

WINSOCK 2

WINDOWS SOCKET: PROGRAM EXAMPLES PART 12

Machine's OS is standalone Windows XP Pro with SP2 except whenever mentioned. Compiler used was Visual C++ 2003 .Net 1.1. Beware the codes that span more than one line. Program examples have been tested for Non Destructive Test. All information compiled for Windows 2000 (NT5.0) above and...

1. The story was discussed at [Winsock introduction story](#).
2. Related functions, structures and macros used in the program examples have been dumped at [Winsock structure & function 1](#) and [Winsock structure & function 2](#).
3. Other related and required information (if any) not available in no. 2 can be found at [MSDN & Visual C++ online reference](#).

Abilities

- Able to understand Winsock implementation and operations of the IPv6 through the APIs and program examples.
- Able to gather, understand and use the Winsock [functions](#), [structures](#) and [macros](#) in your programs.
- Able to build programs that use Microsoft C/Standard C programming language and Winsock APIs.

And the client program for IPv6 example.

```
// Client for IPv6 enabled program example
```

```
#define WIN32_LEAN_AND_MEAN
```

```
#include <winsock2.h>
```

```
#include <ws2tcpip.h>
```

```
#ifndef IPPROTO_IPV6
```

```
// For IPv6
```

```
#include <tcpv6.h>
```

```
#endif
```

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
#include <string.h>
```

```
// This code assumes that at the transport level, the system only supports  
// one stream protocol (TCP) and one datagram protocol (UDP). Therefore,  
// specifying a socket type of SOCK_STREAM is equivalent to specifying TCP  
// and specifying a socket type of SOCK_DGRAM is equivalent to specifying  
UDP.  
//
```

```
// Will use the loopback interface
```

```

#define DEFAULT_SERVER  NULL
// Accept either IPv4 or IPv6
#define DEFAULT_FAMILY  PF_UNSPEC
// TCP socket type
#define DEFAULT_SOCKETYPE  SOCK_STREAM
// Arbitrary, test port
#define DEFAULT_PORT  "2007"
// Number of "extra" bytes to send
#define DEFAULT_EXTRA  0
#define BUFFER_SIZE  65536

void Usage(char *ProgName)
{
    fprintf(stderr, "\nSimple socket IPv6 client program.\n");
    fprintf(stderr, "\n%s [-s server] [-f family] [-t transport] [-p port] [-b bytes] [-n
number]\n\n", ProgName);
    fprintf(stderr, "  server\tServer name or IP address. (default: %s)\n",
        (DEFAULT_SERVER == NULL) ? "loopback address" :
DEFAULT_SERVER);
    fprintf(stderr, "  family\tOne of PF_INET, PF_INET6 or PF_UNSPEC. (default:
%s)\n",
        (DEFAULT_FAMILY == PF_UNSPEC) ? "PF_UNSPEC" :
        ((DEFAULT_FAMILY == PF_INET) ? "PF_INET" : "PF_INET6"));
    fprintf(stderr, "  transport\tEither TCP or UDP. (default: %s)\n",
        (DEFAULT_SOCKETYPE == SOCK_STREAM) ? "TCP" : "UDP");
    fprintf(stderr, "  port\t\tPort on which to connect. (default: %s)\n",
        DEFAULT_PORT);
    fprintf(stderr, "  bytes\t\tBytes of extra data to send. (default: %d)\n",
        DEFAULT_EXTRA);
    fprintf(stderr, "  number\tNumber of sends to perform. (default: 1)\n");
    fprintf(stderr, "  (-n by itself makes client run in an infinite loop,");
    fprintf(stderr, " Hit Ctrl-C to terminate)\n");
    WSACleanup();
    exit(1);
}
LPSTR DecodeError(int ErrorCode)
{
    static char Message[1024];

    // If this program was multi-threaded, we'd want to use
    // FORMAT_MESSAGE_ALLOCATE_BUFFER instead of a static buffer here.
    // (And of course, free the buffer when we were done with it)

    FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM |
FORMAT_MESSAGE_IGNORE_INSERTS |
        FORMAT_MESSAGE_MAX_WIDTH_MASK,

```

```

        NULL, ErrorCode, MAKELANGID(LANG_NEUTRAL,
SUBLANG_DEFAULT),
        (LPSTR)Message, 1024, NULL);
    return Message;
}

int ReceiveAndPrint(SOCKET ConnSocket, char *Buffer, int BufLen)
{
    int AmountRead;

