Network Working Group
Request for Comments: 2292
Category: Informational

W. Stevens
Consultant
M. Thomas
AltaVista

February 1998

Advanced Sockets API 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

Specifications are in progress for changes to the sockets API to support IP version 6 [RFC-2133]. These changes are for TCP and UDP-based applications and will support most end-user applications in use today: Telnet and FTP clients and servers, HTTP clients and servers, and the like.

But another class of applications exists that will also be run under IPv6. We call these "advanced" applications and today this includes programs such as Ping, Traceroute, routing daemons, multicast routing daemons, router discovery daemons, and the like. The API feature typically used by these programs that make them "advanced" is a raw socket to access ICMPv4, IGMPv4, or IPv4, along with some knowledge of the packet header formats used by these protocols. To provide portability for applications that use raw sockets under IPv6, some standardization is needed for the advanced API features.

There are other features of IPv6 that some applications will need to access: interface identification (specifying the outgoing interface and determining the incoming interface) and IPv6 extension headers that are not addressed in [RFC-2133]: Hop-by-Hop options, Destination options, and the Routing header (source routing). This document provides API access to these features too.

Table of Contents

1. Introduction

Specifications are in progress for changes to the sockets API to support IP version 6 [RFC-2133]. These changes are for TCP and UDP-based applications. The current document defines some the "advanced" features of the sockets API that are required for applications to take advantage of additional features of IPv6.

Today, the portability of applications using IPv4 raw sockets is quite high, but this is mainly because most IPv4 implementations started from a common base (the Berkeley source code) or at least started with the Berkeley headers. This allows programs such as Ping and Traceroute, for example, to compile with minimal effort on many hosts that support the sockets API. With IPv6, however, there is no common source code base that implementors are starting from, and the possibility for divergence at this level between different implementations is high. To avoid a complete lack of portability amongst applications that use raw IPv6 sockets, some standardization is necessary.

There are also features from the basic IPv6 specification that are not addressed in [RFC-2133]: sending and receiving Hop-by-Hop options, Destination options, and Routing headers, specifying the outgoing interface, and being told of the receiving interface.

This document can be divided into the following main sections.

1. Definitions of the basic constants and structures required for applications to use raw IPv6 sockets. This includes structure definitions for the IPv6 and ICMPv6 headers and all associated constants (e.g., values for the Next Header field).

2. Some basic semantic definitions for IPv6 raw sockets. For example, a raw ICMPv4 socket requires the application to calculate and store the ICMPv4 header checksum. But with IPv6 this would require the application to choose the source IPv6 address because the source address is part of the pseudo header that ICMPv6 now uses for its checksum computation. It should be defined that with a raw ICMPv6 socket the kernel always calculates and stores the ICMPv6 header checksum.

3. Packet information: how applications can obtain the received interface, destination address, and received hop limit, along with specifying these values on a per-packet basis. There are a class of applications that need this capability and the technique should be portable.

4. Access to the optional Hop-by-Hop, Destination, and Routing headers.

5. Additional features required for IPv6 application portability.

The packet information along with access to the extension headers (Hop-by-Hop options, Destination options, and Routing header) are specified using the "ancillary data" fields that were added to the 4.3BSD Reno sockets API in 1990. The reason is that these ancillary data fields are part of the Posix.1g standard (which should be approved in 1997) and should therefore be adopted by most vendors.

This document does not address application access to either the authentication header or the encapsulating security payload header.

All examples in this document omit error checking in favor of brevity and clarity.

We note that many of the functions and socket options defined in this document may have error returns that are not defined in this document. Many of these possible error returns will be recognized only as implementations proceed.

Datatypes in this document follow the Posix.1g format: intN_t means a signed integer of exactly N bits (e.g., int16_t) and uintN_t means an unsigned integer of exactly N bits (e.g., uint32_t).

Note that we use the (unofficial) terminology ICMPv4, IGMPv4, and ARPv4 to avoid any confusion with the newer ICMPv6 protocol.

2. Common Structures and Definitions

Many advanced applications examine fields in the IPv6 header and set and examine fields in the various ICMPv6 headers. Common structure definitions for these headers are required, along with common constant definitions for the structure members.

Two new headers are defined: <netinet/ip6.h> and <netinet/icmp6.h>.

When an include file is specified, that include file is allowed to include other files that do the actual declaration or definition.

2.1. The ip6_hdr Structure

The following structure is defined as a result of including <netinet/ip6.h>. Note that this is a new header.

```
struct ip6_hdr {
  union {
    struct ip6_hdrctl {
      uint32_t ip6_un1_flow;   /* 24 bits of flow-ID */
      uint16_t ip6_un1_plen;   /* payload length */
      uint8_t  ip6_un1_nxt;    /* next header */
      uint8_t  ip6_un1_hlim;   /* hop limit */
    } ip6_un1;
    uint8_t ip6_un2_vfc;       /* 4 bits version, 4 bits priority */
  } ip6_ctlun;
  struct in6_addr ip6_src;      /* source address */
  struct in6_addr ip6_dst;      /* destination address */
};

#define ip6_vfc   ip6_ctlun.ip6_un2_vfc
#define ip6_flow  ip6_ctlun.ip6_un1.ip6_un1_flow
#define ip6_plen  ip6_ctlun.ip6_un1.ip6_un1_plen
#define ip6_nxt   ip6_ctlun.ip6_un1.ip6_un1_nxt
#define ip6_hlim  ip6_ctlun.ip6_un1.ip6_un1_hlim
```

```
 #define ip6_hops  ip6_ctlun.ip6_un1.ip6_un1_hlim
```

2.1.1. IPv6 Next Header Values

IPv6 defines many new values for the Next Header field. The following constants are defined as a result of including <netinet/in.h>.

```
#define IPPROTO_HOPOPTS        0 /* IPv6 Hop-by-Hop options */
#define IPPROTO_IPV6          41 /* IPv6 header */
#define IPPROTO_ROUTING       43 /* IPv6 Routing header */
#define IPPROTO_FRAGMENT      44 /* IPv6 fragmentation header */
#define IPPROTO_ESP           50 /* encapsulating security payload */
#define IPPROTO_AH            51 /* authentication header */
#define IPPROTO_ICMPV6        58 /* ICMPv6 */
#define IPPROTO_NONE          59 /* IPv6 no next header */
#define IPPROTO_DSTOPTS       60 /* IPv6 Destination options */
```

Berkeley-derived IPv4 implementations also define IPPROTO_IP to be 0. This should not be a problem since IPPROTO_IP is used only with IPv4 sockets and IPPROTO_HOPOPTS only with IPv6 sockets.

2.1.2. IPv6 Extension Headers

Six extension headers are defined for IPv6. We define structures for all except the Authentication header and Encapsulating Security Payload header, both of which are beyond the scope of this document. The following structures are defined as a result of including <netinet/ip6.h>.

```
/* Hop-by-Hop options header */
/* XXX should we pad it to force alignment on an 8-byte boundary? */
struct ip6_hbh {
  uint8_t  ip6h_nxt;        /* next header */
  uint8_t  ip6h_len;        /* length in units of 8 octets */
    /* followed by options */
};

/* Destination options header */
/* XXX should we pad it to force alignment on an 8-byte boundary? */
struct ip6_dest {
  uint8_t  ip6d_nxt;        /* next header */
  uint8_t  ip6d_len;        /* length in units of 8 octets */
    /* followed by options */
};

/* Routing header */
struct ip6_rthdr {
```

```
     uint8_t  ip6r_nxt;        /* next header */
     uint8_t  ip6r_len;        /* length in units of 8 octets */
     uint8_t  ip6r_type;       /* routing type */
     uint8_t  ip6r_segleft;    /* segments left */
       /* followed by routing type specific data */
   };

   /* Type 0 Routing header */
   struct ip6_rthdr0 {
     uint8_t  ip6r0_nxt;       /* next header */
     uint8_t  ip6r0_len;       /* length in units of 8 octets */
     uint8_t  ip6r0_type;      /* always zero */
     uint8_t  ip6r0_segleft;   /* segments left */
     uint8_t  ip6r0_reserved;  /* reserved field */
     uint8_t  ip6r0_slmap[3];  /* strict/loose bit map */
     struct in6_addr  ip6r0_addr[1];  /* up to 23 addresses */
   };

   /* Fragment header */
   struct ip6_frag {
     uint8_t   ip6f_nxt;       /* next header */
     uint8_t   ip6f_reserved;  /* reserved field */
     uint16_t  ip6f_offlg;     /* offset, reserved, and flag */
     uint32_t  ip6f_ident;     /* identification */
   };

   #if     BYTE_ORDER == BIG_ENDIAN
   #define IP6F_OFF_MASK       0xfff8  /* mask out offset from _offlg */
   #define IP6F_RESERVED_MASK  0x0006  /* reserved bits in ip6f_offlg */
   #define IP6F_MORE_FRAG      0x0001  /* more-fragments flag */
   #else   /* BYTE_ORDER == LITTLE_ENDIAN */
   #define IP6F_OFF_MASK       0xf8ff  /* mask out offset from _offlg */
   #define IP6F_RESERVED_MASK  0x0600  /* reserved bits in ip6f_offlg */
   #define IP6F_MORE_FRAG      0x0100  /* more-fragments flag */
   #endif
```

Defined constants for fields larger than 1 byte depend on the byte ordering that is used. This API assumes that the fields in the protocol headers are left in the network byte order, which is big-endian for the Internet protocols. If not, then either these constants or the fields being tested must be converted at run-time, using something like htons() or htonl().

(Note: We show an implementation that supports both big-endian and little-endian byte ordering, assuming a hypothetical compile-time #if test to determine the byte ordering. The constant that we show, BYTE_ORDER, with values of BIG_ENDIAN and LITTLE_ENDIAN, are for example purposes only. If an implementation runs on only one type of hardware it need only define the set of constants for that hardware's byte ordering.)

2.2. The icmp6_hdr Structure

The ICMPv6 header is needed by numerous IPv6 applications including Ping, Traceroute, router discovery daemons, and neighbor discovery daemons. The following structure is defined as a result of including <netinet/icmp6.h>. Note that this is a new header.

```
struct icmp6_hdr {
  uint8_t     icmp6_type;   /* type field */
  uint8_t     icmp6_code;   /* code field */
  uint16_t    icmp6_cksum;  /* checksum field */
  union {
    uint32_t  icmp6_un_data32[1]; /* type-specific field */
    uint16_t  icmp6_un_data16[2]; /* type-specific field */
    uint8_t   icmp6_un_data8[4];  /* type-specific field */
  } icmp6_dataun;
};

#define icmp6_data32    icmp6_dataun.icmp6_un_data32
#define icmp6_data16    icmp6_dataun.icmp6_un_data16
#define icmp6_data8     icmp6_dataun.icmp6_un_data8
#define icmp6_pptr      icmp6_data32[0]  /* parameter prob */
#define icmp6_mtu       icmp6_data32[0]  /* packet too big */
#define icmp6_id        icmp6_data16[0]  /* echo request/reply */
#define icmp6_seq       icmp6_data16[1]  /* echo request/reply */
#define icmp6_maxdelay  icmp6_data16[0]  /* mcast group membership */
```

2.2.1. ICMPv6 Type and Code Values

In addition to a common structure for the ICMPv6 header, common definitions are required for the ICMPv6 type and code fields. The following constants are also defined as a result of including <netinet/icmp6.h>.

```
#define ICMP6_DST_UNREACH             1
#define ICMP6_PACKET_TOO_BIG          2
#define ICMP6_TIME_EXCEEDED           3
#define ICMP6_PARAM_PROB              4

#define ICMP6_INFOMSG_MASK  0x80    /* all informational messages */

#define ICMP6_ECHO_REQUEST          128
#define ICMP6_ECHO_REPLY            129
```

```
#define ICMP6_MEMBERSHIP_QUERY      130
#define ICMP6_MEMBERSHIP_REPORT     131
#define ICMP6_MEMBERSHIP_REDUCTION  132

#define ICMP6_DST_UNREACH_NOROUTE     0 /* no route to destination */
#define ICMP6_DST_UNREACH_ADMIN       1 /* communication with */
                                        /* destination */
                                        /* administratively */
                                        /* prohibited */
#define ICMP6_DST_UNREACH_NOTNEIGHBOR 2 /* not a neighbor */
#define ICMP6_DST_UNREACH_ADDR        3 /* address unreachable */
#define ICMP6_DST_UNREACH_NOPORT      4 /* bad port */

#define ICMP6_TIME_EXCEED_TRANSIT     0 /* Hop Limit == 0 in transit */
#define ICMP6_TIME_EXCEED_REASSEMBLY  1 /* Reassembly time out */

#define ICMP6_PARAMPROB_HEADER        0 /* erroneous header field */
#define ICMP6_PARAMPROB_NEXTHEADER    1 /* unrecognized Next Header */
#define ICMP6_PARAMPROB_OPTION        2 /* unrecognized IPv6 option */
```

The five ICMP message types defined by IPv6 neighbor discovery (133-137) are defined in the next section.

