IPv4 nodes, the system returns the peer's address to the application in the `accept()`, `recvfrom()`, or `getpeername()` call using a `sockaddr_in6` structure encoded this way.

Few applications will likely need to know which type of node they are interoperating with. However, for those applications that do need to know, the `IN6_IS_ADDR_V4MAPPED()` macro, defined in Section 6.7, is provided.

### 3.8 IPv6 Wildcard Address

While the `bind()` function allows applications to select the source IP address of UDP packets and TCP connections, applications often want the system to select the source address for them. With IPv4, one specifies the address as the symbolic constant `INADDR_ANY` (called the "wildcard" address) in the `bind()` call, or simply omits the `bind()` entirely.

Since the IPv6 address type is a structure (`struct in6_addr`), a symbolic constant can be used to initialize an IPv6 address variable, but cannot be used in an assignment. Therefore systems provide the IPv6 wildcard address in two forms.

The first version is a global variable named `"in6addr_any"` that is an `in6_addr` structure. The extern declaration for this variable is defined in `<netinet/in.h>`:

```
extern const struct in6_addr in6addr_any;
```

Applications use `in6addr_any` similarly to the way they use `INADDR_ANY` in IPv4. For example, to bind a socket to port number 23, but let the system select the source address, an application could use the following code:

```
struct sockaddr_in6 sin6;
. . .
sin6.sin6_family = AF_INET6;
sin6.sin6_flowinfo = 0;
sin6.sin6_port = htons(23);
sin6.sin6_addr = in6addr_any; /* structure assignment */
. . .
if (bind(s, (struct sockaddr *) &sin6, sizeof(sin6)) == -1)
. . .
```

The other version is a symbolic constant named `IN6ADDR_ANY_INIT` and is defined in `<netinet/in.h>`. This constant can be used to initialize an `in6_addr` structure:

```
struct in6_addr anyaddr = IN6ADDR_ANY_INIT;
```

Note that this constant can be used ONLY at declaration time. It can not be used to assign a previously declared `in6_addr` structure. For example, the following code will not work:

```
/* This is the WRONG way to assign an unspecified address */
struct sockaddr_in6 sin6;
. . .
sin6.sin6_addr = IN6ADDR_ANY_INIT; /* will NOT compile */
```

Be aware that the IPv4 `INADDR_xxx` constants are all defined in host byte order but the IPv6 `IN6ADDR_xxx` constants and the IPv6 `in6addr_xxx` externals are defined in network byte order.

### 3.9 IPv6 Loopback Address

Applications may need to send UDP packets to, or originate TCP connections to, services residing on the local node. In IPv4, they can do this by using the constant IPv4 address `INADDR_LOOPBACK` in their `connect()`, `sendto()`, or `sendmsg()` call.

IPv6 also provides a loopback address to contact local TCP and UDP services. Like the unspecified address, the IPv6 loopback address is provided in two forms -- a global variable and a symbolic constant.

The global variable is an `in6_addr` structure named `"in6addr_loopback."` The extern declaration for this variable is defined in `<netinet/in.h>`:

```
extern const struct in6_addr in6addr_loopback;
```

Applications use `in6addr_loopback` as they would use `INADDR_LOOPBACK` in IPv4 applications (but beware of the byte ordering difference mentioned at the end of the previous section). For example, to open a TCP connection to the local telnet server, an application could use the following code:

```
struct sockaddr_in6 sin6;
. . .
sin6.sin6_family = AF_INET6;
sin6.sin6_flowinfo = 0;
sin6.sin6_port = htons(23);
sin6.sin6_addr = in6addr_loopback; /* structure assignment */
. . .
if (connect(s, (struct sockaddr *) &sin6, sizeof(sin6)) == -1)
. . .
```

The symbolic constant is named `IN6ADDR_LOOPBACK_INIT` and is defined in `<netinet/in.h>`. It can be used at declaration time ONLY; for example:

```
struct in6_addr loopbackaddr = IN6ADDR_LOOPBACK_INIT;
```

Like `IN6ADDR_ANY_INIT`, this constant cannot be used in an assignment to a previously declared IPv6 address variable.

### 3.10 Portability Additions

One simple addition to the sockets API that can help application writers is the `"struct sockaddr_storage"`. This data structure can simplify writing code portable across multiple address families and platforms. This data structure is designed with the following goals.

- It has a large enough implementation specific maximum size to store the desired set of protocol specific socket address data structures. Specifically, it is at least large enough to accommodate `sockaddr_in` and `sockaddr_in6` and possibly other protocol specific socket addresses too.
- It is aligned at an appropriate boundary so protocol specific socket address data structure pointers can be cast to it and access their fields without alignment problems. (e.g. pointers to `sockaddr_in6` and/or `sockaddr_in` can be cast to it and access fields without alignment problems).

- It has the initial field(s) isomorphic to the fields of the "struct sockaddr" data structure on that implementation which can be used as a discriminants for deriving the protocol in use. These initial field(s) would on most implementations either be a single field of type "sa\_family\_t" (isomorphic to sa\_family field, 16 bits) or two fields of type uint8\_t and sa\_family\_t respectively, (isomorphic to sa\_len and sa\_family\_t, 8 bits each).

An example implementation design of such a data structure would be as follows.

```
/*
 * Desired design of maximum size and alignment
 */
#define _SS_MAXSIZE      128 /* Implementation specific max size */
#define _SS_ALIGNSIZE    (sizeof (int64_t))
                        /* Implementation specific desired alignment */
/*
 * Definitions used for sockaddr_storage structure paddings design.
 */
#define _SS_PAD1SIZE     (_SS_ALIGNSIZE - sizeof (sa_family_t))
#define _SS_PAD2SIZE     (_SS_MAXSIZE - (sizeof (sa_family_t)+
                        _SS_PAD1SIZE + _SS_ALIGNSIZE))
struct sockaddr_storage {
    sa_family_t  __ss_family;      /* address family */
    /* Following fields are implementation specific */
    char         __ss_pad1[_SS_PAD1SIZE];
                /* 6 byte pad, this is to make implementation
                /* specific pad up to alignment field that */
                /* follows explicit in the data structure */
    int64_t      __ss_align;       /* field to force desired structure */
                /* storage alignment */
    char         __ss_pad2[_SS_PAD2SIZE];
                /* 112 byte pad to achieve desired size, */
                /* _SS_MAXSIZE value minus size of ss_family */
                /* __ss_pad1, __ss_align fields is 112 */
};
```

