



**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](http://www.expresslogic.com)

**©2002-2013 by Express Logic, Inc.**

All rights reserved. This document and the associated NetX Duo software are the sole property of Express Logic, Inc. Each contains proprietary information of Express Logic, Inc. Reproduction or duplication by any means of any portion of this document without the prior written consent of Express Logic, Inc. is expressly forbidden.

Express Logic, Inc. reserves the right to make changes to the specifications described herein at any time and without notice in order to improve design or reliability of NetX Duo. The information in this document has been carefully checked for accuracy; however, Express Logic, Inc. makes no warranty pertaining to the correctness of this document.

**Trademarks**

NetX, NetX Duo, Piconet, and UDP Fast Path are trademarks of Express Logic, Inc. ThreadX is a registered trademark of Express Logic, Inc.

All other product and company names are trademarks or registered trademarks of their respective holders.

**Warranty Limitations**

Express Logic, Inc. makes no warranty of any kind that the NetX Duo products will meet the USER's requirements, or will operate in the manner specified by the USER, or that the operation of the NetX Duo products will operate uninterrupted or error free, or that any defects that may exist in the NetX Duo products will be corrected after the warranty period. Express Logic, Inc. makes no warranties of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose, with respect to the NetX Duo products. No oral or written information or advice given by Express Logic, Inc., its dealers, distributors, agents, or employees shall create any other warranty or in any way increase the scope of this warranty, and licensee may not rely on any such information or advice.

Part Number: 000-1050

Revision 5.3

# Contents

---

Chapter 1 Introduction to NetX Duo BSD.....	4
BSD Socket API Compliancy Wrapper Source .....	4
Chapter 2 Installation and Use of NetX Duo BSD .....	5
Product Distribution .....	5
NetX Duo BSD Installation .....	5
Building ThreadX and NetX Duo components of a BSD Application .....	6
Using NetX Duo BSD .....	6
NetX Duo BSD Raw Socket Support.....	7
Eliminating Internal BSD Thread .....	8
NetX Duo BSD Limitations .....	8
Configuration Options.....	9
Small IPv4 Example .....	13
Small IPv6 Example System .....	21
Chapter 3 NetX Duo BSD Services.....	29

# 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:

<i>nxd_bsd.c</i> :	Wrapper source code
<i>nxd_bsd.h</i> :	Main header file

Sample demo programs:

<i>bsd_netxdue_demo_tcp.c</i>	<i>Demo with a single TCP server and client (IPv6/IPv4)</i>
<i>bsd_demo_udp.c</i>	<i>Demo with two UDP peers (IPv4 only)</i>
<i>bsd_demo_single_client.c</i> :	<i>Demo with single TCP client and server (IPv4 only)</i>
<i>bsd_demo_tcp_multi_clients.c</i> :	<i>Demo with TCP multiple clients/one server (IPv4 only)</i>
<i>bsd_demo_tcp_server_threads.c</i> :	<i>Demo multiple server threads/multiple clients (IPv4 only)</i>
<i>bsd_demo_tcp.h</i> :	<i>Header file for IPv4 BSD demo applications</i>

## 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:

<b>nxd_bsd.h</b>	Header file for NetX Duo BSD
<b>nxd_bsd.c</b>	C Source file for NetX Duo BSD
<b>nxd_bsd.pdf</b>	User Guide for NetX Duo BSD

Demo files:

**bsd\_netxduo\_demo\_tcp.c**  
**bsd\_demo\_udp.c**  
**bsd\_demo\_single\_client.c:**  
**bsd\_demo\_tcp\_multi\_clients.c:**  
**bsd\_demo\_tcp\_server\_threads.c:**  
**bsd\_demo\_tcp.h**

### 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 ThreadX and NetX Duo components of a BSD Application

---

### ThreadX

The ThreadX library must be configured to include the 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, user is free to use 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.

## **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 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.

## **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:

<b>Define</b>	<b>Meaning</b>
<b>NX_BSD_TCP_WINDOW</b>	Used in TCP socket create calls. 64k is typical window size for 100Mb Ethernet. The default value is 65535.
<b>NX_BSD_SOCKFD_START</b>	This is the logical index for the BSD socket file descriptor start value. By default this option is 32.
<b>NX_BSD_MAX_SOCKETS</b>	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.
<b>NX_BSD_MAX_LISTEN_BACKLOG</b>	This specifies the size of the listen queue ('backlog') for BSD TCP sockets. The default value is 5.
<b>NX_CPU_TICKS_PER_SECOND</b>	Specifies the number of timer ticks per second. The default is 100 ticks per second for a ThreadX timer running at 100 Hz.

