Network Working Group
Request for Comments: 3493
Obsoletes: 2553
Category: Informational

R. Gilligan
Intransa, Inc.
S. Thomson
Cisco
J. Bound
J. McCann
Hewlett-Packard
W. Stevens

February 2003

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

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.

Table of Contents

1. Introduction....3
2. Design Considerations....4
2.1 What Needs to be Changed....4
2.2 Data Types....6
2.3 Headers....6
2.4 Structures....6
3. Socket Interface....6
3.1 IPv6 Address Family and Protocol Family....6
3.2 IPv6 Address Structure....7
3.3 Socket Address Structure for 4.3BSD-Based Systems....7
3.4 Socket Address Structure for 4.4BSD-Based Systems....9
3.5 The Socket Functions....9
3.6 Compatibility with IPv4 Applications....10
3.7 Compatibility with IPv4 Nodes....11
3.8 IPv6 Wildcard Address....11
3.9 IPv6 Loopback Address....13
3.10 Portability Additions....14
4. Interface Identification....16
4.1 Name-to-Index....17
4.2 Index-to-Name....17
4.3 Return All Interface Names and Indexes....18
4.4 Free Memory....18
5. Socket Options....18
5.1 Unicast Hop Limit....19
5.2 Sending and Receiving Multicast Packets....19
5.3 IPV6_V6ONLY option for AF_INET6 Sockets....22
6. Library Functions....22
6.1 Protocol-Independent Nodename and Service Name Translation....23
6.2 Socket Address Structure to Node Name and Service Name....28
6.3 Address Conversion Functions....31
6.4 Address Testing Macros....33
7. Summary of New Definitions....33
8. Security Considerations....35
9. Changes from RFC 2553....35
10. Acknowledgments....36
11. References....37
12. Authors' Addresses....38
13. Full Copyright Statement....39

1. Introduction

While IPv4 addresses are 32 bits long, IPv6 addresses are 128 bits long. 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, 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. It defines "basic" extensions that are of use to a broad range of applications. A companion document, the "advanced" API [4], covers extensions that are of use to more specialized applications, examples of which include routing daemons, and the "ping" and "traceroute" utilities.

The development of this API was started in 1994 in the IETF IPng working group. The API has evolved over the years, published first in RFC 2133, then again in RFC 2553, and reaching its final form in this document.

As the API matured and stabilized, it was incorporated into the Open Group's Networking Services (XNS) specification, issue 5.2, which was subsequently incorporated into a joint Open Group/IEEE/ISO standard [3].

Effort has been made to ensure that this document and [3] contain the same information with regard to the API definitions. However, the reader should note that this document is for informational purposes only, and that the official standard specification of the sockets API is [3].

It is expected that any future standardization work on this API would be done by the Open Group Base Working Group [6].

It should also be noted that this document describes only those portions of the API needed for IPv4 and IPv6 communications. Other potential uses of the API, for example the use of getaddrinfo() and getnameinfo() with the AF_UNIX address family, are beyond the scope of this document.

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 [3]. (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 IPv4 name-to-address translation functions in the socket interface are gethostbyname() and gethostbyaddr(). These are left as is, and new functions are defined which support both IPv4 and IPv6.

The IPv4 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. A new interface is needed to support the IPv6 hop limit header field. 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. Some of these extensions are described in [4].

2.2 Data Types

The data types of the structure elements given in this memo are intended to track the relevant standards. uintN_t means an unsigned integer of exactly N bits (e.g., uint16_t). The sa_family_t and in_port_t types are defined in [3].

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 [3]. (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 AF_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 flow information */
    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 intended to contain flow-related information. The exact way this field is mapped to or from a packet is not currently specified. Until such time as its use is specified, applications should set this field to zero when constructing a sockaddr_in6, and ignore this field in a sockaddr_in6 structure constructed by the system.

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 [2] of the address carried in the sin6_addr field. The mapping of sin6_scope_id to an interface or set of interfaces is left to implementation and future specifications on the subject of scoped addresses.

