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

Trademarks

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

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:

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

Sample demo programs:

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

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 socke t 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*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 nclude 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"
```

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


#define          DEMO_STACK_SIZE     (16*1024)



#define          SERVER_PORT          87
#define          CLIENT_PORT          77

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

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

/* Define some global data. */
CHAR    *msg0 = "Client 1: ABCDEFGHIJKLMNOPQRSTUVWXYZ ";
INT      maxfd;

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

ULONG            error_counter;

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

/* To send IPv6 packets, define DUO.  */
#define  DUO



/* Define thread prototypes.  */

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


/* Define main entry point.  */

int main()
{

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

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

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


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

    /* Create a server thread.  */
    tx_thread_create(&thread_server, "Server", thread_server_entry, 0,
                          pointer, DEMO_STACK_SIZE, 1, 1, TX_NO_TIME_SLICE,
                         TX_AUTO_START);

    pointer =  pointer + DEMO_STACK_SIZE;

    /* Create a Client thread.  */
    tx_thread_create(&thread_client, "Client", thread_client_entry, 0,
                          pointer, DEMO_STACK_SIZE, 3, 3, TX_NO_TIME_SLICE,
                         TX_AUTO_START);

    pointer =  pointer + DEMO_STACK_SIZE;


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

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

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

    if (status)
    {
          error_counter++;
    }

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

    /* Check ARP enable status.  */
    if (status)
    {
        error_counter++;
    }

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

    status = nx_tcp_enable(&bsd_ip);

    /* Check TCP enable status.  */
    if (status)
    {
        error_counter++;
    }

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


/* Define the Server thread.  */

VOID  thread_server_entry(ULONG thread_input)
{


INT         status,  actual_status, sock, sock_tcp_server;
CHAR        rcvBuffer1[1000];
INT         Clientlen;
INT         i;
UINT        is_set = NX_FALSE;
#ifdef DUO
NXD_ADDRESS ip_address;
struct      sockaddr_in6 serverAddr;
struct      sockaddr_in6 ClientAddr;
#else
struct      sockaddr_in serverAddr;
struct      sockaddr_in ClientAddr;
#endif

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

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

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

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


    status = nxd_ipv6_linklocal_address_set(&bsd_ip, NX_NULL);

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

    status = nxd_ipv6_global_address_set(&bsd_ip, &ip_address, 64);
    if (status)
        return;

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

#endif

    /* Create BSD TCP Socket */
#ifdef DUO
    sock_tcp_server = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
#else
    sock_tcp_server = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
#endif

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

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


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

#else
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = IP_ADDRESS(1,2,3,4);
    serverAddr.sin_port = SERVER_PORT;
#endif

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

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

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

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

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

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

        read_ready = master_list;

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


        /* Let the underlying TCP stack determine the timeout. */
        status = select(maxfd + 1, &read_ready, 0, 0, 0);
        if ((status == 0xFFFFFFFF) || (status == 0))
        {
            printf("Error with select? Status 0x%x. Try again\n", status);

            continue;
        }

        /* Detected a connection request. */


        is_set = FD_ISSET(sock_tcp_server,&read_ready);

        if(is_set)
        {


            Clientlen = sizeof(ClientAddr);

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

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

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

                maxfd = sock;
            }

            continue;
        }

        /* Check the set of 'ready' sockets, e.g connected to remote host and
           waiting for notice of packets received. */
        for (i = 0; i < (maxfd+1); i++)
        {

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

                while(1)
                {


                    status = recv(i + NX_BSD_SOCKFD_START, (VOID *)rcvBuffer1,
                                 strlen(rcvBuffer1),0);

                    if (status == 0)
                    {
                        printf("\nError: BSD Server socket received no data\n");
                        break;
                    }
                    else if (status != 0xFFFFFFFF)
                    {
                        printf("\nServer socket %d received %lu bytes: %s\n",
                                sock_tcp_server, strlen(rcvBuffer1),rcvBuffer1);
                    }
                    else
                    {
