Raw Sockets 11 Part 1

What do we have in this chapter 11 part 1?

1. **Raw Socket Creation**
2. **ICMP**
3. **Ping Example**

Note: Common resolve.h and resolve.cpp files used in the program examples can be found in the last program sample.

A raw socket is one that allows access to the underlying transport protocol. This chapter is dedicated to illustrating how raw sockets can be used to simulate IP utilities, such as Traceroute and Ping. Raw sockets can also be used to manipulate IP header information. This chapter is concerned with the IPv4 and IPv6 protocols only; we will not address raw sockets with any other protocol because most protocols (except ATM) do not support raw sockets. All raw sockets are created using the SOCK_RAW socket type and are currently supported only under Winsock 2. Therefore, neither Microsoft Windows CE nor Windows 95 (without the Winsock 2 update) can use raw sockets.
In addition, using raw sockets requires substantial knowledge of the underlying protocol structure, which is not the focus of this book. In this chapter, we will discuss ICMP, ICMPv6, and UDP. ICMP (both versions) is used by the Ping utility, which can detect whether a route to a host is valid and whether the host machine is responding. Developers often need a programmatic method of determining whether a machine is alive and reachable. We will also examine UDP in conjunction with the IP_HDRINCL socket option to send completely fabricated IP packets. For all of these protocols, we will cover only the aspects necessary to fully explain the code in this chapter and in the example programs.

Raw Socket Creation

The first step in using raw sockets is creating the socket. You can use either socket() or WSASocket(). Note that for Windows 95, Windows 98, and Windows Me, no catalog entry in Winsock for IP has the SOCK_RAW socket type. However, this does not prevent you from creating this type of socket. It just means that you cannot create a raw socket using a WSAPROTOCOL_INFO structure. Refer back to Chapter 2 for information about enumerating protocol entries with the WSAEnumProtocols() function and the WSAPROTOCOL_INFO structure. You must specify the SOCK_RAW flag yourself in socket creation. The following code snippet illustrates the creation of a raw socket using ICMP as the underlying IP protocol:

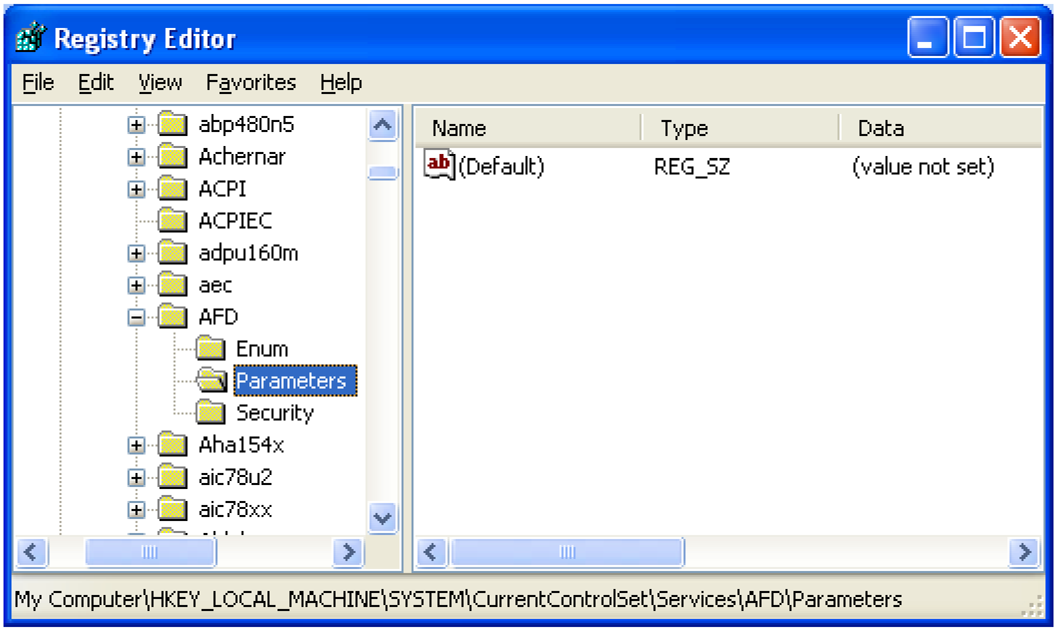
```
SOCKET    s;

s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
// Or
s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_ICMP, NULL,
0, WSA_FLAG_OVERLAPPED);
if (s == INVALID_SOCKET)
{
    // Socket creation failed
}
```

When creating a raw socket, the protocol parameter of the socket call becomes the protocol value in the IP header. That is, if a raw AF_INET6 socket is created with the protocol value 66, the IPv6 header for outgoing packets will contain the value 66 in the next header field.

Because raw sockets offer the capability to manipulate the underlying transport, they can be used for malicious purposes and are a security issue in Windows NT. Therefore, only members of the Administrators group can create sockets of type SOCK_RAW. Anyone can create a raw socket on Windows NT, but non-Administrators will not be able to do anything with it because the bind API will fail with WSAEACCES. Windows 95, Windows 98, and Windows Me do not impose any kind of limitation.

To work around this limitation on Windows NT, you can disable the security check on raw sockets by creating the following registry variable and setting its value to the integer 1 as a DWORD type.

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\AFD\Parameters\DisableRawSecurity
```



After the registry change, you need to reboot the machine. In the socket creation code in the example, we used the ICMP protocol, but you can also use IGMP, UDP, IP, or raw IP using the flags IPPROTO_IGMP, IPPROTO_UDP, IPPROTO_IP, or IPPROTO_RAW, respectively. However, be aware that on Windows 95 (with Winsock 2), Windows 98, and Windows NT 4, you can use only IGMP and ICMP when creating raw sockets. The protocol flags IPPROTO_UDP, IPPROTO_IP, and IPPROTO_RAW require the use of the socket option IP_HDRINCL, which is not supported on those platforms. Windows Me and Windows 2000 and later versions support IP_HDRINCL, so it is possible to manipulate the IP header (IPPROTO_RAW), the TCP header (IPPROTO_TCP), and the UDP header (IPPROTO_UDP).

Once the raw socket is created with the appropriate protocol flags, you can use the socket handle in send and receive calls. When creating raw sockets, the IP header will be included in the data returned upon any receive, regardless of whether the IP_HDRINCL option is set. Applications will have to know the layout of the IP header and have to determine the length of the IP header to find the payload data within the received buffer.

ICMP

ICMP is used as a means of messaging between hosts. Also, there are two versions of ICMP. The original ICMP is used with IPv4 to pass informational messages between two hosts, usually relating to

communications errors, such as destination unreachable or TTL exceeded. With IPv6, a new version of ICMP was created: **ICMPv6**. ICMPv6 includes the informational messages but also incorporates **ND** and **MLD**. As we discussed in Chapter 3, ND is the IPv6 equivalent to ARP and MLD is equivalent to IGMP. Our discussion of both versions of ICMP is limited to the informational messages.
As we mentioned previously, ICMP uses IPv4 addressing because it is a protocol encapsulated directly within an IPv4 datagram. Figure 11-1 illustrates the layout of an ICMP message. ICMPv6 is encapsulated in an IPv6 datagram and is identical in structure to the ICMP packet (as least in terms of the first four bytes).

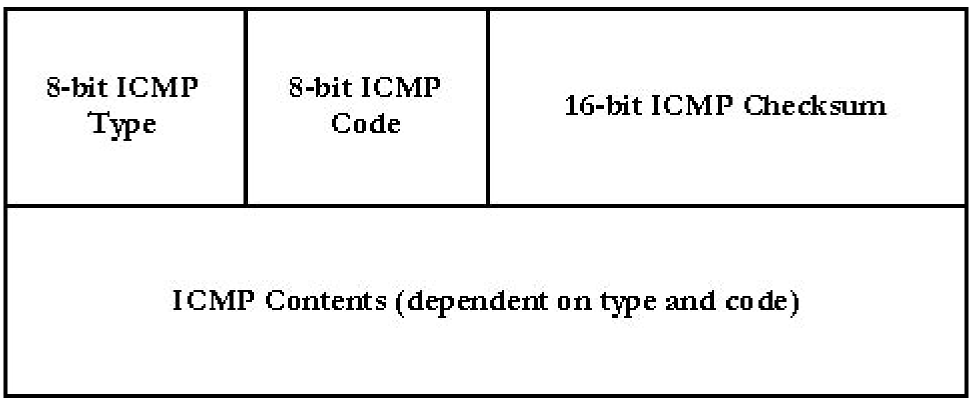


Figure 11-1 ICMP header

The first field is the ICMP message type, which is typically classified as either a query or an error. The code field further defines the type of query or message. The checksum field is the 16-bit one's complement sum of the ICMP header. Note that the checksum computation is different for IPv4 and IPv6. For IPv4, the checksum is calculated over the ICMP header and its payload only, and for ICMPv6, the checksum is calculated over the IPv6 pseudo-header followed by the ICMPv6 header and payload. The IPv6 pseudo-header is comprised of the following fields:

1. 128-bit IPv6 source address.
2. 128-bit IPv6 destination address.
3. 32-bit upper layer protocol packet length.

4. 24-bit zeroed field.
5. 8-bit next header protocol value.

IPv6 requires this pseudo-header calculation for any checksum calculation by an upper layer protocol that includes addresses from the IP header. This includes both UDP and ICMPv6. If the upper layer protocol contains its own packet length field, that value is used in the pseudo-header computation. Otherwise, the payload length from the IPv6 header is used, minus the size of all IPv6 extension headers present. Figure 11-5, later in this chapter, illustrates the IPv6 pseudo-header along with the UDP header and payload. Finally, the ICMP contents depend on the ICMP type and code. Table 11-1 lists the most common types and codes for ICMP, and Table 11-2 lists the common types and codes for ICMPv6. The type and code of the ICMP packet dictates what is to follow the ICMP header.

Table 11-1 ICMP Message Types

Type	**Query/Error (Error Type)**	**Code**	**Description**
0	Query	0	Echo reply
3	Error: Destination unreachable	0	Network unreachable
		1	Host unreachable
		2	Protocol unreachable
		3	Port unreachable
		4	Fragmentation needed, but the Don't Fragment bit has been set
		5	Source route failed
		6	Destination network unknown
		7	Destination host unknown
		8	Source host isolated (obsolete)
3	Error: Destination unreachable	9	Destination network administratively prohibited
		10	Destination host administratively prohibited
		11	Network unreachable for TOS
		12	Host unreachable for TOS

		13	Communication administratively prohibited by filtering
		14	Host precedence violation
		15	Precedence cutoff in effect
4	Error	0	Source quench
5	Error: Redirect	0	Redirect for network
		1	Redirect for host
		2	Redirect for TOS and network
		3	Redirect for TOS and host
8	Query	0	Echo request
9	Query	0	Router advertisement
10	Query	0	Router solicitation
11	Error: Time exceeded	0	TTL equals 0 during transit
		1	TTL equals 0 during reassembly
12	Error: Parameter problem	0	IP header bad
		1	Required option missing

When an ICMP error message is generated, it always contains as much of the IP header and IP payload that caused the error to occur without exceeding the MTU size. This allows the host receiving the ICMP error to associate the message with one particular protocol and process associated with that error. In our case, Ping relies on the echo request and echo reply ICMP queries rather than on error messages. In the next section, we will discuss how to use ICMP with a raw socket to generate a Ping request by using the echo request and echo reply messages. If you require more information about ICMP errors or the other types of ICMP queries, consult more in-depth sources, such as Stevens's TCP/IP Illustrated, Volume 1. Also, see RFCs 792 and 2463 for more information on ICMP and ICMPv6, respectively.

Table 11-2 ICMPv6 Message Types

Type	Query/Error (Error Type)	Code	Description
1	Error: Destination unreachable	0	No route to destination
		1	Communication with destination administratively prohibited
		3	Address unreachable
		4	Port unreachable
2	Error: Packet too big	0	Packet is larger than MTU size and cannot be forwarded
3	Error: Time exceeded	0	Hop limit exceeded in transit
		1	Fragment reassembly time exceeded
4	Error: Parameter problem	0	Erroneous header field encountered
		1	Unrecognized Next Header type encountered
		2	Unrecognized IPv6 option encountered
128	Query: Echo request	0	Request the destination to echo back the ICMP payload
129	Query: Echo reply	0	Reply to an echo request query

Ping Example

Ping is often used to determine whether a particular host is alive and reachable through the network. By generating an ICMP echo request and directing it to the host you are interested in, you can determine whether you can successfully reach that machine. Of course, this does not guarantee that a socket client will be able to connect to a process on that host (for example, a process on the remote server might not be listening); it just means that the network layer of the remote host is responding to network events. Finally, most operating systems offer the capability to turn off responding to ICMP echo requests, which is often the case for machines running firewalls. Essentially, the Ping example performs the following steps.

1. Creates a socket of address family AF_INET, type SOCK_RAW, and protocol IPPROTO_ICMP. For IPv6, the address family is AF_INET6, type SOCK_RAW, and protocol value 58.
2. Creates and initializes the ICMP header.
3. Calls sendto or WSASendTo to send the ICMP request to the remote host.
4. Calls recvfrom or WSARecvFrom to receive any ICMP responses.

Initializing the ICMP header is a straightforward task. First, the ICMP header is initialized with the type and code. Remember that the header is the same for ICMP and ICMPv6 (as shown in Figure 11-1). Following the type and code header, the echo request header must be supplied. This header is shown in Figure 11-2.

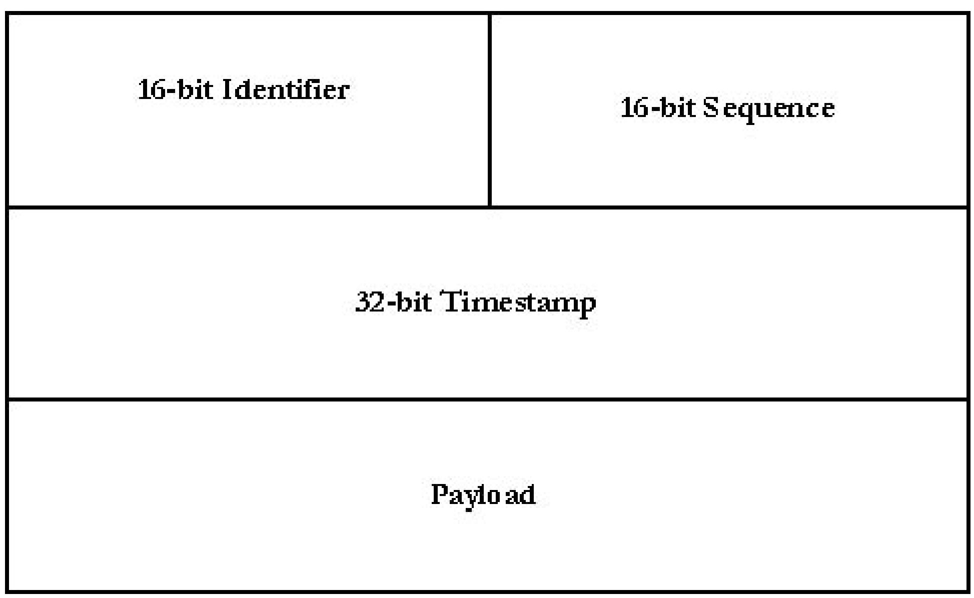


Figure 11-2 Echo request header

The first field is a 16-bit identifier, which is used to uniquely identify this request and is used to correlate echo replies received to your request and not some other process's request. Typically, the process identifier for the sending process is used. The next field is the sequence number, which identifies a given request packet from another. The 32-bit timestamp field is present only for ICMP requests (and not ICMPv6 requests). Following the request header is any payload. The following code sample illustrates initializing and sending an ICMP echo request for IPv4:

```
icmp = (ICMP_HDR *)buf;
icmp->icmp_type = 8;                          // echo
request type
icmp->icmp_code = 0;
icmp->icmp_id   = GetCurrentProcessId();
icmp->icmp_checksum = 0;               // zero field
before computing checksum
icmp->icmp_sequence = 0;
icmp->icmp_timestamp = GetTickCount();
// Fill in the payload with a random character
memset(&buf[sizeof(ICMP_HDR)], '@', 32);
// Compute the checksum over the ICMP header and
payload
//   The checksum() function computes the 16-bit one's
//   complement on the specified buffer.
icmp->icmp_checksum =
checksum(buf, sizeof(ICMP_HDR)+32);

s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

// Initialize the destination SOCKADDR_STORAGE
((SOCKADDR_IN *)&dest)->sin_family = AF_INET;
((SOCKADDR_IN *)&dest)->sin_port   = htons(0);
// port is ignored for ICMP
((SOCKADDR_IN *)&dest)->sin_addr.s_addr =
inet_addr("1.2.3.4");

sendto(s, buf, sizeof(ICMP_HDR)+32, 0, (SOCKADDR
*)&dest, sizeof(dest));
```

The only other difference between ICMP and ICMPv6 echo requests is computing the checksum contained in the ICMP header. For IPv4, the checksum is computed only over the ICMP header and payload. However, for IPv6 it is more complicated because IPv6 requires that the checksum include the IPv6 pseudo-header before the ICMPv6 header and payload. This means the Ping application must know the IPv6 source and destination address that will be in the IPv6 header to compute the checksum for any outgoing ICMPv6 requests. Because we are not building the IPv6 header by ourselves (as the case would be with the

IPV6_HDRINCL option), we have no control over what goes into the IPv6 header. However, it is possible to query the transport for which local interface will be used to reach a given destination. This is performed with the SIO_ROUTING_INTERFACE_QUERY ioctl. Once this query is done, we have all the necessary information to compute the pseudo-header checksum.

When you send the ICMP echo request, the remote machine intercepts it and sends an echo reply message back to you. If for some reason the host is not reachable, the appropriate ICMP error message, such as destination host unreachable, will be returned by a router somewhere along the path to the intended recipient. If the physical network connection to the host is good but the remote host is either down or not responding to network events, you need to perform your own timeout to determine this. Because the timestamp in the echo request is echoed, when the reply is received the elapsed time is easily calculated.

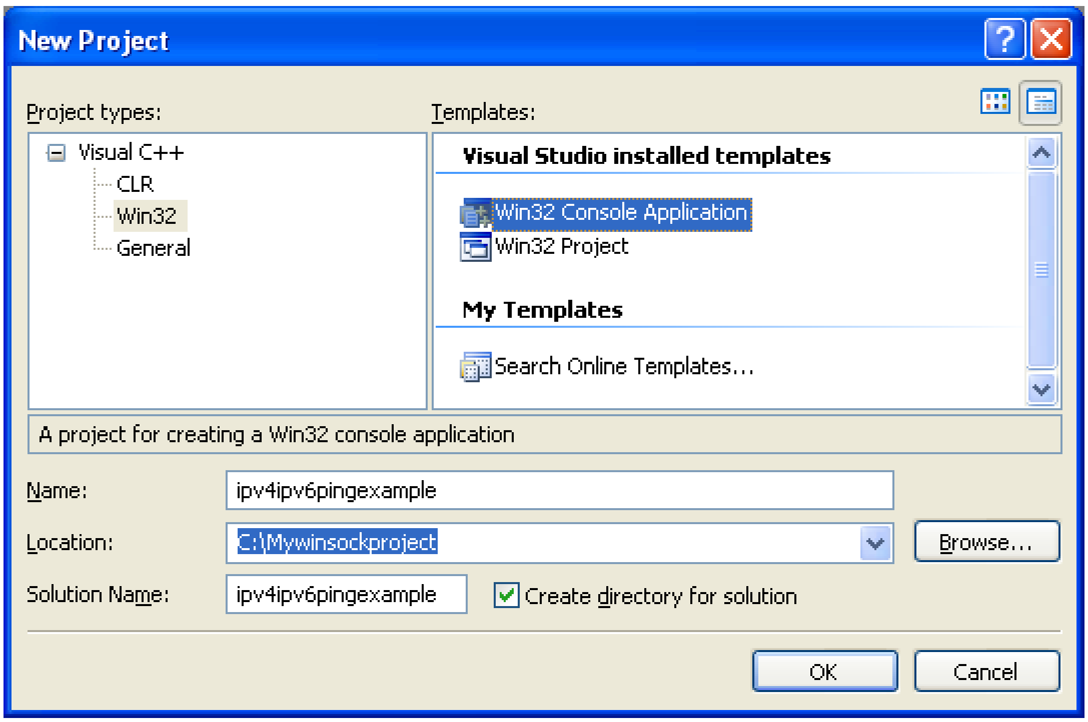
Raw Sockets 11 Part 2

What do we have in this chapter 11 part 2?

4. IPv4 and IPv6 Ping Program Example

IPv4 and IPv6 Ping Program Example

The following program example illustrates how to create a socket capable of sending and receiving ICMP packets, as well as how to use the IP_OPTIONS socket option to implement the record route option (supported for IPv4 only).



Add the following source code.