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(AF_INET, SOCK_STREAM, 0);
```

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

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

Applications may create IPv6/TCP and IPv6/UDP sockets (which may also handle IPv4 communication as described in section 3.7) by simply using the constant AF_INET6 instead of AF_INET in the first argument. For example, to create an IPv6/TCP socket, applications make the call:

```
s = socket(AF_INET6, SOCK_STREAM, 0);
```

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

```
s = socket(AF_INET6, SOCK_DGRAM, 0);
```

Once the application has created a AF_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 AF_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 AF_INET sockets and the sockaddr_in address structure. Applications must be able to create IPv4/TCP and IPv4/UDP sockets using the AF_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 getaddrinfo() function, as described in Section 6.1.

Applications may use AF_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 AF_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 6.4, 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 that is portable across multiple address families and platforms. This data structure is designed with the following goals.

- Large enough to accommodate all supported protocol-specific address structures.

- Aligned at an appropriate boundary so that pointers to it can be cast as pointers to protocol specific address structures and used to access the fields of those structures without alignment problems.

The sockaddr_storage structure contains field ss_family which is of type sa_family_t. When a sockaddr_storage structure is cast to a sockaddr structure, the ss_family field of the sockaddr_storage structure maps onto the sa_family field of the sockaddr structure. When a sockaddr_storage structure is cast as a protocol specific address structure, the ss_family field maps onto a field of that structure that is of type sa_family_t and that identifies the protocol's address family.

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 */
};
```

The above example implementation illustrates a data structure which will align on a 64-bit boundary. An implementation-specific field "__ss_align" along with "__ss_pad1" is used to force a 64-bit alignment which covers proper alignment good enough for the needs of sockaddr_in6 (IPv6), sockaddr_in (IPv4) address data structures. The size of padding field __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_ALIGNSIZE (with chosen value 8). Constants _SS_PAD1SIZE (derived value 6) and _SS_PAD2SIZE (derived value 112) are also for illustration and not required. The derived values assume sa_family_t is 2 bytes. 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.

On implementations where the 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 (uint8_t) + 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 */
};
```

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 5.2). 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 ifname is the name of an interface, the if_nametoindex() function shall return the interface index corresponding to name ifname; otherwise, it shall return zero. No errors are defined.

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);
```

When this function is called, the ifname argument shall point to a buffer of at least IF_NAMESIZE bytes. The function shall place in this buffer the name of the interface with index ifindex. (IF_NAMESIZE is also defined in <net/if.h> and its value includes a terminating null byte at the end of the interface name.) If ifindex is an interface index, then the function shall return the value supplied in ifname, which points to a buffer now containing the interface name. Otherwise, the function shall return a NULL pointer and set errno to indicate the error. If there is no interface corresponding to the specified index, errno is set to ENXIO. If there was a system error (such as running out of memory), 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.

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

   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 ptr argument shall be a pointer that was returned by if_nameindex(). After if_freenameindex() has been called, the application shall not use the array of which ptr is the address.

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;
socklen_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 multicast packets by simply specifying an IPv6 multicast address as the destination address, for example in the destination 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. If the interface index is specified as zero, the system selects the interface (for example, by looking up the address in a routing table and using the resulting interface).

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 use the resulting interface.

Argument type: struct ipv6_mreq

IPV6_LEAVE_GROUP

Leave a multicast group on a specified interface. If the interface index is specified as 0, the system may choose a multicast group membership to drop by matching the multicast address only.

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 to which datagrams will be sent. UDP applications must also 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.

5.3 IPV6_V6ONLY option for AF_INET6 Sockets

This socket option restricts AF_INET6 sockets to IPv6 communications only. As stated in section <3.7 Compatibility with IPv4 Nodes>, AF_INET6 sockets may be used for both IPv4 and IPv6 communications. Some applications may want to restrict their use of an AF_INET6 socket to IPv6 communications only. For these applications the IPV6_V6ONLY socket option is defined. When this option is turned on, the socket can be used to send and receive IPv6 packets only. This is an IPPROTO_IPV6 level option. This option takes an int value. This is a boolean option. By default this option is turned off.

