

Network Working Group  
Request for Comments: 2133  
Category: Informational

R. Gilligan  
Freegate  
S. Thomson  
Bellcore  
J. Bound  
Digital  
W. Stevens  
Consultant  
April 1997

## Basic Socket Interface Extensions for IPv6

### Status of this Memo

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

### 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 [5].

### Table of Contents

1. Introduction .....	2
2. Design Considerations .....	3
2.1. What Needs to be Changed .....	3
2.2. Data Types .....	5
2.3. Headers .....	5
2.4. Structures .....	5
3. Socket Interface .....	5
3.1. IPv6 Address Family and Protocol Family .....	5
3.2. IPv6 Address Structure .....	6

3.3.	Socket Address Structure for 4.3BSD-Based Systems .....	6
3.4.	Socket Address Structure for 4.4BSD-Based Systems .....	7
3.5.	The Socket Functions .....	8
3.6.	Compatibility with IPv4 Applications .....	9
3.7.	Compatibility with IPv4 Nodes .....	9
3.8.	IPv6 Wildcard Address .....	10
3.9.	IPv6 Loopback Address .....	11
4.	Interface Identification .....	12
4.1.	Name-to-Index .....	13
4.2.	Index-to-Name .....	13
4.3.	Return All Interface Names and Indexes .....	14
4.4.	Free Memory .....	14
5.	Socket Options .....	14
5.1.	Changing Socket Type .....	15
5.2.	Unicast Hop Limit .....	16
5.3.	Sending and Receiving Multicast Packets .....	17
6.	Library Functions .....	19
6.1.	Hostname-to-Address Translation .....	19
6.2.	Address To Hostname Translation .....	22
6.3.	Protocol-Independent Hostname and Service Name Translation .....	22
6.4.	Socket Address Structure to Hostname and Service Name .....	25
6.5.	Address Conversion Functions .....	27
6.6.	Address Testing Macros .....	28
7.	Summary of New Definitions .....	29
8.	Security Considerations .....	31
9.	Acknowledgments .....	31
10.	References .....	31
11.	Authors' Addresses .....	32

## 1. Introduction

While IPv4 addresses are 32 bits long, IPv6 interfaces are identified by 128-bit addresses. The socket interface make 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., flow label and priority), 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.
- 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.

The name-to-address translation functions in the socket interface are gethostbyname() and gethostbyaddr(). These must be modified to support IPv6 and the semantics defined must provide 100% backward compatibility for all existing IPv4 applications, along with IPv6 support for new applications. Additionally, the POSIX 1003.g work in progress [4] specifies a new hostname-to-address translation function which is protocol independent. This function can also be used with 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 flow label, priority, 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 [5].

## 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, POSIX 1003.1g data types are used: `u_intN_t` means an unsigned integer of exactly N bits (e.g., `u_int16_t`) and `u_intNm_t` means an unsigned integer of at least N bits (e.g., `u_int32m_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.

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 data structure to hold a single IPv6 address is defined as follows:

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

struct in6_addr {
    u_int8_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.

### 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 structure is defined to carry IPv6 addresses:

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

struct sockaddr_in6 {
    u_int16m_t    sin6_family;    /* AF_INET6 */
    u_int16m_t    sin6_port;      /* transport layer port # */
    u_int32m_t    sin6_flowinfo;  /* IPv6 flow information */
    struct in6_addr sin6_addr;     /* IPv6 address */
};
```

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 24-bit IPv6 flow label and the 4-bit priority field. The contents and interpretation of this member is unspecified at this time.

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 the `sin6_addr` field will be aligned on a 64-bit boundary. This is done for optimum performance on 64-bit architectures.

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:

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

#define SIN6_LEN

struct sockaddr_in6 {
    u_char      sin6_len;           /* length of this struct */
    u_char      sin6_family;        /* AF_INET6 */
    u_int16m_t  sin6_port;          /* transport layer port # */
    u_int32m_t  sin6_flowinfo;      /* IPv6 flow information */
    struct in6_addr sin6_addr;      /* IPv6 address */
};
```

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 are often generated automatically by the `gethostbyname()` 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 6.6, 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.

