INTERNET-DRAFT
Expires: December 24, 1999
Obsoletes RFC 2292

W. Richard Stevens (Consultant)
Matt Thomas (Consultant)
Erik Nordmark (Sun)

Oct 22, 1999

Advanced Sockets API for IPv6

<draft-ietf-ipngwg-rfc2292bis-01.txt>

Abstract

A separate specification [RFC-2553] contain changes to the sockets API to support IP version 6. Those 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-2553]: The Routing header (source routing), Hop-by-Hop options, and Destination options. This document provides API access to these features too.

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any

time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet Draft expires April 22, 2000.

Table of Contents

1. Introduction

A separate specification [RFC-2553] contain changes to the sockets API to support IP version 6. Those changes are for TCP and UDP-based applications. This 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 header files. 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-2553]: sending and receiving Routing headers, Hop-by-Hop options, and Destination options, 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 Routing header, Hop-by-Hop, and

Destination extension headers.

5. Additional features required for improved IPv6 application portability.

The packet information along with access to the extension headers (Routing header, Hop-by-Hop options, and Destination options) 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 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 protocol headers are required, along with common constant definitions for the structure members.

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().

Two new header files 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; /* 4 bits version, 8 bits TC, 24 bits 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, top 4 bits tclass */
      } 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 */
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 */
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 */
  uint32_t ip6r0_reserved;  /* reserved field */
    /* followed by up to 127 struct in6_addr */
};

/* 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
```

2.1.3. IPv6 Options

Eleven options are defined for IPv6 at the time of writing this document. We define structures for all except the unspecified EID option. The following structures are defined as a result of including <netinet/ip6.h>.

```
/* IPv6 options */
structip6_opt {
    uint8_tip6o_type;
    uint8_tip6o_len;
};

/*
 * The high-order 3 bits of the option type define the behavior
 * when processing an unknown option and whether or not the option
 * content changes in flight.
 */
#defineIP6OPT_TYPE(o)((o) & 0xc0)
#defineIP6OPT_TYPE_SKIP0x00
#defineIP6OPT_TYPE_DISCARD0x40
#defineIP6OPT_TYPE_FORCEICMP0x80
#defineIP6OPT_TYPE_ICMP0xc0
#defineIP6OPT_MUTABLE0x20

#defineIP6OPT_PAD10x00/* 00 0 00000 */
#defineIP6OPT_PADN0x01/* 00 0 00001 */
#defineIP6OPT_JUMBO0xc2/* 11 0 00010 = 194 */
#defineIP6OPT_NSAP_ADDR0xc3/* 11 0 00011 */
#defineIP6OPT_TUNNEL_LIMIT0x04/* 00 0 00100 */
#defineIP6OPT_ROUTER_ALERT0x05/* 00 0 00101 */
```

```
#defineIP6OPT_BINDING_UPDATE0xc6/* 11 0 00110 */
#defineIP6OPT_BINDING_ACK0x07/* 00 0 00111 */
#defineIP6OPT_BINDING_REQ0x08/* 00 0 01000 */
#defineIP6OPT_HOME_ADDRESS0xc9/* 11 0 01001 */
#defineIP6OPT_EID0x8a/* 10 0 01010 */

/* Jumbo Payload Option */
structip6_opt_jumbo {
    uint8_tip6oj_type;
    uint8_tip6oj_len;
    uint8_t ip6oj_jumbo_len[4];
};
#defineIP6OPT_JUMBO_LEN6

/* NSAP Address Option */
structip6_opt_nsap {
    uint8_tip6on_type;
    uint8_tip6on_len;
    uint8_t ip6on_src_nsap_len;
    uint8_t ip6on_dst_nsap_len;
      /* followed by source NSAP */
      /* followed by destination NSAP */
};

/* Tunnel Limit Option */
structip6_opt_tunnel {
    uint8_tip6ot_type;
    uint8_tip6ot_len;
    uint8_t ip6ot_encap_limit;
};

/* Router Alert Option */
structip6_opt_router {
    uint8_tip6or_type;
    uint8_tip6or_len;
    uint8_t ip6or_value[2];
};

/* Router alert values (in network byte order) */
#ifdef _BIG_ENDIAN
#defineIP6_ALERT_MLD0x0000
#defineIP6_ALERT_RSVP0x0001
#defineIP6_ALERT_AN0x0002
#else
#defineIP6_ALERT_MLD0x0000
#defineIP6_ALERT_RSVP0x0100
#defineIP6_ALERT_AN0x0200
#endif
```

```
/* Binding Update Option */
structip6_opt_binding_update {
    uint8_tip6ou_type;
    uint8_tip6ou_len;
    uint8_t ip6ou_flags;
    uint8_t ip6ou_prefixlen;
    uint8_t ip6ou_seqno[2];
    uint8_t ip6ou_lifetime[4];
    uint8_t ip6ou_coa[16];/* Optional based on flags */
      /* followed by sub-options */
};

/* Binding Update Flags */
#defineIP6_BUF_ACK0x80/* Request a binding ack */
#defineIP6_BUF_HOME0x40/* Home Registration */
#defineIP6_BUF_COA0x20/* Care-of-address present in option */
#defineIP6_BUF_ROUTER0x10/* Sending mobile node is a router */

/* Binding Ack Option */
structip6_opt_binding_ack {
    uint8_tip6oa_type;
    uint8_tip6oa_len;
    uint8_t ip6oa_status;
    uint8_t ip6oa_seqno[2];
    uint8_t ip6oa_lifetime[4];
    uint8_t ip6oa_refresh[4];
      /* followed by sub-options */
};

/* Binding Request Option */
structip6_opt_binding_request {
    uint8_tip6or_type;
    uint8_tip6or_len;
      /* followed by sub-options */
};

/* Home Address Option */
structip6_opt_home_address {
    uint8_tip6oh_type;
    uint8_tip6oh_len;
    uint8_t ip6oh_addr[16];/* Home Address */
      /* followed by sub-options */
};
```

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 */
                                               /* admin. prohibited */
       #define ICMP6_DST_UNREACH_BEYONDSCOPE 2 /* beyond scope of source
address */
       #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.

2.2.3. Multicast Listener Discovery Type and Code Values

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

```
struct mld_hdr {
  struct icmp6_hdr  mld_hdr;
  struct in6_addr   mld_addr; /* multicast address */
};
#define mld_type                mld_hdr.icmp6_type
#define mld_code                mld_hdr.icmp6_code
#define mld_cksum               mld_hdr.icmp6_cksum
#define mld_maxdelay            mld_hdr.icmp6_data16[0]
#define mld_reserved            mld_hdr.icmp6_data16[1]
```

2.2.4. ICMPv6 Router Renumbering Type and Code Values

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

```
struct icmp6_router_renum {  /* router renumbering header */
  struct icmp6_hdr  rr_hdr;
  u_int8_t          rr_segnum;
  u_int8_t          rr_flags;
  u_int16_t         rr_maxdelay;
  u_int32_t         rr_reserved;
};
#define rr_type                 rr_hdr.icmp6_type
#define rr_code                 rr_hdr.icmp6_code
#define rr_cksum                rr_hdr.icmp6_cksum
#define rr_seqnum               rr_hdr.icmp6_data32[0]

/* Router renumbering flags */
#define ICMP6_RR_FLAGS_TEST        0x80
#define ICMP6_RR_FLAGS_REQRESULT   0x40
#define ICMP6_RR_FLAGS_FORCEAPPLY  0x20
#define ICMP6_RR_FLAGS_SPECSITE    0x10
#define ICMP6_RR_FLAGS_PREVDONE    0x08


struct rr_pco_match {    /* match prefix part */
  u_int8_t          rpm_code;
  u_int8_t          rpm_len;
  u_int8_t          rpm_ordinal;
  u_int8_t          rpm_matchlen;
  u_int8_t          rpm_minlen;
  u_int8_t          rpm_maxlen;
  u_int16_t         rpm_reserved;
```

```
  struct in6_addr   rpm_prefix;
};

/* PCI code values */
#define RPM_PCO_ADD             1
#define RPM_PCO_CHANGE          2
#define RPM_PCO_SETGLOBAL       3

struct rr_pco_use {    /* use prefix part */
  u_int8_t          rpu_uselen;
  u_int8_t          rpu_keeplen;
  u_int8_t          rpu_ramask;
  u_int8_t          rpu_raflags;
  u_int32_t         rpu_vltime;
  u_int32_t         rpu_pltime;
  u_int32_t         rpu_flags;
  struct in6_addr   rpu_prefix;
};
#define ICMP6_RR_PCOUSE_RAFLAGS_ONLINK   0x20
#define ICMP6_RR_PCOUSE_RAFLAGS_AUTO     0x10