Here is an example of setting this option:

```
int on = 1;

if (setsockopt(s, IPPROTO_IPV6, IPV6_V6ONLY,
               (char *)&on, sizeof(on)) == -1)
    perror("setsockopt IPV6_V6ONLY");
else
    printf("IPV6_V6ONLY set\n");
```

Note - This option has no effect on the use of IPv4 Mapped addresses which enter a node as a valid IPv6 addresses for IPv6 communications as defined by Stateless IP/ICMP Translation Algorithm (SIIT) [5].

An example use of this option is to allow two versions of the same server process to run on the same port, one providing service over IPv6, the other providing the same service over IPv4.

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.

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 gethostbyname2() function was deprecated in RFC 2553 and is no longer part of the basic API.

6.1 Protocol-Independent Nodename and Service Name Translation

Nodename-to-address translation is done in a protocol-independent fashion using the getaddrinfo() function.

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


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

void freeaddrinfo(struct addrinfo *ai);

struct addrinfo {
  int     ai_flags;     /* AI_PASSIVE, AI_CANONNAME,
                           AI_NUMERICHOST, .. */
  int     ai_family;    /* AF_xxx */
  int     ai_socktype;  /* SOCK_xxx */
  int     ai_protocol;  /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
  socklen_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 getaddrinfo() function translates the name of a service location (for example, a host name) and/or a service name and returns a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.

The nodename and servname arguments are either null pointers or pointers to null-terminated strings. One or both of these two arguments must be a non-null pointer.

The format of a valid name depends on the address family or families. If a specific family is not given and the name could be interpreted as valid within multiple supported families, the implementation will attempt to resolve the name in all supported families and, in absence of errors, one or more results shall be returned.

If the nodename argument is not null, it can be a descriptive name or can be an address string. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, valid descriptive names include host names. If the specified address family is AF_INET or AF_UNSPEC, address strings using Internet standard dot notation as specified in inet_addr() are valid. If the specified address family is AF_INET6 or AF_UNSPEC, standard IPv6 text forms described in inet_pton() are valid.

If nodename is not null, the requested service location is named by nodename; otherwise, the requested service location is local to the caller.

If servname is null, the call shall return network-level addresses for the specified nodename. If servname is not null, it is a null-terminated character string identifying the requested service. This can be either a descriptive name or a numeric representation suitable for use with the address family or families. If the specified address family is AF_INET, AF_INET6 or AF_UNSPEC, the service can be specified as a string specifying a decimal port number.

If the argument hints is not null, it refers to a structure containing input values that may direct the operation by providing options and by limiting the returned information to a specific socket type, address family and/or protocol. In this hints structure every member other than ai_flags, ai_family, ai_socktype and ai_protocol shall be set to zero or a null pointer. A value of AF_UNSPEC for ai_family means that the caller shall accept any address family. A value of zero for ai_socktype means that the caller shall accept any socket type. A value of zero for ai_protocol means that the caller shall accept any protocol. If hints is a null pointer, the behavior shall be as if it referred to a structure containing the value zero for the ai_flags, ai_socktype and ai_protocol fields, and AF_UNSPEC for the ai_family field.

Note:

1. If the caller handles only TCP and not UDP, for example, then the ai_protocol member of the hints structure should be set to IPPROTO_TCP when getaddrinfo() is called.

2. If the caller handles only IPv4 and not IPv6, then the ai_family member of the hints structure should be set to AF_INET when getaddrinfo() is called.

The ai_flags field to which hints parameter points shall be set to zero or be the bitwise-inclusive OR of one or more of the values AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST, AI_NUMERICSERV, AI_V4MAPPED, AI_ALL, and AI_ADDRCONFIG.

