



BSD 4.3 Sockets API Compliancy Wrapper for NetX Duo

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

©2002-2012 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 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 5.2

Contents

Chapter 1 Introduction to NetX Duo BSD.....	4
BSD Sockets API Compliancy Wrapper Source.....	4
Chapter 2 Installation and Use of NetX Duo BSD	5
Product Distribution	5
NetX Duo BSD Installation	5
Using NetX Duo BSD	5
NetX Duo BSD Multihome Support	6
NetX Duo BSD Limitations	6
Configuration Options.....	6
Small Example System	8
Chapter 3 List of NetX Duo BSD Services	16
Appendix A NetX Duo BSD Extended Services	17

Chapter 1

Introduction to NetX Duo BSD

The BSD Sockets API Compliancy Wrapper supports some of the basic BSD Sockets API calls, with some limitations and utilizes NetX Duo 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 Duo primitives and bypasses basic NetX Duo error checking.

BSD Sockets API Compliancy Wrapper Source

The Wrapper source code is designed for simplicity and is comprised of only two files, namely *nxd_bsd.h* and *nxd_bsd.c*. The *nxd_bsd.h* file defines all the necessary BSD Sockets API wrapper constants and subroutine prototypes, while *nxd_bsd.c* contains the actual BSD Sockets 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_netxduo_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:

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:	
demo_netxduo_demo_tcp.c	
demo_netxduo_demo_tcp_extended.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.

Using NetX Duo BSD

Using DHCP for NetX Duo is easy. Basically, the application code must include *nxd_bsd.h* after it includes *tx_api.h* and *nx_api.h*, in order to use ThreadX and NetX Duo, respectively. Once *nxd_bsd.h* is included, the application code is then able to make the BSD function calls 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 host application must create an IP instance, a packet pool, and initialize this BSD services by calling *bsd_initialize()*. This is demonstrated in the “Small Example” section later in this document but the prototype is shown below:

```
INT    bsd_initialize(NX_IP *default_ip, NX_PACKET_POOL *default_pool,
                    CHAR *free_memory_ptr);
```

Note that the last parameter *free_memory_ptr* is retained for legacy purposes for use in older versions of NetX BSD; however it is no longer in use and the host application may give it a value of 0.

Note: in contrast to BSD sockets, which works with network Endianism, NetX works in the native Endian mode. For source compatibility reasons, the macros *htons()*, *ntohs()*, *htonl()*, *ntohl()* have been defined, but do not modify the argument passed.

NetX Duo BSD Multihome Support

Multihome support is available in NetX Duo BSD starting with v5.6 depending on the NetX Duo environment. For applications using secondary network interfaces, the host application need update the *NX_MAX_PHYSICAL_INTERFACES* to 2 from the default value of 1 and rebuild the NetX Duo library. In the *tx_application_define* the host application must attach the secondary interface. See the NetX Duo User Guide for more details on multihomed applications.

Thereafter the host application can start socket communications on secondary interfaces using the NetX BSD services such as *send*, *sendto*, *recv* and so on. NetX Duo will automatically handle the details of packet transmission and reception on secondary interfaces.

NetX Duo BSD Limitations

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

select: works with only *fd_set *readfds*, other arguments in this call. *fd_set *writefds*, *fd_set *exceptfds* are not supported.

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

Configuration Options

User configurable options in *nxd_bsd.h* allow the host application to fine tune NetX Duo 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. 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_CPU_TICKS_PER_SECOND	Specifies the number of timer ticks per second. The default is 10 ms per tick.
NX_MICROSECOND_PER_CPU_TICK	Specifies the number of microseconds per timer interrupt
NX_BSD_TIMEOUT	Specifies the timeout in timer ticks on NetX Duo internal calls required by BSD. The default value is $20 \times \text{NX_CPU_TICKS_PER_SECOND}$.

Small Example System

An example of how easy it is to use NetX Duo BSD is described in Figure 1.0. 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 24 and 25. Note that this demo uses a ram (virtual) 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 71 and 77. After IP instance successful creation on line 96, the IP instance is enabled for TCP services on line 119. The last requirement before BSD services can be used is to call *bsd_initialize* on line 129 to set up all data structures and NetX, and ThreadX resources needed by BSD.

The Server thread waits for the driver to initialize NetX Duo with network parameters. Once the link is enabled, the application, if defined to use IPv6 communication (`#ifdef DUO`) enables IPv6 and ICMPv6 services on the IP instance in lines 164 and 172. It then sets its link local address and global address in lines 180 and 189. It allows enough time for NetX Duo (e.g. Duplicate Address Detection) to complete address validation in the thread sleep call on lines 194.

To create an IPv6 enabled socket in BSD, the socket call must set the socket family type to `AF_INET6` in line 200. Otherwise to use IPv4 connection, the socket call should use `AF_INET` for the socket family type (line 202). Once a socket is created, from this point on there is no difference using IPv4 and IPv6 since NetX Duo handles the details internally.

The socket is bound to a the specified IP address and port in the *serverAddr* address type in line 232. It is set to listen in line 248, and then the *select* and *accept* call enables the master socket to detect sockets on its array of available sockets.

The Client thread also waits for the driver initialization to complete. It also, if IPv6 communication is defined, enables IPv6 and ICMPv6 services on the IP instance on lines 393 and 402, and registers the link local and global addresses on lines 411 and 420. After waiting the IPv6 addresses to be validated, the Client thread is ready to create a socket, in lines 432-434.

```

1  /* This is a small demo of BSD wrapper for the high-performance NetX Duo TCP/IP
2     stack. This demo used standard BSD services for TCP connection, disconnection,
3     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
13 #define          DEMO_STACK_SIZE      (16*1024)
14
15
16
17 #define          SERVER_PORT          87
18 #define          CLIENT_PORT          77
19
20 /* Define the ThreadX and NetX object control blocks... */
21
22 TX_THREAD        thread_server;
23 TX_THREAD        thread_client;
24 NX_PACKET_POOL   bsd_pool;
25 NX_IP            bsd_ip;
26
27 /* Define some global data. */
28 CHAR             *msg0 = "Client 1: ABCDEFGHIJKLMNOPQRSTUVWXYZ ";
29 INT              maxfd;
30
31 /* Define the counters used in the demo application... */
32
33 ULONG            error_counter;
34
35 /* Define fd_sets for the BSD server socket. */
36 fd_set           master_list, read_ready;
37
38 /* To send IPV6 packets, define DUO. */
39 #define          DUO
40
41
42
43 /* Define thread prototypes. */
44
45 VOID             thread_server_entry(ULONG thread_input);
46 VOID             thread_client_entry(ULONG thread_input);
47 void             _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);
48
49
50 /* Define main entry point. */
51
52 int main()
53 {
54
55     /* Enter the ThreadX kernel. */
56     tx_kernel_enter();
57 }
58
59 /* Define what the initial system looks like. */
60
61 void tx_application_define(void *first_unused_memory)
62 {
63     CHAR *pointer;
64     UINT status;
65
66
67     /* Setup the working pointer. */
68     pointer = (CHAR *) first_unused_memory;
69
70     /* Create a server thread. */
71     tx_thread_create(&thread_server, "Server", thread_server_entry, 0,
72                     pointer, DEMO_STACK_SIZE, 1, 1, TX_NO_TIME_SLICE,
73                     TX_AUTO_START);
74
75     pointer = pointer + DEMO_STACK_SIZE;
76
77     /* Create a client thread. */
78     tx_thread_create(&thread_client, "Client", thread_client_entry, 0,
79                     pointer, DEMO_STACK_SIZE, 3, 3, TX_NO_TIME_SLICE,
80                     TX_AUTO_START);
81
82     pointer = pointer + DEMO_STACK_SIZE;
83
84     /* Initialize the NetX system. */
85     nx_system_initialize();
86
87     /* Create a BSD packet pool. */
88     status = nx_packet_pool_create(&bsd_pool, "NetX BSD Packet Pool", 128, pointer,