<b>NX_MICROSECOND_PER_CPU_TICK</b>	Specifies the number of microseconds per scheduler timer tick
<b>NX_BSD_TIMEOUT</b>	Specifies the timeout in timer ticks on NetX Duo internal calls required by BSD. The default value is (20 * NX_CPU_TICKS_PER_SECOND).
<b>NX_BSD_PRINT_ERROR</b>	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 <i>nxd_bsd.h</i>
<b>NX_BSD_TIMEOUT_PROCESS_IN_TIMER</b>	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”.
<b>NX_BSD_INHERIT_LISTENER_SOCKET_SETTINGS</b>	<p>If set, secondary sockets inherit the certain socket features) of the master (listening) socket. These are:</p> <ul style="list-style-type: none"> <li>non-blocking (O_NONBLOCK)</li> <li>port re-use (SO_REUSEADDR)</li> <li>receive window size (SO_RCVBUF)</li> </ul> <p>By default this option is enabled.</p>

The following list of NetX Duo BSD socket options can be enabled (or disabled) at run time 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 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 <i>getsockopt</i> 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 <i>nx_user.h</i> . 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 *IPPROTO* 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 *IPPROTO*.

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

<i>IP_MULTICAST_TTL</i>	This sets the time to live for UDP sockets. The default value is <i>NX_IP_TIME_TO_LIVE</i> (0x80) when the socket is created. This value can be overridden by calling <i>setsockopt</i> with this socket option.
<i>IP_RAW_IPV6_HDRINCL</i>	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.
<i>IP_ADD_MEMBERSHIP</i>	If set, this options enables the BSD socket (applies only to UDP sockets) to join the specified IGMP group.
<i>IP_DROP_MEMBERSHIP</i>	If set, this options enables the BSD socket (applies only to UDP sockets) to leave the specified IGMP group.
<i>IP_HDRINCL</i>	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	<p>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.</p> <p>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.</p> <p>This option has no effect on either IPv4 or IPv6 transmission data.</p>

### 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"
7  #include      "nx_api.h"
8  #include      "nxd_bsd.h"
9  #include      <string.h>
10 #include      <stdlib.h>
11
12 #define        DEMO_STACK_SIZE      (16*1024)
13 #define        SERVER_PORT          87
14 #define        CLIENT_PORT          77
15
16 /* Define the ThreadX and NetX object control blocks... */
17
18 TX_THREAD      thread_server;
19 TX_THREAD      thread_client;
20 NX_PACKET_POOL bsd_pool;
```

```

21 NX_IP          bsd_ip;
22
23 /* Define some global data. */
24 CHAR *msg0 = "Client 1:
    ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABCDEFGHIJKLMNOPQRSTUVWXYZ
    STUVWXYZ<>END";

25 INT          maxfd;
26
27 /* Define the counters used in the demo application... */
28
29 ULONG          error_counter;
30
31 /* Define fd_sets for the BSD server socket. */
32 fd_set          master_list, read_ready;
33
34 /* Define thread prototypes. */
35
36 VOID          thread_server_entry(ULONG thread_input);
37 VOID          thread_client_entry(ULONG thread_input);
38 void          _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);
39
40
41 /* Define main entry point. */
42
43 int main()
44 {
45
46     /* Enter the ThreadX kernel. */
47     tx_kernel_enter();
48 }
49
50 /* Define what the initial system looks like. */
51
52 void tx_application_define(void *first_unused_memory)
53 {
54     CHAR *pointer;
55     UINT status;
56
57     /* Setup the working pointer. */
58     pointer = (CHAR *) first_unused_memory;
59
60     /* Create a server thread. */
61     tx_thread_create(&thread_server, "Server", thread_server_entry, 0,
62                     pointer, DEMO_STACK_SIZE, 8, 8, TX_NO_TIME_SLICE,
63                     TX_AUTO_START);
64
65     pointer = pointer + DEMO_STACK_SIZE;
66
67     /* Create a Client thread. */
68     tx_thread_create(&thread_client, "Client", thread_client_entry, 0,
69                     pointer, DEMO_STACK_SIZE, 16, 16, TX_NO_TIME_SLICE,
70                     TX_AUTO_START);
71
72     pointer = pointer + DEMO_STACK_SIZE;
73
74     /* Initialize the NetX system. */
75     nx_system_initialize();
76
77     /* Create a BSD packet pool. */
78     status = nx_packet_pool_create(&bsd_pool, "NetX BSD Packet Pool", 128,
79                                   pointer, 16384);
79
80     pointer = pointer + 16384;
81     if (status)
82     {
83         error_counter++;
84         printf("Error in creating BSD packet pool\n!");
85     }
86
87     /* Create an IP instance for BSD. */
88     status = nx_ip_create(&bsd_ip, "BSD IP Instance", IP_ADDRESS(1,2,3,4),
89                           0xFFFFFFFFUL, &bsd_pool, _nx_ram_network_driver,
90                           pointer, 2048, 1);
91
92     pointer = pointer + 2048;
93
94     if (status)
95     {
96         error_counter++;
97         printf("Error creating BSD IP instance\n!");
98     }
99 }

```