If the AI_PASSIVE flag is specified, the returned address information shall be suitable for use in binding a socket for accepting incoming connections for the specified service (i.e., a call to bind()). In this case, if the nodename argument is null, then the IP address portion of the socket address structure shall be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE flag is not specified, the returned address information shall be suitable for a call to connect() (for a connection-mode protocol) or for a call to connect(), sendto() or sendmsg() (for a connectionless protocol). In this case, if the nodename argument is null, then the IP address portion of the socket address structure shall be set to the loopback address. This flag is ignored if the nodename argument is not null.

If the AI_CANONNAME flag is specified and the nodename argument is not null, the function shall attempt to determine the canonical name corresponding to nodename (for example, if nodename is an alias or shorthand notation for a complete name).

If the AI_NUMERICHOST flag is specified, then a non-null nodename string supplied shall be a numeric host address string. Otherwise, an [EAI_NONAME] error is returned. This flag shall prevent any type of name resolution service (for example, the DNS) from being invoked.

If the AI_NUMERICSERV flag is specified, then a non-null servname string supplied shall be a numeric port string. Otherwise, an [EAI_NONAME] error shall be returned. This flag shall prevent any type of name resolution service (for example, NIS+) from being invoked.

If the AI_V4MAPPED flag is specified along with an ai_family of AF_INET6, then getaddrinfo() shall return IPv4-mapped IPv6 addresses on finding no matching IPv6 addresses (ai_addrlen shall be 16).

> For example, when using the DNS, if no AAAA records are found then a query is made for A records and any found are returned as IPv4-mapped IPv6 addresses.

The AI_V4MAPPED flag shall be ignored unless ai_family equals AF_INET6.

If the AI_ALL flag is used with the AI_V4MAPPED flag, then getaddrinfo() shall return all matching IPv6 and IPv4 addresses.

For example, when using the DNS, queries are made for both AAAA records and A records, and getaddrinfo() returns the combined results of both queries. Any IPv4 addresses found are returned as IPv4-mapped IPv6 addresses.

The AI_ALL flag without the AI_V4MAPPED flag is ignored.

Note:

When ai_family is not specified (AF_UNSPEC), AI_V4MAPPED and AI_ALL flags will only be used if AF_INET6 is supported.

If the AI_ADDRCONFIG flag is specified, IPv4 addresses shall be returned only if an IPv4 address is configured on the local system, and IPv6 addresses shall be returned only if an IPv6 address is configured on the local system. The loopback address is not considered for this case as valid as a configured address.

For example, when using the DNS, a query for AAAA records should occur only if the node has at least one IPv6 address configured (other than IPv6 loopback) and a query for A records should occur only if the node has at least one IPv4 address configured (other than the IPv4 loopback).

The ai_socktype field to which argument hints points specifies the socket type for the service, as defined for socket(). If a specific socket type is not given (for example, a value of zero) and the service name could be interpreted as valid with multiple supported socket types, the implementation shall attempt to resolve the service name for all supported socket types and, in the absence of errors, all possible results shall be returned. A non-zero socket type value shall limit the returned information to values with the specified socket type.

If the ai_family field to which hints points has the value AF_UNSPEC, addresses shall be returned for use with any address family that can be used with the specified nodename and/or servname. Otherwise, addresses shall be returned for use only with the specified address family. If ai_family is not AF_UNSPEC and ai_protocol is not zero, then addresses are returned for use only with the specified address family and protocol; the value of ai_protocol shall be interpreted as in a call to the socket() function with the corresponding values of ai_family and ai_protocol.

The freeaddrinfo() function frees one or more addrinfo structures returned by getaddrinfo(), along with any additional storage associated with those structures (for example, storage pointed to by the ai_canonname and ai_addr fields; an application must not

reference this storage after the associated addrinfo structure has been freed). If the ai_next field of the structure is not null, the entire list of structures is freed. The freeaddrinfo() function must support the freeing of arbitrary sublists of an addrinfo list originally returned by getaddrinfo().