On implementations where sockaddr data structure includes a "sa\_len", field this data structure would look like this:

```
/*
 * Definitions used for sockaddr_storage structure paddings design.
 */
#define _SS_PAD1SIZE     (_SS_ALIGNSIZE -
                        (sizeof (uint8_t) + sizeof (sa_family_t))
#define _SS_PAD2SIZE     (_SS_MAXSIZE - (sizeof (sa_family_t)+
```

```

                                _SS_PAD1SIZE + _SS_ALIGNSIZE))
struct sockaddr_storage {
    uint8_t    __ss_len;          /* address length */
    sa_family_t __ss_family;      /* address family */
    /* Following fields are implementation specific */
    char       __ss_pad1[_SS_PAD1SIZE];
                /* 6 byte pad, this is to make implementation
                /* specific pad up to alignment field that */
                /* follows explicit in the data structure */
    int64_t    __ss_align; /* field to force desired structure */
                /* storage alignment */
    char       __ss_pad2[_SS_PAD2SIZE];
                /* 112 byte pad to achieve desired size, */
                /* _SS_MAXSIZE value minus size of ss_len, */
                /* __ss_family, __ss_pad1, __ss_align fields is 112 */
};

```

The above example implementation illustrates a data structure which will align on a 64 bit boundary. An implementation specific field "`__ss_align`" along "`__ss_pad1`" is used to force a 64-bit alignment which covers proper alignment good enough for needs of `sockaddr_in6` (IPv6), `sockaddr_in` (IPv4) address data structures. The size of padding fields `__ss_pad1` depends on the chosen alignment boundary. The size of padding field `__ss_pad2` depends on the value of overall size chosen for the total size of the structure. This size and alignment are represented in the above example by implementation specific (not required) constants `_SS_MAXSIZE` (chosen value 128) and `_SS_ALIGNMENT` (with chosen value 8). Constants `_SS_PAD1SIZE` (derived value 6) and `_SS_PAD2SIZE` (derived value 112) are also for illustration and not required. The implementation specific definitions and structure field names above start with an underscore to denote implementation private namespace. Portable code is not expected to access or reference those fields or constants.

The `sockaddr_storage` structure solves the problem of declaring storage for automatic variables which is large enough and aligned enough for storing socket address data structure of any family. For example, code with a file descriptor and without the context of the address family can pass a pointer to a variable of this type where a pointer to a socket address structure is expected in calls such as `getpeername()` and determine the address family by accessing the received content after the call.

The `sockaddr_storage` structure may also be useful and applied to certain other interfaces where a generic socket address large enough and aligned for use with multiple address families may be needed. A discussion of those interfaces is outside the scope of this document.

Also, much existing code assumes that any socket address structure can fit in a generic `sockaddr` structure. While this has been true for IPv4 socket address structures, it has always been false for Unix domain socket address structures (but in practice this has not been a problem) and it is also false for IPv6 socket address structures (which can be a problem).

So now an application can do the following:

```
struct sockaddr_storage __ss;
struct sockaddr_in6 *sin6;
sin6 = (struct sockaddr_in6 *) &__ss;
```

#### 4. Interface Identification

This API uses an interface index (a small positive integer) to identify the local interface on which a multicast group is joined (Section 5.3). Additionally, the advanced API [4] uses these same interface indexes to identify the interface on which a datagram is received, or to specify the interface on which a datagram is to be sent.

Interfaces are normally known by names such as "le0", "sl1", "ppp2", and the like. On Berkeley-derived implementations, when an interface is made known to the system, the kernel assigns a unique positive integer value (called the interface index) to that interface. These are small positive integers that start at 1. (Note that 0 is never used for an interface index.) There may be gaps so that there is no current interface for a particular positive interface index.

This API defines two functions that map between an interface name and index, a third function that returns all the interface names and indexes, and a fourth function to return the dynamic memory allocated by the previous function. How these functions are implemented is left up to the implementation. 4.4BSD implementations can implement these functions using the existing `sysctl()` function with the `NET_RT_IFLIST` command. Other implementations may wish to use `ioctl()` for this purpose.

##### 4.1 Name-to-Index

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

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

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



If the specified interface name does not exist, the return value is 0, and `errno` is set to `ENXIO`. If there was a system error (such as running out of memory), the return value is 0 and `errno` is set to the proper value (e.g., `ENOMEM`).

## 4.2 Index-to-Name

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

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

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

The `ifname` argument must point to a buffer of at least `IF_NAMESIZE` bytes into which the interface name corresponding to the specified index is returned. (`IF_NAMESIZE` is also defined in `<net/if.h>` and its value includes a terminating null byte at the end of the interface name.) This pointer is also the return value of the function. If there is no interface corresponding to the specified index, `NULL` is returned, and `errno` is set to `ENXIO`, if there was a system error (such as running out of memory), `if_indextoname` returns `NULL` and `errno` would be set to the proper value (e.g., `ENOMEM`).

## 4.3 Return All Interface Names and Indexes

The `if_nameindex` structure holds the information about a single interface and is defined as a result of including the `<net/if.h>` header.

```
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, one structure per interface.

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

The end of the array of structures is indicated by a structure with an `if_index` of 0 and an `if_name` of `NULL`. The function returns a `NULL` pointer upon an error, and would set `errno` to the appropriate value.

The memory used for this array of structures along with the interface names pointed to by the `if_name` members is obtained dynamically. This memory is freed by the next function.

#### 4.4 Free Memory

The following function frees the dynamic memory that was allocated by `if_nameindex()`.

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

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

The argument to this function must be a pointer that was returned by `if_nameindex()`.

Currently `net/if.h` doesn't have prototype definitions for functions and it is recommended that these definitions be defined in `net/if.h` as well and the `struct if_nameindex{}`.

#### 5. Socket Options

A number of new socket options are defined for IPv6. All of these new options are at the `IPPROTO_IPV6` level. That is, the "level" parameter in the `getsockopt()` and `setsockopt()` calls is `IPPROTO_IPV6` when using these options. The constant name prefix `IPV6_` is used in all of the new socket options. This serves to clearly identify these options as applying to IPv6.

The declaration for `IPPROTO_IPV6`, the new IPv6 socket options, and related constants defined in this section are obtained by including the header `<netinet/in.h>`.

##### 5.1 Unicast Hop Limit

A new `setsockopt()` option controls the hop limit used in outgoing unicast IPv6 packets. The name of this option is `IPV6_UNICAST_HOPS`, and it is used at the `IPPROTO_IPV6` layer. The following example illustrates how it is used:

```
int hoplimit = 10;

if (setsockopt(s, IPPROTO_IPV6, IPV6_UNICAST_HOPS,
              (char *) &hoplimit, sizeof(hoplimit)) == -1)
    perror("setsockopt IPV6_UNICAST_HOPS");
```

When the `IPV6_UNICAST_HOPS` option is set with `setsockopt()`, the option value given is used as the hop limit for all subsequent unicast packets sent via that socket. If the option is not set, the system selects a default value. The integer hop limit value (called `x`) is interpreted as follows:

```
x < -1:      return an error of EINVAL
x == -1:      use kernel default
0 <= x <= 255: use x
x >= 256:     return an error of EINVAL
```

The `IPV6_UNICAST_HOPS` option may be used with `getsockopt()` to determine the hop limit value that the system will use for subsequent unicast packets sent via that socket. For example:

```
int  hoplimit;
size_t len = sizeof(hoplimit);

if (getsockopt(s, IPPROTO_IPV6, IPV6_UNICAST_HOPS,
               (char *) &hoplimit, &len) == -1)
    perror("getsockopt IPV6_UNICAST_HOPS");
else
    printf("Using %d for hop limit.\n", hoplimit);
```

## 5.2 Sending and Receiving Multicast Packets

IPv6 applications may send UDP multicast packets by simply specifying an IPv6 multicast address in the address argument of the `sendto()` function.

Three socket options at the `IPPROTO_IPV6` layer control some of the parameters for sending multicast packets. Setting these options is not required: applications may send multicast packets without using these options. The `setsockopt()` options for controlling the sending of multicast packets are summarized below. These three options can also be used with `getsockopt()`.

### `IPV6_MULTICAST_IF`

Set the interface to use for outgoing multicast packets. The argument is the index of the interface to use.

Argument type: unsigned int

### `IPV6_MULTICAST_HOPS`

Set the hop limit to use for outgoing multicast packets. (Note a separate option - `IPV6_UNICAST_HOPS` - is provided to set the hop limit to use for outgoing unicast packets.)

The interpretation of the argument is the same as for the `IPV6_UNICAST_HOPS` option:

```
x < -1:      return an error of EINVAL
x == -1:      use kernel default
0 <= x <= 255: use x
x >= 256:     return an error of EINVAL
```

If IPV6\_MULTICAST\_HOPS is not set, the default is 1  
(same as IPv4 today)

Argument type: int

#### IPV6\_MULTICAST\_LOOP

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is looped back by the IP layer for local delivery if this option is set to 1. If this option is set to 0 a copy is not looped back. Other option values return an error of EINVAL.

If IPV6\_MULTICAST\_LOOP is not set, the default is 1 (loopback; same as IPv4 today).

Argument type: unsigned int

The reception of multicast packets is controlled by the two `setsockopt()` options summarized below. An error of `EOPNOTSUPP` is returned if these two options are used with `getsockopt()`.

#### IPV6\_JOIN\_GROUP

Join a multicast group on a specified local interface. If the interface index is specified as 0, the kernel chooses the local interface. For example, some kernels look up the multicast group in the normal IPv6 routing table and using the resulting interface.

Argument type: struct `ipv6_mreq`

#### IPV6\_LEAVE\_GROUP

Leave a multicast group on a specified interface.

Argument type: struct `ipv6_mreq`

The argument type of both of these options is the `ipv6_mreq` structure, defined as a result of including the `<netinet/in.h>` header;

```
struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr; /* IPv6 multicast addr */
    unsigned int     ipv6mr_interface; /* interface index */
};
```

Note that to receive multicast datagrams a process must join the multicast group and bind the UDP port to which datagrams will be sent. Some processes also bind the multicast group address to the socket, in addition to the port, to prevent other datagrams destined to that same port from being delivered to the socket.

## 6. Library Functions

New library functions are needed to perform a variety of operations with IPv6 addresses. Functions are needed to lookup IPv6 addresses in the Domain Name System (DNS). Both forward lookup (nodename-to-address translation) and reverse lookup (address-to-nodename translation) need to be supported. Functions are also needed to convert IPv6 addresses between their binary and textual form.

We note that the two existing functions, `gethostbyname()` and `gethostbyaddr()`, are left as-is. New functions are defined to handle both IPv4 and IPv6 addresses.

### 6.1 Nodename-to-Address Translation

The commonly used function `gethostbyname()` is inadequate for many applications, first because it provides no way for the caller to specify anything about the types of addresses desired (IPv4 only, IPv6 only, IPv4-mapped IPv6 are OK, etc.), and second because many implementations of this function are not thread safe. RFC 2133 defined a function named `gethostbyname2()` but this function was also inadequate, first because its use required setting a global option (`RES_USE_INET6`) when IPv6 addresses were required, and second because a flag argument is needed to provide the caller with additional control over the types of addresses required.

The following function is new and must be thread safe:

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

struct hostent *getipnodebyname(const char *name, int af, int flags
                                int *error_num);
```

The name argument can be either a node name or a numeric address string (i.e., a dotted-decimal IPv4 address or an IPv6 hex address). The af argument specifies the address family, either `AF_INET` or

AF\_INET6. The error\_num value is returned to the caller, via a pointer, with the appropriate error code in error\_num, to support thread safe error code returns. error\_num will be set to one of the following values:

HOST\_NOT\_FOUND

No such host is known.

NO\_ADDRESS

The server recognised the request and the name but no address is available. Another type of request to the name server for the domain might return an answer.

NO\_RECOVERY

An unexpected server failure occurred which cannot be recovered.

TRY\_AGAIN

A temporary and possibly transient error occurred, such as a failure of a server to respond.

The flags argument specifies the types of addresses that are searched for, and the types of addresses that are returned. We note that a special flags value of AI\_DEFAULT (defined below) should handle most applications.

That is, porting simple applications to use IPv6 replaces the call

```
hptr = gethostbyname(name);
```

with

```
hptr = getipnodebyname(name, AF_INET6, AI_DEFAULT, &error_num);
```

and changes any subsequent error diagnosis code to use error\_num instead of externally declared variables, such as h\_errno.

Applications desiring finer control over the types of addresses searched for and returned, can specify other combinations of the flags argument.

A flags of 0 implies a strict interpretation of the af argument:

- If flags is 0 and af is AF\_INET, then the caller wants only IPv4 addresses. A query is made for A records. If successful, the IPv4 addresses are returned and the h\_length member of the hostent structure will be 4, else the function returns a NULL pointer.
- If flags is 0 and if af is AF\_INET6, then the caller wants only IPv6 addresses. A query is made for AAAA records. If successful, the IPv6 addresses are returned and the h\_length member of the hostent structure will be 16, else the function returns a NULL pointer.

