BSD 4.3 Sockets API Compliancy Wrapper for NetX

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 BSD

The BSD Sockets API Compliancy Wrapper supports some of the basic BSD Sockets API calls with some limitations and utilizes NetX primitives underneath. This BSD Sockets API compatibility layer should perform as fast or slightly faster than typical BSD implementations, since this Wrapper utilizes internal NetX primitives and bypasses basic NetX error checking.

BSD Sockets API Compliancy Wrapper Source

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

The package consists of:

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

Sample demo programs:

bsd_demo_tcp.c
Demo with a single TCP server and client
bsd_demo_udp.c
Demo with two UDP clients and a UDP server

Chapter 2

Installation and Use of NetX BSD

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

Product Distribution

NetX 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:

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

Demo files:
bsd_demo_tcp.c
bsd_demo_udp.c

NetX BSD Installation

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

Building the ThreadX and NetX 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

Before using NetX BSD Services, the NetX library must be built with NX_ENABLE_EXTENDED_NOTIFY_SUPPORT defined (e.g. in *nx_user.h*). By default it is not defined.

Using NetX BSD

Using BSD for NetX is easy. Basically, the application code must include *nx_bsd.h* after it includes *tx_api.h* and *nx_api.h*, in order to use ThreadX and NetX, respectively. Once *nx_bsd.h* is included, the application code is then able to use the BSD services specified later in this guide. The application must also include *nx_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 BSD.

To utilize NetX BSD services, the application must create an IP instance, a packet pool, and initialize BSD services by calling *bsd_initialize.* This is demonstrated in the “Small Example” section later in this guide. The prototype is shown below:

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

The last three parameters are used for creating a thread for performing periodic tasks such as checking for TCP events and define the thread stack space.

Note: in contrast to BSD sockets, which work in network bye order, NetX works in the host byte order of the host processor. For source compatibility reasons, the macros htons(), ntohs(), htonl(), ntohl() have been defined, but do not modify the argument passed.

NetX BSD Limitations

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

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

NetX BSD with DNS Support

If NX_BSD_ENABLE_DNS is defined, NetX 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* for adding 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
- nx_dns_cname_get

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

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.

See **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 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.

Configuration Options

User configurable options in *nx_bsd.h* allow the application to fine tune NetX BSD sockets for its particular requirements. The following is a list of these parameters:

Define	Meaning
NX_BSD_TCP_WINDOW	Used in TCP socket create calls. 65535 is a 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 timer interrupt
NX_BSD_TIMEOUT	Specifies the timeout in timer ticks on NetX 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 disconnect call. The default value is 1.
NX_BSD_PRINT_ERRORS	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 *nx_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 BSD”.
NX_BSD_ENABLE_DNS	If enabled, NetX 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_IPV4_ADDR_MAX_NUM	Maximum number of IPv4 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*. The default value is 5.
NX_BSD_IPV4_ADDR_PER_HOST	Defines maximum IPv4 addresses stored per DNS query. The default value is 5.

BSD Socket Options

The following list of NetX 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. 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 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 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 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 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_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.

Small Example System

An example of how to use NetX BSD is shown in Figure 1.0 below. In this example, the include file *nx_bsd.h* is brought in at line 7. 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 (line 41). 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 line 303 and 309 in *tx_application_define* which sets up the application and is defined on lines 293-361. After IP instance successful creation on line 327, the IP instance is enabled for TCP services on line 350. The last requirement before BSD services can be used is to call *bsd_initialize* on line 360 to set up all data structures and NetX, and ThreadX resources needed by BSD.

In the server thread entry function, *thread_1_entry,* which is defined on lines 381-397, the application waits for the driver to initialize NetX with network parameters. Once this is done, it calls *tcpServer,* defined on lines 146-253, to handle the details of setting up the TCP server socket.

tcpServer creates the master socket by calling the *socket* service on line 159 and binds it to the listening socket using the *bind* call on line 176. It is then configured for listening for connection requests on line 191. Note that the master socket does not accept a connection request. It runs in a continuous loop which calls *select* each time to detect connection requests. A secondary BSD socket chosen from an array of BSD sockets is assigned the connection request after calling the *accept* service on line 218.

On the Client side, the client thread entry function, *thread_0_entry*, defined on lines 366-377, should also wait for NetX to be initialized by the driver. Here we just wait for the server side to do so. It then calls *tcpClient* defined on line 54-142, to handle the details of setting up the TCP client socket and requesting a TCP connection.

