BSD 4.3 Sockets API Compliancy Wrapper for NetX

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

Trademarks
NetX, 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 products will meet the USER's requirements, or will operate in the manner specified by the USER, or that the operation of the NetX products will operate uninterrupted or error free, or that any defects that may exist in the NetX 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 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 5.7.1

Contents

Chapter 1 Introduction to NetX BSD 4
BSD Sockets API Compliancy Wrapper Source..4
Chapter 2 Installation and Use of NetX BSD 4
Product Distribution..5
NetX BSD Installation..5
Using NetX BSD..5
NetX BSD Multihome Support...6
NetX BSD Limitations..6
Configuration Options..6
Small Example System 7
Chapter 3 NetX BSD Services 14
Appendix A NetX Extended Services for BSD 16
NetX BSD Extended Services (API)...16
Internal BSD Extended Services 17
BSD Socket Options with Extended Services 17
Configurable Options For NetX BSD Extended Services 19

Chapter 1

Introduction to NetX BSD

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

BSD Sockets API Compliancy Wrapper Source

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

The package consists of:

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

Sample demo programs:

bsd_demo_tcp.c

Demo with a single TCP server and client

bsd_demo_udp.c

Demo with two UDP clients and a UDP server

Chapter 2

Installation and Use of NetX BSD

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

Product Distribution

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

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

Demo files:
bsd_demo_tcp.c
bsd_demo_udp.c

NetX BSD Installation

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

Using NetX BSD

Using BSD for NetX is easy. Basically, the application code must include *nx_bsd.h* after it includes *tx_api.h* and *nx_api.h*, in order to use ThreadX and NetX, respectively. Once *nx_bsd.h* is included, the application code is then able to use the BSD services specified later in this guide. The application must also include *nx_bsd.c* in the build process. This file must be compiled in the same manner as other application files and its object form must be linked along with the files of the application. This is all that is required to use NetX BSD.

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

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

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

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

NetX BSD Multihome Support

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

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

NetX BSD Limitations

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

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

Configuration Options

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

Define	**Meaning**
NX_BSD_TCP_WINDOW	Used in TCP socket create calls. 65535 is a typical window size for 100Mb Ethernet. The default value is 65535.

NX_BSD_SOCKFD_START

This is the logical index for the BSD socket file descriptor start value. By default this option is 32.

NX_BSD_MAX_SOCKETS

Specifies the maximum number of total sockets available in the BSD layer and must be a multiple of 32.The value is defaulted to 32.

NX_BSD_MAX_LISTEN_BACKLOG

This specifies the size of the listen queue (‘backlog’) for BSD TCP sockets. The default value is 5.

NX_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 internal calls required by BSD. The default value is 20*NX_CPU_TICKS_PER_SECOND.

NX_BSD_PRINT_ERRORS

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

Small Example System

An example of how to use NetX BSD is shown in Figure 1.0 below. In this example, the include file *nx_bsd.h* is brought in at line 7. Next, the IP instance `bsd_ip` and packet pool `bsd_pool` are created as global variables at line 20 and 21.   Note that this demo uses a ram (virtual) network driver (line 41). The client and server will share the same IP address on single IP instance in this example.

The client and server threads are created on line 303 and 309 in *tx_application_define* which sets up the application and is defined on lines 293-361. After IP instance successful creation on line 327, the IP instance is enabled for TCP services on line 350. The last requirement before BSD services can be used is to call *bsd_initialize* on line 360 to set up all data structures and NetX, and ThreadX resources needed by BSD.