```
// Sample: IPv4 and IPv6 Ping Sample
//
// Files:
//    iphdrv2.h        - IPv4 and IPv6 packet header definitions
//    ipv4ipv6pingexamplesrc.cpp       - this file
```

```
 //    resolve.cpp   - Common name resolution routine
 //    resolve.h     - Header file for common name resolution
 routines
//
// Description:
//    This sample illustrates how to use raw sockets to send
ICMP
//    echo requests and receive their response. This sample
performs
//    both IPv4 and IPv6 ICMP echo requests. When using raw
sockets,
//    the protocol value supplied to the socket API is used as
the
//    protocol field (or next header field) of the IP packet.
Then
//    as a part of the data submitted to sendto, we include both
//    the ICMP request and data.
//
//    For IPv4 the IP record route option is supported via the
//    IP_OPTIONS socket option.
//
// Command Line Options/Parameters:
//     ipv4ipv6pingexample [-a 4|6] [-i ttl] [-l datasize] [-r]
[host]
//
//     -a       Address family (IPv4 or IPv6)
//     -i ttl   TTL value to set on socket
//     -l size  Amount of data to send as part of the ICMP
request
//     -r       Use IPv4 record route
//     host     Hostname or literal address
//
#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

// Link to ws2_32.lib
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
#include "resolve.h"
#include "iphdrv2.h"
```

```
#define DEFAULT_DATA_SIZE      32                   //
default data size
#define DEFAULT_SEND_COUNT     4               // number of
ICMP requests to send
#define DEFAULT_RECV_TIMEOUT   6000     // six second
#define DEFAULT_TTL            128

int   gAddressFamily=AF_UNSPEC,          // Address
family to use
      gProtocol=IPPROTO_ICMP,                         // Protocol value
      gTtl=DEFAULT_TTL;                                      // Default
TTL value
int   gDataSize=DEFAULT_DATA_SIZE;       // Amount of
data to send
BOOL  bRecordRoute=FALSE;                            // Use IPv4 record
route?
char *gDestination=NULL;                          //
Destination

// Function: usage
// Description: Print usage information.
int usage(char *progname)
{
    printf("Usage: %s -r <host> [data size]\n", progname);
    printf("       -a 4|6       Address family\n");
    printf("       -i ttl       Time to live\n");
    printf("       -l bytes     Amount of data to send\n");
    printf("       -r           Record route (IPv4 only)\n");
    printf("        host        Remote machine to ping\n");
    printf("Example: ipv4ipv6pingexample -a 4 -l 64 -r
www.google.com\n");
    printf("\n");
    return 0;
}

// Function: InitIcmpHeader
// Description: Helper function to fill in various stuff in our
ICMP request.
void InitIcmpHeader(char *buf, int datasize)
{
    ICMP_HDR   *icmp_hdr=NULL;
    char       *datapart=NULL;

    icmp_hdr = (ICMP_HDR *)buf;
```

```
    icmp_hdr->icmp_type     =
ICMPV4_ECHO_REQUEST_TYPE;         // request an ICMP echo
    icmp_hdr->icmp_code     = ICMPV4_ECHO_REQUEST_CODE;
    icmp_hdr->icmp_id       = (USHORT)GetCurrentProcessId();
    icmp_hdr->icmp_checksum = 0;
    icmp_hdr->icmp_sequence = 0;
    icmp_hdr->icmp_timestamp= GetTickCount();

    datapart = buf + sizeof(ICMP_HDR);

    // Place some junk in the buffer.
    memset(datapart, 'E', datasize);
}

// Function: InitIcmp6Header
// Description: Initialize the ICMP6 header as well as the echo
request header.
int InitIcmp6Header(char *buf, int datasize)
{
    ICMPV6_HDR          *icmp6_hdr=NULL;
    ICMPV6_ECHO_REQUEST *icmp6_req=NULL;
    char                *datapart=NULL;

    // Initialize the ICMP6 header fields
    icmp6_hdr = (ICMPV6_HDR *)buf;
    icmp6_hdr->icmp6_type     = ICMPV6_ECHO_REQUEST_TYPE;
    icmp6_hdr->icmp6_code     = ICMPV6_ECHO_REQUEST_CODE;
    icmp6_hdr->icmp6_checksum = 0;
    // Initialize the echo request fields
    icmp6_req = (ICMPV6_ECHO_REQUEST *)(buf
+ sizeof(ICMPV6_HDR));
    icmp6_req->icmp6_echo_id       =
(USHORT)GetCurrentProcessId();
    icmp6_req->icmp6_echo_sequence = 0;

    datapart = (char *)buf + sizeof(ICMPV6_HDR)
+ sizeof(ICMPV6_ECHO_REQUEST);
    memset(datapart, '$', datasize);
    return (sizeof(ICMPV6_HDR) + sizeof(ICMPV6_ECHO_REQUEST));
}

// Function: checksum
// Description: This function calculates the 16-bit one's
complement sum of the supplied buffer (ICMP) header.
USHORT checksum(USHORT *buffer, int size)
```

```
{
    unsigned long cksum=0;

    while (size > 1)
    {
        cksum += *buffer++;
        size -= sizeof(USHORT);
    }

    if (size)
    {
        cksum += *(UCHAR*)buffer;
    }
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >>16);
    return (USHORT)(~cksum);
}

// Function: ValidateArgs
// Description: Parse the command line arguments.
void ValidateArgs(int argc, char **argv)
{
    int    i;

    for(i=1; i < argc ;i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'a':        // address family
                    if (i+1 >= argc)
                        usage(argv[0]);
                    if (argv[i+1][0] == '4')
                        gAddressFamily = AF_INET;
                    else if (argv[i+1][0] == '6')
                        gAddressFamily = AF_INET6;
                    else
                        usage(argv[0]);
                    i++;
                    break;
                case 'i':        // Set TTL value
                    if (i+1 >= argc)
                        usage(argv[0]);
                    gTtl = atoi(argv[++i]);
                    break;
```

```
                case 'l':        // buffer size tos end
                    if (i+1 >= argc)
                        usage(argv[0]);
                    gDataSize = atoi(argv[++i]);
                    break;
                case 'r':        // record route option
                    bRecordRoute = TRUE;
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
        else
        {
            gDestination = argv[i];
        }
    }
    return;
}

// Function: SetIcmpSequence
// Description: This routine sets the sequence number of the
ICMP request packet.
void SetIcmpSequence(char *buf)
{
    ULONG    sequence=0;

    sequence = GetTickCount();
    if (gAddressFamily == AF_INET)
    {
        ICMP_HDR    *icmpv4=NULL;

        icmpv4 = (ICMP_HDR *)buf;
        icmpv4->icmp_sequence = (USHORT)sequence;
    }
    else if (gAddressFamily == AF_INET6)
    {
        ICMPV6_HDR          *icmpv6=NULL;
        ICMPV6_ECHO_REQUEST *req6=NULL;

        icmpv6 = (ICMPV6_HDR *)buf;
        req6   = (ICMPV6_ECHO_REQUEST *)(buf
+ sizeof(ICMPV6_HDR));

        req6->icmp6_echo_sequence = (USHORT)sequence;
```

```
    }
}

// Function: ComputeIcmp6PseudoHeaderChecksum
// Description:
//    This routine computes the ICMP6 checksum which includes
the pseudo
//    header of the IPv6 header (see RFC2460 and RFC2463). The
one difficulty
//    here is we have to know the source and destination IPv6
addresses which
//    will be contained in the IPv6 header in order to compute
the checksum.
//    To do this we call the SIO_ROUTING_INTERFACE_QUERY ioctl
to find which
//    local interface for the outgoing packet.
USHORT ComputeIcmp6PseudoHeaderChecksum(SOCKET
s, char *icmppacket, int icmplen, struct addrinfo *dest)
{
    SOCKADDR_STORAGE localif;
    DWORD            bytes;
    char             tmp[65535], *ptr=NULL, proto=0, zero=0;
    int              rc, total, length, i;

    // Find out which local interface for the destination
    rc = WSAIoctl(s, SIO_ROUTING_INTERFACE_QUERY, dest-
>ai_addr,dest->ai_addrlen,
            (SOCKADDR *)&localif, sizeof(localif), &bytes, NULL,
NULL);
    if (rc == SOCKET_ERROR)
    {
        fprintf(stderr, "WSAIoctl() failed with error code
%d\n", WSAGetLastError());
        return -1;
    }
    else
        printf("WSAIoctl() is OK!\n");
    // We use a temporary buffer to calculate the pseudo header.
    ptr = tmp;
    total = 0;

    // Copy source address
    memcpy(ptr, &((SOCKADDR_IN6 *)&localif)-
>sin6_addr, sizeof(struct in6_addr));
    ptr   += sizeof(struct in6_addr);
    total += sizeof(struct in6_addr);
```

```
    // Copy destination address
    memcpy(ptr, &((SOCKADDR_IN6 *)dest->ai_addr)-
>sin6_addr, sizeof(struct in6_addr));
    ptr   += sizeof(struct in6_addr);
    total += sizeof(struct in6_addr);
    // Copy ICMP packet length
    length = htonl(icmplen);

    memcpy(ptr, &length, sizeof(length));
    ptr   += sizeof(length);
    total += sizeof(length);
    // Zero the 3 bytes
    memset(ptr, 0, 3);
    ptr   += 3;
    total += 3;
    // Copy next hop header
    proto = IPPROTO_ICMP6;

    memcpy(ptr, &proto, sizeof(proto));
    ptr   += sizeof(proto);
    total += sizeof(proto);
    // Copy the ICMP header and payload
    memcpy(ptr, icmppacket, icmplen);
    ptr   += icmplen;
    total += icmplen;

    for(i=0; i < icmplen%2 ;i++)
    {
        *ptr = 0;
        ptr++;
        total++;
    }
    return checksum((USHORT *)tmp, total);
}

// Function: ComputeIcmpChecksum
// Description:
//    This routine computes the checksum for the ICMP request.
For IPv4 its
//    easy, just compute the checksum for the ICMP packet and
data. For IPv6,
//    its more complicated. The pseudo checksum has to be
computed for IPv6
//    which includes the ICMP6 packet and data plus portions of
the IPv6
```

```
//    header which is difficult since we aren't building our own
IPv6 header
void ComputeIcmpChecksum(SOCKET
s, char *buf, int packetlen, struct addrinfo *dest)
{
    if (gAddressFamily == AF_INET)
    {
        ICMP_HDR    *icmpv4=NULL;

        icmpv4 = (ICMP_HDR *)buf;
        icmpv4->icmp_checksum = 0;
        icmpv4->icmp_checksum = checksum((USHORT *)buf,
packetlen);
    }
    else if (gAddressFamily == AF_INET6)
    {
        ICMPV6_HDR  *icmpv6=NULL;

        icmpv6 = (ICMPV6_HDR *)buf;
        icmpv6->icmp6_checksum = 0;
        icmpv6->icmp6_checksum =
ComputeIcmp6PseudoHeaderChecksum(s, buf, packetlen, dest);
    }
}

// Function: PostRecvfrom
// Description: This routine posts an overlapped WSARecvFrom on
the raw socket.
int PostRecvfrom(SOCKET s, char *buf, int buflen,
SOCKADDR *from, int *fromlen, WSAOVERLAPPED *ol)
{
    WSABUF  wbuf;
    DWORD   flags, bytes;
    int     rc;

    wbuf.buf = buf;
    wbuf.len = buflen;

    flags = 0;

    rc = WSARecvFrom(s, &wbuf, 1, &bytes, &flags, from, fromlen,
ol, NULL);
    if (rc == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSA_IO_PENDING)
```

```
        {
            fprintf(stderr, "WSARecvFrom() failed with error
code %d\n", WSAGetLastError());
            return SOCKET_ERROR;
        }
    }
    return NO_ERROR;
}

// Function: PrintPayload
// Description:
//    This routine is for IPv4 only. It determines if there are
any IP options
//    present (by seeing if the IP header length is greater than
20 bytes) and
//    if so it prints the IP record route options.
void PrintPayload(char *buf, int bytes)
{
    int     hdrlen=0, routes=0, i;

    if (gAddressFamily == AF_INET)
    {
        SOCKADDR_IN      hop;
        IPV4_OPTION_HDR *v4opt=NULL;
        IPV4_HDR        *v4hdr=NULL;

        hop.sin_family = (USHORT)gAddressFamily;
        hop.sin_port   = 0;

        v4hdr = (IPV4_HDR *)buf;
        hdrlen = (v4hdr->ip_verlen & 0x0F) * 4;
        // If the header length is greater than the size of the
basic IPv4
        //    header then there are options present. Find them
and print them.
        if (hdrlen > sizeof(IPV4_HDR))
        {
            v4opt = (IPV4_OPTION_HDR *)(buf + sizeof(IPV4_HDR));
            routes = (v4opt->opt_ptr / sizeof(ULONG)) - 1;
            for(i=0; i < routes ;i++)
            {
                hop.sin_addr.s_addr = v4opt->opt_addr[i];
                // Print the route
                if (i == 0)
                    printf("    Route: ");
                else
```

```
                    printf("           ");
                PrintAddress((SOCKADDR *)&hop, sizeof(hop));

                if (i < routes-1)
                    printf(" ->\n");
                else
                    printf("\n");
            }
        }
    }
    return;
}

// Function: SetTtl
// Description: Sets the TTL on the socket.
int SetTtl(SOCKET s, int ttl)
{
    int     optlevel, option, rc;

    rc = NO_ERROR;
    if (gAddressFamily == AF_INET)
    {
        optlevel = IPPROTO_IP;
        option   = IP_TTL;
    }
    else if (gAddressFamily == AF_INET6)
    {
        optlevel = IPPROTO_IPV6;
        option   = IPV6_UNICAST_HOPS;
    }
    else
    {
        rc = SOCKET_ERROR;
    }
    if (rc == NO_ERROR)
    {
        rc = setsockopt(s, optlevel, option,
(char *)&ttl, sizeof(ttl));
    }
    if (rc == SOCKET_ERROR)
    {
        fprintf(stderr, "SetTtl: setsockopt() failed: %d\n",
WSAGetLastError());
    }
    return rc;
}
```

```
// Function: main
// Description:
//    Setup the ICMP raw socket and create the ICMP header. Add
//    the appropriate IP option header and start sending ICMP
//    echo requests to the endpoint. For each send and receive
we
//    set a timeout value so that we don't wait forever for a
//    response in case the endpoint is not responding. When we
receive a packet decode it.
int main(int argc, char **argv)
{
    WSADATA              wsd;
    WSAOVERLAPPED        recvol;
    SOCKET               s=INVALID_SOCKET;
    char                *icmpbuf=NULL, recvbuf[0xFFFF];
    struct addrinfo     *dest=NULL, *local=NULL;
    IPV4_OPTION_HDR      ipopt;
    SOCKADDR_STORAGE     from;
    DWORD                bytes, flags;
    int                  packetlen=0, recvbuflen=0xFFFF, fromlen,
time=0, rc, i;

    if(argc < 2)
    {
        usage(argv[0]);
        exit(1);
    }

    // Load Winsock
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("WSAStartup() failed with error code %d\n",
WSAGetLastError());
        return -1;
    }
    else
        printf("WSAStartup() should be fine!\n");

    // Parse the command line
    ValidateArgs(argc, argv);
    // Resolve the destination address
```

```
    dest = ResolveAddress(gDestination, "0", gAddressFamily, 0,
0);
    if (dest == NULL)
    {
        printf("Bad name %s\n", gDestination);
        return -1;
    }
    else
        printf("ResolveAddress() is pretty damn OK!\n");

    gAddressFamily = dest->ai_family;

    if (gAddressFamily == AF_INET)
        gProtocol = IPPROTO_ICMP;
    else if (gAddressFamily == AF_INET6)
        gProtocol = IPPROTO_ICMP6;
    // Get the bind address
    local = ResolveAddress(NULL, "0", gAddressFamily, 0, 0);
    if (local == NULL)
    {
        printf("Unable to obtain the bind address!\n");
        return -1;
    }
    else
        printf("ResolveAddress() is pretty damn OK!\n");

    // Create the raw socket
    s = socket(gAddressFamily, SOCK_RAW, gProtocol);
    if (s == INVALID_SOCKET)
    {
        printf("socket() failed with error code %d\n",
WSAGetLastError());
        return -1;
    }
    else
        printf("socket() is OK!\n");

    SetTtl(s, gTtl);

    // Figure out the size of the ICMP header and payload
    if (gAddressFamily == AF_INET)
        packetlen += sizeof(ICMP_HDR);
    else if (gAddressFamily == AF_INET6)
        packetlen += sizeof(ICMPV6_HDR)
+ sizeof(ICMPV6_ECHO_REQUEST);
```

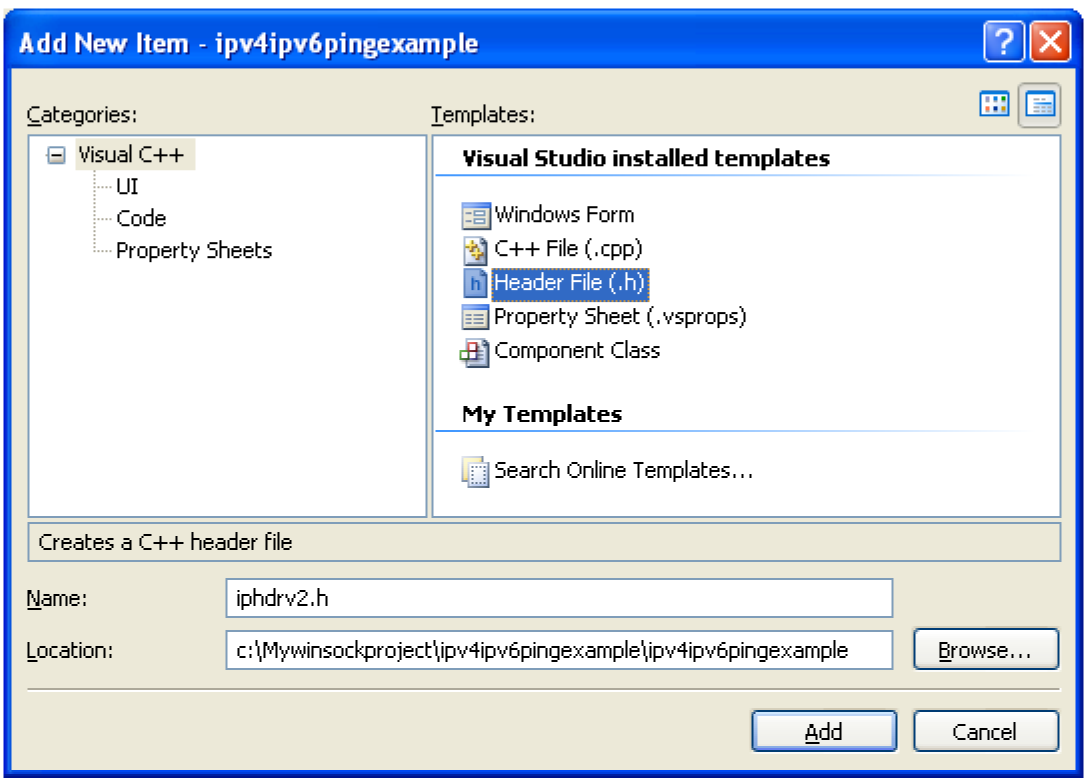
```
    // Add in the data size
    packetlen += gDataSize;
    // Allocate the buffer that will contain the ICMP request
    icmpbuf = (char *)HeapAlloc(GetProcessHeap(),
HEAP_ZERO_MEMORY, packetlen);
    if (icmpbuf == NULL)
    {
        fprintf(stderr, "HeapAlloc() for ICMP buffer failed with
error code %d\n", GetLastError());
        return -1;
    }
    else
        printf("HeapAlloc() for ICMP buffer is OK!\n");

    // Initialize the ICMP headers
    if (gAddressFamily == AF_INET)
    {
        if (bRecordRoute)
        {
            // Setup the IP option header to go out on every
ICMP packet
            ZeroMemory(&ipopt, sizeof(ipopt));
            ipopt.opt_code = IP_RECORD_ROUTE; // record route
option
            ipopt.opt_ptr  = 4;                  // point to the
first addr offset
            ipopt.opt_len  = 39;               // length of
option header

            rc = setsockopt(s, IPPROTO_IP, IP_OPTIONS,
(char *)&ipopt, sizeof(ipopt));
            if (rc == SOCKET_ERROR)
            {
                fprintf(stderr, "setsockopt(IP_OPTIONS) failed
with error code %d\n", WSAGetLastError());
                return -1;
            }
            else
                printf("setsockopt(IP_OPTIONS) is OK!\n");
        }
        InitIcmpHeader(icmpbuf, gDataSize);
    }
    else if (gAddressFamily == AF_INET6)
    {
        InitIcmp6Header(icmpbuf, gDataSize);
    }
```

```
    // Bind the socket -- need to do this since we post a
receive first
    rc = bind(s, local->ai_addr, local->ai_addrlen);
    if (rc == SOCKET_ERROR)
    {
        fprintf(stderr, "bind() failed with error code %d\n",
WSAGetLastError());
        return -1;
    }
    else
        printf("bind() looks fine!\n");

    // Setup the receive operation
    memset(&recvol, 0, sizeof(recvol));
    recvol.hEvent = WSACreateEvent();
    // Post the first overlapped receive
    fromlen = sizeof(from);
    PostRecvfrom(s, recvbuf, recvbuflen, (SOCKADDR *)&from,
&fromlen, &recvol);

    printf("\nPinging --> ");
    PrintAddress(dest->ai_addr, dest->ai_addrlen);
    printf(" with %d bytes of data\n\n", gDataSize);
    // Start sending the ICMP requests
    for(i=0; i < DEFAULT_SEND_COUNT ;i++)
    {
        // Set the sequence number and compute the checksum
        SetIcmpSequence(icmpbuf);
        ComputeIcmpChecksum(s, icmpbuf, packetlen, dest);

        time = GetTickCount();
        rc = sendto(s, icmpbuf, packetlen, 0, dest->ai_addr,
dest->ai_addrlen);
        if (rc == SOCKET_ERROR)
        {
            fprintf(stderr, "sendto() failed with error code
%d\n", WSAGetLastError());
            return -1;
        }
        else
            printf("sendto() is OK!\n");

        // Waite for a response
        rc = WaitForSingleObject((HANDLE)recvol.hEvent,
DEFAULT_RECV_TIMEOUT);
```

```
        if (rc == WAIT_FAILED)
        {
            fprintf(stderr, "WaitForSingleObject() failed with
error code %d\n", GetLastError());
            return -1;
        }
        else if (rc == WAIT_TIMEOUT)
        {
            printf("Request timed out.\n");
        }
        else
        {
            printf("WaitForSingleObject() looks OK!\n");
            rc = WSAGetOverlappedResult(s, &recvol, &bytes,
FALSE, &flags);
            if (rc == FALSE)
            {
                fprintf(stderr, "WSAGetOverlappedResult() failed
with error code %d\n", WSAGetLastError());
            }
            else
                printf("WSAGetOverlappedResult() is OK!\n");

            time = time - GetTickCount();

            WSAResetEvent(recvol.hEvent);

            printf("Reply from ");
            PrintAddress((SOCKADDR *)&from, fromlen);
            if (time == 0)
                printf(": bytes=%d time<1ms TTL=%d\n",
gDataSize, gTtl);
            else
                printf(": bytes=%d time=%dms TTL=%d\n",
gDataSize, time, gTtl);

            PrintPayload(recvbuf, bytes);

            if (i < DEFAULT_SEND_COUNT)
            {
                fromlen = sizeof(from);
                PostRecvfrom(s, recvbuf, recvbuflen, (SOCKADDR
*)&from, &fromlen, &recvol);
            }
        }
        Sleep(1000);
```

```
    }

    // Cleanup
    printf("Clean up steps....\n");
    freeaddrinfo(dest);
    freeaddrinfo(local);

    if (s != INVALID_SOCKET)
        closesocket(s);
    HeapFree(GetProcessHeap(), 0, icmpbuf);

    WSACleanup();
    return 0;
}
```

Add the iphdrv2.h header file



Add the source code.

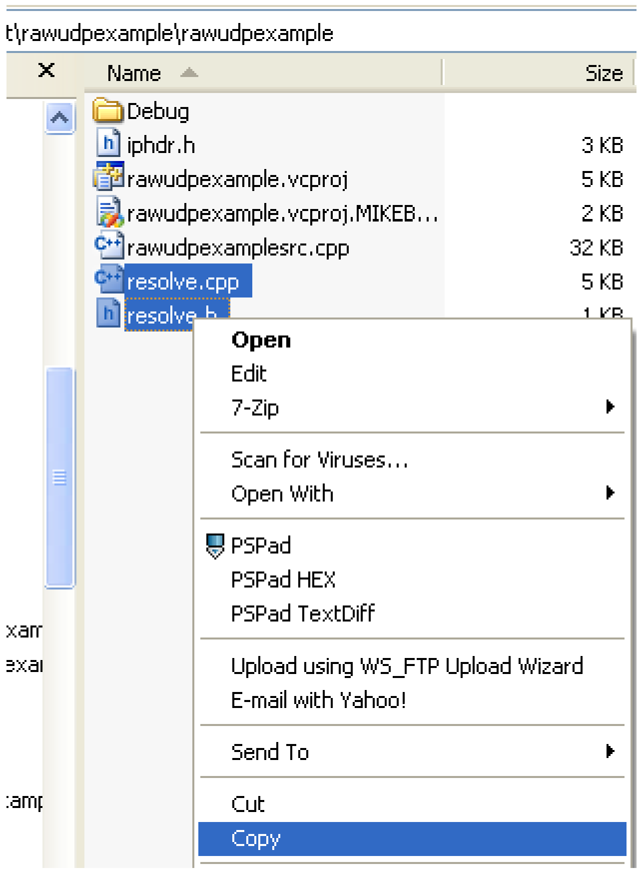
```
// Sample: Protocol header definitions used by ping (raw
sockets)
//
```

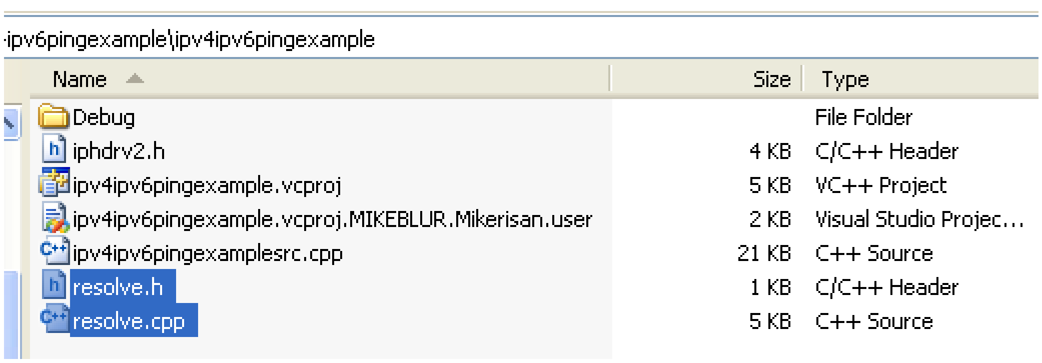
```
// Files:
//      iphdrv2.h      - this file
//
// Description:
//      This file contains various protocol header definitions
used by
//      the raw socket ping sample.
//
// Align on a 1-byte boundary
#include <pshpack1.h>

// IPv4 header
typedef struct ip_hdr
{
    unsigned char  ip_verlen;          // 4-bit IPv4 version 4-
bit header length (in 32-bit words)
    unsigned char  ip_tos;               // IP type of service
    unsigned short ip_totallength;   // Total length
    unsigned short ip_id;                // Unique identifier
    unsigned short ip_offset;          // Fragment offset field
    unsigned char  ip_ttl;                 // Time to live
    unsigned char  ip_protocol;       // Protocol(TCP,UDP etc)
    unsigned short ip_checksum;    // IP checksum
    unsigned int   ip_srcaddr;           // Source address
    unsigned int   ip_destaddr;        // Source address
} IPV4_HDR, *PIPV4_HDR, FAR * LPIPV4_HDR;