Other constants can be logically-ORed into the flags argument, to modify the behavior of the function.

- If the AI\_V4MAPPED flag is specified along with an af of AF\_INET6, then the caller will accept IPv4-mapped IPv6 addresses. That is, if no AAAA records are found then a query is made for A records and any found are returned as IPv4-mapped IPv6 addresses (h\_length will be 16). The AI\_V4MAPPED flag is ignored unless af equals AF\_INET6.
- The AI\_ALL flag is used in conjunction with the AI\_V4MAPPED flag, and is only used with the IPv6 address family. When AI\_ALL is logically or'd with AI\_V4MAPPED flag then the caller wants all addresses: IPv6 and IPv4-mapped IPv6. A query is first made for AAAA records and if successful, the IPv6 addresses are returned. Another query is then made for A records and any found are returned as IPv4-mapped IPv6 addresses. h\_length will be 16. Only if both queries fail does the function return a NULL pointer. This flag is ignored unless af equals AF\_INET6.
- The AI\_ADDRCONFIG flag specifies that a query for AAAA records should occur only if the node has at least one IPv6 source address configured and a query for A records should occur only if the node has at least one IPv4 source address configured.

For example, if the node has no IPv6 source addresses configured, and af equals AF\_INET6, and the node name being looked up has both AAAA and A records, then:

- (a) if only AI\_ADDRCONFIG is specified, the function returns a NULL pointer;
- (b) if AI\_ADDRCONFIG | AI\_V4MAPPED is specified, the A records are returned as IPv4-mapped IPv6 addresses;

The special flags value of `AI_DEFAULT` is defined as

```
#define AI_DEFAULT (AI_V4MAPPED | AI_ADDRCONFIG)
```

We noted that the `getipnodebyname()` function must allow the name argument to be either a node name or a literal address string (i.e., a dotted-decimal IPv4 address or an IPv6 hex address). This saves applications from having to call `inet_pton()` to handle literal address strings.

There are four scenarios based on the type of literal address string and the value of the `af` argument.

The two simple cases are:

When name is a dotted-decimal IPv4 address and `af` equals `AF_INET`, or when name is an IPv6 hex address and `af` equals `AF_INET6`. The members of the returned `hostent` structure are: `h_name` points to a copy of the name argument, `h_aliases` is a NULL pointer, `h_addrtype` is a copy of the `af` argument, `h_length` is either 4 (for `AF_INET`) or 16 (for `AF_INET6`), `h_addr_list[0]` is a pointer to the 4-byte or 16-byte binary address, and `h_addr_list[1]` is a NULL pointer.

When name is a dotted-decimal IPv4 address and `af` equals `AF_INET6`, and flags equals `AI_V4MAPPED`, an IPv4-mapped IPv6 address is returned: `h_name` points to an IPv6 hex address containing the IPv4-mapped IPv6 address, `h_aliases` is a NULL pointer, `h_addrtype` is `AF_INET6`, `h_length` is 16, `h_addr_list[0]` is a pointer to the 16-byte binary address, and `h_addr_list[1]` is a NULL pointer. If `AI_V4MAPPED` is set (with or without `AI_ALL`) return IPv4-mapped otherwise return NULL.

It is an error when name is an IPv6 hex address and `af` equals `AF_INET`. The function's return value is a NULL pointer and `error_num` equals `HOST_NOT_FOUND`.

## 6.2 Address-To-Nodename Translation

The following function has the same arguments as the existing `gethostbyaddr()` function, but adds an error number.

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

struct hostent *getipnodebyaddr(const void *src, size_t len,
                                int af, int *error_num);
```



As with `getipnodebyname()`, `getipnodebyaddr()` must be thread safe. The `error_num` value is returned to the caller with the appropriate error code, to support thread safe error code returns. The following error conditions may be returned for `error_num`:

#### HOST\_NOT\_FOUND

No such host is known.

#### NO\_ADDRESS

The server recognized the request and the name but no address is available. Another type of request to the name server for the domain might return an answer.

#### NO\_RECOVERY

An unexpected server failure occurred which cannot be recovered.

#### TRY\_AGAIN

A temporary and possibly transient error occurred, such as a failure of a server to respond.

One possible source of confusion is the handling of IPv4-mapped IPv6 addresses and IPv4-compatible IPv6 addresses, but the following logic should apply.

1. If `af` is `AF_INET6`, and if `len` equals 16, and if the IPv6 address is an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address, then skip over the first 12 bytes of the IPv6 address, set `af` to `AF_INET`, and set `len` to 4.
2. If `af` is `AF_INET`, lookup the name for the given IPv4 address (e.g., query for a PTR record in the `in-addr.arpa` domain).
3. If `af` is `AF_INET6`, lookup the name for the given IPv6 address (e.g., query for a PTR record in the `ip6.int` domain).
4. If the function is returning success, then the single address that is returned in the `hostent` structure is a copy of the first argument to the function with the same address family that was passed as an argument to this function.

All four steps listed are performed, in order. Also note that the IPv6 hex addresses "::" and "::1" MUST NOT be treated as IPv4-compatible addresses, and if the address is "::", HOST\_NOT\_FOUND MUST be returned and a query of the address not performed.

Also for the macro in section 6.7 IN6\_IS\_ADDR\_V4COMPAT MUST return false for "::" and "::1".

### 6.3 Freeing memory for getipnodebyname and getipnodebyaddr

The hostent structure does not change from its existing definition. This structure, and the information pointed to by this structure, are dynamically allocated by getipnodebyname and getipnodebyaddr. The following function frees this memory:

```
#include <netdb.h>

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

### 6.4 Protocol-Independent Nodename and Service Name Translation

Nodename-to-address translation is done in a protocol-independent fashion using the getaddrinfo() function that is taken from the Institute of Electrical and Electronic Engineers (IEEE) POSIX 1003.1g (Protocol Independent Interfaces) draft specification [3].

The official specification for this function will be the final POSIX standard, with the following additional requirements:

- getaddrinfo() (along with the getnameinfo() function described in the next section) must be thread safe.
- The AI\_NUMERICHOST is new with this document.
- All fields in socket address structures returned by getaddrinfo() that are not filled in through an explicit argument (e.g., sin6\_flowinfo and sin\_zero) must be set to 0. (This makes it easier to compare socket address structures.)
- getaddrinfo() must fill in the length field of a socket address structure (e.g., sin6\_len) on systems that support this field.