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

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

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

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

```
1     /* This is a small demo of BSD Wrapper for the high-performance NetX TCP/IP stack.
2        This demo demonstrate TCP connection, disconnection, sending, and receiving using
3        ARP and a simulated Ethernet driver.  */
4
5     #include          "tx_api.h"
6     #include          "nx_api.h"
7     #include          "nx_bsd.h"
8     #include          <string.h>
9     #include          <stdlib.h>
10
11    #define           DEMO_STACK_SIZE     (16*1024)
12
13
14    /* Define the ThreadX and NetX object control blocks... */
15
16    TX_THREAD         thread_0;
17    TX_THREAD         thread_1;
18
19
```

```
NX_PACKET_POOL   bsd_pool;
NX_IP            bsd_ip;


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

ULONG            error_counter;

/* Define fd_set for select call */
fd_set           master_list,read_ready,read_ready1;


/* Define thread prototypes.  */

VOID             thread_0_entry(ULONG thread_input);
VOID             thread_1_entry(ULONG thread_input);

VOID             tcpClient(CHAR *msg0);
VOID             tcpServer(VOID);
INT              HandleClient(INT  sock);

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();
}


VOID  tcpClient(CHAR *msg0)
{

INT         status,status1,send_counter;
INT         sock_tcp_1,length,length1;
struct      sockaddr_in echoServAddr;               /* Echo server address */
struct      sockaddr_in localAddr;                  /* Local address */
struct      sockaddr_in remoteAddr;                 /* Remote address */

UINT         echoServPort;                          /*  Echo Server Port */
CHAR         rcvBuffer1[32];


    /* Create BSD TCP Socket */
    sock_tcp_1 = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock_tcp_1 == -1)
    {
        printf("\nError: BSD TCP Client socket create \n");
        return;
    }

    printf("\nBSD TCP Client socket created %lu \n", sock_tcp_1);
    /* Fill destination port and IP address */
    echoServPort = 12;
    memset(&echoServAddr, 0, sizeof(echoServAddr));
    echoServAddr.sin_family = PF_INET;
    echoServAddr.sin_addr.s_addr = htonl(0x01020304);
    echoServAddr.sin_port = echoServPort;

    /* Now connect this client the server */
    status1 = connect(sock_tcp_1, (struct sockaddr *)&echoServAddr,
                     sizeof(echoServAddr));
    /* Check for error.  */
    if (status1 != OK)
    {
        printf("\nError: BSD TCP Client socket Connect, %d \n",sock_tcp_1);
        status = soc_close(sock_tcp_1);
        if (status != ERROR)
            printf("\nConnect ERROR so BSD Client Socket Closed: %d\n",sock_tcp_1);
        else
            printf("\nError: BSD Client Socket close %d\n",sock_tcp_1);
        return;
    }

    /* Get and print source and destination information */
    printf("\nBSD TCP Client socket: %d connected \n", sock_tcp_1);

    status = getsockname(sock_tcp_1, (struct sockaddr *)&localAddr, &length);
    printf("Client port = %lu , Client = %lu,", localAddr.sin_port,
           localAddr.sin_addr.s_addr);

    status = getpeername( sock_tcp_1, (struct sockaddr *) &remoteAddr, &length1);
    printf("Remote port = %lu, Remote IP = %lu \n", remoteAddr.sin_port,
            remoteAddr.sin_addr.s_addr);

    send_counter = 1;

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

        printf("\nClient sock: %d Sending packet No: %d to
               server\n",sock_tcp_1,send_counter);

        status = send(sock_tcp_1,msg0, ( strlen(msg0)+1), 0);
        if (status == ERROR)
            printf("Error: BSD Client Socket send %d\n",sock_tcp_1);
        else
        {
            printf("\nMessage sent: %s\n",msg0);
            send_counter++;
        }

        status = recv(sock_tcp_1, (VOID *)rcvBuffer1, 31,0);
        if (status == 0)
            break;

        rcvBuffer1[status] = 0;

        if (status != ERROR)
            printf("\nBSD Client Socket: %d received %lu bytes: %s ",
                    sock_tcp_1,strlen(rcvBuffer1),rcvBuffer1);
        else
            printf("\nError: BSD Client Socket receive %d \n",sock_tcp_1);

    }

    /* close this client socket */
    status = soc_close(sock_tcp_1);
    if (status != ERROR)
        printf("\nBSD Client Socket Closed %d\n",sock_tcp_1);
    else
        printf("\nError: BSD Client Socket close %d \n",sock_tcp_1);

    /* End */
}



void tcpServer(void)
{

INT         status,status1,sock,sock_tcp_2,i;
struct      sockaddr_in echoServAddr;               /* Echo server address */
struct      sockaddr_in ClientAddr;

INT         Clientlen;
UINT        echoServPort;                          /*  Echo Server Port */

INT         maxfd;

    /* Create BSD TCP Server Socket */
    sock_tcp_2 = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock_tcp_2 == -1)
    {
        printf("Error: BSD TCP Server socket create\n");
        return;
    }
    else
        printf("BSD TCP Server socket created \n");

    /* Now fill server side information */
    echoServPort = 12;
    memset(&echoServAddr, 0, sizeof(echoServAddr));
    echoServAddr.sin_family = PF_INET;
    echoServAddr.sin_addr.s_addr = htonl(0x01020304);
    echoServAddr.sin_port = echoServPort;

    /* Bind this server socket */
    status = bind(sock_tcp_2, (struct sockaddr *) &echoServAddr,
                  sizeof(echoServAddr));
    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_2,&master_list);
    maxfd = sock_tcp_2;

    /* Now listen for any client connections for this server socket */
    status = listen(sock_tcp_2,5);
    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(2);   /* Allow some time to other threads too */
        status = select(maxfd+1,&read_ready,0,0,0);
        if (status == ERROR)
        {
            continue;
        }

        status = FD_ISSET(sock_tcp_2,&read_ready);
        if(status)
        {
            sock = accept(sock_tcp_2,(struct sockaddr*)&ClientAddr, &Clientlen);

            /* Add this new connection to our master list */
            FD_SET(sock,&master_list);
            if ( sock > maxfd)
            {
                maxfd = sock;
            }

            continue;
        }
        for (i = 0; i < (maxfd+1); i++)
        {
            if (( i != sock_tcp_2) && (FD_ISSET(i,&master_list)) &&
                (FD_ISSET(i,&read_ready)))
            {
                status1 = HandleClient(i);
                if (status1 == 0)
                {
                    status1 = soc_close(i);
                    if (status1 == OK)
                    {
                        FD_CLR(i,&master_list);
                        printf("\nBSD Server Socket:%d closed\n",i);
                    }
                    else
                        printf("\nError:BSD Server Socket:%d close\n",i);
                }

            }
        }

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

    } /* while(1) ends */

