

NetX Duo™

BSD 4.3 Socket API Wrapper for NetX Duo

User Guide

Renesas Synergy™ Platform

All information contained in these materials, including products and product specifications, represents information on the product at the time of publication and is subject to change by Renesas Electronics Corp. without notice. Please review the latest information published by Renesas Electronics Corp. through various means, including the Renesas Electronics Corp. website (<http://www.renesas.com>).

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
"Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
"High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user's manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

Renesas Synergy Specific Information

If you are using NetX Duo BSD for the Renesas Synergy platform, please use the following information.

Installation

Page 5: If you are using Renesas Synergy SSP and the e2 studio IDE, BSD will already be installed. You can ignore the Installation and Use of NetX Duo BSD section.

Product Distribution

Page 5: The distribution of BSD included with the Renesas Synergy SSP installation does not include the files **bsd_demo_udp.c**, **bsd_demo_tcp.c**, and **bsd_demo_raw.c**. Please ignore references to these files.

DNS Support

Page 10: NetX Duo BSD support for DNS has not been tested for SSP v1.5.0.



BSD 4.3 Socket API Wrapper for NetX Duo

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

©2002-2018 by Express Logic, Inc.

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

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

Trademarks

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

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

Warranty Limitations

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

Part Number: 000-1050

Revision 5.11

Contents

Chapter 1 Introduction to NetX Duo BSD.....	4
BSD Socket API Compliancy Wrapper Source.....	4
Chapter 2 Installation and Use of NetX Duo BSD	5
Product Distribution.....	5
NetX Duo BSD Installation	5
Building the ThreadX and NetX Duo components of a BSD Application	5
Using NetX Duo BSD.....	6
NetX Duo BSD Raw Socket Support	6
NetX Duo BSD Raw Packet Support.....	8
Eliminating Internal BSD Thread	10
NetX Duo BSD with DNS Support	10
NetX Duo BSD Limitations.....	11
General Configuration Options.....	12
BSD Socket Options	15
Small IPv4 Example	17
Small IPv6 Example System.....	25
Chapter 3 NetX Duo BSD Services	33

Chapter 1

Introduction to NetX Duo BSD

The BSD Socket API Compliancy Wrapper supports some of the basic BSD Socket API calls, with some limitations and utilizes NetX Duo primitives underneath.

BSD Socket API Compliancy Wrapper Source

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

The package consists of:

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

Sample demo programs:

<i>bsd_demo_udp.c</i>	
<i>Demo with two UDP peers (IPv4 only)</i>	
<i>bsd_demo_tcp.c</i>	
<i>Demo with a single TCP server and client</i>	
<i>bsd_demo_raw.c</i>	
<i>Demo with two RAW peers</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:

bsd_demo_udp.c
bsd_demo_tcp.c
bsd_demo_raw.c

NetX Duo BSD Installation

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

Building the ThreadX and NetX Duo components of a BSD Application

ThreadX

The ThreadX library must define `bsd_errno` in the thread local storage. We recommend the following procedure:

1. In *tx_port.h*, set one of the `TX_THREAD_EXTENSION` macros as follows:


```
#define TX_THREAD_EXTENSION_3      int bsd_errno
```

2. Rebuild the ThreadX library.

Note that if `TX_THREAD_EXTENSION_3` is already used, the user is free to use one of the other `TX_THREAD_EXTENSION` macros.

NetX Duo

Before using NetX Duo BSD Services, the NetX Duo library must be built with `NX_ENABLE_EXTENDED_NOTIFY_SUPPORT` defined. By default it is not defined. If the BSD raw sockets are to be used, the NetX Duo library must be built with `NX_ENABLE_IP_RAW_PACKET_FILTER` defined.

Using NetX Duo BSD

A NetX Duo BSD application project must include *nxd_bsd.h* after it includes *tx_api.h* and *nx_api.h* to be able to use BSD services specified later in this guide. The application must also include *nxd_bsd.c* in the build process. This file must be compiled in the same manner as other application files and its object form must be linked along with the files of the application. This is all that is required to use NetX Duo BSD.

To utilize NetX Duo BSD services, the application must create an IP instance, create a packet pool for the BSD layer to allocate packets from, allocate memory space for the internal BSD thread stack, and specify the priority of the internal BSD thread. The BSD layer is initialized by calling *bsd_initialize* and passing in the parameters. This is demonstrated in the “Small Examples” later in this document but the prototype is shown below:

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

The `default_ip` is the IP instance the BSD layer operates on. The `default_pool` is used by the BSD services to allocate packets from. The next two parameters: `bsd_thread_stack_area`, `bsd_thread_stack_size` defines the stack area used by the internal BSD thread, and the last parameter, `bsd_thread_priority`, sets the priority of the thread.

NetX Duo BSD Raw Socket Support

NetX Duo BSD also supports raw sockets. To use raw sockets in NetX Duo BSD, the NetX Duo library must be compiled with `NX_ENABLE_IP_RAW_PACKET_FILTER` defined. By default it is not defined. The application must then enable raw socket processing for a previously created IP instance by calling `nx_ip_raw_packet_enable`.

To create a raw socket in NetX Duo BSD, the application uses the `socket` create service and specifies the protocol family, socket type and protocol:

```
sock_1 = socket(INT protocolFamily, INT socket_type, INT protocol)
```

`protocolFamily` is `AF_INET` for IPv4 sockets, or `AF_INET6` for IPv6 sockets, assuming IPv6 is enabled on the IP instance. The `socket_type` must be set to `SOCK_RAW`. `protocol` is application specific.

To send and receive raw packets as well as close a raw socket, the application typically uses the same BSD services as for UDP e.g. `sendto`, `recvfrom`, `select` and `soc_close`. Raw sockets do not support either `accept` or `listen` BSD services.

