NETX Duo

the high-performance real-time implementation
of TCP/IP standards

User Guide

Version 5

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259
http://www.expresslogic.com

Part Number: 000-1015
Revision 5.6

Contents

Figures

Express Logic, Inc.

About This Guide

This guide contains comprehensive information about NetX Duo, the high-performance IPv4/IPv6 dual network stack from Express Logic, Inc.

It is intended for embedded real-time software developers familiar with basic networking concepts, the ThreadX RTOS, and the C programming language.

Organization

Chapter 1	Introduces NetX Duo
Chapter 2	Gives the basic steps to install and use NetX Duo with your ThreadX application.
Chapter 3	Provides a functional overview of the NetX Duo system and basic information about the TCP/IP networking standards.
Chapter 4	Details the application's interface to NetX Duo.
Chapter 5	Describes network drivers for NetX Duo.
Appendix A	NetX Duo Services
Appendix B	NetX Duo Constants
Appendix C	NetX Duo Data Types
Appendix D	BSD-Compatible Socket API

Guide Conventions

Italics Typeface denotes book titles, emphasizes important words, and indicates variables.

Boldface Typeface denotes file names, key words, and further emphasizes important words and variables.

Information symbols draw attention to important or additional information that could affect performance or function.

Warning symbols draw attention to situations that developers should avoid because they could cause fatal errors.

NetX Duo Data Types

In addition to the custom NetX Duo control structure data types, there are several special data types that are used in NetX Duo service call interfaces. These special data types map directly to data types of the underlying C compiler. This is done to ensure portability between different C compilers. The exact implementation is inherited from ThreadX and can be found in the ***tx_port.h*** file included in the ThreadX distribution.

The following is a list of NetX Duo service call data types and their associated meanings:

UINT	Basic unsigned integer. This type must support 32-bit unsigned data; however, it is mapped to the most convenient unsigned data type.
ULONG	Unsigned long type. This type must support 32-bit unsigned data.
VOID	Almost always equivalent to the compiler's void type.
CHAR	Most often a standard 8-bit character type.

Additional data types are used within the NetX Duo source. They are located in either the ***tx_port.h*** or ***nx_port.h*** files.

Customer Support Center

Support engineers	858.613.6640
Support fax	858.521.4259
Support email	support@expresslogic.com
Web page	http://www.expresslogic.com

Latest Product Information

Visit the Express Logic web site and select the "Support" menu option to find the latest online support information, including information about the latest NetX product releases.

What We Need From You

To more efficiently resolve your support request, provide us with the following information in your email request:

1. A detailed description of the problem, including frequency of occurrence and whether it can be reliably reproduced.
2. A detailed description of any changes to the application and/or NetX Duo that preceded the problem.
3. The contents of the ***_tx_version_id*** and ***_nx_version_id*** strings found in the ***tx_port.h*** and ***nx_port.h*** files of your distribution. These strings will provide us valuable information regarding your run-time environment.
4. The contents in RAM of the following ULONG variables:

   _tx_build_options
   _nx_system_build_options1
   _nx_system_build_options2
   _nx_system_build_options3

_nx_system_build_options4
_nx_system_build_options5

These variables will give us information on how your ThreadX and NetX Duo libraries were built.

5. A trace buffer captured immediately after the problem was detected. This is accomplished by building the ThreadX and NetX Duo libraries with **TX_ENABLE_EVENT_TRACE** and calling ***tx_trace_enable*** with the trace buffer information. Refer to the *TraceX User Guide* for details.

Where to Send Comments About This Guide

The staff at Express Logic is always striving to provide you with better products. To help us achieve this goal, email any comments and suggestions to the Customer Support Center at

support@expresslogic.com

Please enter “NetX Duo User Guide” in the subject line.

CHAPTER 1

Introduction to NetX Duo

NetX Duo is a high-performance real-time implementation of the TCP/IP standards designed exclusively for embedded ThreadX-based applications. This chapter contains an introduction to NetX Duo and a description of its applications and benefits.

NetX Duo Unique Features

Unlike other TCP/IP implementations, NetX Duo is designed to be versatile—easily scaling from small micro-controller-based applications to those that use powerful RISC and DSP processors. This is in sharp contrast to public domain or other commercial implementations originally intended for workstation environments but then squeezed into embedded designs.

Piconet™ Architecture

Underlying the superior scalability and performance of NetX Duo is *Piconet*, a software architecture especially designed for embedded systems. Piconet architecture maximizes scalability by implementing NetX Duo services as a C library. In this way, only those services actually used by the application are brought into the final runtime image. Hence, the actual size of NetX Duo is completely determined by the application. For most applications, the instruction image requirements of NetX Duo ranges between 5 KBytes and 30 KBytes in size. With IPv6 and ICMPv6 enabled for IPv6 address configuration and neighbor discovery protocols, NetX Duo ranges in size from 30k to 45k.

NetX Duo achieves superior network performance by layering internal component function calls only when it is absolutely necessary. In addition, much of NetX Duo processing is done directly in-line, resulting in outstanding performance advantages over the workstation network software used in embedded designs in the past.

Zero-copy Implementation

NetX Duo provides a packet-based, zero-copy implementation of TCP/IP. Zero copy means that data in the application's packet buffer are never copied inside NetX Duo. This greatly improves performance and frees up valuable processor cycles to the application, which is extremely important in embedded applications.