    AmountRead = recv(ConnSocket, Buffer, BufLen, 0);
    if (AmountRead == SOCKET_ERROR)
    {
        fprintf(stderr, "Client: recv() failed with error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
        closesocket(ConnSocket);
        WSACleanup();
        exit(1);
    }
    else
        printf("Client: recv() is OK.\n");

    // We are not likely to see this with UDP, since there is no 'connection'
established.
    if (AmountRead == 0)
    {
        printf("Client: Server closed the connection...\n");
        closesocket(ConnSocket);
        WSACleanup();
        exit(0);
    }

    printf("Client: Received %d bytes from server: \"%s\"\n", AmountRead,
AmountRead, Buffer);
    return AmountRead;
}

int main(int argc, char **argv)
{
    char Buffer[BUFFER_SIZE], AddrName[NI_MAXHOST];
    char *Server = DEFAULT_SERVER;
    int Family = DEFAULT_FAMILY;
    int SocketType = DEFAULT_SOCKETTYPE;
    char *Port = DEFAULT_PORT;
    int i, RetVal, AddrLen, AmountToSend;
    int ExtraBytes = DEFAULT_EXTRA;
    unsigned int Iteration, MaxIterations = 1;

```

```

BOOL RunForever = FALSE;
WSADATA wsaData;
ADDRINFO Hints, *AddrInfo, *AI;
SOCKET ConnSocket;
struct sockaddr_storage Addr;

printf("Usage: %s [-s server] [-f family] [-t transport] [-p port] [-b bytes] [-n
number]\n", argv[0]);
printf("Example: %s -s 127.0.0.1 -f PF_INET6 -t TCP -p 1234 -b 1024 -n
4\n", argv[0]);
printf("Else, default values used.\n\n");
if (argc > 1)
{
for (i = 1; i < argc; i++)
{
if (((argv[i][0] == '-') || (argv[i][0] == '/')) &&
(argv[i][1] != 0) && (argv[i][2] == 0))
{
switch(tolower(argv[i][1]))
{
case 'f':
if (!argv[i+1])
Usage(argv[0]);
if (!strcmp(argv[i+1], "PF_INET"))
Family = PF_INET;
else if (!strcmp(argv[i+1], "PF_INET6"))
Family = PF_INET6;
else if (!strcmp(argv[i+1], "PF_UNSPEC"))
Family = PF_UNSPEC;
else
Usage(argv[0]);
i++;
break;
case 't':
if (!argv[i+1])
Usage(argv[0]);
if (!strcmp(argv[i+1], "TCP"))
SocketType = SOCK_STREAM;
else if (!strcmp(argv[i+1], "UDP"))
SocketType = SOCK_DGRAM;
else
Usage(argv[0]);
i++;
break;
case 's':
if (argv[i+1])
{

```

```

        if (argv[i+1][0] != '-')
        {
            Server = argv[++i];
            break;
        }
    }
    Usage(argv[0]);
    break;
case 'p':
    if (argv[i+1])
    {
        if (argv[i+1][0] != '-')
        {
            Port = argv[++i];
            break;
        }
    }
    Usage(argv[0]);
    break;
case 'b':
    if (argv[i+1])
    {
        if (argv[i+1][0] != '-')
        {
            ExtraBytes = atoi(argv[++i]);
            if (ExtraBytes > sizeof(Buffer) - sizeof("Message
#4294967295"))
                Usage(argv[0]);
            break;
        }
    }
    Usage(argv[0]);
    break;
case 'n':
    if (argv[i+1])
    {
        if (argv[i+1][0] != '-')
        {
            MaxIterations = atoi(argv[++i]);
            break;
        }
    }
    RunForever = TRUE;
    break;
default:
    Usage(argv[0]);

```