The TCP client socket is created on line 68. The socket is bound to the specified IP address and attempts to connect to the TCP server by calling *connect* on line 84. It is now ready to begin sending and receiving packets.

```
1    /* This is a small demo of BSD Wrapper for the high-performance NetX TCP/IP stack.
2       This demo demonstrate TCP connection, disconnection, sending, and receiving using
3       ARP and a simulated Ethernet driver.  */
4
5    #include         "tx_api.h"
6    #include         "nx_api.h"
7    #include         "nx_bsd.h"
8    #include         <string.h>
9    #include         <stdlib.h>
10
```

```
11  #define          DEMO_STACK_SIZE     (16*1024)
12
13
14  /* Define the ThreadX and NetX object control blocks... */
15
16  TX_THREAD        thread_0;
17  TX_THREAD        thread_1;
18
19
20  NX_PACKET_POOL   bsd_pool;
21  NX_IP            bsd_ip;
22
23
24  /* Define the counters used in the demo application...  */
25
26  ULONG            error_counter;
27
28  /* Define fd_set for select call */
29  fd_set           master_list,read_ready,read_ready1;
30
31
32  /* Define thread prototypes.  */
33
34  VOID             thread_0_entry(ULONG thread_input);
35  VOID             thread_1_entry(ULONG thread_input);
36
37  VOID             tcpClient(CHAR *msg0);
38  VOID             tcpServer(VOID);
39  INT              HandleClient(INT  sock);
40
41  VOID             _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);
42
43
44  /* Define main entry point.  */
45
46  int main()
47  {
48
49      /* Enter the ThreadX kernel.  */
50      tx_kernel_enter();
51  }
52
53
54  VOID  tcpClient(CHAR *msg0)
55  {
56
57  INT         status,status1,send_counter;
58  INT         sock_tcp_1,length,length1;
59  struct      sockaddr_in echoServAddr;              /* Echo server address */
60  struct      sockaddr_in localAddr;                 /* Local address */
61  struct      sockaddr_in remoteAddr;                /* Remote address */
62
63  UINT         echoServPort;                         /*  Echo Server Port */
64  CHAR         rcvBuffer1[32];
65
66
67      /* Create BSD TCP Socket */
68      sock_tcp_1 = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP);
69      if (sock_tcp_1 == -1)
70      {
71          printf("\nError: BSD TCP Client socket create \n");
72          return;
73      }
74
75      printf("\nBSD TCP Client socket created %lu \n", sock_tcp_1);
76      /* Fill destination port and IP address */
77      echoServPort = 12;
78      memset(&echoServAddr, 0, sizeof(echoServAddr));
79      echoServAddr.sin_family = PF_INET;
80      echoServAddr.sin_addr.s_addr = htonl(0x01020304);
81      echoServAddr.sin_port = echoServPort;
82
83      /* Now connect this client the server */
84      status1 = connect(sock_tcp_1, (struct sockaddr *)&echoServAddr,
                          sizeof(echoServAddr));
85      /* Check for error.  */
86      if (status1 != OK)
87      {
88          printf("\nError: BSD TCP Client socket Connect, %d \n",sock_tcp_1);
89          status = soc_close(sock_tcp_1);
90          if (status != ERROR)
```

```
91              printf("\nConnect ERROR so BSD Client Socket Closed: %d\n",sock_tcp_1);
92          else
93              printf("\nError: BSD Client Socket close %d\n",sock_tcp_1);
94          return;
95      }

96      /* Get and print source and destination information */
97      printf("\nBSD TCP Client socket: %d connected \n", sock_tcp_1);
98
99      status = getsockname(sock_tcp_1, (struct sockaddr *)&localAddr, &length);
100     printf("Client port = %lu , Client = %lu,", localAddr.sin_port,
               localAddr.sin_addr.s_addr);

101     status = getpeername( sock_tcp_1, (struct sockaddr *) &remoteAddr, &length1);
102     printf("Remote port = %lu, Remote IP = %lu \n", remoteAddr.sin_port,
                remoteAddr.sin_addr.s_addr);
103
104     send_counter = 1;
105
106     /* Now receive the echoed packet from the server */
107     while(1)
108     {
109         tx_thread_sleep(2);
110
111         printf("\nClient sock: %d Sending packet No: %d to
                       server\n",sock_tcp_1,send_counter);

112         status = send(sock_tcp_1,msg0, ( strlen(msg0)+1), 0);
113         if (status == ERROR)
114             printf("Error: BSD Client Socket send %d\n",sock_tcp_1);
115         else
116         {
117             printf("\nMessage sent: %s\n",msg0);
118             send_counter++;