// IPv4 option header
typedef struct ipv4_option_hdr
{
    unsigned char   opt_code;        // option type
    unsigned char   opt_len;            // length of the option
header
    unsigned char   opt_ptr;            // offset into options
    unsigned long   opt_addr[9];     // list of IPv4 addresses
} IPV4_OPTION_HDR, *PIPV4_OPTION_HDR, FAR *LPIPV4_OPTION_HDR;

// ICMP header
typedef struct icmp_hdr
{
    unsigned char   icmp_type;
    unsigned char   icmp_code;
    unsigned short  icmp_checksum;
    unsigned short  icmp_id;
    unsigned short  icmp_sequence;
    unsigned long   icmp_timestamp;
```

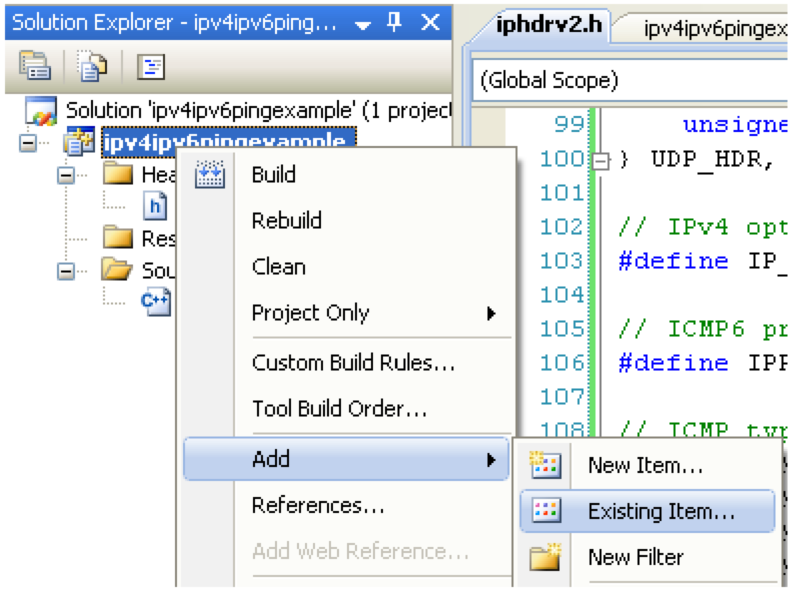
```
} ICMP_HDR, *PICMP_HDR, FAR *LPICMP_HDR;

// IPv6 protocol header
typedef struct ipv6_hdr
{
    unsigned long   ipv6_vertcflow;         // 4-bit IPv6
version, 8-bit traffic class, 20-bit flow label
    unsigned short  ipv6_payloadlen;     // payload length
    unsigned char   ipv6_nexthdr;           // next header
protocol value
    unsigned char   ipv6_hoplimit;          // TTL
    struct in6_addr ipv6_srcaddr;           // Source address
    struct in6_addr ipv6_destaddr;          // Destination
address
} IPV6_HDR, *PIPV6_HDR, FAR * LPIPV6_HDR;

// IPv6 fragment header
typedef struct ipv6_fragment_hdr
{
    unsigned char   ipv6_frag_nexthdr;
    unsigned char   ipv6_frag_reserved;
    unsigned short  ipv6_frag_offset;
    unsigned long   ipv6_frag_id;
} IPV6_FRAGMENT_HDR, *PIPV6_FRAGMENT_HDR, FAR *
LPIPV6_FRAGMENT_HDR;

// ICMPv6 header
typedef struct icmpv6_hdr {
    unsigned char   icmp6_type;
    unsigned char   icmp6_code;
    unsigned short  icmp6_checksum;
} ICMPV6_HDR;

// ICMPv6 echo request body
typedef struct icmpv6_echo_request
{
    unsigned short  icmp6_echo_id;
    unsigned short  icmp6_echo_sequence;
} ICMPV6_ECHO_REQUEST;

// Define the UDP header
typedef struct udp_hdr
{
    unsigned short src_portno;       // Source port no.
    unsigned short dst_portno;       // Dest. port no.
```

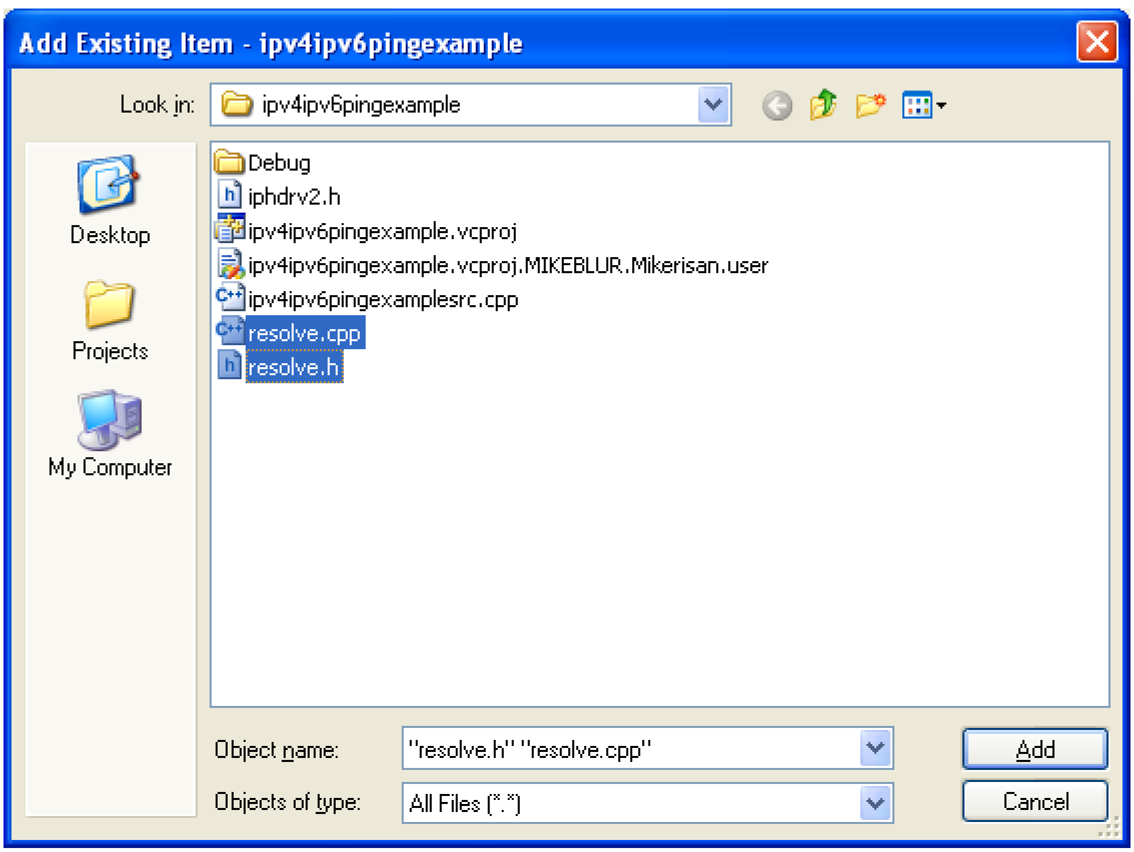
```
    unsigned short udp_length;        // Udp packet length
    unsigned short udp_checksum;      // Udp checksum (optional)
} UDP_HDR, *PUDP_HDR;

// IPv4 option for record route
#define IP_RECORD_ROUTE      0x7

// ICMP6 protocol value (used in the socket call and IPv6
header)
#define IPPROTO_ICMP6        58

// ICMP types and codes
#define ICMPV4_ECHO_REQUEST_TYPE   8
#define ICMPV4_ECHO_REQUEST_CODE   0
#define ICMPV4_ECHO_REPLY_TYPE     0
#define ICMPV4_ECHO_REPLY_CODE     0
#define ICMPV4_MINIMUM_HEADER      8

// ICPM6 types and codes
#define ICMPV6_ECHO_REQUEST_TYPE   128
#define ICMPV6_ECHO_REQUEST_CODE   0
#define ICMPV6_ECHO_REPLY_TYPE     129
#define ICMPV6_ECHO_REPLY_CODE     0

// Restore byte alignment back to default
#include <poppack.h>
```

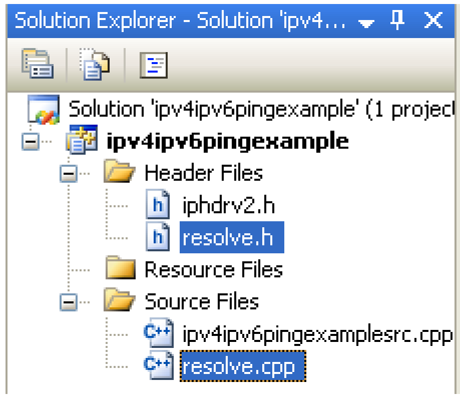
Add the resolve.h and its definition files into this project which were created in the previous project.


t\rawudpexample\rawudpexample
Name
Size
Debug
iphdr.h
3 KB
rawudpexample.vcproj
5 KB
rawudpexample.vcproj.MIKEB...
2 KB
rawudpexamplesrc.cpp
32 KB
resolve.cpp
5 KB
Open
Edit
7-Zip
Scan for Viruses...
Open With
PSPad
PSPad HEX
PSPad TextDiff
Upload using WS_FTP Upload Wizard
E-mail with Yahoo!
Send To
Cut
Copy

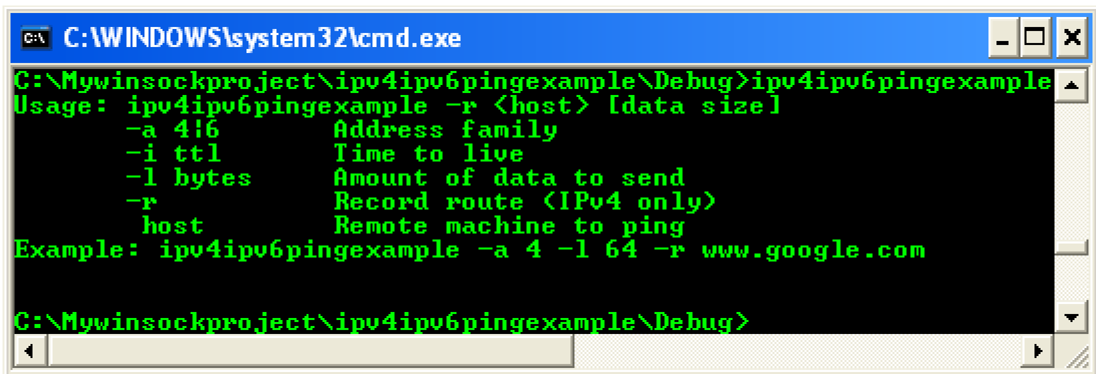

ipv6pingexample\ipv4ipv6pingexample
Name
Size
Type
Debug
File Folder
iphdrv2.h
4 KB
C/C++ Header
ipv4ipv6pingexample.vcproj
5 KB
VC++ Project
ipv4ipv6pingexample.vcproj.MIKEBLUR.Mikerisan.user
2 KB
Visual Studio Projec...
ipv4ipv6pingexamplesrc.cpp
21 KB
C++ Source
resolve.h
1 KB
C/C++ Header
resolve.cpp
5 KB
C++ Source


Solution Explorer - ipv4ipv6ping...
iphdrv2.h
ipv4ipv6pingex
(Global Scope)
Solution 'ipv4ipv6pingexample' (1 project
ipv4ipv6pingexample
Build
Rebuild
Clean
Project Only
Custom Build Rules...
Tool Build Order...
Add
References...
Add Web Reference...
New Item...
Existing Item...
New Filter
99 unsigne
100 } UDP_HDR,
101
102 // IPv4 opt
103 #define IP_
104
105 // ICMP6 pr
106 #define IPP
107

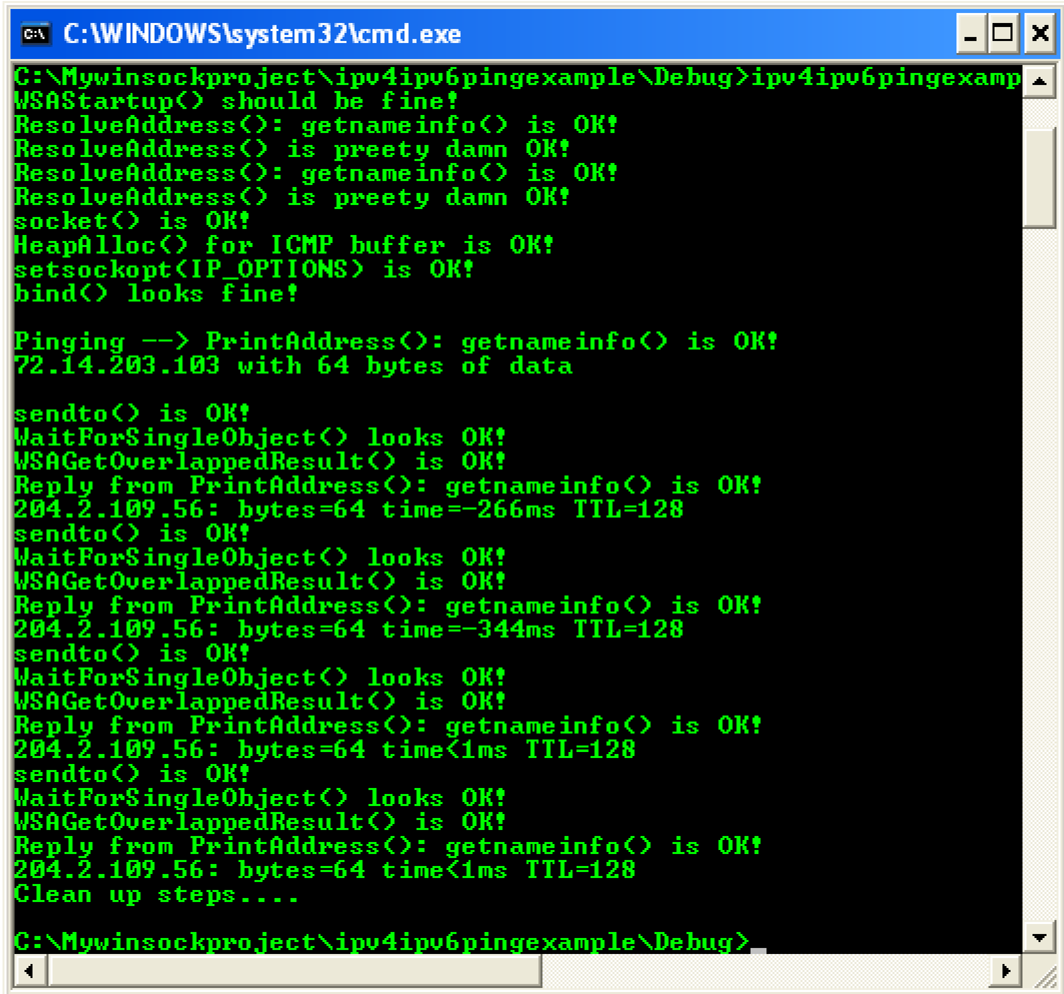

Add Existing Item - ipv4ipv6pingexample
Look in:
ipv4ipv6pingexample
Desktop
Projects
My Computer
Debug
iphdrv2.h
ipv4ipv6pingexample.vcproj
ipv4ipv6pingexample.vcproj.MIKEBLUR.Mikerisan.user
ipv4ipv6pingexamplesrc.cpp
resolve.cpp
resolve.h
Object name:
"resolve.h" "resolve.cpp"
Objects of type:
All Files (*.*)
Add
Cancel



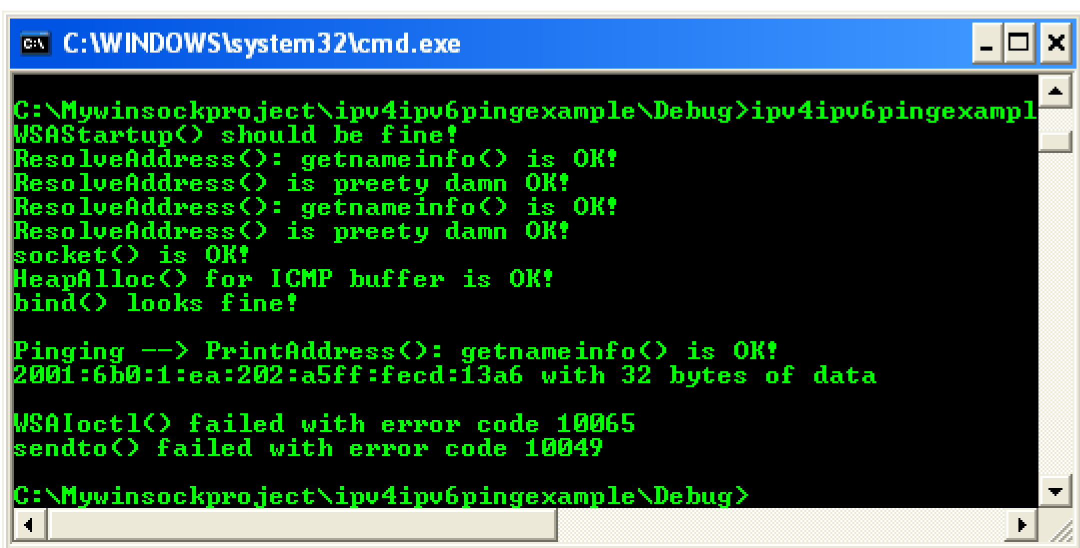
Build and run the project.



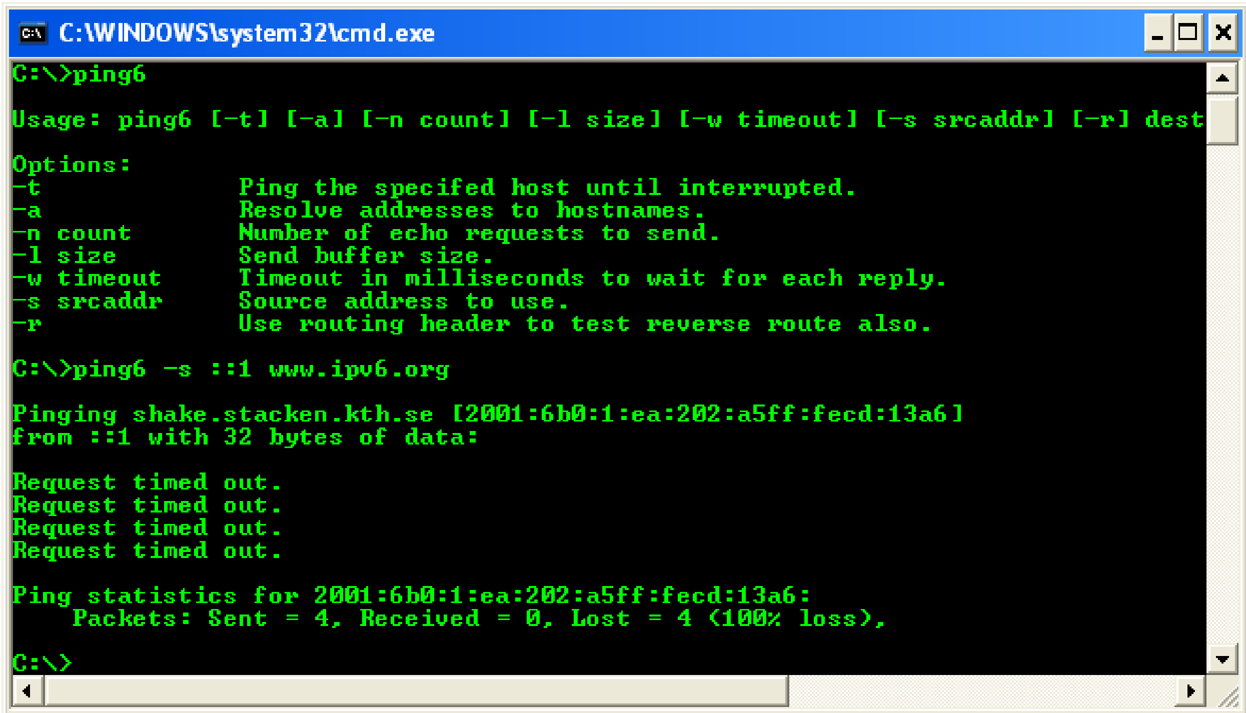
ipv4ipv6pingexample -a 4 -l 64 -r www.google.com



ipv4ipv6pingexample -a 6 www.ipv6.org



The following screenshots shows the Windows ping6 utility program with its’ options/switches.



One noticeable feature of the Ping example is its use of the IP_OPTIONS socket option. We use the record route IPv4 option so that when the ICMP packet hits a router, its IPv4 address is added into the IPv4 option header at the location indicated by the offset field in the IPv4 option header. This offset is also incremented by four each time a router adds its address. The increment value is based on the fact that an IPv4 address is 4 bytes long. Once you receive the echo reply, decode the option header and print the IP addresses and host names of the routers visited.

Raw Sockets 11 Part 3

What do we have in this chapter 11 part 3?

5. Traceroute

6. The Traceroute Program Example

Traceroute

Another valuable IP networking tool is the Traceroute utility. It allows you to determine the IP addresses of the routers that are traversed to reach a certain host on the network. With Ping, using the record route option in the IPv4 option header also allows you to determine the IPv4 addresses of intermediary routers, but Ping is limited to only 9 hops, the maximum space allocated for addresses in the option header. Also, there is no equivalent option for IPv6. A hop occurs whenever an IP datagram must pass through a router to traverse multiple physical networks.

The idea behind Traceroute is to send a UDP packet to the destination and incrementally change the TTL value. Initially, the TTL value is 1, which means the UDP packet will reach the first router, where the TTL will expire. The expiration will cause the router to generate an ICMP time-exceeded packet. Then the initial TTL value increases by 1, so the UDP packet gets one router farther and an ICMP time-exceeded packet is sent from that router. Collecting each of the ICMP messages gives you a clear path of the IP addresses traversed to reach the endpoint. Once the TTL is incremented enough so that packets actually reach the endpoint, an ICMP port-unreachable message is most likely returned because no process on the recipient is waiting for this message.

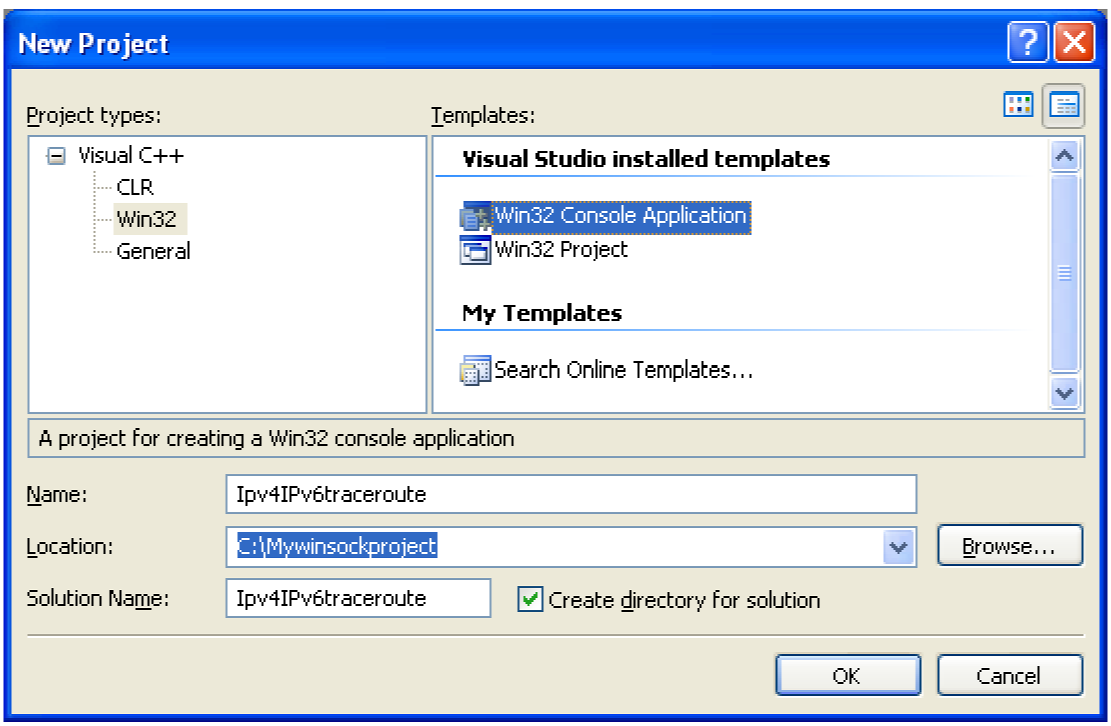
Traceroute is a useful utility because it gives you a lot of information about the route to a particular host, which is often a concern when you use multicasting or when you experience routing problems. Fewer applications need to perform Traceroute programmatically than Ping, but certain tasks might require Traceroute-like capabilities.

Two methods can be used to implement the Traceroute program. First, you can use UDP packets and send datagrams, incrementally changing the TTL. Each time the TTL expires, an ICMP message will be returned to you. This method requires one socket of UDP to send the messages and

another socket of ICMP to read them. The UDP socket is a normal UDP socket. The ICMP socket is a raw socket, which we already discussed how to create. The TTL of the UDP socket needs to be manipulated via the IP_TTL or IPV6_UNICAST_HOPS socket option. Alternatively, you can create a UDP socket and use the IP_HDRINCL option (discussed later in this chapter) to set the TTL manually within the IP header, but this is quite a lot of work.
The other method is simply to send ICMP packets to the destination, also incrementally changing the TTL. This also results in ICMP error messages being returned when the TTL expires. This method resembles the Ping example because it requires only one socket (of ICMP).

The Traceroute Program Example

The following program example is a simple Traceroute sample that uses ICMP packets. The traceroute is similar in structure and code to the Ping sample. The only difference is that the TTL value is incremented with each send.



Add the following source code.