Functions getaddrinfo() and freeaddrinfo() must be thread-safe.

A zero return value for getaddrinfo() indicates successful completion; a non-zero return value indicates failure. The possible values for the failures are listed below under Error Return Values.

Upon successful return of getaddrinfo(), the location to which res points shall refer to a linked list of addrinfo structures, each of which shall specify a socket address and information for use in creating a socket with which to use that socket address. The list shall include at least one addrinfo structure. The ai_next field of each structure contains a pointer to the next structure on the list, or a null pointer if it is the last structure on the list. Each structure on the list shall include values for use with a call to the socket() function, and a socket address for use with the connect() function or, if the AI_PASSIVE flag was specified, for use with the bind() function. The fields ai_family, ai_socktype, and ai_protocol shall be usable as the arguments to the socket() function to create a socket suitable for use with the returned address. The fields ai_addr and ai_addrlen are usable as the arguments to the connect() or bind() functions with such a socket, according to the AI_PASSIVE flag.

If nodename is not null, and if requested by the AI_CANONNAME flag, the ai_canonname field of the first returned addrinfo structure shall point to a null-terminated string containing the canonical name corresponding to the input nodename; if the canonical name is not available, then ai_canonname shall refer to the nodename argument or a string with the same contents. The contents of the ai_flags field of the returned structures are undefined.

All fields in socket address structures returned by getaddrinfo() that are not filled in through an explicit argument (for example, sin6_flowinfo) shall be set to zero.

Note: This makes it easier to compare socket address structures.

Error Return Values:

The getaddrinfo() function shall fail and return the corresponding value if:

[EAI_AGAIN] The name could not be resolved at this time. Future attempts may succeed.

[EAI_BADFLAGS] The flags parameter had an invalid value.

[EAI_FAIL] A non-recoverable error occurred when attempting to resolve the name.

[EAI_FAMILY] The address family was not recognized.

[EAI_MEMORY] There was a memory allocation failure when trying to allocate storage for the return value.

[EAI_NONAME] The name does not resolve for the supplied parameters. Neither nodename nor servname were supplied. At least one of these must be supplied.

[EAI_SERVICE] The service passed was not recognized for the specified socket type.

[EAI_SOCKTYPE] The intended socket type was not recognized.

[EAI_SYSTEM] A system error occurred; the error code can be found in errno.

The gai_strerror() function provides a descriptive text string corresponding to an EAI_xxx error value.

```
#include <netdb.h>

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

The argument is one of the EAI_xxx values defined for the getaddrinfo() and getnameinfo() functions. 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.

6.2 Socket Address Structure to Node Name and Service Name

The getnameinfo() function is used to translate the contents of a socket address structure to a node name and/or service name.

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

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                    char *node, socklen_t nodelen,
                    char *service, socklen_t servicelen,
                      int flags);
```

The getnameinfo() function shall translate a socket address to a node name and service location, all of which are defined as in getaddrinfo().

The sa argument points to a socket address structure to be translated.

The salen argument holds the size of the socket address structure pointed to by sa.

If the socket address structure contains an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address, the implementation shall extract the embedded IPv4 address and lookup the node name for that IPv4 address.

> Note: The IPv6 unspecified address ("::") and the IPv6 loopback address ("::1") are not IPv4-compatible addresses. If the address is the IPv6 unspecified address ("::"), a lookup is not performed, and the [EAI_NONAME] error is returned.

If the node argument is non-NULL and the nodelen argument is nonzero, then the node argument points to a buffer able to contain up to nodelen characters that receives the node name as a null-terminated string. If the node argument is NULL or the nodelen argument is zero, the node name shall not be returned. If the node's name cannot be located, the numeric form of the node's address is returned instead of its name.

If the service argument is non-NULL and the servicelen argument is non-zero, then the service argument points to a buffer able to contain up to servicelen bytes that receives the service name as a null-terminated string. If the service argument is NULL or the servicelen argument is zero, the service name shall not be returned. If the service's name cannot be located, the numeric form of the service address (for example, its port number) shall be returned instead of its name.