```

```
}

CHAR         rcvBuffer[128];

INT     HandleClient(INT  sock)
{

INT          status;


    status = recv(sock, (VOID *)rcvBuffer, 128,0);
    if (status == ERROR )
    {
        printf("\n BSD Server Socket:%d receive \n",sock);
        return(ERROR);
    }

    /* a zero return from a recv() call indicates client is terminated! */
    if (status == 0)
    {
        /* Done with this client , close this secondary server socket */
        return(status);
    }

    /* print data received from the client */
    printf("\nBSD Server Socket:%d received %lu bytes: %s \n", sock,
           strlen(rcvBuffer),rcvBuffer);

    /* And echo the same data to the client */
    status = send(sock,rcvBuffer, ( strlen(rcvBuffer)+1), 0);
    if (status == ERROR )
    {
        printf("\nError: BSD Server Socket:%d send \n",sock);
        return(ERROR);
    }
    return(status);
}


/* 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 client thread.  */
    tx_thread_create(&thread_0, "Client1", thread_0_entry, 0,
            pointer, DEMO_STACK_SIZE, 2, 2, TX_NO_TIME_SLICE, TX_AUTO_START);

    pointer =  pointer + DEMO_STACK_SIZE;

    /* Create a server thread.  */
    tx_thread_create(&thread_1, "Server", thread_1_entry, 0,
            pointer, DEMO_STACK_SIZE,1,1, 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, "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++;
        printf("Error creating BSD IP instance\n!");
    }

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

    /* Check ARP enable status.  */
    if (status)
    {
        error_counter++;
        printf("Error in Enable ARP and supply ARP cache memory to BSD IP
                instance\n");
    }

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

    status = nx_tcp_enable(&bsd_ip);

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

    /* Now initialize BSD Scoket Wrapper */
    status = bsd_initialize(&bsd_ip, &bsd_pool,pointer, 2048, 1);
}


/* Define the test threads.  */

void    thread_0_entry(ULONG thread_input)
{

CHAR    *msg0 = "Client 1:
                “ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABCDEFGHIJKLMNOPQRSTUVWXYZ<> \
                “ABCDEFGHIJKLMNOPQRSTUVWXYZ<>END";

    /* Wait till Server side is all set */
    tx_thread_sleep(2);
    while (1)
    {
        tcpClient(msg0);
        tx_thread_sleep(1);
    }
}

/* Define the server thread entry function. */
void    thread_1_entry(ULONG thread_input)
{

UINT    status;
ULONG   actual_status;

    /* Ensure the IP instance has been initialized.  */
    status =  nx_ip_status_check(&bsd_ip, NX_IP_INITIALIZE_DONE, &actual_status,
100);

    /* Check status...  */
    if (status != NX_SUCCESS)
    {
        error_counter++;
        return;
    }
    /* Start a TCP Server */
    tcpServer();
}

```

Chapter 3

NetX BSD Services

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

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

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

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

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

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

in_addr_t inet_addr(const_CHAR *buffer);

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

CHAR inet_ntoa(struct in_addr address_to_convert);

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 Extended Services for BSD

Description of NetX Extended Services for BSD

The NetX BSD Extended Services adds new socket features to NetX BSD sockets. These include socket error handling, non-blocking sockets and keep-alive TCP sockets. Extended Services are available for those versions of NetX that support multiple interfaces (v5.4 or later).

