BSD 4.3 Socket API Wrapper for NetX Duo

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

Part Number: 000-1050
Revision 5.10

Contents

Chapter 1

Introduction to NetX Duo BSD

The BSD Socket API Compliancy Wrapper supports some of the basic BSD Socket API calls, with some limitations and utilizes NetX Duo primitives underneath.

BSD Socket API Compliancy Wrapper Source

The Wrapper source code is designed for simplicity and is comprised of two files, namely *nxd_bsd.h* and *nxd_bsd.c*. The *nxd_bsd.h* file defines all the necessary BSD Socket API wrapper constants and subroutine prototypes, while *nxd_bsd.c* contains the actual BSD Socket API compatibility source code. These Wrapper source files are common to all NetX Duo support packages.

The package consists of:

nxd_bsd.c:	Wrapper source code
nxd_bsd.h:	Main header file

Sample demo programs:

bsd_demo_udp.c
Demo with two UDP peers (IPv4 only)
bsd_demo_tcp.c
Demo with a single TCP server and client
bsd_demo_raw.c
Demo with two RAW peers

Chapter 2

Installation and Use of NetX Duo BSD

This chapter contains a description of various issues related to installation, setup, and usage of the NetX Duo BSD component.

Product Distribution

NetX Duo BSD is shipped on a single CD-ROM compatible disk. The package includes two source files and a PDF file that contains this document, as follows:

nxd_bsd.h	Header file for NetX Duo BSD
nxd_bsd.c	C Source file for NetX Duo BSD
nxd_bsd.pdf	User Guide for NetX Duo BSD

Demo files:

bsd_demo_udp.c
bsd_demo_tcp.c
bsd_demo_raw.c

NetX Duo BSD Installation

In order to use NetX Duo BSD the entire distribution mentioned previously should be copied to the same directory where NetX Duo is installed. For example, if NetX Duo is installed in the directory "*\threadx\arm7\green*" then the *nxd_bsd.h* and *nxd_bsd.c* files should be copied into this directory.

Building the ThreadX and NetX Duo components of a BSD Application

ThreadX

The ThreadX library must define `bsd_errno` in the thread local storage. We recommend the following procedure:

1. In *tx_port.h*, set one of the TX_THREAD_EXTENSION macros as follows:

```
#define TX_THREAD_EXTENSION_3       int bsd_errno
```

2. Rebuild the ThreadX library.

Note that if TX_THREAD_EXTENSION_3 is already used, the user is free to use one of the other TX_THREAD_EXTENSION macros.

NetX Duo

Before using NetX Duo BSD Services, the NetX Duo library must be built with NX_ENABLE_EXTENDED_NOTIFY_SUPPORT defined. By default it is not defined. If the BSD raw sockets are to be used, the NetX Duo library must be built with NX_ENABLE_IP_RAW_PACKET_FILTER defined.

Using NetX Duo BSD

A NetX Duo BSD application project must include *nxd_bsd.h* after it includes *tx_api.h* and *nx_api.h* to be able to use BSD services specified later in this guide. The application must also include *nxd_bsd.c* in the build process. This file must be compiled in the same manner as other application files and its object form must be linked along with the files of the application. This is all that is required to use NetX Duo BSD.

To utilize NetX Duo BSD services, the application must create an IP instance, create a packet pool for the BSD layer to allocate packets from, allocate memory space for the internal BSD thread stack, and specify the priority of the internal BSD thread. The BSD layer is initialized by calling *bsd_initialize* and passing in the parameters. This is demonstrated in the “Small Examples” later in this document but the prototype is shown below:

```
INT   bsd_initialize(NX_IP *default_ip, NX_PACKET_POOL *default_pool,
                      CHAR *bsd_thread_stack_area,
                      ULONG bsd_thread_stack_size,
                      UINT bsd_thread_priority);
```

The default_ip is the IP instance the BSD layer operates on. The default_pool is used by the BSD services to allocate packets from. The next two parameters: bsd_thread_stack_area, bsd_thread_stack_size defines the stack area used by the internal BSD thread, and the last parameter, bsd_thread_priority, sets the priority of the thread.

NetX Duo BSD Raw Socket Support

NetX Duo BSD also supports raw sockets. To use raw sockets in NetX Duo BSD, the NetX Duo library must be compiled with NX_ENABLE_IP_RAW_PACKET_FILTER defined. By default it is not defined. The application must then enable raw socket processing for a previously created IP instance by calling *nx_ip_raw_packet_enable.*

To create a raw socket in NetX Duo BSD, the application uses the socket create service *socket* and specifies the protocol family, socket type and protocol:

```
sock_1 = socket(INT protocolFamily, INT socket_type, INT protocol)
```

`protocolFamily` is `AF_INET` for IPv4 sockets, or `AF_INET6` for IPv6 sockets, assuming IPv6 is enabled on the IP instance. The `socket_type` must be set to `SOCK_RAW`. `protocol` is application specific.

To send and receive raw packets as well as close a raw socket, the application typically uses the same BSD services as for UDP e.g. *sendto, recvfrom, select* and *soc_close*. Raw sockets do not support either *accept* or *listen* BSD services.

- By default, received IPv4 raw data includes the IPv4 header. Conversely, received IPv6 raw data does not include the IPv6 header.
- By default, when sending either raw IPv6 or IPv4 packets, the BSD wrapper layer adds the IPv6 or IPv4 header before sending the data.

NetX Duo BSD supports additional raw socket options, including `IP_RAW_RX_NO_HEADER`, `IP_HDRINCL` and `IP_RAW_IPV6_HDRINCL`.

If `IP_RAW_RX_NO_HEADER` is set, the IPv4 header is removed so that the received data does not contain the IPv4 header, and the reported message length does not include the IPv4 header. For IPv6 sockets, by default the raw socket receive does not include IPv6 header, equivalent to having the `IP_RAW_RX_NO_HEADER` option set. Application may use the *setsockopt* service to clear the `IP_RAW_RX_NO_HEADER` option, Once the `IP_RAW_RX_NO_HEADER` option is cleared, the received IPv6 raw data would include the IPv6 header, and the reported message length includes the IPv6 header.

This option has no effect on IPv4 or IPv6 transmitted data.

If `IP_HDRINCL` is set, the application includes the IPv4 header when sending data. This option has no effect on IPv6 transmission and is not defined by default. If IP_RAW_IPV6_HDRINCL is set, the application includes the IPv6 header when sending data. This option has no effect on IPv4 transmission and is not defined by default.

IP_HDRINCL and IP_RAW_IPV6_HDRINCL have no effect on IPv4 or IPv6 reception.

Note! The BSD 4.3 Socket specification specifies that the kernel must copy the raw packet to each socket receive buffer. However in NetX Duo BSD, if multiple sockets are created sharing the same protocol, the behavior is undefined.

NetX Duo BSD Raw Packet Support

To enable the raw packet support for PPPoE, NetX Duo BSD wrapper needs to be built with NX_BSD_RAW_PPPOE_SUPPORT enabled.