```

```

16384);
88     pointer = pointer + 16384;
89     if (status)
90     {
91         error_counter++;
92         printf("Error in creating BSD packet pool\n!");
93     }
94
95     /* Create an IP instance for BSD. */
96     status = nx_ip_create(&bsd_ip, "BSD IP Instance", IP_ADDRESS(1,2,3,4),
97                         0xFFFFFFFFUL, &bsd_pool, _nx_ram_network_driver,
98                         pointer, 2048, 1);
99     pointer = pointer + 2048;
100
101     if (status)
102     {
103         error_counter++;
104     }
105
106     /* Enable ARP and supply ARP cache memory for BSD IP Instance */
107     status = nx_arp_enable(&bsd_ip, (void *) pointer, 1024);
108     pointer = pointer + 1024;
109
110     /* Check ARP enable status. */
111     if (status)
112     {
113         error_counter++;
114     }
115
116     /* Enable TCP processing for BSD IP instances. */
117
118     status = nx_tcp_enable(&bsd_ip);
119
120     /* Check TCP enable status. */
121     if (status)
122     {
123         error_counter++;
124     }
125
126     /* Now initialize BSD Socket wrapper */
127     bsd_initialize (&bsd_ip, &bsd_pool, pointer);
128 }
129
130 /* Define the Server thread. */
131
132 VOID thread_server_entry(ULONG thread_input)
133 {
134     INT          status, actual_status, sock, sock_tcp_server;
135     CHAR         rcvBuffer[1000];
136     INT          clientlen;
137     INT          i;
138     UINT         is_set = NX_FALSE;
139     #ifdef DUO
140     NXD_ADDRESS ip_address;
141     struct       sockaddr_in6 serverAddr;
142     struct       sockaddr_in6 ClientAddr;
143     #else
144     struct       sockaddr_in serverAddr;
145     struct       sockaddr_in ClientAddr;
146     #endif
147
148     tx_thread_sleep(100);
149     status = nx_ip_status_check(&bsd_ip, NX_IP_INITIALIZE_DONE, &actual_status,
150 100);
151
152     /* Check status... */
153     if (status != NX_SUCCESS)
154     {
155         return;
156     }
157
158     #ifdef DUO
159     /* Enable IPv6 */
160     status = nx_ipv6_enable(&bsd_ip);
161     if((status != NX_SUCCESS) && (status != NX_ALREADY_ENABLED))
162     {
163         printf("Error with IPv6 enable 0x%x\n", status);
164         return;

```

```

169     }
170
171     /* Enable ICMPv6 */
172     status = nxd_icmp_enable(&bsd_ip);
173     if(status)
174     {
175         printf("Error with ICMPv6 enable 0x%x\n", status);
176         return;
177     }
178
179
180     status = nxd_ipv6_linklocal_address_set(&bsd_ip, NX_NULL);
181
182     /* Set ip_0 interface address. */
183     ip_address.nxd_ip_version = NX_IP_VERSION_V6;
184     ip_address.nxd_ip_address.v6[0] = 0x20010db8;
185     ip_address.nxd_ip_address.v6[1] = 0x0000f101;
186     ip_address.nxd_ip_address.v6[2] = 0;
187     ip_address.nxd_ip_address.v6[3] = 0x101;
188
189     status = nxd_ipv6_global_address_set(&bsd_ip, &ip_address, 64);
190     if (status)
191         return;
192
193     /* Wait for IPV6 stack to finish DAD process. */
194     tx_thread_sleep(400);
195
196 #endif
197
198     /* Create BSD TCP Socket */
199 #ifdef DUO
200     sock_tcp_server = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
201 #else
202     sock_tcp_server = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
203 #endif
204
205     if (sock_tcp_server == -1)
206     {
207         printf("\nError: BSD TCP Server socket create \n");
208         return;
209     }
210
211     printf("\nBSD TCP Server socket created %lu \n", sock_tcp_server);
212
213
214     /* Set the server port and IP address */
215 #ifdef DUO
216     memset(&serverAddr, 0, sizeof(serverAddr));
217     serverAddr.sin6_addr._S6_un._S6_u32[0] = 0x20010db8;
218     serverAddr.sin6_addr._S6_un._S6_u32[1] = 0xf101;
219     serverAddr.sin6_addr._S6_un._S6_u32[2] = 0x0;
220     serverAddr.sin6_addr._S6_un._S6_u32[3] = 0x0101;
221     serverAddr.sin6_port = SERVER_PORT;
222     serverAddr.sin6_family = AF_INET6;
223
224 #else
225     memset(&serverAddr, 0, sizeof(serverAddr));
226     serverAddr.sin_family = AF_INET;
227     serverAddr.sin_addr.s_addr = IP_ADDRESS(1,2,3,4);
228     serverAddr.sin_port = SERVER_PORT;
229 #endif
230
231     /* Bind this server socket */
232     status = bind (sock_tcp_server, (struct sockaddr *) &serverAddr,
233                    sizeof(serverAddr));
234
235     if (status < 0)
236     {
237         printf("Error: BSD TCP Server Socket Bind \n");
238         return;
239     }
240     else
241         printf("BSD TCP Server Socket bound \n");
242
243     FD_ZERO(&master_list);
244     FD_ZERO(&read_ready);
245     FD_SET(sock_tcp_server, &master_list);
246     maxfd = sock_tcp_server;
247
248     /* Now listen for any client connections for this server socket */
249     status = listen (sock_tcp_server, 5);