#if BYTE_ORDER == BIG_ENDIAN
#define ICMP6_RR_PCOUSE_FLAGS_DECRVLTIME 0x80000000
#define ICMP6_RR_PCOUSE_FLAGS_DECRPLTIME 0x40000000
#elif BYTE_ORDER == LITTLE_ENDIAN
#define ICMP6_RR_PCOUSE_FLAGS_DECRVLTIME 0x80
#define ICMP6_RR_PCOUSE_FLAGS_DECRPLTIME 0x40
#endif

struct rr_result {    /* router renumbering result message */
  u_int16_t         rrr_flags;
  u_int8_t          rrr_ordinal;
  u_int8_t          rrr_matchedlen;
  u_int32_t         rrr_ifid;
  struct in6_addr   rrr_prefix;
};

#if BYTE_ORDER == BIG_ENDIAN
#define ICMP6_RR_RESULT_FLAGS_OOB        0x0002
#define ICMP6_RR_RESULT_FLAGS_FORBIDDEN  0x0001
#elif BYTE_ORDER == LITTLE_ENDIAN
#define ICMP6_RR_RESULT_FLAGS_OOB        0x0200
#define ICMP6_RR_RESULT_FLAGS_FORBIDDEN  0x0100
#endif
```

2.3. Address Testing Macros

The basic API ([RFC-2553]) 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 *);
```

This macro returns non-zero if the addresses are equal; otherwise it returns zero.

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(PF_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 used the host's byte order for certain IP header fields.

Another difference from IPv4 raw sockets is that complete packets (that is, IPv6 packets with extension headers) cannot be sent or received using the IPv6 raw sockets API. Instead, ancillary data objects are used to transfer the extension headers and hoplimit information, as described in Section 6. 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 and on receipt the application will only receive the payload i.e. the data after the IPv6 header and all the extension headers.

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, to have it mean that the application will send down complete packets including the IPv4 header. (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.) 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).

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-2460]. 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.

An attempt to set IPV6_CHECKSUM for an ICMPv6 socket will fail.

(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 multicast listener discovery 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>.

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. Access to IPv6 and Extension Headers

Applications need to be able to control IPv6 header and extension header content when sending as well as being able to receive the content of these headers. This is done by defining socket option types which can be used both with setsockopt and with ancillary data. Ancillary data is discussed in Appendix A. The following optional information can be exchanged between the application and the kernel:

1. The send/receive interface and source/destination address,
2. The hop limit,
3. Next hop address,
4. Routing header.
5. Hop-by-Hop options, and
6. Destination options (both before and after a 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_RECVPKTINFO,  &on, sizeof(on));
    setsockopt(fd, IPPROTO_IPV6, IPV6_RECVHOPLIMIT, &on, sizeof(on));
    setsockopt(fd, IPPROTO_IPV6, IPV6_RECVRTHDR,    &on, sizeof(on));
    setsockopt(fd, IPPROTO_IPV6, IPV6_RECVHOPOPTS,  &on, sizeof(on));
    setsockopt(fd, IPPROTO_IPV6, IPV6_RECVDSTOPTS,  &on, sizeof(on));
    setsockopt(fd, IPPROTO_IPV6, IPV6_RECVRTHDRDSTOPTS,
               &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.

Two different mechanisms exist for sending this optional information:

1. Using setsockopt to specify the option content for a socket. These are known an "sticky" options since they affect all transmitted packets on the socket until either the a new setsockopt is done or the options are overridden using ancillary data.

2. Using ancillary data to specify the option content for a single datagram. This only applies to datagram and raw sockets; not to TCP sockets.

The three socket option parameters and the three cmsghdr fields that describe the options/ancillary data objects are summarized as:

opt level/ cmsg_level	optname/ cmsg_type	optval/ cmsg_data[]
IPPROTO_IPV6	IPV6_PKTINFO	in6_pktinfo structure
IPPROTO_IPV6	IPV6_HOPLIMIT	int
IPPROTO_IPV6	IPV6_NEXTHOP	socket address structure
IPPROTO_IPV6	IPV6_RTHDR	ip6_rthdr structure
IPPROTO_IPV6	IPV6_HOPOPTS	ip6_hbh structure
IPPROTO_IPV6	IPV6_DSTOPTS	ip6_dest structure
IPPROTO_IPV6	IPV6_RTHDRDSTOPTS	ip6_dest structure

All these options are described in detail in Section 6, 7, 8 and 9. All the constants beginning with IPV6_ are defined as a result of including the <netinet/in.h>.

Issuing getsockopt() for the above options will return the sticky

option value i.e. the value set with setsockopt().

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").

The application does not explicitly need to access the data structures for the Routing header option, Hop-by-Hop option, and Destination options, since the API to these features is through a set of inet6_rth_XXX() and inet6_opt_XXX() functions that we define in Section 8 and Section 10. Those functions simplify the interface to these features instead of requiring the application to know the intimate details of the extension header formats.

4.1. TCP Implications

It is not possible to use ancillary data to transmit the above options for TCP since there is not a one-to-one mapping between send operations and the TCP segments being transmitted. Instead an application can use setsockopt to specify them as sticky options. When the application uses setsockopt to specify the above options it is expected that TCP will start using the new information when sending segments. However, TCP may or may not use the new information when retransmitting segments that were originally sent when the old sticky options were in effect.

Applications using TCP can use ancillary data (after enabling the desired IPV6_RECVxxx options) to receive the IPv6 and extension header information. However, since there is not a one-to-one mapping between received TCP segments and recv operations seen by the application, when different TCP segments have different IPv6 and extension headers the application might not be able to observe all received headers. For efficiency reasons it is recommended that a TCP implementation not send ancillary data items with every received segment but instead try to detect the points in the data stream when the requested IPv6 and extension header content changes and only send a single ancillary data item at the time of the change. Also, TCP should send ancillary data items at the start of the connection and when the application enables a new IPV6_RECVxxx option.

For example, assume an application has enabled IPV6_RECVHOPLIMIT before a connection is established. Then the first recvmsg() would have an IPV6_HOPLIMIT item indicating the hop limit in the first data segment. Should the hoplimit in the received data segment change a subsequent recvmsg() will also have an IPV6_HOPLIMIT item. However,

the application must be prepared to handle ancillary data items even though the hop limit did not change. Note that should the hop limit in received ACK-only segments be different than the hop limit in data segments the application might only be able to observe the hop limit in the received data segments.

The above example was for hop limit but the application should be prepared to handle the corresponding behavior for the other option information.

The above recvmsg() behavior allows the application to detect changes in the received IPv6 and extension headers without resorting to periodic getsockopt() calls.

4.2. UDP and Raw Socket Implications

The receive behavior for UDP and raw sockets is quite straightforward. After the application has enabled an IPV6_RECVxxx socket option it will receive ancillary data items for every recvmsg() call containing the requested information. However, if the information is not present in the packet the ancillary data item will not be included. For example, if the application enables IPV6_RECVRTHDR and a received datagram does not contain a Routing header there will not be an IPV6_RTHDR ancillary data item. Note that due to buffering in the socket implementation there might be some packets queued when an IPV6_RECVxxx option is enabled and they might not have the ancillary data information.

For sending the application has the choice between using sticky options and ancillary data. The application can also use both having the sticky options specify the "default" and using ancillary data to override the default options. Note that if any ancillary data is specified in a call to sendmsg(), all of the sticky options are overridden for that datagram. For example, if the application has set IPV6_RTHDR using a sticky option and later passes IPV6_HOPLIMIT as ancillary data this will override the IPV6_RTHDR sticky option and no Routing header will be sent with that datagram.

5. Extensions to Socket Ancillary Data

This specification uses ancillary data as defined in Posix.1g which the following compatible extensions:

- CMSG_NEXTHDR has been extended to handle a NULL 2nd argument to mean "get the first header". See Section 22.3.2.

- A new CMSG_SPACE macro is defined. It is used to determine how much space need to be allocated for an ancillary data item. See Section 22.3.4.

- A new CMSG_LEN macro is defined. It returns the value to store in the cmsg_len member of the cmsghdr structure, taking into account any padding needed to satisfy alignment requirements. See Section 22.3.5.

6. 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 set with setsockopt() or 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>.