The following command creates a BSD socket to handle PPPoE raw packets:

```
Sockfd = socket(AF_PACKET, SOCK_RAW, protocol);
```

The current BSD implementation only supports two protocol types in AF_PACKET family

- `ETHERTYPE_PPPOE_DISC`
  PPPoE Discovery packets. In the MAC data frame, the Discovery packets have the Ethernet frame type 0x8863.
- `ETHERTYPE_PPPOE_SESS`
  PPP Session packets. In the MAC data frame, the Session packets have the Ethernet frame type 0x8864.

The structure `sockaddr_ll` is used to specify parameters when sending or receiving PPPoE frames.

`struct sockaddr_ll` is declared as:

```
struct sockaddr_ll
{
    USHORT sll_family;   /* Must be AF_PACKET */
    USHORT sll_protocol; /* LL frame type     */
    INT    sll_ifindex;  /* Interface Index.  */
    USHORT sll_hatype;   /* Header type       */
    UCHAR  sll_pkttype;  /* Packet type       */
    UCHAR  sll_halen;    /* Length of address */
    UCHAR  sll_addr[8];  /* Physical layer address */
};
```

Note that not every field in the structure is used by *`sendto()`* or *`recvfrom()`*. See the description below on how to set up the `sockaddr_ll` for sending and receiving PPPoE packets.

Socket created in the AF_PACKET family can be used to send either PPPoE Discovery packets or PPP session packets, regardless of the protocol specified. When transmitting a PPPoE packet, application must prepare the buffer that includes properly formatted PPPoE frame, including the MAC headers (Destination MAC address, source MAC address, and frame type.) The size of the buffer includes the 14-byte Ethernet header.

The `sockaddr_ll` struct, the `sll_ifindex` is used to indicate the physical interface to be used for sending this packet. The rest of the fields in the structure are not used. Values set to the unused fields are ignored by the BSD internal process.

The following code block illustrates how to transmit a PPPoE packet:

```
struct sockaddr_ll peer_addr;

/* Transmit the PPPoE frame using the primary network
   interface. */
peer_addr.sll_ifindex = 0;
n = sendto(sockfd, frame, frame_size, 0, (struct
           sockaddr*)&peer_addr, sizeof(peer_addr));
```

The return value indicates the number of bytes transmitted. Since PPPoE packets are message-based, on a successful transmission, the number of bytes sent matches the size of the packet, including the 14-byte Ethernet header.

PPPoE packets can be received using `recvfrom()`. The receive buffer must be big enough to accommodate message of Ethernet MTU size. The received PPPoE packet includes 14-byte Ethernet header. On receiving PPPoE packets, PPPoE Discovery packets can only be received by socket created with protocol `ETHERTYPE_PPPOE_DISC`. Similarly, PPP session packets can only be received by socket created with protocol `ETHERTYPE_PPPOE_SESS`. If multiple sockets are created for the same protocol type, arriving PPPoE packets are forwarded to the socket created first. If the first socket created for the protocol is closed, the next socket in the order of creation is used for receiving these packets.

After a PPPoE packet is received, the following fields in the `sockaddr_ll` struct are valid:

- sll_family: Set by the BSD internal to be AF_PACKET
- sll_ifindex: Indicates the interface from which the packet is received
- sll_protocol: Set to the type of packet received: `ETHERTYPE_PPPOE_DISC` or `ETHERTYPE_PPPOE_SESS`

Eliminating Internal BSD Thread

By default, BSD utilizes an internal thread to perform some of its processing. In systems with tight memory constraints, BSD can be built with `NX_BSD_TIMEOUT_PROCESS_IN_TIMER` defined, which eliminates the internal BSD thread and instead uses an internal timer to perform the same processing. This eliminates the memory required for the internal BSD thread control block and stack. However, overall timer processing is significantly increased and the BSD processing may also execute at a higher priority than needed.

To configure BSD sockets to run in the ThreadX timer context, define `NX_BSD_TIMEOUT_PROCESS_IN_TIMER` in *nxd_bsd.h*. If the BSD layer is configured to execute the BSD tasks in the timer context, in the call to *bsd_initialize*, the following three parameters are ignored, and should be set to NULL:

- `bsd_thread_stack_area`
- `bsd_thread_stack_size`
- `bsd_thread_priority`

NetX Duo BSD with DNS Support

If `NX_BSD_ENABLE_DNS` is defined, NetX Duo BSD can send DNS queries to obtain hostname or host IP information. This feature requires a NetX DNS Client to be previously created using the *nx_dns_create* service.

One or more known DNS server IP addresses must be registered with the DNS instance using the *nx_dns_server_add* service for adding IPv4 server addresses, or using the *nxd_dns_server_add* service for adding either IPv4 or IPv6 server addresses.

DNS services and memory allocation are used by *getaddrinfo* and *getnameinfo* services:

```
INT  getaddrinfo(const CHAR *node, const CHAR *service,
                 const struct addrinfo *hints, struct addrinfo **res)

INT  getnameinfo(const struct sockaddr *sa, socklen_t salen,
                 char *host, size_t hostlen, char *serv, size_t servlen, int flags)
```

When the BSD application calls *getaddrinfo* with a hostname, NetX BSD will call any of the below services to obtain the IP address:

- nx_dns_ipv4_address_by_name_get
- nxd_dns_ipv6_address_by_name_get
- nx_dns_cname_get

For *nx_dns_ipv4_address_by_name_get* and *nxd_dns_ipv6_address_by_name_get*, NetX BSD uses the ipv4_addr_buffer and ipv6_addr_buffer memory areas respectively. The size of these buffers are defined by (NX_BSD_IPV4_ADDR_PER_HOST * 4) and (NX_BSD_IPV6_ADDR_PER_HOST * 16) respectively.

For returning address information from *getaddrinfo*, NetX BSD uses the ThreadX block memory table nx_bsd_addrinfo_pool_memory, whose memory area is defined by another set of configurable options, NX_BSD_IPV4_ADDR_MAX_NUM and NX_BSD_IPV6_ADDR_MAX_NUM.

See **General Configuration Options** for more details on the above configuration options.

Additionally, if NX_DNS_ENABLE_EXTENDED_RR_TYPES is defined, and the host input is a canonical name, NetX Duo BSD will allocate memory dynamically from a previously created block pool _nx_bsd_cname_block_pool

Note that after calling *getaddrinfo* the BSD application is responsible for releasing the memory pointed to by the res argument back to the block table using the *freeaddrinfo* service.

NetX Duo BSD Limitations

Due to performance and architecture issues, NetX Duo BSD does not support all the BSD 4.3 socket features:

INT flags are not supported for *send, recv, sendto* and *recvfrom* calls.

General Configuration Options

User configurable options in *nxd_bsd.h* allow the application to fine tune NetX Duo BSD sockets for its particular application requirements.