```

```

249     if (status < 0)
250     {
251         printf("Error: BSD TCP Server Socket Listen\n");
252         return;
253     }
254     else
255         printf("BSD TCP Server Socket Listen complete, ");
256
257     /* All set to accept client connections */
258     printf("Now accepting client connections\n");
259
260     /* Loop to create and establish server connections. */
261     while(1)
262     {
263
264         read_ready = master_list;
265
266         tx_thread_sleep(20); /* Allow some time to other threads too */
267
268         /* Let the underlying TCP stack determine the timeout. */
269         status = select(maxfd + 1, &read_ready, 0, 0, 0);
270         if ((status == 0xFFFFFFFF) || (status == 0))
271         {
272             printf("Error with select? Status 0x%x. Try again\n", status);
273             continue;
274         }
275
276         /* Detected a connection request. */
277
278         is_set = FD_ISSET(sock_tcp_server, &read_ready);
279
280         if(is_set)
281         {
282
283             clientlen = sizeof(ClientAddr);
284             sock = accept(sock_tcp_server, (struct sockaddr*)&ClientAddr,
285                           &clientlen);
286
287             /* Add this new connection to our master list */
288             FD_SET(sock, &master_list);
289
290             if ( sock > maxfd)
291             {
292                 printf("New connection %d\n", sock);
293                 maxfd = sock;
294             }
295             continue;
296         }
297
298         /* Check the set of 'ready' sockets, e.g connected to remote host and
299         waiting for notice of packets received. */
300         for (i = 0; i < (maxfd+1); i++)
301         {
302             if (((i + NX_BSD_SOCKETFD_START) != sock_tcp_server) &&
303                 (FD_ISSET(i + NX_BSD_SOCKETFD_START, &master_list)) &&
304                 (FD_ISSET(i + NX_BSD_SOCKETFD_START, &read_ready)))
305             {
306                 while(1)
307                 {
308                     status = recv(i + NX_BSD_SOCKETFD_START, (VOID *)rcvBuffer1,
309                                   strlen(rcvBuffer1), 0);
310
311                     if (status == 0)
312                     {
313                         printf("\nError: BSD Server socket received no data\n");
314                         break;
315                     }
316                     else if (status != 0xFFFFFFFF)
317                     {
318                         printf("\nServer socket %d received %lu bytes: %s\n",

```

```

328         sock_tcp_server, strlen(rcvBuffer1),rcvBuffer1);
329     }
330     else
331     {
332         printf("\nError: BSD Server socket error \n");
333         break;
334     }
335     printf("Server sock %d sending message back\n",
336           sock_tcp_server);
337
338     status = send(i + NX_BSD_SOCKFD_START, "Hello\n",
339                 strlen("Hello\n")+1, 0);
340
341     if (status == ERROR)
342         printf("Error: BSD Server socket send %d\n",i);
343     else
344     {
345         printf("\nServer message sent: Hello\n");
346     }
347
348     /* close this client socket */
349     status = soc_close(i+ NX_BSD_SOCKFD_START);
350
351     if (status != ERROR)
352         printf("\nBSD Client Socket Closed %d\n", i);
353     else
354         printf("\nError: BSD Client Socket close %d \n", i);
355 }
356
357 /* Loop back to check any next client connection */
358 }
359 }
360
361 VOID thread_client_entry(ULONG thread_input)
362 {
363
364     INT          status, actual_status;
365     INT          sock_tcp_client, length;
366     CHAR         rcvBuffer1[32];
367
368     #ifdef DUO
369     NXD_ADDRESS ip_address;
370     struct sockaddr_in6 echoServAddr6;           /* Echo server address */
371     struct sockaddr_in6 localAddr6;              /* Local address */
372     struct sockaddr_in6 remoteAddr6;             /* Remote address */
373     #else
374     struct sockaddr_in echoServAddr;              /* Echo server address */
375     struct sockaddr_in localAddr;                 /* Local address */
376     struct sockaddr_in remoteAddr;               /* Remote address */
377     #endif
378
379
380     tx_thread_sleep(100);
381
382     status = nx_ip_status_check(&bsd_ip, NX_IP_INITIALIZE_DONE, &actual_status,
383 100);
384
385     /* Check status... */
386     if (status != NX_SUCCESS)
387     {
388         return;
389     }
390
391     #ifdef DUO
392     /* Enable IPV6 */
393     status = nxd_ipv6_enable(&bsd_ip);
394     if((status != NX_SUCCESS) && (status != NX_ALREADY_ENABLED))
395     {
396         printf("Error with IPV6 enable 0x%x\n", status);
397         return;
398     }
399
400     /* Enable ICMPV6 */
401     status = nxd_icmp_enable(&bsd_ip);
402     if(status)
403     {

```

```

405         printf("Error with ICMPv6 enable 0x%x\n", status);
406         return;
407     }
408
409
410
411     status = nxd_ipv6_linklocal_address_set(&bsd_ip, NX_NULL);
412
413     /* Set ip_0 interface address. */
414     ip_address.nxd_ip_version = NX_IP_VERSION_V6;
415     ip_address.nxd_ip_address.v6[0] = 0x20010db8;
416     ip_address.nxd_ip_address.v6[1] = 0x0000f101;
417     ip_address.nxd_ip_address.v6[2] = 0;
418     ip_address.nxd_ip_address.v6[3] = 0x101;
419
420     status = nxd_ipv6_global_address_set(&bsd_ip, &ip_address, 64);
421     if (status)
422         return;
423
424
425     /* Wait for IPv6 stack to finish DAD process. */
426     tx_thread_sleep(400);
427
428 #endif
429
430     /* Create BSD TCP Socket */
431 #ifdef DUO
432     sock_tcp_client = socket( AF_INET6, SOCK_STREAM, IPPROTO_TCP);
433 #else
434     sock_tcp_client = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP);
435 #endif
436
437     if (sock_tcp_client == -1)
438     {
439         printf("\nError: BSD TCP Client socket create \n");
440         return;
441     }
442
443     printf("\nBSD TCP client socket created %lu \n", sock_tcp_client);
444
445     /* Fill Local and Server port and IP address */
446 #ifdef DUO
447     memset(&localAddr6, 0, sizeof(localAddr6));
448     localAddr6.sin6_addr._S6_un._S6_u32[0] = 0x20010db8;
449     localAddr6.sin6_addr._S6_un._S6_u32[1] = 0xf101;
450     localAddr6.sin6_addr._S6_un._S6_u32[2] = 0x0;
451     localAddr6.sin6_addr._S6_un._S6_u32[3] = 0x0101;
452     localAddr6.sin6_port = CLIENT_PORT;
453     localAddr6.sin6_family = AF_INET6;
454
455     memset(&echoServAddr6, 0, sizeof(echoServAddr6));
456     echoServAddr6.sin6_addr._S6_un._S6_u32[0] = 0x20010db8;
457     echoServAddr6.sin6_addr._S6_un._S6_u32[1] = 0xf101;
458     echoServAddr6.sin6_addr._S6_un._S6_u32[2] = 0x0;
459     echoServAddr6.sin6_addr._S6_un._S6_u32[3] = 0x0101;
460     echoServAddr6.sin6_port = SERVER_PORT;
461     echoServAddr6.sin6_family = AF_INET6;
462 #else
463     memset(&localAddr, 0, sizeof(localAddr));
464     localAddr.sin_family = AF_INET;
465     localAddr.sin_addr.s_addr = IP_ADDRESS(1,2,3,4);
466     localAddr.sin_port = CLIENT_PORT;
467
468     memset(&echoServAddr, 0, sizeof(echoServAddr));
469     echoServAddr.sin_family = AF_INET;
470     echoServAddr.sin_addr.s_addr = IP_ADDRESS(1,2,3,4);
471     echoServAddr.sin_port = SERVER_PORT;
472 #endif
473
474     /* Now connect this client to the server */
475 #ifdef DUO
476     status = connect(sock_tcp_client, (struct sockaddr *)&echoServAddr6,
477                     sizeof(echoServAddr6));
478 #else
479     status = connect(sock_tcp_client, (struct sockaddr *)&echoServAddr,
480                     sizeof(echoServAddr));
481 #endif
482
483     /* Check for error. */
484     if (status != OK)