#### 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 [5] 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_LIST` 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 does not exist, the return value is 0.

#### 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 `IFNAMSIZ` bytes into which the interface name corresponding to the specified index is returned. (`IFNAMSIZ` 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.

#### 4.3. Return All Interface Names and Indexes

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

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

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

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.

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()`.

#### 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. Changing Socket Type

Unix allows open sockets to be passed between processes via the `exec()` call and other means. It is a relatively common application practice to pass open sockets across `exec()` calls. Thus it is possible for an application using the original API to pass an open `PF_INET` socket to an application that is expecting to receive a `PF_INET6` socket. Similarly, it is possible for an application using the extended API to pass an open `PF_INET6` socket to an application using the original API, which would be equipped only to deal with `PF_INET` sockets. Either of these cases could cause problems, because the application that is passed the open socket might not know how to decode the address structures returned in subsequent socket functions.

To remedy this problem, a new `setsockopt()` option is defined that allows an application to "convert" a `PF_INET6` socket into a `PF_INET` socket and vice versa.

An IPv6 application that is passed an open socket from an unknown process may use the `IPV6_ADDRFORM` `setsockopt()` option to "convert" the socket to `PF_INET6`. Once that has been done, the system will return `sockaddr_in6` address structures in subsequent socket functions.

An IPv6 application that is about to pass an open `PF_INET6` socket to a program that is not be IPv6 capable can "downgrade" the socket to `PF_INET` before calling `exec()`. After that, the system will return `sockaddr_in` address structures to the application that was `exec()`'ed. Be aware that you cannot downgrade an IPv6 socket to an IPv4 socket unless all nonwildcard addresses already associated with the IPv6 socket are IPv4-mapped IPv6 addresses.

The `IPV6_ADDRFORM` option is valid at both the `IPPROTO_IP` and `IPPROTO_IPV6` levels. The only valid option values are `PF_INET6` and `PF_INET`. For example, to convert a `PF_INET6` socket to `PF_INET`, a program would call:

```
int  addrform = PF_INET;

if (setsockopt(s, IPPROTO_IPV6, IPV6_ADDRFORM,
              (char *) &addrform, sizeof(addrform)) == -1)
    perror("setsockopt IPV6_ADDRFORM");
```

An application may use `IPV6_ADDRFORM` with `getsockopt()` to learn whether an open socket is a `PF_INET` or `PF_INET6` socket. For example:

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

if (getsockopt(s, IPPROTO_IPV6, IPV6_ADDRFORM,
              (char *) &addrform, &len) == -1)
    perror("getsockopt IPV6_ADDRFORM");
else if (addrform == PF_INET)
    printf("This is an IPv4 socket.\n");
else if (addrform == PF_INET6)
    printf("This is an IPv6 socket.\n");
else
    printf("This system is broken.\n");
```

## 5.2. 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);
```

### 5.3. 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:

#### 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

Argument type: int

### IPV6\_MULTICAST\_LOOP

Controls whether outgoing multicast packets sent should be delivered back to the local application. A toggle. If the option is set to 1, multicast packets are looped back. If it is set to 0, they are not.

Argument type: unsigned int

The reception of multicast packets is controlled by the two `setsockopt()` options summarized below:

### IPV6\_ADD\_MEMBERSHIP

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\_DROP\_MEMBERSHIP

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:

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

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 (hostname-to-address translation) and reverse lookup (address-to-hostname translation) need to be supported. Functions are also needed to convert IPv6 addresses between their binary and textual form.

### 6.1. Hostname-to-Address Translation

The commonly used function `gethostbyname()` remains unchanged as does the `hostent` structure to which it returns a pointer. Existing applications that call this function continue to receive only IPv4 addresses that are the result of a query in the DNS for A records. (We assume the DNS is being used; some environments may be using a hosts file or some other name resolution system, either of which may impede renumbering. We also assume that the `RES_USE_INET6` resolver option is not set, which we describe in more detail shortly.)