```

}
else
    printf("Client: socket() is OK.\n");
    // Notice that nothing in this code is specific to whether we
    // are using UDP or TCP.
    //
    // When connect() is called on a datagram socket, it does not
    // actually establish the connection as a stream (TCP) socket
    // would. Instead, TCP/IP establishes the remote half of the
    // (LocalIPAddress, LocalPort, RemoteIP, RemotePort) mapping.
    // This enables us to use send() and recv() on datagram sockets,
    // instead of recvfrom() and sendto().
    printf("Client: Attempting to connect to: %s\n", Server ? Server : "localhost");
    if (connect(ConnSocket, AI->ai_addr, int(AI->ai_addrlen)) !=
SOCKET_ERROR)
        break;
    else
        printf("Client: connect() is OK.\n");

    i = WSAGetLastError();
    if (getnameinfo(AI->ai_addr, int(AI->ai_addrlen), AddrName,
sizeof(AddrName), NULL, 0, NI_NUMERICHOST) != 0)
        strcpy(AddrName, "<unknown>");
        fprintf(stderr, "Client: connect() to %s failed with error %d: %s\n",
AddrName, i, DecodeError(i));
        closesocket(ConnSocket);
    }

    if (AI == NULL)
    {
        fprintf(stderr, "Client: Fatal error: unable to connect to the server.\n");
        WSACleanup();
        return -1;
    }

    // This demonstrates how to determine to where a socket is connected.
    AddrLen = sizeof(Addr);
    if (getpeername(ConnSocket, (LPSOCKADDR)&Addr, &AddrLen) ==
SOCKET_ERROR)
    {
        fprintf(stderr, "Client: getpeername() failed with error %d: %s\n",
WSAGetLastError(), DecodeError(WSAGetLastError()));
    }
    else
    {
        if (getnameinfo((LPSOCKADDR)&Addr, AddrLen, AddrName,
sizeof(AddrName), NULL, 0, NI_NUMERICHOST) != 0)

```

```

    strcpy(AddrName, "<unknown>");
    printf("Client: Connected to %s, port %d, protocol %s, protocol family %s\n",
AddrName, ntohs(SS_PORT(&Addr)), (AI->ai_socktype == SOCK_STREAM) ?
"TCP" : "UDP", (AI->ai_family == PF_INET) ? "PF_INET" : "PF_INET6");
}

// We are done with the address info chain, so we can free it.
freeaddrinfo(AddrInfo);

// Find out what local address and port the system picked for us.
AddrLen = sizeof(Addr);
if (getsockname(ConnSocket, (LPSOCKADDR)&Addr, &AddrLen) ==
SOCKET_ERROR)
{
    fprintf(stderr, "Client: getsockname() failed with error %d: %s\n",
WSAGetLastError(), DecodeError(WSAGetLastError()));
}
else
{
    printf("Client: getsockname() is OK.\n");
    if (getnameinfo((LPSOCKADDR)&Addr, AddrLen, AddrName,
sizeof(AddrName), NULL, 0, NI_NUMERICHOST) != 0)
        strcpy(AddrName, "<unknown>");
    printf("Client: Using local address %s, port %d\n", AddrName,
ntohs(SS_PORT(&Addr)));
}

// Send and receive in a loop for the requested number of iterations.
for (Iteration = 0; RunForever || Iteration < MaxIterations; Iteration++)
{
    // compose a message to send.
    AmountToSend = sprintf(Buffer, "This is message #%u", Iteration + 1);
    for (i = 0; i < ExtraBytes; i++)
    {
        Buffer[AmountToSend++] = (char)((i & 0x3f) + 0x20);
    }

    // Send the message. Since we are using a blocking socket, this
    // call shouldn't return until it's able to send the entire amount.
    RetVal = send(ConnSocket, Buffer, AmountToSend, 0);
    if (RetVal == SOCKET_ERROR)
    {
        fprintf(stderr, "Client: send() failed with error %d: %s\n",
WSAGetLastError(), DecodeError(WSAGetLastError()));
        WSACleanup();
        return -1;
    }
}

```

```

else
    printf("Client: send() is OK.\n");

    printf("Client: Sent %d bytes (out of %d bytes) of data: \"%s\"\n", RetVal,
AmountToSend, AmountToSend, Buffer);

    // Clear buffer just to prove we're really receiving something.
    memset(Buffer, 0, sizeof(Buffer));

    // Receive and print server's reply.
    ReceiveAndPrint(ConnSocket, Buffer, sizeof(Buffer));
}

// Tell system we're done sending.
printf("Client: Sending done...\n");
shutdown(ConnSocket, SD_SEND);

// Since TCP does not preserve message boundaries, there may still
// be more data arriving from the server. So we continue to receive
// data until the server closes the connection.
if (SocketType == SOCK_STREAM)
    while (ReceiveAndPrint(ConnSocket, Buffer, sizeof(Buffer)) != 0)
        ;

    closesocket(ConnSocket);
    WSACleanup();
    return 0;
}

```

The client program's output.