```

```

484     {
485         printf("\nError: BSD TCP Client socket Connect\n");
486         status = soc_close(sock_tcp_client);
487         return;
488     }
489     /* Get and print source and destination information */
490     printf("\nBSD TCP Client socket: %d connected \n", sock_tcp_client);
491
492     #ifdef DUO
493     status = getsockname(sock_tcp_client, (struct sockaddr *)&localAddr6, &length);
494     printf("Client port = %lu, Client = 0x%x 0x%x 0x%x 0x%x\n",
495           localAddr6.sin6_port,
496           localAddr6.sin6_addr._S6_un._S6_u32[0],
497           localAddr6.sin6_addr._S6_un._S6_u32[1],
498           localAddr6.sin6_addr._S6_un._S6_u32[2],
499           localAddr6.sin6_addr._S6_un._S6_u32[3]);
500
501     length = sizeof(struct sockaddr_in6);
502     status = getpeername(sock_tcp_client, (struct sockaddr *)&remoteAddr6,
503           &length);
504     printf("Remote port = %lu, Remote IP = 0x%x 0x%x 0x%x 0x%x \n",
505           remoteAddr6.sin6_port,
506           remoteAddr6.sin6_addr._S6_un._S6_u32[0],
507           remoteAddr6.sin6_addr._S6_un._S6_u32[1],
508           remoteAddr6.sin6_addr._S6_un._S6_u32[2],
509           remoteAddr6.sin6_addr._S6_un._S6_u32[3]);
510     #else
511     status = getsockname(sock_tcp_client, (struct sockaddr *)&localAddr, &length);
512     printf("Client port = %lu, Client = 0x%x\n", localAddr.sin_port,
513           localAddr.sin_addr.s_addr);
514     length = sizeof(struct sockaddr_in);
515     status = getpeername(sock_tcp_client, (struct sockaddr *)&remoteAddr,
516           &length);
517     printf("Remote port = %lu, Remote IP = 0x%x \n", remoteAddr.sin_port,
518           remoteAddr.sin_addr.s_addr);
519     #endif
520     /* Now receive the echoed packet from the server */
521     while(1)
522     {
523         printf("\nClient sock: %d Sending packet to server\n", sock_tcp_client);
524         status = send(sock_tcp_client, "Hello", (strlen("Hello")+1), 0);
525         if (status == ERROR)
526             printf("Error: BSD Client Socket send %d\n", sock_tcp_client);
527         else
528         {
529             printf("\nClient sent message Hello\n");
530         }
531         status = recv(sock_tcp_client, (VOID *)rcvBuffer1, 32, 0);
532         if (status <= 0)
533         {
534             if (status < 0)
535             {
536                 printf("\nError: BSD Client Socket receive %d \n", sock_tcp_client);
537             }
538             else
539             {
540                 printf("Nothing received by Client\n");
541             }
542             break;
543         }
544     }
545     /* close this client socket */
546     status = soc_close(sock_tcp_client);
547     if (status != ERROR)
548         printf("\nBSD Client Socket Closed %d\n", sock_tcp_client);
549     else
550         printf("\nError: BSD Client Socket close %d \n", sock_tcp_client);
551 }

```

Chapter 3

List of 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  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);

```


Appendix A

NetX Duo BSD Extended Services

Description of BSD extended services

The BSD extended services adds new services to NetX Duo BSD sockets to bring the BSD wrapper into closer compliance with actual BSD 4.3 sockets. These include asynchronous notification of TCP connection and disconnection completion, and various socket options such as socket error handling, non blocking sockets and keep alive TCP sockets.

To use the BSD extended services, the NetX Duo library must be enabled with the NetX Duo have `NX_DISABLE_EXTENDED_NOTIFY_SUPPORT` disabled which it is by default. In addition, the host BSD application must define `NX_EXTENDED_BSD_SOCKET_SUPPORT` either at the project level or in `nxd_bsd.h` and in the host application code where BSD API are invoked.

Below lists the following steps to set up a host application for BSD extended services.

1. In `tx_user.h`, the `TX_THREAD_USER_EXTENSION` must be defined to use socket error codes as follows:


```
#define TX_THREAD_USER_EXTENSION      int bsd_errno
```
2. In `tx_port.h`, define `TX_INCLUDE_USER_DEFINE_FILE` to enable the changes made to `tx_user.h` above.
3. Rebuild the ThreadX library.
3. Build NetX Duo with `NX_DISABLE_EXTENDED_NOTIFY_SUPPORT` disabled
4. The host BSD application must define `NX_EXTENDED_BSD_SOCKET_SUPPORT` at the project level or in both `nxd_bsd.h` and in the host application code.

To utilize the new callback notification functions, the host application (and BSD demo files) must define the disconnect and connect (establish) notify callbacks. This can be done in the `tx_application_define()` function. See the **Small Example for Extended BSD Services** below for how to do this.

Small Example for Extended BSD Services

An example of how to use NetX Duo BSD advanced features is described in Figure 1.1. The include file *nxd_bsd.h* is brought in at line 9. On line 14, the application sets the BSD option `NX_EXTENDED_BSD_SOCKET_SUPPORT` to enable BSD extended features. This same option must also be defined at the top of *nxd_bsd.h*. Alternatively it can be defined at the project level.

Note that this demo uses a ram (virtual) network driver and is limited to IPv4.

The client and server share the same IP address on single IP instance in this example. After creating the NetX Duo and ThreadX data blocks for thread, packet pool and IP instance, there is a series of conditional defines in lines 49 - 74 to enable one or more of the socket options available with BSD extended services. This particular example defines the establish and disconnect callback notification services in lines 60 and 63. The actual user defined callbacks are defined in lines 541-579 at the bottom of the file. On lines 168 and 182, the host application uses the new BSD services *nx_bsd_set_socket_establish_notify* and *nx_bsd_set_socket_disconnect_complete_notify* to notify BSD what functions to call on connection complete ("established") and disconnect complete.

The client and server threads are created on lines 104 and 110. After IP instance successful creation on line 129, the IP instance is enabled for TCP services on line 152. The last requirement before BSD services can be used is to call *bsd_initialize* on line 195 to set up all data structures and NetX, and ThreadX resources needed by BSD.

In this example, the Client task is fairly generic and designed to simply make connections, exchange packets and close the connection. The Server task is where the advanced features are applied.

The Client thread waits briefly for the Server side to be set up before creating an IPv4 TCP socket and attempting to connect to the server in lines 215 and 232 respectively. It then sends and receives packets on lines 255-278 with the server until it detects the connection is closed.

The Server task is defined in *thread_1_entry* starting on line 229. It verifies the host IP address is established in lines 242 - 249. It then creates a TCP socket on line 267. Note this binds a TCP socket to listen for connection requests on line 272. All its socket descriptor sets are cleared before .

The server thread creates a TCP IPv4 master socket on line 322, and sets various socket options on the master socket in lines 336 - 353. Because by default socket inheritance is enabled in NetX Duo BSD (see description of **`NX_BSD_INHERIT_LISTENER_SOCKET_SETTINGS`** in "Configuration options" below), all secondary listening and connecting sockets will inherit these options.

Non blocking is set using the `fcntl` service on line 350. The socket is then bound to the server listening port on line 362, the socket descriptor sets are cleared, and the master socket is promoted to the TCP listen state in line 376.

Because the socket is non blocking note that `select` must define the *timeout* argument (lines 403-405). To check for errors on the select call, BSD is queried for socket error status on lines 425 - 436 using the `getsockopt` call with the `SO_ERROR` option.. Socket error status is automatically available if BSD extended services are enabled

After the `select` call, if an establish connection callback had not been set, the server task would need to query read ready FD and if a connection request is detected it would have to call `accept` to complete the connection in lines 462-482 as is typically done in BSD standard applications. However, with the establish callback set, BSD internal operations handles these details. Note that the establish callback does need to update the `maxfd` variable for the BSD server socket to know what sockets should be polled for packets received in line 564.

Now the execution can skip to the for loop which checks for sockets notified of data ready to receive in starting on line 490.

```

1  /* This is a small demo of BSD Wrapper for the high-performance NetX Duo TCP/IP
2     stack. This demo used advanced BSD services for TCP connection, disconnection,
3     sending, and receiving using a simulated Ethernet driver. */
4
5
6
7  #include      "tx_api.h"
8  #include      "nx_api.h"
9  #include      "nxd_bsd.h"
10 #include      <string.h>
11 #include      <stdlib.h>
12
13 /* Enable the extended BSD features (asynchronous connect,disconnect notification,
14    socket error setting etc. */
14 #define NX_EXTENDED_BSD_SOCKET_SUPPORT
15
16
17 #define        DEMO_STACK_SIZE      (16*1024)
18 #define        SERVER_PORT          87
19 #define        CLIENT_PORT          77
20
21 /* Define the ThreadX and NetX object control blocks... */
22
23 TX_THREAD      thread_server;
24 TX_THREAD      thread_client;
25 NX_PACKET_POOL bsd_pool;
26 NX_IP          bsd_ip;
27
28 /* Define some global data. */
29 CHAR    *msg0 = "ABCDEFGHJKLMNOPQRSTUVWXYZ<>END";
30 INT      maxfd;
31
32 /* Define the counters used in the demo application... */
33

```

```

34  ULONG          error_counter;
35
36  /* Define fd_sets for the BSD server socket. */
37  fd_set          master_list, read_ready;
38
39
40  /* Set a flag to indicate if a callback for connection complete is set. */
41  UINT  establish_cb_set = NX_FALSE;
42
43  /* Define thread prototypes. */
44
45  VOID          thread_server_entry(ULONG thread_input);
46  VOID          thread_client_entry(ULONG thread_input);
47  void          _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);
48
49  #ifdef NX_EXTENDED_BSD_SOCKET_SUPPORT
50
51  /* Set the following socket options available with BSD extended support. */
52
53
54  /* Set sockets to non blocking for connecting, disconnecting and rx/tx'ing
    packets */
55  #define ENABLE_NONBLOCKING
56
57  /* Set the establish callback which NetX Duo will invoke when a connection is
    complete ("established"). This eliminates the requirement to use the
    accept() service on a TCP server socket because BSD internal operations handles
    the details of completing the connection request. */
58  #define ENABLE_ESTABLISH_CB
59
60  /* Set the disconnect callback which NetX Duo will invoke when a disconnection is
    complete. */
61  #define ENABLE_DISCONNECT_CB
62
63  /* Set the socket option to keep a TCP connection alive. The NetX Duo library must
    be compiled with the NX_TCP_KEEP_ALIVE option defined. Check the NetX Duo
    User manual if it supports the NX_TCP_KEEP_ALIVE option.
    #define ENABLE_KEEPALIVE
    */
64
65  /* Set the socket to withhold a Server TCP connection from the available BSD
    socket pool for the specified 'linger time' to capture
    any packets intended for this connection. Non blocking must be disabled for
    this option to have any effect.
    #define ENABLE_LINGER
    */
66
67  /* Declare the callbacks for BSD extended services. Note that the
    ENABLE_DISCONNECT_CB and ENABLE_ESTABLISH_CB
    must also be defined for these to have any effect. */
68  VOID          bsd_tcp_establish_notify(NX_TCP_SOCKET *socket_ptr);
69  VOID          bsd_tcp_disconnect_complete_notify(NX_TCP_SOCKET *socket_ptr);
70
71  #endif /* NX_EXTENDED_BSD_SOCKET_SUPPORT */
72
73  /* Define main entry point. */
74
75  int main()
76  {
77
78      /* Enter the ThreadX kernel. */

```