```

95     }
96
97     /* Enable ARP and supply ARP cache memory for BSD IP Instance */
98     status = nx_arp_enable(&bsd_ip, (void *) pointer, 1024);
99     pointer = pointer + 1024;
100
101     /* Check ARP enable status. */
102     if (status)
103     {
104         error_counter++;
105         printf("Error in Enable ARP \n");
106     }
107
108     /* Enable TCP processing for BSD IP instances. */
109
110     status = nx_tcp_enable(&bsd_ip);
111
112     /* Check TCP enable status. */
113     if (status)
114     {
115         error_counter++;
116         printf("Error in Enable TCP \n");
117     }
118
119     /* Now initialize BSD Scket Wrapper */
120     status = bsd_initialize (&bsd_ip, &bsd_pool, pointer, 2048, 2);
121 }
122
123
124 /* Define the Server thread. */
125 CHAR      Server_Rcv_Buffer[100];
126
127 VOID  thread_server_entry(ULONG thread_input)
128 {
129
130     INT      status, sock, sock_tcp_server;
131     ULONG    actual_status;
132     INT      ClientLen;
133     INT      i;
134     UINT     is_set = NX_FALSE;
135     struct   sockaddr_in serverAddr;
136     struct   sockaddr_in ClientAddr;
137
138
139     tx_thread_sleep(100);
140     status = nx_ip_status_check(&bsd_ip, NX_IP_INITIALIZE_DONE,
141                                &actual_status, 100);
142
143     /* Check status... */
144     if (status != NX_SUCCESS)
145     {
146         return;
147     }
148
149     /* Create BSD TCP Socket */
150     sock_tcp_server = socket(AF_INET, SOCK_STREAM, 0);
151
152     if (sock_tcp_server == -1)
153     {
154         printf("Error on Server socket %d create \n", sock_tcp_server);
155         return;
156     }
157
158     printf("Server socket %d created\n", sock_tcp_server);
159
160     /* Set the server port and IP address */
161     memset(&serverAddr, 0, sizeof(serverAddr));
162     serverAddr.sin_family = AF_INET;
163     serverAddr.sin_addr.s_addr = htonl(IP_ADDRESS(1,2,3,4));
164     serverAddr.sin_port = htons(SERVER_PORT);
165
166     /* Bind this server socket */
167     status = bind (sock_tcp_server, (struct sockaddr *) &serverAddr,
168                   sizeof(serverAddr));
169
170     if (status < 0)
171     {
172         printf("Error on Server Socket %d Bind \n", sock_tcp_server);
173         return;
174     }

```



```

174 FD_ZERO(&master_list);
175 FD_ZERO(&read_ready);
176 FD_SET(sock_tcp_server,&master_list);
177 maxfd = sock_tcp_server;
178
179 /* Now listen for any client connections for this server socket */
180 status = listen (sock_tcp_server, 5);
181 if (status < 0)
182 {
183     printf("Error on Server Socket %d Listen\n", sock_tcp_server);
184     return;
185 }
186 else
187     printf("Server socket %d listen complete\n", sock_tcp_server);
188
189 /* All set to accept client connections */
190
191 /* Loop to create and establish server connections. */
192 while(1)
193 {
194
195     printf("\n");
196
197     read_ready = master_list;
198
199     tx_thread_sleep(20); /* Allow some time to other threads too */
200
201     /* Let the underlying TCP stack determine the timeout. */
202     status = select(maxfd + 1, &read_ready, 0, 0, 0);
203
204     if ((status == 0xFFFFFFFF) || (status == 0))
205     {
206
207         printf("Error with select. Status 0x%x\n", status);
208
209         continue;
210     }
211
212     /* Check for a connection request. */
213     is_set = FD_ISSET(sock_tcp_server, &read_ready);
214
215     if(is_set)
216     {
217
218         clientlen = sizeof(ClientAddr);
219
220         sock = accept(sock_tcp_server,(struct sockaddr*)&ClientAddr,
221                     &Clientlen);
222
223         /* Add this new connection to our master list */
224         FD_SET(sock, &master_list);
225
226         if ( sock > maxfd)
227         {
228             maxfd = sock;
229         }
230
231         continue;
232     }
233
234     /* Check the set of 'ready' sockets, e.g connected to remote host and
235     waiting for notice of packets received. */
236     for (i = NX_BSD_SOCKETFD_START; i < (maxfd+1+NX_BSD_SOCKETFD_START); i++)
237     {
238
239         if ((i != sock_tcp_server) &&
240             (FD_ISSET(i , &master_list)) &&
241             (FD_ISSET(i , &read_ready)))
242         {
243
244             while(1)
245             {
246
247                 status = recv(i, (VOID *)Server_Rcv_Buffer, 100, 0);
248
249                 if (status == 0)
250                 {
251                     printf("(Server received no data from client on
252                        socket %d)\n", i);
253                     break;

```