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

In the socket option and cmsghdr level will be IPPROTO_IPV6, the type will be IPV6_PKTINFO, and the first byte of the option value and cmsg_data[] will be the first byte of the in6_pktinfo structure. An application can clear any sticky IPV6_PKTINFO option by either doing a setsockopt for option with optlen being zero, or by doing a "regular" setsockopt with ipi6_addr being in6addr_any and ipi6_ifindex being zero.

This information is returned as ancillary data by recvmsg() only if the application has enabled the IPV6_RECVPKTINFO socket option:

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_RECVPKTINFO, &on, sizeof(on));
```

(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_RECVPKTINFO 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.)

6.1. Specifying/Receiving the Interface

Interfaces on an IPv6 node are identified by a small positive integer, as described in Section 4 of [RFC-2553]. 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-2553]), 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.

6.2. 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 or in6addr_any), 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.

6.3. 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-2553]. 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_RECVHOPLIMIT socket option:

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_RECVHOPLIMIT, &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() or using setsockopt(). As specified in [RFC-2553], 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
```

6.4. 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.

6.5. Additional Errors with sendmsg() and setsockopt()

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() or setsockopt(). Note that some implementations might only be able to return this type of errors for setsockopt(). 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.

7. 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-2460]. This type supports up to 127 intermediate nodes (limited by the length field in the extension header). With this maximum number of intermediate nodes, a source, and a destination, there are 128 hops.

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, and the ability to use sticky options or ancillary data to communicate this information between the application and the kernel using the IPV6_RTHDR option.

Three functions build a Routing header:

```
inet6_rth_space()    - return #bytes required for Routing header
inet6_rth_init()     - initialize buffer data for Routing header
inet6_rth_add()      - add one IPv6 address to the Routing header
```

Three functions deal with a returned Routing header:

```
inet6_rth_reverse()  - reverse a Routing header
inet6_rth_segments() - return #segments in a Routing header
inet6_rth_getaddr()  - fetch one address from a Routing header
```

The function prototypes for these functions are defined as a result of including the <netinet/in.h>.

To receive a Routing header the application must enable the IPV6_RECVRTHDR socket option:

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_RECVRTHDR, &on, sizeof(on));
```

To send a Routing header the application specifies it either as ancillary data in a call to sendmsg() or using setsockopt().

The application can remove any sticky Routing header by calling

setsockopt() for IPV6_RTHDR with a zero option length.

When using ancillary data a Routing header is passed between the application and the kernel as follows: 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 six functions that we are about to describe.

The following constant is defined as a result of including the <netinet/in.h>:

```
#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.

7.1. inet6_rth_space

```
size_t inet6_rth_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 0 and 127, inclusive. The return value is just the space for the Routing header. When the application uses ancillary data it must pass the returned length to CMSG_LEN() to determine how much memory is needed for the ancillary data object (including the cmsghdr structure).

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.)

7.2. inet6_rth_init

```
void *inet6_rth_init(void *bp, int bp_len, int type, int segments);
```

This function initializes the buffer pointed to by bp to contain a Routing header of the specified type and sets ip6r0_len based on the segments parameter. bp_len is only used to verify that the buffer is large enough. The ip6r0_segleft field is set to zero; inet6_rth_add() will increment it.

When the application uses ancillary data the application must initialize any cmsghdr fields.

The caller must allocate the buffer and its size can be determined by calling inet6_rth_space().

Upon success the return value is the pointer to the buffer (bp), and this is then used as the first argument to the next two functions. Upon an error the return value is NULL.

7.3. inet6_rth_add

```
int inet6_rth_add(void *bp, const struct in6_addr *addr);
```

This function adds the IPv6 address pointed to by addr to the end of the Routing header being constructed.

If successful, the segleft member of the Routing Header 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.

7.4. inet6_rth_reverse

```
int inet6_rth_reverse(const void *in, void *out)
```

This function takes a Routing header extension header (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.

7.5. inet6_rth_segments

```
int inet6_rth_segments(const void *bp);
```

This function returns the number of segments (addresses) contained in the Routing header described by bp. On success the return value is zero or greater. The return value of the function is -1 upon an error.

7.6. inet6_rth_getaddr

```
struct in6_addr *inet6_rth_getaddr(const void *bp, int index);
```

This function returns a pointer to the IPv6 address specified by index (which must have a value between 0 and one less than the value returned by inet6_rth_segments()) in the Routing header described by bp. An application should first call inet6_rth_segments() to obtain the number of segments in the Routing header.

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

8. 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 four Hop-by-Hop options are defined for IPv6 [RFC-2460]: Jumbo Payload, Pad1, PadN, and router-alert. The two pad options are for alignment purposes and are automatically inserted by the inet6_opt_XXX() routines and ignored by the inet6_opt_XXX() routines on the receive side.

This section of the API is therefore defined for future Hop-by-Hop options (as well as destination options) that an application may need to specify and receive. This IPv6 API defines seven functions that the application calls to build and examine a Hop-by_Hop options header, and the ability to use sticky options or ancillary data to communicate this information between the application and the kernel. This uses the IPV6_HOPOPTS for hob-by-hop options.

Four functions build a options header:

```
  inet6_opt_init()     - initialize buffer data for options header
  inet6_opt_append()   - add one TLV option to the options header
  inet6_opt_finish()   - finish adding TLV options to the options header
  inet6_opt_set_val()  - add one component of the option content to the
option
```

Three functions deal with a returned options header:

```
  inet6_opt_next()     - extract the next option from the options header
  inet6_opt_find()     - extract an option of a specified type from the
header
  inet6_opt_get_val()  - retrieve one component of the option content
```

Individual Hop-by-Hop options (and Destination options, which are described in Section 9 and are very 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 B of [RFC-2460]. The alignment of first byte 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. The functions below need to know the alignment of the end of the option (which is always in the form "xn," where x can have the values 1, 2, 4, or 8) and the total size of the data portion of the option. These are passed as the "align" and "len" arguments to inet6_opt_append().

Multiple Hop-by-Hop options must be specified by the application by placing them in a single extension header.

Finally, we note that use of some Hop-by-Hop options or 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.

8.1. Receiving Hop-by-Hop Options

To receive Hop-by-Hop options the application must enable the IPV6_RECVHOPOPTS socket option:

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_RECVHOPOPTS, &on, sizeof(on));
```

When using ancillary data a Hop-by-hop options is passed between the application and the kernel as follows: 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_opt_next(), inet6_opt_find(), and inet6_opt_get_val() functions, described in Section 10.6.

8.2. Sending Hop-by-Hop Options

To send a Hop-by-Hop options header, the application specifies the header either as ancillary data in a call to sendmsg() or using setsockopt().

The application can remove any sticky Hop-by-Hop extension header by calling setsockopt() for IPV6_HOPOPTS with a zero option length.

All the Hop-by-Hop options must specified by a single ancillary data object. 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_opt_init(), inet6_opt_append(), inet6_opt_finish(), and inet6_set_val() functions, described in Section 10.

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

9. Destination Options

A variable number of Destination options can appear in one or more Destination option headers. As defined in [RFC-2460], 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-2460], although proposals exist to use Destination options with Mobile IPv6.

9.1. Receiving Destination Options

To receive Destination options appearing after a Routing header (or in a packet without a Routing header) the application must enable the IPV6_RECVDSTOPTS socket option:

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_RECVDSTOPTS, &on, sizeof(on));
```

To receive Destination options appearing before a Routing header the application must enable the IPV6_RECVRTHDRDSTOPTS socket option:

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_RECVRTHDRDSTOPTS,
           &on, sizeof(on));
```

All the Destination options appearing before a Routing header are returned as one ancillary data object described by a cmsghdr structure (with cmsg_type set to IPV6_RTHDRDSTOPTS) and all the Destination options appearing after a Routing header (or in a packet without a Routing header) are returned as another ancillary data object described by a cmsghdr structure (with cmsg_type set to IPV6_DSTOPTS). For all these ancillary data objects, the cmsg_level member will be IPPROTO_IPV6.

These options are then processed by calling the inet6_opt_next(), inet6_opt_find(), and inet6_opt_get_value() functions.

9.2. Sending Destination Options

To send a Destination options header, the application specifies it either as ancillary data in a call to sendmsg() or using setsockopt().

The application can remove any sticky Destination extension header by calling setsockopt() for IPV6_RTHDRDSTOPTS/IPV6_DSTOPTS with a zero option length.

As described in Section 6 one set of Destination options can appear before a Routing header, and one set can appear after a Routing header (or in a packet with no Routing header). Each set can consist of one or more options but each set is a single extension header.