The following is the list of configurable options that are set at compile time:

Define	**Meaning**
NX_BSD_TCP_WINDOW	Used in TCP socket create calls. 64k is typical window size for 100Mb Ethernet. The default value is 65535.
NX_BSD_SOCKFD_START	This is the logical index for the BSD socket file descriptor start value. By default this option is 32.
NX_BSD_MAX_SOCKETS	Specifies the maximum number of total sockets available in the BSD layer and must be a multiple of 32. The value is defaulted to 32.
NX_BSD_MAX_LISTEN_BACKLOG	This specifies the size of the listen queue (‘backlog’) for BSD TCP sockets. The default value is 5.
NX_MICROSECOND_PER_CPU_TICK	Specifies the number of microseconds per scheduler timer tick

NX_BSD_TIMEOUT Specifies the timeout in timer ticks on NetX Duo internal calls required by BSD. The default value is (20 * NX_IP_PERIODIC_RATE).

NX_BSD_TCP_SOCKET_DISCONNECT_TIMEOUT
Specifies the timeout in timer ticks on NetX Duo disconnect call. The default value is 1.

NX_BSD_PRINT_ERROR If set, the error status return of a BSD function returns a line number and type of error e.g. NX_SOC_ERROR where the error occurs. This requires the application developer to define the debug output. The default setting is disabled and no debug output is specified in *nxd_bsd.h*

NX_BSD_TIMEOUT_PROCESS_IN_TIMER If set, this option allows the BSD timeout process to execute in the system timer context. The default behavior is disabled. This feature is described in more detail in Chapter 2 "Installation and Use of NetX Duo BSD".

NX_BSD_RAW_PPPOE_SUPPORT Enable PPPoE raw packet support. By default this option is not enabled.

NX_BSD_ENABLE_DNS If enabled, NetX Duo BSD will send a DNS query for a hostname or host IP address. Requires a DNS Client instance to be previously created and started. By default it is not enabled.

NX_BSD_SOCKET_RAW_PROTOCOL_TABLE_SIZE
Defines the size of the raw socket table. The value must be a power of two. NetX BSD creates an array of sockets of type NX_BSD_SOCKETS for sending

	and receiving raw packets. NX_ENABLE_IP_RAW_PACKET_ _FILTER must be enabled. By default it is 32.
NX_BSD_IPV4_ADDR_MAX_NUM	Maximum number of IPv4 addresses returned by *getaddrinfo*. This along with NX_BSD_IPV6_ADDR_MAX_NUM defines the size of the NetX BSD block pool `nx_bsd_addrinfo_block_pool` for dynamically allocating memory to address information storage in *getaddrinfo*. The default value is 5.
NX_BSD_IPV6_ADDR_MAX_NUM	Maximum number of IPv6 addresses returned by *getaddrinfo*. This along with NX_BSD_IPV4_ADDR_MAX_NUM defines the size of the NetX BSD block pool `nx_bsd_addrinfo_block_pool` for dynamically allocating memory to address information storage in *getaddrinfo*.
NX_BSD_IPV4_ADDR_PER_HOST	Defines maximum IPv4 addresses stored per DNS query. The default value is 5.
NX_BSD_IPV6_ADDR_PER_HOST	Defines maximum IPv6 addresses stored per DNS query. The default value is 2.

BSD Socket Options

The following list of NetX Duo BSD socket options can be enabled (or disabled) at run time on a per socket basis using the *setsockopt* service:

```
INT  setsockopt(INT sockID, INT option_level, INT option_name, const
               void *option_value, INT option_length);
```

There are two different settings for `option_level`.

The first type of run time socket options is `SOL_SOCKET` for socket level options. To enable a socket level option, call *setsockopt* with `option_level` set to `SOL_SOCKET` and `option_name` set to the specific option e.g. `SO_BROADCAST`. To retrieve an option setting, call *getsockopt* for the `option_name` with `option_level` again set to `SOL_SOCKET`.

The list of run time socket level options is shown below.

SO_BROADCAST	If set, this enables sending and receiving broadcast packets from Netx sockets. This is the default behavior for NetX Duo. All sockets have this capability.
SO_ERROR	Used to obtain socket status on the previous socket operation of the specified socket, using the *getsockopt* service. All sockets have this capability.
SO_KEEPALIVE	If set, this enables the TCP Keep Alive feature. This requires the NetX Duo library to be built with `NX_TCP_ENABLE_KEEPALIVE` defined in *nx_user.h*. By default this feature is disabled.
SO_RCVTIMEO	This sets the wait option in seconds for receiving packets on NetX Duo BSD sockets. The default value is the NX_WAIT_FOREVER (0xFFFFFFFF) or, if non-blocking is enabled, NX_NO_WAIT (0x0).
SO_RCVBUF	This sets the window size of the TCP socket. The default value,

NX_BSD_TCP_WINDOW, is set to 64k for BSD TCP sockets. To set the size over 65535 requires the NetX Duo library to be built with the `NX_TCP_ENABLE_WINDOW_SCALING` be defined.

SO_REUSEADDR

If set, this enables multiple sockets to be mapped to one port. The typical usage is for the TCP Server socket. This is the default behavior of NetX Duo sockets.

The second type of run time socket options is the IP option level. To enable an IP level option, call *setsockopt* with `option_level` set to `IP_PROTO` and `option_name` set to the option e.g. `IP_MULTICAST_TTL`. To retrieve an option setting, call *getsockopt* for the `option_name` with `option_level` again set to `IP_PROTO`.

The list of run time IP level options is shown below.

IP_MULTICAST_TTL

This sets the time to live for UDP sockets. The default value is NX_IP_TIME_TO_LIVE (0x80) when the socket is created. This value can be overridden by calling *setsockopt* with this socket option.

IP_RAW_IPV6_HDRINCL

If this option is set, the calling application must append an IPv6 header and optionally application headers to data being transmitted on raw IPv6 sockets created by BSD. To use this option, raw socket processing must be enabled on the IP task.

IP_ADD_MEMBERSHIP

If set, this options enables the BSD socket (applies only to UDP sockets) to join the specified IGMP group.

IP_DROP_MEMBERSHIP	If set, this options enables the BSD socket (applies only to UDP sockets) to leave the specified IGMP group.
IP_HDRINCL	If this option is set, the calling application must append the IP header and optionally application headers to data being transmitted on raw IPv4 sockets created in BSD. To use this option, raw socket processing must be enabled on the IP task.
IP_RAW_RX_NO_HEADER	If cleared, the IPv6 header is included with the received data for raw IPv6 sockets created in BSD. IPv6 headers are removed by default in BSD raw IPv6 sockets, and the packet length does not include the IPv6 header. If set, the IPv4 header is removed from received data on BSD raw sockets of type IPv4. IPv4 headers are included by default in BSD raw IPv4 sockets and packet length includes the IPv4 header. This option has no effect on either IPv4 or IPv6 transmission data.

Small IPv4 Example

An example of how to use NetX Duo BSD services for IPv4 networks is described below. In this example, the include file *nxd_bsd.h* is brought in at line 8. Next, the IP instance *bsd_ip* and packet pool *bsd_pool* are created as global variables at line 20 and 21. Note that this demo uses a ram (virtual) network driver, *_nx_ram_network_driver*. The client and server will share the same IP address on single IP instance in this example.