- By default, received IPv4 raw data includes the IPv4 header. Conversely, received IPv6 raw data does not include the IPv6 header.
- By default, when sending either raw IPv6 or IPv4 packets, the BSD wrapper layer adds the IPv6 or IPv4 header before sending the data.

NetX Duo BSD supports additional raw socket options, including `IP_RAW_RX_NO_HEADER`, `IP_HDRINCL` and `IP_RAW_IPV6_HDRINCL`.

If `IP_RAW_RX_NO_HEADER` is set, the IPv4 header is removed so that the received data does not contain the IPv4 header, and the reported message length does not include the IPv4 header. For IPv6 sockets, by default the raw socket receive does not include IPv6 header, equivalent to having the `IP_RAW_RX_NO_HEADER` option set. Application may use the `setsockopt` service to clear the `IP_RAW_RX_NO_HEADER` option. Once the `IP_RAW_RX_NO_HEADER` option is cleared, the received IPv6 raw data would include the IPv6 header, and the reported message length includes the IPv6 header.

This option has no effect on IPv4 or IPv6 transmitted data.

If `IP_HDRINCL` is set, the application includes the IPv4 header when sending data. This option has no effect on IPv6 transmission and is not defined by default. If `IP_RAW_IPV6_HDRINCL` is set, the application includes

the IPv6 header when sending data. This option has no effect on IPv4 transmission and is not defined by default.

IP_HDRINCL and IP_RAW_IPV6_HDRINCL have no effect on IPv4 or IPv6 reception.

Note! The BSD 4.3 Socket specification specifies that the kernel must copy the raw packet to each socket receive buffer. However in NetX Duo BSD, if multiple sockets are created sharing the same protocol, the behavior is undefined.

NetX Duo BSD Raw Packet Support

To enable the raw packet support for PPPoE, NetX Duo BSD wrapper needs to be built with NX_BSD_RAW_PPPOE_SUPPORT enabled.

The following command creates a BSD socket to handle PPPoE raw packets:

```
Socketfd = socket(AF_PACKET, SOCK_RAW, protocol);
```

The current BSD implementation only supports two protocol types in AF_PACKET family

- ETHERTYPE_PPPOE_DISC
PPPoE Discovery packets. In the MAC data frame, the Discovery packets have the Ethernet frame type 0x8863.
- ETHERTYPE_PPPOE_SESS
PPP Session packets. In the MAC data frame, the Session packets have the Ethernet frame type 0x8864.

The structure `sockaddr_ll` is used to specify parameters when sending or receiving PPPoE frames.

`struct sockaddr_ll` is declared as:

```
struct sockaddr_ll
{
    USHORT sll_family;    /* Must be AF_PACKET */
    USHORT sll_protocol; /* LL frame type */
    INT     sll_ifindex;  /* Interface Index. */
    USHORT sll_hatype;    /* Header type */
    UCHAR  sll_pkttype;    /* Packet type */
    UCHAR  sll_halen;      /* Length of address */
    UCHAR  sll_addr[8];    /* Physical layer address */
};
```

Note that not every field in the structure is used by *sendto()* or *recvfrom()*. See the description below on how to set up the `sockaddr_ll` for sending and receiving PPPoE packets.

Socket created in the `AF_PACKET` family can be used to send either PPPoE Discovery packets or PPP session packets, regardless of the protocol specified. When transmitting a PPPoE packet, application must prepare the buffer that includes properly formatted PPPoE frame, including the MAC headers (Destination MAC address, source MAC address, and frame type.) The size of the buffer includes the 14-byte Ethernet header.

The `sockaddr_ll` struct, the `sll_ifindex` is used to indicate the physical interface to be used for sending this packet. The rest of the fields in the structure are not used. Values set to the unused fields are ignored by the BSD internal process.

The following code block illustrates how to transmit a PPPoE packet:

```
struct sockaddr_ll peer_addr;

/* Transmit a PPPoE frame using the primary network interface. */
peer_addr.sll_ifindex = 0;
n = sendto(sockfd, frame, frame_size, 0, (struct
      sockaddr*)&peer_addr, sizeof(peer_addr));
```

The return value indicates the number of bytes transmitted. Since PPPoE packets are message-based, on a successful transmission, the number of bytes sent matches the size of the packet, including the 14-byte Ethernet header.

PPPoE packets can be received using *recvfrom()*. The receive buffer must be big enough to accommodate message of Ethernet MTU size. The received PPPoE packet includes 14-byte Ethernet header. On

receiving PPPoE packets, PPPoE Discovery packets can only be received by socket created with protocol `ETHERTYPE_PPPOE_DISC`. Similarly, PPP session packets can only be received by socket created with protocol `ETHERTYPE_PPPOE_SESS`. If multiple sockets are created for the same protocol type, arriving PPPoE packets are forwarded to the socket created first. If the first socket created for the protocol is closed, the next socket in the order of creation is used for receiving these packets.

After a PPPoE packet is received, the following fields in the `sockaddr_ll` struct are valid:

- `sll_family`: Set by the BSD internal to be `AF_PACKET`
- `sll_ifindex`: Indicates the interface from which the packet is received
- `sll_protocol`: Set to the type of packet received:
`ETHERTYPE_PPPOE_DISC` or `ETHERTYPE_PPPOE_SESS`

Eliminating Internal BSD Thread

By default, BSD utilizes an internal thread to perform some of its processing. In systems with tight memory constraints, BSD can be built with `NX_BSD_TIMEOUT_PROCESS_IN_TIMER` defined, which eliminates the internal BSD thread and instead uses an internal timer to perform the same processing. This eliminates the memory required for the internal BSD thread control block and stack. However, overall timer processing is significantly increased and the BSD processing may also execute at a higher priority than needed.