                        printf("\nError: BSD Server socket error \n");
                        break;
                    }

                    printf("Server sock %d  sending message back\n",
                                sock_tcp_server);

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

                    if (status == ERROR)
                        printf("Error: BSD Server socket send %d\n",i);
                    else
                    {
                        printf("\nServer message sent: Hello\n");
                    }
                }

                /* close this client socket */
                status = soc_close(i+ NX_BSD_SOCKFD_START);

                if (status != ERROR)
                    printf("\nBSD Client Socket Closed %d\n", i);
                else
                   printf("\nError: BSD Client Socket close %d \n", i);
            }
        }

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

VOID  thread_client_entry(ULONG thread_input)
{


INT         status, actual_status;
INT         sock_tcp_client, length;
CHAR        rcvBuffer1[32];

#ifdef DUO
NXD_ADDRESS ip_address;
struct      sockaddr_in6 echoServAddr6;               /* Echo server address */
struct      sockaddr_in6 localAddr6;                  /* Local address */
struct      sockaddr_in6 remoteAddr6;                 /* Remote address */
#else
struct      sockaddr_in echoServAddr;               /* Echo server address */
struct      sockaddr_in localAddr;                  /* Local address */
struct      sockaddr_in remoteAddr;                 /* Remote address */
#endif


    tx_thread_sleep(100);

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

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

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


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



    status = nxd_ipv6_linklocal_address_set(&bsd_ip, NX_NULL);

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

    status = nxd_ipv6_global_address_set(&bsd_ip, &ip_address, 64);
    if (status)
        return;


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

#endif

    /* Create BSD TCP Socket */
#ifdef DUO
    sock_tcp_client = socket( AF_INET6, SOCK_STREAM, IPPROTO_TCP);
#else
    sock_tcp_client = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP);
#endif

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

    printf("\nBSD TCP Client socket created %lu \n", sock_tcp_client);

    /* Fill Local and Server port and IP address */
#ifdef DUO
    memset(&localAddr6, 0, sizeof(localAddr6));
    localAddr6.sin6_addr._S6_un._S6_u32[0] = 0x20010db8;
    localAddr6.sin6_addr._S6_un._S6_u32[1] = 0xf101;
    localAddr6.sin6_addr._S6_un._S6_u32[2] = 0x0;
    localAddr6.sin6_addr._S6_un._S6_u32[3] = 0x0101;
    localAddr6.sin6_port = CLIENT_PORT;
    localAddr6.sin6_family = AF_INET6;

    memset(&echoServAddr6, 0, sizeof(echoServAddr6));
    echoServAddr6.sin6_addr._S6_un._S6_u32[0] = 0x20010db8;
    echoServAddr6.sin6_addr._S6_un._S6_u32[1] = 0xf101;
    echoServAddr6.sin6_addr._S6_un._S6_u32[2] = 0x0;
    echoServAddr6.sin6_addr._S6_un._S6_u32[3] = 0x0101;
    echoServAddr6.sin6_port = SERVER_PORT;
    echoServAddr6.sin6_family = AF_INET6;

#else
    memset(&localAddr, 0, sizeof(localAddr));
    localAddr.sin_family = AF_INET;
    localAddr.sin_addr.s_addr = IP_ADDRESS(1,2,3,4);
    localAddr.sin_port = CLIENT_PORT;

    memset(&echoServAddr, 0, sizeof(echoServAddr));
    echoServAddr.sin_family = AF_INET;
    echoServAddr.sin_addr.s_addr = IP_ADDRESS(1,2,3,4);
    echoServAddr.sin_port = SERVER_PORT;
#endif

    /* Now connect this client to the server */
#ifdef DUO
    status = connect(sock_tcp_client, (struct sockaddr *)&echoServAddr6,
                         sizeof(echoServAddr6));
#else
    status = connect(sock_tcp_client, (struct sockaddr *)&echoServAddr,
                         sizeof(echoServAddr));
#endif

    /* Check for error.  */
    if (status != OK)
    {
        printf("\nError: BSD TCP Client socket Connect\n");
        status = soc_close(sock_tcp_client);
        return;

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

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

    length = sizeof(struct sockaddr_in6);
    status = getpeername( sock_tcp_client, (struct sockaddr *) &remoteAddr6,
                                &length);
    printf("Remote port = %lu, Remote IP = 0x%x 0x%x 0x%x 0x%x \n",
                        remoteAddr6.sin6_port,
                        remoteAddr6.sin6_addr._S6_un._S6_u32[0],
                        remoteAddr6.sin6_addr._S6_un._S6_u32[1],
                        remoteAddr6.sin6_addr._S6_un._S6_u32[2],
                        remoteAddr6.sin6_addr._S6_un._S6_u32[3]);
#else
    status = getsockname(sock_tcp_client, (struct sockaddr *)&localAddr, &length);
    printf("Client port = %lu , Client = 0x%x,", localAddr.sin_port,
                 localAddr.sin_addr.s_addr);
    length = sizeof(struct sockaddr_in);
    status = getpeername( sock_tcp_client, (struct sockaddr *) &remoteAddr,
                         &length);
    printf("Remote port = %lu, Remote IP = 0x%x \n", remoteAddr.sin_port,
                         remoteAddr.sin_addr.s_addr);
#endif

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

        printf("\nClient sock: %d Sending packet to server\n",sock_tcp_client);

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

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

        status = recv(sock_tcp_client, (VOID *)rcvBuffer1, 32,0);

        if (status <= 0)
        {

            if (status < 0)
            {
                printf("\nError: BSD Client Socket receive %d \n",sock_tcp_client);
            }
            else
            {
                printf("Nothing received by Client\n");
            }
            break;
        }
    }

    /* close this client socket */
    status = soc_close(sock_tcp_client);
    if (status != ERROR)
        printf("\nBSD Client Socket Closed %d\n",sock_tcp_client);
    else
       printf("\nError: BSD Client Socket close %d \n",sock_tcp_client);

}
```