The client and server threads are created on lines 62 and 68. The BSD packet pool for transmitting packets is created on line 78 and used in the IP instance creation on line 87. Note that the IP thread task is given priority 1 in the *nx_ip_create* call. This thread should be the highest priority task defined in the program for optimal NetX performance.

The IP instance is enabled for ARP and TCP services on lines 88 and 110 respectively. The last requirement before BSD services can be used is to call *bsd_initialize* on line 120 to set up all data structures and NetX and ThreadX resources needed by BSD.

The server thread entry function is defined next. The BSD TCP socket is created on line 149. The server IP address and port are set on lines 160-163. Note the use of host to network byte order macros *htonl* and *htons* applied to the IP address and port. This is in compliance with BSD socket specification that multi byte data is submitted to the BSD services in network byte order.

Next, the master server socket is bound to the port using the *bind* service on line 166. This is the listening socket for TCP connection requests using the *listen* service on line 180. From here the server thread function, *thread_server_entry*, loops to check for receive events using the *select* call on line 202. If a receive event is a connection request, which is determined by comparing the read ready list, it calls *accept* on line 213. A child server socket is assigned to handle the connection request and added to the master list of TCP server sockets connected to a Client on line 223. If there are no new connection requests, the server thread then checks all the currently connected sockets for receive events in the `for` loop starting on line 236. When a receive event waiting is detected, it calls *send* and *recv* on that socket until no data is received (connection closed on the other side) and the socket is closed using the *soc_close* service on line 277.