```
// Sample: IPv4 and IPv6 Trace Route Sample
```

```
//
// Files:
//    iphdrv3.h     - IPv4 and IPv6 packet header definitions
//    Ipv4IPv6traceroutesrc.cpp   - this file
//    resolve.cpp   - Common name resolution routine
//    resolve.h     - Header file for common name resolution
routines
//
// Description:
//    This sample illustrates how to use raw sockets to send
ICMP
//    echo requests and receive their response in order to
determine the
//    route to a particular destination. This sample performs
//    both IPv4 and IPv6 trace route operations. When using raw
sockets,
//    the protocol value supplied to the socket API is used as
the
//    protocol field (or next header field) of the IP packet.
Then
//    as a part of the data submitted to sendto, we include both
//    the ICMP request and data. We start by setting the TTL
value to one
//    and sending a request. When an intermediate router
intercepts the
//    packet the TTL is decremented. If the value is zero, it
sends an
//    ICMP TTL expired message which we receive. From the IP
packet's
//    source field we have a location in the route to the
destination.
//    With each send, the TTL is incremented by one until the
specified
//    destination is reached.
//
// Command Line Options/Parameters:
//     Ipv4IPv6traceroute [-a 4|6] [-d] [-h ttl] [-w timeout]
[host]
//
//     -a       Address family (IPv4 or IPv6)
//     -d       Do not resolve the addresses to hostnames
//     -h ttl   Maximum TTL value
//     -w time  Timeout in milliseconds to wait for a response
//     host     Hostname or literal address
//
#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
```

```
#endif

// Link to ws2_32.lib
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

#include "resolve.h"
#include "iphdrv3.h"

#define DEFAULT_DATA_SIZE       32        // default data
size
#define DEFAULT_RECV_TIMEOUT    6000      // six second
#define DEFAULT_TTL             30        // default
timeout

// Global variables
int   gAddressFamily = AF_UNSPEC,          // Address
family to use
      gProtocol      = IPPROTO_ICMP,      // Protocol value
      gTtl           = DEFAULT_TTL,       // Default TTL value
      gTimeout       = DEFAULT_RECV_TIMEOUT; // Amount of data
to send
BOOL  bResolve       = TRUE;              // Resolve addresses
to host names
char *gDestination   = NULL;              // Destination

// Function: usage
// Description:  Print usage information.
int usage(char *progname)
{
    printf("Usage: %s -r <host> [data size]\n", progname);
    printf("       -a 4|6       Address family\n");
    printf("       -d           Do not resolve addresses to
hostnames\n");
    printf("       -h ttl       Maximum hops to search for
target\n");
    printf("       -w timeout   Wait timeout in milliseconds for
each reply\n");
    printf("        host        Remote machine to ping\n");
            printf("\n");
    return 0;
```

```
}

// Function: InitIcmpHeader
// Description: Helper function to fill in various stuff in our
ICMP request.
void InitIcmpHeader(char *buf, int datasize)
{
    ICMP_HDR   *icmp_hdr=NULL;
    char       *datapart=NULL;

    icmp_hdr = (ICMP_HDR *)buf;
    icmp_hdr->icmp_type     =
ICMPV4_ECHO_REQUEST_TYPE;        // request an ICMP echo
    icmp_hdr->icmp_code     = ICMPV4_ECHO_REQUEST_CODE;
    icmp_hdr->icmp_id       = (USHORT)GetCurrentProcessId();
    icmp_hdr->icmp_checksum = 0;
    icmp_hdr->icmp_sequence = 0;
    icmp_hdr->icmp_timestamp= GetTickCount();

    datapart = buf + sizeof(ICMP_HDR);

    // Place some junk in the buffer.
    memset(datapart, 'E', datasize);
}

// Function: InitIcmp6Header
// Description: Initialize the ICMP6 header as well as the echo
request header.
int InitIcmp6Header(char *buf, int datasize)
{
    ICMPV6_HDR          *icmp6_hdr=NULL;
    ICMPV6_ECHO_REQUEST *icmp6_req=NULL;
    char                *datapart=NULL;

    // Initialize the ICMP6 header fields
    icmp6_hdr = (ICMPV6_HDR *)buf;
    icmp6_hdr->icmp6_type     = ICMPV6_ECHO_REQUEST_TYPE;
    icmp6_hdr->icmp6_code     = ICMPV6_ECHO_REQUEST_CODE;
    icmp6_hdr->icmp6_checksum = 0;

    // Initialize the echo request fields
    icmp6_req = (ICMPV6_ECHO_REQUEST *)(buf
+ sizeof(ICMPV6_HDR));
    icmp6_req->icmp6_echo_id       =
(USHORT)GetCurrentProcessId();
```

```
    icmp6_req->icmp6_echo_sequence = 0;

    datapart = (char *)buf + sizeof(ICMPV6_HDR)
+ sizeof(ICMPV6_ECHO_REQUEST);
    memset(datapart, '$', datasize);

    return (sizeof(ICMPV6_HDR) + sizeof(ICMPV6_ECHO_REQUEST));
}

// Function: checksum
// Description:
//    This function calculates the 16-bit one's complement sum
//    of the supplied buffer (ICMP) header.
USHORT checksum(USHORT *buffer, int size)
{
    unsigned long cksum=0;

    while (size > 1)
    {
        cksum += *buffer++;
        size -= sizeof(USHORT);
    }
    if (size)
    {
        cksum += *(UCHAR*)buffer;
    }
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >>16);
    return (USHORT)(~cksum);
}

// Function: ValidateArgs
// Description: Parse the command line arguments.
void ValidateArgs(int argc, char **argv)
{
    int    i;

    for(i=1; i < argc ;i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'a':        // address family
                    if (i+1 >= argc)
```

```
                        usage(argv[0]);
                    if (argv[i+1][0] == '4')
                        gAddressFamily = AF_INET;
                    else if (argv[i+1][0] == '6')
                        gAddressFamily = AF_INET6;
                    else
                        usage(argv[0]);
                    i++;
                    break;
                case 'd':        // Don't resolve addresses
                    bResolve = FALSE;
                    break;
                case 'h':        // Set TTL value
                    if (i+1 >= argc)
                        usage(argv[0]);
                    gTtl = atoi(argv[++i]);
                    break;
                case 'w':        // Timeout in milliseconds for
each reply
                    if (i+1 >= argc)
                        usage(argv[0]);
                    gTimeout = atoi(argv[++i]);
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
        else
        {
            gDestination = argv[i];
        }
    }
    return;
}



// Function: SetIcmpSequence
// Description: This routine sets the sequence number of the
ICMP request packet.
void SetIcmpSequence(char *buf)
{
    ULONG    sequence=0;
```

```
    sequence = GetTickCount();
    if (gAddressFamily == AF_INET)
    {
        ICMP_HDR    *icmpv4=NULL;

        icmpv4 = (ICMP_HDR *)buf;
        icmpv4->icmp_sequence = (USHORT)sequence;
    }
    else if (gAddressFamily == AF_INET6)
    {
        ICMPV6_HDR          *icmpv6=NULL;
        ICMPV6_ECHO_REQUEST *req6=NULL;

        icmpv6 = (ICMPV6_HDR *)buf;
        req6   = (ICMPV6_ECHO_REQUEST *)(buf
+ sizeof(ICMPV6_HDR));

        req6->icmp6_echo_sequence = (USHORT)sequence;
    }
}

// Function: ComputeIcmp6PseudoHeaderChecksum
// Description:
//    This routine computes the ICMP6 checksum which includes
the pseudo
//    header of the IPv6 header (see RFC2460 and RFC2463). The
one difficulty
//    here is we have to know the source and destination IPv6
addresses which
//    will be contained in the IPv6 header in order to compute
the checksum.
//    To do this we call the SIO_ROUTING_INTERFACE_QUERY ioctl
to find which
//    local interface for the outgoing packet.
USHORT ComputeIcmp6PseudoHeaderChecksum(SOCKET
s, char *icmppacket, int icmplen, struct addrinfo *dest)
{
    SOCKADDR_STORAGE localif;
    DWORD            bytes;
    char             tmp[65535], *ptr=NULL, proto=0, zero=0;
    int              rc, total, length, i;

    // Find out which local interface for the destination
    rc = WSAIoctl(s, SIO_ROUTING_INTERFACE_QUERY, dest-
>ai_addr,dest->ai_addrlen,
```

```
                        (SOCKADDR *)&localif, sizeof(localif),
&bytes, NULL, NULL);
    if (rc == SOCKET_ERROR)
    {
        fprintf(stderr, "WSAIoctl() failed with error code
%d\n", WSAGetLastError());
        return -1;
    }
    else
        printf("WSAIoctl() is OK!\n");

    // We use a temporary buffer to calculate the pseudo header.
    ptr = tmp;
    total = 0;

    // Copy source address
    memcpy(ptr, &((SOCKADDR_IN6 *)&localif)-
>sin6_addr, sizeof(struct in6_addr));
    ptr   += sizeof(struct in6_addr);
    total += sizeof(struct in6_addr);
    // Copy destination address
    memcpy(ptr, &((SOCKADDR_IN6 *)dest->ai_addr)-
>sin6_addr, sizeof(struct in6_addr));
    ptr   += sizeof(struct in6_addr);
    total += sizeof(struct in6_addr);
    // Copy ICMP packet length
    length = htonl(icmplen);

    memcpy(ptr, &length, sizeof(length));
    ptr   += sizeof(length);
    total += sizeof(length);
    // Zero the 3 bytes
    memset(ptr, 0, 3);
    ptr   += 3;
    total += 3;
    // Copy next hop header
    proto = IPPROTO_ICMP6;

    memcpy(ptr, &proto, sizeof(proto));
    ptr   += sizeof(proto);
    total += sizeof(proto);
    // Copy the ICMP header and payload
    memcpy(ptr, icmppacket, icmplen);
    ptr   += icmplen;
    total += icmplen;
```

```
    for(i=0; i < icmplen%2 ;i++)
    {
        *ptr = 0;
        ptr++;
        total++;
    }
    return checksum((USHORT *)tmp, total);
}

// Function: ComputeIcmpChecksum
// Description:
//    This routine computes the checksum for the ICMP request.
For IPv4 its
//    easy, just compute the checksum for the ICMP packet and
data. For IPv6,
//    its more complicated. The pseudo checksum has to be
computed for IPv6
//    which includes the ICMP6 packet and data plus portions of
the IPv6
//    header which is difficult since we aren't building our own
IPv6 header.
void ComputeIcmpChecksum(SOCKET
s, char *buf, int packetlen, struct addrinfo *dest)
{
    if (gAddressFamily == AF_INET)
    {
        ICMP_HDR    *icmpv4=NULL;

        icmpv4 = (ICMP_HDR *)buf;
        icmpv4->icmp_checksum = 0;
        icmpv4->icmp_checksum = checksum((USHORT *)buf,
packetlen);
    }
    else if (gAddressFamily == AF_INET6)
    {
        ICMPV6_HDR  *icmpv6=NULL;

        icmpv6 = (ICMPV6_HDR *)buf;
        icmpv6->icmp6_checksum = 0;
        icmpv6->icmp6_checksum =
ComputeIcmp6PseudoHeaderChecksum(s, buf, packetlen, dest);
    }
}

// Function: PostRecvfrom
```

```
// Description: This routine posts an overlapped WSARecvFrom on
the raw socket.
int PostRecvfrom(SOCKET s, char *buf, int buflen,
SOCKADDR *from, int *fromlen, WSAOVERLAPPED *ol)
{
    WSABUF  wbuf;
    DWORD   flags, bytes;
    int     rc;

    wbuf.buf = buf;
    wbuf.len = buflen;

    flags = 0;

    rc = WSARecvFrom(s, &wbuf, 1, &bytes, &flags, from, fromlen,
ol, NULL);
    if (rc == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSA_IO_PENDING)
        {
            fprintf(stderr, "WSARecvFrom() failed with error
code %d\n", WSAGetLastError());
            return SOCKET_ERROR;
        }
    }
    else
            printf("WSARecvFrom() is OK!\n");

    return NO_ERROR;
}

// Function: AnalyzePacket
// Description:
//    This routines finds the ICMP packet within the
encapsulated header and
//    verifies that the ICMP packet is a TTL expired or echo
reply message. If not then an error is returned.
int AnalyzePacket(char *buf, int bytes)
{
    int     hdrlen=0, routes=0, rc;

    rc = NO_ERROR;
    if (gAddressFamily == AF_INET)
    {
        IPV4_HDR        *v4hdr=NULL;
```

```
        ICMP_HDR        *icmphdr=NULL;

        v4hdr = (IPV4_HDR *)buf;
        hdrlen = (v4hdr->ip_verlen & 0x0F) * 4;

        if (v4hdr->ip_protocol == IPPROTO_ICMP)
        {
            icmphdr = (ICMP_HDR *)&buf[hdrlen];

            if ((icmphdr->icmp_type != ICMPV4_TIMEOUT) &&
                    (icmphdr->icmp_type !=
ICMPV4_ECHO_REPLY_TYPE) &&
                    (icmphdr->icmp_code !=
ICMPV4_ECHO_REPLY_CODE) )
            {
                printf("Received ICMP message type %d instead of
TTL expired!\n", icmphdr->icmp_type);
                rc = SOCKET_ERROR;
            }
        }
    }
    else if (gAddressFamily == AF_INET6)
    {
        IPV6_HDR        *v6hdr=NULL;
        ICMPV6_HDR      *icmp6=NULL;

        v6hdr = (IPV6_HDR *)buf;

        if (v6hdr->ipv6_nexthdr == IPPROTO_ICMP6)
        {
            icmp6 = (ICMPV6_HDR *)&buf[sizeof(IPV6_HDR)];

            if ((icmp6->icmp6_type != ICMPV6_TIME_EXCEEDED_TYPE)
&&
                (icmp6->icmp6_code != ICMPV6_TIME_EXCEEDED_CODE)
&&
                (icmp6->icmp6_type != ICMPV6_ECHO_REPLY_TYPE) &&
                (icmp6->icmp6_code != ICMPV6_ECHO_REPLY_CODE) )
            {
                printf("Received ICMP6 message type %d instead
of TTL expired!\n", icmp6->icmp6_type);
                rc = SOCKET_ERROR;
            }
        }
    }
    return rc;
```

```
}

// Function: SetTtl
// Description: Sets the TTL on the socket.
int SetTtl(SOCKET s, int ttl)
{
    int     optlevel, option, rc;

    rc = NO_ERROR;
    if (gAddressFamily == AF_INET)
    {
        optlevel = IPPROTO_IP;
        option   = IP_TTL;
    }
    else if (gAddressFamily == AF_INET6)
    {
        optlevel = IPPROTO_IPV6;
        option   = IPV6_UNICAST_HOPS;
    }
    else
    {
        rc = SOCKET_ERROR;
    }

    if (rc == NO_ERROR)
    {
        rc = setsockopt(s, optlevel, option,
(char *)&ttl, sizeof(ttl));
    }

    if (rc == SOCKET_ERROR)
    {
        fprintf(stderr, "SetTtl(): setsockopt() failed with
error code %d\n", WSAGetLastError());
    }
    else
        printf("SetTtl(): setsockopt() should be fine!\n");

    return rc;
}

// Function: IsSockaddrEqual
// Description:
//    This routines compares two SOCKADDR structure to determine
```

```
//    whether the address portion of them are equal. Zero is
returned
//    for equal; non-zero for not equal.
int IsSockaddrEqual(SOCKADDR *sa1, SOCKADDR *sa2)
{
    int rc;

    rc = 1;
    if (sa1->sa_family == sa2->sa_family)
    {
        if (sa1->sa_family == AF_INET)
        {
            rc = memcmp(
                    &((SOCKADDR_IN *)sa1)->sin_addr,
                    &((SOCKADDR_IN *)sa2)->sin_addr,
                    sizeof(struct in_addr)
                    );
            rc = rc;
        }
        else if (sa1->sa_family == AF_INET6)
        {
            rc = memcmp(
                    &((SOCKADDR_IN6 *)sa1)->sin6_addr,
                    &((SOCKADDR_IN6 *)sa2)->sin6_addr,
                    sizeof(struct in6_addr)
                    );
            rc = rc;
        }
    }
    return rc;
}

// Function: main
// Description:
//    Setup the ICMP raw socket and create the ICMP header. Add
//    the appropriate IP option header and start sending ICMP
//    echo requests to the endpoint. For each send and receive
we
//    set a timeout value so that we don't wait forever for a
//    response in case the endpoint is not responding. When we
//    receive a packet decode it.
int main(int argc, char **argv)
{
    WSADATA            wsd;
    WSAOVERLAPPED      recvol;
    SOCKET             s=INVALID_SOCKET;
```

```
    char                 *icmpbuf=NULL, recvbuf[0xFFFF],
hopname[512];
    struct addrinfo   *dest=NULL, *local=NULL;
    SOCKADDR_STORAGE   from;
    DWORD              bytes, flags;
    int                packetlen=0, recvbuflen=0xFFFF,
hopbuflen=512,fromlen,notdone, time=0, ttl,rc;

    if(argc < 2)
    {
         usage(argv[0]);
         exit(1);
    }

    // Load Winsock
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("WSAStartup() failed with error code %d\n",
WSAGetLastError());
        return -1;
    }
    else
        printf("WSAStartup() is pretty fine!\n");

    // Parse the command line
    ValidateArgs(argc, argv);

    // Resolve the destination address
    dest = ResolveAddress(gDestination, "0", gAddressFamily, 0,
0);
    if (dest == NULL)
    {
        printf("Bad name %s\n", gDestination);
        return -1;
    }
    else
        printf("ResolveAddress() is OK!\n");

    gAddressFamily = dest->ai_family;

    if (gAddressFamily == AF_INET)
        gProtocol = IPPROTO_ICMP;
    else if (gAddressFamily == AF_INET6)
        gProtocol = IPPROTO_ICMP6;
```

```
    // Get the bind address
    local = ResolveAddress(NULL, "0", gAddressFamily, 0, 0);
    if (local == NULL)
    {
        printf("Unable to obtain the bind address!\n");
        return -1;
    }
    else
        printf("ResolveAddress() is fine!\n");

    // Create the raw socket
    s = socket(gAddressFamily, SOCK_RAW, gProtocol);
    if (s == INVALID_SOCKET)
    {
        printf("socket() failed with error code %d\n",
WSAGetLastError());
        return -1;
    }
    else
       printf("socket() is OK!\n");

    // Figure out the size of the ICMP header and payload
    if (gAddressFamily == AF_INET)
        packetlen += sizeof(ICMP_HDR);
    else if (gAddressFamily == AF_INET6)
        packetlen += sizeof(ICMPV6_HDR)
+ sizeof(ICMPV6_ECHO_REQUEST);

    // Add in the data size
    packetlen += DEFAULT_DATA_SIZE;

    // Allocate the buffer that will contain the ICMP request
    icmpbuf = (char *)HeapAlloc(GetProcessHeap(),
HEAP_ZERO_MEMORY, packetlen);
    if (icmpbuf == NULL)
    {
        fprintf(stderr, "HeapAlloc() for ICMP buffer failed with
error code %d\n", GetLastError());
        return -1;
    }
    else
        printf("HeapAlloc() for ICMP buffer is OK!\n");

    // Initialize the ICMP headers
    if (gAddressFamily == AF_INET)
    {
```

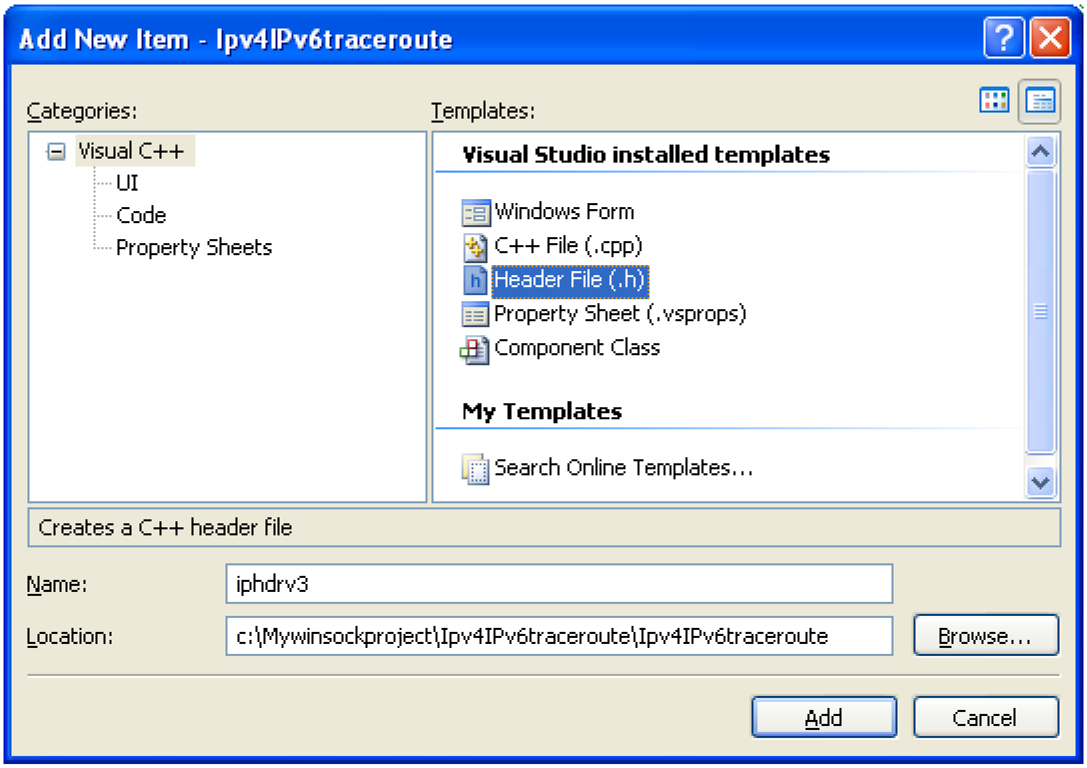
```
        InitIcmpHeader(icmpbuf, DEFAULT_DATA_SIZE);
    }
    else if (gAddressFamily == AF_INET6)
    {
        InitIcmp6Header(icmpbuf, DEFAULT_DATA_SIZE);
    }

    // Bind the socket -- need to do this since we post a
receive first
    rc = bind(s, local->ai_addr, local->ai_addrlen);
    if (rc == SOCKET_ERROR)
    {
        fprintf(stderr, "bind() failed with error code %d\n",
WSAGetLastError());
        return -1;
    }
    else
        printf("bind() is OK!\n");