```

253         }
254         else if (status == NX_SOC_ERROR)
255         {
256             printf("Error on Server receiving data from client on
                socket %d\n", i);
257             break;
258         }
259
260         printf("Server socket %d received %d bytes: %s\n ",
261             i, strlen(Server_Rcv_Buffer), Server_Rcv_Buffer);
262
263
264         status = send(i, "Hello\n", strlen("Hello\n")+1, 0);
265
266         if (status == NX_SOC_ERROR)
267         {
268             printf("Error on Server socket %d sending data to
                Client\n", i);
269         }
270         else
271         {
272             printf("Server socket %d message sent to Client:
                Hello\n", i);
273         }
274     }
275
276     /* Close this socket */
277     status = soc_close(i);
278
279     if (status != NX_SOC_ERROR)
280     {
281         printf("Server socket %d closed \n", i);
282     }
283     else
284     {
285         printf("Error on closing Server socket %d \n", i);
286     }
287 }
288 }
289
290 /* Loop back to check any next client connection */
291 }
292 }
293
294 CHAR        Client_Rcv_Buffer[100];
295
296 VOID thread_client_entry(ULONG thread_input)
297 {
298
299     INT        status;
300     INT        sock_tcp_client, length;
301     struct     sockaddr_in echoServAddr;
302     struct     sockaddr_in localAddr;
303
304
305     /* Let the server side get set up. */
306     tx_thread_sleep(200);
307
308     /* Set local port for displaying IP address and port. */
309     memset(&localAddr, 0, sizeof(localAddr));
310     localAddr.sin_family = AF_INET;
311     localAddr.sin_addr.s_addr = htonl(IP_ADDRESS(1,2,3,4));
312     localAddr.sin_port = htons(CLIENT_PORT);
313
314     /* Set server port and IP address which we need to connect. */
315     memset(&echoServAddr, 0, sizeof(echoServAddr));
316     echoServAddr.sin_family = AF_INET;
317     echoServAddr.sin_addr.s_addr = htonl(IP_ADDRESS(1,2,3,4));
318     echoServAddr.sin_port = htons(SERVER_PORT);
319
320     /* Now make client connections with the server. */
321     while (1)
322     {
323
324         printf("\n");
325         /* Create BSD TCP Socket */
326         sock_tcp_client = socket( AF_INET, SOCK_STREAM, 0);
327
328         if (sock_tcp_client == -1)
329         {
330             printf("Error on Client socket %d create \n", sock_tcp_client);

```

```

331         return;
332     }
333
334     printf("Client socket %d created\n", sock_tcp_client);
335
336     /* Now connect this client to the server */
337     status = connect(sock_tcp_client, (struct sockaddr *)&echoServAddr,
338                     sizeof(echoServAddr));
339
340     /* Check for error. */
341     if (status != OK)
342     {
343         printf("Error on Client socket %d connect\n", sock_tcp_client);
344         soc_close(sock_tcp_client);
345         return;
346     }
347     /* Get and print source and destination information */
348     printf("Client socket %d connected \n", sock_tcp_client);
349
350     status = getsockname(sock_tcp_client, (struct sockaddr *)&localAddr,
351                         &length);
352     printf("Client port = %lu, Client = 0x%x,",
353           htonl(localAddr.sin_port), htonl(localAddr.sin_addr.s_addr));
354     length = sizeof(struct sockaddr_in);
355     status = getpeername(sock_tcp_client, (struct sockaddr *)
356                         &echoServAddr, &length);
357     printf("Remote port = %lu, Remote IP = 0x%x \n",
358           htonl(echoServAddr.sin_port),
359           htonl(echoServAddr.sin_addr.s_addr));
360
361     /* Now receive the echoed packet from the server */
362     while(1)
363     {
364         printf("Client sock %d sending packet to server\n",
365               sock_tcp_client);
366
367         status = send(sock_tcp_client, "Hello", (strlen("Hello")+1), 0);
368
369         if (status == ERROR)
370         {
371             printf("Error: Client Socket (%d) send \n", sock_tcp_client);
372         }
373         else
374         {
375             printf("Client socket %d sent message Hello\n",
376                   sock_tcp_client);
377         }
378
379         status = recv(sock_tcp_client, (VOID *)Client_Rcv_Buffer, 100, 0);
380
381         if (status <= 0)
382         {
383             if (status < 0)
384             {
385                 printf("Error on Client receiving on socket %d \n",
386                       sock_tcp_client);
387             }
388             else
389             {
390                 printf("Nothing received by client on socket %d\n",
391                       sock_tcp_client);
392             }
393             break;
394         }
395         else
396         {
397             printf("Client socket %d received %d bytes: %s\n",
398                   sock_tcp_client,
399                   strlen(Client_Rcv_Buffer), Client_Rcv_Buffer);
400         }
401     }
402
403     /* close this client socket */
404     status = soc_close(sock_tcp_client);
405
406     if (status != ERROR)

```