After the server thread sets up, the Client thread entry function, *thread_client_entry,* creates a socket on line 326 and connects with the TCP server socket using the *connect* call on line 337. It then loops to send and receive packets using the *send* and *recv* services respectively. When no more data is received, it closes the socket on line 398 using the *soc_close* service. After disconnection, the client thread entry function creates a new TCP socket and makes another connection request in the while loop started on line 321.

```
1    /* This is a small demo of BSD Wrapper for the high-performance NetX Duo
2       TCP/IP stack which uses standard BSD services for TCP connection, sending,
3       and  receiving using a simulated Ethernet driver.  */
4
5
6    #include         "tx_api.h"
```

```
#include        "nx_api.h"
#include        "nxd_bsd.h"
#include        <string.h>
#include        <stdlib.h>

#define         DEMO_STACK_SIZE     (16*1024)
#define         SERVER_PORT          87
#define         CLIENT_PORT          77

/* Define the ThreadX and NetX object control blocks... */

TX_THREAD        thread_server;
TX_THREAD        thread_client;
NX_PACKET_POOL   bsd_pool;
NX_IP            bsd_ip;

/* Define some global data. */
CHAR    *msg0 = "Client 1:
   ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABCDEFGHIJKLMNOPQR
   STUVWXYZ<>END";

INT      maxfd;

/* Define the counters used in the demo application...  */

ULONG            error_counter;

/* Define fd_sets for the BSD server socket.  */
fd_set      master_list, read_ready;

/* Define thread prototypes.  */

VOID        thread_server_entry(ULONG thread_input);
VOID        thread_client_entry(ULONG thread_input);
void        _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);


/* Define main entry point.  */

int main()
{

    /* Enter the ThreadX kernel.  */
    tx_kernel_enter();
}

/* Define what the initial system looks like.  */

void    tx_application_define(void *first_unused_memory)
{
CHAR    *pointer;
UINT    status;


    /* Setup the working pointer.  */
    pointer =  (CHAR *) first_unused_memory;

    /* Create a server thread.  */
    tx_thread_create(&thread_server, "Server", thread_server_entry, 0,
                     pointer, DEMO_STACK_SIZE, 8, 8, TX_NO_TIME_SLICE,
                     TX_AUTO_START);

    pointer =  pointer + DEMO_STACK_SIZE;

    /* Create a Client thread.  */
    tx_thread_create(&thread_client, "Client", thread_client_entry, 0,
                     pointer, DEMO_STACK_SIZE, 16, 16, TX_NO_TIME_SLICE,
                     TX_AUTO_START);

    pointer =  pointer + DEMO_STACK_SIZE;


    /* Initialize the NetX system.  */
    nx_system_initialize();

    /* Create a BSD packet pool.  */
    status =  nx_packet_pool_create(&bsd_pool, "NetX BSD Packet Pool", 128,
                                    pointer, 16384);
    pointer = pointer + 16384;
    if (status)
    {
```

```
            error_counter++;
            printf("Error in creating BSD packet pool\n!");
        }

        /* Create an IP instance for BSD.  */
        status = nx_ip_create(&bsd_ip, "BSD IP Instance", IP_ADDRESS(1,2,3,4),
                              0xFFFFFF00UL,  &bsd_pool, _nx_ram_network_driver,
                              pointer, 2048, 1);
        pointer =  pointer + 2048;

        if (status)
        {
        error_counter++;
            printf("Error creating BSD IP instance\n!");
        }

        /* Enable ARP and supply ARP cache memory for BSD IP Instance */
        status =  nx_arp_enable(&bsd_ip, (void *) pointer, 1024);
        pointer = pointer + 1024;

        /* Check ARP enable status.  */
        if (status)
        {
            error_counter++;
            printf("Error in Enable ARP \n");
        }

        /* Enable TCP processing for BSD IP instances.  */

        status = nx_tcp_enable(&bsd_ip);

        /* Check TCP enable status.  */
        if (status)
        {
            error_counter++;
            printf("Error in Enable TCP \n");
        }

        /* Now initialize BSD Scoket Wrapper */
        status = bsd_initialize (&bsd_ip, &bsd_pool,pointer, 2048, 2);
    }


    /* Define the Server thread.  */
    CHAR        Server_Rcv_Buffer[100];

    VOID  thread_server_entry(ULONG thread_input)
    {

    INT         status,  sock, sock_tcp_server;
    ULONG       actual_status;
    INT         Clientlen;
    INT         i;
    UINT        is_set = NX_FALSE;
    struct      sockaddr_in serverAddr;
    struct      sockaddr_in ClientAddr;


        tx_thread_sleep(100);
        status =  nx_ip_status_check(&bsd_ip, NX_IP_INITIALIZE_DONE,
                                     &actual_status, 100);

        /* Check status...  */
        if (status != NX_SUCCESS)
        {
            return;
        }

        /* Create BSD TCP Socket */
        sock_tcp_server = socket(AF_INET, SOCK_STREAM, 0);

        if (sock_tcp_server == -1)
        {
            printf("Error on Server socket %d create \n", sock_tcp_server);
            return;
        }

        printf("Server socket %d created\n", sock_tcp_server);

        /* Set the server port and IP address */
        memset(&serverAddr, 0, sizeof(serverAddr));
```

```
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = htonl(IP_ADDRESS(1,2,3,4));
    serverAddr.sin_port = htons(SERVER_PORT);

    /* Bind this server socket */
    status = bind (sock_tcp_server, (struct sockaddr *) &serverAddr,
                   sizeof(serverAddr));

    if (status < 0)
    {
        printf("Error on Server Socket %d Bind \n", sock_tcp_server);
        return;
    }

    FD_ZERO(&master_list);
    FD_ZERO(&read_ready);
    FD_SET(sock_tcp_server,&master_list);
    maxfd = sock_tcp_server;

    /* Now listen for any client connections for this server socket */
    status = listen (sock_tcp_server, 5);
    if (status < 0)
    {
        printf("Error on Server Socket %d Listen\n", sock_tcp_server);
        return;
    }
    else
        printf("Server socket %d listen complete\n", sock_tcp_server);

    /* All set to accept client connections */

    /* Loop to create and establish server connections.  */
    while(1)
    {

        printf("\n");

        read_ready = master_list;

        tx_thread_sleep(20);   /* Allow some time to other threads too */

        /* Let the underlying TCP stack determine the timeout. */
        status = select(maxfd + 1, &read_ready, 0, 0, 0);

        if ((status == 0xFFFFFFFF) || (status == 0))
        {

            printf("Error with select. Status 0x%x\n", status);

            continue;
        }

        /* Check for a connection request. */
        is_set = FD_ISSET(sock_tcp_server, &read_ready);

        if(is_set)
        {

            Clientlen = sizeof(ClientAddr);

            sock = accept(sock_tcp_server,(struct sockaddr*)&ClientAddr,
                          &Clientlen);

            /* Add this new connection to our master list */
            FD_SET(sock, &master_list);

            if ( sock > maxfd)
            {

                maxfd = sock;
            }

            continue;
        }

        /* Check the set of 'ready' sockets, e.g connected to remote host and
           waiting for notice of packets received. */
        for (i = NX_BSD_SOCKFD_START; i < (maxfd+1+NX_BSD_SOCKFD_START); i++)
        {

            if ((i != sock_tcp_server) &&
```

```
                 (FD_ISSET(i , &master_list)) &&
                 (FD_ISSET(i , &read_ready)))
            {

                while(1)
                {

                    status = recv(i, (VOID *)Server_Rcv_Buffer, 100, 0);

                    if (status == 0)
                    {
                        printf("(Server received no data from Client on
                                socket %d)\n", i);
                        break;
                    }
                    else if (status == NX_SOC_ERROR)
                    {
                        printf("Error on Server receiving data from Client on
                                socket %d\n", i);
                        break;
                    }

                    printf("Server socket %d received %d bytes: %s\n ",
                           i, strlen(Server_Rcv_Buffer), Server_Rcv_Buffer);


                    status = send(i, "Hello\n", strlen("Hello\n")+1, 0);

                    if (status == NX_SOC_ERROR)
                    {
                        printf("Error on Server socket %d sending data to
                                Client\n", i);
                    }
                    else
                    {
                        printf("Server socket %d message sent to Client:
                         Hello\n", i);
                    }
                }

                /* Close this socket */
                status = soc_close(i);

                if (status != NX_SOC_ERROR)
                {
                    printf("Server socket %d closed \n", i);
                }
                else
                {
                    printf("Error on closing Server socket %d \n", i);
                }
            }
        }

        /* Loop back to check any next client connection */
    }
}

CHAR        Client_Rcv_Buffer[100];

VOID  thread_client_entry(ULONG thread_input)
{

INT         status;
INT         sock_tcp_client, length;
struct      sockaddr_in echoServAddr;
struct      sockaddr_in localAddr;                  /


    /* Let the server side get set up. */
    tx_thread_sleep(200);

    /* Set local port for displaying IP address and port. */
    memset(&localAddr, 0, sizeof(localAddr));
    localAddr.sin_family = AF_INET;
    localAddr.sin_addr.s_addr = htonl(IP_ADDRESS(1,2,3,4));
    localAddr.sin_port = htons(CLIENT_PORT);

    /* Set server port and IP address which we need to connect. */
    memset(&echoServAddr, 0, sizeof(echoServAddr));
    echoServAddr.sin_family = AF_INET;
```

```
    echoServAddr.sin_addr.s_addr = htonl(IP_ADDRESS(1,2,3,4));
    echoServAddr.sin_port = htons(SERVER_PORT);

    /* Now make client connections with the server. */
    while (1)
    {

         printf("\n");
        /* Create BSD TCP Socket */
        sock_tcp_client = socket( AF_INET, SOCK_STREAM, 0);

        if (sock_tcp_client == -1)
        {
            printf("Error on Client socket %d create \n", sock_tcp_client);
            return;
        }

        printf("Client socket %d created\n", sock_tcp_client);

        /* Now connect this client to the server */
        status = connect(sock_tcp_client, (struct sockaddr *)&echoServAddr,
                         sizeof(echoServAddr));

        /* Check for error.  */
        if (status != OK)
        {
            printf("Error on Client socket %d connect\n", sock_tcp_client);
            soc_close(sock_tcp_client);
            return;

        }
        /* Get and print source and destination information */
        printf("Client socket %d connected \n", sock_tcp_client);

        status = getsockname(sock_tcp_client, (struct sockaddr *)&localAddr,
                             &length);
        printf("Client port = %lu , Client = 0x%x,",
               htonl(localAddr.sin_port), htonl(localAddr.sin_addr.s_addr));
        length = sizeof(struct sockaddr_in);
        status = getpeername( sock_tcp_client, (struct sockaddr *)
                              &echoServAddr, &length);
        printf("Remote port = %lu, Remote IP = 0x%x \n",
                htonl(echoServAddr.sin_port),
                htonl(echoServAddr.sin_addr.s_addr));

        /* Now receive the echoed packet from the server */
        while(1)
        {

            printf("Client sock %d sending packet to server\n",
                    sock_tcp_client);

            status = send(sock_tcp_client, "Hello", (strlen("Hello")+1), 0);

            if (status == ERROR)
            {
                printf("Error: Client Socket (%d) send \n", sock_tcp_client);
            }
            else
            {
                printf("Client socket %d sent message Hello\n",
                        sock_tcp_client);
            }

            status = recv(sock_tcp_client, (VOID *)Client_Rcv_Buffer,100,0);

            if (status <= 0)
            {

                if (status < 0)
                {
                    printf("Error on Client receiving on socket %d \n",
                            sock_tcp_client);
                }
                else
                {
                    printf("Nothing received by Client on socket %d\n",
                            sock_tcp_client);
                }

                break;
```

```
            }
            else
            {
                printf("Client socket %d received %d bytes: %s\n",
                       sock_tcp_client,
                       strlen(Client_Rcv_Buffer), Client_Rcv_Buffer);
            }

        }

        /* close this client socket */
        status = soc_close(sock_tcp_client);

        if (status != ERROR)
        {
            printf("Client Socket %d closed\n", sock_tcp_client);
        }
        else
        {
            printf("Error on Client Socket %d on close \n", sock_tcp_client);
        }

        /* Make another Client connection...*/

    }
}
```

Small IPv6 Example System

An example of how to use NetX Duo BSD services for IPv6 networks is described in the program below. This example is very similar to the IPv4 demo program previously described with a few important differences. The client and server threads, BSD packet pool, IP instance and BSD initialization happens as it does for IPv4 BSD sockets.