119         }
120
121         status = recv(sock_tcp_1, (VOID *)rcvBuffer1, 31,0);
122         if (status == 0)
123             break;
124
125         rcvBuffer1[status] = 0;
126
127         if (status != ERROR)
128             printf("\nBSD Client Socket: %d received %lu bytes: %s ",
                           sock_tcp_1,strlen(rcvBuffer1),rcvBuffer1);
129         else
130             printf("\nError: BSD Client Socket receive %d \n",sock_tcp_1);
131
132     }
133
134     /* close this client socket */
135     status = soc_close(sock_tcp_1);
136     if (status != ERROR)
137         printf("\nBSD Client Socket Closed %d\n",sock_tcp_1);
138     else
139         printf("\nError: BSD Client Socket close %d \n",sock_tcp_1);
140
141     /* End */
142 }
143
144
145
146 void tcpServer(void)
147 {
148
149 INT         status,status1,sock,sock_tcp_2,i;
150 struct      sockaddr_in echoServAddr;               /* Echo server address */
151 struct      sockaddr_in ClientAddr;
152
153 INT         Clientlen;
154 UINT        echoServPort;                          /*  Echo Server Port */
155
156 INT         maxfd;
157
158     /* Create BSD TCP Server Socket */
159     sock_tcp_2 = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP);
160     if (sock_tcp_2 == -1)
161     {
162         printf("Error: BSD TCP Server socket create\n");
163         return;
164     }
```

```
165     else
166         printf("BSD TCP Server socket created \n");
167
168     /* Now fill server side information */
169     echoServPort = 12;
170     memset(&echoServAddr, 0, sizeof(echoServAddr));
171     echoServAddr.sin_family = PF_INET;
172     echoServAddr.sin_addr.s_addr = htonl(0x01020304);
173     echoServAddr.sin_port = echoServPort;
174
175     /* Bind this server socket */
176     status = bind(sock_tcp_2, (struct sockaddr *) &echoServAddr,
                      sizeof(echoServAddr));
177     if (status < 0)
178     {
179         printf("Error: BSD TCP Server Socket Bind \n");
180         return;
181     }
182     else
183         printf("BSD TCP Server Socket bound \n");
184
185     FD_ZERO(&master_list);
186     FD_ZERO(&read_ready);
187     FD_SET(sock_tcp_2,&master_list);
188     maxfd = sock_tcp_2;
189
190     /* Now listen for any client connections for this server socket */
191     status = listen(sock_tcp_2,5);
192     if (status < 0)
193     {
194         printf("Error: BSD TCP Server Socket Listen\n");
195         return;
196     }
197     else
198         printf("BSD TCP Server Socket Listen complete,    ");
199
200     /* All set to accept client connections */
201     printf("Now accepting client connections\n");
202
203     /* Loop to create and establish server connections.  */
204     while(1)
205     {
206
207         read_ready = master_list;
208         tx_thread_sleep(2);   /* Allow some time to other threads too */
209         status = select(maxfd+1,&read_ready,0,0,0);
210         if (status == ERROR)
211         {
212             continue;
213         }
214
215         status = FD_ISSET(sock_tcp_2,&read_ready);
216         if(status)
217         {
218             sock = accept(sock_tcp_2,(struct sockaddr*)&ClientAddr, &Clientlen);
219
220             /* Add this new connection to our master list */
221             FD_SET(sock,&master_list);
222             if ( sock > maxfd)
223             {
224                 maxfd = sock;
225             }
226
227             continue;
228         }
229         for (i = 0; i < (maxfd+1); i++)
230         {
231             if (( i != sock_tcp_2) && (FD_ISSET(i,&master_list)) &&
                    (FD_ISSET(i,&read_ready)))
232             {
233                 status1 = HandleClient(i);
234                 if (status1 == 0)
235                 {
236                     status1 = soc_close(i);
237                     if (status1 == OK)
238                     {
239                         FD_CLR(i,&master_list);
240                         printf("\nBSD Server Socket:%d closed\n",i);
241                     }
242                     else
243                         printf("\nError:BSD Server Socket:%d close\n",i);
```

```
244                 }
245
246             }
247         }
248
249         /* Loop back to check any next client connection */
250
251     } /* While(1) ends */
252
253 }
254
255 CHAR         rcvBuffer[128];
256
257 INT     HandleClient(INT  sock)
258 {
259
260 INT          status;
261
262