2.2.2. ICMPv6 Neighbor Discovery Type and Code Values

The following structures and definitions are defined as a result of including <netinet/icmp6.h>.

```
#define ND_ROUTER_SOLICIT           133
#define ND_ROUTER_ADVERT            134
#define ND_NEIGHBOR_SOLICIT         135
#define ND_NEIGHBOR_ADVERT          136
#define ND_REDIRECT                 137

struct nd_router_solicit {     /* router solicitation */
  struct icmp6_hdr  nd_rs_hdr;
    /* could be followed by options */
};

#define nd_rs_type               nd_rs_hdr.icmp6_type
#define nd_rs_code               nd_rs_hdr.icmp6_code
#define nd_rs_cksum              nd_rs_hdr.icmp6_cksum
#define nd_rs_reserved           nd_rs_hdr.icmp6_data32[0]

struct nd_router_advert {     /* router advertisement */
  struct icmp6_hdr  nd_ra_hdr;
  uint32_t   nd_ra_reachable;   /* reachable time */
  uint32_t   nd_ra_retransmit;  /* retransmit timer */
```

```
        /* could be followed by options */
    };

    #define nd_ra_type               nd_ra_hdr.icmp6_type
    #define nd_ra_code               nd_ra_hdr.icmp6_code
    #define nd_ra_cksum              nd_ra_hdr.icmp6_cksum
    #define nd_ra_curhoplimit        nd_ra_hdr.icmp6_data8[0]
    #define nd_ra_flags_reserved     nd_ra_hdr.icmp6_data8[1]
    #define ND_RA_FLAG_MANAGED       0x80
    #define ND_RA_FLAG_OTHER         0x40
    #define nd_ra_router_lifetime    nd_ra_hdr.icmp6_data16[1]

    struct nd_neighbor_solicit {   /* neighbor solicitation */
      struct icmp6_hdr  nd_ns_hdr;
      struct in6_addr   nd_ns_target; /* target address */
        /* could be followed by options */
    };

    #define nd_ns_type               nd_ns_hdr.icmp6_type
    #define nd_ns_code               nd_ns_hdr.icmp6_code
    #define nd_ns_cksum              nd_ns_hdr.icmp6_cksum
    #define nd_ns_reserved           nd_ns_hdr.icmp6_data32[0]

    struct nd_neighbor_advert {    /* neighbor advertisement */
      struct icmp6_hdr  nd_na_hdr;
      struct in6_addr   nd_na_target; /* target address */
        /* could be followed by options */
    };

    #define nd_na_type               nd_na_hdr.icmp6_type
    #define nd_na_code               nd_na_hdr.icmp6_code
    #define nd_na_cksum              nd_na_hdr.icmp6_cksum
    #define nd_na_flags_reserved     nd_na_hdr.icmp6_data32[0]
    #if     BYTE_ORDER == BIG_ENDIAN
    #define ND_NA_FLAG_ROUTER        0x80000000
    #define ND_NA_FLAG_SOLICITED     0x40000000
    #define ND_NA_FLAG_OVERRIDE      0x20000000
    #else   /* BYTE_ORDER == LITTLE_ENDIAN */
    #define ND_NA_FLAG_ROUTER        0x00000080
    #define ND_NA_FLAG_SOLICITED     0x00000040
    #define ND_NA_FLAG_OVERRIDE      0x00000020
    #endif

    struct nd_redirect {           /* redirect */
      struct icmp6_hdr  nd_rd_hdr;
      struct in6_addr   nd_rd_target; /* target address */
      struct in6_addr   nd_rd_dst;    /* destination address */
        /* could be followed by options */
```

```
   };

   #define nd_rd_type               nd_rd_hdr.icmp6_type
   #define nd_rd_code               nd_rd_hdr.icmp6_code
   #define nd_rd_cksum              nd_rd_hdr.icmp6_cksum
   #define nd_rd_reserved           nd_rd_hdr.icmp6_data32[0]

   struct nd_opt_hdr {           /* Neighbor discovery option header */
     uint8_t  nd_opt_type;
     uint8_t  nd_opt_len;        /* in units of 8 octets */
       /* followed by option specific data */
   };

   #define  ND_OPT_SOURCE_LINKADDR       1
   #define  ND_OPT_TARGET_LINKADDR       2
   #define  ND_OPT_PREFIX_INFORMATION    3
   #define  ND_OPT_REDIRECTED_HEADER     4
   #define  ND_OPT_MTU                   5

   struct nd_opt_prefix_info {    /* prefix information */
     uint8_t   nd_opt_pi_type;
     uint8_t   nd_opt_pi_len;
     uint8_t   nd_opt_pi_prefix_len;
     uint8_t   nd_opt_pi_flags_reserved;
     uint32_t  nd_opt_pi_valid_time;
     uint32_t  nd_opt_pi_preferred_time;
     uint32_t  nd_opt_pi_reserved2;
     struct in6_addr  nd_opt_pi_prefix;
   };

   #define ND_OPT_PI_FLAG_ONLINK        0x80
   #define ND_OPT_PI_FLAG_AUTO          0x40

   struct nd_opt_rd_hdr {         /* redirected header */
     uint8_t   nd_opt_rh_type;
     uint8_t   nd_opt_rh_len;
     uint16_t  nd_opt_rh_reserved1;
     uint32_t  nd_opt_rh_reserved2;
       /* followed by IP header and data */
   };

   struct nd_opt_mtu {            /* MTU option */
     uint8_t   nd_opt_mtu_type;
     uint8_t   nd_opt_mtu_len;
     uint16_t  nd_opt_mtu_reserved;
     uint32_t  nd_opt_mtu_mtu;
   };
```

We note that the nd_na_flags_reserved flags have the same byte ordering problems as we discussed with ip6f_offlg.

2.3. Address Testing Macros

The basic API ([RFC-2133]) defines some macros for testing an IPv6 address for certain properties. This API extends those definitions with additional address testing macros, defined as a result of including <netinet/in.h>.

```
int  IN6_ARE_ADDR_EQUAL(const struct in6_addr *,
                        const struct in6_addr *);
```

2.4. Protocols File

Many hosts provide the file /etc/protocols that contains the names of the various IP protocols and their protocol number (e.g., the value of the protocol field in the IPv4 header for that protocol, such as 1 for ICMP). Some programs then call the function getprotobyname() to obtain the protocol value that is then specified as the third argument to the socket() function. For example, the Ping program contains code of the form

```
struct protoent  *proto;

proto = getprotobyname("icmp");

s = socket(AF_INET, SOCK_RAW, proto->p_proto);
```

Common names are required for the new IPv6 protocols in this file, to provide portability of applications that call the getprotoXXX() functions.

We define the following protocol names with the values shown. These are taken from ftp://ftp.isi.edu/in-notes/iana/assignments/protocol-numbers.

```
hopopt           0    # hop-by-hop options for ipv6
ipv6            41    # ipv6
ipv6-route      43    # routing header for ipv6
ipv6-frag       44    # fragment header for ipv6
esp             50    # encapsulating security payload for ipv6
ah              51    # authentication header for ipv6
ipv6-icmp       58    # icmp for ipv6
ipv6-nonxt      59    # no next header for ipv6
ipv6-opts       60    # destination options for ipv6
```

3. IPv6 Raw Sockets

Raw sockets bypass the transport layer (TCP or UDP). With IPv4, raw sockets are used to access ICMPv4, IGMPv4, and to read and write IPv4 datagrams containing a protocol field that the kernel does not process. An example of the latter is a routing daemon for OSPF, since it uses IPv4 protocol field 89. With IPv6 raw sockets will be used for ICMPv6 and to read and write IPv6 datagrams containing a Next Header field that the kernel does not process. Examples of the latter are a routing daemon for OSPF for IPv6 and RSVP (protocol field 46).

All data sent via raw sockets MUST be in network byte order and all data received via raw sockets will be in network byte order. This differs from the IPv4 raw sockets, which did not specify a byte ordering and typically used the host's byte order.

Another difference from IPv4 raw sockets is that complete packets (that is, IPv6 packets with extension headers) cannot be read or written using the IPv6 raw sockets API. Instead, ancillary data objects are used to transfer the extension headers, as described later in this document. Should an application need access to the complete IPv6 packet, some other technique, such as the datalink interfaces BPF or DLPI, must be used.

All fields in the IPv6 header that an application might want to change (i.e., everything other than the version number) can be modified using ancillary data and/or socket options by the application for output. All fields in a received IPv6 header (other than the version number and Next Header fields) and all extension headers are also made available to the application as ancillary data on input. Hence there is no need for a socket option similar to the IPv4 IP_HDRINCL socket option.

When writing to a raw socket the kernel will automatically fragment the packet if its size exceeds the path MTU, inserting the required fragmentation headers. On input the kernel reassembles received fragments, so the reader of a raw socket never sees any fragment headers.

When we say "an ICMPv6 raw socket" we mean a socket created by calling the socket function with the three arguments PF_INET6, SOCK_RAW, and IPPROTO_ICMPV6.

Most IPv4 implementations give special treatment to a raw socket created with a third argument to socket() of IPPROTO_RAW, whose value is normally 255. We note that this value has no special meaning to an IPv6 raw socket (and the IANA currently reserves the value of 255 when used as a next-header field). (Note: This feature was added to IPv4 in 1988 by Van Jacobson to support traceroute, allowing a complete IP header to be passed by the application, before the IP_HDRINCL socket option was added.)

3.1. Checksums

The kernel will calculate and insert the ICMPv6 checksum for ICMPv6 raw sockets, since this checksum is mandatory.

For other raw IPv6 sockets (that is, for raw IPv6 sockets created with a third argument other than IPPROTO_ICMPV6), the application must set the new IPV6_CHECKSUM socket option to have the kernel (1) compute and store a checksum for output, and (2) verify the received checksum on input, discarding the packet if the checksum is in error. This option prevents applications from having to perform source address selection on the packets they send. The checksum will incorporate the IPv6 pseudo-header, defined in Section 8.1 of [RFC-1883]. This new socket option also specifies an integer offset into the user data of where the checksum is located.

```
int  offset = 2;
setsockopt(fd, IPPROTO_IPV6, IPV6_CHECKSUM, &offset, sizeof(offset));
```

By default, this socket option is disabled. Setting the offset to -1 also disables the option. By disabled we mean (1) the kernel will not calculate and store a checksum for outgoing packets, and (2) the kernel will not verify a checksum for received packets.

(Note: Since the checksum is always calculated by the kernel for an ICMPv6 socket, applications are not able to generate ICMPv6 packets with incorrect checksums (presumably for testing purposes) using this API.)

3.2. ICMPv6 Type Filtering

ICMPv4 raw sockets receive most ICMPv4 messages received by the kernel. (We say "most" and not "all" because Berkeley-derived kernels never pass echo requests, timestamp requests, or address mask requests to a raw socket. Instead these three messages are processed entirely by the kernel.) But ICMPv6 is a superset of ICMPv4, also including the functionality of IGMPv4 and ARPv4. This means that an ICMPv6 raw socket can potentially receive many more messages than would be received with an ICMPv4 raw socket: ICMP messages similar to ICMPv4, along with neighbor solicitations, neighbor advertisements, and the three group membership messages.

Most applications using an ICMPv6 raw socket care about only a small subset of the ICMPv6 message types. To transfer extraneous ICMPv6 messages from the kernel to user can incur a significant overhead. Therefore this API includes a method of filtering ICMPv6 messages by the ICMPv6 type field.

Each ICMPv6 raw socket has an associated filter whose datatype is defined as

```
struct icmp6_filter;
```

This structure, along with the macros and constants defined later in this section, are defined as a result of including the <netinet/icmp6.h> header.

The current filter is fetched and stored using getsockopt() and setsockopt() with a level of IPPROTO_ICMPV6 and an option name of ICMP6_FILTER.

Six macros operate on an icmp6_filter structure:

```
void ICMP6_FILTER_SETPASSALL (struct icmp6_filter *);
void ICMP6_FILTER_SETBLOCKALL(struct icmp6_filter *);

void ICMP6_FILTER_SETPASS ( int, struct icmp6_filter *);
void ICMP6_FILTER_SETBLOCK( int, struct icmp6_filter *);

int  ICMP6_FILTER_WILLPASS (int, const struct icmp6_filter *);
int  ICMP6_FILTER_WILLBLOCK(int, const struct icmp6_filter *);
```

The first argument to the last four macros (an integer) is an ICMPv6 message type, between 0 and 255. The pointer argument to all six macros is a pointer to a filter that is modified by the first four macros examined by the last two macros.

The first two macros, SETPASSALL and SETBLOCKALL, let us specify that all ICMPv6 messages are passed to the application or that all ICMPv6 messages are blocked from being passed to the application.

The next two macros, SETPASS and SETBLOCK, let us specify that messages of a given ICMPv6 type should be passed to the application or not passed to the application (blocked).

The final two macros, WILLPASS and WILLBLOCK, return true or false depending whether the specified message type is passed to the application or blocked from being passed to the application by the filter pointed to by the second argument.

When an ICMPv6 raw socket is created, it will by default pass all ICMPv6 message types to the application.

As an example, a program that wants to receive only router advertisements could execute the following:

```
struct icmp6_filter  myfilt;

fd = socket(PF_INET6, SOCK_RAW, IPPROTO_ICMPV6);

ICMP6_FILTER_SETBLOCKALL(&myfilt);
ICMP6_FILTER_SETPASS(ND_ROUTER_ADVERT, &myfilt);
setsockopt(fd, IPPROTO_ICMPV6, ICMP6_FILTER, &myfilt, sizeof(myfilt));
```

The filter structure is declared and then initialized to block all messages types. The filter structure is then changed to allow router advertisement messages to be passed to the application and the filter is installed using setsockopt().

The icmp6_filter structure is similar to the fd_set datatype used with the select() function in the sockets API. The icmp6_filter structure is an opaque datatype and the application should not care how it is implemented. All the application does with this datatype is allocate a variable of this type, pass a pointer to a variable of this type to getsockopt() and setsockopt(), and operate on a variable of this type using the six macros that we just defined.

Nevertheless, it is worth showing a simple implementation of this datatype and the six macros.

```
struct icmp6_filter {
  uint32_t  icmp6_filt[8];  /* 8*32 = 256 bits */
};

#define ICMP6_FILTER_WILLPASS(type, filterp) \
    ((((filterp)->icmp6_filt[(type) >> 5]) & (1 << ((type) & 31))) != 0)
#define ICMP6_FILTER_WILLBLOCK(type, filterp) \
    ((((filterp)->icmp6_filt[(type) >> 5]) & (1 << ((type) & 31))) == 0)
#define ICMP6_FILTER_SETPASS(type, filterp) \
    ((((filterp)->icmp6_filt[(type) >> 5]) |=  (1 << ((type) & 31))))
#define ICMP6_FILTER_SETBLOCK(type, filterp) \
    ((((filterp)->icmp6_filt[(type) >> 5]) &= ~(1 << ((type) & 31))))
#define ICMP6_FILTER_SETPASSALL(filterp) \
    memset((filterp), 0xFF, sizeof(struct icmp6_filter))
#define ICMP6_FILTER_SETBLOCKALL(filterp) \
    memset((filterp), 0, sizeof(struct icmp6_filter))
```

(Note: These sample definitions have two limitations that an implementation may want to change. The first four macros evaluate their first argument two times. The second two macros require the inclusion of the <string.h> header for the memset() function.)

4. Ancillary Data

4.2BSD allowed file descriptors to be transferred between separate processes across a UNIX domain socket using the sendmsg() and recvmsg() functions. Two members of the msghdr structure, msg_accrights and msg_accrightslen, were used to send and receive the descriptors. When the OSI protocols were added to 4.3BSD Reno in 1990 the names of these two fields in the msghdr structure were changed to msg_control and msg_controllen, because they were used by the OSI protocols for "control information", although the comments in the source code call this "ancillary data".