```
401     {
402         printf("Client Socket %d closed\n", sock_tcp_client);
403     }
404     else
405     {
406         printf("Error on Client Socket %d on close \n", sock_tcp_client);
407     }
408
409     /* Make another Client connection...*/
410
411 }
412 }
413
```

## 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.

```

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,
3  disconnection, sending, and receiving using a simulated Ethernet driver. */
4
5
6  #include      "tx_api.h"
7  #include      "nx_api.h"
8  #include      "nxd_bsd.h"
9  #include      <string.h>
10 #include      <stdlib.h>
11
12 #define        DEMO_STACK_SIZE      (16*1024)
13 #define        SERVER_PORT          87
14 #define        CLIENT_PORT          77
15
16 /* Define the ThreadX and NetX object control blocks... */
17
18 TX_THREAD      thread_server;
19 TX_THREAD      thread_client;
20 NX_PACKET_POOL bsd_pool;
21 NX_IP           bsd_ip;
22
23 /* Define some global data. */
24 CHAR *msg0 = "Client 1:
                ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABC
                DEFGHIJKLMNOPQRSTUVWXYZ<>END";
25
26 INT            maxfd;
27
28 /* Define the counters used in the demo application... */
29
30 ULONG          error_counter;
31
32 /* Define fd_sets for the BSD server socket. */
33 fd_set         master_list, read_ready;
34
35 /* Define thread prototypes. */
36
37 VOID          thread_server_entry(ULONG thread_input);
38 VOID          thread_client_entry(ULONG thread_input);
39 void          _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);
40
41
42 /* Define main entry point. */
43
44 int main()
45 {
46     /* Enter the ThreadX kernel. */
47     tx_kernel_enter();
48 }
49
50 /* Define what the initial system looks like. */
51
52 void tx_application_define(void *first_unused_memory)
53 {
54     CHAR *pointer;
55     UINT status;
56
57
58     /* Setup the working pointer. */
59     pointer = (CHAR *) first_unused_memory;
60
61     /* Create a server thread. */
62     tx_thread_create(&thread_server, "Server", thread_server_entry, 0,
63                     pointer, DEMO_STACK_SIZE, 8, 8,
64                     TX_NO_TIME_SLICE, TX_AUTO_START);
65
66     pointer = pointer + DEMO_STACK_SIZE;
67
68     /* Create a Client thread. */

```

```

69     tx_thread_create(&thread_client, "Client", thread_client_entry, 0,
70                     pointer, DEMO_STACK_SIZE, 16, 16,
71                     TX_NO_TIME_SLICE, TX_AUTO_START);
72     pointer = pointer + DEMO_STACK_SIZE;
73
74     /* Initialize the NetX system. */
75     nx_system_initialize();
76
77     /* Create a BSD packet pool. */
78     status = nx_packet_pool_create(&bsd_pool, "NetX BSD Packet Pool",
79                                   128, pointer, 16384);
80     pointer = pointer + 16384;
81     if (status)
82     {
83         error_counter++;
84         printf("Error in creating BSD packet pool\n!");
85     }
86
87     /* Create an IP instance for BSD. */
88     status = nx_ip_create(&bsd_ip, "BSD IP Instance", IP_ADDRESS(1,2,3,4),
89                          0xFFFFFFFFUL, &bsd_pool, _nx_ram_network_driver,
90                          pointer, 2048, 1);
91     pointer = pointer + 2048;
92     if (status)
93     {
94         error_counter++;
95         printf("Error creating BSD IP instance\n!");
96     }
97
98     /* Enable ARP and supply ARP cache memory for BSD IP Instance */
99     status = nx_arp_enable(&bsd_ip, (void *) pointer, 1024);
100     pointer = pointer + 1024;
101
102     /* Check ARP enable status. */
103     if (status)
104     {
105         error_counter++;
106         printf("Error in enable ARP on BSD IP instance\n");
107     }
108
109     /* Enable TCP processing for BSD IP instances. */
110
111     status = nx_tcp_enable(&bsd_ip);
112
113     /* Check TCP enable status. */
114     if (status)
115     {
116         error_counter++;
117         printf("Error in Enable TCP \n");
118     }
119
120     /* Now initialize BSD Scket wrapper */
121     status = bsd_initialize(&bsd_ip, &bsd_pool, pointer, 2048, 2);
122
123     /* Check BSD initialize status. */
124     if (status)
125     {
126         error_counter++;
127         printf("Error in BSD initialize \n");
128     }
129
130     pointer = pointer + 2048;
131 }
132
133 /* Define the Server thread. */
134 CHAR    Server_Rcv_Buffer[100];
135
136 VOID    thread_server_entry(ULONG thread_input)
137 {
138
139
140
141     INT        status, sock, sock_tcp_server;
142     ULONG      actual_status;
143     INT        clientlen;
144     INT        i;
145     UINT       is_set = NX_FALSE;
146     NXD_ADDRESS ip_address;

```