Two new changes are made to support IPv6 addresses. First, the following function is new:

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

struct hostent *gethostbyname2(const char *name, int af);
```

The `af` argument specifies the address family. The default operation of this function is simple:

- If the `af` argument is `AF_INET`, then a query is made for A records. If successful, IPv4 addresses are returned and the `h_length` member of the `hostent` structure will be 4, else the function returns a NULL pointer.
- If the `af` argument is `AF_INET6`, then a query is made for AAAA records. If successful, IPv6 addresses are returned and the `h_length` member of the `hostent` structure will be 16, else the function returns a NULL pointer.

The second change, that provides additional functionality, is a new resolver option `RES_USE_INET6`, which is defined as a result of including the `<resolv.h>` header. (This option is provided starting with the BIND 4.9.4 release.) There are three ways to set this option.

- The first way is

```
res_init();
_res.options |= RES_USE_INET6;
```

and then call either `gethostbyname()` or `gethostbyname2()`. This option then affects only the process that is calling the resolver.

- The second way to set this option is to set the environment variable `RES_OPTIONS`, as in `RES_OPTIONS=inet6`. (This example is for the Bourne and Korn shells.) This method affects any processes that see this environment variable.
- The third way is to set this option in the resolver configuration file (normally `/etc/resolv.conf`) and the option then affects all applications on the host. This final method should not be done until all applications on the host are capable of dealing with IPv6 addresses.

There is no priority among these three methods. When the `RES_USE_INET6` option is set, two changes occur:

- `gethostbyname(host)` first calls `gethostbyname2(host, AF_INET6)` looking for AAAA records, and if this fails it then calls `gethostbyname2(host, AF_INET)` looking for A records.
- `gethostbyname2(host, AF_INET)` always returns IPv4-mapped IPv6 addresses with the `h_length` member of the `hostent` structure set to 16.

An application must not enable the `RES_USE_INET6` option until it is prepared to deal with 16-byte addresses in the returned `hostent` structure.

The following table summarizes the operation of the existing `gethostbyname()` function, the new function `gethostbyname2()`, along with the new resolver option `RES_USE_INET6`.

	RES_USE_INET6 option	
	off	on
<code>gethostbyname</code> (host)	Search for A records. If found, return IPv4 addresses (h_length=4). Else error.  Provides backward compatibility with all existing IPv4 appls.	Search for AAAA records. If found, return IPv6 addresses (h_length=16). Else search for A records. If found, return IPv4-mapped IPv6 addresses (h_length=16). Else error.
<code>gethostbyname2</code> (host, AF_INET)	Search for A records. If found, return IPv4 addresses (h_length=4). Else error.	Search for A records. If found, return IPv4-mapped IPv6 addresses (h_length=16). Else error.
<code>gethostbyname2</code> (host, AF_INET6)	Search for AAAA records. If found, return IPv6 addresses (h_length=16). Else error.	Search for AAAA records. If found, return IPv6 addresses (h_length=16). Else error.

It is expected that when a typical naive application that calls `gethostbyname()` today is modified to use IPv6, it simply changes the program to use IPv6 sockets and then enables the `RES_USE_INET6` resolver option before calling `gethostbyname()`. This application will then work with either IPv4 or IPv6 peers.

Note that `gethostbyname()` and `gethostbyname2()` are not thread-safe, since both return a pointer to a static hostent structure. But several vendors have defined a thread-safe `gethostbyname_r()` function that requires four additional arguments. We expect these vendors to also define a `gethostbyname2_r()` function.

## 6.2. Address To Hostname Translation

The existing `gethostbyaddr()` function already requires an address family argument and can therefore work with IPv6 addresses:

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