UDP Fast Path™ Technology

With *UDP Fast Path Technology*, NetX Duo provides the fastest possible UDP processing. On the sending side, UDP processing—including the optional UDP checksum—is completely contained within the ***nx_udp_socket_send*** service. No additional function calls are made until the packet is ready to be sent via the internal NetX Duo IP send routine. This routine is also flat (i.e., its function call nesting is minimal) so the packet is quickly dispatched to the application's network driver. When the UDP packet is received, the NetX Duo packet-receive processing places the packet directly on the appropriate UDP socket's receive queue or gives it to the first thread suspended waiting for a receive packet from the UDP socket's receive queue. No additional ThreadX context switches are necessary.

ANSI C Source Code

NetX Duo is written completely in ANSI C and is portable immediately to virtually any processor architecture that has an ANSI C compiler and ThreadX support.

Not A Black Box

Most distributions of NetX Duo include the complete C source code. This eliminates the "black-box" problems that occur with many commercial network stacks. By using NetX Duo, applications developers can see exactly what the network stack is doing—there are no mysteries!

Having the source code also allows for application specific modifications. Although not recommended, it is certainly beneficial to have the ability to modify the network stack if it is required.

These features are especially comforting to developers accustomed to working with in-house or public domain network stacks. They expect to have source code and the ability to modify it. NetX Duo is the ultimate network software for such developers.

BSD-Compatible Socket API

For legacy applications, NetX Duo also provides a BSD-compatible socket interface that makes calls to the high-performance NetX Duo API underneath. This helps in migrating existing network application code to NetX Duo.

RFCs Supported by NetX Duo

NetX Duo support of RFCs describing basic network protocols includes but is not limited to the following network protocols. NetX Duo follows all general recommendations and basic requirements within the constraints of a real-time operating system with small memory footprint and efficient execution.

RFC	Description
RFC 1112	Host Extensions for IP Multicasting (IGMPv1)
RFC 2236	Internet Group Management Protocol, Version 2
RFC 768	User Datagram Protocol (UDP)
RFC 791	Internet Protocol (IP)
RFC 792	Internet Control Message Protocol (ICMP)
RFC 793	Transmission Control Protocol (TCP)

RFC	Description
RFC 826	Ethernet Address Resolution Protocol (ARP)
RFC 903	Reverse Address Resolution Protocol (RARP)

Below are the IPv6-related RFCs supported by NetX Duo.

RFC	Description
RFC 1981	Path MTU Discovery for Internet Protocol v6 (IPv6)
RFC 2460	Internet Protocol v6 (IPv6) Specification
RFC 2464	Transmission of IPv6 Packets over Ethernet Networks
RFC 2581	TCP Congestion Control
RFC 4291	Internet Protocol v6 (IPv6) Addressing Architecture
RFC 4443	Internet Control Message Protocol (ICMPv6) for Internet Protocol v6 (IPv6) Specification
RFC 4861	Neighbor Discovery for IP v6
RFC 4862	IPv6 Stateless Address Auto Configuration

Embedded Network Applications

Embedded network applications are applications that need network access and execute on microprocessors hidden inside products such as cellular phones, communication equipment, automotive engines, laser printers, medical devices, and so forth. Such applications almost always have some memory and performance constraints. Another distinction of embedded network applications is that their software and hardware have a dedicated purpose.

Real-time Network Software

Basically, network software that must perform its processing within an exact period of time is called *real-time network* software, and when time constraints are imposed on network applications, they are classified as real-time applications. Embedded network applications are almost always real-time because of their inherent interaction with the external world.

NetX Duo Benefits

The primary benefits of using NetX Duo for embedded applications are high-speed Internet connectivity and very small memory requirements. NetX Duo is also completely integrated with the high-performance, multitasking ThreadX real-time operating system.

Improved Responsiveness

The high-performance NetX Duo protocol stack enables embedded network applications to respond faster than ever before. This is especially important for embedded applications that either have a significant volume of network traffic or stringent processing requirements on a single packet.

Software Maintenance

Using NetX Duo allows developers to easily partition the network aspects of their embedded application. This partitioning makes the entire development process easy and significantly enhances future software maintenance.

Increased Throughput

NetX Duo provides the highest-performance networking available, which directly transfers to the embedded application. NetX Duo applications are able to process many more packets than non-NetX Duo applications!

Processor Isolation

NetX Duo provides a robust, processor-independent interface between the application and the underlying processor and network hardware. This allows developers to concentrate on the network aspects of the application rather than spending extra time dealing with hardware issues directly affecting networking.

Ease of Use

NetX Duo is designed with the application developer in mind. The NetX Duo architecture and service call interface are easy to understand. As a result, NetX Duo developers can quickly use its advanced features.

Improve Time to Market

The powerful features of NetX Duo accelerate the software development process. NetX Duo abstracts most processor and network hardware issues, thereby removing these concerns from a majority of application network-specific areas. This, coupled with the ease-of-use and advanced feature set, result in a faster time to market!

Protecting the Software Investment

NetX Duo is written exclusively in ANSI C and is fully integrated with the ThreadX real-time operating system. This means NetX Duo applications are instantly portable to all ThreadX supported processors. Better still, a completely new processor architecture can be supported with ThreadX in a matter of weeks. As a result, using NetX Duo ensures the application’s migration path and protects the original development investment.

IPv6 Ready Logo Certification

NetX Duo “IPv6 Ready” certification was obtained through the “IPv6 Core Protocol (Phase 2) Self Test” package available from the IPv6 Ready Organization. Refer to the following IPv6-Ready project websites for more information on the test platform and test cases:

`http://www.ipv6ready.org/about_phase2_test.html`

`http://www.tahi.org/logo/phase2-core/`

The Phase 2 IPv6 Core Protocol Self Test Suite validates that an IPv6 stack observes the requirements set forth in the following RFCs with extensive testing:

Section 1: RFC 2460
Section 2: RFC 4861
Section 3: RFC 4862
Section 4: RFC 1981
Section 5: RFC 4443

Installation and Use of NetX Duo

This chapter contains a description of various issues related to installation, setup, and use of the high-performance network stack NetX Duo, including the following:

Host Considerations

Embedded development is usually performed on Windows or Linux (Unix) host computers. After the application is compiled, linked, and the executable is generated on the host, it is downloaded to the target hardware for execution.