    // Setup the receive operation
    memset(&recvol, 0, sizeof(recvol));
    recvol.hEvent = WSACreateEvent();

    // Post the first overlapped receive
    fromlen = sizeof(from);
    PostRecvfrom(s, recvbuf, recvbuflen, (SOCKADDR *)&from,
&fromlen, &recvol);

    printf("\nTraceroute to %s [", gDestination);
    PrintAddress(dest->ai_addr, dest->ai_addrlen);
    printf("]\nover a maximum of %d hops\n\n", gTtl);

    ttl = 1;

    // Start sending the ICMP requests
    do
    {
        notdone = 1;

        SetTtl(s, ttl);

        // Set the sequence number and compute the checksum
        SetIcmpSequence(icmpbuf);
        ComputeIcmpChecksum(s, icmpbuf, packetlen, dest);
        // Send the ICMP echo request
```

```
        time = GetTickCount();
        rc = sendto(s, icmpbuf, packetlen, 0, dest->ai_addr,
dest->ai_addrlen);
        if (rc == SOCKET_ERROR)
        {
            fprintf(stderr, "sendto() failed with error code
%d\n", WSAGetLastError());
            return -1;
        }
        else
            printf("sendto() is OK\n");

        // Wait for a response
        rc = WaitForSingleObject((HANDLE)recvol.hEvent,
gTimeout);
        if (rc == WAIT_FAILED)
        {
            fprintf(stderr, "WaitForSingleObject() failed with
error code %d\n", GetLastError());
            return -1;
        }
        else if (rc == WAIT_TIMEOUT)
        {
            printf("Request timed out.\n");
        }
        else
        {
            printf("WaitForSingleObject() should be fine
lol!\n");
            // Check for an error
            rc = WSAGetOverlappedResult(s, &recvol, &bytes,
FALSE, &flags);
            if (rc == FALSE)
            {
                fprintf(stderr, "WSAGetOverlappedResult() failed
with error code %d\n", WSAGetLastError());
            }
            else
                printf("WSAGetOverlappedResult() looks OK!\n");

            time = time - GetTickCount();
            WSAResetEvent(recvol.hEvent);
            // See if we got an ICMP ttl expired or echo reply,
if not ignore and receive again.
            if (AnalyzePacket(recvbuf, bytes) == NO_ERROR)
            {
                if (bResolve)
```

```
                {
                    if(ReverseLookup((SOCKADDR *)&from, fromlen,
hopname, hopbuflen) != NO_ERROR)
                    {
                            printf("ReverseLookup() failed
lor!\n");
                     }
                     else
                     {
                           printf("TTL:%d   Time:%d ms   Hop
name: %s [", ttl, time, hopname);
                           PrintAddress((SOCKADDR *)&from,
fromlen);
                           printf("]\n");
                      }
                }
                else
                {
                    printf("TTL:%d   Time:%d ms   ", ttl, time);
                    PrintAddress((SOCKADDR *)&from, fromlen);
                    printf("\n");
                }
                // See if the response is from the desired
destination
                notdone = IsSockaddrEqual(dest->ai_addr,
(SOCKADDR *)&from);
                // Increment the TTL
                ttl++;
            }
            // Post another receive
            if (notdone)
            {
                fromlen = sizeof(from);
                PostRecvfrom(s, recvbuf, recvbuflen, (SOCKADDR
*)&from, &fromlen, &recvol);
            }
        }
        Sleep(1000);
    } while ((notdone) && (ttl < gTtl));

    // Cleanup
    printf("Cleaning up stage...\n");
    freeaddrinfo(dest);
    freeaddrinfo(local);
    if (s != INVALID_SOCKET)
        closesocket(s);
    HeapFree(GetProcessHeap(), 0, icmpbuf);
```

```
    WSACleanup();
    return 0;
}
```

Add the iphdrv3.h header file.



Add the source code.

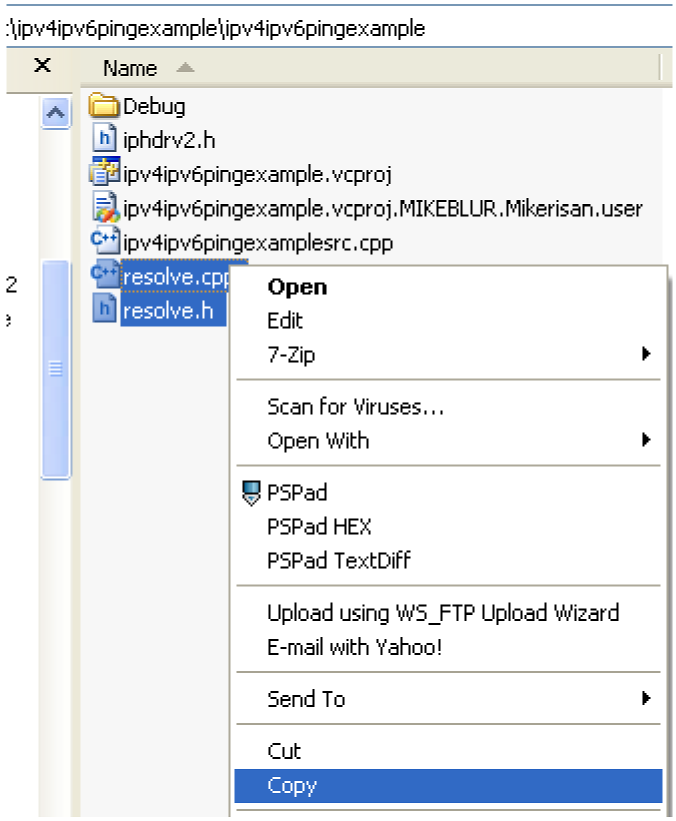
```
// Sample: Protocol header definitions used by trace route (ICMP
raw sockets)
// Files:
//      iphdrv3.h     - this file
// Description:
//      This routine contains protocol header structure
definitions used by
//      the trace route sample.
//
// Set alignment boundary to 1 byte
#include <pshpack1.h>

// IPv4 header
typedef struct ip_hdr
```

```
{
    unsigned char  ip_verlen;        // 4-bit IPv4 version, 4-
bit header length (in 32-bit words)
    unsigned char  ip_tos;           // IP type of service
    unsigned short ip_totallength;   // Total length
    unsigned short ip_id;            // Unique identifier
    unsigned short ip_offset;        // Fragment offset field
    unsigned char  ip_ttl;           // Time to live
    unsigned char  ip_protocol;      // Protocol(TCP,UDP etc)
    unsigned short ip_checksum;      // IP checksum
    unsigned int   ip_srcaddr;       // Source address
    unsigned int   ip_destaddr;      // Source address
} IPV4_HDR, *PIPV4_HDR, FAR * LPIPV4_HDR;

// IPv4 option header
typedef struct ipv4_option_hdr
{
    unsigned char   opt_code;           // option type
    unsigned char   opt_len;            // length of the option
header
    unsigned char   opt_ptr;            // offset into options
    unsigned long   opt_addr[9];        // list of IPv4
addresses
} IPV4_OPTION_HDR, *PIPV4_OPTION_HDR, FAR *LPIPV4_OPTION_HDR;

// ICMP header
typedef struct icmp_hdr
{
    unsigned char   icmp_type;
    unsigned char   icmp_code;
    unsigned short  icmp_checksum;
    unsigned short  icmp_id;
    unsigned short  icmp_sequence;
    unsigned long   icmp_timestamp;
} ICMP_HDR, *PICMP_HDR, FAR *LPICMP_HDR;

// IPv6 protocol header
typedef struct ipv6_hdr
{
    unsigned long   ipv6_vertcflow;         // 4-bit IPv6
version, 8-bit traffic class, 20-bit flow label
    unsigned short  ipv6_payloadlen;        // payload length
    unsigned char   ipv6_nexthdr;           // next header
protocol value
    unsigned char   ipv6_hoplimit;          // TTL
    struct in6_addr ipv6_srcaddr;           // Source address
```

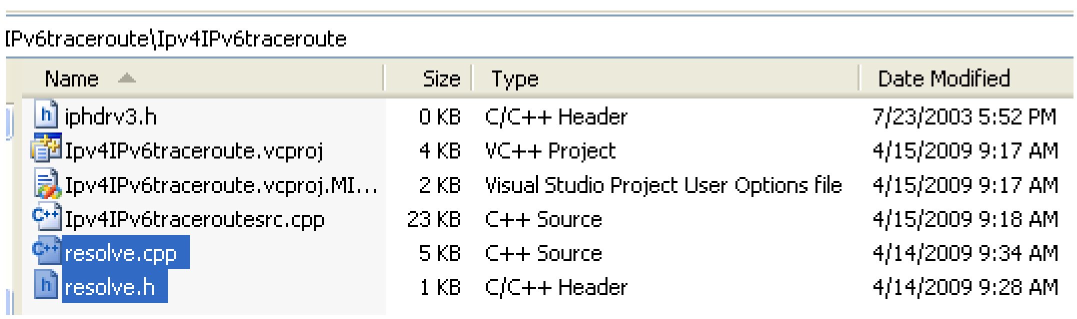
```
    struct in6_addr ipv6_destaddr;         // Destination
address
} IPV6_HDR, *PIPV6_HDR, FAR * LPIPV6_HDR;

// IPv6 fragment header
typedef struct ipv6_fragment_hdr
{
    unsigned char   ipv6_frag_nexthdr;
    unsigned char   ipv6_frag_reserved;
    unsigned short  ipv6_frag_offset;
    unsigned long   ipv6_frag_id;
} IPV6_FRAGMENT_HDR, *PIPV6_FRAGMENT_HDR, FAR *
LPIPV6_FRAGMENT_HDR;

// ICMPv6 header
typedef struct icmpv6_hdr {
    unsigned char   icmp6_type;
    unsigned char   icmp6_code;
    unsigned short  icmp6_checksum;
} ICMPV6_HDR;

// ICMPv6 echo request body
typedef struct icmpv6_echo_request
{
    unsigned short  icmp6_echo_id;
    unsigned short  icmp6_echo_sequence;
} ICMPV6_ECHO_REQUEST;

// Define the UDP header
typedef struct udp_hdr
{
    unsigned short src_portno;       // Source port no.
    unsigned short dst_portno;       // Dest. port no.
    unsigned short udp_length;       // Udp packet length
    unsigned short udp_checksum;     // Udp checksum (optional)
} UDP_HDR, *PUDP_HDR;

#define IP_RECORD_ROUTE      0x7

// ICMP6 protocol value
#define IPPROTO_ICMP6        58

// ICMP types and codes
#define ICMPV4_ECHO_REQUEST_TYPE   8
```

```
#define ICMPV4_ECHO_REQUEST_CODE   0
#define ICMPV4_ECHO_REPLY_TYPE     0
#define ICMPV4_ECHO_REPLY_CODE     0

#define ICMPV4_DESTUNREACH    3
#define ICMPV4_SRCQUENCH      4
#define ICMPV4_REDIRECT       5
#define ICMPV4_ECHO           8
#define ICMPV4_TIMEOUT       11
#define ICMPV4_PARMERR       12

// ICMP6 types and codes
#define ICMPV6_ECHO_REQUEST_TYPE   128
#define ICMPV6_ECHO_REQUEST_CODE   0
#define ICMPV6_ECHO_REPLY_TYPE     129
#define ICMPV6_ECHO_REPLY_CODE     0
#define ICMPV6_TIME_EXCEEDED_TYPE  3
#define ICMPV6_TIME_EXCEEDED_CODE  0