To configure BSD sockets to run in the ThreadX timer context, define `NX_BSD_TIMEOUT_PROCESS_IN_TIMER` in *nxd_bsd.h*. If the BSD layer is configured to execute the BSD tasks in the timer context, in the call to *bsd_initialize*, the following three parameters are ignored, and should be set to NULL:

- `bsd_thread_stack_area`
- `bsd_thread_stack_size`
- `bsd_thread_priority`

NetX Duo BSD with DNS Support

If `NX_BSD_ENABLE_DNS` is defined, NetX Duo BSD can send DNS queries to obtain hostname or host IP information. This feature requires a NetX DNS Client to be previously created using the *nx_dns_create* service. One or more known DNS server IP addresses must be registered with the DNS instance using the *nx_dns_server_add* service for adding IPv4

server addresses, or using the *nxd_dns_server_add* service for adding either IPv4 or IPv6 server addresses.

DNS services and memory allocation are used by *getaddrinfo* and *getnameinfo* services:

```
INT getaddrinfo(const CHAR *node, const CHAR *service,
               const struct addrinfo *hints, struct addrinfo **res)

INT getnameinfo(const struct sockaddr *sa, socklen_t salen,
               char *host, size_t hostlen, char *serv, size_t servlen, int flags)
```

When the BSD application calls *getaddrinfo* with a hostname, NetX BSD will call any of the below services to obtain the IP address:

- *nx_dns_ipv4_address_by_name_get*
- *nxd_dns_ipv6_address_by_name_get*
- *nx_dns_cname_get*

For *nx_dns_ipv4_address_by_name_get* and *nxd_dns_ipv6_address_by_name_get*, NetX BSD uses the *ipv4_addr_buffer* and *ipv6_addr_buffer* memory areas respectively. The size of these buffers are defined by $(NX_BSD_IPV4_ADDR_PER_HOST * 4)$ and $(NX_BSD_IPV6_ADDR_PER_HOST * 16)$ respectively.

For returning address information from *getaddrinfo*, NetX BSD uses the ThreadX block memory table *nx_bsd_addrinfo_pool_memory*, whose memory area is defined by another set of configurable options, *NX_BSD_IPV4_ADDR_MAX_NUM* and *NX_BSD_IPV6_ADDR_MAX_NUM*.

See **General Configuration Options** for more details on the above configuration options.

Additionally, if *NX_DNS_ENABLE_EXTENDED_RR_TYPES* is defined, and the host input is a canonical name, NetX Duo BSD will allocate memory dynamically from a previously created block pool *_nx_bsd_cname_block_pool*

Note that after calling *getaddrinfo* the BSD application is responsible for releasing the memory pointed to by the *res* argument back to the block table using the *freeaddrinfo* service.

NetX Duo BSD Limitations

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

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

General Configuration Options

User configurable options in *nxd_bsd.h* allow the application to fine tune NetX Duo BSD sockets for its particular application requirements.

The following is the list of configurable options that are set at compile time:

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_SOCKET_QUEUE_MAX	Specifies the maximum number of UDP packets stored on the receive socket queue. The value is defaulted to 5.
NX_BSD_MAX_LISTEN_BACKLOG	This specifies the size of the listen queue ('backlog') for BSD TCP sockets. The default value is 5.

NX_MICROSECOND_PER_CPU_TICK	Specifies the number of microseconds per scheduler timer tick
NX_BSD_TIMEOUT	Specifies the timeout in timer ticks on NetX Duo internal calls required by BSD. The default value is (20 * NX_IP_PERIODIC_RATE).
NX_BSD_TCP_SOCKET_DISCONNECT_TIMEOUT	Specifies the timeout in timer ticks on NetX Duo disconnect call. The default value is 1.
NX_BSD_PRINT_ERRORS	If set, the error status return of a BSD function returns a line number and type of error e.g. NX_SOC_ERROR where the error occurs. This requires the application developer to define the debug output. The default setting is disabled and no debug output is specified in <i>nxd_bsd.h</i>
NX_BSD_TIMER_RATE	Interval after which BSD periodic timer task runs. The default value is 1 second (1 * NX_IP_PERIODIC_RATE).
NX_BSD_TIMEOUT_PROCESS_IN_TIMER	If set, this option allows the BSD timeout process to execute in the system timer context. The default behavior is disabled. This feature is described in more detail in Chapter 2 "Installation and Use of NetX Duo BSD".
NX_BSD_RAW_PPPOE_SUPPORT	Enable PPPoE raw packet support. By default this option is not enabled.
NX_BSD_ENABLE_DNS	If enabled, NetX Duo BSD will send a DNS query for a hostname or host IP address. Requires a DNS Client instance to be previously created and

started. By default it is not enabled.

NX_BSD_SOCKET_RAW_PROTOCOL_TABLE_SIZE

Defines the size of the raw socket table. The value must be a power of two. NetX BSD creates an array of sockets of type `NX_BSD_SOCKETS` for sending and receiving raw packets. `NX_ENABLE_IP_RAW_PACKET_FILTER` must be enabled. By default it is 32.

NX_BSD_IPV4_ADDR_MAX_NUM

Maximum number of IPv4 addresses returned by *getaddrinfo*. This along with `NX_BSD_IPV6_ADDR_MAX_NUM` defines the size of the NetX BSD block pool `nx_bsd_addrinfo_block_pool` for dynamically allocating memory to address information storage in *getaddrinfo*. The default value is 5.

NX_BSD_IPV6_ADDR_MAX_NUM

Maximum number of IPv6 addresses returned by *getaddrinfo*. This along with `NX_BSD_IPV4_ADDR_MAX_NUM` defines the size of the NetX BSD block pool `nx_bsd_addrinfo_block_pool` for dynamically allocating memory to address information storage in *getaddrinfo*.