Usually the target download is done from within the development tool's debugger. After download, the debugger is responsible for providing target execution control (go, halt, breakpoint, etc.) as well as access to memory and processor registers.

Most development tool debuggers communicate with the target hardware via on-chip debug (OCD) connections such as JTAG (IEEE 1149.1) and Background Debug Mode (BDM). Debuggers also communicate with target hardware through In-Circuit Emulation (ICE) connections. Both OCD and ICE connections provide robust solutions with minimal intrusion on the target resident software.

As for resources used on the host, the source code for NetX Duo is delivered in ASCII format and requires approximately 1 Mbytes of space on the host computer's hard disk.

*Review the supplied **readme_netx_duo_generic.txt** file for additional host system considerations and options.*

Target Considerations

NetX Duo requires between 5 KBytes and 45 KBytes of Read-Only Memory (ROM) on the target. Another 1 to 5KBytes of the target's Random Access Memory

(RAM) are required for the NetX Duo thread stack and other global data structures.

In addition, NetX Duo requires the use of a ThreadX timer and a ThreadX mutex object. These facilities are used for periodic processing needs and thread protection inside the NetX Duo protocol stack.

Product Distribution

Two types of NetX Duo packages are available—*standard* and *premium*. The *standard* package includes minimal source code, while the *premium* package contains complete NetX Duo source code. Either package is shipped on a single CD.

The exact contents of the distribution CD depends on the target processor, development tools, and the NetX Duo package purchased. Following is a list of the important files common to most product distributions:

readme_netx_duo_generic.txt
This file contains specific information about the NetX Duo release.

nx_api.h This C header file contains all system equates, data structures, and service prototypes.

nx_port.h This C header file contains all target-specific and development tool-specific data definitions and structures.

demo_netx_duo_tcp.c
demo_netx_duo_udp.c
These C files contain small TCP and UDP applications.

nx.a (or nx.lib)

This is the binary version of the NetX Duo C library. It is distributed with the *standard* package.

All files are in lower-case, making it easy to convert the commands to Linux (Unix) development platforms.

NetX Duo Installation

Installation of NetX Duo is straightforward. The following instructions apply to virtually any installation. However, examine the ***readme_netx_duo.txt*** file for changes specific to the actual development tool environment.

Step 1: Backup the NetX Duo distribution disk and store it in a safe location.

Step 2: On the host hard drive, copy all the files of the NetX Duo distribution into the previously created and installed ThreadX directory.

Step 3: If installing the *standard* package, NetX Duo installation is now complete. Otherwise, if installing the premium package, you must build the NetX Duo runtime library.

*Application software needs access to the NetX Duo library file, usually called **nx.a** (or **nx.lib**), and the C include files **nx_api.h** and **nx_port.h**. This is accomplished either by setting the appropriate path for the development tools or by copying these files into the application development area.*

Using NetX Duo

Using NetX Duo is easy. Basically, the application code must include ***nx_api.h*** during compilation and link with the NetX Duo library ***nx.a*** (or ***nx.lib***).

There are four easy steps required to build a NetX Duo application:

Step 1: Include the ***nx_api.h*** file in all application files that use NetX Duo services or data structures.

Step 2: Initialize the NetX Duo system by calling ***nx_system_initialize*** from the ***tx_application_define*** function or an application thread.

Step 3: Create an IP instance, enable the Address Resolution Protocol (ARP), if necessary, and any sockets after ***nx_system_initialize*** is called.

Step 4: Compile application source and link with the NetX Duo runtime library ***nx.a*** (or ***nx.lib***). The resulting image can be downloaded to the target and executed!

Troubleshooting

Each NetX Duo port is delivered with a demonstration application that executes with a simulated network driver. This same demonstration is delivered with all versions of NetX Duo and provides the ability to run NetX Duo without any network hardware. It is always a good idea to get the demonstration system running first.

*See the **readme_netx_duo_generic.txt** file supplied with the distribution for more specific details regarding the demonstration system.*

If the demonstration system does not run properly, perform the following operations to narrow the problem:

1. Determine how much of the demonstration is running.
2. Increase stack sizes in any new application threads.
3. Recompile the NetX Duo library with the appropriate debug options listed in the configuration option section.
4. Examine the NX_IP structure to see if packets are being sent or received.
5. Examine the default packet pool to see if there are available packets.
6. Ensure network driver is supplying ARP and IP packets with their headers on 4-byte boundaries for applications requiring IPv4 or IPv6 connectivity.
7. Temporarily bypass any recent changes to see if the problem disappears or changes. Such information should prove useful to Express Logic support engineers.

Follow the procedures outlined in the "What We Need From You" on page 12 to send the information gathered from the troubleshooting steps.

Configuration Options

There are several configuration options when building the NetX Duo library and the application using NetX Duo. The options below can be defined in the application source, on the command line, or within the ***nx_user.h*** include file unless otherwise specified.

*Options defined in **nx_user.h** are applied only if the application and NetX Duo library are built with **NX_INCLUDE_USER_DEFINE_FILE** defined.*

Review the ***readme_netx_duo_generic.txt*** file for additional options for your specific version of NetX Duo. The following sections describe configuration options available in NetX Duo. General options applicable to both IPv4 and IPv6 are listed first, followed by IPv6 specific options.