```
struct hostent *gethostbyaddr(const char *src, int len, int af);
```

One possible source of confusion is the handling of IPv4-mapped IPv6 addresses and IPv4-compatible IPv6 addresses. This is addressed in [6] and involves the following logic:

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`, then query for a PTR record in the `in-addr.arpa` domain.
3. If `af` is `AF_INET6`, then query for a PTR record in the `ip6.int` domain.
4. If the function is returning success, and if `af` equals `AF_INET`, and if the `RES_USE_INET6` option was set, then the single address that is returned in the hostent structure (a copy of the first argument to the function) is returned as an IPv4-mapped IPv6 address and the `h_length` member is set to 16.

All four steps listed are performed, in order. The same caveats regarding a thread-safe version of `gethostbyname()` that were made at the end of the previous section apply here as well.

## 6.3. Protocol-Independent Hostname and Service Name Translation

Hostname-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) work in progress specification [4].

The official specification for this function will be the final POSIX standard. We are providing this independent description of the function because POSIX standards are not freely available (as are IETF documents). Should there be any discrepancies between this description and the POSIX description, the POSIX description takes precedence.

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

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

The `addrinfo` structure is defined as:

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

struct addrinfo {
    int      ai_flags;      /* AI_PASSIVE, AI_CANONNAME */
    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 hostname */
    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 hostname 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 hostname
<code>EAI_NONAME</code>	hostname nor <code>servname</code> provided, or not known
<code>EAI_SERVICE</code>	<code>servname</code> 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 `hostname` 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 `hostname` and `servname` are specified. In the normal server scenario, only the `servname` is specified. A non-`NULL` `hostname` string can be either a host 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 `hostname` 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 `hostname` 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 `hostname`.

All of the information returned by `getaddrinfo()` is dynamically allocated: the `addrinfo` structures, and the socket address structures and canonical host 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.

#### 6.4. Socket Address Structure to Hostname and Service Name

The POSIX 1003.1g specification includes no function to perform the reverse conversion from `getaddrinfo()`: to look up a hostname 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, size_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 hostname 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 hostname 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 hostname 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 `gethostbyaddr()`). 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 (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()`.

## 6.5. 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 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.

## 6.6. 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.

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

<netdb.h>       AI_CANONNAME
<netdb.h>       AI_PASSIVE
<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_ADDRFORM
<netinet/in.h>  IPV6_ADD_MEMBERSHIP
<netinet/in.h>  IPV6_DROP_MEMBERSHIP
<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{};

<resolv.h>      RES_USE_INET6

<sys/socket.h>  AF_INET6
<sys/socket.h>  PF_INET6

```

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_nameindex(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 *, size_t,
                                char *, size_t, char *, size_t, int);
<netdb.h>       void freeaddrinfo(struct addrinfo *);
<netdb.h>       char *gai_strerror(int);
<netdb.h>       struct hostent *gethostbyname(const char *);
<netdb.h>       struct hostent *gethostbyaddr(const char *, int, int);
<netdb.h>       struct hostent *gethostbyname2(const char *, int);

<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. A companion memo detailing the extensions to the socket interfaces to support IPv6 security is being written [3].

## 9. Acknowledgments

Thanks to the many people who made suggestions and provided feedback to the numerous revisions of 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, Tim Hartrick, Tom Herbert, Bob Hinden, Wan-Yen Hsu, Christian Huitema, Koji Imada, Markus Jork, Ron Lee, Alan Lloyd, Charles Lynn, Jack McCann, Dan McDonald, Dave Mitton, Thomas Narten, Erik Nordmark, 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 Kazuhiko Yamamoto,

The `getaddrinfo()` and `getnameinfo()` functions are taken from an earlier Work in Progress by Keith Sklower. As noted in that document, 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 made many contributions to this document. Ramesh Govindan made a number of contributions and co-authored an earlier version of this memo.

## 10. References

- [1] Deering, S., and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 1883, December 1995.
- [2] Hinden, R., and S. Deering, "IP Version 6 Addressing Architecture", RFC 1884, December 1995.
- [3] McDonald, D., "A Simple IP Security API Extension to BSD Sockets", Work in Progress.

- [4] IEEE, "Protocol Independent Interfaces", IEEE Std 1003.1g, DRAFT 6.3, November 1995.
- [5] Stevens, W., and M. Thomas, "Advanced Sockets API for IPv6", Work in Progress.
- [6] Vixie, P., "Reverse Name Lookups of Encapsulated IPv4 Addresses in IPv6", Work in Progress.

## 11. Authors' Addresses

Robert E. Gilligan  
Freigate Corporation  
710 Lakeway Dr. STE 230  
Sunnyvale, CA 94086

Phone: +1 408 524 4804  
EMail: gilligan@freigate.net

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  
Digital Equipment Corporation  
110 Spitbrook Road ZK3-3/U14  
Nashua, NH 03062-2698

Phone: +1 603 881 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