Other than the OSI protocols, the use of ancillary data has been rare. In 4.4BSD, for example, the only use of ancillary data with IPv4 is to return the destination address of a received UDP datagram if the IP_RECVDSTADDR socket option is set. With Unix domain sockets ancillary data is still used to send and receive descriptors.

Nevertheless the ancillary data fields of the msghdr structure provide a clean way to pass information in addition to the data that is being read or written. The inclusion of the msg_control and msg_controllen members of the msghdr structure along with the cmsghdr structure that is pointed to by the msg_control member is required by the Posix.1g sockets API standard (which should be completed during 1997).

In this document ancillary data is used to exchange the following optional information between the application and the kernel:

1. the send/receive interface and source/destination address,
2. the hop limit,
3. next hop address,
4. Hop-by-Hop options,
5. Destination options, and
6. Routing header.

Before describing these uses in detail, we review the definition of the msghdr structure itself, the cmsghdr structure that defines an ancillary data object, and some functions that operate on the ancillary data objects.

4.1. The msghdr Structure

The msghdr structure is used by the recvmsg() and sendmsg() functions. Its Posix.1g definition is:

```
 struct msghdr {
   void      *msg_name;        /* ptr to socket address structure */
   socklen_t  msg_namelen;     /* size of socket address structure */
   struct iovec  *msg_iov;     /* scatter/gather array */
   size_t     msg_iovlen;      /* # elements in msg_iov */
   void      *msg_control;     /* ancillary data */
   socklen_t  msg_controllen;  /* ancillary data buffer length */
   int        msg_flags;       /* flags on received message */
 };
```

The structure is declared as a result of including <sys/socket.h>.

(Note: Before Posix.1g the two "void *" pointers were typically "char *", and the two socklen_t members and the size_t member were typically integers. Earlier drafts of Posix.1g had the two socklen_t members as size_t, but Draft 6.6 of Posix.1g, apparently the final draft, changed these to socklen_t to simplify binary portability for 64-bit implementations and to align Posix.1g with X/Open's Networking Services, Issue 5. The change in msg_control to a "void *" pointer affects any code that increments this pointer.)

Most Berkeley-derived implementations limit the amount of ancillary data in a call to sendmsg() to no more than 108 bytes (an mbuf). This API requires a minimum of 10240 bytes of ancillary data, but it is recommended that the amount be limited only by the buffer space reserved by the socket (which can be modified by the SO_SNDBUF socket option). (Note: This magic number 10240 was picked as a value that should always be large enough. 108 bytes is clearly too small as the maximum size of a Type 0 Routing header is 376 bytes.)

4.2. The cmsghdr Structure

The cmsghdr structure describes ancillary data objects transferred by recvmsg() and sendmsg(). Its Posix.1g definition is:

```
 struct cmsghdr {
   socklen_t  cmsg_len;   /* #bytes, including this header */
   int        cmsg_level; /* originating protocol */
   int        cmsg_type;  /* protocol-specific type */
              /* followed by unsigned char cmsg_data[]; */
 };
```

This structure is declared as a result of including <sys/socket.h>.

As shown in this definition, normally there is no member with the name cmsg_data[]. Instead, the data portion is accessed using the CMSG_xxx() macros, as described shortly. Nevertheless, it is common to refer to the cmsg_data[] member.

(Note: Before Posix.1g the cmsg_len member was an integer, and not a socklen_t. See the Note in the previous section for why socklen_t is used here.)

When ancillary data is sent or received, any number of ancillary data objects can be specified by the msg_control and msg_controllen members of the msghdr structure, because each object is preceded by a cmsghdr structure defining the object's length (the cmsg_len member). Historically Berkeley-derived implementations have passed only one object at a time, but this API allows multiple objects to be passed in a single call to sendmsg() or recvmsg(). The following example shows two ancillary data objects in a control buffer.

```
|<--------------------------- msg_controllen -------------------------->|
|                                                                       |
|<----- ancillary data object ----->|<----- ancillary data object ----->|
|<---------- CMSG_SPACE() --------->|<---------- CMSG_SPACE() --------->|
|                                   |                                   |
|<---------- cmsg_len ---------->|  |<--------- cmsg_len ----------->|  |
|<--------- CMSG_LEN() --------->|  |<-------- CMSG_LEN() ---------->|  |
|                                |  |                                |  |
+-----+-----+-----+--+-----------+--+-----+-----+-----+--+-----------+--+
|cmsg_|cmsg_|cmsg_|XX|           |XX|cmsg_|cmsg_|cmsg_|XX|           |XX|
|len  |level|type |XX|cmsg_data[]|XX|len  |level|type |XX|cmsg_data[]|XX|
+-----+-----+-----+--+-----------+--+-----+-----+-----+--+-----------+--+
 ^
 |
msg_control
points here
```

The fields shown as "XX" are possible padding, between the cmsghdr structure and the data, and between the data and the next cmsghdr structure, if required by the implementation.

4.3. Ancillary Data Object Macros

To aid in the manipulation of ancillary data objects, three macros from 4.4BSD are defined by Posix.1g: CMSG_DATA(), CMSG_NXTHDR(), and CMSG_FIRSTHDR(). Before describing these macros, we show the following example of how they might be used with a call to recvmsg().

```
 struct msghdr   msg;
 struct cmsghdr  *cmsgptr;
```

```
/* fill in msg */

/* call recvmsg() */

for (cmsgptr = CMSG_FIRSTHDR(&msg); cmsgptr != NULL;
     cmsgptr = CMSG_NXTHDR(&msg, cmsgptr)) {
    if (cmsgptr->cmsg_level == ... && cmsgptr->cmsg_type == ... ) {
        u_char  *ptr;

        ptr = CMSG_DATA(cmsgptr);
        /* process data pointed to by ptr */
    }
}
```

We now describe the three Posix.1g macros, followed by two more that are new with this API: CMSG_SPACE() and CMSG_LEN(). All these macros are defined as a result of including <sys/socket.h>.

4.3.1. CMSG_FIRSTHDR

```
struct cmsghdr *CMSG_FIRSTHDR(const struct msghdr *mhdr);
```

CMSG_FIRSTHDR() returns a pointer to the first cmsghdr structure in the msghdr structure pointed to by mhdr. The macro returns NULL if there is no ancillary data pointed to the by msghdr structure (that is, if either msg_control is NULL or if msg_controllen is less than the size of a cmsghdr structure).

One possible implementation could be

```
#define CMSG_FIRSTHDR(mhdr) \
    ( (mhdr)->msg_controllen >= sizeof(struct cmsghdr) ? \
      (struct cmsghdr *)(mhdr)->msg_control : \
      (struct cmsghdr *)NULL )
```

(Note: Most existing implementations do not test the value of msg_controllen, and just return the value of msg_control. The value of msg_controllen must be tested, because if the application asks recvmsg() to return ancillary data, by setting msg_control to point to the application's buffer and setting msg_controllen to the length of this buffer, the kernel indicates that no ancillary data is available by setting msg_controllen to 0 on return. It is also easier to put this test into this macro, than making the application perform the test.)

4.3.2. CMSG_NXTHDR

```
struct cmsghdr *CMSG_NXTHDR(const struct msghdr *mhdr,
                            const struct cmsghdr *cmsg);
```

CMSG_NXTHDR() returns a pointer to the cmsghdr structure describing the next ancillary data object. mhdr is a pointer to a msghdr structure and cmsg is a pointer to a cmsghdr structure. If there is not another ancillary data object, the return value is NULL.

The following behavior of this macro is new to this API: if the value of the cmsg pointer is NULL, a pointer to the cmsghdr structure describing the first ancillary data object is returned. That is, CMSG_NXTHDR(mhdr, NULL) is equivalent to CMSG_FIRSTHDR(mhdr). If there are no ancillary data objects, the return value is NULL. This provides an alternative way of coding the processing loop shown earlier:

```
struct msghdr  msg;
struct cmsghdr  *cmsgptr = NULL;

/* fill in msg */

/* call recvmsg() */

while ((cmsgptr = CMSG_NXTHDR(&msg, cmsgptr)) != NULL) {
    if (cmsgptr->cmsg_level == ... && cmsgptr->cmsg_type == ... ) {
        u_char  *ptr;

        ptr = CMSG_DATA(cmsgptr);
        /* process data pointed to by ptr */
    }
}
```

One possible implementation could be:

```
#define CMSG_NXTHDR(mhdr, cmsg) \
    ( ((cmsg) == NULL) ? CMSG_FIRSTHDR(mhdr) : \
      (((u_char *)(cmsg) + ALIGN((cmsg)->cmsg_len) \
                         + ALIGN(sizeof(struct cmsghdr)) > \
        (u_char *)((mhdr)->msg_control) + (mhdr)->msg_controllen) ? \
       (struct cmsghdr *)NULL : \
       (struct cmsghdr *)((u_char *)(cmsg) + ALIGN((cmsg)->cmsg_len))) )
```

The macro ALIGN(), which is implementation dependent, rounds its argument up to the next even multiple of whatever alignment is required (probably a multiple of 4 or 8 bytes).

4.3.3. CMSG_DATA

```
unsigned char *CMSG_DATA(const struct cmsghdr *cmsg);
```

CMSG_DATA() returns a pointer to the data (what is called the cmsg_data[] member, even though such a member is not defined in the structure) following a cmsghdr structure.

One possible implementation could be:

```
#define CMSG_DATA(cmsg) ( (u_char *)(cmsg) + \
                          ALIGN(sizeof(struct cmsghdr)) )
```

4.3.4. CMSG_SPACE

```
unsigned int CMSG_SPACE(unsigned int length);
```

This macro is new with this API. Given the length of an ancillary data object, CMSG_SPACE() returns the space required by the object and its cmsghdr structure, including any padding needed to satisfy alignment requirements. This macro can be used, for example, to allocate space dynamically for the ancillary data. This macro should not be used to initialize the cmsg_len member of a cmsghdr structure; instead use the CMSG_LEN() macro.

One possible implementation could be:

```
#define CMSG_SPACE(length) ( ALIGN(sizeof(struct cmsghdr)) + \
                             ALIGN(length) )
```

4.3.5. CMSG_LEN

```
unsigned int CMSG_LEN(unsigned int length);
```

This macro is new with this API. Given the length of an ancillary data object, CMSG_LEN() returns the value to store in the cmsg_len member of the cmsghdr structure, taking into account any padding needed to satisfy alignment requirements.

One possible implementation could be:

```
#define CMSG_LEN(length) ( ALIGN(sizeof(struct cmsghdr)) + length
)
```

Note the difference between CMSG_SPACE() and CMSG_LEN(), shown also in the figure in Section 4.2: the former accounts for any required padding at the end of the ancillary data object and the latter is the actual length to store in the cmsg_len member of the ancillary data object.

4.4. Summary of Options Described Using Ancillary Data

There are six types of optional information described in this document that are passed between the application and the kernel using ancillary data:

1. the send/receive interface and source/destination address,
2. the hop limit,
3. next hop address,
4. Hop-by-Hop options,
5. Destination options, and
6. Routing header.

First, to receive any of this optional information (other than the next hop address, which can only be set), the application must call setsockopt() to turn on the corresponding flag:

```
int  on = 1;

setsockopt(fd, IPPROTO_IPV6, IPV6_PKTINFO,  &on, sizeof(on));
setsockopt(fd, IPPROTO_IPV6, IPV6_HOPLIMIT, &on, sizeof(on));
setsockopt(fd, IPPROTO_IPV6, IPV6_HOPOPTS,  &on, sizeof(on));
setsockopt(fd, IPPROTO_IPV6, IPV6_DSTOPTS,  &on, sizeof(on));
setsockopt(fd, IPPROTO_IPV6, IPV6_RTHDR,    &on, sizeof(on));
```

When any of these options are enabled, the corresponding data is returned as control information by recvmsg(), as one or more ancillary data objects.

Nothing special need be done to send any of this optional information; the application just calls sendmsg() and specifies one or more ancillary data objects as control information.

We also summarize the three cmsghdr fields that describe the ancillary data objects:

cmsg_level	cmsg_type	cmsg_data[]	#times
IPPROTO_IPV6	IPV6_PKTINFO	in6_pktinfo structure	once
IPPROTO_IPV6	IPV6_HOPLIMIT	int	once
IPPROTO_IPV6	IPV6_NEXTHOP	socket address structure	once
IPPROTO_IPV6	IPV6_HOPOPTS	implementation dependent	mult.

```
IPPROTO_IPV6  IPV6_DSTOPTS   implementation dependent  mult.
IPPROTO_IPV6  IPV6_RTHDR     implementation dependent  once
```

The final column indicates how many times an ancillary data object of that type can appear as control information. The Hop-by-Hop and Destination options can appear multiple times, while all the others can appear only one time.

All these options are described in detail in following sections. All the constants beginning with IPV6_ are defined as a result of including the <netinet/in.h> header.

(Note: We intentionally use the same constant for the cmsg_level member as is used as the second argument to getsockopt() and setsockopt() (what is called the "level"), and the same constant for the cmsg_type member as is used as the third argument to getsockopt() and setsockopt() (what is called the "option name"). This is consistent with the existing use of ancillary data in 4.4BSD: returning the destination address of an IPv4 datagram.)

(Note: It is up to the implementation what it passes as ancillary data for the Hop-by-Hop option, Destination option, and Routing header option, since the API to these features is through a set of inet6_option_XXX() and inet6_rthdr_XXX() functions that we define later. These functions serve two purposes: to simplify the interface to these features (instead of requiring the application to know the intimate details of the extension header formats), and to hide the actual implementation from the application. Nevertheless, we show some examples of these features that store the actual extension header as the ancillary data. Implementations need not use this technique.)

4.5. IPV6_PKTOPTIONS Socket Option

The summary in the previous section assumes a UDP socket. Sending and receiving ancillary data is easy with UDP: the application calls sendmsg() and recvmsg() instead of sendto() and recvfrom().

But there might be cases where a TCP application wants to send or receive this optional information. For example, a TCP client might want to specify a Routing header and this needs to be done before calling connect(). Similarly a TCP server might want to know the received interface after accept() returns along with any Destination options.