System Configuration Options

Define	**Meaning**
NX_DEBUG	Defined, this option enables the optional print debug information available from the RAM Ethernet network driver.
NX_DEBUG_PACKET	Defined, this option enables the optional debug packet dumping available in the RAM Ethernet network driver.
NX_DISABLE_ERROR_CHECKING	Defined, this option removes the basic NetX Duo error checking API and results in a 15-percent performance improvement. API return codes not affected by disabling error checking are listed in bold typeface in the API definition. This define is typically used after the application is debugged sufficiently and its use improves performance and decreases code size.

Define	Meaning
NX_DRIVER_DEFERRED_PROCESSING	Defined, this option enables deferred network driver packet handling. This allows the network driver to place a packet on the IP instance and have the network's real processing routine called from the NetX Duo internal IP helper thread.
NX_LITTLE_ENDIAN	Defined, this option performs the necessary byte swapping on little endian environments to ensure the protocol headers are in proper big endian format. Note that the default is typically setup in ***nx_port.h***.
NX_MAX_PHYSICAL_INTERFACES	Specifies the total number of physical network interfaces on the device. The default value is 1 and is defined in ***nx_api.h***; a device must have at least one physical interface. Note this does not include the loopback interface.
NX_PHYSICAL_HEADER	Specifies the size in bytes of the physical packet header. The default value is 16 (based on a typical 14-byte Ethernet frame aligned to 32-bit boundary) and is defined in ***nx_api.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_PHYSICAL_TRAILER	Specifies the size in bytes of the physical packet trailer and is typically used to reserve storage for things like Ethernet CRCs, etc. The default value is 4 and is defined in ***nx_api.h***.

ARP Configuration Options

Define	Meaning
NX_DISABLE_ARP_INFO	Defined, this option disables ARP information gathering.
NX_ARP_DISABLE_AUTO_ARP_ENTRY	Defined, this option disables entering ARP request information in the ARP cache.
NX_ARP_EXPIRATION_RATE	This define specifies the number of seconds ARP entries remain valid. The default value of zero disables expiration or aging of ARP entries and is defined in ***nx_api.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_ARP_MAX_QUEUE_DEPTH	This define specifies the maximum number of packets that can be queued while waiting for an ARP response. The default value is 4 and is defined in ***nx_api.h***.
NX_ARP_MAXIMUM_RETRIES	This define specifies the maximum number of ARP retries made without an ARP response. The default value is 18 and is defined in ***nx_api.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_ARP_UPDATE_RATE	This define specifies the number of seconds between ARP retries. The default value is 10, which represents 10 seconds, and is defined in ***nx_api.h***. The application can override the default by defining the value before ***nx_api.h*** is included.

ICMP Configuration Options

Define	Meaning
NX_DISABLE_ICMP_INFO	Defined, this option disables ICMP information gathering.
NX_DISABLE_ICMP_RX_CHECKSUM	Defined, this option disables ICMP checksum computation on received ICMP packets
NX_DISABLE_ICMP_TX_CHECKSUM	Defined, this option disables ICMP checksum computation on transmitted ICMP packets

IGMP Configuration Options

Define	Meaning
NX_DISABLE_IGMP_INFO	Defined, this option disables IGMP information gathering.
NX_DISABLE_IGMPV2	Defined, IGMP v2 support is disabled. By default this option is not set, and is defined in ***nx_api.h***.
NX_MAX_MULTICAST_GROUPS	This define specifies the maximum number of multicast groups that can be joined. The default value is 7 and is defined in ***nx_api.h***. The application can override the default by defining the value before ***nx_api.h*** is included.

IP Configuration Options

Define	Meaning
NX_DISABLE_FRAGMENTATION	This define disables IP fragmentation logic.
NX_DISABLE_IP_INFO	Defined, this option disables IP information gathering.
NX_DISABLE_IP_RX_CHECKSUM	Defined, this option disables checksum logic on received IP packets. This is useful if the link-layer has reliable checksum or CRC logic.
NX_DISABLE_IP_TX_CHECKSUM	Defined, this option disables checksum logic on IP packets sent. This is only useful in situations in which the receiving network node has received IP checksum logic disabled, or the underlying network device is capable of generating IP header checksum.
NX_DISABLE_LOOPBACK_INTERFACE	Defined, this option disables NetX Duo support for the loopback interface.
NX_DISABLE_RX_SIZE_CHECKING	Defined, this option disables the addition size checking on received packets.
NX_ENABLE_IP_STATIC_ROUTING	Defined, this enables static routing in which a destination address can be assigned a specific next hop address. By default static routing is disabled.
NX_IP_PERIODIC_RATE	This define specifies the number of ThreadX timer ticks in one second. The default value is 100 (based on a 10ms ThreadX timer interrupt) and is defined in ***nx_port.h***. The application can override the default by defining the value before ***nx_api.h*** is included.

NX_IP_ROUTING_TABLE_SIZE	This define sets the maximum number of entries in the routing table, which is a list of an outgoing interface and the next hop addresses for a given destination address. The default value is 8 and is defined in ***nx_api.h.***
NX_MAX_IP_INTERFACES	This define sets the total number of logical network interfaces on the target. The default value, which is defined in ***nx_api.h***, depends if the loopback interface is enabled. If so, the value is the number of physical interface NX_MAX_PHYSICAL_INTERFACES (which is defaulted to 1) plus the loopback interface. Otherwise the value is number of physical interfaces.