```

C:\>myclient -s 127.0.0.1 -f PF_INET -t TCP -p 12345 -b 32 -n 4
Usage: myclient [-s server] [-f family] [-t transport] [-p port] [-b bytes] [-n number]
Example: myclient -s 127.0.0.1 -f PF_INET6 -t TCP -p 1234 -b 1024 -n 4
Else, default values used.

```

```

Client: WSStartup() is OK.
Client: getaddrinfo() is OK, name resolved.
Client: socket() is OK.
Client: Attempting to connect to: 127.0.0.1
Client: Connected to 127.0.0.1, port 12345, protocol TCP, protocol family
PF_INET
Client: getsockname() is OK.
Client: Using local address 127.0.0.1, port 1032
Client: send() is OK.
Client: Sent 50 bytes (out of 50 bytes) of data: "This is message #1 !"#$$%&'()*+,-
./0123456789;<=>?"
Client: recv() is OK.

```

```
Client: Received 50 bytes from server: "This is message #1 !"#$$%&'()*+,-
./0123456789;.<=>?"
Client: send() is OK.
Client: Sent 50 bytes (out of 50 bytes) of data: "This is message #2 !"#$$%&'()*+,-
./0123456789;.<=>?"
Client: recv() is OK.
Client: Received 50 bytes from server: "This is message #2 !"#$$%&'()*+,-
./0123456789;.<=>?"
Client: send() is OK.
Client: Sent 50 bytes (out of 50 bytes) of data: "This is message #3 !"#$$%&'()*+,-
./0123456789;.<=>?"
Client: recv() is OK.
Client: Received 50 bytes from server: "This is message #3 !"#$$%&'()*+,-
./0123456789;.<=>?"
Client: send() is OK.
Client: Sent 50 bytes (out of 50 bytes) of data: "This is message #4 !"#$$%&'()*+,-
./0123456789;.<=>?"
Client: recv() is OK.
Client: Received 50 bytes from server: "This is message #4 !"#$$%&'()*+,-
./0123456789;.<=>?"
Client: Sending done...
Client: recv() is OK.
Client: Server closed the connection...
```

```
C:\>
```

Finally the previous server output.

```
C:\>mywinsock -f
Usage: mywinsock [-f family] [-t transport] [-p port] [-a address]
Example: mywinsock -f PF_INET6 -t TCP -p 1234 -a 127.0.0.1
Else, default values used. By the way, the -a not usable for server...
Ctrl + C to terminate the server program.
```

Simple socket server program.

```
mywinsock [-f family] [-t transport] [-p port] [-a address]
```

family	One of PF_INET, PF_INET6 or PF_UNSPEC. (default PF_UNSPEC)
transport	Either TCP or UDP. (default: TCP)
port	Port on which to bind. (default 2007)
address	IP address on which to bind. (default: unspecified address)

```
C:\>mywinsock -f PF_INET -t TCP -p 12345
Usage: mywinsock [-f family] [-t transport] [-p port] [-a address]
Example: mywinsock -f PF_INET6 -t TCP -p 1234 -a 127.0.0.1
Else, default values used. By the way, the -a not usable for server...
Ctrl + C to terminate the server program.
```

Server: WSStartup() is OK.
Server: getaddrinfo() is OK.
Server: socket() is OK.
Server: bind() is OK.
Server: listen() is OK.
I'm listening and waiting on port 12345, protocol TCP, protocol family PF_INET
Server: select() is OK.
Server: accept() is OK.

Accepted connection from 127.0.0.1.

Server: recv() is OK.
Server: Received 50 bytes from client: "This is message #1 !"#\$\$%&'()*+,-
./0123456789;<=>?"

Server: Echoing the same data back to client...

Server: send() is OK.

Server: recv() is OK.

Server: Received 50 bytes from client: "This is message #2 !"#\$\$%&'()*+,-
./0123456789;<=>?"

Server: Echoing the same data back to client...

Server: send() is OK.

Server: recv() is OK.

Server: Received 50 bytes from client: "This is message #3 !"#\$\$%&'()*+,-
./0123456789;<=>?"

Server: Echoing the same data back to client...

Server: send() is OK.

Server: recv() is OK.

Server: Received 50 bytes from client: "This is message #4 !"#\$\$%&'()*+,-
./0123456789;<=>?"

Server: Echoing the same data back to client...

Server: send() is OK.

Server: recv() is OK.

Server: Client closed the connection.

^C

C:\>