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 is 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 is a binds a TCP socket to listen for connection rquests 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 wouldl 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,
        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 = "ABCDEFGHIJKLMNOPQRSTUVWXYZ<>END";
30   INT      maxfd;
31
32   /* Define the counters used in the demo application...  */
33
```

```
ULONG            error_counter;

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


/* Set a flag to indicate if a callback for connection complete is set.  */
UINT  establish_cb_set = NX_FALSE;

/* Define thread prototypes.  */

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

#ifdef NX_EXTENDED_BSD_SOCKET_SUPPORT

/* Set the following socket options available with BSD extended support. */


/*  Set sockets to non blocking for connecting, disconnecting and rx/tx'ing
    packets */
#define ENABLE_NONBLOCKING

/* 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. */
#define ENABLE_ESTABLISH_CB

/* Set the disconnect callback which NetX Duo will invoke when a disconnection is
   complete. */
#define ENABLE_DISCONNECT_CB

/* 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
*/

/* 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
*/

/* 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. */
VOID        bsd_tcp_establish_notify(NX_TCP_SOCKET *socket_ptr);
VOID        bsd_tcp_disconnect_complete_notify(NX_TCP_SOCKET *socket_ptr);

#endif /* NX_EXTENDED_BSD_SOCKET_SUPPORT */

/* Define main entry point.  */

int main()
{

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

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

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


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

    /* Create a server thread.  */
    tx_thread_create(&thread_server, "Server", thread_server_entry, 0,
                          pointer, DEMO_STACK_SIZE, 2, 2, TX_NO_TIME_SLICE,
                          TX_AUTO_START);

    pointer =  pointer + DEMO_STACK_SIZE;

    /* Create a client thread.  */
    tx_thread_create(&thread_client, "Client", thread_client_entry, 0,
                         pointer, DEMO_STACK_SIZE, 4, 4, TX_NO_TIME_SLICE,
                         TX_AUTO_START);

    pointer =  pointer + DEMO_STACK_SIZE;


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

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

    /* Create an IP instance for BSD.  */
    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);
    pointer =  pointer + 2048;

    if (status)
    {
         error_counter++;
    }

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

    /* Check ARP enable status.  */
    if (status)
    {
        error_counter++;
    }

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

    status = nx_tcp_enable(&bsd_ip);

    /* Check TCP enable status.  */
    if (status)
    {
        error_counter++;
    }

   /* Enable BSD socket callbacks if BSD extended support is set.  */

#ifdef NX_EXTENDED_BSD_SOCKET_SUPPORT

#ifdef ENABLE_ESTABLISH_CB

    /* Note that this callback eliminates the need to call accept(). */
    status = nx_bsd_set_socket_establish_notify(bsd_tcp_establish_notify);

    /* Check completion status.  */
    if (status)
    {
        return;
    }

    establish_cb_set = NX_TRUE;
#endif

#ifdef ENABLE_DISCONNECT_CB

    status = nx_bsd_set_socket_disconnect_complete_notify
                                (bsd_tcp_disconnect_complete_notify);

    /* Check completion status.  */
    if (status)
    {
        printf("disconnect complete notify called...\n");
        return;
    }
#endif

#endif /* NX_EXTENDED_BSD_SOCKET_SUPPORT */

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


VOID  thread_client_entry(ULONG thread_input)
{

INT         status;
INT         bytes;
INT         sock_tcp_client, length;
struct      sockaddr_in echoServAddr;               /* Echo server address */
struct      sockaddr_in localAddr;                  /* Local address */
struct      sockaddr_in remoteAddr;                 /* Remote address */
CHAR        ClientBuffer[132];


```

```
    /* Give the server thread time to set up. */
    tx_thread_sleep(100);

    /* Create Client TCP Socket */
    sock_tcp_client = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP);

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

    printf("\nBSD TCP Client socket created %lu \n", sock_tcp_client);