When using ancillary data a Destination options header is passed between the application and the kernel as follows: The set preceding a Routing header are specified with the cmsg_level member is set to IPPROTO_IPV6 and the cmsg_type member is set to IPV6_RTHDRDSTOPTS. Any setsockopt or ancillary data for IPV6_RTHDRDSTOPTS is silently ignore when sending packets unless a Routing header is also

specified.

The set of Destination options after a Routing header, which are also used when no Routing header is present, are specified with the cmsg_level member is set to IPPROTO_IPV6 and the cmsg_type member is set to IPV6_DSTOPTS.

The Destination options are normally constructed using the inet6_opt_init(), inet6_opt_append(), inet6_opt_finish(), and inet6_set_val() functions, described in Section 10.

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

10. 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. These functions assume the formatting rules specified in Appendix B in [RFC-2460] i.e. that the largest field is placed last in the option.

The function prototypes for these functions are defined as a result of including the <netinet/in.h>.

The first 3 functions (init, append, and finish) are used both to calculate the needed buffer size for the options, and to actually encode the options once the application has allocated a buffer for the header. In order to only calculate the size the application must pass a NULL extbuf and a zero extlen to those functions.

10.1. inet6_opt_init

```
int inet6_opt_init(void *extbuf, size_t extlen);
```

This function returns the number of bytes needed for the empty extension header i.e. without any options. If extbuf is not NULL it also initializes the extension header to have the correct length field. If the extlen value is not a positive (i.e., non-zero) multiple of 8 the function fails and returns -1.

10.2. inet6_opt_append

```
int inet6_opt_append(void *extbuf, size_t extlen, int prevlen,
                     uint8_t type, size_t len, uint_t align,
                     void **databufp);
```

Prevlen should be the length returned by inet6_opt_init() or a previous inet6_opt_append(). This function returns the updated total length taking into account adding an option with length 'len' and alignment 'align'. If extbuf is not NULL then, in addition to returning the length, the function inserts any needed pad option, initializes the option (setting the type and length fields) and returns a pointer to the location for the option content in databufp. If the option does not fit in the extension header buffer the function returns -1.

type is the 8-bit option type. len is the length of the option data (i.e. excluding the option type and option length fields).

Once inet6_opt_append() has been called the application can use the databuf directly, or use inet6_opt_set_val() to specify the content of the option.

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.

The align parameter must have a value of 1, 2, 4, or 8. The align value can not exceed the value of len.

10.3. inet6_opt_finish

```
int inet6_opt_finish(void *extbuf, size_t extlen, int prevlen);
```

Prevlen should be the length returned by inet6_opt_init() or inet6_opt_append(). This function returns the updated total length taking into account the final padding of the extension header to make it a multiple of 8 bytes. If extbuf is not NULL the function also initializes the option by inserting a Pad1 or PadN option of the proper length.

If the necessary pad does not fit in the extension header buffer the

function returns -1.

10.4. inet6_opt_set_val

```
int inet6_opt_set_val(void *databuf, size_t offset, void *val,
                      int vallen);
```

Databuf should be a pointer returned by inet6_opt_append(). This function inserts data items of various sizes (1, 2, 4, or 8 bytes) in the data portion of the option. val should point to the data to be inserted. Offset specifies where in the data portion of the option the value should be inserted; the first byte after the option type and length is accessed by specifying an offset of zero.

The function returns the offset for the next field (i.e., offset + vallen) which can be used when composing option content with multiple fields.

10.5. inet6_opt_next

```
int inet6_opt_next(void *extbuf, size_t extlen, int prevlen,
                   uint8_t *typep, size_t *lenp,
                   void **databufp);
```

This function parses received extension headers returning the next option. Extbuf and extlen specifies the extension header. Prevlen should either be zero (for the first option) or the length returned by a previous call to inet6_opt_next() or inet6_opt_find(). It specifies the position where to continue scanning the extension buffer. The next option is returned by updating typep, lenp, and databufp. This function returns the updated "previous" length computed by advancing past the option that was returned. This returned "previous" length can then be passed to subsequent calls to inet6_opt_next(). This function does not return any PAD1 or PADN options. When there are no more options the return value is -1.

10.6. inet6_opt_find

```
int inet6_opt_find(void *extbuf, size_t extlen, int prevlen,
                   uint8_t type, size_t *lenp,
                   void **databufp);
```

This function is similar to the previously described inet6_opt_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 extension header.

If an option of the specified type is located, the function returns the updated "previous" total length computed by advancing past the option that was returned and past any options that didn't match the type. This returned "previous" length can then be passed to subsequent calls to inet6_opt_find() for finding the next occurance of the same option type.

If an option of the specified type is not located, the return value is -1. If an error occurs, the return value is -1.

10.7. inet6_opt_get_val

```
int inet6_opt_get_val(void *databuf, size_t offset, void *val,
                      int vallen);
```

Databuf should be a pointer returned by inet6_opt_next() or inet6_opt_find(). This function extracts data items of various sizes (1, 2, 4, or 8 bytes) in the data portion of the option. val should point to the destination for the extracted data. Offset specifies from where in the data portion of the option the value should be extracted; the first byte after the option type and length is accessed by specifying an offset of zero.

The function returns the offset for the next field (i.e., offset + vallen) which can be used when extracting option content with multiple fields. XXX Perhaps we should add a note to point out that robust receivers should verify alignment before calling inet6_opt_get_val(). XXX Or check alignment and fail by returning -1?

11. Additional Advanced API Functions

11.1. Sending with the Minimum MTU

Some applications might not want to incur the overhead of path MTU discovery, especially if the applications only send a single datagram to a destination. A potential example is a DNS server.

This specification defines a mechanism to avoid fragmentation by sending at the minimum IPv6 MTU (1280 bytes). This can be enabled using the IPV6_USE_MIN_MTU socket option.

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &on, sizeof(on));
```

By default, this socket option is disabled. Setting the value to 0 also disables the option. This option can also be sent as ancillary data.

11.2. Path MTU Discovery and UDP

UDP and raw socket applications need to be able to determine the "maximum send transport-message size" (Section 5.1 of [RFC-1981]) to a given destination so that those applications can participate in path MTU discovery. This lets those applications send smaller datagrams to the destination, avoiding fragmentation.

This is accomplished using a new ancillary data item (IPV6_PATHMTU) which is delivered to recvmsg() without any actual data. The application enable the receipt of IPV6_PATHMTU ancillary data items by enabing IPV6_RECVPATHMTU.

```
int  on = 1;
setsockopt(fd, IPPROTO_IPV6, IPV6_RECVPATHMTU, &on, sizeof(on));
```

By default, this socket option is disabled. Setting the value to 0 also disables the option.

When the application is sending packets too big for the path MTU recvmsg will return zero (indicating no data) but there will be a cmsghdr with cmsg_type set to IPV6_PATHMTU, and cmsg_len will indicate that cmsg_data is 4 bytes long. CMSG_DATA will point to an integer carrying the path MTU to use.

11.3. Neighbor Reachability and UDP

UDP and raw socket application might know that communication is making forward progress i.e. that the path from the node to the next hop is working. In such a case the applications, similarly to TCP as specified in [RFC-2461], has the option indicate to the internet layer that the neighbor is reachable. See section 7.3.1 of [RFC-2461]. This could save unneeded neighbor solicitation and neighbor advertisement messages.

This is done by including an ancilary data item with cmsg_type being IPV6_REACHCONF and with no attached CMSG_DATA.

12. 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(): the Routing header, Hop-by-Hop options, and Destination options. When multiple ancillary data objects are transferred via 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 for using these ancillary data objects with sendmsg().

All Hop-by-Hop options must be specified in a single ancillary data object. Should multiple hop-by-hop ancillary data objects be specified the implementation might choose an arbitrary one or drop the packet.

All Destination options that precede a Routing header must be specified in a single ancillary data object. If there is no Routing header ancillary data object the IPV6_RTHDRDSTOPTS object will be silently ignored.

All Destination options that follow a Routing header (or are used without a Routing header) must be specified in a single ancillary data object.

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.

13. 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.

14. Extended interfaces for rresvport, rcmd and rexec

TBD

14.1. 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 AF_INET TCP socket, it can also create an AF_INET6 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.)

14.2. rcmd_af

The existing rcmd() function can not transparently use AF_INET6 sockets since the an application would not be prepared to handle AF_INET6 addresses returned by e.g. getpeername on the file descriptor created by rcmd. Thus a new function is needed.