A new socket option is defined that provides access to the optional information described in the previous section, but without using recvmsg() and sendmsg(). Setting the socket option specifies any of the optional output fields:

```
setsockopt(fd, IPPROTO_IPV6, IPV6_PKTOPTIONS, &buf, len);
```

The fourth argument points to a buffer containing one or more ancillary data objects, and the fifth argument is the total length of all these objects. The application fills in this buffer exactly as if the buffer were being passed to sendmsg() as control information.

The options set by calling setsockopt() for IPV6_PKTOPTIONS are called "sticky" options because once set they apply to all packets sent on that socket. The application can call setsockopt() again to change all the sticky options, or it can call setsockopt() with a length of 0 to remove all the sticky options for the socket.

The corresponding receive option

```
getsockopt(fd, IPPROTO_IPV6, IPV6_PKTOPTIONS, &buf, &len);
```

returns a buffer with one or more ancillary data objects for all the optional receive information that the application has previously specified that it wants to receive. The fourth argument points to the buffer that is filled in by the call. The fifth argument is a pointer to a value-result integer: when the function is called the integer specifies the size of the buffer pointed to by the fourth argument, and on return this integer contains the actual number of bytes that were returned. The application processes this buffer exactly as if the buffer were returned by recvmsg() as control information.

To simplify this document, in the remaining sections when we say "can be specified as ancillary data to sendmsg()" we mean "can be specified as ancillary data to sendmsg() or specified as a sticky option using setsockopt() and the IPV6_PKTOPTIONS socket option". Similarly when we say "can be returned as ancillary data by recvmsg()" we mean "can be returned as ancillary data by recvmsg() or returned by getsockopt() with the IPV6_PKTOPTIONS socket option".

4.5.1. TCP Sticky Options

When using getsockopt() with the IPV6_PKTOPTIONS option and a TCP socket, only the options from the most recently received segment are retained and returned to the caller, and only after the socket option has been set. That is, TCP need not start saving a copy of the options until the application says to do so.

The application is not allowed to specify ancillary data in a call to sendmsg() on a TCP socket, and none of the ancillary data that we describe in this document is ever returned as control information by recvmsg() on a TCP socket.

4.5.2. UDP and Raw Socket Sticky Options

The IPV6_PKTOPTIONS socket option can also be used with a UDP socket or with a raw IPv6 socket, normally to set some of the options once, instead of with each call to sendmsg().

Unlike the TCP case, the sticky options can be overridden on a per-packet basis with ancillary data specified in a call to sendmsg() on a UDP or raw IPv6 socket. If any ancillary data is specified in a call to sendmsg(), none of the sticky options are sent with that datagram.

5. Packet Information

There are four pieces of information that an application can specify for an outgoing packet using ancillary data:

1. the source IPv6 address,
2. the outgoing interface index,
3. the outgoing hop limit, and
4. the next hop address.

Three similar pieces of information can be returned for a received packet as ancillary data:

1. the destination IPv6 address,
2. the arriving interface index, and
3. the arriving hop limit.

The first two pieces of information are contained in an in6_pktinfo structure that is sent as ancillary data with sendmsg() and received as ancillary data with recvmsg(). This structure is defined as a result of including the <netinet/in.h> header.

```
struct in6_pktinfo {
  struct in6_addr ipi6_addr;    /* src/dst IPv6 address */
  unsigned int    ipi6_ifindex; /* send/recv interface index */
};
```

In the cmsghdr structure containing this ancillary data, the cmsg_level member will be IPPROTO_IPV6, the cmsg_type member will be IPV6_PKTINFO, and the first byte of cmsg_data[] will be the first byte of the in6_pktinfo structure.

This information is returned as ancillary data by recvmsg() only if the application has enabled the IPV6_PKTINFO socket option:

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_PKTINFO, &on, sizeof(on));
```

Nothing special need be done to send this information: just specify the control information as ancillary data for sendmsg().

(Note: The hop limit is not contained in the in6_pktinfo structure for the following reason. Some UDP servers want to respond to client requests by sending their reply out the same interface on which the request was received and with the source IPv6 address of the reply equal to the destination IPv6 address of the request. To do this the application can enable just the IPV6_PKTINFO socket option and then use the received control information from recvmsg() as the outgoing control information for sendmsg(). The application need not examine or modify the in6_pktinfo structure at all. But if the hop limit were contained in this structure, the application would have to parse the received control information and change the hop limit member, since the received hop limit is not the desired value for an outgoing packet.)

5.1. Specifying/Receiving the Interface

Interfaces on an IPv6 node are identified by a small positive integer, as described in Section 4 of [RFC-2133]. That document also describes a function to map an interface name to its interface index, a function to map an interface index to its interface name, and a function to return all the interface names and indexes. Notice from this document that no interface is ever assigned an index of 0.

When specifying the outgoing interface, if the ipi6_ifindex value is 0, the kernel will choose the outgoing interface. If the application specifies an outgoing interface for a multicast packet, the interface specified by the ancillary data overrides any interface specified by the IPV6_MULTICAST_IF socket option (described in [RFC-2133]), for that call to sendmsg() only.

When the IPV6_PKTINFO socket option is enabled, the received interface index is always returned as the ipi6_ifindex member of the in6_pktinfo structure.

5.2. Specifying/Receiving Source/Destination Address

The source IPv6 address can be specified by calling bind() before each output operation, but supplying the source address together with the data requires less overhead (i.e., fewer system calls) and requires less state to be stored and protected in a multithreaded application.

When specifying the source IPv6 address as ancillary data, if the ipi6_addr member of the in6_pktinfo structure is the unspecified address (IN6ADDR_ANY_INIT), then (a) if an address is currently bound to the socket, it is used as the source address, or (b) if no address is currently bound to the socket, the kernel will choose the source address. If the ipi6_addr member is not the unspecified address, but the socket has already bound a source address, then the ipi6_addr value overrides the already-bound source address for this output operation only.

The kernel must verify that the requested source address is indeed a unicast address assigned to the node.

When the in6_pktinfo structure is returned as ancillary data by recvmsg(), the ipi6_addr member contains the destination IPv6 address from the received packet.

5.3. Specifying/Receiving the Hop Limit

The outgoing hop limit is normally specified with either the IPV6_UNICAST_HOPS socket option or the IPV6_MULTICAST_HOPS socket option, both of which are described in [RFC-2133]. Specifying the hop limit as ancillary data lets the application override either the kernel’s default or a previously specified value, for either a unicast destination or a multicast destination, for a single output operation. Returning the received hop limit is useful for programs such as Traceroute and for IPv6 applications that need to verify that the received hop limit is 255 (e.g., that the packet has not been forwarded).

The received hop limit is returned as ancillary data by recvmsg() only if the application has enabled the IPV6_HOPLIMIT socket option:

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_HOPLIMIT, &on, sizeof(on));
```

In the cmsghdr structure containing this ancillary data, the cmsg_level member will be IPPROTO_IPV6, the cmsg_type member will be IPV6_HOPLIMIT, and the first byte of cmsg_data[] will be the first byte of the integer hop limit.

Nothing special need be done to specify the outgoing hop limit: just specify the control information as ancillary data for sendmsg(). As specified in [RFC-2133], the interpretation of the integer hop limit value is

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

5.4. Specifying the Next Hop Address

The IPV6_NEXTHOP ancillary data object specifies the next hop for the datagram as a socket address structure. In the cmsghdr structure containing this ancillary data, the cmsg_level member will be IPPROTO_IPV6, the cmsg_type member will be IPV6_NEXTHOP, and the first byte of cmsg_data[] will be the first byte of the socket address structure.

This is a privileged option. (Note: It is implementation defined and beyond the scope of this document to define what "privileged" means. Unix systems use this term to mean the process must have an effective user ID of 0.)

If the socket address structure contains an IPv6 address (e.g., the sin6_family member is AF_INET6), then the node identified by that address must be a neighbor of the sending host. If that address equals the destination IPv6 address of the datagram, then this is equivalent to the existing SO_DONTROUTE socket option.

5.5. Additional Errors with sendmsg()

With the IPV6_PKTINFO socket option there are no additional errors possible with the call to recvmsg(). But when specifying the outgoing interface or the source address, additional errors are possible from sendmsg(). The following are examples, but some of these may not be provided by some implementations, and some implementations may define additional errors:

ENXIO The interface specified by ipi6_ifindex does not exist.

ENETDOWN The interface specified by ipi6_ifindex is not enabled for IPv6 use.

EADDRNOTAVAIL ipi6_ifindex specifies an interface but the address ipi6_addr is not available for use on that interface.

EHOSTUNREACH No route to the destination exists over the interface specified by ifi6_ifindex.

6. Hop-By-Hop Options

A variable number of Hop-by-Hop options can appear in a single Hop-by-Hop options header. Each option in the header is TLV-encoded with a type, length, and value.

Today only three Hop-by-Hop options are defined for IPv6 [RFC-1883]: Jumbo Payload, Pad1, and PadN, although a proposal exists for a router-alert Hop-by-Hop option. The Jumbo Payload option should not be passed back to an application and an application should receive an error if it attempts to set it. This option is processed entirely by the kernel. It is indirectly specified by datagram-based applications as the size of the datagram to send and indirectly passed back to these applications as the length of the received datagram. The two pad options are for alignment purposes and are automatically inserted by a sending kernel when needed and ignored by the receiving kernel. This section of the API is therefore defined for future Hop-by-Hop options that an application may need to specify and receive.

Individual Hop-by-Hop options (and Destination options, which are described shortly, and which are similar to the Hop-by-Hop options) may have specific alignment requirements. For example, the 4-byte Jumbo Payload length should appear on a 4-byte boundary, and IPv6 addresses are normally aligned on an 8-byte boundary. These requirements and the terminology used with these options are discussed in Section 4.2 and Appendix A of [RFC-1883]. The alignment of each option is specified by two values, called x and y, written as "xn + y". This states that the option must appear at an integer multiple of x bytes from the beginning of the options header (x can have the values 1, 2, 4, or 8), plus y bytes (y can have a value between 0 and 7, inclusive). The Pad1 and PadN options are inserted as needed to maintain the required alignment. Whatever code builds either a Hop-by-Hop options header or a Destination options header must know the values of x and y for each option.

Multiple Hop-by-Hop options can be specified by the application. Normally one ancillary data object describes all the Hop-by-Hop options (since each option is itself TLV-encoded) but the application can specify multiple ancillary data objects for the Hop-by-Hop options, each object specifying one or more options. Care must be taken designing the API for these options since

1. it may be possible for some future Hop-by-Hop options to be generated by the application and processed entirely by the application (e.g., the kernel may not know the alignment restrictions for the option),

2. it must be possible for the kernel to insert its own Hop-by-Hop options in an outgoing packet (e.g., the Jumbo Payload option),

3. the application can place one or more Hop-by-Hop options into a single ancillary data object,

4. if the application specifies multiple ancillary data objects, each containing one or more Hop-by-Hop options, the kernel must combine these a single Hop-by-Hop options header, and

5. it must be possible for the kernel to remove some Hop-by-Hop options from a received packet before returning the remaining Hop-by-Hop options to the application. (This removal might consist of the kernel converting the option into a pad option of the same length.)

Finally, we note that access to some Hop-by-Hop options or to some Destination options, might require special privilege. That is, normal applications (without special privilege) might be forbidden from setting certain options in outgoing packets, and might never see certain options in received packets.

6.1. Receiving Hop-by-Hop Options

To receive Hop-by-Hop options the application must enable the IPV6_HOPOPTS socket option:

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_HOPOPTS, &on, sizeof(on));
```

All the Hop-by-Hop options are returned as one ancillary data object described by a cmsghdr structure. The cmsg_level member will be IPPROTO_IPV6 and the cmsg_type member will be IPV6_HOPOPTS. These options are then processed by calling the inet6_option_next() and inet6_option_find() functions, described shortly.

6.2. Sending Hop-by-Hop Options

To send one or more Hop-by-Hop options, the application just specifies them as ancillary data in a call to sendmsg(). No socket option need be set.

Normally all the Hop-by-Hop options are specified by a single ancillary data object. Multiple ancillary data objects, each containing one or more Hop-by-Hop options, can also be specified, in which case the kernel will combine all the Hop-by-Hop options into a single Hop-by-Hop extension header. But it should be more efficient to use a single ancillary data object to describe all the Hop-by-Hop options. The cmsg_level member is set to IPPROTO_IPV6 and the cmsg_type member is set to IPV6_HOPOPTS. The option is normally constructed using the inet6_option_init(), inet6_option_append(), and inet6_option_alloc() functions, described shortly.

Additional errors may be possible from sendmsg() if the specified option is in error.

6.3. Hop-by-Hop and Destination Options Processing

Building and parsing the Hop-by-Hop and Destination options is complicated for the reasons given earlier. We therefore define a set of functions to help the application. The function prototypes for these functions are all in the <netinet/in.h> header.

6.3.1. inet6_option_space

```
int inet6_option_space(int nbytes);
```

This function returns the number of bytes required to hold an option when it is stored as ancillary data, including the cmsghdr structure at the beginning, and any padding at the end (to make its size a multiple of 8 bytes). The argument is the size of the structure defining the option, which must include any pad bytes at the beginning (the value y in the alignment term "xn + y"), the type byte, the length byte, and the option data.

(Note: If multiple options are stored in a single ancillary data object, which is the recommended technique, this function overestimates the amount of space required by the size of N-1 cmsghdr structures, where N is the number of options to be stored in the object. This is of little consequence, since it is assumed that most Hop-by-Hop option headers and Destination option headers carry only one option (p. 33 of [RFC-1883]).)

6.3.2. inet6_option_init

```
int inet6_option_init(void *bp, struct cmsghdr **cmsgp, int
type);
```

This function is called once per ancillary data object that will contain either Hop-by-Hop or Destination options. It returns 0 on success or -1 on an error.

bp is a pointer to previously allocated space that will contain the ancillary data object. It must be large enough to contain all the individual options to be added by later calls to inet6_option_append() and inet6_option_alloc().

cmsgp is a pointer to a pointer to a cmsghdr structure. *cmsgp is initialized by this function to point to the cmsghdr structure constructed by this function in the buffer pointed to by bp.

type is either IPV6_HOPOPTS or IPV6_DSTOPTS. This type is stored in the cmsg_type member of the cmsghdr structure pointed to by *cmsgp.

6.3.3. inet6_option_append