    /* Fill destination port and IP address */
    memset(&echoServAddr, 0, sizeof(echoServAddr));
    echoServAddr.sin_family = PF_INET;
    echoServAddr.sin_addr.s_addr = IP_ADDRESS(1,2,3,4);
    echoServAddr.sin_port = SERVER_PORT;

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

    /* Check for error.  */
    if (status != OK)
    {

        printf("Client socket %d error on connect\n", sock_tcp_client);

        status = soc_close(sock_tcp_client);

        return;
    }

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

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


    /* Now receive the echoed packet from the server */
    while(1)
    {
        tx_thread_sleep(2);

        printf("Client sock %d sending packet to server\n", sock_tcp_client);
        status = send(sock_tcp_client, msg0, (strlen(msg0)+1), 0);

        if (status == ERROR)
            printf("Client socket %d error on send\n", sock_tcp_client);
        else
        {
            printf("Client %d sent message: %s\n", sock_tcp_client, msg0);
        }

        bytes = recv(sock_tcp_client, (VOID *)ClientBuffer, 132 ,0);
        if (bytes == 0)
            break;

        if (bytes != NX_SOC_ERROR)
            printf("Client socket %d received %lu bytes: %s\n", sock_tcp_client +
                        NX_BSD_SOCKFD_START, bytes, ClientBuffer);
        else
            printf("Client socket %d error on receive\n", sock_tcp_client, bytes);

    }

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

    if (status != ERROR)
        printf("Client socket closed %d\n", sock_tcp_client);
    else
       printf("Client socket error on close\n");

    /* End */
}

/* Define the Server thread.  */

VOID  thread_server_entry(ULONG thread_input)
{


INT         status, sock, sock_tcp_server;
INT         i;
UINT        is_set = NX_FALSE;
struct      sockaddr_in echoServAddr;
struct      sockaddr_in ClientAddr;
INT         ClientLength;
CHAR        ServerBuffer[132];
#ifdef NX_EXTENDED_BSD_SOCKET_SUPPORT
struct      timeval select_timeout;
UINT        nonblocking_enabled = NX_FALSE;
INT         result;
INT         option_length;
#ifdef ENABLE_KEEPALIVE
struct      sock_keepalive keepalive;
#endif
#if ENABLE_LINGER
struct      sock_linger  linger;
#endif
#endif


    /* Let NetX and the driver get initialized. */
    tx_thread_sleep(100);

    /* Create the Server TCP Socket */
    sock_tcp_server = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP);

    if (sock_tcp_server == -1)
    {
        printf("Server socket error on creating secondary socket.\n");
        return;
    }

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

#ifdef NX_EXTENDED_BSD_SOCKET_SUPPORT

    /* Enable various socket options if BSD extended socket support is set. */

#ifdef ENABLE_KEEPALIVE
    /* To enable keepalive, the NetX Duo library must be compiled with
       NX_TCP_ENABLE_KEEPALIVE is defined. */
    keepalive.keepalive_enabled = NX_TRUE;
    status = setsockopt(sock_tcp_server, SOL_SOCKET, SO_KEEPALIVE, &keepalive,
                        sizeof(keepalive));
#endif

#if ENABLE_LINGER
    linger.linger_onoff = NX_TRUE;
    linger.linger_time = 15;
    status = setsockopt(sock_tcp_server, SOL_SOCKET, SO_LINGER, &linger,
                        sizeof(linger));
#endif

#ifdef ENABLE_NONBLOCKING
    /* If not lingering, set to non blocking */
    fnctl(sock_tcp_server, F_SETFL, SO_NONBLOCK);
    nonblocking_enabled = NX_TRUE;
#endif
#endif

    /* Set the server port and IP address */
    memset(&echoServAddr, 0, sizeof(echoServAddr));
    echoServAddr.sin_family = AF_INET;
    echoServAddr.sin_addr.s_addr = IP_ADDRESS(1,2,3,4);
    echoServAddr.sin_port = SERVER_PORT;

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

    if (status < 0)
    {

        return;
    }

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

    /* Now listen for any client connections for this server socket */
    status = listen (sock_tcp_server, 5);
    if (status < 0)
    {

        return;
    }
    else
        printf("Server socket is listening...\n");

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

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

        read_ready = master_list;

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

#ifdef NX_EXTENDED_BSD_SOCKET_SUPPORT

        /* Check if nonblock flag set on this master socket. */
        if (nonblocking_enabled)
        {

            /* For a non blocking call, select() cannot send in a null timeout! */
            select_timeout.tv_sec   = 0;
            select_timeout.tv_usec  = 0;
            status = select(maxfd + 1, &read_ready, 0, 0, &select_timeout);
        }
        else
        {