```
int rcmd_af(char **ahost, unsigned short rport, const char *locuser,
            const char *remuser, const char *cmd, int *fd2p, int af)
```

This function behaves the same as the existing rcmd() function, but instead of creating an AF_INET TCP socket, it can also create an AF_INET6 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.

14.3. rexec_af

The existing rexec() function can not transparently use AF_INET6 sockets since the an application would not be prepared to handle AF_INET6 addresses returned by e.g. getpeername on the file descriptor created by rexec. Thus a new function is needed.

```
int rexec_af(char **ahost, unsigned short rport, const char *name,
             const char *pass, const char *cmd, int *fd2p, int af)
```

This function behaves the same as the existing rexec() function, but instead of creating an AF_INET TCP socket, it can also create an AF_INET6 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.

15. 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_BEYONDSCOPE
<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_RECVDSTOPTS
<netinet/in.h>    IPV6_RECVHOPLIMIT
<netinet/in.h>    IPV6_RECVHOPOPTS
<netinet/in.h>    IPV6_RECVPKTINFO
```

```
  <netinet/in.h>    IPV6_RECVRTHDR
  <netinet/in.h>    IPV6_RECVRTHDRDSTOPTS
  <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_RTHDR
  <netinet/in.h>    IPV6_RTHDRDSTOPTS
  <netinet/in.h>    IPV6_RTHDR_TYPE_0
  <netinet/in.h>    IPV6_USE_MIN_MTU
  <netinet/in.h>    IPV6_RECVPATHMTU
  <netinet/in.h>    IPV6_PATHMTU
  <netinet/in.h>    IPV6_REACHCONF
  <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>    int inet6_opt_append(void *, size_t, int,
                                     uint8_t, size_t, uint_8, void **);
  <netinet/in.h>    int inet6_opt_get_val(void *, size_t, void *, int);
```

```
<netinet/in.h>    int inet6_opt_find(void *, size_t, int, uint8_t ,
                                     size_t *, void **);
<netinet/in.h>    int inet6_opt_finish(void *, size_t, int);
<netinet/in.h>    int inet6_opt_init(void *, size_t);
<netinet/in.h>    int inet6_opt_next(void *, size_t, int, uint8_t *,
                                     size_t *, void **);
<netinet/in.h>    int inet6_opt_set_val(void *, size_t, void *, int);

<netinet/in.h>    int inet6_rth_add(void *,
                                    const struct in6_addr *);
<netinet/in.h>    struct in6_addr inet6_rth_getaddr(const void *,
                                                    int);
<netinet/in.h>    void *inet6_rth_init(void *, int, int, int);
<netinet/in.h>    int inet6_rth_reverse(const void *, void *);
<netinet/in.h>    int inet6_rth_segments(const void *);
<netinet/in.h>    size_t inet6_rth_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);
<unistd.h>        int rcmd_af(char **, unsigned short, const char *,
                              const char *, const char *, int *, int);
<unistd.h>        int rexec_af(char **, unsigned short , const char *,
                               const char *, const char *, int *, int);
```

16. 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.

The ability to specify an arbitrary source address using IPV6_PKTINFO must be prevented; at least for non-privileged processes.

17. Change History

Changes from RFC 2292:

- Removed the IPV6_PKTOPTIONS socket option by allowing sticky

options to be set with individual setsockopt calls. This simplifies the protocol stack implementation by not having to handle options within options and also clarifies the failure semantics when some option is incorrectly formatted.

- Added the IPV6_RTHDRDSTOPTS for a Destination header before the Routing header. This is necessary to allow setting these Destination headers without IPV6_PKTOPTIONS.

- Removed the ability to be able to specify Hop-by-Hop and Destination options using multiple ancillary data items. The application, using the inet6_option_*() routines, is responsible for formatting the whole extension header. This removes the need for the protocol stack to somehow guess the alignment restrictions on options when concatenating them together.

- Added separate IPV6_RECVxxx options to enable the receipt of the corresponding ancillary data items. This makes the API cleaner since it allows the application to retrieve with getsockopt the sticky options it has set with setsockopt.

- Clarified how sticky options are turned off.

- Clarified how and when TCP returns ancillary data.

- Removed the support for the loose/strict Routing header since that has been removed from the IPv6 specification.

- Modified the inet6_rthdr_XXX() functions to not assume a cmsghdr structure in order to work with both sticky options and ancillary data. Renamed the functions to inet6_rth_XXX() to allow implementations to provide both the old and new functions.

- Modified the inet6_option_XXX() functions to not assume a cmsghdr structure in order to work with both sticky options and ancillary data. Renamed the functions to inet6_opt_XXX() to allow implementations to provide both the old and new functions.

- The new inet6_opt_XXX() functions were made different that the old as to not require structure declarations but instead use functions to add the individual fields to the option.

- Changed inet6_rthdr_getaddr() to operate on index O through N-1 (used to be 1 through N).

- Changed the comments in the struct ip6_hdr from "priority" to "traffic class".

- Clarified the alignment issues involving ancillary data to allow for separate alignment of cmsghdr structures and the data. Made CMSG_SPACE() return an upper bound on the needed space.

- Added rcmd_af() and rexec_af().

Changes since -00:

- Changed ICMP unreachable code 2 name to be "beyond scope of source address".

- Added motivation for rcmd_af() and rexec_af().

- Added option definitions (IP6OPT_PAD1 etc) to ip6.h.

- Added MLD and router renumbering definitions to icmp6.h

- Removed ip6r0_addr field - replaced by a comment.

- Made the content of IPV6_RTHDR, IPV6_HOPOPTS etc be specified as the extension header format (struct ip6_rthdr etc) instead of the previous "implementation dependent".

- Removed attempt at RFC 2292 compatibility.

- Excluded pad options from inet6_opt_next().

- Added IPV6_USE_MIN_MTU socket option for applications to avoid fragmentation by sending at the minimum IPv6 MTU.

- Added MTU notification so that UDP and raw socket applications can participate in path MTU discovery.

- Added Reachability confirmation for UDP and raw socket applications.

- Clarified that if the application asks for e.g., IPV6_RTHDR and a received datagram does not contain a Routing header an implementation will exclude the IPV6_RTHDR ancillary data item.

- Removed the constraints for jumbo option.

- Moved the new CMSG macros and changes from the appendix.

- Add text about inet6_opt_ depending on 2260 appendix B formatting rules i.e. largest field last in the option.

- Specified that getsockopt() of a sticky option returns what was

set with setsockopt().

18. TODO and Open Issues

Items left to do:

- Add credits for UDP MTU stuff
- Move information about mapping from inet6_opt to setsockopt and cmsg.
- Clarify IPV6_RTHDRDSTOPTS's interaction with IPV6_RTHDR.
- Make the new inet6_opt_set_val() and inet6_opt_get_val() check the length of the data item.

Open issues:

- Anything special for mobility options? Restrict setting at API? Filter out on receipt? If received what does the home address option contain?
- Specify "change" for TCP especially when there are multiple HBH option headers etc.
- Specify binding to scope-id => implies filtering of addresses with that scope if the address you are bound to is link-local etc. What about conflicts with bound scope-id and sendto/connect scope-id?
- Specify order for ifindex selection. Put in separate section. Different cases for sending to link-local (scope_id including nexthop scope_id) and global. Is multicast different?
- bind() and sin6_scope_id. Should have been in Basic API. Error checks when bind/sendto sin6_scope_id does not match?
- Specify notion of default multicast interface? In Basic API?
- What about site names and site ids? Need for interfaces to map? Requires that site-prefixes pass name - does name need to use DNS format to handle character sets?

19. References

[RFC-2460] Deering, S., Hinden, R., "Internet Protocol, Version 6 (IPv6), Specification", RFC 2460, Dec. 1998.

[RFC-2553] Gilligan, R. E., Thomson, S., Bound, J., Stevens, W., "Basic Socket Interface Extensions for IPv6", RFC 2553, March 1999.

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

[RFC-2461] Narten, T., Nordmark, E., Simpson, W., "Neighbor Discovery for IP Version 6 (IPv6)", RFC 2461, Dec. 1998.

20. 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, Jim Bound, Don Coolidge, Matt Crawford, Sam T. Denton, Richard Draves, Francis Dupont, Bob Gilligan, Jun-ichiro Hagino, Gerri Harter, Tim Hartrick, Bob Halley, Masaki Hirabaru, Yoshinobu Inoue, Mukesh Kacker, A. N. Kuznetsov, Sam Manthorpe, Pedro Marques, Jack McCann, der Mouse, John Moy, Thomas Narten, Steve Parker, Charles Perkins, Ken Powell, Tom Pusateri, Pedro Roque, Sameer Shah, Peter Sjodin, Stephen P. Spackman, Jinmei Tatuya, Karen Tracey, Sowmini Varadhan, Quaizar Vohra, Carl Williams, Steve Wise, Eric Wong, Farrell Woods, Kazu Yamamoto, and Vlad Yasevich.