```
int inet6_option_append(struct cmsghdr *cmsg, const uint8_t *typep,
                        int multx, int plusy);
```

This function appends a Hop-by-Hop option or a Destination option into an ancillary data object that has been initialized by inet6_option_init(). This function returns 0 if it succeeds or -1 on an error.

cmsg is a pointer to the cmsghdr structure that must have been initialized by inet6_option_init().

typep is a pointer to the 8-bit option type. It is assumed that this field is immediately followed by the 8-bit option data length field, which is then followed immediately by the option data. The caller initializes these three fields (the type-length-value, or TLV) before calling this function.

The option type must have a value from 2 to 255, inclusive. (0 and 1 are reserved for the Pad1 and PadN options, respectively.)

The option data length must have a value between 0 and 255, inclusive, and is the length of the option data that follows.

multx is the value x in the alignment term "xn + y" described earlier. It must have a value of 1, 2, 4, or 8.

plusy is the value y in the alignment term "xn + y" described earlier. It must have a value between 0 and 7, inclusive.

6.3.4. inet6_option_alloc

```
uint8_t *inet6_option_alloc(struct cmsghdr *cmsg, int datalen,
                             int multx, int plusy);
```

This function appends a Hop-by-Hop option or a Destination option into an ancillary data object that has been initialized by inet6_option_init(). This function returns a pointer to the 8-bit option type field that starts the option on success, or NULL on an error.

The difference between this function and inet6_option_append() is that the latter copies the contents of a previously built option into the ancillary data object while the current function returns a pointer to the space in the data object where the option's TLV must then be built by the caller.

cmsg is a pointer to the cmsghdr structure that must have been initialized by inet6_option_init().

datalen is the value of the option data length byte for this option. This value is required as an argument to allow the function to determine if padding must be appended at the end of the option. (The inet6_option_append() function does not need a data length argument since the option data length must already be stored by the caller.)

multx is the value x in the alignment term "xn + y" described earlier. It must have a value of 1, 2, 4, or 8.

plusy is the value y in the alignment term "xn + y" described earlier. It must have a value between 0 and 7, inclusive.

6.3.5. inet6_option_next

```
int inet6_option_next(const struct cmsghdr *cmsg, uint8_t
**tptrp);
```

This function processes the next Hop-by-Hop option or Destination option in an ancillary data object. If another option remains to be processed, the return value of the function is 0 and *tptrp points to the 8-bit option type field (which is followed by the 8-bit option data length, followed by the option data). If no more options remain to be processed, the return value is -1 and *tptrp is NULL. If an error occurs, the return value is -1 and *tptrp is not NULL.

cmsg is a pointer to cmsghdr structure of which cmsg_level equals IPPROTO_IPV6 and cmsg_type equals either IPV6_HOPOPTS or IPV6_DSTOPTS.

tptrp is a pointer to a pointer to an 8-bit byte and *tptrp is used by the function to remember its place in the ancillary data object each time the function is called. The first time this function is called for a given ancillary data object, *tptrp must be set to NULL.

Each time this function returns success, *tptrp points to the 8-bit option type field for the next option to be processed.

6.3.6. inet6_option_find

```
int inet6_option_find(const struct cmsghdr *cmsg, uint8_t *tptrp,
                      int type);
```

This function is similar to the previously described inet6_option_next() function, except this function lets the caller specify the option type to be searched for, instead of always returning the next option in the ancillary data object. cmsg is a pointer to cmsghdr structure of which cmsg_level equals IPPROTO_IPV6 and cmsg_type equals either IPV6_HOPOPTS or IPV6_DSTOPTS.

tptrp is a pointer to a pointer to an 8-bit byte and *tptrp is used by the function to remember its place in the ancillary data object each time the function is called. The first time this function is called for a given ancillary data object, *tptrp must be set to NULL.

This function starts searching for an option of the specified type beginning after the value of *tptrp. If an option of the specified type is located, this function returns 0 and *tptrp points to the 8-bit option type field for the option of the specified type. If an option of the specified type is not located, the return value is -1 and *tptrp is NULL. If an error occurs, the return value is -1 and *tptrp is not NULL.

6.3.7. Options Examples

We now provide an example that builds two Hop-by-Hop options. First we define two options, called X and Y, taken from the example in Appendix A of [RFC-1883]. We assume that all options will have structure definitions similar to what is shown below.

```
        /* option X and option Y are defined in [RFC-1883], pp. 33-34 */
#define IP6_X_OPT_TYPE       X   /* replace X with assigned value */
#define IP6_X_OPT_LEN       12
#define IP6_X_OPT_MULTX      8   /* 8n + 2 alignment */
#define IP6_X_OPT_OFFSETY    2

struct ip6_X_opt {
  uint8_t   ip6_X_opt_pad[IP6_X_OPT_OFFSETY];
  uint8_t   ip6_X_opt_type;
  uint8_t   ip6_X_opt_len;
  uint32_t  ip6_X_opt_val1;
  uint64_t  ip6_X_opt_val2;
};
```

```
#define IP6_Y_OPT_TYPE       Y   /* replace Y with assigned value */
#define IP6_Y_OPT_LEN        7
#define IP6_Y_OPT_MULTX      4   /* 4n + 3 alignment */
#define IP6_Y_OPT_OFFSETY    3

struct ip6_Y_opt {
  uint8_t   ip6_Y_opt_pad[IP6_Y_OPT_OFFSETY];
  uint8_t   ip6_Y_opt_type;
  uint8_t   ip6_Y_opt_len;
  uint8_t   ip6_Y_opt_val1;
  uint16_t  ip6_Y_opt_val2;
  uint32_t  ip6_Y_opt_val3;
};
```

We now show the code fragment to build one ancillary data object containing both options.

```
struct msghdr  msg;
struct cmsghdr  *cmsgptr;
struct ip6_X_opt  optX;
struct ip6_Y_opt  optY;

msg.msg_control = malloc(inet6_option_space(sizeof(optX) +
                                            sizeof(optY)));

inet6_option_init(msg.msg_control, &cmsgptr, IPV6_HOPOPTS);

optX.ip6_X_opt_type = IP6_X_OPT_TYPE;
optX.ip6_X_opt_len  = IP6_X_OPT_LEN;
optX.ip6_X_opt_val1 = <32-bit value>;
optX.ip6_X_opt_val2 = <64-bit value>;
inet6_option_append(cmsgptr, &optX.ip6_X_opt_type,
                    IP6_X_OPT_MULTX, IP6_X_OPT_OFFSETY);

optY.ip6_Y_opt_type = IP6_Y_OPT_TYPE;
optY.ip6_Y_opt_len  = IP6_Y_OPT_LEN;
optY.ip6_Y_opt_val1 = <8-bit value>;
optY.ip6_Y_opt_val2 = <16-bit value>;
optY.ip6_Y_opt_val3 = <32-bit value>;
inet6_option_append(cmsgptr, &optY.ip6_Y_opt_type,
                    IP6_Y_OPT_MULTX, IP6_Y_OPT_OFFSETY);

msg.msg_controllen = cmsgptr->cmsg_len;
```

The call to inet6_option_init() builds the cmsghdr structure in the control buffer.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_len = CMSG_LEN(0) = 12                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_level = IPPROTO_IPV6                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_type = IPV6_HOPOPTS                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Here we assume a 32-bit architecture where sizeof(struct cmsghdr) equals 12, with a desired alignment of 4-byte boundaries (that is, the ALIGN() macro shown in the sample implementations of the CMSG_xxx() macros rounds up to a multiple of 4).

The first call to inet6_option_append() appends the X option. Since this is the first option in the ancillary data object, 2 bytes are allocated for the Next Header byte and for the Hdr Ext Len byte. The former will be set by the kernel, depending on the type of header that follows this header, and the latter byte is set to 1. These 2 bytes form the 2 bytes of padding (IP6_X_OPT_OFFSETY) required at the beginning of this option.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_len = 28                                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_level = IPPROTO_IPV6                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_type = IPV6_HOPOPTS                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next Header  | Hdr Ext Len=1 | Option Type=X |Opt Data Len=12|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         4-octet field                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         8-octet field                         +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The cmsg_len member of the cmsghdr structure is incremented by 16, the size of the option.

The next call to inet6_option_append() appends the Y option to the ancillary data object.

```
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |       cmsg_len = 44                                           |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |       cmsg_level = IPPROTO_IPV6                               |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |       cmsg_type = IPV6_HOPOPTS                                |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |  Next Header  | Hdr Ext Len=3 | Option Type=X |Opt Data Len=12|
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                         4-octet field                         |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                                                               |
 +                         8-octet field                         +
 |                                                               |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 | PadN Option=1 |Opt Data Len=1 |       0       | Option Type=Y |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |Opt Data Len=7 | 1-octet field |         2-octet field         |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                         4-octet field                         |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 | PadN Option=1 |Opt Data Len=2 |       0       |       0       |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

16 bytes are appended by this function, so cmsg_len becomes 44. The inet6_option_append() function notices that the appended data requires 4 bytes of padding at the end, to make the size of the ancillary data object a multiple of 8, and appends the PadN option before returning. The Hdr Ext Len byte is incremented by 2 to become 3.

Alternately, the application could build two ancillary data objects, one per option, although this will probably be less efficient than combining the two options into a single ancillary data object (as just shown). The kernel must combine these into a single Hop-by-Hop extension header in the final IPv6 packet.

```
struct msghdr  msg;
struct cmsghdr  *cmsgptr;
struct ip6_X_opt  optX;
struct ip6_Y_opt  optY;

msg.msg_control = malloc(inet6_option_space(sizeof(optX)) +
                         inet6_option_space(sizeof(optY)));

inet6_option_init(msg.msg_control, &cmsgptr, IPPROTO_HOPOPTS);

optX.ip6_X_opt_type = IP6_X_OPT_TYPE;
```

```
    optX.ip6_X_opt_len  = IP6_X_OPT_LEN;
    optX.ip6_X_opt_val1 = <32-bit value>;
    optX.ip6_X_opt_val2 = <64-bit value>;
    inet6_option_append(cmsgptr, &optX.ip6_X_opt_type,
                        IP6_X_OPT_MULTX, IP6_X_OPT_OFFSETY);
    msg.msg_controllen = CMSG_SPACE(sizeof(optX));

    inet6_option_init((u_char *)msg.msg_control + msg.msg_controllen,
                      &cmsgptr, IPPROTO_HOPOPTS);

    optY.ip6_Y_opt_type = IP6_Y_OPT_TYPE;
    optY.ip6_Y_opt_len  = IP6_Y_OPT_LEN;
    optY.ip6_Y_opt_val1 = <8-bit value>;
    optY.ip6_Y_opt_val2 = <16-bit value>;
    optY.ip6_Y_opt_val3 = <32-bit value>;
    inet6_option_append(cmsgptr, &optY.ip6_Y_opt_type,
                        IP6_Y_OPT_MULTX, IP6_Y_OPT_OFFSETY);
    msg.msg_controllen += cmsgptr->cmsg_len;
```

Each call to inet6_option_init() builds a new cmsghdr structure, and the final result looks like the following:

```
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |       cmsg_len = 28                                            |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |       cmsg_level = IPPROTO_IPV6                                |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |       cmsg_type = IPV6_HOPOPTS                                 |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |  Next Header  | Hdr Ext Len=1 | Option Type=X |Opt Data Len=12|
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |                         4-octet field                         |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |                                                               |
     +                         8-octet field                         +
     |                                                               |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |       cmsg_len = 28                                            |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |       cmsg_level = IPPROTO_IPV6                                |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |       cmsg_type = IPV6_HOPOPTS                                 |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |  Next Header  | Hdr Ext Len=1 | Pad1 Option=0 | Option Type=Y |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |Opt Data Len=7 | 1-octet field |         2-octet field         |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |                         4-octet field                         |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     | PadN Option=1 |Opt Data Len=2 |       0       |       0       |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

When the kernel combines these two options into a single Hop-by-Hop extension header, the first 3 bytes of the second ancillary data object (the Next Header byte, the Hdr Ext Len byte, and the Pad1 option) will be combined into a PadN option occupying 3 bytes.

The following code fragment is a redo of the first example shown (building two options in a single ancillary data object) but this time we use inet6_option_alloc().

```
uint8_t  *typep;
struct msghdr  msg;
struct cmsghdr  *cmsgptr;
struct ip6_X_opt  *optXp;  /* now a pointer, not a struct */
struct ip6_Y_opt  *optYp;  /* now a pointer, not a struct */

msg.msg_control = malloc(inet6_option_space(sizeof(*optXp) +
                                            sizeof(*optYp)));
```

```
inet6_option_init(msg.msg_control, &cmsgptr, IPV6_HOPOPTS);

typep = inet6_option_alloc(cmsgptr, IP6_X_OPT_LEN,
                           IP6_X_OPT_MULTX, IP6_X_OPT_OFFSETY);
optXp = (struct ip6_X_opt *) (typep - IP6_X_OPT_OFFSETY);
optXp->ip6_X_opt_type = IP6_X_OPT_TYPE;
optXp->ip6_X_opt_len  = IP6_X_OPT_LEN;
optXp->ip6_X_opt_val1 = <32-bit value>;
optXp->ip6_X_opt_val2 = <64-bit value>;

typep = inet6_option_alloc(cmsgptr, IP6_Y_OPT_LEN,
                           IP6_Y_OPT_MULTX, IP6_Y_OPT_OFFSETY);
optYp = (struct ip6_Y_opt *) (typep - IP6_Y_OPT_OFFSETY);
optYp->ip6_Y_opt_type = IP6_Y_OPT_TYPE;
optYp->ip6_Y_opt_len  = IP6_Y_OPT_LEN;
optYp->ip6_Y_opt_val1 = <8-bit value>;
optYp->ip6_Y_opt_val2 = <16-bit value>;
optYp->ip6_Y_opt_val3 = <32-bit value>;

msg.msg_controllen = cmsgptr->cmsg_len;
```

Notice that inet6_option_alloc() returns a pointer to the 8-bit option type field. If the program wants a pointer to an option structure that includes the padding at the front (as shown in our definitions of the ip6_X_opt and ip6_Y_opt structures), the y-offset at the beginning of the structure must be subtracted from the returned pointer.

The following code fragment shows the processing of Hop-by-Hop options using the inet6_option_next() function.