```

89     tx_kernel_enter();
90 }
91
92 /* Define what the initial system looks like. */
93
94 void    tx_application_define(void *first_unused_memory)
95 {
96     CHAR    *pointer;
97     UINT    status;
98
99
100     /* Setup the working pointer. */
101     pointer = (CHAR *) first_unused_memory;
102
103     /* Create a server thread. */
104     tx_thread_create(&thread_server, "Server", thread_server_entry, 0,
105                     pointer, DEMO_STACK_SIZE, 2, 2, TX_NO_TIME_SLICE,
106                     TX_AUTO_START);
107
108     pointer = pointer + DEMO_STACK_SIZE;
109
110     /* Create a client thread. */
111     tx_thread_create(&thread_client, "Client", thread_client_entry, 0,
112                     pointer, DEMO_STACK_SIZE, 4, 4, TX_NO_TIME_SLICE,
113                     TX_AUTO_START);
114
115     pointer = pointer + DEMO_STACK_SIZE;
116
117     /* Initialize the NetX system. */
118     nx_system_initialize();
119
120     /* Create a BSD packet pool. */
121     status = nx_packet_pool_create(&bsd_pool, "NetX BSD Packet Pool", 128,
122                                   pointer, 16384);
123
124     pointer = pointer + 16384;
125     if (status)
126     {
127         error_counter++;
128     }
129
130     /* Create an IP instance for BSD. */
131     status = nx_ip_create(&bsd_ip, "NetX IP Instance 2", IP_ADDRESS(1,2,3,4),
132                           0xFFFFFFFFUL, &bsd_pool, _nx_ram_network_driver,
133                           pointer, 2048, 1);
134
135     pointer = pointer + 2048;
136
137     if (status)
138     {
139         error_counter++;
140     }
141
142     /* Enable ARP and supply ARP cache memory for BSD IP Instance */
143     status = nx_arp_enable(&bsd_ip, (void *) pointer, 1024);
144     pointer = pointer + 1024;
145
146     /* Check ARP enable status. */
147     if (status)
148     {
149         error_counter++;
150     }

```

```

149
150     /* Enable TCP processing for BSD IP instances.  */
151
152     status = nx_tcp_enable(&bsd_ip);
153
154     /* Check TCP enable status.  */
155     if (status)
156     {
157         error_counter++;
158     }
159
160
161     /* Enable BSD socket callbacks if BSD extended support is set.  */
162
163     #ifdef NX_EXTENDED_BSD_SOCKET_SUPPORT
164
165     #ifdef ENABLE_ESTABLISH_CB
166
167         /* Note that this callback eliminates the need to call accept().  */
168         status = nx_bsd_set_socket_establish_notify(bsd_tcp_establish_notify);
169
170         /* Check completion status.  */
171         if (status)
172         {
173             return;
174         }
175
176         establish_cb_set = NX_TRUE;
177     #endif
178 #endif
179
180     #ifdef ENABLE_DISCONNECT_CB
181
182     status = nx_bsd_set_socket_disconnect_complete_notify
183             (bsd_tcp_disconnect_complete_notify);
184
185     /* Check completion status.  */
186     if (status)
187     {
188         printf("disconnect complete notify called...\n");
189         return;
190     }
191 #endif
192 #endif /* NX_EXTENDED_BSD_SOCKET_SUPPORT */
193
194     /* Now initialize BSD Socket Wrapper */
195     status = bsd_initialize (&bsd_ip, &bsd_pool, pointer);
196 }
197
198
199 VOID  thread_client_entry(ULONG thread_input)
200 {
201
202     INT      status;
203     INT      bytes;
204     INT      sock_tcp_client, length;
205     struct    sockaddr_in echoServAddr;           /* Echo server address */
206     struct    sockaddr_in localAddr;              /* Local address */
207     struct    sockaddr_in remoteAddr;             /* Remote address */
208     CHAR      ClientBuffer[132];
209
210

```

```

211     /* Give the server thread time to set up. */
212     tx_thread_sleep(100);
213
214     /* Create Client TCP Socket */
215     sock_tcp_client = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP);
216
217     if (sock_tcp_client == NX_SOC_ERROR)
218     {
219         printf("Client socket %d error on create\n", sock_tcp_client);
220         return;
221     }
222
223     printf("\nBSD TCP Client socket created %lu \n", sock_tcp_client);
224
225     /* Fill destination port and IP address */
226     memset(&echoServAddr, 0, sizeof(echoServAddr));
227     echoServAddr.sin_family = PF_INET;
228     echoServAddr.sin_addr.s_addr = IP_ADDRESS(1,2,3,4);
229     echoServAddr.sin_port = SERVER_PORT;
230
231     /* Now connect this client to the server */
232     status = connect(sock_tcp_client, (struct sockaddr *)&echoServAddr,
                     sizeof(echoServAddr));
233
234     /* Check for error. */
235     if (status != OK)
236     {
237
238         printf("Client socket %d error on connect\n", sock_tcp_client);
239
240         status = soc_close(sock_tcp_client);
241
242         return;
243     }
244
245     /* Get and print source and destination information */
246     printf("Client socket %d connected!\n", sock_tcp_client);
247
248     status = getsockname(sock_tcp_client, (struct sockaddr *)&localAddr, &length);
249     printf("Client port = %lu , Client address = 0x%x\n", localAddr.sin_port,
           localAddr.sin_addr.s_addr);
250     status = getpeername(sock_tcp_client, (struct sockaddr *)&remoteAddr, &length);
251     printf("Remote port = %lu, Remote IP address= 0x%x \n", remoteAddr.sin_port,
           remoteAddr.sin_addr.s_addr);
252
253
254     /* Now receive the echoed packet from the server */
255     while(1)
256     {
257         tx_thread_sleep(2);
258
259         printf("Client sock %d sending packet to server\n", sock_tcp_client);
260         status = send(sock_tcp_client, msg0, (strlen(msg0)+1), 0);
261
262         if (status == ERROR)
263             printf("Client socket %d error on send\n", sock_tcp_client);
264         else
265         {
266             printf("Client %d sent message: %s\n", sock_tcp_client, msg0);
267         }
268     }

```

```

269         bytes = recv(sock_tcp_client, (VOID *)ClientBuffer, 132 ,0);
270         if (bytes == 0)
271             break;
272
273         if (bytes != NX_SOC_ERROR)
274             printf("Client socket %d received %lu bytes: %s\n", sock_tcp_client +
                    NX_BSD_SOCKETFD_START, bytes, ClientBuffer);
275         else
276             printf("Client socket %d error on receive\n", sock_tcp_client, bytes);
277     }
278 }
279
280 /* close this client socket */
281 status = soc_close(sock_tcp_client);
282
283 if (status != ERROR)
284     printf("Client socket closed %d\n", sock_tcp_client);
285 else
286     printf("Client socket error on close\n");
287
288 /* End */
289 }
290
291 /* Define the Server thread. */
292
293 VOID thread_server_entry(ULONG thread_input)
294 {
295
296
297     INT         status, sock, sock_tcp_server;
298     INT         i;
299     UINT        is_set = NX_FALSE;
300     struct      sockaddr_in echoServAddr;
301     struct      sockaddr_in ClientAddr;
302     INT         ClientLength;
303     CHAR        ServerBuffer[132];
304     #ifdef NX_EXTENDED_BSD_SOCKET_SUPPORT
305     struct      timeval select_timeout;
306     UINT        nonblocking_enabled = NX_FALSE;
307     INT         result;
308     INT         option_length;
309     #ifdef ENABLE_KEEPALIVE
310     struct      sock_keepalive keepalive;
311     #endif
312     #if ENABLE_LINGER
313     struct      sock_linger linger;
314     #endif
315     #endif
316
317
318     /* Let NetX and the driver get initialized. */
319     tx_thread_sleep(100);
320
321     /* Create the Server TCP Socket */
322     sock_tcp_server = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP);
323
324     if (sock_tcp_server == -1)
325     {
326         printf("Server socket error on creating secondary socket.\n");
327         return;
328     }

```