NX_BSD_IPV4_ADDR_PER_HOST

Defines maximum IPv4 addresses stored per DNS query. The default value is 5.

NX_BSD_IPV6_ADDR_PER_HOST

Defines maximum IPv6 addresses stored per DNS query. The default value is 2.

BSD Socket Options

The following list of NetX Duo BSD socket options can be enabled (or disabled) at run time on a per socket basis using the *setsockopt* service:

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

There are two different settings for *option_level*.

The first type of run time socket options is *SOL_SOCKET* for socket level options. To enable a socket level option, call *setsockopt* with *option_level* set to *SOL_SOCKET* and *option_name* set to the specific option e.g. *SO_BROADCAST*. To retrieve an option setting, call *getsockopt* for the *option_name* with *option_level* again set to *SOL_SOCKET*.

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

<i>SO_BROADCAST</i>	If set, this enables sending and receiving broadcast packets from Netx sockets. This is the default behavior for NetX Duo. All sockets have this capability.
<i>SO_ERROR</i>	Used to obtain socket status on the previous socket operation of the specified socket, using the <i>getsockopt</i> service. All sockets have this capability.
<i>SO_KEEPALIVE</i>	If set, this enables the TCP Keep Alive feature. This requires the NetX Duo library to be built with <i>NX_TCP_ENABLE_KEEPALIVE</i> defined in <i>nx_user.h</i> . By default this feature is disabled.
<i>SO_RCVTIMEO</i>	This sets the wait option in seconds for receiving packets on NetX Duo BSD sockets. The default value is the <i>NX_WAIT_FOREVER</i> (0xFFFFFFFF) or, if non-blocking is enabled, <i>NX_NO_WAIT</i> (0x0).

SO_RCVBUF

This sets the window size of the TCP socket. The default value, `NX_BSD_TCP_WINDOW`, is set to 64k for BSD TCP sockets. To set the size over 65535 requires the NetX Duo library to be built with the `NX_TCP_ENABLE_WINDOW_SCALING` be defined.

SO_REUSEADDR

If set, this enables multiple sockets to be mapped to one port. The typical usage is for the TCP Server socket. This is the default behavior of NetX Duo sockets.

The second type of run time socket options is the IP option level. To enable an IP level option, call *setsockopt* with `option_level` set to `IPPROTO` and `option_name` set to the option e.g. `IP_MULTICAST_TTL`. To retrieve an option setting, call *getsockopt* for the `option_name` with `option_level` again set to `IPPROTO`.

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

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_RAW_IPV6_HDRINCL

If this option is set, the calling application must append an IPv6 header and optionally application headers to data being transmitted on raw IPv6 sockets created by BSD. To use this option, raw socket processing must be enabled on the IP task.

IP_ADD_MEMBERSHIP

If set, this options enables the BSD socket (applies only to UDP sockets) to join the specified IGMP group.

IP_DROP_MEMBERSHIP	If set, this options enables the BSD socket (applies only to UDP sockets) to leave the specified IGMP group.
IP_HDRINCL	If this option is set, the calling application must append the IP header and optionally application headers to data being transmitted on raw IPv4 sockets created in BSD. To use this option, raw socket processing must be enabled on the IP task.
IP_RAW_RX_NO_HEADER	<p>If cleared, the IPv6 header is included with the received data for raw IPv6 sockets created in BSD. IPv6 headers are removed by default in BSD raw IPv6 sockets, and the packet length does not include the IPv6 header.</p> <p>If set, the IPv4 header is removed from received data on BSD raw sockets of type IPv4. IPv4 headers are included by default in BSD raw IPv4 sockets and packet length includes the IPv4 header.</p> <p>This option has no effect on either IPv4 or IPv6 transmission data.</p>

Small IPv4 Example

An example of how to use NetX Duo BSD services for IPv4 networks is described below. In this example, the include file *nxd_bsd.h* is brought in at line 8. Next, the IP instance *bsd_ip* and packet pool *bsd_pool* are created as global variables at line 20 and 21. Note that this demo uses a ram (virtual) network driver, *_nx_ram_network_driver*. The client and server will share the same IP address on single IP instance in this example.

The client and server threads are created on lines 62 and 68. The BSD packet pool for transmitting packets is created on line 78 and used in the IP instance creation on line 87. Note that the IP thread task is given priority 1 in the *nx_ip_create* call. This thread should be the highest priority task defined in the program for optimal NetX performance.

The IP instance is enabled for ARP and TCP services on lines 88 and 110 respectively. The last requirement before BSD services can be used is to call *bsd_initialize* on line 120 to set up all data structures and NetX and ThreadX resources needed by BSD.

The server thread entry function is defined next. The BSD TCP socket is created on line 149. The server IP address and port are set on lines 160-163. Note the use of host to network byte order macros *htonl* and *htons* applied to the IP address and port. This is in compliance with BSD socket specification that multi byte data is submitted to the BSD services in network byte order.

Next, the master server socket is bound to the port using the *bind* service on line 166. This is the listening socket for TCP connection requests using the *listen* service on line 180. From here the server thread function, *thread_server_entry*, loops to check for receive events using the *select* call on line 202. If a receive event is a connection request, which is determined by comparing the read ready list, it calls *accept* on line 213. A child server socket is assigned to handle the connection request and added to the master list of TCP server sockets connected to a Client on line 223. If there are no new connection requests, the server thread then checks all the currently connected sockets for receive events in the for loop starting on line 236. When a receive event waiting is detected, it calls *send* and *recv* on that socket until no data is received (connection closed on the other side) and the socket is closed using the *soc_close* service on line 277.