Packet Configuration Options

Define	**Meaning**
NX_DISABLE_PACKET_INFO	Defined, this option disables packet pool information gathering.
NX_PACKET_HEADER_PAD	Defined, this option allows padding towards the end of the NX_PACKET control block.
NX_PACKET_HEADER_PAD_SIZE	This option sets the number of ULONG words to be padded to the NX_PACKET structure, allows the packet payload area to start at desired alignment.

RARP Configuration Options

Define	Meaning
NX_DISABLE_RARP_INFO	Defined, this option disables RARP information gathering.

TCP Configuration Options

Define	Meaning
NX_DISABLE_RESET_DISCONNECT	Defined, this option disables the reset processing during disconnect when the timeout value supplied is specified as NX_NO_WAIT.
NX_DISABLE_TCP_INFO	Defined, this option disables TCP information gathering.
NX_DISABLE_TCP_RX_CHECKSUM	Defined, this option disables checksum logic on received TCP packets. This is only useful in situations in which the link-layer has reliable checksum or CRC processing.
NX_DISABLE_TCP_TX_CHECKSUM	Defined, this option disables checksum logic for sending TCP packets. This is only useful in situations in which the receiving network node has received TCP checksum logic disabled or the underlying network driver is capable of generating TCP checksum.

NX_MAX_LISTEN_REQUESTS	This option specifies the maximum number of server listen requests. The default value is 10 and is defined in ***nx_api.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_ACK_EVERY_N_PACKETS	This option specifies the number of TCP packets to receive before sending an ACK. The default value is 2 where an ACK packet is sent for every 2 packets received. Note if NX_TCP_IMMEDIATE_ACK is enabled but NX_TCP_ACK_EVERY_N_PACKETS is not, this value is automatically set to 1 for backward compatibility.
NX_TCP_ACK_TIMER_RATE	This option specifies how the number of system ticks (NX_IP_PERIODIC_RATE) is divided to calculate the timer rate for the TCP delayed ACK processing. The default value is 5, which represents 200ms, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_ENABLE_KEEPALIVE	Defined, this option enables the optional TCP keepalive timer.
NX_TCP_ENABLE_WINDOW_SCALING	This option enables the window scaling option for TCP applications. If defined, window scaling option is negotiated during TCP connection phase, and the application is able to specify a window size larger than 64K. The default setting is not enabled (not defined).

NX_TCP_FAST_TIMER_RATE	This option specifies how the number of system ticks (NX_IP_PERIODIC_RATE) is divided to calculate the fast TCP timer rate. The fast TCP timer is used to drive the various TCP timers, including the delayed ACK timer. The default value is 10, which represents 100ms, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_IMMEDIATE_ACK	Defined, this option enables the optional TCP immediate ACK response processing. Defining this symbol is equivalent to defining NX_TCP_ACK_EVERY_N_PACKETS to be 1
NX_TCP_KEEPALIVE_INITIAL	This option specifies the number of seconds of inactivity before the keepalive timer activates. The default value is 7200, which represents 2 hours, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_KEEPALIVE_RETRY	This option specifies the number of seconds between retries of the keepalive timer assuming the other side of the connection is not responding. The default value is 75, which represents 75 seconds between retries, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_KEEPALIVE_RETRIES	This option specifies how many keepalive retries are allowed before the connection is deemed broken. The default value is 10, which represents 10 retries, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.

NX_TCP_MAXIMUM_RETRIES	This option specifies how many transmit retries are allowed before the connection is deemed broken. The default value is 10, which represents 10 retries, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_MAXIMUM_TX_QUEUE	This option specifies the maximum depth of the TCP transmit queue before TCP send requests are suspended or rejected. The default value is 20, which means that a maximum of 20 packets can be in the transmit queue at any given time. Note that packets stay in the transmit queue until an ACK is received from the other side of the connection. This constant is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_RETRY_SHIFT	This option specifies how the retransmit timeout period changes between retries. If this value is 0, the initial retransmit timeout is the same as subsequent retransmit timeouts. If this value is 1, each successive retransmit is twice as long. If this value is 2, each subsequent retransmit timeout is four times as long. The default value is 0 and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_TRANSMIT_TIMER_RATE	This option specifies how the number of system ticks (NX_IP_PERIODIC_RATE) is divided to calculate the timer rate for the TCP transmit retry processing. The default value is 1, which represents 1 second, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.

UDP Configuration Options

Define	Meaning
NX_DISABLE_UDP_INFO	Defined, this option disables UDP information gathering.
NX_DISABLE_UDP_RX_CHECKSUM	Defined, this option disables the UDP checksum computation on incoming UDP packets
NX_DISABLE_UDP_TX_CHECKSUM	Defined, this option disables the UDP checksum computation on outgoing UDP packets

IPv6 Options

Define	Meaning
NX_DISABLE_IPV6	This option disables IPv6 functionality when the NetX Duo library is built. For applications that do not need IPv6, this avoids pulling in code and additional storage space needed to support IPv6.
NX_MAX_IPV6_ADDRESSES	This option specifies the number of entries in the IPv6 address pool. During interface configuration, NetX Duo uses IPv6 entries from the pool. It is defaulted to (NX_MAX_PHYSICAL_INTERFACES * 3) to allow each interface to have at least one link local address and two global addresses. Note that all interfaces share the IPv6 address pool.