To use the NetX BSD Extended Services, the NetX library must be built with NX_ENABLE_EXTENDED_NOTIFY_SUPPORT defined. The default setting is disabled (not defined). In addition, NX_EXTENDED_BSD_SOCKET_SUPPORT must be enabled (defined) in *nx_bsd.h*. This default setting is enabled (defined).

The following steps describe how to set up a NetX BSD application for NetX 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.

4. Build NetX with NX_ENABLE_EXTENDED_NOTIFY_SUPPORT enabled (defined).

5. The BSD application must define NX_EXTENDED_BSD_SOCKET_SUPPORT at the project level or in both *nx_bsd.h* and in the application code.

NetX BSD Extended Services (API)

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

Enables or disables the specified socket ID with the specified options. Currently only non-blocking is supported. Flag type *is set to F_SETFL to set, and* f_option *set to O_NONBLOCK to enable the non-blocking option. To disable non-blocking, use* fcntl *with* f_options *set to NX_FALSE.*

.

INT ioctl(INT sock_ID, UINT command, UINT *result);
The FIONBIO command enables or disables the socket for 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 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

Internal BSD Extended Services

If NX_EXTENDED_BSD_SOCKET_SUPPORT is enabled in *nx_bsd.h*, BSD sockets have a socket error status. The application can obtain the status of a socket operation by calling *getsockopt* with the SO_ERROR socket option and socket level set to SOL_SOCKET:

```
INT result;
INT option_length;

option_length = sizeof(INT);

/* Check for error on previous socket operation. */
status = getsockopt(tcp_sock_id, SOL_SOCKET, SO_ERROR, (INT *)&result,
                    &option_length);
```

Additionally, when a TCP socket is created, NetX BSD applies a disconnect complete callback and connection established callback function.   These are not BSD API but serve to notify the BSD when a TCP connection or disconnect is complete such that all threads suspended on the associated socket are resumed.

BSD Socket Options with Extended Services

BSD socket options in *nx_bsd.h* allow the application to enable various BSD socket features on a per socket basis for its particular requirements. The following is a list of these parameters:

The first set of extended socket options are of the socket category and take the socket level SOL_SOCKET in the *setsockopt* and *getsockopt* service calls.

SO_BROADCAST	Enables sending and receiving broadcast packets from Netx sockets. This is the default behavior for NetX. All sockets have this capability.

SO_ERROR

This is used to obtain socket status on the previous socket operation of the specified socket, using the *getsockopt* service. All sockets have this capability.

SO_KEEPALIVE

Enables the TCP Keep-Alive feature. This requires the NetX library to be built with NX_TCP_ENABLE_KEEPALIVE defined in *nx_user.h*. By default this feature is disabled.

SO_RCVTIMEO

This sets the wait option in seconds for receiving packets on NetX BSD sockets. The default value is the NX_WAIT_FOREVER (0xFFFFFFFF) or, if non-blocking is enabled, NX_NO_WAIT (0x0).

SO_RCVBUF

This sets the window size of the TCP socket. The default value, NX_BSD_TCP_WINDOW, is set to 65535 for BSD TCP sockets. To set the size above 65535 requires the NetX library to be built with the NX_TCP_ENABLE_WINDOW_SCALING be defined.

SO_REUSEADDR

This enables multiple sockets to be mapped to the same port. The typical usage is for the TCP Server socket. This is the default behavior of NetX sockets.

The second set of extended socket options are of the IP category and take the socket level IPPROTO_IP in the *setsockopt* and *getsockopt* service calls.

IP_MULTICAST_TTL

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

	setsockopt with this socket option.
IP_ADD_MEMBERSHIP	This flag if set enables the BSD socket (applies only to UDP sockets) to join the specified IGMP group.
IP_DROP_MEMBERSHIP	This flag if set enables the BSD socket (applies only to UDP sockets) to leave the specified IGMP group.

Configurable Options For NetX BSD Extended Services

Define	**Meaning**
NX_EXTENDED_BSD_SOCKET_SUPPORT	Enables BSD to support extended services and configurable options. By default it is enabled.
NX_BSD_TIMER_RATE	Rate at which the BSD timer runs. The BSD thread processes network tasks on every timer event set by the BSD timer function. By default it is 1 second, or (1 * NX_CPU_TICKS_PER_SECOND).
NX_BSD_INHERIT_LISTENER_SOCKET_SETTINGS	If defined, secondary sockets inherit the certain socket features) of the master (listening) socket. These are: non-blocking port re-use (SO_REUSEADDR) receive window size (SO_RCVBUF) By default this option is enabled.