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

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

            if (status == 0xFFFFFFFF)
            {

                option_length = sizeof(INT);

                /* Demonstrate socket error handling. Check if socket error is a
                   result of timing out (in progress, for example)
                   or an actual socket connect error.  */
                status = getsockopt(sock_tcp_server, SOL_SOCKET, SO_ERROR,
                                 (INT *)&result, &option_length);

                /* Check if this is a nonblocking socket error. */
                if (result == EWOULDBLOCK)
                {
                    /* This is a non blocking error; we simply just don't have a
                       receive/connect event yet. */
                    printf("Server socket status on select: In progress...\n");
                    tx_thread_sleep(100);
                }
                else
                    /* The connection failed.  */
                    printf("Server socket error status on select: 0x%x...\n",
                               result);
            }

            continue;
        }

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

        if (status <= 0)
        {
            if (status < 0)
            {

                printf("Server error on select. Try again\n");
            }

            continue;
        }

        printf("Detect a connection request\n");
#endif

        /* If the BSD server socket does not have an connection callback function,
           handle the connection request here. */
        if (establish_cb_set == NX_FALSE)
        {

            is_set = FD_ISSET(sock_tcp_server, &read_ready);

            if(is_set)
            {

                ClientLength = sizeof(ClientAddr);

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

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

                if ( sock > maxfd)
                {
                    printf("Server has a new connection on socket %d\n", sock);

                    maxfd = sock;
                }

                continue;
            }
        }

        /* Check the set of 'ready' sockets, e.g connected to remote host and
           waiting for  notice of packets received. */
        for (i = 0; i < (maxfd+1); i++)
        {

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

                printf("Server received data on socket %d\n", i +
                        NX_BSD_SOCKFD_START);
                while(1)
                {
                    tx_thread_sleep(2);

                    status = recv(i + NX_BSD_SOCKFD_START, (VOID *)ServerBuffer,
                                 132,0);

                    if (status == 0)
                        break;
                    if (status != 0xFFFFFFFF)
                    {
                        printf("\nServer socket %d received %lu bytes: %s ",
                        sock_tcp_server, strlen(ServerBuffer),ServerBuffer);
                    }
                    else
                    {
                        printf("Server socket %d received data\n",
                               sock_tcp_server);
                        break;
                    }

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

                    if (status == ERROR)
                        printf("Server socket %d error on send\n", i +
                               NX_BSD_SOCKFD_START);
                    else
                    {
                        printf("Server socket %d sent message Hello\n", i +
                        NX_BSD_SOCKFD_START);
                    }
                }

                /* close this client socket */
                status = soc_close(i+ NX_BSD_SOCKFD_START);

                if (status != ERROR)
                    printf("Server socket %d closing \n", i+ NX_BSD_SOCKFD_START);
                else
                   printf("Server socket %d error on close\n", i+
                        NX_BSD_SOCKFD_START);
            }
        }

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


#ifdef NX_EXTENDED_BSD_SOCKET_SUPPORT

/* Define TCP callback function.  Note that both Server and Clients socket
   connections  will activate these callbacks, so the socket ID is the key to
   determining which socket is notified of connection completed or disconnect
   completed.  */
/*  Process the connection for the host application e.g update the socket list of
    ready sockets  with the new connection. */
VOID  bsd_tcp_establish_notify(NX_TCP_SOCKET *socket_ptr)
{

UINT bsd_socket_index;


    /* Figure out what BSD socket this is.  */
    bsd_socket_index =  (UINT) socket_ptr -> nx_tcp_socket_reserved_ptr;
    printf("Host has a connection on socket %d!\n", bsd_socket_index +
                 NX_BSD_SOCKFD_START);

    if (bsd_socket_index + NX_BSD_SOCKFD_START > maxfd)
    {

        /* This is now the highest socket index to check. */
        maxfd = bsd_socket_index;
    }

    FD_SET(bsd_socket_index + NX_BSD_SOCKFD_START, &master_list);
    return;
}

/*  Process the disconnection for the host application e.g update the socket list
    of ready sockets with the current connection removed. */
VOID  bsd_tcp_disconnect_complete_notify(NX_TCP_SOCKET *socket_ptr)
{

UINT bsd_socket_index;


    bsd_socket_index =  (UINT) socket_ptr -> nx_tcp_socket_reserved_ptr;

    printf("Host disconnect completed for %d!\n", bsd_socket_index +
                 NX_BSD_SOCKFD_START);

    FD_CLR(bsd_socket_index + NX_BSD_SOCKFD_START, &master_list);

    return;
}

#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_KEEPALIVE
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_ASYNCH_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.