21. Authors' Addresses

W. Richard Stevens
1202 E. Paseo del Zorro
Tucson, AZ 85718
Email: rstevens@kohala.com

Matt Thomas
3am Software Foundry
8053 Park Villa Circle
Cupertino, CA 95014
Email: matt@3am-software.com

Erik Nordmark
Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303, USA
Email: erik.nordmark@eng.sun.com

22. Appendix A: Ancillary Data Overview

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.

22.1. 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.)

(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.)

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 Routing header is 2048 bytes.)

22.2. The cmsghdr Structure

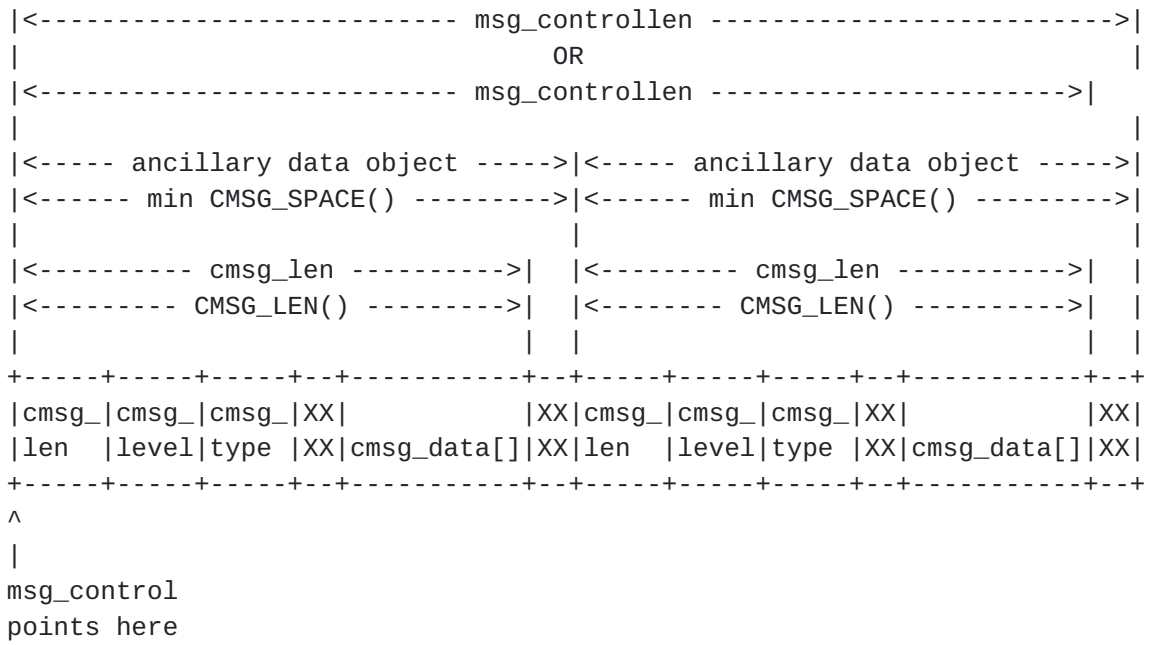
The cmsghdr structure describes ancillary data objects transferred by recvmsg() and sendmsg(). Its Posix.1g definition is:

```
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 in Section 22.3. Nevertheless, it is common to refer to the cmsg_data[] member.

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.



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. While sending an application may or may not include padding at the end of last ancillary data in msg_controllen and implementations must accept both as valid. On receiving a portable application must provide space for padding at the end of the last ancillary data as implementations may copy out the padding at the end of the control message buffer and include it in the received msg_controllen. When recvmsg() is called if msg_controllen is too small for all the ancillary data items including any trailing padding after the last item an implementation

may set MSG_CTRUNC.

22.3. 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>.

22.3.1. 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 by the 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.)

22.3.2. 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_H((cmsg)->cmsg_len) \
                          + ALIGN_D(sizeof(struct cmsghdr)) > \
         (u_char *)((mhdr)->msg_control) + (mhdr)->msg_controllen) ? \
        (struct cmsghdr *)NULL : \
        (struct cmsghdr *)((u_char *)(cmsg) + ALIGN_H((cmsg)->cmsg_len))))
```

The macros ALIGN_H() and ALIGN_D(), which are implementation dependent, round their arguments up to the next even multiple of whatever alignment is required for the start of the cmsghdr structure and the data, respectively. (This is probably a multiple of 4 or 8 bytes.) They are often the same macro in implementations platforms where alignment requirement for header and data is chosen to be identical.

22.3.3. 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_D(sizeof(struct cmsghdr)) )
```

22.3.4. 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 an upper bound on 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, when allocating 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_D(sizeof(struct cmsghdr)) + \
                             ALIGN_H(length) )
```

22.3.5. 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_D(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.

23. Appendix B: Examples using the inet6_rth_XXX() functions

Here we show an example for both sending Routing headers and processing and reversing a received Routing header.

23.1. Sending a Routing Header

As an example of these Routing header functions defined in this document, we go through the function calls for the example on p. 17 of [RFC-2460]. The source is S, the destination is D, and the three intermediate nodes are I1, I2, and I3.

```
            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
```

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.