After the server thread sets up, the Client thread entry function, *thread_client_entry*, creates a socket on line 326 and connects with the TCP server socket using the *connect* call on line 337. It then loops to send and receive packets using the *send* and *recv* services respectively. When no more data is received, it closes the socket on line 398 using the *soc_close* service. After disconnection, the client thread entry function creates a new TCP socket and makes another connection request in the while loop started on line 321.

```

1  /* This is a small demo of BSD wrapper for the high-performance NetX Duo
2     TCP/IP stack which uses standard BSD services for TCP connection, sending,
3     and receiving using a simulated Ethernet driver. */
4
5
```

```

6  #include      "tx_api.h"
7  #include      "nx_api.h"
8  #include      "nxd_bsd.h"
9  #include      <string.h>
10 #include      <stdlib.h>
11
12 #define        DEMO_STACK_SIZE      (16*1024)
13 #define        SERVER_PORT          87
14 #define        CLIENT_PORT          77
15
16 /* Define the ThreadX and NetX object control blocks... */
17
18 TX_THREAD      thread_server;
19 TX_THREAD      thread_client;
20 NX_PACKET_POOL bsd_pool;
21 NX_IP           bsd_ip;
22
23 /* Define some global data. */
24 CHAR *msg0 = "Client 1:
      ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABCDEFGHIJKLMNOPQRSTUVWXYZ<>END";
25
26 INT            maxfd;
27
28 /* Define the counters used in the demo application... */
29
30 ULONG          error_counter;
31
32 /* Define fd_sets for the BSD server socket. */
33 fd_set         master_list, read_ready;
34
35 /* Define thread prototypes. */
36
37 VOID           thread_server_entry(ULONG thread_input);
38 VOID           thread_client_entry(ULONG thread_input);
39 void           _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);
40
41 /* Define main entry point. */
42
43 int main()
44 {
45     /* Enter the ThreadX kernel. */
46     tx_kernel_enter();
47 }
48
49 /* Define what the initial system looks like. */
50
51 void tx_application_define(void *first_unused_memory)
52 {
53     CHAR *pointer;
54     UINT status;
55
56     /* Setup the working pointer. */
57     pointer = (CHAR *) first_unused_memory;
58
59     /* Create a server thread. */
60     tx_thread_create(&thread_server, "Server", thread_server_entry, 0,
61                     pointer, DEMO_STACK_SIZE, 8, 8, TX_NO_TIME_SLICE,
62                     TX_AUTO_START);
63
64     pointer = pointer + DEMO_STACK_SIZE;
65
66     /* Create a Client thread. */
67     tx_thread_create(&thread_client, "Client", thread_client_entry, 0,
68                     pointer, DEMO_STACK_SIZE, 16, 16, TX_NO_TIME_SLICE,
69                     TX_AUTO_START);
70
71     pointer = pointer + DEMO_STACK_SIZE;
72
73     /* Initialize the NetX system. */
74     nx_system_initialize();
75
76     /* Create a BSD packet pool. */
77     status = nx_packet_pool_create(&bsd_pool, "NetX BSD Packet Pool", 128,
78                                   pointer, 16384);
79     pointer = pointer + 16384;
80     if (status)

```

```

81     {
82         error_counter++;
83         printf("Error in creating BSD packet pool\n!");
84     }
85
86     /* Create an IP instance for BSD. */
87     status = nx_ip_create(&bsd_ip, "BSD IP Instance", IP_ADDRESS(1,2,3,4),
88                          0xFFFFFFFFUL, &bsd_pool, _nx_ram_network_driver,
89                          pointer, 2048, 1);
88     pointer = pointer + 2048;
89
90     if (status)
91     {
92         error_counter++;
93         printf("Error creating BSD IP instance\n!");
94     }
95
96     /* Enable ARP and supply ARP cache memory for BSD IP Instance */
97     status = nx_arp_enable(&bsd_ip, (void *) pointer, 1024);
98     pointer = pointer + 1024;
99
100    /* Check ARP enable status. */
101    if (status)
102    {
103        error_counter++;
104        printf("Error in Enable ARP \n");
105    }
106
107    /* Enable TCP processing for BSD IP instances. */
108    status = nx_tcp_enable(&bsd_ip);
109
110    /* Check TCP enable status. */
111    if (status)
112    {
113        error_counter++;
114        printf("Error in Enable TCP \n");
115    }
116
117    /* Now initialize BSD Scket Wrapper */
118    status = bsd_initialize (&bsd_ip, &bsd_pool, pointer, 2048, 2);
119 }
120
121
122
123
124 /* Define the Server thread. */
125 CHAR      Server_Rcv_Buffer[100];
126
127 VOID  thread_server_entry(ULONG thread_input)
128 {
129
130     INT      status, sock, sock_tcp_server;
131     ULONG    actual_status;
132     INT      ClientLen;
133     INT      i;
134     UINT     is_set = NX_FALSE;
135     struct   sockaddr_in serverAddr;
136     struct   sockaddr_in ClientAddr;
137
138
139     tx_thread_sleep(100);
140     status = nx_ip_status_check(&bsd_ip, NX_IP_INITIALIZE_DONE,
141                                &actual_status, 100);
142
143     /* Check status... */
144     if (status != NX_SUCCESS)
145     {
146         return;
147     }
148
149     /* Create BSD TCP Socket */
150     sock_tcp_server = socket(AF_INET, SOCK_STREAM, 0);
151
152     if (sock_tcp_server == -1)
153     {
154         printf("Error on Server socket %d create \n", sock_tcp_server);
155         return;
156     }
157
158     printf("Server socket %d created\n", sock_tcp_server);
159
160     /* Set the server port and IP address */