In the server thread entry function, *thread_server_entry*, defines a couple IPv6 variables using *sockaddr_in6* and *NXD_ADDRESS* data types on lines 145-148. The NXD_ADDRESS data type can actually store both IPv4 and IPv6 address types.

Next, the server thread enables IPv6 and ICMPv6 on the IP instance using the *nxd_ipv6_enable* and *nxd_icmpv6_enable* service respectively on line 161 and 169. Next, the link local and global IP addresses are registered with the IP instance. This is done using the *nxd_ipv6_address_set* service on lines 180 and 195. It then sleeps long enough for the IP thread task to complete the Duplicate Address Detection protocol and register these addresses as valid addresses on the *tx_thread_sleep* call on line 201.

Next, the TCP server socket is created with the AF_INET6 socket type input argument on line 204. The socket IPv6 address and port are set on lines 216-221, again noting the use of *htonl* and *htons* macros to put data in network byte order for BSD socket services. From here on, the server thread entry function is virtually identical to the IPv4 example.

The client thread entry function, *thread_client_entry*, is defined next. Note that because the TCP client in this example shares the same IP instance and IPv6 address as the TCP server, we do not need to enable IPv6 or ICMPv6 services on the IP instance again. Further, the IPv6 address is also already registered with the IP instance. Instead, the client thread entry function simply waits on line 368 for the server to set up. The server address and port are set, using the host to network byte order macros on lines 387-392, and then the Client can connect with the TCP server on line 412. Note that the local IP address data types in lines 378-383 are used only to demonstrate the *getsockname* and *getpeername* services on lines 425 and 434 respectively. Because the data is coming from the network, the network to host byte order macros as used in lines 378-383.

Next the client thread entry function enters a loop in which it creates a TCP socket, makes a TCP connection and sends and receives data with the TCP server until no more data is received virtually the same as the IPv4 example. It then closes the socket on line 483, pauses briefly and creates another TCP socket and requests a TCP server connection.