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

```
    void  *extptr;
    int   extlen;
    struct msghdr  msg;
    struct cmsghdr  *cmsgptr;
    int   cmsglen;
    struct sockaddr_in6  I1, I2, I3, D;

    extlen = inet6_rth_space(IPV6_RTHDR_TYPE_0, 3);
    cmsglen = CMSG_SPACE(extlen);
    cmsgptr = malloc(cmsglen);
    cmsgptr->cmsg_len = CMSG_LEN(extlen);
    cmsgptr->cmsg_level = IPPROTO_IPV6;
    cmsgptr->cmsg_type = IPV6_RTHDR;

    optptr = CMSG_DATA(cmsgptr);
    optptr = inet6_rth_init(optptr, optlen, IPV6_RTHDR_TYPE_0, 3);

    inet6_rth_add(optptr, &I1.sin6_addr);
    inet6_rth_add(optptr, &I2.sin6_addr);
    inet6_rth_add(optptr, &I3.sin6_addr);

    msg.msg_control = cmsgptr;
    msg.msg_controllen = cmsglen;

    /* 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].

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_RECVPKTINFO and IPV6_RECVRTHDR 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). The number of segments in the Routing header is known from the Hdr Ext Len field in the Routing

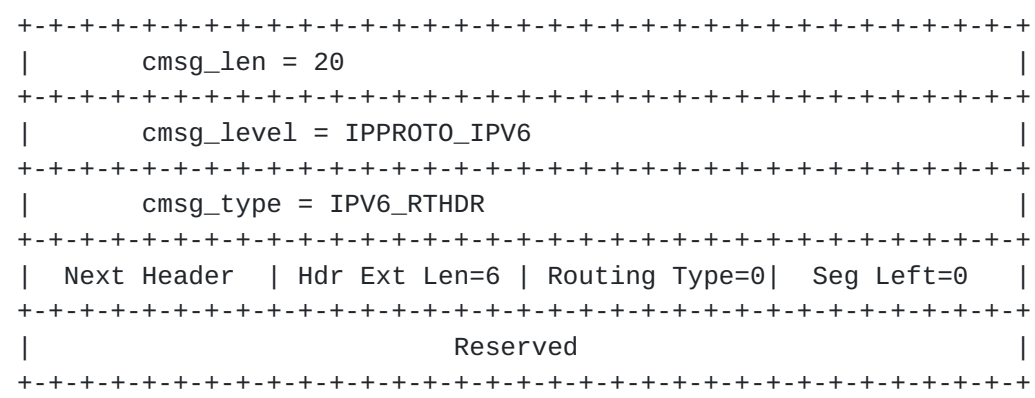
header (a value of 6, indicating 3 addresses).

The return value from inet6_rth_segments() will be 3 and inet6_rth_getaddr(0) will return I1, inet6_rth_getaddr(1) will return I2, and inet6_rth_getaddr(2) will return I3,

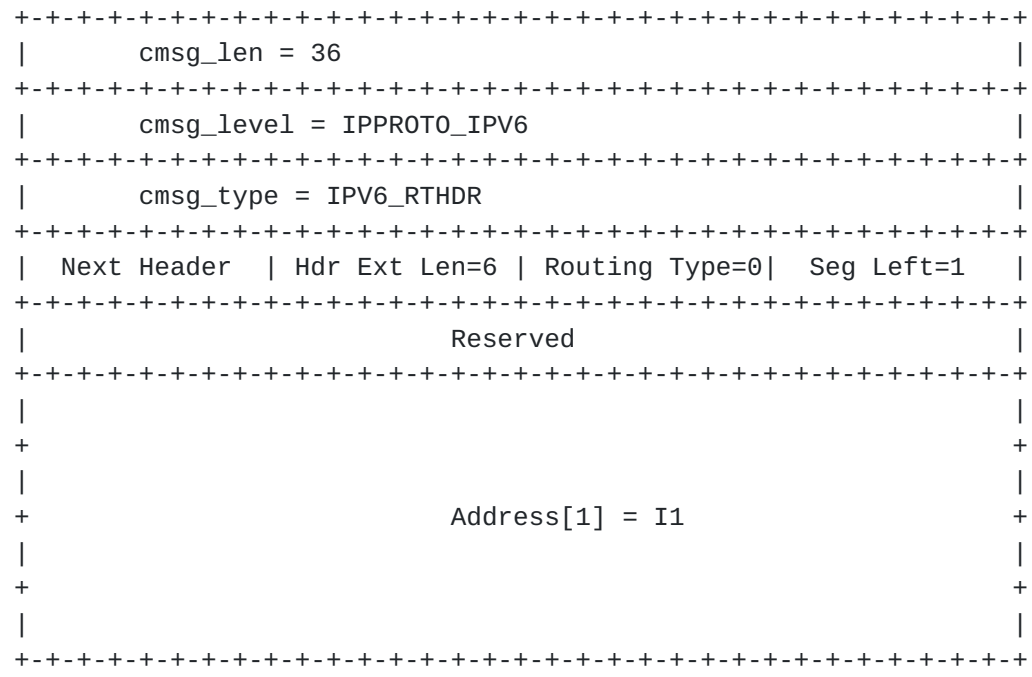
If the receiving application then calls inet6_rth_reverse(), the order of the three addresses will become I3, I2, and I1.

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_rth_space(3) returns 68: 12 bytes for the cmsghdr structure and 56 bytes for the Routing header (8 + 3*16).

The call to inet6_rth_init() initializes the ancillary data object to contain a Type 0 Routing header:

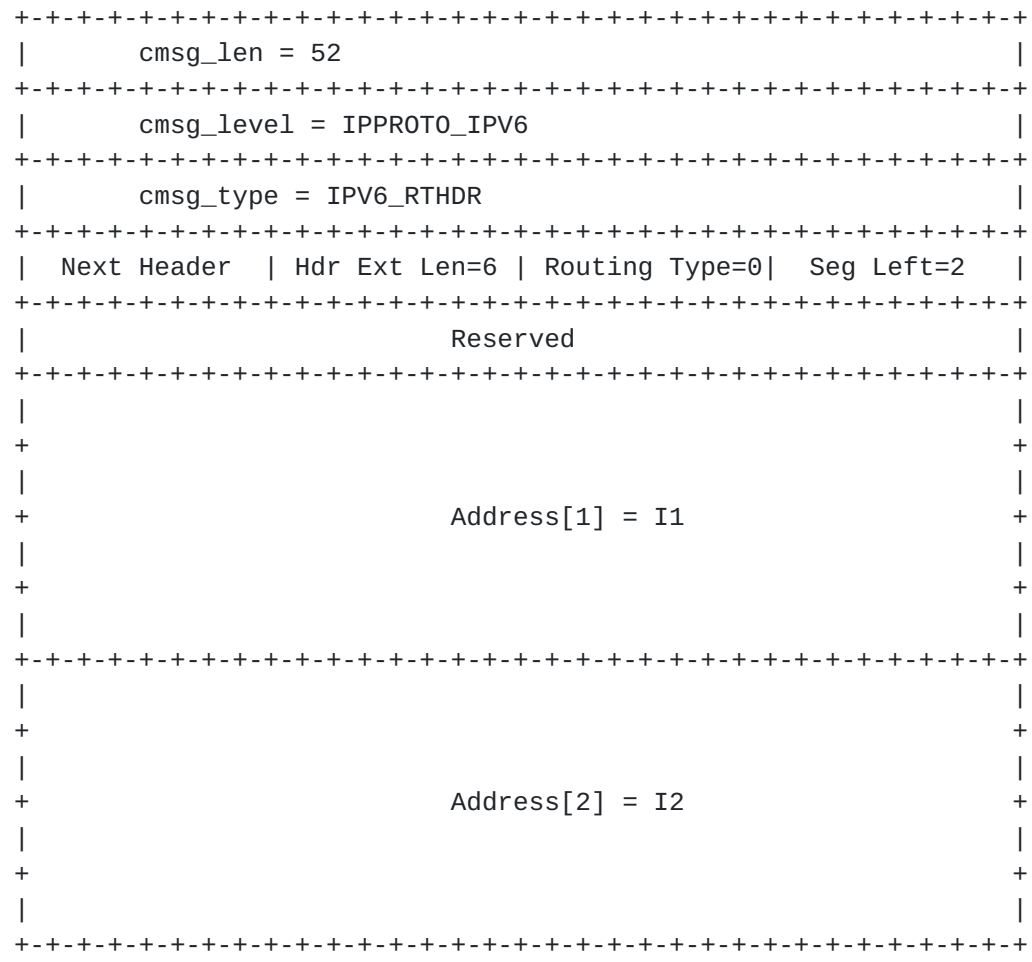


The first call to inet6_rth_add() adds I1 to the list.



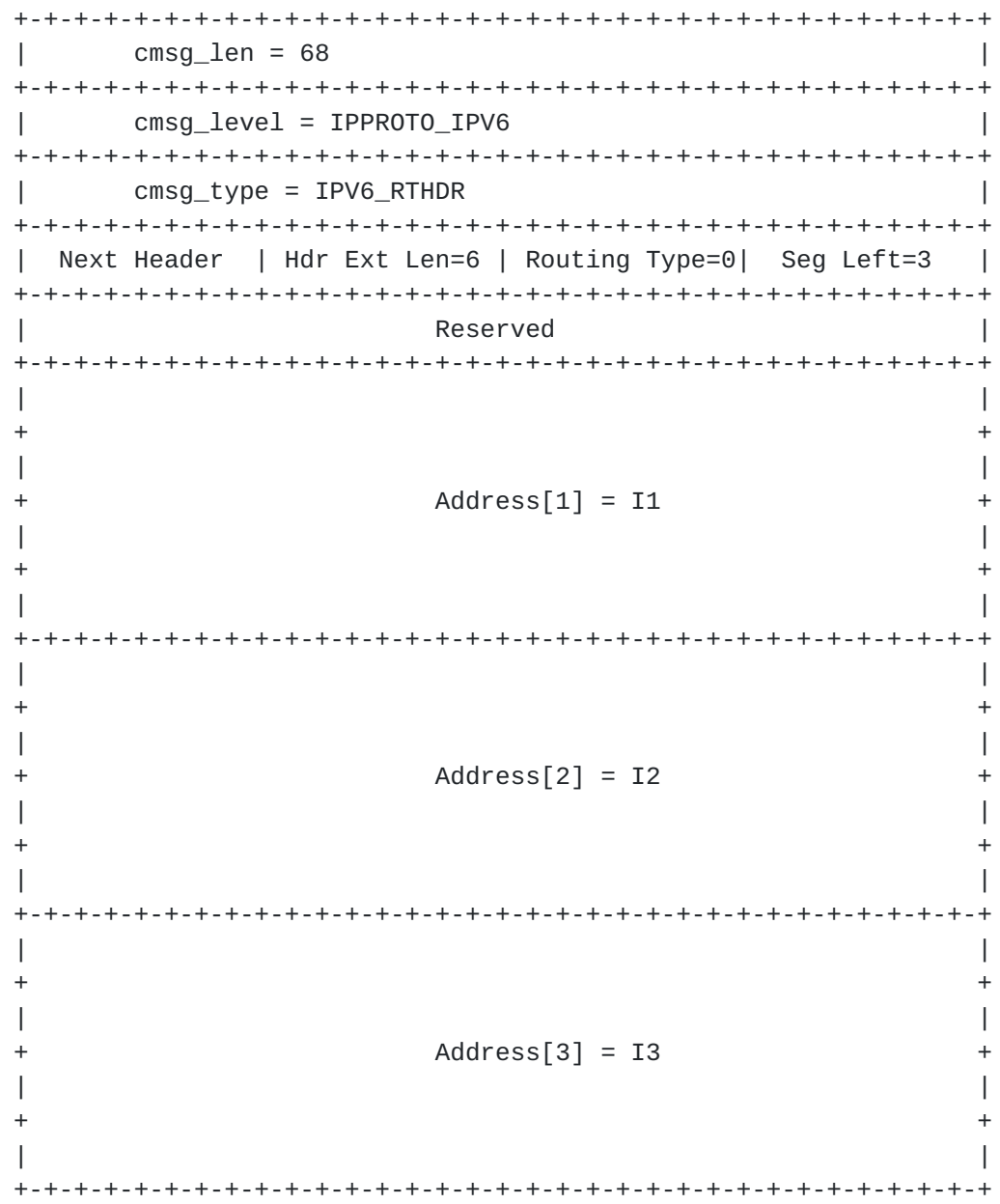
cmsg_len is incremented by 16, and the Segments Left field is incremented by 1.

The next call to inet6_rth_add() adds I2 to the list.



cmsg_len is incremented by 16, and the Segments Left field is incremented by 1.

The last call to inet6_rth_add() adds I3 to the list.



cmsg_len is incremented by 16, and the Segments Left field is incremented by 1.

23.2. Receiving Routing Headers

This example assumes that the application has enabled IPV6_RECVRTHDR socket option. The application prints and reverses a source route and uses that to echo the received data.