```

```

160     memset(&serverAddr, 0, sizeof(serverAddr));
161     serverAddr.sin_family = AF_INET;
162     serverAddr.sin_addr.s_addr = htonl(IP_ADDRESS(1,2,3,4));
163     serverAddr.sin_port = htons(SERVER_PORT);
164
165     /* Bind this server socket */
166     status = bind (sock_tcp_server, (struct sockaddr *) &serverAddr,
                    sizeof(serverAddr));
167
168     if (status < 0)
169     {
170         printf("Error on Server Socket %d Bind \n", sock_tcp_server);
171         return;
172     }
173
174     FD_ZERO(&master_list);
175     FD_ZERO(&read_ready);
176     FD_SET(sock_tcp_server,&master_list);
177     maxfd = sock_tcp_server;
178
179     /* Now listen for any client connections for this server socket */
180     status = listen (sock_tcp_server, 5);
181     if (status < 0)
182     {
183         printf("Error on Server Socket %d Listen\n", sock_tcp_server);
184         return;
185     }
186     else
187         printf("Server socket %d listen complete\n", sock_tcp_server);
188
189     /* All set to accept client connections */
190
191     /* Loop to create and establish server connections. */
192     while(1)
193     {
194
195         printf("\n");
196
197         read_ready = master_list;
198
199         tx_thread_sleep(20); /* Allow some time to other threads too */
200
201         /* Let the underlying TCP stack determine the timeout. */
202         status = select(maxfd + 1, &read_ready, 0, 0, 0);
203
204         if ((status == 0xFFFFFFFF) || (status == 0))
205         {
206
207             printf("Error with select. Status 0x%x\n", status);
208
209             continue;
210         }
211
212         /* Check for a connection request. */
213         is_set = FD_ISSET(sock_tcp_server, &read_ready);
214
215         if(is_set)
216         {
217
218             clientlen = sizeof(ClientAddr);
219
220             sock = accept(sock_tcp_server,(struct sockaddr*)&ClientAddr,
                           &clientlen);
221
222             /* Add this new connection to our master list */
223             FD_SET(sock, &master_list);
224
225             if ( sock > maxfd)
226             {
227
228                 maxfd = sock;
229             }
230
231             continue;
232         }
233
234         /* Check the set of 'ready' sockets, e.g connected to remote host and
235            waiting for notice of packets received. */
236         for (i = NX_BSD_SOCKFD_START; i < (maxfd+1+NX_BSD_SOCKFD_START); i++)
237         {
238

```



```

239         if ((i != sock_tcp_server) &&
240             (FD_ISSET(i, &master_list)) &&
241             (FD_ISSET(i, &read_ready)))
242         {
243             while(1)
244             {
245                 status = recv(i, (VOID *)Server_Rcv_Buffer, 100, 0);
246                 if (status == 0)
247                 {
248                     printf("(Server received no data from client on
249                        socket %d)\n", i);
250                     break;
251                 }
252                 else if (status == NX_SOC_ERROR)
253                 {
254                     printf("Error on Server receiving data from client on
255                        socket %d\n", i);
256                     break;
257                 }
258                 printf("Server socket %d received %d bytes: %s\n ",
259                    i, strlen(Server_Rcv_Buffer), Server_Rcv_Buffer);
260
261                 status = send(i, "Hello\n", strlen("Hello\n")+1, 0);
262                 if (status == NX_SOC_ERROR)
263                 {
264                     printf("Error on Server socket %d sending data to
265                        client\n", i);
266                 }
267                 else
268                 {
269                     printf("Server socket %d message sent to Client:
270                        Hello\n", i);
271                 }
272             }
273             /* Close this socket */
274             status = soc_close(i);
275             if (status != NX_SOC_ERROR)
276             {
277                 printf("Server socket %d closed \n", i);
278             }
279             else
280             {
281                 printf("Error on closing Server socket %d \n", i);
282             }
283         }
284     }
285     /* Loop back to check any next client connection */
286 }
287
288 CHAR        Client_Rcv_Buffer[100];
289
290 VOID thread_client_entry(ULONG thread_input)
291 {
292     INT        status;
293     INT        sock_tcp_client, length;
294     struct      sockaddr_in echoServAddr;
295     struct      sockaddr_in localAddr;
296
297     /* Let the server side get set up. */
298     tx_thread_sleep(200);
299
300     /* Set local port for displaying IP address and port. */
301     memset(&localAddr, 0, sizeof(localAddr));
302     localAddr.sin_family = AF_INET;
303     localAddr.sin_addr.s_addr = htonl(IP_ADDRESS(1,2,3,4));
304     localAddr.sin_port = htons(CLIENT_PORT);
305
306     /* Set server port and IP address which we need to connect. */
307     memset(&echoServAddr, 0, sizeof(echoServAddr));