```
 struct msghdr   msg;
 struct cmsghdr  *cmsgptr;

 /* fill in msg */

 /* call recvmsg() */

 for (cmsgptr = CMSG_FIRSTHDR(&msg); cmsgptr != NULL;
      cmsgptr = CMSG_NXTHDR(&msg, cmsgptr)) {
     if (cmsgptr->cmsg_level == IPPROTO_IPV6 &&
         cmsgptr->cmsg_type == IPV6_HOPOPTS) {

         uint8_t  *tptr = NULL;

         while (inet6_option_next(cmsgptr, &tptr) == 0) {
             if (*tptr == IP6_X_OPT_TYPE) {
                 struct ip6_X_opt  *optXp;
```

```
                    optXp = (struct ip6_X_opt *) (tptr - IP6_X_OPT_OFFSETY);
                    <do whatever with> optXp->ip6_X_opt_val1;
                    <do whatever with> optXp->ip6_X_opt_val2;

                } else if (*tptr == IP6_Y_OPT_TYPE) {
                    struct ip6_Y_opt  *optYp;

                    optYp = (struct ip6_Y_opt *) (tptr - IP6_Y_OPT_OFFSETY);
                    <do whatever with> optYp->ip6_Y_opt_val1;
                    <do whatever with> optYp->ip6_Y_opt_val2;
                    <do whatever with> optYp->ip6_Y_opt_val3;
                }
            }
            if (tptr != NULL)
                <error encountered by inet6_option_next()>;
        }
    }
```

7. Destination Options

A variable number of Destination options can appear in one or more Destination option headers. As defined in [RFC-1883], a Destination options header appearing before a Routing header is processed by the first destination plus any subsequent destinations specified in the Routing header, while a Destination options header appearing after a Routing header is processed only by the final destination. As with the Hop-by-Hop options, each option in a Destination options header is TLV-encoded with a type, length, and value.

Today no Destination options are defined for IPv6 [RFC-1883], although proposals exist to use Destination options with mobility and anycasting.

7.1. Receiving Destination Options

To receive Destination options the application must enable the IPV6_DSTOPTS socket option:

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_DSTOPTS, &on, sizeof(on));
```

All the Destination options appearing before a Routing header are returned as one ancillary data object described by a cmsghdr structure and all the Destination options appearing after a Routing header are returned as another ancillary data object described by a cmsghdr structure. For these ancillary data objects, the cmsg_level member will be IPPROTO_IPV6 and the cmsg_type member will be IPV6_HOPOPTS. These options are then processed by calling the inet6_option_next() and inet6_option_find() functions.

7.2. Sending Destination Options

To send one or more Destination options, the application just specifies them as ancillary data in a call to sendmsg(). No socket option need be set.

As described earlier, one set of Destination options can appear before a Routing header, and one set can appear after a Routing header. Each set can consist of one or more options.

Normally all the Destination options in a set are specified by a single ancillary data object, since each option is itself TLV-encoded. Multiple ancillary data objects, each containing one or more Destination options, can also be specified, in which case the kernel will combine all the Destination options in the set into a single Destination extension header. But it should be more efficient to use a single ancillary data object to describe all the Destination options in a set. The cmsg_level member is set to IPPROTO_IPV6 and the cmsg_type member is set to IPV6_DSTOPTS. The option is normally constructed using the inet6_option_init(), inet6_option_append(), and inet6_option_alloc() functions.

Additional errors may be possible from sendmsg() if the specified option is in error.

8. Routing Header Option

Source routing in IPv6 is accomplished by specifying a Routing header as an extension header. There can be different types of Routing headers, but IPv6 currently defines only the Type 0 Routing header [RFC-1883]. This type supports up to 23 intermediate nodes. With this maximum number of intermediate nodes, a source, and a destination, there are 24 hops, each of which is defined as a strict or loose hop.

Source routing with IPv4 sockets API (the IP_OPTIONS socket option) requires the application to build the source route in the format that appears as the IPv4 header option, requiring intimate knowledge of the IPv4 options format. This IPv6 API, however, defines eight functions that the application calls to build and examine a Routing header. Four functions build a Routing header:

```
inet6_rthdr_space()    - return #bytes required for ancillary data
inet6_rthdr_init()     - initialize ancillary data for Routing header
```

```
inet6_rthdr_add()      - add IPv6 address & flags to Routing header
inet6_rthdr_lasthop()  - specify the flags for the final hop
```

Four functions deal with a returned Routing header:

```
 inet6_rthdr_reverse()  - reverse a Routing header
 inet6_rthdr_segments() - return #segments in a Routing header
 inet6_rthdr_getaddr()  - fetch one address from a Routing header
 inet6_rthdr_getflags() - fetch one flag from a Routing header
```

The function prototypes for these functions are all in the <netinet/in.h> header.

To receive a Routing header the application must enable the IPV6_RTHDR socket option:

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_RTHDR, &on, sizeof(on));
```

To send a Routing header the application just specifies it as ancillary data in a call to sendmsg().

A Routing header is passed between the application and the kernel as an ancillary data object. The cmsg_level member has a value of IPPROTO_IPV6 and the cmsg_type member has a value of IPV6_RTHDR. The contents of the cmsg_data[] member is implementation dependent and should not be accessed directly by the application, but should be accessed using the eight functions that we are about to describe.

The following constants are defined in the <netinet/in.h> header:

```
#define IPV6_RTHDR_LOOSE     0 /* this hop need not be a neighbor */
#define IPV6_RTHDR_STRICT    1 /* this hop must be a neighbor */

#define IPV6_RTHDR_TYPE_0    0 /* IPv6 Routing header type 0 */
```

When a Routing header is specified, the destination address specified for connect(), sendto(), or sendmsg() is the final destination address of the datagram. The Routing header then contains the addresses of all the intermediate nodes.

8.1. inet6_rthdr_space

```
size_t inet6_rthdr_space(int type, int segments);
```

This function returns the number of bytes required to hold a Routing header of the specified type containing the specified number of segments (addresses). For an IPv6 Type 0 Routing header, the number of segments must be between 1 and 23, inclusive. The return value includes the size of the cmsghdr structure that precedes the Routing header, and any required padding.

If the return value is 0, then either the type of the Routing header is not supported by this implementation or the number of segments is invalid for this type of Routing header.

(Note: This function returns the size but does not allocate the space required for the ancillary data. This allows an application to allocate a larger buffer, if other ancillary data objects are desired, since all the ancillary data objects must be specified to sendmsg() as a single msg_control buffer.)

8.2. inet6_rthdr_init

```
struct cmsghdr *inet6_rthdr_init(void *bp, int type);
```

This function initializes the buffer pointed to by bp to contain a cmsghdr structure followed by a Routing header of the specified type. The cmsg_len member of the cmsghdr structure is initialized to the size of the structure plus the amount of space required by the Routing header. The cmsg_level and cmsg_type members are also initialized as required.

The caller must allocate the buffer and its size can be determined by calling inet6_rthdr_space().

Upon success the return value is the pointer to the cmsghdr structure, and this is then used as the first argument to the next two functions. Upon an error the return value is NULL.

8.3. inet6_rthdr_add

```
int inet6_rthdr_add(struct cmsghdr *cmsg,
                    const struct in6_addr *addr, unsigned int flags);
```

This function adds the address pointed to by addr to the end of the Routing header being constructed and sets the type of this hop to the value of flags. For an IPv6 Type 0 Routing header, flags must be either IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT.

If successful, the cmsg_len member of the cmsghdr structure is updated to account for the new address in the Routing header and the return value of the function is 0. Upon an error the return value of the function is -1.

8.4. inet6_rthdr_lasthop

```
int inet6_rthdr_lasthop(struct cmsghdr *cmsg,
                        unsigned int flags);
```

This function specifies the Strict/Loose flag for the final hop of a Routing header. For an IPv6 Type 0 Routing header, flags must be either IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT.

The return value of the function is 0 upon success, or -1 upon an error.

Notice that a Routing header specifying N intermediate nodes requires N+1 Strict/Loose flags. This requires N calls to inet6_rthdr_add() followed by one call to inet6_rthdr_lasthop().

8.5. inet6_rthdr_reverse

```
int inet6_rthdr_reverse(const struct cmsghdr *in, struct cmsghdr *out);
```

This function takes a Routing header that was received as ancillary data (pointed to by the first argument) and writes a new Routing header that sends datagrams along the reverse of that route. Both arguments are allowed to point to the same buffer (that is, the reversal can occur in place).

The return value of the function is 0 on success, or -1 upon an error.

8.6. inet6_rthdr_segments

```
int inet6_rthdr_segments(const struct cmsghdr *cmsg);
```

This function returns the number of segments (addresses) contained in the Routing header described by cmsg. On success the return value is between 1 and 23, inclusive. The return value of the function is -1 upon an error.

8.7. inet6_rthdr_getaddr

```
struct in6_addr *inet6_rthdr_getaddr(struct cmsghdr *cmsg, int
index);
```

This function returns a pointer to the IPv6 address specified by index (which must have a value between 1 and the value returned by inet6_rthdr_segments()) in the Routing header described by cmsg. An application should first call inet6_rthdr_segments() to obtain the number of segments in the Routing header.

Upon an error the return value of the function is NULL.

8.8. inet6_rthdr_getflags

```
int inet6_rthdr_getflags(const struct cmsghdr *cmsg, int index);
```

This function returns the flags value specified by index (which must have a value between 0 and the value returned by inet6_rthdr_segments()) in the Routing header described by cmsg. For an IPv6 Type 0 Routing header the return value will be either IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT.

Upon an error the return value of the function is -1.

(Note: Addresses are indexed starting at 1, and flags starting at 0, to maintain consistency with the terminology and figures in [RFC-1883].)

8.9. Routing Header Example

As an example of these Routing header functions, we go through the function calls for the example on p. 18 of [RFC-1883]. The source is S, the destination is D, and the three intermediate nodes are I1, I2, and I3. f0, f1, f2, and f3 are the Strict/Loose flags for each hop.

```
                f0        f1        f2        f3
            S -----> I1 -----> I2 -----> I3 -----> D

    src:    *    S         S         S         S   S
    dst:    D   I1        I2        I3         D   D
    A[1]:  I1   I2        I1        I1        I1  I1
    A[2]:  I2   I3        I3        I2        I2  I2
    A[3]:  I3    D         D         D        I3  I3
    #seg:   3    3         2         1         0   3

    check: f0        f1        f2        f3
```

src and dst are the source and destination IPv6 addresses in the IPv6 header. A[1], A[2], and A[3] are the three addresses in the Routing header. #seg is the Segments Left field in the Routing header. check indicates which bit of the Strict/Loose Bit Map (0 through 3, specified as f0 through f3) that node checks.

The six values in the column beneath node S are the values in the Routing header specified by the application using sendmsg(). The function calls by the sender would look like:

```
    void  *ptr;
    struct msghdr  msg;
    struct cmsghdr  *cmsgptr;
    struct sockaddr_in6  I1, I2, I3, D;
    unsigned int  f0, f1, f2, f3;

    ptr = malloc(inet6_rthdr_space(IPV6_RTHDR_TYPE_0, 3));
    cmsgptr = inet6_rthdr_init(ptr, IPV6_RTHDR_TYPE_0);

    inet6_rthdr_add(cmsgptr, &I1.sin6_addr, f0);
    inet6_rthdr_add(cmsgptr, &I2.sin6_addr, f1);
    inet6_rthdr_add(cmsgptr, &I3.sin6_addr, f2);
    inet6_rthdr_lasthop(cmsgptr, f3);

    msg.msg_control = ptr;
    msg.msg_controllen = cmsgptr->cmsg_len;

    /* finish filling in msg{}, msg_name = D */
    /* call sendmsg() */
```

We also assume that the source address for the socket is not specified (i.e., the asterisk in the figure).

The four columns of six values that are then shown between the five nodes are the values of the fields in the packet while the packet is in transit between the two nodes. Notice that before the packet is sent by the source node S, the source address is chosen (replacing the asterisk), I1 becomes the destination address of the datagram, the two addresses A[2] and A[3] are "shifted up", and D is moved to A[3]. If f0 is IPV6_RTHDR_STRICT, then I1 must be a neighbor of S.

The columns of values that are shown beneath the destination node are the values returned by recvmsg(), assuming the application has enabled both the IPV6_PKTINFO and IPV6_RTHDR socket options. The source address is S (contained in the sockaddr_in6 structure pointed to by the msg_name member), the destination address is D (returned as an ancillary data object in an in6_pktinfo structure), and the ancillary data object specifying the Routing header will contain three addresses (I1, I2, and I3) and four flags (f0, f1, f2, and f3). The number of segments in the Routing header is known from the Hdr Ext Len field in the Routing header (a value of 6, indicating 3 addresses).

The return value from inet6_rthdr_segments() will be 3 and inet6_rthdr_getaddr(1) will return I1, inet6_rthdr_getaddr(2) will return I2, and inet6_rthdr_getaddr(3) will return I3, The return value from inet6_rthdr_flags(0) will be f0, inet6_rthdr_flags(1) will return f1, inet6_rthdr_flags(2) will return f2, and inet6_rthdr_flags(3) will return f3.

If the receiving application then calls inet6_rthdr_reverse(), the order of the three addresses will become I3, I2, and I1, and the order of the four Strict/Loose flags will become f3, f2, f1, and f0.

We can also show what an implementation might store in the ancillary data object as the Routing header is being built by the sending process. If we assume a 32-bit architecture where sizeof(struct cmsghdr) equals 12, with a desired alignment of 4-byte boundaries, then the call to inet6_rthdr_space(3) returns 68: 12 bytes for the cmsghdr structure and 56 bytes for the Routing header (8 + 3*16).

The call to inet6_rthdr_init() initializes the ancillary data object to contain a Type 0 Routing header:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_len = 20                                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_level = IPPROTO_IPV6                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_type = IPV6_RTHDR                                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next Header  | Hdr Ext Len=0 | Routing Type=0|  Seg Left=0   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |             Strict/Loose Bit Map              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The first call to inet6_rthdr_add() adds I1 to the list.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_len = 36                                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_level = IPPROTO_IPV6                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_type = IPV6_RTHDR                                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next Header  | Hdr Ext Len=2 | Routing Type=0|  Seg Left=1   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |X|           Strict/Loose Bit Map              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                         Address[1] = I1                       +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Bit 0 of the Strict/Loose Bit Map contains the value f0, which we just mark as X. cmsg_len is incremented by 16, the Hdr Ext Len field is incremented by 2, and the Segments Left field is incremented by 1.

The next call to inet6_rthdr_add() adds I2 to the list.