```

329
330     printf("Server socket created secondary socket %lu \n", sock_tcp_server);
331
332 #ifdef NX_EXTENDED_BSD_SOCKET_SUPPORT
333
334     /* Enable various socket options if BSD extended socket support is set. */
335
336 #ifdef ENABLE_KEEPALIVE
337     /* To enable keepalive, the NetX Duo library must be compiled with
338        NX_TCP_ENABLE_KEEPALIVE is defined. */
338     keepalive.keepalive_enabled = NX_TRUE;
339     status = setsockopt(sock_tcp_server, SOL_SOCKET, SO_KEEPALIVE, &keepalive,
340                        sizeof(keepalive));
340 #endif
341
342 #if ENABLE_LINGER
343     linger.linger_onoff = NX_TRUE;
344     linger.linger_time = 15;
345     status = setsockopt(sock_tcp_server, SOL_SOCKET, SO_LINGER, &linger,
346                        sizeof(linger));
346 #endif
347
348 #ifdef ENABLE_NONBLOCKING
349     /* If not lingering, set to non blocking */
350     fnctl(sock_tcp_server, F_SETFL, SO_NONBLOCK);
351     nonblocking_enabled = NX_TRUE;
352 #endif
353 #endif
354
355     /* Set the server port and IP address */
356     memset(&echoServAddr, 0, sizeof(echoServAddr));
357     echoServAddr.sin_family = AF_INET;
358     echoServAddr.sin_addr.s_addr = IP_ADDRESS(1,2,3,4);
359     echoServAddr.sin_port = SERVER_PORT;
360
361     /* Bind this server socket */
362     status = bind (sock_tcp_server, (struct sockaddr *) &echoServAddr,
363                  sizeof(echoServAddr));
364
365     if (status < 0)
366     {
367         return;
368     }
369
370     FD_ZERO(&master_list);
371     FD_ZERO(&read_ready);
372     FD_SET(sock_tcp_server, &master_list);
373     maxfd = sock_tcp_server;
374
375     /* Now listen for any client connections for this server socket */
376     status = listen (sock_tcp_server, 5);
377     if (status < 0)
378     {
379
380         return;
381     }
382     else
383         printf("Server socket is listening...\n");
384
385     /* All set to accept client connections */

```

```

386     printf("Now accepting client connections\n");
387
388     /* Loop to create and establish server connections. */
389     while(1)
390     {
391
392         read_ready = master_list;
393
394         tx_thread_sleep(20); /* Allow some time to other threads too */
395
396 #ifdef NX_EXTENDED_BSD_SOCKET_SUPPORT
397
398         /* Check if nonblock flag set on this master socket. */
399         if (nonblocking_enabled)
400         {
401
402             /* For a non blocking call, select() cannot send in a null timeout! */
403             select_timeout.tv_sec = 0;
404             select_timeout.tv_usec = 0;
405             status = select(maxfd + 1, &read_ready, 0, 0, &select_timeout);
406         }
407         else
408         {
409
410             /* Let the underlying TCP stack determine the timeout. */
411             status = select(maxfd + 1, &read_ready, 0, 0, 0);
412         }
413
414         if ((status == 0xFFFFFFFF) || (status == 0))
415         {
416
417             if (status == 0xFFFFFFFF)
418             {
419
420                 option_length = sizeof(INT);
421
422                 /* Demonstrate socket error handling. Check if socket error is a
423                  result of timing out (in progress, for example)
424                  or an actual socket connect error. */
425                 status = getsockopt(sock_tcp_server, SOL_SOCKET, SO_ERROR,
426                                     (INT *)&result, &option_length);
427
428                 /* Check if this is a nonblocking socket error. */
429                 if (result == EWOULDBLOCK)
430                 {
431                     /* This is a non blocking error; we simply just don't have a
432                      receive/connect event yet. */
433                     printf("Server socket status on select: In progress...\n");
434                     tx_thread_sleep(100);
435                 }
436                 else
437                 {
438                     /* The connection failed. */
439                     printf("Server socket error status on select: 0x%x...\n",
440                             result);
441                 }
442             }
443             continue;
444         }
445 #endif
446
447         /* Let the underlying TCP stack determine the timeout. */

```

```

444     status = select(maxfd + 1, &read_ready, 0, 0, 0);
445
446     if (status <= 0)
447     {
448         if (status < 0)
449         {
450
451             printf("Server error on select. Try again\n");
452         }
453
454         continue;
455     }
456
457     printf("Detect a connection request\n");
458 #endif
459
460     /* If the BSD server socket does not have an connection callback function,
461        handle the connection request here. */
462     if (establish_cb_set == NX_FALSE)
463     {
464
465         is_set = FD_ISSET(sock_tcp_server, &read_ready);
466
467         if(is_set)
468         {
469
470             ClientLength = sizeof(ClientAddr);
471
472             sock = accept(sock_tcp_server, (struct sockaddr*)&ClientAddr,
473                           &ClientLength);
474
475             /* Add this new connection to our master list */
476             FD_SET(sock, &master_list);
477
478             if ( sock > maxfd)
479             {
480                 printf("Server has a new connection on socket %d\n", sock);
481
482                 maxfd = sock;
483             }
484
485             continue;
486         }
487
488     /* Check the set of 'ready' sockets, e.g connected to remote host and
489        waiting for notice of packets received. */
490     for (i = 0; i < (maxfd+1); i++)
491     {
492
493         if (((i+ NX_BSD_SOCKFD_START) != sock_tcp_server) &&
494             (FD_ISSET(i + NX_BSD_SOCKFD_START, &master_list)) &&
495             (FD_ISSET(i + NX_BSD_SOCKFD_START, &read_ready)))
496         {
497
498             printf("Server received data on socket %d\n", i +
499                   NX_BSD_SOCKFD_START);
500             while(1)
501             {
502                 tx_thread_sleep(2);

```