```

```

316 echoServAddr.sin_family = AF_INET;
317 echoServAddr.sin_addr.s_addr = htonl(IP_ADDRESS(1,2,3,4));
318 echoServAddr.sin_port = htons(SERVER_PORT);
319
320 /* Now make client connections with the server. */
321 while (1)
322 {
323
324     printf("\n");
325     /* Create BSD TCP Socket */
326     sock_tcp_client = socket( AF_INET, SOCK_STREAM, 0);
327
328     if (sock_tcp_client == -1)
329     {
330         printf("Error on Client socket %d create \n", sock_tcp_client);
331         return;
332     }
333
334     printf("Client socket %d created\n", sock_tcp_client);
335
336     /* Now connect this client to the server */
337     status = connect(sock_tcp_client, (struct sockaddr *)&echoServAddr,
338                     sizeof(echoServAddr));
339
340     /* Check for error. */
341     if (status != OK)
342     {
343         printf("Error on Client socket %d connect\n", sock_tcp_client);
344         soc_close(sock_tcp_client);
345         return;
346     }
347     /* Get and print source and destination information */
348     printf("Client socket %d connected \n", sock_tcp_client);
349
350     status = getsockname(sock_tcp_client, (struct sockaddr *)&localAddr,
351                          &length);
352     printf("Client port = %lu , Client = 0x%x",
353            htonl(localAddr.sin_port), htonl(localAddr.sin_addr.s_addr));
354     length = sizeof(struct sockaddr_in);
355     status = getpeername( sock_tcp_client, (struct sockaddr *)
356                          &echoServAddr, &length);
357     printf("Remote port = %lu, Remote IP = 0x%x \n",
358            htonl(echoServAddr.sin_port),
359            htonl(echoServAddr.sin_addr.s_addr));
360
361     /* Now receive the echoed packet from the server */
362     while(1)
363     {
364         printf("Client sock %d sending packet to server\n",
365                sock_tcp_client);
366
367         status = send(sock_tcp_client, "Hello", (strlen("Hello")+1), 0);
368
369         if (status == ERROR)
370         {
371             printf("Error: Client Socket (%d) send \n", sock_tcp_client);
372         }
373         else
374         {
375             printf("Client socket %d sent message Hello\n",
376                    sock_tcp_client);
377         }
378
379         status = recv(sock_tcp_client, (VOID *)Client_Rcv_Buffer,100,0);
380
381         if (status <= 0)
382         {
383             if (status < 0)
384             {
385                 printf("Error on Client receiving on socket %d \n",
386                        sock_tcp_client);
387             }
388             else
389             {
390                 printf("Nothing received by client on socket %d\n",
391                        sock_tcp_client);
392             }
393         }
394     }
395 }

```

```

387         break;
388     }
389     else
390     {
391         printf("Client socket %d received %d bytes: %s\n",
                sock_tcp_client,
                strlen(Client_Rcv_Buffer), Client_Rcv_Buffer);
392     }
393 }
394
395 }
396
397 /* close this client socket */
398 status = soc_close(sock_tcp_client);
399
400 if (status != ERROR)
401 {
402     printf("Client Socket %d closed\n", sock_tcp_client);
403 }
404 else
405 {
406     printf("Error on Client Socket %d on close \n", sock_tcp_client);
407 }
408
409 /* Make another Client connection...*/
410
411 }
412 }
413

```

Small IPv6 Example System

An example of how to use NetX Duo BSD services for IPv6 networks is described in the program below. This example is very similar to the IPv4 demo program previously described with a few important differences. The client and server threads, BSD packet pool, IP instance and BSD initialization happens as it does for IPv4 BSD sockets.

In the server thread entry function, *thread_server_entry*, defines a couple IPv6 variables using *sockaddr_in6* and *NXD_ADDRESS* data types on lines 145-148. The *NXD_ADDRESS* data type can actually store both IPv4 and IPv6 address types.

Next, the server thread enables IPv6 and ICMPv6 on the IP instance using the *nxd_ipv6_enable* and *nxd_icmpv6_enable* service respectively on line 161 and 169. Next, the link local and global IP addresses are registered with the IP instance. This is done using the *nxd_ipv6_address_set* service on lines 180 and 195. It then sleeps long enough for the IP thread task to complete the Duplicate Address Detection protocol and register these addresses as valid addresses on the *tx_thread_sleep* call on line 201.

Next, the TCP server socket is created with the *AF_INET6* socket type input argument on line 204. The socket IPv6 address and port are set on lines 216-221, again noting the use of *htonl* and *htons* macros to put data in network byte order for BSD socket services. From here on, the server thread entry function is virtually identical to the IPv4 example.

The client thread entry function, *thread_client_entry*, is defined next. Note that because the TCP client in this example shares the same IP instance and IPv6 address as the TCP server, we do not need to enable IPv6 or ICMPv6 services on the IP instance again. Further, the IPv6 address is also already registered with the IP instance. Instead, the client thread entry function simply waits on line 368 for the server to set up. The server address and port are set, using the host to network byte order macros on lines 387-392, and then the Client can connect with the TCP server on line 412. Note that the local IP address data types in lines 378-383 are used only to demonstrate the *getsockname* and *getpeername* services on lines 425 and 434 respectively. Because the data is coming from the network, the network to host byte order macros as used in lines 378-383.

Next the client thread entry function enters a loop in which it creates a TCP socket, makes a TCP connection and sends and receives data with the TCP server until no more data is received virtually the same as the IPv4 example. It then closes the socket on line 483, pauses briefly and creates another TCP socket and requests a TCP server connection.

One important difference with the IPv4 example is the *socket* calls specify an IPv6 socket using the AF_INET6 input argument. Another important difference is that the TCP Client *connect* call takes an *sockaddr_in6* data type and a length argument set to the size of the *sockaddr_in6* data type.

```

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