263     status = recv(sock, (VOID *)rcvBuffer, 128,0);
264     if (status == ERROR )
265     {
266         printf("\n BSD Server Socket:%d receive \n",sock);
267         return(ERROR);
268     }
269
270     /* a zero return from a recv() call indicates client is terminated! */
271     if (status == 0)
272     {
273         /* Done with this client , close this secondary server socket */
274         return(status);
275     }
276
277     /* print data received from the client */
278     printf("\nBSD Server Socket:%d received %lu bytes: %s \n", sock,
            strlen(rcvBuffer),rcvBuffer);
279
280     /* And echo the same data to the client */
281     status = send(sock,rcvBuffer, ( strlen(rcvBuffer)+1), 0);
282     if (status == ERROR )
283     {
284         printf("\nError: BSD Server Socket:%d send \n",sock);
285         return(ERROR);
286     }
287     return(status);
288 }
289
290
291 /* Define what the initial system looks like.  */
292
293 void    tx_application_define(void *first_unused_memory)
294 {
295
296 CHAR    *pointer;
297 UINT    status;
298
299     /* Setup the working pointer.  */
300     pointer =  (CHAR *) first_unused_memory;
301
302     /* Create a client thread.  */
303     tx_thread_create(&thread_0, "Client1", thread_0_entry, 0,
304             pointer, DEMO_STACK_SIZE, 2, 2, TX_NO_TIME_SLICE, TX_AUTO_START);
305
306     pointer =  pointer + DEMO_STACK_SIZE;
307
308     /* Create a server thread.  */
309     tx_thread_create(&thread_1, "Server", thread_1_entry, 0,
310             pointer, DEMO_STACK_SIZE,1,1, TX_NO_TIME_SLICE, TX_AUTO_START);
311
312     pointer =  pointer + DEMO_STACK_SIZE;
313
314     /* Initialize the NetX system.  */
315     nx_system_initialize();
316
317     /* Create a BSD packet pool.  */
318     status =  nx_packet_pool_create(&bsd_pool,"NetX BSD Packet Pool", 128,
                                        pointer, 16384);
319     pointer = pointer + 16384;
320     if (status)
321     {
322         error_counter++;
```

```
323         printf("Error in creating BSD packet pool\n!");
324     }
325
326     /* Create an IP instance for BSD.  */
327     status = nx_ip_create(&bsd_ip, "NetX IP Instance 2", IP_ADDRESS(1, 2, 3, 4),
                          0xFFFFFF00UL,  &bsd_pool, _nx_ram_network_driver,
                          pointer, 2048, 1);
328
329     pointer =  pointer + 2048;
330
331     if (status)
332     {
333         error_counter++;
334         printf("Error creating BSD IP instance\n!");
335     }
336
337     /* Enable ARP and supply ARP cache memory for BSD IP Instance */
338     status =  nx_arp_enable(&bsd_ip, (void *) pointer, 1024);
339     pointer = pointer + 1024;
340
341     /* Check ARP enable status.  */
342     if (status)
343     {
344         error_counter++;
345         printf("Error in Enable ARP and supply ARP cache memory to BSD IP
                  instance\n");
346     }
347
348     /* Enable TCP processing for BSD IP instances.  */
349
350     status = nx_tcp_enable(&bsd_ip);
351
352     /* Check TCP enable status.  */
353     if (status)
354     {
355         error_counter++;
356         printf("Error in Enable TCP \n");
357     }
358
359     /* Now initialize BSD Scoket Wrapper */
360     status = bsd_initialize(&bsd_ip, &bsd_pool,pointer, 2048, 1);
361 }
362
363
364 /* Define the test threads.  */
365
366 void    thread_0_entry(ULONG thread_input)
367 {
368
369 CHAR    *msg0 = "Client 1:
                  “ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABCDEFGHIJKLMNOPQRSTUVWXYZ<> \
                  “ABCDEFGHIJKLMNOPQRSTUVWXYZ<>END";
370
371     /* Wait till Server side is all set */
372     tx_thread_sleep(2);
373     while (1)
374     {
375         tcpClient(msg0);
376         tx_thread_sleep(1);
377     }
378 }
379
380 /* Define the server thread entry function. */
381 void    thread_1_entry(ULONG thread_input)
382 {
383
384 UINT    status;
385 ULONG   actual_status;
386
387     /* Ensure the IP instance has been initialized.  */
388     status =  nx_ip_status_check(&bsd_ip, NX_IP_INITIALIZE_DONE, &actual_status,
100);
389
390     /* Check status...  */
391     if (status != NX_SUCCESS)
392     {
393         error_counter++;
394         return;
395     }
396     /* Start a TCP Server */
397     tcpServer();
```

```
398  }
399
```

Chapter 3

NetX BSD Services

This chapter contains a description of all NetX 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);*