NXDUO_DESTINATION_TABLE_SIZE	This option specifies the number of entries in the IPv6 destination table. This stores information about next hop addresses for IPv6 addresses. Defined in ***nx_api.h***, the default value is 8.
NX_IPV6_DEFAULT_ROUTER_TABLE_SIZE	This option specifies the number of entries in the IPv6 routing table. At least one entry is needed for the default router. Defined in ***nx_api.h***, the default value is 8.
NX_IPV6_PREFIX_LIST_TABLE_SIZE	This option specifies the size of the prefix table. Prefix information is obtained from router advertisements and is part of the IPv6 address configuration. Defined in ***nx_api.h***, the default value is 8.
NX_DISABLE_IPV6_PATH_MTU_DISCOVERY	Defined, this option disables path MTU discovery, which is used to determine the maximum MTU in the path to a target in the NetX Duo host destination table. This enables the NetX Duo host to send the largest possible packet that will not require fragmentation. By default, this option is defined (path MTU is disabled).
NX_PATH_MTU_INCREASE_WAIT_INTERVAL	This option specifies the wait interval in timer ticks to reset the path MTU for a specific target in the destination table. If NX_DISABLE_IPV6_PATH_MTU_DISCOVERY is defined, this has no effect.

Neighbor Cache Configuration Options

Define	Meaning
NXDUO_DISABLE_DAD	Defined, this option disables Duplicate Address Detection (DAD) during IPv6 address assignment. Addresses are set either by manual configuration or through Stateless Address Auto Configuration.
NX_DUP_ADDR_DETECT_TRANSMITS	This option specifies the number of Neighbor Solicitation messages to be sent before NetX Duo marks an interface address as valid. If NXDUO_DISABLE_DAD is defined (DAD disabled), setting this option has no effect. Alternatively, a value of zero (0) turns off DAD but leaves the DAD functionality in NetX Duo. Defined in ***nx_api.h***, the default value is 3.
NX_IPV6_NEIGHBOR_CACHE_SIZE	This option specifies the number of entries in the IPv6 Neighbor Cache table. Defined in ***nx_nd_cache.h***, the default value is 16.
NX_IPV6_DISABLE_PURGE_UNUSED_CACHE_ENTRIES	Defined, this option prevents NetX Duo from removing older cache table entries before their timeout expires to make room for new entries when the table is full. Static and router entries are never purged.
NX_RETRANS_TIMER	This option specifies in milliseconds the length of delay between solicitation packets sent by NetX Duo. Defined in ***nx_nd_cache.h***, the default value is 1000.

NX_REACHABLE_TIME	This option specifies the time out in seconds for a cache entry to exist in the REACHABLE state with no packets received from the cache destination IPv6 address. Defined in ***nx_nd_cache.h***, the default value is 4.
NX_DELAY_FIRST_PROBE_TIME	This option specifies the delay in seconds before the first solicitation is sent out for a cache entry in the STALE state. Defined in ***nx_nd_cache.h***, the default value is 4.
NX_MAX_UNICAST_SOLICIT	This option specifies the number of Neighbor Solicitation messages NetX Duo transmits to determine a specific neighbor's reachability. Defined in ***nx_nd_cache.h***, the default value is 3.
NX_MAX_MULTICAST_SOLICIT	This option specifies the number of Neighbor Solicitation messages NetX Duo transmits as part of the IPv6 Neighbor Discovery protocol when mapping between IPv6 address and MAC address is required. Defined in ***nx_nd_cache.h***, the default value is 3.

Miscellaneous ICMPv6 Configuration Options

Define	Meaning
NXDUO_DISABLE_ICMPV6_REDIRECT_PROCESS	Defined, this option disables ICMPv6 redirect packet processing. NetX Duo by default processes redirect messages and updates the destination table with next hop IP address information.
NXDUO_DESTINATION_TABLE_SIZE	This option specifies the size of the IPv6 destination table used for redirect processing of packets. If NXDUO_DISABLE_ICMPV6_REDIRECT_PROCESS is not defined, this has no effect. Defined in ***nx_api.h***, the default value is 8.
NXDUO_DISABLE_ICMPV6_ROUTER_ADVERTISEMENT_PROCESS	Defined, this option disables NetX Duo from processing information received in IPv6 router advertisement packets.
NXDUO_DISABLE_ICMPV6_ROUTER_SOLICITATION	Defined, this option disables NetX Duo from sending IPv6 router solicitation messages at regular intervals to the router.
NXDUO_DISABLE_ICMPV6_ERROR_MESSAGE	Defined, this option disables NetX Duo from sending an ICMPv6 error message in response to a problem packet (e.g., improperly formatted header or packet header type is deprecated) received from another host.

i *Additional development tool options are described in the* ***readme_netx_duo_generic.txt*** *file supplied on the distribution disk.*

NetX Duo Version ID

The current version of NetX Duo is available to both the user and the application software during runtime. The programmer can find the NetX Duo version in the ***readme_netx_duo_generic.txt*** file. This file also contains a version history of the corresponding port. Application software can obtain the NetX Duo version by examining the global string ***_nx_version_id*** in ***nx_port.h***.

Application software can also obtain release information from the constants shown below defined in ***nx_api.h***. These constants identify the current product release by name and the product major and minor version.

```
#define __PRODUCT_NETXDUO__
#define __NETXDUO_MAJOR_VERSION__
#define __NETXDUO_MINOR_VERSION__
```

CHAPTER 3

Functional Components of NetX Duo

This chapter contains a description of the high-performance NetX Duo TCP/IP stack from a functional perspective.

Execution Overview

There are five types of program execution within a NetX Duo application: initialization, application interface calls, internal IP thread, IP periodic timers, and the network driver.

NetX Duo assumes the existence of ThreadX and depends on its thread execution, suspension, periodic timers, and mutual exclusion facilities.

Initialization

The service ***nx_system_initialize*** must be called before any other NetX Duo service is called. System initialization can be called either from the ThreadX ***tx_application_define*** routine or from application threads.