We are providing this independent description of the function because POSIX standards are not freely available (as are IETF documents).

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

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

The `addrinfo` structure is defined as a result of including the `<netdb.h>` header.

```
struct addrinfo {
    int     ai_flags;      /* AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST */
    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 */
};
```

The return value from the function is 0 upon success or a nonzero error code. The following names are the nonzero error codes from `getaddrinfo()`, and are defined in `<netdb.h>`:

<code>EAI_ADDRFAMILY</code>	address family for nodename not supported
<code>EAI_AGAIN</code>	temporary failure in name resolution
<code>EAI_BADFLAGS</code>	invalid value for <code>ai_flags</code>
<code>EAI_FAIL</code>	non-recoverable failure in name resolution
<code>EAI_FAMILY</code>	<code>ai_family</code> not supported
<code>EAI_MEMORY</code>	memory allocation failure
<code>EAI_NODATA</code>	no address associated with nodename
<code>EAI_NONAME</code>	nodename nor servname provided, or not known
<code>EAI_SERVICE</code>	servname not supported for <code>ai_socktype</code>
<code>EAI_SOCKTYPE</code>	<code>ai_socktype</code> not supported
<code>EAI_SYSTEM</code>	system error returned in <code>errno</code>

The `nodename` and `servname` arguments are pointers to null-terminated strings or `NULL`. One or both of these two arguments must be a non-`NULL` pointer. In the normal client scenario, both the `nodename` and `servname` are specified. In the normal server scenario, only the `servname` is specified. A non-`NULL` `nodename` string can be either a node name or a numeric host address string (i.e., a dotted-decimal IPv4 address or an IPv6 hex address). A non-`NULL` `servname` string can be either a service name or a decimal port number.

The caller can optionally pass an `addrinfo` structure, pointed to by the third argument, to provide hints concerning the type of socket that the caller supports. In this hints structure all members other than `ai_flags`, `ai_family`, `ai_socktype`, and `ai_protocol` must be zero or a `NULL` pointer. A value of `PF_UNSPEC` for `ai_family` means the

caller will accept any protocol family. A value of 0 for `ai_socktype` means the caller will accept any socket type. A value of 0 for `ai_protocol` means the caller will accept any protocol. For example, if the caller handles only TCP and not UDP, then the `ai_socktype` member of the hints structure should be set to `SOCK_STREAM` when `getaddrinfo()` is called. If the caller handles only IPv4 and not IPv6, then the `ai_family` member of the hints structure should be set to `PF_INET` when `getaddrinfo()` is called. If the third argument to `getaddrinfo()` is a NULL pointer, this is the same as if the caller had filled in an `addrinfo` structure initialized to zero with `ai_family` set to `PF_UNSPEC`.

Upon successful return a pointer to a linked list of one or more `addrinfo` structures is returned through the final argument. The caller can process each `addrinfo` structure in this list by following the `ai_next` pointer, until a NULL pointer is encountered. In each returned `addrinfo` structure the three members `ai_family`, `ai_socktype`, and `ai_protocol` are the corresponding arguments for a call to the `socket()` function. In each `addrinfo` structure the `ai_addr` member points to a filled-in socket address structure whose length is specified by the `ai_addrlen` member.

If the `AI_PASSIVE` bit is set in the `ai_flags` member of the hints structure, then the caller plans to use the returned socket address structure in a call to `bind()`. In this case, if the `nodename` argument is a NULL pointer, then the IP address portion of the socket address structure will be set to `INADDR_ANY` for an IPv4 address or `IN6ADDR_ANY_INIT` for an IPv6 address.

If the `AI_PASSIVE` bit is not set in the `ai_flags` member of the hints structure, then the returned socket address structure will be ready for a call to `connect()` (for a connection-oriented protocol) or either `connect()`, `sendto()`, or `sendmsg()` (for a connectionless protocol). In this case, if the `nodename` argument is a NULL pointer, then the IP address portion of the socket address structure will be set to the loopback address.

If the `AI_CANONNAME` bit is set in the `ai_flags` member of the hints structure, then upon successful return the `ai_canonname` member of the first `addrinfo` structure in the linked list will point to a null-terminated string containing the canonical name of the specified `nodename`.

If the `AI_NUMERICHOST` bit is set in the `ai_flags` member of the hints structure, then a non-NULL `nodename` string must be a numeric host address string. Otherwise an error of `EAI_NONAME` is returned. This flag prevents any type of name resolution service (e.g., the DNS) from being called.

All of the information returned by `getaddrinfo()` is dynamically allocated: the `addrinfo` structures, and the socket address structures and canonical node name strings pointed to by the `addrinfo` structures. To return this information to the system the function `freeaddrinfo()` is called:

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

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

The `addrinfo` structure pointed to by the `ai` argument is freed, along with any dynamic storage pointed to by the structure. This operation is repeated until a `NULL ai_next` pointer is encountered.

To aid applications in printing error messages based on the `EAI_XXX` codes returned by `getaddrinfo()`, the following function is defined.

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

char *gai_strerror(int ecode);
```

The argument is one of the `EAI_XXX` values defined earlier and the return value points to a string describing the error. If the argument is not one of the `EAI_XXX` values, the function still returns a pointer to a string whose contents indicate an unknown error.

## 6.5 Socket Address Structure to Nodename and Service Name

The POSIX 1003.1g specification includes no function to perform the reverse conversion from `getaddrinfo()`: to look up a nodename and service name, given the binary address and port. Therefore, we define the following function:

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

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

This function looks up an IP address and port number provided by the caller in the DNS and system-specific database, and returns text strings for both in buffers provided by the caller. The function indicates successful completion by a zero return value; a non-zero return value indicates failure.

The first argument, `sa`, points to either a `sockaddr_in` structure (for IPv4) or a `sockaddr_in6` structure (for IPv6) that holds the IP address and port number. The `salen` argument gives the length of the `sockaddr_in` or `sockaddr_in6` structure.

The function returns the nodename associated with the IP address in the buffer pointed to by the `host` argument. The caller provides the size of this buffer via the `hostlen` argument. The service name associated with the port number is returned in the buffer pointed to by `serv`, and the `servlen` argument gives the length of this buffer. The caller specifies not to return either string by providing a zero value for the `hostlen` or `servlen` arguments. Otherwise, the caller must provide buffers large enough to hold the nodename and the service name, including the terminating null characters.

Unfortunately most systems do not provide constants that specify the maximum size of either a fully-qualified domain name or a service name. Therefore to aid the application in allocating buffers for these two returned strings the following constants are defined in `<netdb.h>`:

```
#define NI_MAXHOST 1025
#define NI_MAXSERV 32
```

The first value is actually defined as the constant `MAXDNAME` in recent versions of BIND's `<arpa/nameser.h>` header (older versions of BIND define this constant to be 256) and the second is a guess based on the services listed in the current Assigned Numbers RFC.

The final argument is a flag that changes the default actions of this function. By default the fully-qualified domain name (FQDN) for the host is looked up in the DNS and returned. If the flag bit `NI_NOFQDN` is set, only the nodename portion of the FQDN is returned for local hosts.

If the flag bit `NI_NUMERICHOST` is set, or if the host's name cannot be located in the DNS, the numeric form of the host's address is returned instead of its name (e.g., by calling `inet_ntop()` instead of `getipnodebyaddr()`). If the flag bit `NI_NAMEREQD` is set, an error is returned if the host's name cannot be located in the DNS.

If the flag bit `NI_NUMERICSERV` is set, the numeric form of the service address is returned (e.g., its port number) instead of its name. The two `NI_NUMERICxxx` flags are required to support the `"-n"` flag that many commands provide.

A fifth flag bit, `NI_DGRAM`, specifies that the service is a datagram service, and causes `getservbyport()` to be called with a second argument of "udp" instead of its default of "tcp". This is required for the few ports (e.g. 512-514) that have different services for UDP and TCP.

These `NI_xxx` flags are defined in `<netdb.h>` along with the `AI_xxx` flags already defined for `getaddrinfo()`.

## 6.6 Address Conversion Functions

The two functions `inet_addr()` and `inet_ntoa()` convert an IPv4 address between binary and text form. IPv6 applications need similar functions. The following two functions convert both IPv6 and IPv4 addresses:

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

int inet_pton(int af, const char *src, void *dst);

const char *inet_ntop(int af, const void *src,
                      char *dst, size_t size);
```

The `inet_pton()` function converts an address in its standard text presentation form into its numeric binary form. The `af` argument specifies the family of the address. Currently the `AF_INET` and `AF_INET6` address families are supported. The `src` argument points to the string being passed in. The `dst` argument points to a buffer into which the function stores the numeric address. The address is returned in network byte order. `inet_pton()` returns 1 if the conversion succeeds, 0 if the input is not a valid IPv4 dotted-decimal string or a valid IPv6 address string, or -1 with `errno` set to `EAFNOSUPPORT` if the `af` argument is unknown. The calling application must ensure that the buffer referred to by `dst` is large enough to hold the numeric address (e.g., 4 bytes for `AF_INET` or 16 bytes for `AF_INET6`).

If the `af` argument is `AF_INET`, the function accepts a string in the standard IPv4 dotted-decimal form:

`ddd.ddd.ddd.ddd`

where `ddd` is a one to three digit decimal number between 0 and 255. Note that many implementations of the existing `inet_addr()` and `inet_aton()` functions accept nonstandard input: octal numbers, hexadecimal numbers, and fewer than four numbers. `inet_pton()` does not accept these formats.

If the `af` argument is `AF_INET6`, then the function accepts a string in one of the standard IPv6 text forms defined in Section 2.2 of the addressing architecture specification [2].

The `inet_ntop()` function converts a numeric address into a text string suitable for presentation. The `af` argument specifies the family of the address. This can be `AF_INET` or `AF_INET6`. The `src` argument points to a buffer holding an IPv4 address if the `af` argument is `AF_INET`, or an IPv6 address if the `af` argument is `AF_INET6`, the address must be in network byte order. The `dst` argument points to a buffer where the function will store the resulting text string. The `size` argument specifies the size of this buffer. The application must specify a non-NULL `dst` argument. For IPv6 addresses, the buffer must be at least 46-octets. For IPv4 addresses, the buffer must be at least 16-octets. In order to allow applications to easily declare buffers of the proper size to store IPv4 and IPv6 addresses in string form, the following two constants are defined in `<netinet/in.h>`:

```
#define INET_ADDRSTRLEN    16
#define INET6_ADDRSTRLEN   46
```

The `inet_ntop()` function returns a pointer to the buffer containing the text string if the conversion succeeds, and NULL otherwise. Upon failure, `errno` is set to `EAFNOSUPPORT` if the `af` argument is invalid or `ENOSPC` if the size of the result buffer is inadequate.

## 6.7 Address Testing Macros

The following macros can be used to test for special IPv6 addresses.

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

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 *);

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 *);
```



The first seven macros return true if the address is of the specified type, or false otherwise. The last five test the scope of a multicast address and return true if the address is a multicast address of the specified scope or false if the address is either not a multicast address or not of the specified scope. Note that `IN6_IS_ADDR_LINKLOCAL` and `IN6_IS_ADDR_SITELOCAL` return true only for the two local-use IPv6 unicast addresses. These two macros do not return true for IPv6 multicast addresses of either link-local scope or site-local scope.

## 7. Summary of New Definitions

The following list summarizes the constants, structure, and extern definitions discussed in this memo, sorted by header.

```

<net/if.h>      IF_NAMESIZE
<net/if.h>      struct if_nameindex{};

<netdb.h>       AI_ADDRCONFIG
<netdb.h>       AI_DEFAULT
<netdb.h>       AI_ALL
<netdb.h>       AI_CANONNAME
<netdb.h>       AI_NUMERICHOST
<netdb.h>       AI_PASSIVE
<netdb.h>       AI_V4MAPPED
<netdb.h>       EAI_ADDRFAMILY
<netdb.h>       EAI_AGAIN
<netdb.h>       EAI_BADFLAGS
<netdb.h>       EAI_FAIL
<netdb.h>       EAI_FAMILY
<netdb.h>       EAI_MEMORY
<netdb.h>       EAI_NODATA
<netdb.h>       EAI_NONAME
<netdb.h>       EAI_SERVICE
<netdb.h>       EAI_SOCKTYPE
<netdb.h>       EAI_SYSTEM
<netdb.h>       NI_DGRAM
<netdb.h>       NI_MAXHOST
<netdb.h>       NI_MAXSERV
<netdb.h>       NI_NAMEREQD
<netdb.h>       NI_NOFQDN
<netdb.h>       NI_NUMERICHOST
<netdb.h>       NI_NUMERICSERV
<netdb.h>       struct addrinfo{};