One important difference with the IPv4 example is the *socket* calls specify an IPv6 socket using the AF_INET6 input argument. Another important difference is that the TCP Client *connect* call takes an *sockaddr_in6* data type and a length argument set to the size of the *sockaddr_in6* data type.

```
/* This is a small demo of BSD Wrapper for the high-performance NetX Duo
TCP/IP stack which uses standard BSD services for TCP connection,
disconnection, sending, and receiving using a simulated Ethernet driver.  */


#include         "tx_api.h"
#include         "nx_api.h"
#include         "nxd_bsd.h"
#include         <string.h>
#include         <stdlib.h>

#define          DEMO_STACK_SIZE     (16*1024)
#define          SERVER_PORT          87
#define          CLIENT_PORT          77

/* Define the ThreadX and NetX object control blocks... */

TX_THREAD        thread_server;
TX_THREAD        thread_client;
NX_PACKET_POOL   bsd_pool;
NX_IP            bsd_ip;

/* Define some global data. */
CHAR    *msg0 = "Client 1:
                 ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABC
                 DEFGHIJKLMNOPQRSTUVWXYZ<>END";
INT      maxfd;

/* Define the counters used in the demo application...  */

ULONG            error_counter;

/* Define fd_sets for the BSD server socket.  */
fd_set           master_list, read_ready;


/* Define thread prototypes.  */

VOID        thread_server_entry(ULONG thread_input);
VOID        thread_client_entry(ULONG thread_input);
void        _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);


/* Define main entry point.  */

int main()
{

    /* Enter the ThreadX kernel.  */
    tx_kernel_enter();
}

/* Define what the initial system looks like.  */

void    tx_application_define(void *first_unused_memory)
{
CHAR    *pointer;
UINT    status;


    /* Setup the working pointer.  */
    pointer =  (CHAR *) first_unused_memory;

    /* Create a server thread.  */
    tx_thread_create(&thread_server, "Server", thread_server_entry, 0,
                     pointer, DEMO_STACK_SIZE, 8, 8,
                     TX_NO_TIME_SLICE, TX_AUTO_START);

    pointer =  pointer + DEMO_STACK_SIZE;

    /* Create a Client thread.  */
```

```
    tx_thread_create(&thread_client, "Client", thread_client_entry, 0,
                     pointer, DEMO_STACK_SIZE, 16, 16,
                     TX_NO_TIME_SLICE, TX_AUTO_START);

    pointer =  pointer + DEMO_STACK_SIZE;


    /* Initialize the NetX system.  */
    nx_system_initialize();

    /* Create a BSD packet pool.  */
    status =  nx_packet_pool_create(&bsd_pool, "NetX BSD Packet Pool",
                                    128, pointer, 16384);
    pointer = pointer + 16384;
    if (status)
    {
    error_counter++;
        printf("Error in creating BSD packet pool\n!");
    }

    /* Create an IP instance for BSD.  */
    status = nx_ip_create(&bsd_ip, "BSD IP Instance", IP_ADDRESS(1,2,3,4),
                          0xFFFFFF00UL,  &bsd_pool, _nx_ram_network_driver,
                          pointer, 2048, 1);
    pointer =  pointer + 2048;

    if (status)
    {
        error_counter++;
        printf("Error creating BSD IP instance\n!");
    }

    /* Enable ARP and supply ARP cache memory for BSD IP Instance */
    status =  nx_arp_enable(&bsd_ip, (void *) pointer, 1024);
    pointer = pointer + 1024;

    /* Check ARP enable status.  */
    if (status)
    {
        error_counter++;
        printf("Error in enable ARP on BSD IP instance\n");
    }

    /* Enable TCP processing for BSD IP instances.  */

    status = nx_tcp_enable(&bsd_ip);

    /* Check TCP enable status.  */
    if (status)
    {
        error_counter++;
        printf("Error in Enable TCP \n");
    }

    /* Now initialize BSD Scoket Wrapper */
    status = bsd_initialize(&bsd_ip, &bsd_pool,pointer, 2048, 2);

    /* Check BSD initialize status.  */
    if (status)
    {
        error_counter++;
        printf("Error in BSD initialize \n");
    }

    pointer = pointer + 2048;
}


/* Define the Server thread.  */
CHAR        Server_Rcv_Buffer[100];

VOID  thread_server_entry(ULONG thread_input)
{


INT         status,  sock, sock_tcp_server;
ULONG       actual_status;
INT         Clientlen;
INT         i;
UINT        is_set = NX_FALSE;
NXD_ADDRESS ip_address;
```

```
struct      sockaddr_in6 serverAddr;
struct      sockaddr_in6 ClientAddr;
UINT        iface_index, address_index;


    status =  nx_ip_status_check(&bsd_ip, NX_IP_INITIALIZE_DONE,
                                  &actual_status, 100);

    /* Check status...  */
    if (status != NX_SUCCESS)
    {
        return;
    }

    /* Enable IPv6 */
    status = nxd_ipv6_enable(&bsd_ip);
    if((status != NX_SUCCESS) && (status != NX_ALREADY_ENABLED))
    {
        printf("Error with IPv6 enable 0x%x\n", status);
        return;
    }

    /* Enable ICMPv6 */
    status = nxd_icmp_enable(&bsd_ip);
    if(status)
    {
        printf("Error with ECMPv6 enable 0x%x\n", status);
        return;
    }

    /* Set the primary interface for our DNS IPv6 addresses. */
    iface_index = 0;

    /* This assumes we are using the primary network interface (index 0). */
    status = nxd_ipv6_address_set(&bsd_ip, iface_index, NX_NULL, 10,
                                   &address_index);

    if (status)
        return;


    /* Set ip_0 interface address. */
    ip_address.nxd_ip_version = NX_IP_VERSION_V6;
    ip_address.nxd_ip_address.v6[0] = htonl(0x20010db8);
    ip_address.nxd_ip_address.v6[1] = htonl(0x0000f101);
    ip_address.nxd_ip_address.v6[2] = 0;
    ip_address.nxd_ip_address.v6[3] = htonl(0x101);

    /* Set the host global IP address. We are assuming a 64
       bit prefix here but this can be any value (< 128). */
    status = nxd_ipv6_address_set(&bsd_ip, iface_index, &ip_address, 64,
                                   &address_index);

    if (status)
        return;

    /* Wait for IPv6 stack to finish DAD process. */
    tx_thread_sleep(400);

    /* Create BSD TCP Socket */
    sock_tcp_server = socket(AF_INET6, SOCK_STREAM, 0);

    if (sock_tcp_server == -1)
    {
        printf("\nError: BSD TCP Server socket create \n");
        return;
    }

    printf("\nBSD TCP Server socket created %lu \n", sock_tcp_server);

    /* Set the server port and IP address */
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin6_addr._S6_un._S6_u32[0] = htonl(0x20010db8);
    serverAddr.sin6_addr._S6_un._S6_u32[1] = htonl(0xf101);
    serverAddr.sin6_addr._S6_un._S6_u32[2] = 0x0;
    serverAddr.sin6_addr._S6_un._S6_u32[3] = htonl(0x0101);
    serverAddr.sin6_port = htons(SERVER_PORT);
    serverAddr.sin6_family = AF_INET6;

    /* Bind this server socket */
    status = bind(sock_tcp_server, (struct sockaddr *) &serverAddr,
```

```
                         sizeof(serverAddr));

    if (status < 0)
    {
        printf("Error: Server Socket Bind \n");
        return;
    }

    FD_ZERO(&master_list);
    FD_ZERO(&read_ready);
    FD_SET(sock_tcp_server,&master_list);
    maxfd = sock_tcp_server;

    /* Now listen for any client connections for this server socket */
    status = listen(sock_tcp_server, 5);
    if (status < 0)
    {
        printf("Error: Server Socket Listen\n");
        return;
    }
    else
        printf("Server Listen complete\n");

    /* All set to accept client connections */
    printf("Now accepting client connections\n");

    /* Loop to create and establish server connections.  */
    while(1)
    {

        printf("\n");

        read_ready = master_list;

        tx_thread_sleep(20);   /* Allow some time to other threads too */

        /* Let the underlying TCP stack determine the timeout. */
        status = select(maxfd + 1, &read_ready, 0, 0, 0);

        if ( (status == 0xFFFFFFFF) || (status == 0) )
        {

            printf("Error with select? Status 0x%x. Try again\n", status);

            continue;
        }

        /* Detected a connection request. */
        is_set = FD_ISSET(sock_tcp_server,&read_ready);

        if(is_set)
        {

            Clientlen = sizeof(ClientAddr);

            sock = accept(sock_tcp_server,
                          (struct sockaddr*)&ClientAddr,
                          &Clientlen);

            /* Add this new connection to our master list */
            FD_SET(sock, &master_list);

            if ( sock > maxfd)
            {
                printf("New connection %d\n", sock);

                maxfd = sock;
            }

            continue;
        }

        /* Check the set of 'ready' sockets, e.g connected to remote host and
           waiting for notice of packets received. */
        for (i = NX_BSD_SOCKFD_START; i < (maxfd+1+NX_BSD_SOCKFD_START); i++)
        {

            if ((i != sock_tcp_server) &&
                 (FD_ISSET(i, &master_list)) &&
                 (FD_ISSET(i, &read_ready)))
            {
```

```

                    while(1)
                    {

                        status = recv(i, (VOID *)Server_Rcv_Buffer, 100, 0);

                        if (status == 0)
                        {
                            printf("(Server socket %d received no data from
                                    Client)\n", i);
                            break;
                        }
                        else if (status == 0xFFFFFFFF)
                        {
                            printf("Error on Server socket %d receiving data from
                                    Client\n", i);
                            break;
                        }

                        printf("Server socket %d received from Client %lu bytes:
                               %s\n ", i, strlen(Server_Rcv_Buffer),
                               Server_Rcv_Buffer);



                        status = send(i, "Hello\n", strlen("Hello\n")+1, 0);

                        if (status == ERROR)
                        {
                            printf("Error on Server socket %d sending data to
                                    Client \n", i);
                        }
                        else
                        {
                            printf("Server socket %d message sent to Client:
                                    Hello\n", i);
                        }
                    }

                    /* Close this socket */
                    status = soc_close(i);

                    if (status != ERROR)
                    {
                        printf("Server socket %d closing\n", i);
                    }
                    else
                    {

                        printf("Error on Server socket %d closing\n", i);
                    }
                }
            }

            /* Loop back to check any next client connection */
        }
}

#define     CLIENT_BUFFER_SIZE       100
CHAR        Client_Rcv_Buffer[CLIENT_BUFFER_SIZE];

VOID  thread_client_entry(ULONG thread_input)
{

INT         status;
INT         sock_tcp_client, length;
struct      sockaddr_in6 echoServAddr6;
struct      sockaddr_in6 localAddr6; address */


    /* Wait for the server side to get set up, including the DAD process. */
    tx_thread_sleep(500);

    /* ICMPv6 and IPv6 should already be enabled on the IP instance
       by the server thread entry function. */

   /* Further the IPv6 address is already established with the IP instance.
      so no need to wait for DAD completion. */

    /* Set local port and IP address (used only for getsockname call). */
    memset(&localAddr6, 0, sizeof(localAddr6));
```

```
    localAddr6.sin6_addr._S6_un._S6_u32[0] = htonl(0x20010db8);
    localAddr6.sin6_addr._S6_un._S6_u32[1] = htonl(0xf101);
    localAddr6.sin6_addr._S6_un._S6_u32[2] = 0x0;
    localAddr6.sin6_addr._S6_un._S6_u32[3] = htonl(0x0101);
    localAddr6.sin6_port = htons(CLIENT_PORT);
    localAddr6.sin6_family = AF_INET6;

    /* Set Server port and IP address to connect to the TCP server.  */
    memset(&echoServAddr6, 0, sizeof(echoServAddr6));
    echoServAddr6.sin6_addr._S6_un._S6_u32[0] = htonl(0x20010db8);
    echoServAddr6.sin6_addr._S6_un._S6_u32[1] = htonl(0xf101);
    echoServAddr6.sin6_addr._S6_un._S6_u32[2] = 0x0;
    echoServAddr6.sin6_addr._S6_un._S6_u32[3] = htonl(0x0101);
    echoServAddr6.sin6_port = htons(SERVER_PORT);
    echoServAddr6.sin6_family = AF_INET6;

    /* Now make client connections with the server. */
    while (1)
    {

         printf("\n");
        /* Create BSD TCP Socket */

        sock_tcp_client = socket(AF_INET6, SOCK_STREAM, 0);

        if (sock_tcp_client == -1)
        {
            printf("Error on Client socket %d create \n");
            return;
        }

        printf("Client socket %d created \n", sock_tcp_client);

        /* Now connect this client to the server */
        status = connect(sock_tcp_client, (struct sockaddr *)&echoServAddr6,
                         sizeof(echoServAddr6));

        /* Check for error.  */
        if (status != NX_SOC_OK)
        {
            printf("Error on Client socket %d connect\n");
            soc_close(sock_tcp_client);
            return;

        }
        /* Get and print source and destination information */
        printf("Client socket %d connected \n", sock_tcp_client);

        status = getsockname(sock_tcp_client, (struct sockaddr *)&localAddr6,
                             &length);
        printf("Client port = %lu , Client = 0x%x 0x%x 0x%x 0x%x,",
                ntohs(localAddr6.sin6_port),
                ntohl(localAddr6.sin6_addr._S6_un._S6_u32[0]),
                ntohl(localAddr6.sin6_addr._S6_un._S6_u32[1]),
                ntohl(localAddr6.sin6_addr._S6_un._S6_u32[2]),
                ntohl(localAddr6.sin6_addr._S6_un._S6_u32[3]));

        length = sizeof(struct sockaddr_in6);
        status = getpeername(sock_tcp_client, (struct sockaddr *)
                             &echoServAddr6, &length);
        printf("Remote port = %lu, Remote IP = 0x%x 0x%x 0x%x 0x%x \n",
                ntohs(echoServAddr6.sin6_port),
                ntohl(echoServAddr6.sin6_addr._S6_un._S6_u32[0]),
                ntohl(echoServAddr6.sin6_addr._S6_un._S6_u32[1]),
                ntohl(echoServAddr6.sin6_addr._S6_un._S6_u32[2]),
                ntohl(echoServAddr6.sin6_addr._S6_un._S6_u32[3]));

        /* Now receive the echoed packet from the server */
        while(1)
        {

            printf("Client sock %d sending packet to server\n",
                   sock_tcp_client);

            status = send(sock_tcp_client, "Hello", (strlen("Hello")+1), 0);

            if (status == NX_SOC_ERROR)
            {
                printf("Error on Client Socket (%d) send \n",
                       sock_tcp_client);
            }
```

```
            else
            {
                printf("Client socket %d sent message: Hello\n",
                       sock_tcp_client);
            }

            status = recv(sock_tcp_client, (VOID *)Client_Rcv_Buffer,
                          CLIENT_BUFFER_SIZE, 0);

            if (status <= 0)
            {

                if (status < 0)
                {
                    printf("Error on Client receiving on socket %d \n",
                           sock_tcp_client);
                }
                else
                {
                    printf("Client received no data on socket %d\n",
                           sock_tcp_client);
                }

                break;
            }
            else
            {
                printf("Client socket %d received %d bytes and this message:
                        %s\n", sock_tcp_client, strlen(Client_Rcv_Buffer),
                        Client_Rcv_Buffer);
            }

        }

        /* close this client socket */
        status = soc_close(sock_tcp_client);

        if (status != NX_SOC_ERROR)
        {
            printf("Client Socket %d closed\n", sock_tcp_client);
        }
        else
        {
            printf("Error on Client Socket %d on close \n", sock_tcp_client);
        }

        /* Make another Client connection...*/

    }
}
```