After ***nx_system_initialize*** returns, the system is ready to create packet pools and IP instances. Because creating an IP instance requires a default packet pool, at least one NetX Duo packet pool must exist prior to creating an IP instance. Creating packet pools and IP instances is allowed from the ThreadX initialization function ***tx_application_define*** and from application threads.

Internally, creating an IP instance is accomplished in two parts: The first part is done within the context of the caller, either from ***tx_application_define*** or from an application thread's context. This includes setting up the IP data structure and creating various IP resources, including the internal IP thread. The second part is performed during the initial execution from the internal IP thread. This is where the application's network driver, supplied during the first part of IP creation, is first called. Calling the network driver from the internal IP thread enables the network driver to perform I/O and suspend during its initialization processing. When the network driver returns from its initialization processing, the IP creation is complete.

Initialization IPv6 in NetX Duo requires a few additional NetX Duo services. These are described in greater detail in the section ***IPv6 Protocol*** later in this chapter.

*The NetX Duo service **nx_ip_status_check** is available to obtain information on the IP instance (primary interface) status. Such status information includes whether or not the link is initialized, enabled and IP address is resolved. This information is used to synchronize application threads needing to use a newly created IP instance. For multihome hosts, **nx_ip_interface_status_check** is available to obtain information on the specified interface status.*

Application Interface Calls

Calls from the application are largely made from application threads running under the ThreadX RTOS. However, some initialization, create, and enable services may be called from ***tx_application_define***. The "Allowed From" sections in Chapter 4 indicate from which each NetX Duo service can be called from.

For the most part, processing intensive activities such as computing checksums is done within the calling thread's context—without blocking access of other threads to the IP instance. For example, UDP checksum calculation is performed inside the ***nxd_udp_socket_send*** service, prior to calling the underlying IP send function. On a received packet, the UDP checksum is calculated in the ***nx_udp_socket_receive*** service. This helps prevent stalling network requests of higher-priority threads because of processing intensive checksum processing in lower-priority threads.

Internal IP Thread

As mentioned, each IP instance in NetX Duo has its own thread. The priority and stack size of the internal IP thread is defined in the ***nx_ip_create*** service. The internal IP thread is created in a ready-to-execute mode. If the IP thread has a higher priority than the calling thread, preemption may occur inside the IP create call.

The entry point of the internal IP thread is at the function ***_nx_ip_thread_entry***. When started, the internal IP thread first completes network driver initialization, which consists of making three calls to the application-specific network driver. The first call is to make an "attachment," so the network driver instance is able to make an association with the IP interface. The second call is to initialize the network driver. After the network driver returns from initialization (it may suspend while waiting for the hardware to be properly set up), the internal IP thread calls the network driver again to enable the link. After the network driver returns from the link enable call, the internal IP thread enters a forever loop checking for various events that need processing for this IP instance. Events processed in this loop include deferred IP packet reception, IP packet fragment assembly, ICMP ping processing, IGMP processing, TCP packet queue processing, TCP periodic processing, IP fragment assembly timeouts, and IGMP periodic processing. Events also include address resolution activities; ARP packet processing and ARP periodic processing in IPv4, and Duplicate Address Detection, Router Solicitation, and Neighbor Discovery in IPv6.

The "attachment" method is new starting in NetX Duo 5.6. Refer to chapter 5 for more information on NetX Duo and driver interface.

The NetX Duo callback functions, including listen and disconnect callbacks, are called from the internal IP thread—not the original calling thread. The *application must take care not to suspend inside any NetX Duo callback function.*

IP Periodic Timers

There are two ThreadX periodic timers used for each IP instance. The first one is a one-second timer for ARP, IGMP, TCP timeout, and it also drives IP fragment reassemble processing. The second timer is a 100ms-timer to drive the TCP retransmission timeout.

Network Driver

Each IP instance in NetX Duo has a primary interface, which is identified by its device driver specified by the application in the ***nx_ip_create*** service. The network driver is responsible for handling various NetX Duo requests, including packet transmission, packet reception, and requests for status and control. On transmission, the network driver is also responsible for buffering packets that cannot be immediately sent through the physical hardware.

For multihome devices, each additional interface associated with the IP instance has an associated network driver that performs these tasks for the respective interface. On a multihome device with identical MAC modules, one may implement a device driver that supports multiple IP instances. Each instance is assigned to a physical interface.

The network driver must also handle asynchronous events occurring on the media. Asynchronous events from the media include packet reception, packet transmission completion, and status changes. NetX Duo provides the network driver with several access functions to handle various received packets. These functions are designed to be called from the interrupt service routine portion of the network driver. For IPv4 networks, the network driver should forward all ARP packets received to the ***_nx_arp_packet_deferred_receive*** function. All RARP packets should be forwarded to ***_nx_rarp_packet_deferred_receive***. There are two options for IP packets. If fast dispatch of IP packets is required, incoming IP packets should be forwarded to ***_nx_ip_packet_receive*** for immediate processing. This greatly improves NetX Duo performance in handling IP packets and UDP packets. Otherwise, forwarding IP packets to ***_nx_ip_packet_deferred_receive*** should be done. This service places the IP packet in the deferred processing queue where it is then handled by the internal IP thread, which results in the least amount of ISR processing time.

The network driver can also defer interrupt processing to run out of the context of the IP thread. This is accomplished by calling the ***_nx_ip_driver_deferred_processing*** function from the network driver's interrupt routine.

See Chapter 5, “NetX Duo Network Drivers” on page 431 for more detailed information on writing NetX Duo network drivers.

Protocol Layering

The TCP/IP implemented by NetX Duo is a layered protocol, which means more complex protocols are built on top of simpler underlying protocols. In TCP/IP, the lowest layer protocol is at the *link level* and is handled by the network driver. This level is typically targeted towards Ethernet, but it could also be fiber, serial, or virtually any physical media.