```
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |       cmsg_len = 52                                           |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |       cmsg_level = IPPROTO_IPV6                               |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |       cmsg_type = IPV6_RTHDR                                  |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |  Next Header  | Hdr Ext Len=4 | Routing Type=0|  Seg Left=2   |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |   Reserved    |X|X|         Strict/Loose Bit Map              |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |                                                               |
     +                                                               +
     |                                                               |
     +                           Address[1] = I1                     +
     |                                                               |
     +                                                               +
     |                                                               |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
     |                                                               |
     +                                                               +
     |                                                               |
     +                           Address[2] = I2                     +
     |                                                               |
     +                                                               +
     |                                                               |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The next bit of the Strict/Loose Bit Map contains the value f1. cmsg_len is incremented by 16, the Hdr Ext Len field is incremented by 2, and the Segments Left field is incremented by 1.

The last call to inet6_rthdr_add() adds I3 to the list.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_len = 68                                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_level = IPPROTO_IPV6                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       cmsg_type = IPV6_RTHDR                                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next Header  | Hdr Ext Len=6 | Routing Type=0|  Seg Left=3   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Reserved    |X|X|X|       Strict/Loose Bit Map              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                         Address[1] = I1                       +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                         Address[2] = I2                       +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                         Address[3] = I3                       +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The next bit of the Strict/Loose Bit Map contains the value f2. cmsg_len is incremented by 16, the Hdr Ext Len field is incremented by 2, and the Segments Left field is incremented by 1.

Finally, the call to inet6_rthdr_lasthop() sets the next bit of the Strict/Loose Bit Map to the value specified by f3. All the lengths remain unchanged.

9. Ordering of Ancillary Data and IPv6 Extension Headers

Three IPv6 extension headers can be specified by the application and returned to the application using ancillary data with sendmsg() and recvmsg(): Hop-by-Hop options, Destination options, and the Routing header. When multiple ancillary data objects are transferred via sendmsg() or recvmsg() and these objects represent any of these three extension headers, their placement in the control buffer is directly tied to their location in the corresponding IPv6 datagram. This API imposes some ordering constraints when using multiple ancillary data objects with sendmsg().

When multiple IPv6 Hop-by-Hop options having the same option type are specified, these options will be inserted into the Hop-by-Hop options header in the same order as they appear in the control buffer. But when multiple Hop-by-Hop options having different option types are specified, these options may be reordered by the kernel to reduce padding in the Hop-by-Hop options header. Hop-by-Hop options may appear anywhere in the control buffer and will always be collected by the kernel and placed into a single Hop-by-Hop options header that immediately follows the IPv6 header.

Similar rules apply to the Destination options: (1) those of the same type will appear in the same order as they are specified, and (2) those of differing types may be reordered. But the kernel will build up to two Destination options headers: one to precede the Routing header and one to follow the Routing header. If the application specifies a Routing header then all Destination options that appear in the control buffer before the Routing header will appear in a Destination options header before the Routing header and these options might be reordered, subject to the two rules that we just stated. Similarly all Destination options that appear in the control buffer after the Routing header will appear in a Destination options header after the Routing header, and these options might be reordered, subject to the two rules that we just stated.

As an example, assume that an application specifies control information to sendmsg() containing six ancillary data objects: the first containing two Hop-by-Hop options, the second containing one Destination option, the third containing two Destination options, the fourth containing a Routing header, the fifth containing a Hop-by-Hop option, and the sixth containing two Destination options. We also assume that all the Hop-by-Hop options are of different types, as are all the Destination options. We number these options 1-9, corresponding to their order in the control buffer, and show them on the left below.

In the middle we show the final arrangement of the options in the extension headers built by the kernel. On the right we show the four ancillary data objects returned to the receiving application.

```
   Sender's                                          Receiver's
Ancillary Data      -->   IPv6 Extension   -->   Ancillary Data
   Objects                    Headers              Objects
------------------        ---------------        --------------
HOPOPT-1,2 (first)        HOPHDR(J,7,1,2)        HOPOPT-7,1,2
DSTOPT-3                  DSTHDR(4,5,3)          DSTOPT-4,5,3
DSTOPT-4,5                RTHDR(6)               RTHDR-6
RTHDR-6                   DSTHDR(8,9)            DSTOPT-8,9
HOPOPT-7
DSTOPT-8,9 (last)
```

The sender's two Hop-by-Hop ancillary data objects are reordered, as are the first two Destination ancillary data objects. We also show a Jumbo Payload option (denoted as J) inserted by the kernel before the sender's three Hop-by-Hop options. The first three Destination options must appear in a Destination header before the Routing header, and the final two Destination options must appear in a Destination header after the Routing header.

If Destination options are specified in the control buffer after a Routing header, or if Destination options are specified without a Routing header, the kernel will place those Destination options after an authentication header and/or an encapsulating security payload header, if present.

10. IPv6-Specific Options with IPv4-Mapped IPv6 Addresses

The various socket options and ancillary data specifications defined in this document apply only to true IPv6 sockets. It is possible to create an IPv6 socket that actually sends and receives IPv4 packets, using IPv4-mapped IPv6 addresses, but the mapping of the options defined in this document to an IPv4 datagram is beyond the scope of this document.

In general, attempting to specify an IPv6-only option, such as the Hop-by-Hop options, Destination options, or Routing header on an IPv6 socket that is using IPv4-mapped IPv6 addresses, will probably result in an error. Some implementations, however, may provide access to the packet information (source/destination address, send/receive interface, and hop limit) on an IPv6 socket that is using IPv4-mapped IPv6 addresses.

11. rresvport_af

The rresvport() function is used by the rcmd() function, and this function is in turn called by many of the "r" commands such as rlogin. While new applications are not being written to use the rcmd() function, legacy applications such as rlogin will continue to use it and these will be ported to IPv6.

rresvport() creates an IPv4/TCP socket and binds a "reserved port" to the socket. Instead of defining an IPv6 version of this function we define a new function that takes an address family as its argument.

```
#include <unistd.h>

int  rresvport_af(int *port, int family);
```

This function behaves the same as the existing rresvport() function, but instead of creating an IPv4/TCP socket, it can also create an IPv6/TCP socket. The family argument is either AF_INET or AF_INET6, and a new error return is EAFNOSUPPORT if the address family is not supported.

(Note: There is little consensus on which header defines the rresvport() and rcmd() function prototypes. 4.4BSD defines it in <unistd.h>, others in <netdb.h>, and others don’t define the function prototypes at all.)

(Note: We define this function only, and do not define something like rcmd_af() or rcmd6(). The reason is that rcmd() calls gethostbyname(), which returns the type of address: AF_INET or AF_INET6. It should therefore be possible to modify rcmd() to support either IPv4 or IPv6, based on the address family returned by gethostbyname().)

12. Future Items

Some additional items may require standardization, but no concrete proposals have been made for the API to perform these tasks. These may be addressed in a later document.

12.1. Flow Labels

Earlier revisions of this document specified a set of inet6_flow_XXX() functions to assign, share, and free IPv6 flow labels. Consensus, however, indicated that it was premature to specify this part of the API.

12.2. Path MTU Discovery and UDP

A standard method may be desirable for a UDP application to determine the "maximum send transport-message size" (Section 5.1 of [RFC-1981]) to a given destination. This would let the UDP application send smaller datagrams to the destination, avoiding fragmentation.

12.3. Neighbor Reachability and UDP

A standard method may be desirable for a UDP application to tell the kernel that it is making forward progress with a given peer (Section 7.3.1 of [RFC-1970]). This could save unneeded neighbor solicitations and neighbor advertisements.

13. Summary of New Definitions

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

<netinet/icmp6.h> ICMP6_DST_UNREACH
<netinet/icmp6.h> ICMP6_DST_UNREACH_ADDR
<netinet/icmp6.h> ICMP6_DST_UNREACH_ADMIN
<netinet/icmp6.h> ICMP6_DST_UNREACH_NOPORT
<netinet/icmp6.h> ICMP6_DST_UNREACH_NOROUTE
<netinet/icmp6.h> ICMP6_DST_UNREACH_NOTNEIGHBOR
<netinet/icmp6.h> ICMP6_ECHO_REPLY
<netinet/icmp6.h> ICMP6_ECHO_REQUEST
<netinet/icmp6.h> ICMP6_INFOMSG_MASK
<netinet/icmp6.h> ICMP6_MEMBERSHIP_QUERY
<netinet/icmp6.h> ICMP6_MEMBERSHIP_REDUCTION
<netinet/icmp6.h> ICMP6_MEMBERSHIP_REPORT
<netinet/icmp6.h> ICMP6_PACKET_TOO_BIG
<netinet/icmp6.h> ICMP6_PARAMPROB_HEADER
<netinet/icmp6.h> ICMP6_PARAMPROB_NEXTHEADER
<netinet/icmp6.h> ICMP6_PARAMPROB_OPTION
<netinet/icmp6.h> ICMP6_PARAM_PROB
<netinet/icmp6.h> ICMP6_TIME_EXCEEDED
<netinet/icmp6.h> ICMP6_TIME_EXCEED_REASSEMBLY
<netinet/icmp6.h> ICMP6_TIME_EXCEED_TRANSIT
<netinet/icmp6.h> ND_NA_FLAG_OVERRIDE
<netinet/icmp6.h> ND_NA_FLAG_ROUTER
<netinet/icmp6.h> ND_NA_FLAG_SOLICITED
<netinet/icmp6.h> ND_NEIGHBOR_ADVERT
<netinet/icmp6.h> ND_NEIGHBOR_SOLICIT
<netinet/icmp6.h> ND_OPT_MTU
<netinet/icmp6.h> ND_OPT_PI_FLAG_AUTO
<netinet/icmp6.h> ND_OPT_PI_FLAG_ONLINK
<netinet/icmp6.h> ND_OPT_PREFIX_INFORMATION

```
<netinet/icmp6.h> ND_OPT_REDIRECTED_HEADER
<netinet/icmp6.h> ND_OPT_SOURCE_LINKADDR
<netinet/icmp6.h> ND_OPT_TARGET_LINKADDR
<netinet/icmp6.h> ND_RA_FLAG_MANAGED
<netinet/icmp6.h> ND_RA_FLAG_OTHER
<netinet/icmp6.h> ND_REDIRECT
<netinet/icmp6.h> ND_ROUTER_ADVERT
<netinet/icmp6.h> ND_ROUTER_SOLICIT

<netinet/icmp6.h> struct icmp6_filter{};
<netinet/icmp6.h> struct icmp6_hdr{};
<netinet/icmp6.h> struct nd_neighbor_advert{};
<netinet/icmp6.h> struct nd_neighbor_solicit{};
<netinet/icmp6.h> struct nd_opt_hdr{};
<netinet/icmp6.h> struct nd_opt_mtu{};
<netinet/icmp6.h> struct nd_opt_prefix_info{};
<netinet/icmp6.h> struct nd_opt_rd_hdr{};
<netinet/icmp6.h> struct nd_redirect{};
<netinet/icmp6.h> struct nd_router_advert{};
<netinet/icmp6.h> struct nd_router_solicit{};

<netinet/in.h>    IPPROTO_AH
<netinet/in.h>    IPPROTO_DSTOPTS
<netinet/in.h>    IPPROTO_ESP
<netinet/in.h>    IPPROTO_FRAGMENT
<netinet/in.h>    IPPROTO_HOPOPTS
<netinet/in.h>    IPPROTO_ICMPV6
<netinet/in.h>    IPPROTO_IPV6
<netinet/in.h>    IPPROTO_NONE
<netinet/in.h>    IPPROTO_ROUTING
<netinet/in.h>    IPV6_DSTOPTS
<netinet/in.h>    IPV6_HOPLIMIT
<netinet/in.h>    IPV6_HOPOPTS
<netinet/in.h>    IPV6_NEXTHOP
<netinet/in.h>    IPV6_PKTINFO
<netinet/in.h>    IPV6_PKTOPTIONS
<netinet/in.h>    IPV6_RTHDR
<netinet/in.h>    IPV6_RTHDR_LOOSE
<netinet/in.h>    IPV6_RTHDR_STRICT
<netinet/in.h>    IPV6_RTHDR_TYPE_0
<netinet/in.h>    struct in6_pktinfo{};

<netinet/ip6.h>   IP6F_OFF_MASK
<netinet/ip6.h>   IP6F_RESERVED_MASK
<netinet/ip6.h>   IP6F_MORE_FRAG
<netinet/ip6.h>   struct ip6_dest{};
<netinet/ip6.h>   struct ip6_frag{};
<netinet/ip6.h>   struct ip6_hbh{};
```

```
     <netinet/ip6.h>   struct ip6_hdr{};
     <netinet/ip6.h>   struct ip6_rthdr{};
     <netinet/ip6.h>   struct ip6_rthdr0{};

     <sys/socket.h>    struct cmsghdr{};
     <sys/socket.h>    struct msghdr{};
```

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

```
<netinet/icmp6.h> void ICMP6_FILTER_SETBLOCK(int,
                                         struct icmp6_filter *);
<netinet/icmp6.h> void ICMP6_FILTER_SETBLOCKALL(struct icmp6_filter *);
<netinet/icmp6.h> void ICMP6_FILTER_SETPASS(int, struct icmp6_filter *);
<netinet/icmp6.h> void ICMP6_FILTER_SETPASSALL(struct icmp6_filter *);
<netinet/icmp6.h> int  ICMP6_FILTER_WILLBLOCK(int,
                                    const struct icmp6_filter *);
<netinet/icmp6.h> int  ICMP6_FILTER_WILLPASS(int,
                                    const struct icmp6_filter *);

<netinet/in.h>    int IN6_ARE_ADDR_EQUAL(const struct in6_addr *,
                                       const struct in6_addr *);

<netinet/in.h>    uint8_t *inet6_option_alloc(struct cmsghdr *,
                                             int, int, int);
<netinet/in.h>    int inet6_option_append(struct cmsghdr *,
                                         const uint8_t *, int, int);
<netinet/in.h>    int inet6_option_find(const struct cmsghdr *,
                                       uint8_t *, int);
<netinet/in.h>    int inet6_option_init(void *, struct cmsghdr **, int);
<netinet/in.h>    int inet6_option_next(const struct cmsghdr *,
                                       uint8_t **);
<netinet/in.h>    int inet6_option_space(int);

<netinet/in.h>    int inet6_rthdr_add(struct cmsghdr *,
                                     const struct in6_addr *,
                                     unsigned int);
<netinet/in.h>    struct in6_addr inet6_rthdr_getaddr(struct cmsghdr *,
                                                       int);
<netinet/in.h>    int inet6_rthdr_getflags(const struct cmsghdr *, int);
<netinet/in.h>    struct cmsghdr *inet6_rthdr_init(void *, int);
<netinet/in.h>    int inet6_rthdr_lasthop(struct cmsghdr *,
                                                   unsigned int);
<netinet/in.h>    int inet6_rthdr_reverse(const struct cmsghdr *,
                                         struct cmsghdr *);
<netinet/in.h>    int inet6_rthdr_segments(const struct cmsghdr *);
<netinet/in.h>    size_t inet6_rthdr_space(int, int);
```

```
<sys/socket.h>    unsigned char *CMSG_DATA(const struct cmsghdr *);
<sys/socket.h>    struct cmsghdr *CMSG_FIRSTHDR(const struct msghdr *);
<sys/socket.h>    unsigned int CMSG_LEN(unsigned int);
<sys/socket.h>    struct cmsghdr *CMSG_NXTHDR(const struct msghdr *mhdr,
                                              const struct cmsghdr *);
<sys/socket.h>    unsigned int CMSG_SPACE(unsigned int);

<unistd.h>        int rresvport_af(int *, int);
```

14. Security Considerations

The setting of certain Hop-by-Hop options and Destination options may be restricted to privileged processes. Similarly some Hop-by-Hop options and Destination options may not be returned to nonprivileged applications.