```
    struct sockaddr_in6     addr;
```

```
        struct msghdr           msg;
        struct iovec            iov;
        struct cmsghdr          *cmsgptr;
        size_t                  cmsgspace;
        void                    *optptr;
        int                     optlen;

        int                     segments;
        int                     i;
        char                    databuf[8192];

        segments = 100;        /* Enough */
        optlen = inet6_rth_space(IPV6_RTHDR_TYPE_0, segments);
        cmsgspace = CMSG_SPACE(optlen);
        cmsgptr = malloc(cmsgspace);
        if (cmsgptr == NULL) {
                perror("malloc");
                exit(1);
        }
        optptr = CMSG_DATA(cmsgptr);

        msg.msg_control = (char *)cmsgptr;
        msg.msg_controllen = cmsgspace;
        msg.msg_name = (struct sockaddr *)&addr;
        msg.msg_namelen = sizeof (addr);
        msg.msg_iov = &iov;
        msg.msg_iovlen = 1;
        iov.iov_base = databuf;
        iov.iov_len = sizeof (databuf);
        msg.msg_flags = 0;
        if (recvmsg(s, &msg, 0) == -1) {
                perror("recvmsg");
                return;
        }
        if (msg.msg_controllen != 0 &&
            cmsgptr->cmsg_level == IPPROTO_IPV6 &&
            cmsgptr->cmsg_type == IPV6_RTHDR) {
                struct in6_addr *in6;
                char asciiname[INET6_ADDRSTRLEN];
                struct ip6_rthdr0 *rthdr;

                rthdr = (struct ip6_rthdr0 *)optptr;
                segments = inet6_rth_segments(optptr);
                printf("route (%d segments, %d left): ",
                    segments, rthdr->ip6r0_segleft);
                for (i = 0; i < segments; i++) {
                        in6 = inet6_rth_getaddr(optptr, i);
                        if (in6 == NULL)
```

```
                        printf("<NULL> ");
                else
                        printf("%s ", inet_ntop(AF_INET6,
                            (void *)in6->s6_addr,
                            asciiname, INET6_ADDRSTRLEN));
        }
        if (inet6_rth_reverse(optptr, optptr) == -1) {
                printf("reverse failed");
                return;
        }
}
iov.iov_base = databuf;
iov.iov_len = strlen(databuf);
if (sendmsg(s, &msg, 0) == -1)
        perror("sendmsg");
if (cmsgptr != NULL)
        free(cmsgptr);
```

Note: The above example is a simple illustration. It skips some error checks involving the MSG_TRUNC and MSG_CTRUNC flags.

24. Appendix C: Examples using the inet6_opt_XXX() functions

This shows how Hop-by-Hop and Destination options can be both built as well as parsed using the inet6_opt_XXX() functions. This examples assume that there are defined values for OPT_X and OPT_Y.

24.1. Building options

We now provide an example that builds two Hop-by-Hop options using the example in Appendix B of [RFC-2460].

```
void *extbuf;
size_t extlen;
int currentlen;
void *databuf;
size_t offset;
uint8_t value1;
uint16_t value2;
uint32_t value4;
uint64_t value8;

/* Estimate the length */
currentlen = inet6_opt_init(NULL, 0);
if (currentlen == -1)
        return (-1);
```

```
        currentlen = inet6_opt_append(NULL, 0, currentlen, OPT_X, 12, 8, NULL);
        if (currentlen == -1)
                return (-1);
        currentlen = inet6_opt_append(NULL, 0, currentlen, OPT_Y, 7, 4, NULL);
        if (currentlen == -1)
                return (-1);
        currentlen = inet6_opt_finish(NULL, 0, currentlen);
        if (currentlen == -1)
                return (-1);
        extlen = currentlen;

        extbuf = malloc(extlen);
        if (extbuf == NULL) {
                perror("malloc");
                return (-1);
        }
        currentlen = inet6_opt_init(extbuf, extlen);
        if (currentlen == -1)
                return (-1);

        currentlen = inet6_opt_append(extbuf, extlen, currentlen,
            OPT_X, 12, 8, &databuf);
        if (currentlen == -1)
                return (-1);
        /* Insert value 0x12345678 for 4-octet field */
        offset = 0;
        value4 = 0x12345678;
        offset = inet6_opt_set_val(databuf, offset, &value4, sizeof (value4));
        /* Insert value 0x0102030405060708 for 8-octet field */
        value8 = 0x0102030405060708;
        offset = inet6_opt_set_val(databuf, offset, &value8, sizeof (value8));

        currentlen = inet6_opt_append(extbuf, extlen, currentlen,
            OPT_Y, 7, 4, &databuf);
        if (currentlen == -1)
                return (-1);
        /* Insert value 0x01 for 1-octet field */
        offset = 0;
        value1 = 0x01;
        offset = inet6_opt_set_val(databuf, offset, &value1, sizeof (value1));
        /* Insert value 0x1331 for 2-octet field */
        value2 = 0x1331;
        offset = inet6_opt_set_val(databuf, offset, &value2, sizeof (value2));
        /* Insert value 0x01020304 for 4-octet field */
        value4 = 0x01020304;
        offset = inet6_opt_set_val(databuf, offset, &value4, sizeof (value4));

        currentlen = inet6_opt_finish(extbuf, extlen, currentlen);
```

```
        if (currentlen == -1)
                return (-1);
        /* extbuf and extlen are now completely formatted */
```

24.2. Parsing received options

This example parses and prints the content of the two options in the previous example.

```
        int
        print_opt(void *extbuf, size_t extlen)
        {
                ip6_dest_t *ext;
                int currentlen;
                uint8_t type;
                size_t len;
                void *databuf;
                size_t offset;
                uint8_t value1;
                uint16_t value2;
                uint32_t value4;
                uint64_t value8;

                ext = (ip6_dest_t *)extbuf;
                printf("nxt %u, len %u (bytes %d)\n", ext->ip6d_nxt,
                    ext->ip6d_len, (ext->ip6d_len + 1) * 8);

                currentlen = 0;
                while (1) {
                        currentlen = inet6_opt_next(extbuf, extlen, currentlen,
                            &type, &len, &databuf);
                        if (currentlen == -1)
                                break;
                        printf("Received opt %u len %u\n",
                            type, len);
                        switch (type) {
                        case OPT_X:
                                offset = 0;
                                offset = inet6_opt_get_val(databuf, offset,
                                    &value4, sizeof (value4));
                                printf("X 4-byte field %x\n", value4);
                                offset = inet6_opt_get_val(databuf, offset,
                                    &value8, sizeof (value8));
                                printf("X 8-byte field %llx\n", value8);
                                break;
                        case OPT_Y:
```

```
                        offset = 0;
                        offset = inet6_opt_get_val(databuf, offset,
                            &value1, sizeof (value1));
                        printf("Y 1-byte field %x\n", value1);
                        offset = inet6_opt_get_val(databuf, offset,
                            &value2, sizeof (value2));
                        printf("Y 2-byte field %x\n", value2);
                        offset = inet6_opt_get_val(databuf, offset,
                            &value4, sizeof (value4));
                        printf("Y 4-byte field %x\n", value4);
                        break;
                    default:
                        printf("Unknown option %u\n", type);
                        break;
                    }
                }
                return (0);
        }
```