```

503             status = recv(i + NX_BSD_SOCKETFD_START, (VOID *)ServerBuffer,
504                             132,0);
505
506             if (status == 0)
507                 break;
508             if (status != 0xFFFFFFFF)
509             {
510                 printf("\nServer socket %d received %lu bytes: %s ",
511                     sock_tcp_server, strlen(ServerBuffer),ServerBuffer);
512             }
513             else
514             {
515                 printf("Server socket %d received data\n",
516                     sock_tcp_server);
517                 break;
518             }
519
520             status = send(i + NX_BSD_SOCKETFD_START, "Hello\n",
521                             strlen("Hello\n")+1, 0);
522
523             if (status == ERROR)
524                 printf("Server socket %d error on send\n", i +
525                     NX_BSD_SOCKETFD_START);
526             else
527             {
528                 printf("Server socket %d sent message Hello\n", i +
529                     NX_BSD_SOCKETFD_START);
530             }
531         }
532
533         /* close this client socket */
534         status = soc_close(i+ NX_BSD_SOCKETFD_START);
535
536         if (status != ERROR)
537             printf("Server socket %d closing \n", i+ NX_BSD_SOCKETFD_START);
538         else
539             printf("Server socket %d error on close\n", i+
540                 NX_BSD_SOCKETFD_START);
541     }
542 }
543
544 /* Loop back to check any next server connection */
545 }
546
547 #ifdef NX_EXTENDED_BSD_SOCKET_SUPPORT
548
549 /* Define TCP callback function. Note that both Server and Clients socket
550 connections will activate these callbacks, so the socket ID is the key to
551 determining which socket is notified of connection completed or disconnect
552 completed. */
553 /* Process the connection for the host application e.g update the socket list of
554 ready sockets with the new connection. */
555 VOID bsd_tcp_establish_notify(NX_TCP_SOCKET *socket_ptr)
556 {
557     UINT bsd_socket_index;
558
559     /* Figure out what BSD socket this is. */

```

```

557     bsd_socket_index = (UINT) socket_ptr -> nx_tcp_socket_reserved_ptr;
558     printf("Host has a connection on socket %d!\n", bsd_socket_index +
           NX_BSD_SOCKETFD_START);
559
560     if (bsd_socket_index + NX_BSD_SOCKETFD_START > maxfd)
561     {
562         /* This is now the highest socket index to check. */
563         maxfd = bsd_socket_index;
564     }
565
566     FD_SET(bsd_socket_index + NX_BSD_SOCKETFD_START, &master_list);
567     return;
568 }
569
570
571 /* Process the disconnection for the host application e.g update the socket list
572    of ready sockets with the current connection removed. */
573 VOID bsd_tcp_disconnect_complete_notify(NX_TCP_SOCKET *socket_ptr)
574 {
575
576     UINT bsd_socket_index;
577
578
579     bsd_socket_index = (UINT) socket_ptr -> nx_tcp_socket_reserved_ptr;
580
581     printf("Host disconnect completed for %d!\n", bsd_socket_index +
           NX_BSD_SOCKETFD_START);
582
583     FD_CLR(bsd_socket_index + NX_BSD_SOCKETFD_START, &master_list);
584
585     return;
586 }
587
588 #endif /* NX_EXTENDED_BSD_SOCKET_SUPPORT*/

```

List NetX Duo BSD extended services

```

INT  fnctl(INT sock_ID, UINT flag_type, UINT f_options);
      Enables or disables the specified socket ID with non blocking*.

INT  getsockopt(INT sockID, INT option_level, INT option_name, void
      *option_value, INT *option_length);
      Reports the status of the specified socket option

INT  ioctl(INT sockID, INT command, INT *result);
      Sets the socket with the specified command. Supports FIONREAD and FIONBIO only.
      FIONBIO is equivalent to fnctl used with the SO_NONBLOCK option

INT  inet_aton(const CHAR *numstring, struct in_addr *addr)
      Converts an IP address string to a number

in_addr_t inet_addr(const CHAR *stringptr)
      Converts an IP address string to a number

CHAR *inet_ntoa(struct in_addr address_to_convert)
      Converts an IP address to a string

*UINT nx_bsd_set_socket_disconnect_complete_notify(VOID
      (*nx_bsd_tcp_disconnect_complete_notify)(INT sockID))

```

Notifies the host application that a disconnection is completed for both TCP server and client sockets

```
UINT nx_bsd_set_socket_establish_notify(VOID
    (*bsd_tcp_establish_notify)(NX_TCP_SOCKET *socket_ptr));
```

Notifies the host application when a TCP connection has succeeded for both server and client TCP sockets

```
UINT nx_bsd_timed_wait_callback(NX_TCP_SOCKET *tcp_socket_ptr)
```

Notifies the host application that the TCP socket is shutdown and in the timed wait state. If REUSE_ADDR is not enabled on the TCP socket, the socket enters the timed wait state for the interval defined by the NX_BSD_TIMED_WAIT_TIMEOUT option

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

Enables or disables the specified socket option on the socket ID

**The NetX option NX_DISABLE_RESET_DISCONNECT enables a non blocking socket to disconnect from a remote host gracefully (e.g. not sending a RST packet). The BSD socket remains open for any outstanding packets and to await the FIN ACK handshake while the BSD socket application does not have to wait for this event to complete*

*** These following socket options are supported in NetX Duo BSD with extended services enabled, either implicitly by NetX Duo or by setting the specific option using setsockopt:*

SO_BROADCAST

Implicitly supported by NetX Duo

SO_KEEPAIVE

Requires NetX Duo library to be enabled with the NX_TCP_KEEP_ALIVE which is not enabled by default and not available in all NetX Duo releases.

SO_LINGER

SO_NONBLOCK

SO_RCFBUF

SO_RCVTIMEO

SO_SNDTIMEO

SO_REUSEADDR

Implicitly supported by NetX Duo (same as Timed Wait if disabled)

TCP_NODELAY

Implicitly supported by NetX Duo

FIONREAD

FIONBIO (same as SO_NONBLOCK)

Configurable options in NetX Duo Extended Services

Define	Meaning
NX_BSD_INHERIT_LISTENER_SOCKET_SETTINGS	If defined, secondary sockets inherit master socket options and socket flags. By default this option is enabled. This includes the keep alive feature discussed above.
NX_EXTENDED_BSD_LINGER_AND_TIMED_WAIT	If not defined, Linger and Timed Wait are disabled even with the BSD extended features enabled. By default this option is disabled.
NX_BSD_LINGER_TIMER_RATE	This defines the interval when to check socket status for received packets in Timed Wait state. The default value is (1 * NX_CPU_TICKS_PER_SECOND)
NX_BSD_TIMED_WAIT_TIMEOUT	This defines the timeout for sockets in the Timed Wait state. Such sockets must not be enabled with the REUSEADDR to enter this state. The default value is (60 * NX_CPU_TICKS_PER_SECOND)
NX_BSD_TW_TIMER_RATE	This defines the rate at which Timed Wait timer checks sockets in Timed Wait state. The default value is (5 * NX_CPU_TICKS_PER_SECOND)
NX_EXTENDED_BSD_ENABLE_ASYNC_ACCEPT	If enabled, accept is not executed. This is generally the situation if the host application has set an establish callback

function in BSD, which requires `NX_EXTENDED_BSD_SOCKET_SUPPORT` to be enabled. Internally BSD handles the details of completing the connection including setting up a new listening socket to replace the socket connected to the requesting Client. Hence *accept* services no purpose.