The arguments node and service cannot both be NULL.

The flags argument is a flag that changes the default actions of the function. By default the fully-qualified domain name (FQDN) for the host shall be returned, but:

- If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN shall be returned for local hosts.

- If the flag bit NI_NUMERICHOST is set, the numeric form of the host's address shall be returned instead of its name, under all circumstances.

- If the flag bit NI_NAMEREQD is set, an error shall be returned if the host's name cannot be located.

- If the flag bit NI_NUMERICSERV is set, the numeric form of the service address shall be returned (for example, its port number) instead of its name, under all circumstances.

- If the flag bit NI_DGRAM is set, this indicates that the service is a datagram service (SOCK_DGRAM). The default behavior shall assume that the service is a stream service (SOCK_STREAM).

Note:

1. The NI_NUMERICxxx flags are required to support the "-n" flags that many commands provide.

2. The NI_DGRAM flag is required for the few AF_INET and AF_INET6 port numbers (for example, [512,514]) that represent different services for UDP and TCP.

The getnameinfo() function shall be thread safe.

A zero return value for getnameinfo() indicates successful completion; a non-zero return value indicates failure.

Upon successful completion, getnameinfo() shall return the node and service names, if requested, in the buffers provided. The returned names are always null-terminated strings.

Error Return Values:

The getnameinfo() function shall fail and return the corresponding value if:

[EAI_AGAIN] The name could not be resolved at this time. Future attempts may succeed.

[EAI_BADFLAGS] The flags had an invalid value.

[EAI_FAIL] A non-recoverable error occurred.

[EAI_FAMILY] The address family was not recognized or the address length was invalid for the specified family.

[EAI_MEMORY] There was a memory allocation failure.

[EAI_NONAME] The name does not resolve for the supplied parameters. NI_NAMEREQD is set and the host's name cannot be located, or both nodename and servname were null.

[EAI_OVERFLOW] An argument buffer overflowed.

[EAI_SYSTEM] A system error occurred. The error code can be found in errno.

6.3 Address Conversion Functions

The two IPv4 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 <arpa/inet.h>

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

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

The inet_pton() function shall convert an address in its standard text presentation form into its numeric binary form. The af argument shall specify the family of the address. The AF_INET and AF_INET6 address families shall be 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; this shall be large enough to hold the numeric address (32 bits for AF_INET, 128 bits for AF_INET6). The inet_pton() function shall return 1 if the conversion

succeeds, with the address pointed to by dst in network byte order. It shall return 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.

If the af argument of inet_pton() is AF_INET, the src string shall be 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. The inet_pton() function does not accept other formats (such as the octal numbers, hexadecimal numbers, and fewer than four numbers that inet_addr() accepts).

If the af argument of inet_pton() is AF_INET6, the src string shall be in one of the standard IPv6 text forms defined in Section 2.2 of the addressing architecture specification [2].

The inet_ntop() function shall convert a numeric address into a text string suitable for presentation. The af argument shall specify 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 stores the resulting text string; it shall not be NULL. The size argument specifies the size of this buffer, which shall be large enough to hold the text string (INET_ADDRSTRLEN characters for IPv4, INET6_ADDRSTRLEN characters for IPv6).

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 shall return 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.

6.4 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 types of local-use IPv6 unicast addresses (Link-Local and Site-Local) defined in [2], and that by this definition, the IN6_IS_ADDR_LINKLOCAL macro returns false for the IPv6 loopback address (::1). 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_ALL
<netdb.h>       AI_CANONNAME
<netdb.h>       AI_NUMERICHOST
<netdb.h>       AI_NUMERICSERV
<netdb.h>       AI_PASSIVE
<netdb.h>       AI_V4MAPPED
```