```

147 struct      sockaddr_in6 serverAddr;
148 struct      sockaddr_in6 ClientAddr;
149 UINT        iface_index, address_index;
150
151
152     status = nx_ip_status_check(&bsd_ip, NX_IP_INITIALIZE_DONE,
                                &actual_status, 100);
153
154     /* Check status... */
155     if (status != NX_SUCCESS)
156     {
157         return;
158     }
159
160     /* Enable IPV6 */
161     status = nxd_ipv6_enable(&bsd_ip);
162     if((status != NX_SUCCESS) && (status != NX_ALREADY_ENABLED))
163     {
164         printf("Error with IPV6 enable 0x%x\n", status);
165         return;
166     }
167
168     /* Enable ICMPv6 */
169     status = nxd_icmp_enable(&bsd_ip);
170     if(status)
171     {
172         printf("Error with ICMPv6 enable 0x%x\n", status);
173         return;
174     }
175
176     /* Set the primary interface for our DNS IPV6 addresses. */
177     iface_index = 0;
178
179     /* This assumes we are using the primary network interface (index 0). */
180     status = nxd_ipv6_address_set(&bsd_ip, iface_index, NX_NULL, 10,
                                   &address_index);
181
182     if (status)
183     {
184         return;
185     }
186
187     /* Set ip_0 interface address. */
188     ip_address.nxd_ip_version = NX_IP_VERSION_V6;
189     ip_address.nxd_ip_address.v6[0] = htonl(0x20010db8);
190     ip_address.nxd_ip_address.v6[1] = htonl(0x0000f101);
191     ip_address.nxd_ip_address.v6[2] = 0;
192     ip_address.nxd_ip_address.v6[3] = htonl(0x101);
193
194     /* Set the host global IP address. We are assuming a 64
195     bit prefix here but this can be any value (< 128). */
196     status = nxd_ipv6_address_set(&bsd_ip, iface_index, &ip_address, 64,
                                   &address_index);
197
198     if (status)
199     {
200         return;
201     }
202
203     /* Wait for IPV6 stack to finish DAD process. */
204     tx_thread_sleep(400);
205
206     /* Create BSD TCP Socket */
207     sock_tcp_server = socket(AF_INET6, SOCK_STREAM, 0);
208
209     if (sock_tcp_server == -1)
210     {
211         printf("\nError: BSD TCP Server socket create \n");
212         return;
213     }
214
215     printf("\nBSD TCP Server socket created %lu \n", sock_tcp_server);
216
217     /* Set the server port and IP address */
218     memset(&serverAddr, 0, sizeof(serverAddr));
219     serverAddr.sin6_addr._S6_un._S6_u32[0] = htonl(0x20010db8);
220     serverAddr.sin6_addr._S6_un._S6_u32[1] = htonl(0xf101);
221     serverAddr.sin6_addr._S6_un._S6_u32[2] = 0x0;
222     serverAddr.sin6_addr._S6_un._S6_u32[3] = htonl(0x0101);
223     serverAddr.sin6_port = htons(SERVER_PORT);
224     serverAddr.sin6_family = AF_INET6;
225
226     /* Bind this server socket */
227     status = bind(sock_tcp_server, (struct sockaddr *) &serverAddr,

```