// Restore byte alignment to compile default
#include <poppack.h>
```

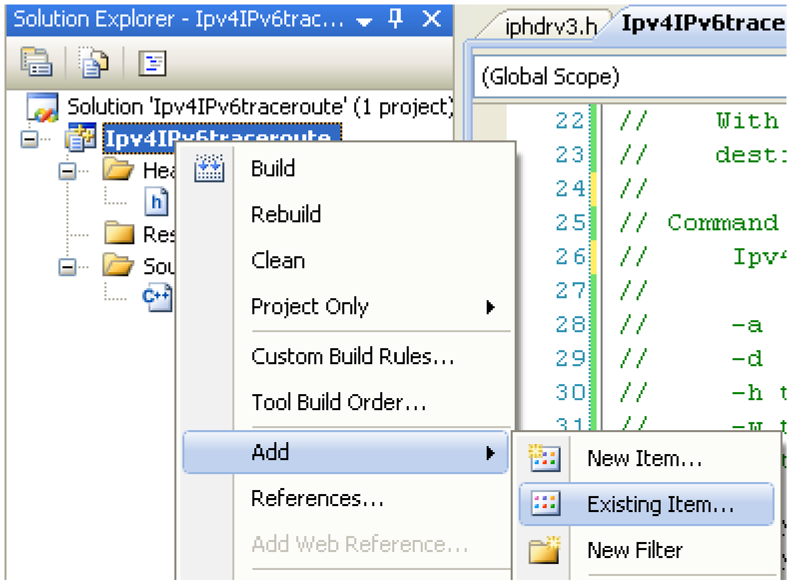
Add the resolve.h and its definition, resolve.cpp that were created in the previous program example.

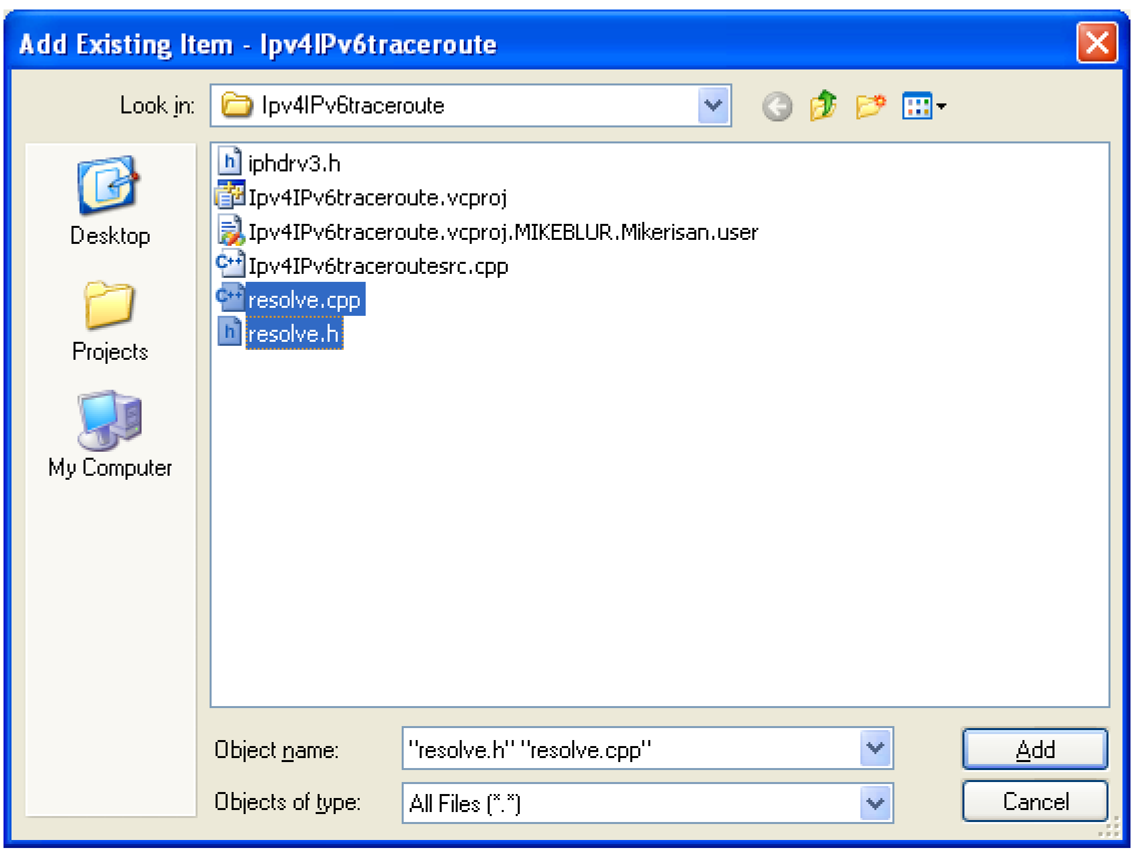


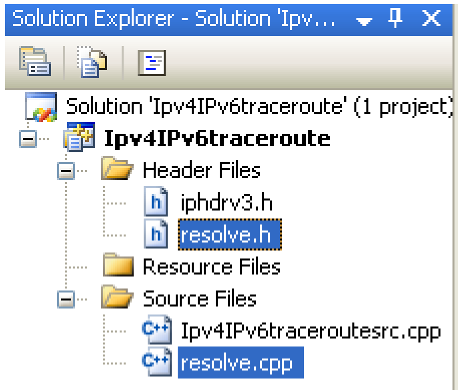
:\ipv4ipv6pingexample\ipv4ipv6pingexample
Name
Debug
iphdrv2.h
ipv4ipv6pingexample.vcproj
ipv4ipv6pingexample.vcproj.MIKEBLUR.Mikerisan.user
ipv4ipv6pingexamplesrc.cpp
resolve.cpp
resolve.h
Open
Edit
7-Zip
Scan for Viruses...
Open With
PSPad
PSPad HEX
PSPad TextDiff
Upload using WS_FTP Upload Wizard
E-mail with Yahoo!
Send To
Cut
Copy



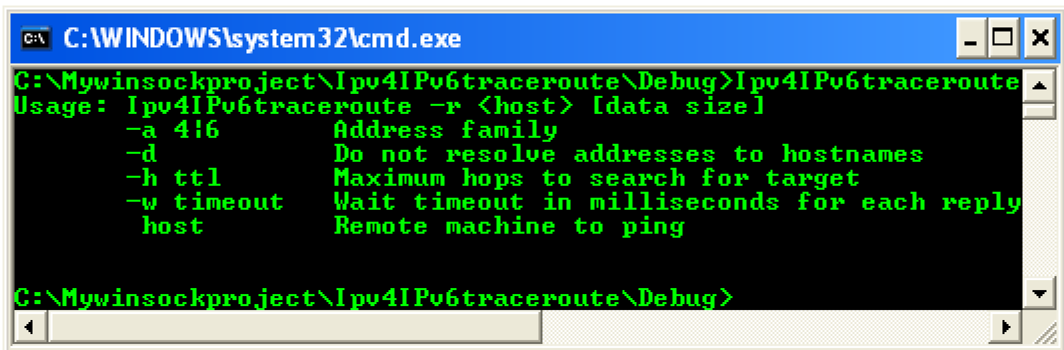
IPv6traceroute\Ipv4IPv6traceroute
Name
Size
Type
Date Modified
iphdrv3.h
0 KB
C/C++ Header
7/23/2003 5:52 PM
Ipv4IPv6traceroute.vcproj
4 KB
VC++ Project
4/15/2009 9:17 AM
Ipv4IPv6traceroute.vcproj.MI...
2 KB
Visual Studio Project User Options file
4/15/2009 9:17 AM
Ipv4IPv6traceroutesrc.cpp
23 KB
C++ Source
4/15/2009 9:18 AM
resolve.cpp
5 KB
C++ Source
4/14/2009 9:34 AM
resolve.h
1 KB
C/C++ Header
4/14/2009 9:28 AM


Solution Explorer - Ipv4IPv6trac...
Solution 'Ipv4IPv6traceroute' (1 project)
Build
Rebuild
Clean
Project Only
Custom Build Rules...
Tool Build Order...
Add
References...
Add Web Reference...
New Item...
Existing Item...
New Filter
iphdrv3.h
(Global Scope)

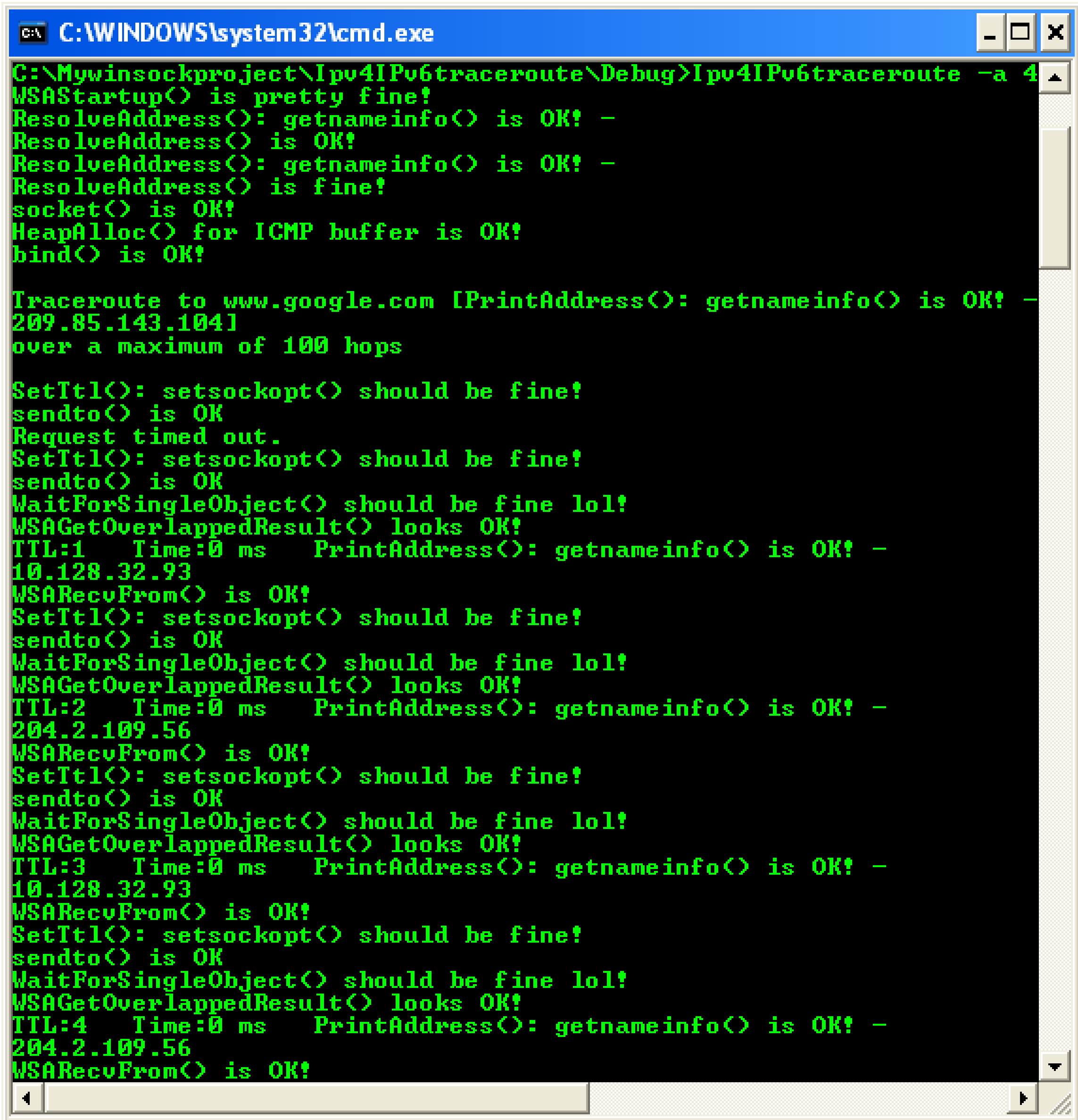

Add Existing Item - Ipv4IPv6traceroute
Look in: Ipv4IPv6traceroute
Desktop
Projects
My Computer
iphdrv3.h
Ipv4IPv6traceroute.vcproj
Ipv4IPv6traceroute.vcproj.MIKEBLUR.Mikerisan.user
Ipv4IPv6traceroutesrc.cpp
resolve.cpp
resolve.h
Object name: "resolve.h" "resolve.cpp"
Objects of type: All Files (*.*)
Add
Cancel



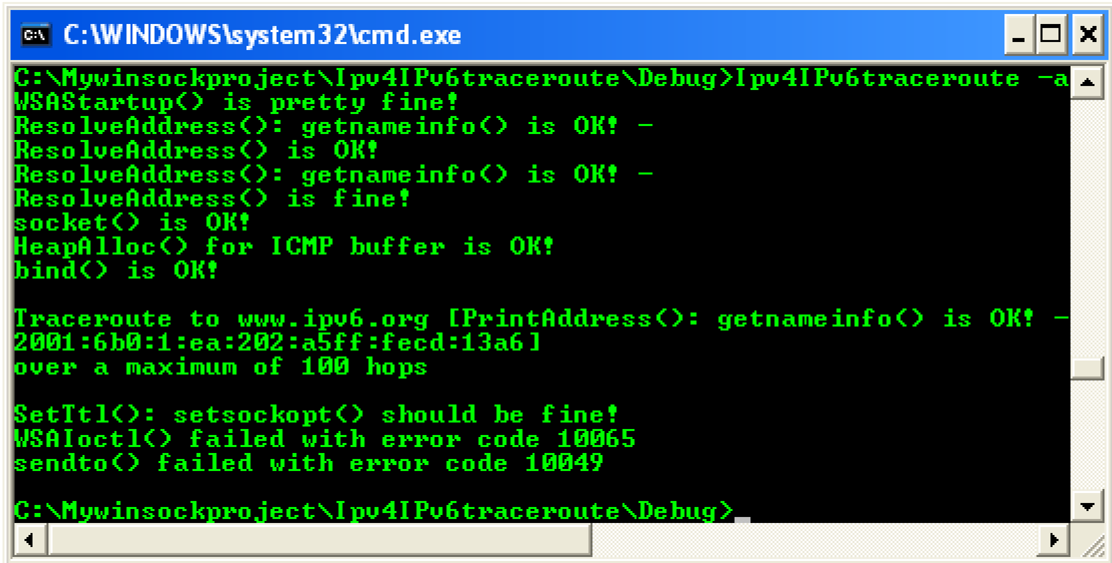
Build and run the project.



Ipv4IPv6traceroute -a 4 -d -h 100 -w 100 www.google.com



Ipv4IPv6traceroute -a 6 -d -h 100 -w 200 www.ipv6.org


C:\WINDOWS\system32\cmd.exe
C:\Mywinsockproject\Ipv4IPv6traceroute\Debug>Ipv4IPv6traceroute -a
WSAStartup() is pretty fine!
ResolveAddress(): getnameinfo() is OK! -
ResolveAddress() is OK!
ResolveAddress(): getnameinfo() is OK! -
ResolveAddress() is fine!
socket() is OK!
HeapAlloc() for ICMP buffer is OK!
bind() is OK!
Traceroute to www.ipv6.org [PrintAddress(): getnameinfo() is OK! -
2001:6b0:1:ea:202:a5ff:fecd:13a6]
over a maximum of 100 hops
SetTtl(): setsockopt() should be fine!
WSAIoctl() failed with error code 10065
sendto() failed with error code 10049
C:\Mywinsockproject\Ipv4IPv6traceroute\Debug>

Raw Sockets 11 Part 4

What do we have in this chapter 11 part 4?

7. Using IP Header Include Option

Using IP Header Include Option

The one limitation of raw sockets is that you can work only with certain protocols that are already defined, such as ICMP and IGMP. You cannot create a raw socket with IPPROTO_UDP and manipulate the UDP header; likewise with TCP. To manipulate the IP header as well as either the TCP or UDP header (or any other protocol encapsulated in IP), you must use the IP_HDRINCL socket option with a raw socket. For IPv6, the option is IPV6_HDRINCL. This option allows you to build your own IP header as well as other protocols' headers.

In addition to manipulating well-known protocols such as UDP, using raw sockets with the header include option allows you to implement your own protocol scheme that is encapsulated in IP. This is done by creating a raw socket and using the IPPROTO_RAW value as the protocol. This allows you to set the protocol field in the IP header manually and build your own custom protocol header. However, in this section we will take a look at how to build your own UDP packets so that you can gain a good understanding of the steps involved. Once you understand how to manipulate the UDP header, creating your own protocol header or manipulating other protocols encapsulated in IP is fairly trivial.

Before getting into the details of using the header include option, you need to know one important difference between using this option with IPv4 and IPv6. For IPv4, the stack still verifies some fields within the supplied IPv4 header. For example, the IPv4 identification field is set by the stack and the stack will fragment the packet if necessary. That is, if you create a raw IPv4 packet and set IP_HDRINCL and send a packet larger than the MTU size, the stack will fragment the data into multiple packets for you. For IPv6, if the IPV6_HDRINCL option is set, it is your responsibility to compute all the headers and fields necessary. If you submit a send larger than the MTU size, your application must create the

IPv6 fragment headers and compute the offsets correctly; otherwise, the IPv6 stack will drop the packet without sending it.
When you use the header include option, you are required to fill in the IP header yourself for every send call, as well as the headers of any other protocols wrapped within. The UDP header is quite a bit simpler than the IP header. It is only 8 bytes long and contains only four fields, as shown in Figure 11-3. The first two fields are the source and destination port numbers. They are 16 bits each. The third field is the UDP length, which is the length, in bytes, of the UDP header and data. The fourth field is the checksum, which we will discuss shortly. The last part of the UDP packet is the data.

16-bit source port	16-bit destination port
16-bit UDP length	16-bit UDP checksum

Figure 11-3 UDP header format

Because UDP is an unreliable protocol, calculating the checksum is optional. Unlike the IPv4 checksum, which covers only the IPv4 header, the UDP checksum covers the data and also includes part of the IPv4 header. The additional fields required to calculate the UDP checksum are known as a pseudo-header. The IPv4 UDP pseudo-header is composed of the following items:

1. 32-bit source IP address (IP header).
2. 32-bit destination IP address (IP header).
3. 8-bit field zeroed out.
4. 8-bit protocol.
5. 16-bit UDP length.

Added to these items are the UDP header and data. The method of calculating the checksum is the 16-bit one's complement sum. Because the data can be an odd number of bytes, it might be necessary to pad a zero byte to the end of the data to calculate the checksum. This pad field is not transmitted as part of the data. Figure 11-4 illustrates all of the fields required for the checksum calculation. The first three 32-bit words make up the UDP pseudo-header. The UDP header and its data follows that. Notice that because the checksum is calculated on 16-bit values, the data might need to be padded with a zero byte.

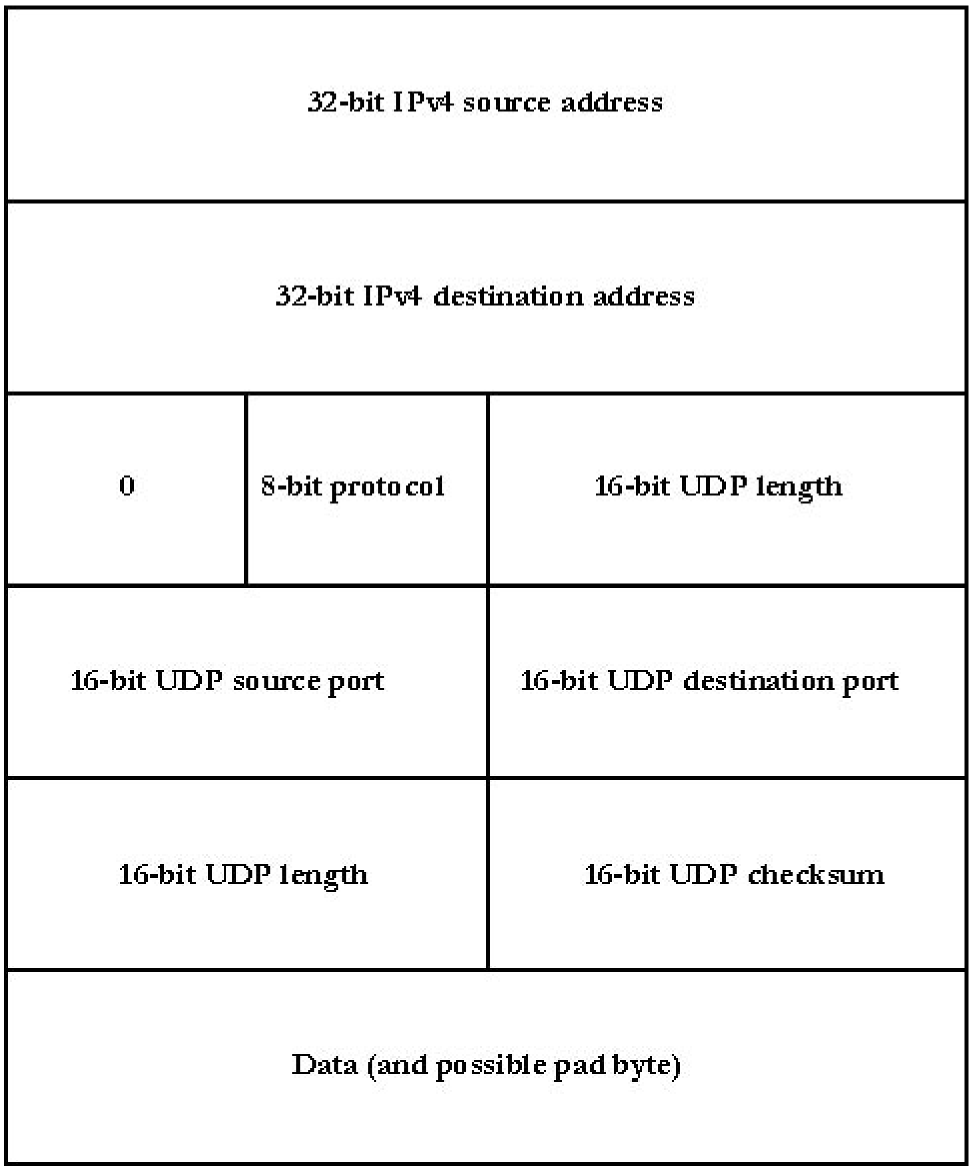


Figure 11-4 IPv4 pseudo-header with UDP packet and data

For IPv6, you have already seen how to calculate the IPv6 pseudo-header as is required to calculate the checksum for ICMPv6 packets. The calculation is the same for UDP with the IPv6 pseudo-header coming first and is followed by the UDP header and payload (zero padded to the next 16-bit boundary if necessary). The IPv6 pseudo-header is shown in Figure 11-5.

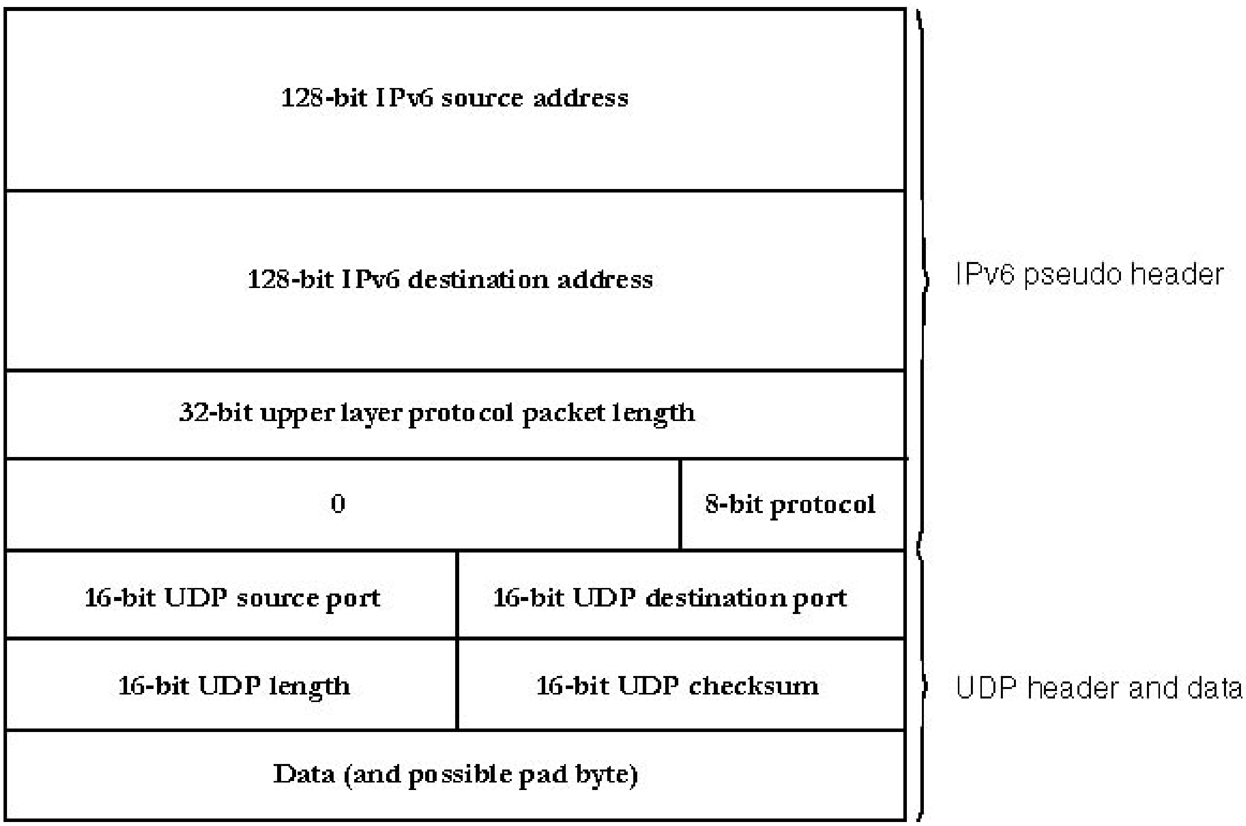


Figure 11-5 IPv6 pseudo-header with UDP packet and data

The following code snippet shows how to build an IPv4 and UDP header:

```
// Define the ICMP header
typedef struct icmp_hdr
{
    unsigned char   icmp_type;
    unsigned char   icmp_code;
    unsigned short  icmp_checksum;
    unsigned short  icmp_id;
    unsigned short  icmp_sequence;
    unsigned long   icmp_timestamp;
} ICMP_HDR, *PICMP_HDR, FAR *LPICMP_HDR;
```

```
ICMP_HDR *icmp=NULL;
SOCKET        s;
SOCKADDR_STORAGE dest;
char      buf[sizeof(ICMP_HDR) + 32];

// IPv4 header
typedef struct ip_hdr
{
    unsigned char  ip_verlen;        // 4-bit IPv4 version 4-bit
header length (in 32-bit words)
    unsigned char  ip_tos;           // IP type of service
    unsigned short ip_totallength;   // Total length
    unsigned short ip_id;            // Unique identifier
    unsigned short ip_offset;        // Fragment offset field
    unsigned char  ip_ttl;           // Time to live
    unsigned char  ip_protocol;      // Protocol(TCP,UDP etc)
    unsigned short ip_checksum;      // IP checksum
    unsigned int   ip_srcaddr;       // Source address
    unsigned int   ip_destaddr;      // Source address
} IPV4_HDR, *PIPV4_HDR, FAR * LPIPV4_HDR;

// Define the UDP header
typedef struct udp_hdr
{
    unsigned short src_portno;       // Source port no.
    unsigned short dst_portno;       // Dest. port no.
    unsigned short udp_length;       // Udp packet length
    unsigned short udp_checksum;     // Udp checksum (optional)
} UDP_HDR, *PUDP_HDR;

SOCKET    s;
char       buf[MAX_BUFFER], // large enough buffer
     *data=NULL;
IPV4_HDR *v4hdr=NULL;
UDP_HDR  *udphdr=NULL;
USHORT   sourceport=5000, Destport=5001;
int       payload=512,  // size of UDP data
         optval;
SOCKADDR_STORAGE dest;

// Initialize the IPv4 header
v4hdr = (IPV4_HDR *)buf;
v4hdr->ip_verlen = (4 << 4) | (sizeof(IPV4_HDR)
/ sizeof(ULONG));
v4hdr->ip_tos    = 0;
```

```
v4hdr->ip_totallength = htons(sizeof(IPV4_HDR) + sizeof(UDP_HDR)
+  payload);
v4hdr->ip_id     = 0;
v4hdr->ip_offset = 0;
v4hdr->ip_ttl    = 8;    // Time-to-live is eight
v4hdr->ip_protocol = IPPROTO_UDP;
v4hdr->ip_checksum = 0;
v4hdr->ip_srcaddr  = inet_addr("1.2.3.4");
v4hdr->ip_destaddr = inet_addr("157.32.159.101");
// Calculate checksum for IPv4 header
//   The checksum() function computes the 16-bit one's
//   complement on the specified buffer.
v4hdr->ip_checksum = checksum(v4hdr, sizeof(IPV4_HDR));

// Initialize the UDP header
udphdr = (UDP_HDR *)&buf[sizeof(IPV4_HDR)];
udphdr->src_portno = htons(sourceport);
udphdr->dst_portno = htons(destport);
udphdr->udp_length = htons(sizeof(UDP_HDR) + payload);
udphdr->udp_checksum = 0;

// Initialize the UDP payload to something
data = &buf[sizeof(IPV4_HDR) + sizeof(UDP_HDR)];
memset(data, '^', payload);




// Calculate the IPv4 and UDP pseudo-header checksum - this
routine
// extracts all the necessary fields from the headers and
calculates
// the checksum over it. See the iphdrinc sample for the
implementation
//    of Ipv4PseudoHeaderChecksum().
udphdr->udp_checksum = Ipv4PseudoHeaderChecksum(v4hdr, udphdr,
data, sizeof(IPV4_HDR) + sizeof(UDP_HDR) + payload);

// Create the raw UDP socket
s = socket(AF_INET, SOCK_RAW, IPPROTO_UDP);

// Set the header include option
optval = 1;
setsockopt(s, IPPROTO_IP, IP_HDRINCL,
(char *)&optval, sizeof(optval));
```

```
// Send the data
((SOCKADDR_IN *)&dest)->sin_family = AF_INET;
((SOCKADDR_IN *)&dest)->sin_port   = htons(destport);
((SOCKADDR_IN *)&dest)->sin_addr.s_addr =
inet_addr("157.32.159.101");

sendto(s, buf, sizeof(IPV4_HDR) + sizeof(UDP_HDR) +
payload, 0, (SOCKADDR *)&dest, sizeof(dest));
```

This code is straightforward and easy to follow. The IPv4 header is initialized with valid entries. In this case, a bogus source IPv4 address is used (1.2.3.4) but a valid destination address is supplied. Also, we set the TTL value to 8. Lastly, the checksum is calculated for the IPv4 header only. After the IPv4 header is the UDP header, as indicated by the ip_protocol field of the IPv4 header being set to IPPROTO_UDP. For that header, the source and destination ports are set in addition to the length of the UDP header and its payload. The last piece is to compute the pseudo-header checksum, which isn't shown but is an easy computation. The necessary fields are extracted out of the various headers after which the checksum can be computed.

The following program example creates raw UDP packets over IPv4 and IPv6. This sample also has a routine to compute the pseudo-header checksum for both IPv4 and IPv6.

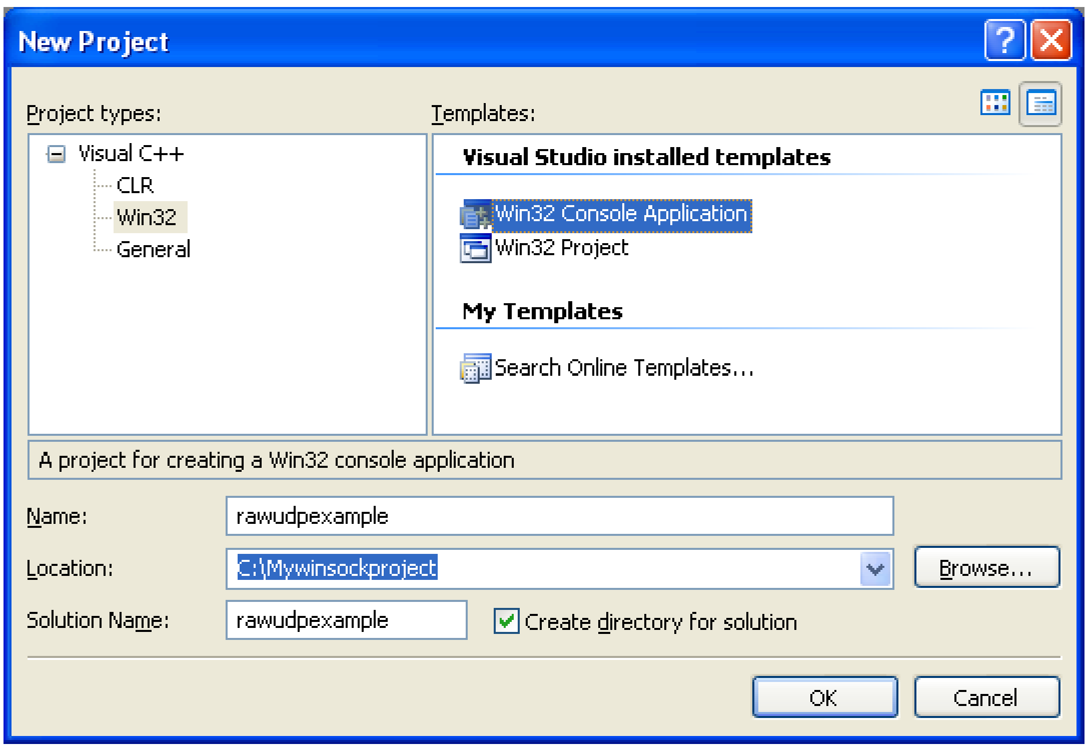
Raw Sockets 11 Part 5

What do we have in this chapter 11 part 5?

8. Program Examples: The UDP RAW Socket

Program Examples: The UDP RAW Socket

Create a new empty Win32 console mode application and add the project/solution name.



Add the following source code.