<netinet/in.h>  IN6ADDR_ANY_INIT
<netinet/in.h>  IN6ADDR_LOOPBACK_INIT
<netinet/in.h>  INET6_ADDRSTRLEN

```

```

<netinet/in.h>  INET_ADDRSTRLEN
<netinet/in.h>  IPPROTO_IPV6
<netinet/in.h>  IPV6_JOIN_GROUP
<netinet/in.h>  IPV6_LEAVE_GROUP
<netinet/in.h>  IPV6_MULTICAST_HOPS
<netinet/in.h>  IPV6_MULTICAST_IF
<netinet/in.h>  IPV6_MULTICAST_LOOP
<netinet/in.h>  IPV6_UNICAST_HOPS
<netinet/in.h>  SIN6_LEN
<netinet/in.h>  extern const struct in6_addr in6addr_any;
<netinet/in.h>  extern const struct in6_addr in6addr_loopback;
<netinet/in.h>  struct in6_addr{};
<netinet/in.h>  struct ipv6_mreq{};
<netinet/in.h>  struct sockaddr_in6{};

<sys/socket.h>  AF_INET6
<sys/socket.h>  PF_INET6
<sys/socket.h>  struct sockaddr_storage;

```

The following list summarizes the function and macro prototypes discussed in this memo, sorted by header.

```

<arpa/inet.h>  int inet_pton(int, const char *, void *);
<arpa/inet.h>  const char *inet_ntop(int, const void *,
                                char *, size_t);

<net/if.h>     char *if_indextoname(unsigned int, char *);
<net/if.h>     unsigned int if_nametoindex(const char *);
<net/if.h>     void if_freenameindex(struct if_nameindex *);
<net/if.h>     struct if_nameindex *if_nameindex(void);

<netdb.h>      int getaddrinfo(const char *, const char *,
                                const struct addrinfo *,
                                struct addrinfo **);
<netdb.h>      int getnameinfo(const struct sockaddr *, socklen_t,
                                char *, size_t, char *, size_t, int);
<netdb.h>      void freeaddrinfo(struct addrinfo *);
<netdb.h>      char *gai_strerror(int);
<netdb.h>      struct hostent *getipnodebyname(const char *, int, int,
                                                int *);
<netdb.h>      struct hostent *getipnodebyaddr(const void *, size_t,
                                                int, int *);
<netdb.h>      void freehostent(struct hostent *);

<netinet/in.h> int IN6_IS_ADDR_LINKLOCAL(const struct in6_addr *);
<netinet/in.h> int IN6_IS_ADDR_LOOPBACK(const struct in6_addr *);
<netinet/in.h> int IN6_IS_ADDR_MC_GLOBAL(const struct in6_addr *);
<netinet/in.h> int IN6_IS_ADDR_MC_LINKLOCAL(const struct in6_addr *);

```