```

        sizeof(serverAddr));
225
226 if (status < 0)
227 {
228     printf("Error: Server Socket Bind \n");
229     return;
230 }
231
232 FD_ZERO(&master_list);
233 FD_ZERO(&read_ready);
234 FD_SET(sock_tcp_server,&master_list);
235 maxfd = sock_tcp_server;
236
237 /* Now listen for any client connections for this server socket */
238 status = listen(sock_tcp_server, 5);
239 if (status < 0)
240 {
241     printf("Error: Server Socket Listen\n");
242     return;
243 }
244 else
245     printf("Server Listen complete\n");
246
247 /* All set to accept client connections */
248 printf("Now accepting client connections\n");
249
250 /* Loop to create and establish server connections. */
251 while(1)
252 {
253
254     printf("\n");
255
256     read_ready = master_list;
257
258     tx_thread_sleep(20); /* Allow some time to other threads too */
259
260     /* Let the underlying TCP stack determine the timeout. */
261     status = select(maxfd + 1, &read_ready, 0, 0, 0);
262
263     if ( (status == 0xFFFFFFFF) || (status == 0) )
264     {
265
266         printf("Error with select? Status 0x%x. Try again\n", status);
267
268         continue;
269     }
270
271     /* Detected a connection request. */
272     is_set = FD_ISSET(sock_tcp_server,&read_ready);
273
274     if(is_set)
275     {
276
277         clientlen = sizeof(ClientAddr);
278
279         sock = accept(sock_tcp_server,
280                     (struct sockaddr*)&ClientAddr,
281                     &clientlen);
282
283         /* Add this new connection to our master list */
284         FD_SET(sock, &master_list);
285
286         if ( sock > maxfd)
287         {
288             printf("New connection %d\n", sock);
289
290             maxfd = sock;
291         }
292         continue;
293     }
294
295     /* Check the set of 'ready' sockets, e.g connected to remote host and
296     waiting for notice of packets received. */
297     for (i = NX_BSD_SOCKETFD_START; i < (maxfd+1+NX_BSD_SOCKETFD_START); i++)
298     {
299         if ((i != sock_tcp_server) &&
300             (FD_ISSET(i, &master_list)) &&
301             (FD_ISSET(i, &read_ready)))
302     {

```

```

303
304         while(1)
305         {
306
307             status = recv(i, (VOID *)Server_Rcv_Buffer, 100, 0);
308
309             if (status == 0)
310             {
311                 printf("(Server socket %d received no data from
312                     Client)\n", i);
313                 break;
314             }
315             else if (status == 0xFFFFFFFF)
316             {
317                 printf("Error on Server socket %d receiving data from
318                     Client\n", i);
319                 break;
320             }
321
322             printf("Server socket %d received from client %lu bytes:
323                 %s\n ", i, strlen(Server_Rcv_Buffer),
324                 Server_Rcv_Buffer);
325
326             status = send(i, "Hello\n", strlen("Hello\n")+1, 0);
327
328             if (status == ERROR)
329             {
330                 printf("Error on Server socket %d sending data to
331                     Client \n", i);
332             }
333             else
334             {
335                 printf("Server socket %d message sent to Client:
336                     Hello\n", i);
337             }
338
339             /* Close this socket */
340             status = soc_close(i);
341
342             if (status != ERROR)
343             {
344                 printf("Server socket %d closing\n", i);
345             }
346             else
347             {
348                 printf("Error on Server socket %d closing\n", i);
349             }
350         }
351     }
352     /* Loop back to check any next client connection */
353 }
354
355 #define CLIENT_BUFFER_SIZE 100
356 CHAR Client_Rcv_Buffer[CLIENT_BUFFER_SIZE];
357
358 VOID thread_client_entry(ULONG thread_input)
359 {
360
361     INT status;
362     INT sock_tcp_client, length;
363     struct sockaddr_in6 echoServAddr6;
364     struct sockaddr_in6 localAddr6; address */
365
366     /* wait for the server side to get set up, including the DAD process. */
367     tx_thread_sleep(500);
368
369     /* ICMPv6 and IPv6 should already be enabled on the IP instance
370     by the server thread entry function. */
371
372     /* Further the IPv6 address is already established with the IP instance.
373     so no need to wait for DAD completion. */
374
375     /* Set local port and IP address (used only for getsockname call). */
376     memset(&localAddr6, 0, sizeof(localAddr6));
377

```