```
// Sample: Raw IPv4/IPv6 UDP with IP_HDRINCL option
//
// Files:
//      rawudpexamplesrc.cpp    - this file
//      iphdr.h       - IPv4, IPv6, and UDP header structure
definitions
//      resolve.cpp    - common name resolution routines
```

```
//      resolve.h     - header file for common name resolution
routines
//
// Description:
//    This is a simple app that demonstrates the usage of the
//    IP_HDRINCL socket option. A raw socket is created of the
//    UDP protocol where we will build our own IP and UDP header
//    that we submit to sendto().//
//    For IPv4 this is fairly simple. Create a raw socket, set
the
//    IP_HDRINCL option, build the IPv4 and UDP headers, and do
a
//    sendto. The IPv4 stack will fragment the data as necessary
and
//    generally leaves the packet unmodified -- it performs
fragmentation
//    and sets the IPv4 ID field.//
//    For IPv6 its a bit more involved as it does not perform
any
//    fragmentation, you have to do it and build the headers
yourself.//
//    The IP_HDRINCL option only works on Windows 2000 or
greater.
//
// Usage:
//      rawudpexample [options]
//          -a  4|6    Address family
//          -sa addr   From (sender) IP address
//          -sp int    From (sender) port number
//          -da addr   To (recipient) IP address
//          -dp int    To (recipient) port number
//          -n  int    Number of times to read message
//          -m  str    String message to fill packet data with
//          -p  proto  Protocol value
//          -r  port   Receive raw (SOCK_RAW) datagrams on the
given port
//          -rd port   Receive datagram (SOCK_DGRAM) on the
given port
//          -t  mtu    MTU size (required for fragmentation)
//          -z  int    Size of message to send
//
#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

// Link to ws2_32.lib
```

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

#include "resolve.h"
#include "iphdr.h"

// Setup some default values
#define DEFAULT_MTU             1496            // default
MTU size
#define DEFAULT_TTL             8               // default
TTL value

#define MAX_PACKET              65535           // maximum
datagram size
#define MAX_PACKET_FRAGMENTS ((MAX_PACKET / DEFAULT_MTU)
+ 1)

#define DEFAULT_PORT            5150            // default
port to send to
#define DEFAULT_COUNT           5               // default
number of messages to send
#define DEFAULT_MESSAGE         "This is a test
string!"  // default message

#define FRAGMENT_HEADER_PROTOCOL    44          // protocol
value for IPv6 fragmentation header

// Global variables
char     *gSrcAddress=NULL,             // IP address to
send from
         *gDestAddress=NULL,            // IP address to send to
         *gSrcPort=NULL,                      // port to send
from
         *gDestPort=NULL,                     // port to send to
         *gMessage=NULL;                    // Message to send as
UDP payload
int       gAddressFamily=AF_UNSPEC,
          gSocketType=SOCK_DGRAM,     // Socket type to pass to
name resolution routines
```

```
          gProtocol=IPPROTO_UDP,     // Protocol value that
we're sending
          gSendSize=0,                             // Data size
of message to send
          gMtuSize=DEFAULT_MTU;    // Maximum transmission unit
to use
DWORD     gSendCount;                    // number of times to
send
BOOL      bSender=TRUE,                 // sending or receiving
data
          bReadRaw=TRUE;                   // Use raw sockets
when reading

// Function: usage:
// Description: Print usage information and exit.
int usage(char *progname)
{
    printf("Usage: %s [-fp int] [-fi str] [-tp int] [-ti str] [-
n int] [-m str]\n"
           "    -a  4|6    Address family\n"
           "    -sa addr   From (sender) IP address\n"
           "    -sp int    From (sender) port number\n"
           "    -da addr   To (recipient) IP address\n"
           "    -dp int    To (recipient) port number\n"
           "    -n  int    Number of times to read message\n"
           "    -m  str    String message to fill packet data
with\n"
           "    -p  proto  Protocol value\n"
           "    -r  port   Receive raw (SOCK_RAW) datagrams on
the given port\n"
           "    -rd port   Receive datagram (SOCK_DGRAM) on the
given port\n"
           "    -t  mtu    MTU size (required for
fragmentation)\n"
           "    -z  int    Size of message to send\n",
           progname
           );
    return 0;
}

// Function: ValidateArgs
// Description:
//    Parse the command line arguments and set some global flags
to indicate what actions to perform.
void ValidateArgs(int argc, char **argv)
{
```

```
int     i;

gMessage = DEFAULT_MESSAGE;
for(i=1; i < argc ;i++)
{
    if ((argv[i][0] == '-') || (argv[i][0] == '/'))
    {
        switch (tolower(argv[i][1]))
        {
            printf("i = %d\n", i);
            case 'a':       // Address family
                if (i+1 > argc)
                    usage(argv[0]);
                if (argv[i+1][0] == '4')
                    gAddressFamily = AF_INET;
                else if (argv[i+1][0] == '6')
                    gAddressFamily = AF_INET6;
                else
                    usage(argv[0]);
                i++;
                break;
            case 's':       // source address
                if (i+1 > argc)
                {
                    printf("i + 1 = %d\n", i + 1);
                    usage(argv[0]);
                }
                if (tolower(argv[i][2]) == 'a')
                {
                    gSrcAddress = argv[++i];
                }
                else if (tolower(argv[i][2]) == 'p')
                {
                    gSrcPort = argv[++i];
                }
                else
                {
                    usage(argv[0]);
                    break;
                }
                break;
            case 'd':       // destination address
                if (i+1 > argc)
                {
                    printf("i + 1 = %d\n", i + 1);
                    usage(argv[0]);
                }
```

```
                    if (tolower(argv[i][2]) == 'a')
                    {
                        gDestAddress = argv[++i];
                    }
                    else if (tolower(argv[i][2]) == 'p')
                    {
                        gDestPort = argv[++i];
                    }
                    else
                    {
                        usage(argv[0]);
                        break;
                    }
                    break;
                case 'n':       // number of times to send
message
                    if (i+1 >= argc)
                        usage(argv[0]);
                    gSendCount = atol(argv[++i]);
                    break;      // String message to copy into
payload
                case 'm':
                    if (i+1 >= argc)
                        usage(argv[0]);
                    gMessage = argv[++i];
                    break;
                case 'p':       // Protocol value
                    if (i+1 >= argc)
                        usage(argv[0]);
                    gProtocol = atoi(argv[++i]);
                    break;
                case 'r':        // Port to receive data on
                    if (i+1 >= argc)
                        usage(argv[0]);
                    if (strlen(argv[i]) == 3)
                        bReadRaw = FALSE;
                    gSrcPort = argv[++i];
                    bSender = FALSE;
                    break;
                case 't':        // MTU size
                    if (i+1 >= argc)
                        usage(argv[0]);
                    gMtuSize = atoi(argv[++i]);
                    break;
                case 'z':       // Send size
                    if (i+1 >= argc)
                        usage(argv[0]);
```

```
                    gSendSize = atoi(argv[++i]);
                    break;
                default:
                    usage(argv[0]);
                    break;
            }
        }
    }
    // If no data size was given, initialize it to the message
supplied
    if (gSendSize == 0)
    {
        gSendSize = strlen(gMessage);
    }
    return;
}

// Function: checksum
// Description: This function calculates the 16-bit one's
complement sum for the supplied buffer.
USHORT checksum(USHORT *buffer, int size)
{
    unsigned long cksum=0;

    while (size > 1)
    {
        cksum += *buffer++;
        size  -= sizeof(USHORT);
    }
    if (size)
    {
        cksum += *(UCHAR*)buffer;
    }
    cksum = (cksum >> 16) + (cksum & 0xffff);
    cksum += (cksum >>16);

    return (USHORT)(~cksum);
}

// Function: InitIpv4Header
// Description:
//    Initialize the IPv4 header with the version, header
length,
//    total length, ttl, protocol value, and source and
destination addresses
```

```
int InitIpv4Header(char *buf, SOCKADDR *src, SOCKADDR
*dest, int ttl, int proto, int payloadlen)
{
    IPV4_HDR    *v4hdr=NULL;

    v4hdr = (IPV4_HDR *)buf;

    v4hdr->ip_verlen      = (4 << 4) | (sizeof(IPV4_HDR)
/ sizeof(unsigned long));
    v4hdr->ip_tos         = 0;
    v4hdr->ip_totallength = htons(sizeof(IPV4_HDR) +
payloadlen);
    v4hdr->ip_id          = 0;
    v4hdr->ip_offset      = 0;
    v4hdr->ip_ttl         = (unsigned char)ttl;
    v4hdr->ip_protocol    = (unsigned char)proto;
    v4hdr->ip_checksum    = 0;
    v4hdr->ip_srcaddr     = ((SOCKADDR_IN *)src)-
>sin_addr.s_addr;
    v4hdr->ip_destaddr    = ((SOCKADDR_IN *)dest)-
>sin_addr.s_addr;

    v4hdr->ip_checksum    =
checksum((unsigned short *)v4hdr, sizeof(IPV4_HDR));

    return sizeof(IPV4_HDR);
}

// Function: InitIpv6Header
// Description:
//    Initialize the IPv6 header with the version, payload
length, next
//    hop protocol, TTL, and source and destination addresses.
int InitIpv6Header(char *buf, SOCKADDR *src, SOCKADDR
*dest, int ttl, int proto, int payloadlen)
{
    IPV6_HDR    *v6hdr=NULL;

    v6hdr = (IPV6_HDR *)buf;

    // We don't explicitly set the traffic class or flow label
fields
    v6hdr->ipv6_vertcflow    = htonl(6 << 28);
    v6hdr->ipv6_payloadlen   =
htons((unsigned short)payloadlen);
```

```
    v6hdr->ipv6_nexthdr      = (unsigned char)proto;
    v6hdr->ipv6_hoplimit     = (unsigned char)ttl;
    v6hdr->ipv6_srcaddr      = ((SOCKADDR_IN6 *)src)->sin6_addr;
    v6hdr->ipv6_destaddr     = ((SOCKADDR_IN6 *)dest)-
>sin6_addr;

    return sizeof(IPV6_HDR);
}

// Function: InitIpv6FragmentHeader
// Description:
//    Initialize the IPv6 fragmentation header. The offset is
the offset
//    from the start of the IPv6 total payload (which includes
the UDP
//    header along with the data) which is why we add the length
of
//    the UDP header if this fragment is not the first fragment.
Also,
//    the lastfragment parameter is a boolean value (0 == not
the last
//    fragment while 1 == this is the last fragment) which is
the opposite
//    value that is supposed to be indicated in the header (i.e.
0 indicates
//    that this fragment is the last fragment).
int InitIpv6FragmentHeader(char *buf, unsigned long offs
et, int nextproto, int id, int lastfragment)
{
    IPV6_FRAGMENT_HDR *frag=NULL;

    frag = (IPV6_FRAGMENT_HDR *)buf;

    // Swap the value of this field
    lastfragment = (lastfragment ? 0 : 1);

    // Account for the size of the UDP header
    if (offset != 0)
        offset += sizeof(UDP_HDR);

    frag->ipv6_frag_nexthdr = (unsigned char)nextproto;
    frag->ipv6_frag_offset  = htons(
(unsigned short)(((offset/8) << 3) | lastfragment));
    frag->ipv6_frag_id      = htonl(id);
```

```
    return sizeof(IPV6_FRAGMENT_HDR);
}

// Function: InitUdpHeader
// Description:
//    Setup the UDP header which is fairly simple. Grab the
ports and
//    stick in the total payload length.
int InitUdpHeader(char *buf, SOCKADDR *src, SOCKADDR
*dest, int payloadlen)
{
    UDP_HDR *udphdr=NULL;

    udphdr = (UDP_HDR *)buf;

    // Port numbers are already in network byte order
    if (src->sa_family == AF_INET)
    {
        udphdr->src_portno = ((SOCKADDR_IN *)src)->sin_port;
        udphdr->dst_portno = ((SOCKADDR_IN *)dest)->sin_port;
    }
    else if (src->sa_family == AF_INET6)
    {
        udphdr->src_portno = ((SOCKADDR_IN6 *)src)->sin6_port;
        udphdr->dst_portno = ((SOCKADDR_IN6 *)dest)->sin6_port;
    }
    udphdr->udp_length = htons(sizeof(UDP_HDR) + payloadlen);

    return sizeof(UDP_HDR);
}
// Function: ComputeUdpPseudoHeaderChecksumV4
// Description:
//    Compute the UDP pseudo header checksum. The UDP checksum
is based
//    on the following fields:
//       - source IP address
//       - destination IP address
//       - 8-bit zero field
//       - 8-bit protocol field
//       - 16-bit UDP length
//       - 16-bit source port
//       - 16-bit destination port
//       - 16-bit UDP packet length
//       - 16-bit UDP checksum (zero)
//       - UDP payload (padded to the next 16-bit boundary)
```

```
//    This routine copies these fields to a temporary buffer and
computes
//    the checksum from that.
void ComputeUdpPseudoHeaderChecksumV4(void *iphdr,
UDP_HDR *udphdr, char *payload, int payloadlen)
{
    IPV4_HDR     *v4hdr=NULL;
    unsigned long zero=0;
    char          buf[MAX_PACKET], *ptr=NULL;
    int           chksumlen=0, i;

    ptr = buf;
    v4hdr = (IPV4_HDR *)iphdr;
    // Include the source and destination IP addresses
    memcpy(ptr, &v4hdr->ip_srcaddr,  sizeof(v4hdr->ip_srcaddr));
    ptr += sizeof(v4hdr->ip_srcaddr);
    chksumlen += sizeof(v4hdr->ip_srcaddr);

    memcpy(ptr, &v4hdr->ip_destaddr, sizeof(v4hdr-
>ip_destaddr));
    ptr += sizeof(v4hdr->ip_destaddr);
    chksumlen += sizeof(v4hdr->ip_destaddr);
    // Include the 8 bit zero field
    memcpy(ptr, &zero, 1);
    ptr++;
    chksumlen += 1;
    // Protocol
    memcpy(ptr, &v4hdr->ip_protocol, sizeof(v4hdr-
>ip_protocol));
    ptr += sizeof(v4hdr->ip_protocol);
    chksumlen += sizeof(v4hdr->ip_protocol);
    // UDP length
    memcpy(ptr, &udphdr->udp_length, sizeof(udphdr-
>udp_length));
    ptr += sizeof(udphdr->udp_length);
    chksumlen += sizeof(udphdr->udp_length);
    // UDP source port
    memcpy(ptr, &udphdr->src_portno, sizeof(udphdr-
>src_portno));
    ptr += sizeof(udphdr->src_portno);
    chksumlen += sizeof(udphdr->src_portno);
    // UDP destination port
    memcpy(ptr, &udphdr->dst_portno, sizeof(udphdr-
>dst_portno));
    ptr += sizeof(udphdr->dst_portno);
    chksumlen += sizeof(udphdr->dst_portno);
```

```
    // UDP length again
    memcpy(ptr, &udphdr->udp_length, sizeof(udphdr-
>udp_length));
    ptr += sizeof(udphdr->udp_length);
    chksumlen += sizeof(udphdr->udp_length);
    // 16-bit UDP checksum, zero
    memcpy(ptr, &zero, sizeof(unsigned short));
    ptr += sizeof(unsigned short);
    chksumlen += sizeof(unsigned short);
    // payload
    memcpy(ptr, payload, payloadlen);
    ptr += payloadlen;
    chksumlen += payloadlen;
    // pad to next 16-bit boundary
    for(i=0 ; i < payloadlen%2 ; i++, ptr++)
    {
        printf("pad one byte\n");
        *ptr = 0;
        ptr++;
        chksumlen++;
    }

    // Compute the checksum and put it in the UDP header
    udphdr->udp_checksum = checksum((USHORT *)buf, chksumlen);

    return;
}

void ComputeUdpPseudoHeaderChecksumV6(void *iphdr,
UDP_HDR *udphdr, char *payload, int payloadlen)
{
    IPV6_HDR     *v6hdr=NULL;
    unsigned long length=0;
    char          buf[MAX_PACKET], proto, *ptr=NULL;
    int           chksumlen=0, i;

    ptr = buf;
    v6hdr = (IPV6_HDR *)iphdr;

    memcpy(ptr, &v6hdr->ipv6_srcaddr,  sizeof(v6hdr-
>ipv6_srcaddr));
    ptr += sizeof(v6hdr->ipv6_srcaddr);
    chksumlen += sizeof(v6hdr->ipv6_srcaddr);
```

```
    memcpy(ptr, &v6hdr->ipv6_destaddr,  sizeof(v6hdr-
>ipv6_destaddr));
    ptr += sizeof(v6hdr->ipv6_destaddr);
    chksumlen += sizeof(v6hdr->ipv6_destaddr);

    printf("payload length = %d\n", payloadlen);
    length = htonl(payloadlen + sizeof(UDP_HDR));

    memcpy(ptr, &length, sizeof(length));
    ptr += sizeof(length);
    chksumlen += sizeof(length);

    memset(ptr, 0, 3);
    ptr += 3;
    chksumlen +=3;

    proto = IPPROTO_UDP;

    memcpy(ptr, &proto, sizeof(proto));
    ptr += sizeof(proto);
    chksumlen += sizeof(proto);

    // UDP source port
    memcpy(ptr, &udphdr->src_portno, sizeof(udphdr-
>src_portno));
    ptr += sizeof(udphdr->src_portno);
    chksumlen += sizeof(udphdr->src_portno);
    // UDP destination port
    memcpy(ptr, &udphdr->dst_portno, sizeof(udphdr-
>dst_portno));
    ptr += sizeof(udphdr->dst_portno);
    chksumlen += sizeof(udphdr->dst_portno);
    // UDP length again
    memcpy(ptr, &udphdr->udp_length, sizeof(udphdr-
>udp_length));
    ptr += sizeof(udphdr->udp_length);
    chksumlen += sizeof(udphdr->udp_length);
    // 16-bit UDP checksum, zero
    memset(ptr, 0, sizeof(unsigned short));
    ptr += sizeof(unsigned short);
    chksumlen += sizeof(unsigned short);
    // payload
    memcpy(ptr, payload, payloadlen);
    ptr += payloadlen;
    chksumlen += payloadlen;
    // pad to next 16-bit boundary
```

```
    for(i=0 ; i < payloadlen%2 ; i++, ptr++)
    {
        printf("pad one byte\n");
        *ptr = 0;
        ptr++;
        chksumlen++;
    }
    // Compute the checksum and put it in the UDP header
    udphdr->udp_checksum = checksum((USHORT *)buf, chksumlen);

    return;
}

// Function: memfill
// Description: Fills a block of memory with a given string
pattern.
void memfill(char *dest, int   destlen, char *data, int
  datalen)
{
    char *ptr=NULL;
    int   copylen;

    ptr = dest;
    while (destlen > 0)
    {
        copylen = ((destlen > datalen) ? datalen : destlen);
        memcpy(ptr, data, copylen);

        destlen -= copylen;
        ptr += copylen;
    }
    return;
}

// Function: PacketizeIpv4
// Description:
//    This routine takes the data buffer and packetizes it for
IPv4.
//    Since the IPv4 stack takes care of fragmentation for us,
this
//    routine simply initializes the IPv4 and UDP headers. The
data
//    is returned in an array of WSABUF structures.
WSABUF *PacketizeIpv4(struct addrinfo *src, struct addrinfo
*dest, char *payload, int payloadlen)
{
```

```
    static WSABUF Packets[MAX_PACKET_FRAGMENTS];
    int           iphdrlen, udphdrlen;

    // Allocate memory for the packet
    Packets[0].buf = (char *)HeapAlloc(GetProcessHeap(),
HEAP_ZERO_MEMORY, sizeof(IPV4_HDR) + sizeof(UDP_HDR) +
payloadlen);
    if (Packets[0].buf == NULL)
    {
        fprintf(stderr, "PacetizeV4: HeapAlloc failed: %d\n",
GetLastError());
        ExitProcess(-1);
    }
    Packets[0].len = sizeof(IPV4_HDR) + sizeof(UDP_HDR) +
payloadlen;
    // Initialize the v4 header
    iphdrlen = InitIpv4Header(Packets[0].buf, src->ai_addr,
dest->ai_addr, DEFAULT_TTL, gProtocol, payloadlen);
    // Initialize the UDP header
    udphdrlen = InitUdpHeader(&Packets[0].buf[iphdrlen], src-
>ai_addr, dest->ai_addr, payloadlen);
    // Compute the UDP checksum
    ComputeUdpPseudoHeaderChecksumV4(Packets[0].buf,(UDP_HDR
*)&Packets[0].buf[iphdrlen],payload,payloadlen);
    // Copy the payload to the end of the header
    memcpy(&Packets[0].buf[iphdrlen + udphdrlen], payload,
payloadlen);
    // Zero out the next WSABUF structure which indicates the
end of
    //    the packets -- caller must free the buffers
    Packets[1].buf = NULL;
    Packets[1].len = 0;

    return Packets;
}




// Function: PacketizeIpv6
// Description:
//    This routine fragments data payload with the appropriate
IPv6
//    headers. The individual fragments are returned via an
array of
```

```
//    WSABUF structures. Each structure is a separate fragment
of the
//    whole message. The end of the fragments is indicated by a
WSABUF
//    entry with a NULL buffer pointer.
WSABUF *PacketizeIpv6(struct addrinfo *src, struct addrinfo
*dest, char *payload, int payloadlen)
{
    static WSABUF Packets[MAX_PACKET_FRAGMENTS];
    static ULONG  fragid=1;
    int           offset=0,           // offset into payload
                  datalen,             // length of the payload
                  hdrlen,               // length of the
header(s)
                  fragment,           // is this a fragment?
                  lastfragment,    // is this the last fragment?
                  iphdrlen,           // length of ip header
                  udphdrlen,        // length of the udp header
                  plushdrs,           // IPv6 length field
includes encapsulated headers
                  numpackets=0, // number of fragments
                  originalpayload;

    originalpayload = payloadlen;
    do
    {
        // Compute the size of this fragment
        lastfragment = 0;
        fragment = 0;
        if ((payloadlen > gMtuSize) && (numpackets == 0))
        {
            // Data needs to be fragmented, this is the first
packet
            hdrlen  = sizeof(IPV6_HDR) + sizeof(UDP_HDR)
+ sizeof(IPV6_FRAGMENT_HDR);
            datalen = gMtuSize - hdrlen;
            plushdrs = sizeof(UDP_HDR)
+ sizeof(IPV6_FRAGMENT_HDR);
            fragment = 1;

            printf("Require fragmentation: FIRST packet\n");
        }
        else if ((payloadlen > gMtuSize) && (numpackets > 0))
        {
            // Data needs to be fragmented, this is packet
number > 0
```

```
            hdrlen = sizeof(IPV6_HDR)
+ sizeof(IPV6_FRAGMENT_HDR);
            datalen = gMtuSize - hdrlen;
            fragment = 1;
            plushdrs = sizeof(IPV6_FRAGMENT_HDR);

            printf("Require fragmentation: packet number >
0\n");
        }
        else if (numpackets == 0)
        {
            // Data doesn't need to be fragmented
            hdrlen = sizeof(IPV6_HDR) + sizeof(UDP_HDR);
            datalen = payloadlen;
            fragment = 0;
            plushdrs = sizeof(UDP_HDR);

            printf("No fragmentation required\n");
        }
        else
        {
            // This is the last fragment
            hdrlen = sizeof(IPV6_HDR)
+ sizeof(IPV6_FRAGMENT_HDR);
            datalen = payloadlen;
            fragment = 1;
            plushdrs = sizeof(IPV6_FRAGMENT_HDR);
            lastfragment = 1;
            printf("Require fragmentation: Last packet\n");
        }
        // Build packet
        // Allocate buffer for this fragment
        Packets[numpackets].buf =
(char *)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, hdrlen +
datalen);
        if (Packets[numpackets].buf == NULL)
        {
            fprintf(stderr, "PacketizeV6: HeapAlloc failed:
%d\n", GetLastError());
            ExitProcess(-1);
        }
        Packets[numpackets].len = hdrlen + datalen;

        // Initialize the V6 header, if we have to fragment the
next header field of
```

```
        //    the v6 header is that of the fragmentation header.
Also the payload
        //    length includes the headers (UDP + fragmentation)
and the payload itself.
        iphdrlen = InitIpv6Header(
                Packets[numpackets].buf,
                src->ai_addr,
                dest->ai_addr,
                DEFAULT_TTL,
                (fragment ? FRAGMENT_HEADER_PROTOCOL :
gProtocol),
                datalen + plushdrs
                );

        // Build the fragmentation header if necessary
        if (fragment)
        {
            iphdrlen += InitIpv6FragmentHeader(
                   &Packets[numpackets].buf[iphdrlen],
                    offset,         // offset from start of
packet
                    gProtocol,
                    fragid,
                    lastfragment
                    );
        }

        // The first fragment includes the UDP header,
subsequent fragments don't
        if (numpackets == 0)
        {
            udphdrlen = InitUdpHeader(
                   &Packets[numpackets].buf[iphdrlen],
                    src->ai_addr,
                    dest->ai_addr,
                    originalpayload // payloadlen
                    );
            // Compute the checksum
            ComputeUdpPseudoHeaderChecksumV6(
                    Packets[numpackets].buf,
                    (UDP_HDR
*)&Packets[numpackets].buf[iphdrlen],
                    payload,
                    payloadlen);
        }
        else
        {
```

```
            udphdrlen = 0;
        }
        // Copy the payload into this fragment
        memcpy(&Packets[numpackets].buf[iphdrlen + udphdrlen],
&payload[offset], datalen);
        // Adjust our counters
        payloadlen = payloadlen - datalen;
        offset += datalen;
        numpackets++;
    } while (payloadlen > 0);

    fragid++;

    // Mark the next WSABUF entry with NULL and zero to indicate
end of fragments
    Packets[numpackets].buf = NULL;
    Packets[numpackets].len = 0;

    return Packets;
}

// Function: main
// Description:
//    First, parse command line arguments and load Winsock. Then
//    create the raw socket and then set the IP_HDRINCL option.
//    Following this assemble the IP and UDP packet headers by
//    assigning the correct values and calculating the
checksums.
//    Then fill in the data and send to its destination.
int main(int argc, char **argv)
{
    WSADATA            wsd;
    SOCKET             s;
    DWORD              bytes;
    WSABUF            *wbuf=NULL;
    struct addrinfo   *ressrc=NULL, *resdest=NULL,
*resbind=NULL;
    int                packets, rc,i, j;

    if(argc < 2)
    {
        usage(argv[0]);
        exit(1);
    }
```

```
    // Parse command line arguments and print them out
    ValidateArgs(argc, argv);

    srand(GetTickCount());

    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("WSAStartup() failed with error code %d\n",
GetLastError());
        return -1;
    }
    else
        printf("WSAStartup() is OK!\n");
    // Convert the source and destination addresses/ports
    ressrc = ResolveAddress(gSrcAddress, gSrcPort,
gAddressFamily, gSocketType, gProtocol);
    if (ressrc == NULL)
    {
        fprintf(stderr, "Unable to resolve address '%s' and port
'%s'\n", gSrcAddress, gSrcPort);
        return -1;
    }
    else
        printf("ResolveAddress(): Address resolved!\n");

    if (bSender)
    {
        resdest = ResolveAddress(gDestAddress, gDestPort,
ressrc->ai_family, ressrc->ai_socktype, ressrc->ai_protocol);
        if (resdest == NULL)
        {
            fprintf(stderr, "Unable to resolve address '%s' and
port '%s'\n",
                    gDestAddress, gDestPort);
            return -1;
        }
        else
            printf("ResolveAddress(): Address resolved!\n");
    }

    //  Creating a raw socket
    //  BUG - For IPv6 if we create the raw socket with
IPPROTO_UDP then the Ipv6
    //  stack will throw away our IPv6 and UDP headers and put
"valid" ones in their
```

```
    //  place. As a workaround, create the socket with a
protocol value of an
    //  unhandled protocol. Of course the IPv6 header should
still indicate that
    //  the encapsulated protocol is UDP.
    if (bSender)
        s = socket(ressrc->ai_family, SOCK_RAW, ((ressrc-
>ai_family == AF_INET6) ? 3 : ressrc->ai_protocol));
    else if (!bSender && bReadRaw)
        s = socket(ressrc->ai_family, SOCK_RAW, ressrc-
>ai_protocol);
    else
        s = socket(ressrc->ai_family, SOCK_DGRAM, ressrc-
>ai_protocol);
    if (s == INVALID_SOCKET)
    {
        fprintf(stderr, "socket() failed with error code %d\n",
WSAGetLastError());
        return -1;
    }
    else
        printf("socket() should be fine!\n");

    if (bSender)
    {
        char   *payload=NULL;
        int     optlevel, option, optval;

        payload = (char *)HeapAlloc(GetProcessHeap(),
HEAP_ZERO_MEMORY, gSendSize);
        if (payload == NULL)
        {
            fprintf(stderr, "HeapAlloc() for buffer failed with
error code %d\n", GetLastError());
            return -1;
        }
        else
            printf("HeapAlloc() for buffer is OK!\n");

        memfill(payload, gSendSize, gMessage, strlen(gMessage));
        // Enable the IP header include option
        optval = 1;
        if (ressrc->ai_family == AF_INET)
        {
            optlevel = IPPROTO_IP;
            option   = IP_HDRINCL;
```

```
        }
        else if (ressrc->ai_family == AF_INET6)
        {
            optlevel = IPPROTO_IPV6;
            option   = IPV6_HDRINCL;
        }
        rc = setsockopt(s, optlevel, option,
(char *)&optval, sizeof(optval));
        if (rc == SOCKET_ERROR)
        {
            fprintf(stderr, "setsockopt() for IP_HDRINCL failed
with error code %d\n", WSAGetLastError());
            return -1;
        }
        else
            printf("setsockopt() for IP_HDRINCL is OK!\n");

        // Packetize and/or perform necessary fragmentation on
data
        if (ressrc->ai_family == AF_INET)
        {
            wbuf = PacketizeIpv4(ressrc, resdest, payload,
gSendSize);
        }
        else if (ressrc->ai_family == AF_INET6)
        {
            wbuf = PacketizeIpv6(ressrc,resdest, payload,
gSendSize);
        }

        // Count how many packets there are
        i=0;
        packets=0;
        while (wbuf[i].buf)
        {
            printf("packet %d buf 0x%p len %d\n", i,
wbuf[i].buf, wbuf[i].len);
            packets++;
            i++;
        }




        // Apparently, this SOCKADDR_IN structure makes no
difference.
```

```
        // Whatever we put as the destination IP addr in the IP
        // header is what goes. Specifying a different dest in
remote
        // will be ignored.
        for(i=0; i < (int)gSendCount ;i++)
        {
            for(j=0; j < packets ;j++)
            {
                rc = sendto(s, wbuf[j].buf, wbuf[j].len, 0,
resdest->ai_addr, resdest->ai_addrlen);
                bytes = rc;
                if (rc == SOCKET_ERROR)
                {
                    printf("sendto() failed with error code
%d\n", WSAGetLastError());
                    break;
                }
                else
                {
                    printf("sendto() is fine!\n");
                    printf("sent %d bytes\n", bytes);
                }
            }
        }

        // Free the packet buffers
        for(i=0; i < packets ;i++)
        {
            HeapFree(GetProcessHeap(), 0, wbuf[i].buf);
        }
    }
    else
    {
        SOCKADDR_STORAGE    safrom;
        char                buf[MAX_PACKET];
        int                 fromlen;

        rc = bind(s, ressrc->ai_addr, ressrc->ai_addrlen);
        if (rc == SOCKET_ERROR)
        {
            fprintf(stderr, "bind() failed with error code
%d\n", WSAGetLastError());
            return -1;
        }
        else
            printf("bind() is OK!\n");
```