```
<netinet/in.h>  int  IN6_IS_ADDR_MC_NODELOCAL(const struct in6_addr *);  
<netinet/in.h>  int  IN6_IS_ADDR_MC_ORGLOCAL(const struct in6_addr *);  
<netinet/in.h>  int  IN6_IS_ADDR_MC_SITELOCAL(const struct in6_addr *);  
<netinet/in.h>  int  IN6_IS_ADDR_MULTICAST(const struct in6_addr *);  
<netinet/in.h>  int  IN6_IS_ADDR_SITELOCAL(const struct in6_addr *);  
<netinet/in.h>  int  IN6_IS_ADDR_UNSPECIFIED(const struct in6_addr *);  
<netinet/in.h>  int  IN6_IS_ADDR_V4COMPAT(const struct in6_addr *);  
<netinet/in.h>  int  IN6_IS_ADDR_V4MAPPED(const struct in6_addr *);
```

## 8. Security Considerations

IPv6 provides a number of new security mechanisms, many of which need to be accessible to applications. Companion memos detailing the extensions to the socket interfaces to support IPv6 security are being written.

## 9. Year 2000 Considerations

There are no issues for this memo concerning the Year 2000 issue regarding the use of dates.

## Changes From RFC 2133

Changes made in the March 1998 Edition (-01 draft):

Changed all "hostname" to "nodename" for consistency with other IPv6 documents.

Section 3.3: changed comment for `sin6_flowinfo` to be "traffic class & flow info" and updated corresponding text description to current definition of these two fields.

Section 3.10 ("Portability Additions") is new.

Section 6: a new paragraph was added reiterating that the existing `gethostbyname()` and `gethostbyaddr()` are not changed.

Section 6.1: change `gethostbyname3()` to `getnodebyname()`. Add `AI_DEFAULT` to handle majority of applications. Renamed `AI_V6ADDRCONFIG` to `AI_ADDRCONFIG` and define it for A records and IPv4 addresses too. Defined exactly what `getnodebyname()` must return if the name argument is a numeric address string.

Section 6.2: change `gethostbyaddr()` to `getnodebyaddr()`. Reword items 2 and 3 in the description of how to handle IPv4-mapped and IPv4-compatible addresses to "lookup a name" for a given address, instead of specifying what type of DNS query to issue.

Section 6.3: added two more requirements to `getaddrinfo()`.

Section 7: added the following constants to the list for `<netdb.h>`: `AI_ADDRCONFIG`, `AI_ALL`, and `AI_V4MAPPED`. Add union `sockaddr_union` and `SA_LEN` to the lists for `<sys/socket.h>`.

Updated references.

Changes made in the November 1997 Edition (-00 draft):

The data types have been changed to conform with Draft 6.6 of the Posix 1003.1g standard.

Section 3.2: data type of `s6_addr` changed to `"uint8_t"`.

Section 3.3: data type of `sin6_family` changed to `"sa_family_t"`. data type of `sin6_port` changed to `"in_port_t"`, data type of `sin6_flowinfo` changed to `"uint32_t"`.

Section 3.4: same as Section 3.3, plus data type of `sin6_len` changed to `"uint8_t"`.

Section 6.2: first argument of `gethostbyaddr()` changed from `"const char *"` to `"const void *"` and second argument changed from `"int"` to `"size_t"`.

Section 6.4: second argument of `getnameinfo()` changed from `"size_t"` to `"socklen_t"`.

The wording was changed when new structures were defined, to be more explicit as to which header must be included to define the structure:

Section 3.2 (`in6_addr{}`), Section 3.3 (`sockaddr_in6{}`), Section 3.4 (`sockaddr_in6{}`), Section 4.3 (`if_nameindex{}`), Section 5.3 (`ipv6_mreq{}`), and Section 6.3 (`addrinfo{}`).

Section 4: `NET_RT_LIST` changed to `NET_RT_IFLIST`.

Section 5.1: The `IPV6_ADDRFORM` socket option was removed.

Section 5.3: Added a note that an option value other than 0 or 1 for `IPV6_MULTICAST_LOOP` returns an error. Added a note that `IPV6_MULTICAST_IF`, `IPV6_MULTICAST_HOPS`, and `IPV6_MULTICAST_LOOP` can also be used with `getsockopt()`, but `IPV6_ADD_MEMBERSHIP` and `IPV6_DROP_MEMBERSHIP` cannot be used with `getsockopt()`.

Section 6.1: Removed the description of `gethostbyname2()` and its associated `RES_USE_INET6` option, replacing it with `gethostbyname3()`.

Section 6.2: Added requirement that `gethostbyaddr()` be thread safe. Reworded step 4 to avoid using the `RES_USE_INET6` option.

Section 6.3: Added the requirement that `getaddrinfo()` and `getnameinfo()` be thread safe. Added the `AI_NUMERICHOST` flag.

Section 6.6: Added clarification about `IN6_IS_ADDR_LINKLOCAL` and `IN6_IS_ADDR_SITELOCAL` macros.

Changes made to the draft -01 specification Sept 98

Changed priority to traffic class in the spec.

Added the need for scope identification in section 2.1.

Added `sin6_scope_id` to struct `sockaddr_in6` in sections 3.3 and 3.4.

Changed 3.10 to use generic storage structure to support holding IPv6 addresses and removed the `SA_LEN` macro.

Distinguished between invalid input parameters and system failures for Interface Identification in Section 4.1 and 4.2.

Added defaults for multicast operations in section 5.2 and changed the names from `ADD` to `JOIN` and `DROP` to `LEAVE` to be consistent with IPv6 multicast terminology.

Changed `getnodebyname` to `getipnodebyname`, `getnodebyaddr` to `getipnodebyaddr`, and added MT safe error code to function parameters in section 6.

Moved `freehostent` to its own sub-section after `getipnodebyaddr` now 6.3 (so this bumps all remaining sections in section 6).

Clarified the use of `AI_ALL` and `AI_V4MAPPED` that these are dependent on the `AF` parameter and must be used as a conjunction in section 6.1.

Removed the restriction that literal addresses cannot be used with a flags argument in section 6.1.

Added Year 2000 Section to the draft

Deleted Reference to the following because the attached is deleted from the ID directory and has expired. But the logic from the aforementioned draft still applies, so that was kept in Section 6.2 bullets after 3rd paragraph.

- [7] P. Vixie, "Reverse Name Lookups of Encapsulated IPv4 Addresses in IPv6", Internet-Draft, <draft-vixie-ipng-ipv4ptr-00.txt>, May 1996.

Deleted the following reference as it is no longer referenced. And the draft has expired.

- [3] D. McDonald, "A Simple IP Security API Extension to BSD Sockets", Internet-Draft, <draft-mcdonald-simple-ipsec-api-01.txt>, March 1997.

Deleted the following reference as it is no longer referenced.

- [4] C. Metz, "Network Security API for Sockets", Internet-Draft, <draft-metz-net-security-api-01.txt>, January 1998.

Update current references to current status.

Added alignment notes for `in6_addr` and `sin6_addr`.

Clarified further that `AI_V4MAPPED` must be used with a dotted IPv4 literal address for `getipnodebyname()`, when address family is `AF_INET6`.

Added text to clarify `:::` and `:::1` when used by `getipnodebyaddr()`.

## Acknowledgments

Thanks to the many people who made suggestions and provided feedback to this document, including: Werner Almesberger, Ran Atkinson, Fred Baker, Dave Borman, Andrew Cherenon, Alex Conta, Alan Cox, Steve Deering, Richard Draves, Francis Dupont, Robert Elz, Marc Hasson, Tom Herbert, Bob Hinden, Wan-Yen Hsu, Christian Huitema, Koji Imada, Markus Jork, Ron Lee, Alan Lloyd, Charles Lynn, Dan McDonald, Dave Mitton, Thomas Narten, Josh Osborne, Craig Partridge, Jean-Luc Richier, Erik Scoredos, Keith Sklower, Matt Thomas, Harvey Thompson, Dean D. Throop, Karen Tracey, Glenn Trewitt, Paul Vixie, David Waitzman, Carl Williams, and Kazu Yamamoto,

The `getaddrinfo()` and `getnameinfo()` functions are taken from an earlier Internet Draft by Keith Sklower. As noted in that draft, William Durst, Steven Wise, Michael Karels, and Eric Allman provided many useful discussions on the subject of protocol-independent name-to-address translation, and reviewed early versions of Keith Sklower's original proposal. Eric Allman implemented the first prototype of `getaddrinfo()`. The observation that specifying the pair of name and service would suffice for connecting to a service independent of protocol details was made by Marshall Rose in a proposal to X/Open for a "Uniform Network Interface".

Craig Metz, Jack McCann, Erik Nordmark, Tim Hartrick, and Mukesh Kacker made many contributions to this document. Ramesh Govindan made a number of contributions and co-authored an earlier version of this memo.

## References

- [1] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.
- [2] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 2373, July 1998.
- [3] IEEE, "Protocol Independent Interfaces", IEEE Std 1003.1g, DRAFT 6.6, March 1997.
- [4] Stevens, W. and M. Thomas, "Advanced Sockets API for IPv6", RFC 2292, February 1998.

## Authors' Addresses

Robert E. Gilligan  
FreeGate Corporation  
1208 E. Arques Ave.  
Sunnyvale, CA 94086

Phone: +1 408 617 1004  
EMail: gilligan@freegate.com

Susan Thomson  
Bell Communications Research  
MRE 2P-343, 445 South Street  
Morristown, NJ 07960

Phone: +1 201 829 4514  
EMail: set@thumper.bellcore.com

Jim Bound  
Compaq Computer Corporation  
110 Spitbrook Road ZK3-3/U14  
Nashua, NH 03062-2698

Phone: +1 603 884 0400  
EMail: bound@zk3.dec.com

W. Richard Stevens  
1202 E. Paseo del Zorro  
Tucson, AZ 85718-2826

Phone: +1 520 297 9416  
EMail: rstevens@kohala.com



## Full Copyright Statement

Copyright (C) The Internet Society (1999). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