```

378 localAddr6.sin6_addr._S6_un._S6_u32[0] = htonl(0x20010db8);
379 localAddr6.sin6_addr._S6_un._S6_u32[1] = htonl(0xf101);
380 localAddr6.sin6_addr._S6_un._S6_u32[2] = 0x0;
381 localAddr6.sin6_addr._S6_un._S6_u32[3] = htonl(0x0101);
382 localAddr6.sin6_port = htons(CLIENT_PORT);
383 localAddr6.sin6_family = AF_INET6;
384
385 /* Set Server port and IP address to connect to the TCP server. */
386 memset(&echoServAddr6, 0, sizeof(echoServAddr6));
387 echoServAddr6.sin6_addr._S6_un._S6_u32[0] = htonl(0x20010db8);
388 echoServAddr6.sin6_addr._S6_un._S6_u32[1] = htonl(0xf101);
389 echoServAddr6.sin6_addr._S6_un._S6_u32[2] = 0x0;
390 echoServAddr6.sin6_addr._S6_un._S6_u32[3] = htonl(0x0101);
391 echoServAddr6.sin6_port = htons(SERVER_PORT);
392 echoServAddr6.sin6_family = AF_INET6;
393
394 /* Now make client connections with the server. */
395 while (1)
396 {
397     printf("\n");
398     /* Create BSD TCP Socket */
399
400     sock_tcp_client = socket(AF_INET6, SOCK_STREAM, 0);
401
402     if (sock_tcp_client == -1)
403     {
404         printf("Error on Client socket %d create \n");
405         return;
406     }
407
408     printf("Client socket %d created \n", sock_tcp_client);
409
410     /* Now connect this client to the server */
411     status = connect(sock_tcp_client, (struct sockaddr *)&echoServAddr6,
412                     sizeof(echoServAddr6));
413
414     /* Check for error. */
415     if (status != NX_SOC_OK)
416     {
417         printf("Error on Client socket %d connect\n");
418         soc_close(sock_tcp_client);
419         return;
420     }
421
422     /* Get and print source and destination information */
423     printf("Client socket %d connected \n", sock_tcp_client);
424
425     status = getsockname(sock_tcp_client, (struct sockaddr *)&localAddr6,
426                          &length);
427     printf("Client port = %lu, Client = 0x%x 0x%x 0x%x 0x%x,",
428            ntohs(localAddr6.sin6_port),
429            ntohl(localAddr6.sin6_addr._S6_un._S6_u32[0]),
430            ntohl(localAddr6.sin6_addr._S6_un._S6_u32[1]),
431            ntohl(localAddr6.sin6_addr._S6_un._S6_u32[2]),
432            ntohl(localAddr6.sin6_addr._S6_un._S6_u32[3]));
433
434     length = sizeof(struct sockaddr_in6);
435     status = getpeername(sock_tcp_client, (struct sockaddr *)&echoServAddr6,
436                          &length);
437     printf("Remote port = %lu, Remote IP = 0x%x 0x%x 0x%x 0x%x \n",
438            ntohs(echoServAddr6.sin6_port),
439            ntohl(echoServAddr6.sin6_addr._S6_un._S6_u32[0]),
440            ntohl(echoServAddr6.sin6_addr._S6_un._S6_u32[1]),
441            ntohl(echoServAddr6.sin6_addr._S6_un._S6_u32[2]),
442            ntohl(echoServAddr6.sin6_addr._S6_un._S6_u32[3]));
443
444     /* Now receive the echoed packet from the server */
445     while(1)
446     {
447         printf("Client sock %d sending packet to server\n",
448                sock_tcp_client);
449
450         status = send(sock_tcp_client, "Hello", (strlen("Hello")+1), 0);
451
452         if (status == NX_SOC_ERROR)
453         {
454             printf("Error on Client Socket (%d) send \n",
455                    sock_tcp_client);
456         }
457     }
458 }

```

```

454     else
455     {
456         printf("Client socket %d sent message: Hello\n",
457             sock_tcp_client);
458     }
459     status = recv(sock_tcp_client, (VOID *)Client_Rcv_Buffer,
460         CLIENT_BUFFER_SIZE, 0);
461     if (status <= 0)
462     {
463         if (status < 0)
464         {
465             printf("Error on client receiving on socket %d \n",
466                 sock_tcp_client);
467         }
468         else
469         {
470             printf("Client received no data on socket %d\n",
471                 sock_tcp_client);
472         }
473         break;
474     }
475     else
476     {
477         printf("Client socket %d received %d bytes and this message:
478             %s\n", sock_tcp_client, strlen(Client_Rcv_Buffer),
479             Client_Rcv_Buffer);
480     }
481 }
482 /* close this client socket */
483 status = soc_close(sock_tcp_client);
484
485 if (status != NX_SOC_ERROR)
486 {
487     printf("Client Socket %d closed\n", sock_tcp_client);
488 }
489 else
490 {
491     printf("Error on Client Socket %d on close \n", sock_tcp_client);
492 }
493
494 /* Make another Client connection...*/
495 }
496 }
497 }
498
499

```

---

## 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);

VOID FD_CLR(INT fd, fd_set *fdset);

INT  FD_ISSET(INT fd, fd_set *fdset);

VOID FD_SET(INT fd, fd_set *fdset);

VOID FD_ZERO (fd_set *fdset);

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

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

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

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

INT  soc_close ( INT sockID);

```