15. Change History

Changes from the June 1997 Edition (-03 draft)

- Added a note that defined constants for multibyte fields are in network byte order. This affects the ip6f_offlg member of the Fragment header (Section 2.1.2) and the nd_na_flags_reserved member of the nd_neighbor_advert structure (Section 2.2.2).

- Section 5: the ipi6_ifindex member of the in6_pktinfo structure should be "unsigned int" instead of "int", for consistency with the interface indexes in [RFC-2133].

- Section 6.3.7: the three calls to inet6_option_space() in the examples needed to be arguments to malloc(). The final one of these was missing the "6" in the name "inet6_option_space".

- Section 8.6: the function prototype for inet6_rthdr_segments() was missing the ending semicolon.

Changes from the March 1997 Edition (-02 draft)

- In May 1997 Draft 6.6 of Posix 1003.1g (called Posix.1g herein) passed ballot and will be forwarded to the IEEE Standards Board later in 1997 for final approval. Some changes made for this final Posix draft are incorporated into this Internet Draft, specifically the datatypes mentioned in Section 1 (and used throughout the text), and the socklen_t datatype used in Section 4.1 and 4.2.

- Section 1: Added the intN_t signed datatypes, changed the datatype u_intN_t to uintN_t (no underscore after the "u"), and removed the datatype u_intNm_t, as per Draft 6.6 of Posix.1g.

- Name space issues for structure and constant names in Section 2: Many of the structure member names and constant names were changed so that the prefixes are the same. The following prefixes are used for structure members: "ip6_", "icmp6_", and "nd_". All constants have the prefixes "ICMP6_" and "ND_".

- New definitions: Section 2.1.2: contains definitions for the IPv6 extension headers, other than AH and ESP. Section 2.2.2: contains additional structures and constants for the neighbor discovery option header and redirected header.

- Section 2.2.2: the enum for the neighbor discovery option field was changed to be a set of #define constants.

- Changed the word "function" to "macro" for references to all the uppercase names in Sections 2.3 (IN6_ARE_ADDR_EQUAL), 3.2 (ICMPV6_FILTER_xxx), and 4.3 (CMSG_xxx).

- Added more protocols to the /etc/protocols file (Section 2.4) and changed the name of "icmpv6" to "ipv6-icmp".

- Section 3: Made it more explicit that an application cannot read or write entire IPv6 packets, that all extension headers are passed as ancillary data. Added a sentence that the kernel fragments packets written to an IPv6 raw socket when necessary. Added a note that IPPROTO_RAW raw IPv6 sockets are not special.

- Section 3.1: Explicitly stated that the checksum option applies to both outgoing packets and received packets.

- Section 3.2: Changed the array name within the icmp6_filter structure from "data" to "icmp6_filt". Changes the prefix for the filter macros from "ICMPV6_" to "ICMP6_", for consistency with the names in Section 2.2. Changed the example from a ping program to a program that wants to receive only router advertisements.

- Section 4.1: Changed msg_namelen and msg_controllen from size_t to the Posix.1g socklen_t datatype. Updated the Note that follows.

- Section 4.2: Changed cmsg_len from size_t to the Posix.1g socklen_t datatype. Updated the Note that follows.

- Section 4.4: Added a Note that the second and third arguments to getsockopt() and setsockopt() are intentionally the same as the cmsg_level and cmsg_type members.

- Section 4.5: Reorganized the section into a description of the option, followed by the TCP semantics, and the UDP and raw socket semantics. Added a sentence on how to clear all the sticky options. Added a note that TCP need not save the options from the most recently received segment until the application says to do so. Added the statement that ancillary data is never passed with sendmsg() or recvmsg() on a TCP socket. Simplified the interaction of the sticky options with ancillary data for UDP or raw IP: none of the sticky options are sent if ancillary data is specified.

- Final paragraph of Section 5.1: ipi6_index should be ipi6_ifindex.

- Section 5.4: Added a note on the term "privileged".

- Section 5.5: Noted that the errors listed are examples, and the actual errors depend on the implementation.

- Removed Section 6 ("Flow Labels") as the consensus is that it is premature to try and specify an API for this feature. Access to the flow label field in the IPv6 header is still provided through the sin6_flowinfo member of the IPv6 socket address structure in [RFC-2133]. Added a subsection to Section 13 that this is a future item.

  All remaining changes are identified by their section number in the previous draft. With the removal of Section 6, the section numbers are decremented by one.

- Section 7.3.7: the calls to malloc() in all three examples should be calls to inet6_option_space() instead. The two calls to inet6_option_append() in the third example should be calls to inet6_option_alloc(). The two calls to CMSG_SPACE() in the first and third examples should be calls to CMSG_LEN(). The second call to CMSG_SPACE() in the second example should be a call to CMSG_LEN().

- Section 7.3.7: All the opt_X_ and opt_Y_ structure member names were changed to be ip6_X_opt_ and ip6_Y_opt_. The two structure names ipv6_opt_X and ipv6_opt_Y were changed to ip6_X_opt and ip6_Y_opt. The constants beginning with IPV6_OPT_X_ and IPV6_OPT_Y_ were changed to begin with IP6_X_OPT_ and IP6_Y_OPT_.

- Use the term "Routing header" throughout the draft, instead of "source routing". Changed the names of the eight inet6_srcrt_XXX() functions in Section 9 to inet6_rthdr_XXX(). Changed the name of the socket option from IPV6_SRCRT to IPV6_RTHDR, and the names of the three IPV6_SRCRT_xxx constants in Section 9 to IPV6_RTHDR_xxx.

- Added a paragraph to Section 9 on how to receive and send a Routing header.

- Changed inet6_rthdr_add() and inet6_rthdr_reverse() so that they return -1 upon an error, instead of an Exxx errno value.

- In the description of inet6_rthdr_space() in Section 9.1, added the qualifier "For an IPv6 Type 0 Routing header" to the restriction of between 1 and 23 segments.

- Refer to final function argument in Sections 9.7 and 9.8 as index, not offset.

- Updated Section 14 with new names from Section 2.

- Changed the References from "[n]" to "[RFC-abcd]".

Changes from the February 1997 Edition (-01 draft)

- Changed the name of the ip6hdr structure to ip6_hdr (Section 2.1) for consistency with the icmp6hdr structure. Also changed the name of the ip6hdrctl structure contained within the ip6_hdr structure to ip6_hdrctl (Section 2.1). Finally, changed the name of the icmp6hdr structure to icmp6_hdr (Section 2.2). All other occurrences of this structure name, within the Neighbor Discovery structures in Section 2.2.1, already contained the underscore.

- The "struct nd_router_solicit" and "struct nd_router_advert" should both begin with "nd6_". (Section 2.2.2).

- Changed the name of in6_are_addr_equal to IN6_ARE_ADDR_EQUAL (Section 2.3) for consistency with basic API address testing functions. The header defining this macro is <netinet/in.h>.

- getprotobyname("ipv6") now returns 41, not 0 (Section 2.4).

- The first occurrence of "struct icmpv6_filter" in Section 3.2 should be "struct icmp6_filter".

- Changed the name of the CMSG_LENGTH() macro to CMSG_LEN() (Section 4.3.5), since LEN is used throughout the <netinet/*.h>

headers.

- Corrected the argument name for the sample implementations of the CMSG_SPACE() and CMSG_LEN() macros to be "length" (Sections 4.3.4 and 4.3.5).

- Corrected the socket option mentioned in Section 5.1 to specify the interface for multicasting from IPV6_ADD_MEMBERSHIP to IPV6_MULTICAST_IF.

- There were numerous errors in the previous draft that specified <netinet/ip6.h> that should have been <netinet/in.h>. These have all been corrected and the locations of all definitions is now summarized in the new Section 14 ("Summary of New Definitions").

Changes from the October 1996 Edition (-00 draft)

- Numerous rationale added using the format (Note: ...).

- Added note that not all errors may be defined.

- Added note about ICMPv4, IGMPv4, and ARPv4 terminology.

- Changed the name of <netinet/ip6_icmp.h> to <netinet/icmp6.h>.

- Changed some names in Section 2.2.1: ICMPV6_PKT_TOOBIG to ICMPV6_PACKET_TOOBIG, ICMPV6_TIME_EXCEED to ICMPV6_TIME_EXCEEDED, ICMPV6_ECHORQST to ICMPV6_ECHOREQUEST, ICMPV6_ECHORPLY to ICMPV6_ECHOREPLY, ICMPV6_PARAMPROB_HDR to ICMPV6_PARAMPROB_HEADER, ICMPV6_PARAMPROB_NXT_HDR to ICMPV6_PARAMPROB_NEXTHEADER, and ICMPV6_PARAMPROB_OPTS to ICMPV6_PARAMPROB_OPTION.

- Prepend the prefix "icmp6_" to the three members of the icmp6_dataun union of the icmp6hdr structure (Section 2.2).

- Moved the neighbor discovery definitions into the <netinet/icmp6.h> header, instead of being in their own header (Section 2.2.1).

- Changed Section 2.3 ("Address Testing"). The basic macros are now in the basic API.

- Added the new Section 2.4 on "Protocols File".

- Added note to raw sockets description that something like BPF or DLPI must be used to read or write entire IPv6 packets.

- Corrected example of IPV6_CHECKSUM socket option (Section 3.1). Also defined value of -1 to disable.

- Noted that <netinet/icmp6.h> defines all the ICMPv6 filtering constants, macros, and structures (Section 3.2).

- Added note on magic number 10240 for amount of ancillary data (Section 4.1).

- Added possible padding to picture of ancillary data (Section 4.2).

- Defined <sys/socket.h> header for CMSG_xxx() functions (Section 4.2).

- Note that the data returned by getsockopt(IPV6_PKTOPTIONS) for a TCP socket is just from the optional headers, if present, of the most recently received segment. Also note that control information is never returned by recvmsg() for a TCP socket.

- Changed header for struct in6_pktinfo from <netinet.in.h> to <netinet/ip6.h> (Section 5).

- Removed the old Sections 5.1 and 5.2, because the interface identification functions went into the basic API.

- Redid Section 5 to support the hop limit field.

- New Section 5.4 ("Next Hop Address").

- New Section 6 ("Flow Labels").

- Changed all of Sections 7 and 8 dealing with Hop-by-Hop and Destination options. We now define a set of inet6_option_XXX() functions.

- Changed header for IPV6_SRCRT_xxx constants from <netinet.in.h> to <netinet/ip6.h> (Section 9).

- Add inet6_rthdr_lasthop() function, and fix errors in description of Routing header (Section 9).

- Reworded some of the Routing header descriptions to conform to the terminology in [RFC-1883].

- Added the example from [RFC-1883] for the Routing header (Section 9.9).

- Expanded the example in Section 10 to show multiple options per ancillary data object, and to show the receiver's ancillary data objects.

- New Section 11 ("IPv6-Specific Options with IPv4-Mapped IPv6 Addresses").

- New Section 12 ("rresvport_af").

- Redid old Section 10 ("Additional Items") into new Section 13 ("Future Items").

16. References

[RFC-1883] Deering, S., and R. Hinden, "Internet Protocol, Version 6 (IPv6), Specification", RFC 1883, December 1995.

[RFC-2133] Gilligan, R., Thomson, S., Bound, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 2133, April 1997.

[RFC-1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, August 1996.

[RFC-1970] Narten, T., Nordmark, E., and W. Simpson, "Neighbor Discovery for IP Version 6 (IPv6)", RFC 1970, August 1996.

17. Acknowledgments

Matt Thomas and Jim Bound have been working on the technical details in this draft for over a year. Keith Sklower is the original implementor of ancillary data in the BSD networking code. Craig Metz provided lots of feedback, suggestions, and comments based on his implementing many of these features as the document was being written.

The following provided comments on earlier drafts: Pascal Anelli, Hamid Asayesh, Ran Atkinson, Karl Auerbach, Hamid Asayesh, Matt Crawford, Sam T. Denton, Richard Draves, Francis Dupont, Bob Gilligan, Tim Hartrick, Masaki Hirabaru, Yoshinobu Inoue, Mukesh Kacker, A. N. Kuznetsov, Pedro Marques, Jack McCann, der Mouse, John Moy, Thomas Narten, Erik Nordmark, Steve Parker, Charles Perkins, Tom Pusateri, Pedro Roque, Sameer Shah, Peter Sjodin, Stephen P. Spackman, Jinmei Tatuya, Karen Tracey, Quaizar Vohra, Carl Williams, Steve Wise, and Kazu Yamamoto.

18. Authors' Addresses

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

EMail: rstevens@kohala.com

Matt Thomas
AltaVista Internet Software
LJO2-1/J8
30 Porter Rd
Littleton, MA 01460
EMail: matt.thomas@altavista-software.com

19. Full Copyright Statement