```

```

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

```

```

147 struct      sockaddr_in6 serverAddr;
148 struct      sockaddr_in6 ClientAddr;
149 UINT        iface_index, address_index;
150
151
152     status = nx_ip_status_check(&bsd_ip, NX_IP_INITIALIZE_DONE,
153                                &actual_status, 100);
154
155     /* Check status... */
156     if (status != NX_SUCCESS)
157     {
158         return;
159     }
160
161     /* Enable IPV6 */
162     status = nxd_ipv6_enable(&bsd_ip);
163     if((status != NX_SUCCESS) && (status != NX_ALREADY_ENABLED))
164     {
165         printf("Error with IPV6 enable 0x%x\n", status);
166         return;
167     }
168
169     /* Enable ICMPv6 */
170     status = nxd_icmp_enable(&bsd_ip);
171     if(status)
172     {
173         printf("Error with ICMPv6 enable 0x%x\n", status);
174         return;
175     }
176
177     /* Set the primary interface for our DNS IPV6 addresses. */
178     iface_index = 0;
179
180     /* This assumes we are using the primary network interface (index 0). */
181     status = nxd_ipv6_address_set(&bsd_ip, iface_index, NX_NULL, 10,
182                                  &address_index);
183
184     if (status)
185     {
186         return;
187     }
188
189     /* Set ip_0 interface address. */
190     ip_address.nxd_ip_version = NX_IP_VERSION_V6;
191     ip_address.nxd_ip_address.v6[0] = htonl(0x20010db8);
192     ip_address.nxd_ip_address.v6[1] = htonl(0x0000f101);
193     ip_address.nxd_ip_address.v6[2] = 0;
194     ip_address.nxd_ip_address.v6[3] = htonl(0x101);
195
196     /* Set the host global IP address. We are assuming a 64
197        bit prefix here but this can be any value (< 128). */
198     status = nxd_ipv6_address_set(&bsd_ip, iface_index, &ip_address, 64,
199                                  &address_index);
200
201     if (status)
202     {
203         return;
204     }
205
206     /* Wait for IPV6 stack to finish DAD process. */
207     tx_thread_sleep(400);
208
209     /* Create BSD TCP Socket */
210     sock_tcp_server = socket(AF_INET6, SOCK_STREAM, 0);
211
212     if (sock_tcp_server == -1)
213     {
214         printf("\nError: BSD TCP Server socket create \n");
215         return;
216     }
217
218     printf("\nBSD TCP Server socket created %lu \n", sock_tcp_server);
219
220     /* Set the server port and IP address */
221     memset(&serverAddr, 0, sizeof(serverAddr));
222     serverAddr.sin6_addr._S6_un._S6_u32[0] = htonl(0x20010db8);
223     serverAddr.sin6_addr._S6_un._S6_u32[1] = htonl(0xf101);
224     serverAddr.sin6_addr._S6_un._S6_u32[2] = 0x0;
225     serverAddr.sin6_addr._S6_un._S6_u32[3] = htonl(0x0101);
226     serverAddr.sin6_port = htons(SERVER_PORT);
227     serverAddr.sin6_family = AF_INET6;
228
229     /* Bind this server socket */
230     status = bind(sock_tcp_server, (struct sockaddr *) &serverAddr,

```

```

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

```



```

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

```

```

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

```

```

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

```

Chapter 3

NetX Duo BSD Services

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

```

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

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

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

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

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

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

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

in_addr_t inet_addr(const CHAR *buffer);

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

CHAR  inet_ntoa(struct in_addr address_to_convert);

const CHAR *inet_ntop(INT af, const VOID *src, CHAR *dst, socklen_t size);

INT  inet_pton(INT af, const CHAR *src, VOID *dst);

INT  listen(INT sockID, INT backlog);

INT  recvfrom(INT sockID, CHAR *buffer, INT buffersize, INT flags,
              struct sockaddr *fromAddr, INT *fromAddrLen);

INT  recv(INT sockID, VOID *rcvBuffer, INT bufferLength, INT flags);

INT  sendto(INT sockID, CHAR *msg, INT msgLength, INT flags,
            struct sockaddr *destAddr, INT destAddrLen);

INT  send(INT sockID, const CHAR *msg, INT msgLength, INT flags);

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

```

```

INT  soc_close ( INT sockID);

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

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

INT  getsockopt(INT sockID, INT option_level, INT option_name, VOID *option_value,
                INT *option_length);

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

INT  getaddrinfo(const CHAR *node, const CHAR *service, const struct addrinfo *hints,
                struct addrinfo **res);

VOID freeaddrinfo(struct addrinfo *res);

INT  getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host,
                size_t hostlen, char *serv, size_t servlen, int flags);

VOID nx_bsd_set_service_list(struct NX_BSD_SERVICE_LIST *serv_list_ptr,
                ULONG serv_list_len);

VOID FD_SET(INT fd, fd_set *fdset);

VOID FD_CLR(INT fd, fd_set *fdset);

INT  FD_ISSET(INT fd, fd_set *fdset);

VOID FD_ZERO (fd_set *fdset);

```

NetX Duo BSD 4.3 Socket API Wrapper for NetX Duo
User Guide

Publication Date: Rev.5.12 Nov 7, 2016

Published by: Renesas Electronics Corporation



SALES OFFICES

Renesas Electronics Corporation

<http://www.renesas.com>

Refer to "<http://www.renesas.com/>" for the latest and detailed information.

Renesas Electronics Corporation

TOYOSU FORESIA, 3-2-24 Toyosu, Koto-ku, Tokyo 135-0061, Japan

Renesas Electronics America Inc.

1001 Murphy Ranch Road, Milpitas, CA 95035, U.S.A.

Tel: +1-408-432-8888, Fax: +1-408-434-5351

Renesas Electronics Canada Limited

9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3

Tel: +1-905-237-2004

Renesas Electronics Europe Limited

Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K

Tel: +44-1628-651-700

Renesas Electronics Europe GmbH

Arcadiastrasse 10, 40472 Düsseldorf, Germany

Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.

Room 1709 Quantum Plaza, No.27 ZhichunLu, Haidian District, Beijing, 100191 P. R. China

Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.

Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, 200333 P. R. China

Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited

Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong

Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.

13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan

Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.

80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949

Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.

Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia

Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.

No.777C, 100 Feet Road, HAL 2nd Stage, Indiranagar, Bangalore 560 038, India

Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.

17F, KAMCO Yangjae Tower, 262, Gangnam-daero, Gangnam-gu, Seoul, 06265 Korea

Tel: +82-2-558-3737, Fax: +82-2-558-5338

NetX Duo BSD 4.3 Socket API Wrapper for NetX Duo User Guide



Renesas Electronics Corporation

R11UM0030EU0512