```
<netdb.h>       EAI_AGAIN
<netdb.h>       EAI_BADFLAGS
<netdb.h>       EAI_FAIL
<netdb.h>       EAI_FAMILY
<netdb.h>       EAI_MEMORY
<netdb.h>       EAI_NONAME
<netdb.h>       EAI_OVERFLOW
<netdb.h>       EAI_SERVICE
<netdb.h>       EAI_SOCKTYPE
<netdb.h>       EAI_SYSTEM
<netdb.h>       NI_DGRAM
<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>  IPV6_V6ONLY
<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 *, socklen_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 *, socklen_t, char *, socklen_t, int);
<netdb.h>       void freeaddrinfo(struct addrinfo *);
<netdb.h>       const char *gai_strerror(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. Companion memos detailing the extensions to the socket interfaces to support IPv6 security are being written.

9. Changes from RFC 2553

1. Add brief description of the history of this API and its relation to the Open Group/IEEE/ISO standards.

2. Alignments with [3].

3. Removed all references to getipnodebyname() and getipnodebyaddr(), which are deprecated in favor of getaddrinfo() and getnameinfo().

4. Added IPV6_V6ONLY IP level socket option to permit nodes to not process IPv4 packets as IPv4 Mapped addresses in implementations.

5. Added SIIT to references and added new contributors.

6. In previous versions of this specification, the sin6_flowinfo field was associated with the IPv6 traffic class and flow label, but its usage was not completely specified. The complete definition of the sin6_flowinfo field, including its association with the traffic class or flow label, is now deferred to a future specification.

10. Acknowledgments

This specification's evolution and completeness were significantly influenced by the efforts of Richard Stevens, who has passed on. Richard's wisdom and talent made the specification what it is today. The co-authors will long think of Richard with great respect.

Thanks to the many people who made suggestions and provided feedback to this document, including:

Werner Almesberger, Ran Atkinson, Fred Baker, Dave Borman, Andrew Cherenson, Alex Conta, Alan Cox, Steve Deering, Richard Draves, Francis Dupont, Robert Elz, Brian Haberman, Jun-ichiro itojun Hagino, 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, Finnbarr Murphy, Thomas Narten, Josh Osborne, Craig Partridge, Jean-Luc Richier, Bill Sommerfield, Erik Scoredos, Keith Sklower, JINMEI Tatuya, Dave Thaler, Matt Thomas, Harvey Thompson, Dean D. Throop, Karen Tracey, Glenn Trewitt, Paul Vixie, David Waitzman, Carl Williams, Kazu Yamamoto, Vlad Yasevich, Stig Venaas, and Brian Zill.

The getaddrinfo() and getnameinfo() functions are taken from an earlier document 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, 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.

11. 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 Std. 1003.1-2001 Standard for Information Technology -- Portable Operating System Interface (POSIX). Open Group Technical Standard: Base Specifications, Issue 6, December 2001. ISO/IEC 9945:2002. http://www.opengroup.org/austin

[4] Stevens, W. and M. Thomas, "Advanced Sockets API for IPv6", RFC 2292, February 1998.

[5] Nordmark, E., "Stateless IP/ICMP Translation Algorithm (SIIT)", RFC 2765, February 2000.

[6] The Open Group Base Working Group http://www.opengroup.org/platform/base.html

12. Authors' Addresses

Bob Gilligan
Intransa, Inc.
2870 Zanker Rd.
San Jose, CA 95134

Phone: 408-678-8647
EMail: gilligan@intransa.com

Susan Thomson
Cisco Systems
499 Thornall Street, 8th floor
Edison, NJ 08837

Phone: 732-635-3086
EMail: sethomso@cisco.com

Jim Bound
Hewlett-Packard Company
110 Spitbrook Road ZKO3-3/W20
Nashua, NH 03062

Phone: 603-884-0062
EMail: Jim.Bound@hp.com

Jack McCann
Hewlett-Packard Company
110 Spitbrook Road ZKO3-3/W20
Nashua, NH 03062

Phone: 603-884-2608
EMail: Jack.McCann@hp.com

13. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.