Chapter 3

NetX Duo BSD Services

This chapter contains a description of all NetX Duo BSD basic services (listed below) in alphabetic order.

*INT accept(INT sockID, struct sockaddr *ClientAddress, INT *addressLength);*

*INT bind (INT sockID, struct sockaddr *localAddress, INT addressLength);*

*INT bsd_initialize(NX_IP *default_ip, NX_PACKET_POOL *default_pool, CHAR *bsd_thread_stack_area, ULONG bsd_thread_stack_size, UINT bsd_thread_priority);*

*INT connect(INT sockID, struct sockaddr *remoteAddress, INT addressLength);*

*INT getpeername(INT sockID, struct sockaddr *remoteAddress, INT *addressLength);*

*INT getsockname(INT sockID, struct sockaddr *localAddress, INT *addressLength);*

*INT ioctl(INT sockID, INT command, INT *result);*

*in_addr_t inet_addr(const_CHAR *buffer);*

*INT inet_aton(const CHAR *cp_arg, struct in_addr *addr);*

CHAR inet_ntoa(struct in_addr address_to_convert);

*const CHAR *inet_ntop(INT af, const VOID *src, CHAR *dst, socklen_t size);*

*INT inet_pton(INT af, const CHAR *src, VOID *dst);*

INT listen(INT sockID, INT backlog);

*INT recvfrom(INT sockID, CHAR *buffer, INT buffersize, INT flags, struct sockaddr *fromAddr, INT *fromAddrLen);*

*INT recv(INT sockID, VOID *rcvBuffer, INT bufferLength, INT flags);*

*INT sendto(INT sockID, CHAR *msg, INT msgLength, INT flags, struct sockaddr *destAddr, INT destAddrLen);*

*INT send(INT sockID, const CHAR *msg, INT msgLength, INT flags);*

*INT select(INT nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);*

```
INT   soc_close ( INT sockID);

INT   socket( INT protocolFamily, INT type, INT protocol);

INT  fcntl(INT sock_ID, UINT flag_type, UINT f_options);

INT  getsockopt(INT sockID, INT option_level, INT option_name, VOID *option_value,
                INT *option_length);

INT  setsockopt(INT sockID, INT option_level, INT option_name,
                const VOID *option_value, INT option_length);

INT  getaddrinfo(const CHAR *node, const CHAR *service, const struct addrinfo *hints,
                struct addrinfo **res);

VOID freeaddrinfo(struct addrinfo *res);

INT  getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host,
                 size_t hostlen, char *serv, size_t servlen, int flags);

VOID nx_bsd_set_service_list(struct NX_BSD_SERVICE_LIST *serv_list_ptr,
                             ULONG serv_list_len);

VOID  FD_SET(INT fd, fd_set *fdset);

VOID  FD_CLR(INT fd, fd_set *fdset);

INT   FD_ISSET(INT fd, fd_set *fdset);

VOID  FD_ZERO (fd_set *fdset);
```