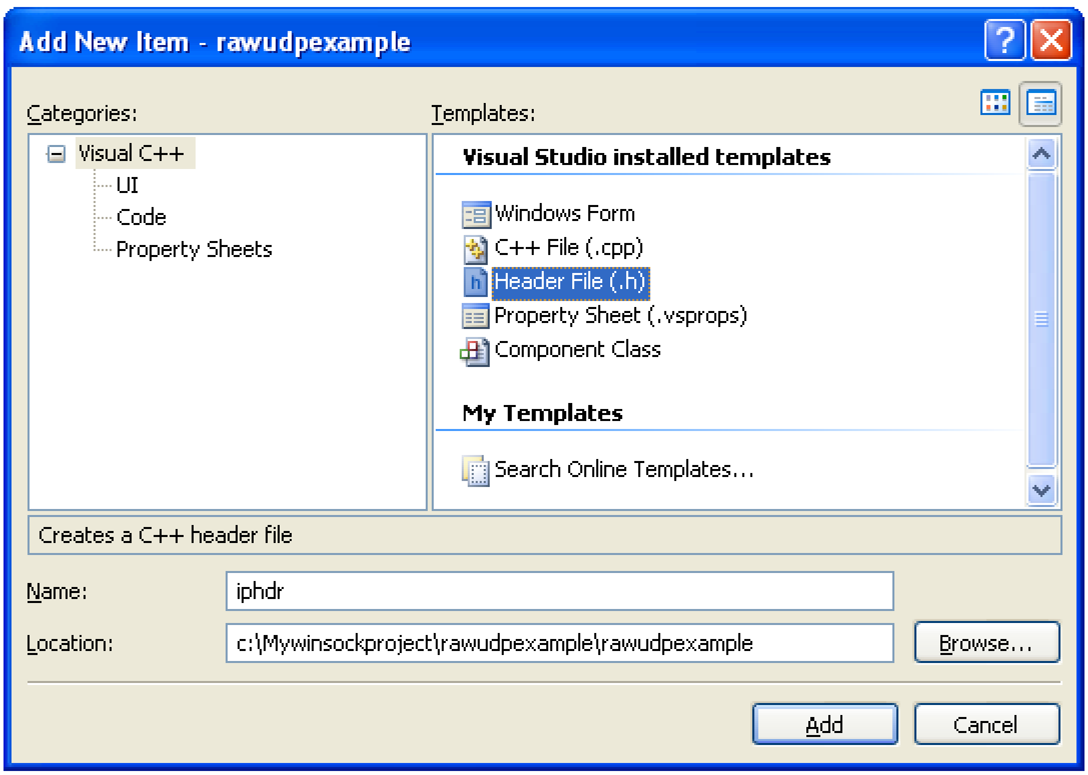
```
        printf("binding to: ");
        PrintAddress(ressrc->ai_addr, ressrc->ai_addrlen);
        printf("\n");

        while (1)
        {
            fromlen = sizeof(safrom);
            rc = recvfrom(s, buf, MAX_PACKET, 0, (SOCKADDR
*)&safrom, &fromlen);
            if (rc == SOCKET_ERROR)
            {
                fprintf(stderr, "recvfrom() failed with error
code %d\n", WSAGetLastError());
                break;
            }
            else
               printf("recvfrom() is OK!\n");

            printf("Read %d bytes from ", rc);
            PrintAddress((SOCKADDR *)&safrom, fromlen);
            printf("\n");
        }
    }
    closesocket(s) ;
    WSACleanup() ;
    return 0;
}
```

Raw Sockets 11 Part 6

What do we have in this chapter 11 part 6?

9. Program Examples: The UDP RAW Socket (continue)

Add the iphdr.h header file to the project.



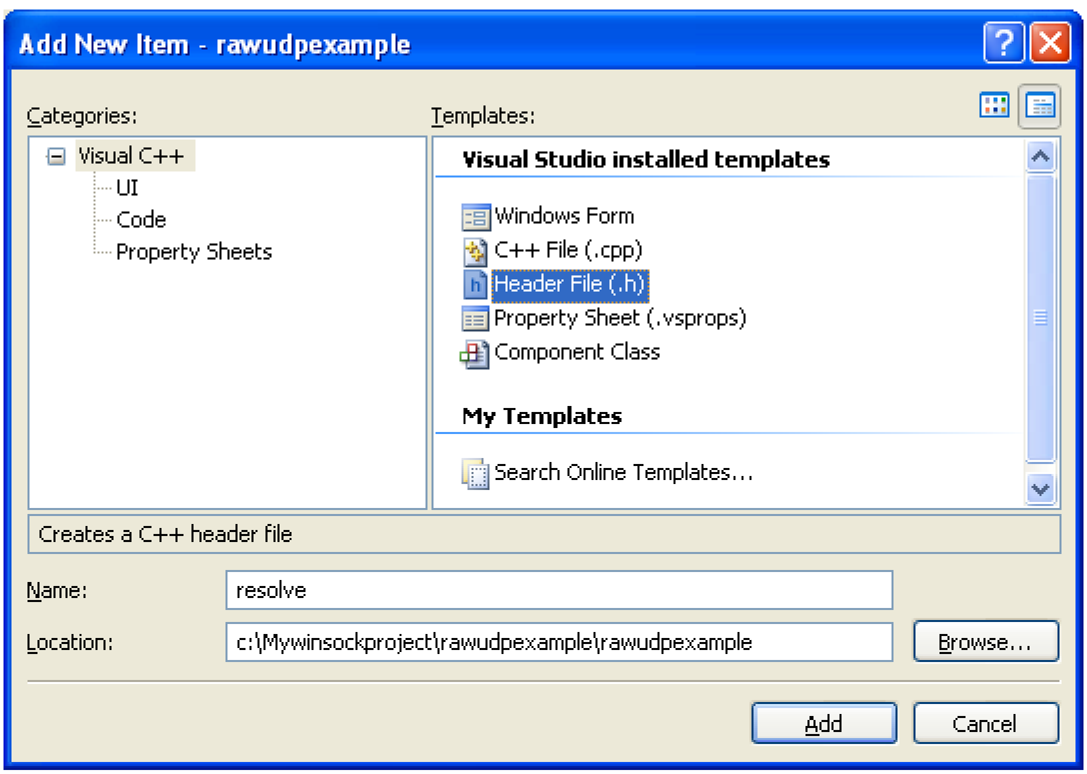
Add the following source code.

```
// Sample: Header definitions for raw UDP sample (IP_HDRINCL
option)
// Files:
//      iphdr.h     - this file
// Description:
//      This file contains the header definitions for the IPv4,
IPv6, UDP,
//      fragment headers, etc. which are used by the raw UDP
sample.
//
```

```
// Set the packing to a 1 byte boundary
#include <pshpack1.h>

// Define the IPv4 header. Make the version and length field one
// character since we can't declare two 4 bit fields without
// the compiler aligning them on at least a 1 byte boundary.
typedef struct ip_hdr
{
    unsigned char  ip_verlen;            // 4-bit IPv4 version,
4-bit header length (in 32-bit words)
    unsigned char  ip_tos;                // IP type of service
    unsigned short ip_totallength;    // Total length
    unsigned short ip_id;                 // Unique identifier
    unsigned short ip_offset;            // Fragment offset
field
    unsigned char  ip_ttl;                  // Time to live
    unsigned char  ip_protocol;       // Protocol(TCP,UDP etc)
    unsigned short ip_checksum;    // IP checksum
    unsigned int   ip_srcaddr;            // Source address
    unsigned int   ip_destaddr;          // Source address
} IPV4_HDR, *PIPV4_HDR, FAR * LPIPV4_HDR;

// IPv6 header
typedef struct ipv6_hdr
{
    unsigned long   ipv6_vertcflow;        // 4-bit IPv6
version, 8-bit traffic class, 20-bit flow label
    unsigned short  ipv6_payloadlen;    // payload length
    unsigned char   ipv6_nexthdr;          // next header
protocol value
    unsigned char   ipv6_hoplimit;         // TTL
    struct in6_addr ipv6_srcaddr;          // Source address
    struct in6_addr ipv6_destaddr;        // Destination address
} IPV6_HDR, *PIPV6_HDR, FAR * LPIPV6_HDR;

// IPv6 fragment header
typedef struct ipv6_fragment_hdr
{
    unsigned char   ipv6_frag_nexthdr;
    unsigned char   ipv6_frag_reserved;
    unsigned short  ipv6_frag_offset;
    unsigned long   ipv6_frag_id;
} IPV6_FRAGMENT_HDR, *PIPV6_FRAGMENT_HDR, FAR *
LPIPV6_FRAGMENT_HDR;
```

```
// Define the UDP header
typedef struct udp_hdr
{
    unsigned short src_portno;       // Source port no.
    unsigned short dst_portno;       // Dest. port no.
    unsigned short udp_length;       // Udp packet length
    unsigned short udp_checksum;     // Udp checksum (optional)
} UDP_HDR, *PUDP_HDR;

// Restore the byte boundary back to the previous value
#include <poppack.h>
```

Add the resolve.h header file.

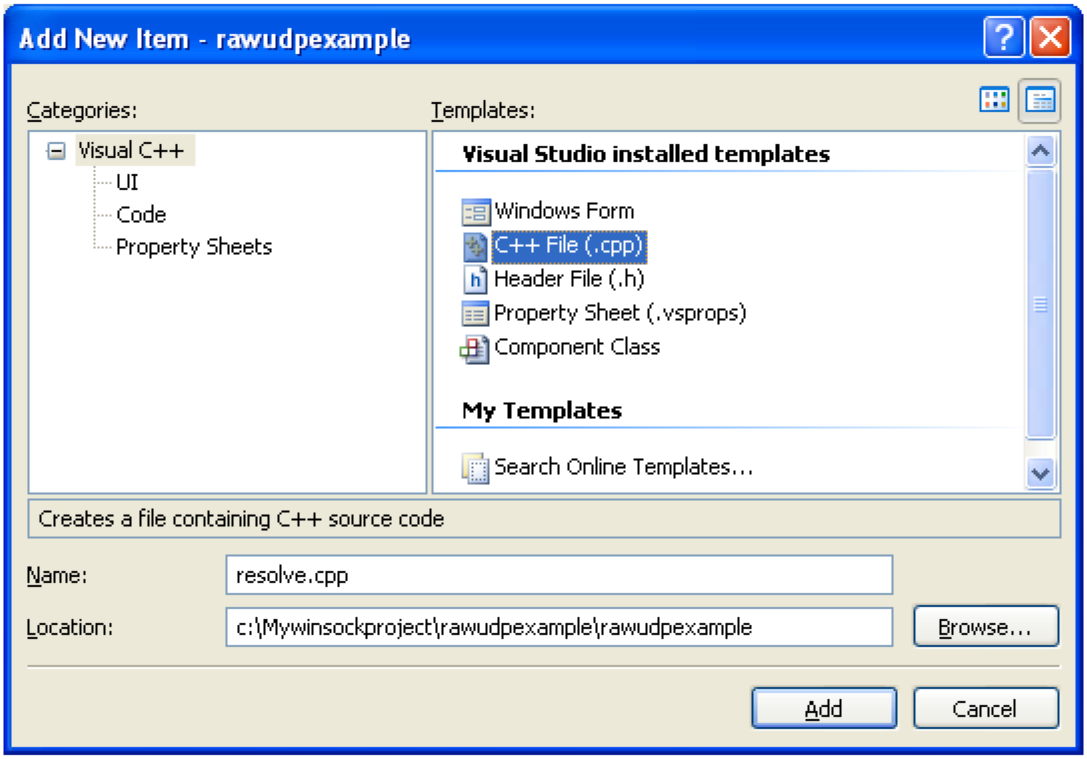


Add the following source code.

```
// Common routines for resolving addresses and hostnames
// Files:
//      resolve.h       - Header file for common routines
```

```
// Description:
//      This file contains common name resolution and name
printing
//      routines and is used by many of the samples on this CD.
//
#ifndef _RESOLVE_H_
#define _RESOLVE_H_

#ifdef _cplusplus
extern "C" {
#endif

int                 PrintAddress(SOCKADDR *sa, int salen);
int                 FormatAddress(SOCKADDR
*sa, int salen, char *addrbuf, int addrbuflen);
int                 ReverseLookup(SOCKADDR
*sa, int salen, char *namebuf, int namebuflen);
struct addrinfo
*ResolveAddress(char *addr, char *port, int af, int typ
e, int proto);

#ifdef _cplusplus
}
#endif
#endif
```

Add the resolve.h header definition file, resolve.cpp.



Add the source code.

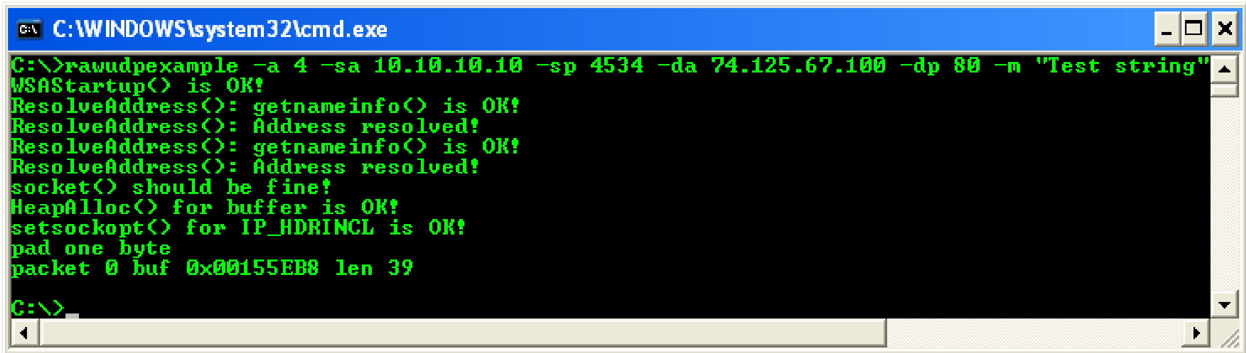
```
// Common routines for resolving addresses and hostnames
// Files:
//      resolve.cpp     - Common routines
//      resolve.h       - Header file for common routines
// Description:
//      This file contains common name resolution and name
printing
//      routines and is used by many of the samples on this CD.
//
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
#include "resolve.h"

// Function: PrintAddress
// Description:
//    This routine takes a SOCKADDR structure and its length and
prints
//    converts it to a string representation. This string is
printed to the console via stdout.
```

```c
int PrintAddress(SOCKADDR *sa, int salen)
{
    char    host[NI_MAXHOST], serv[NI_MAXSERV];
    int     hostlen = NI_MAXHOST, servlen = NI_MAXSERV, rc;

    rc = getnameinfo(sa, salen, host, hostlen, serv, servlen,
NI_NUMERICHOST | NI_NUMERICSERV);
    if (rc != 0)
    {
        fprintf(stderr, "%s: getnameinfo() failed with error
code %d\n", __FILE__, rc);
        return rc;
    }
    else
        printf("PrintAddress(): getnameinfo() is OK!\n");

    // If the port is zero then don't print it
    if (strcmp(serv, "0") != 0)
    {
        if (sa->sa_family == AF_INET)
            printf("[%s]:%s", host, serv);
        else
            printf("%s:%s", host, serv);
    }
    else
        printf("%s", host);

    return NO_ERROR;
}

// Function: FormatAddress
// Description:
//    This is similar to the PrintAddress function except that
instead of
//    printing the string address to the console, it is
formatted into the supplied string buffer.
int FormatAddress(SOCKADDR
*sa, int salen, char *addrbuf, int addrbuflen)
{
    char    host[NI_MAXHOST], serv[NI_MAXSERV];
    int     hostlen = NI_MAXHOST, servlen = NI_MAXSERV, rc;

    rc = getnameinfo(sa, salen, host, hostlen, serv, servlen,
NI_NUMERICHOST | NI_NUMERICSERV);
    if (rc != 0)
```

```
    {
        fprintf(stderr, "%s: getnameinfo() failed with error
code %d\n", __FILE__, rc);
        return rc;
    }
    else
        printf("FormatAddress(): getnameinfo() is OK!\n");

    if ( (strlen(host) + strlen(serv) + 1) >
(unsigned)addrbuflen)
        return WSAEFAULT;
    if (sa->sa_family == AF_INET)
        sprintf_s(addrbuf, sizeof(addrbuf), "%s:%s", host,
serv);
    else if (sa->sa_family == AF_INET6)
        sprintf_s(addrbuf, sizeof(addrbuf), "[%s]:%s", host,
serv);
    else
        addrbuf[0] = '\0';

    return NO_ERROR;
}

// Function: ResolveAddress
// Description:
//    This routine resolves the specified address and returns a
list of addrinfo
//    structure containing SOCKADDR structures representing the
resolved addresses.
//    Note that if 'addr' is non-NULL, then getaddrinfo will
resolve it whether
//    it is a string literal address or a hostname.
struct addrinfo
*ResolveAddress(char *addr, char *port, int af, int typ
e, int proto)
{
    struct addrinfo hints,
    *res = NULL;
    int             rc;

    memset(&hints, 0, sizeof(hints));
    hints.ai_flags  = ((addr) ? 0 : AI_PASSIVE);
    hints.ai_family = af;
    hints.ai_socktype = type;
    hints.ai_protocol = proto;
```

```
    rc = getaddrinfo(addr, port, &hints, &res);
    if (rc != 0)
    {
        printf("Invalid address %s, getaddrinfo() failed with
error code %d\n", addr, rc);
        return NULL;
    }
    else
        printf("ResolveAddress(): getnameinfo() is OK!\n");

    return res;
}

// Function: ReverseLookup
// Description:
//    This routine takes a SOCKADDR and does a reverse lookup
for the name
//    corresponding to that address.
int ReverseLookup(SOCKADDR
*sa, int salen, char *buf, int buflen)
{
    char    host[NI_MAXHOST];
    int     hostlen=NI_MAXHOST, rc;

    rc = getnameinfo(sa, salen, host, hostlen, NULL, 0, 0);
    if (rc != 0)
    {
        fprintf(stderr, "getnameinfo() failed with error code
%d\n", rc);
        return rc;
    }
    else
        printf("ReverseLookup(): getnameinfo() is OK!\n");
    strcpy_s(buf, sizeof(host), host);
    return NO_ERROR;
}
```

Build and run the project.



As we mentioned previously, using the header include option for IPv4 is easy because the stack will perform any fragmentation necessary. However, for IPv6 the stack will not, which means if your application needs to send raw data with a payload that exceeds the MTU, it will have to fragment the packets manually before sending them. This is accomplished by including the IPv6 fragmentation header after the IPv6 header but before the remaining payload. To do this, the IPv6 header's next header value will indicate the IPv6 fragmentation header (whose value is 44). The next header value of the IPv6 fragmentation header will then indicate IPPROTO_UDP. Also note that the UDP header occurs only once. The first fragment will contain the IPv6 header, IPv6 fragmentation header, UDP header, and as much of the payload that will fit into the MTU. The subsequent fragments will contain only the IPv6 header, the IPv6 fragmentation header, and the remaining payload. Figure 11-6 illustrates this example. In this case, the MTU is 1500 bytes but a 2000 byte payload is being sent.

IPv6 Header Next Header = Fragment Header	Fragment Header Next Header = UDP Offset = 0 ID = 10	UDP Header	UDP Payload (1444 bytes)

IPv6 Header Next Header = Fragment Header	Fragment Header Next Header = UDP Offset = 1444 ID = 10	UDP Payload (566 bytes)

Figure 11-6 IPv6 UDP packet with fragmentation

There are two routines of interest: PacketizeIpv4 and PacketizeIpv6. The v4 routine doesn't do anything of interest because we know the stack will fragment the data if required. However, the v6 routine will build the appropriate IPv6 header and fragmentation header for each fragment necessary.