On top of the link layer is the *network layer*. In TCP/IP, this is the IP, which is basically responsible for sending and receiving simple packets—in a best-effort manner—across the network. Management-type protocols like ICMP and IGMP are typically also categorized as network layers, even though they rely on IP for sending and receiving.

The *transport layer* rests on top of the network layer. This layer is responsible for managing the flow of data between hosts on the network. There are two types of transport services supported by NetX Duo: UDP and TCP. UDP services provide best-effort sending and receiving of data between two hosts in a connectionless manner, while TCP provides reliable connection-oriented service between two host entities.

This layering is reflected in the actual network data packets. Each layer in TCP/IP contains a block of information called a header. This technique of surrounding data (and possibly protocol information) with a header is typically called data encapsulation. Figure 1 shows an example of NetX Duo layering and Figure 2 shows the resulting data encapsulation for UDP data being sent.

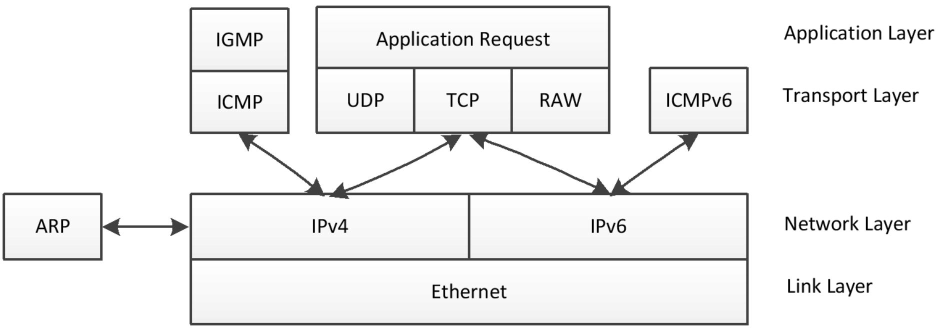


FIGURE 1. Protocol Layering

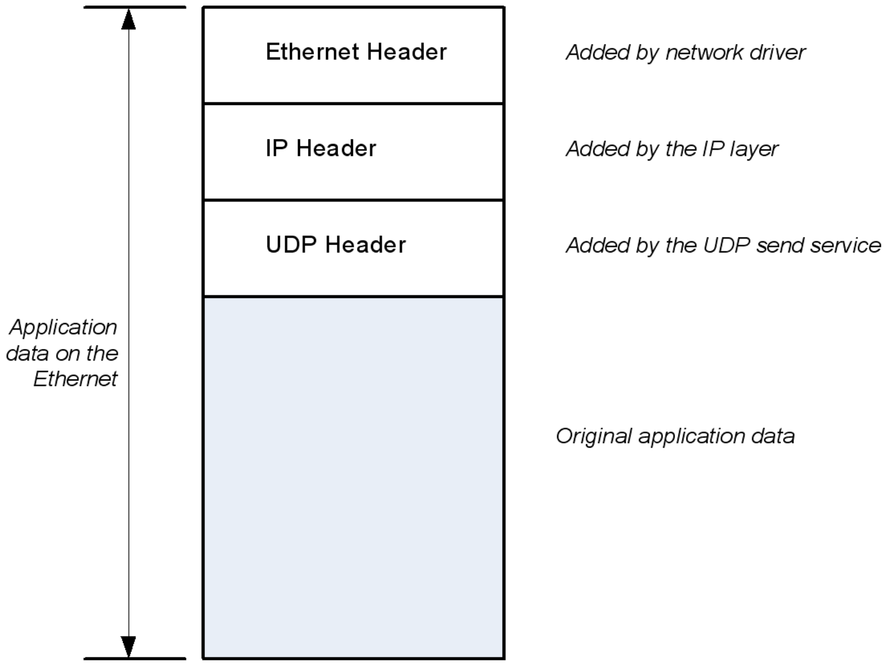


FIGURE 2. UDP Data Encapsulation

Packet Memory Pools

Allocating memory packets in a fast and deterministic manner is always a challenge in real-time networking applications. With this in mind, NetX Duo provides the ability to create and manage multiple pools of fixed-size network packets.

Because NetX Duo packet pools consist of fixed-size memory blocks, there are never any fragmentation problems. Of course, fragmentation causes behavior that is inherently indeterministic. In addition, the time required to allocate and free a NetX Duo packet amounts to simple linked-list manipulation. Furthermore, packet allocation and deallocation is done at the head of the available list. This provides the fastest possible linked list processing.

Lack of flexibility is typically the main drawback of fixed-size packet pools. Determining the optimal packet payload size that also handles the worst-case incoming packet is a difficult task. NetX Duo packets address this problem with packet chaining. An actual network packet can be made of one or more NetX Duo packets linked together. In addition, the packet header maintains a pointer to the top of the packet. As additional protocols are added, this pointer is simply moved backwards and the new header is written directly in front of the data. Without the flexible packet technology, the stack would have to allocate another buffer and copy the data into a new buffer with the new header, which is processing intensive.

Each NetX Duo packet memory pool is a public resource. NetX Duo places no constraints on how packet pools are used.

Creating Packet Pools

Packet memory pools are created either during initialization or during runtime by application threads. There are no limits on the number of packet memory pools in a NetX Duo application.

Packet Header NX_PACKET

By default, NetX Duo places the packet header immediately before the packet payload area. The packet memory pool is basically a series of packets—headers followed immediately by the packet payload. The packet header (***NX_PACKET***) and the layout of the packet pool are pictured in Figure 3.

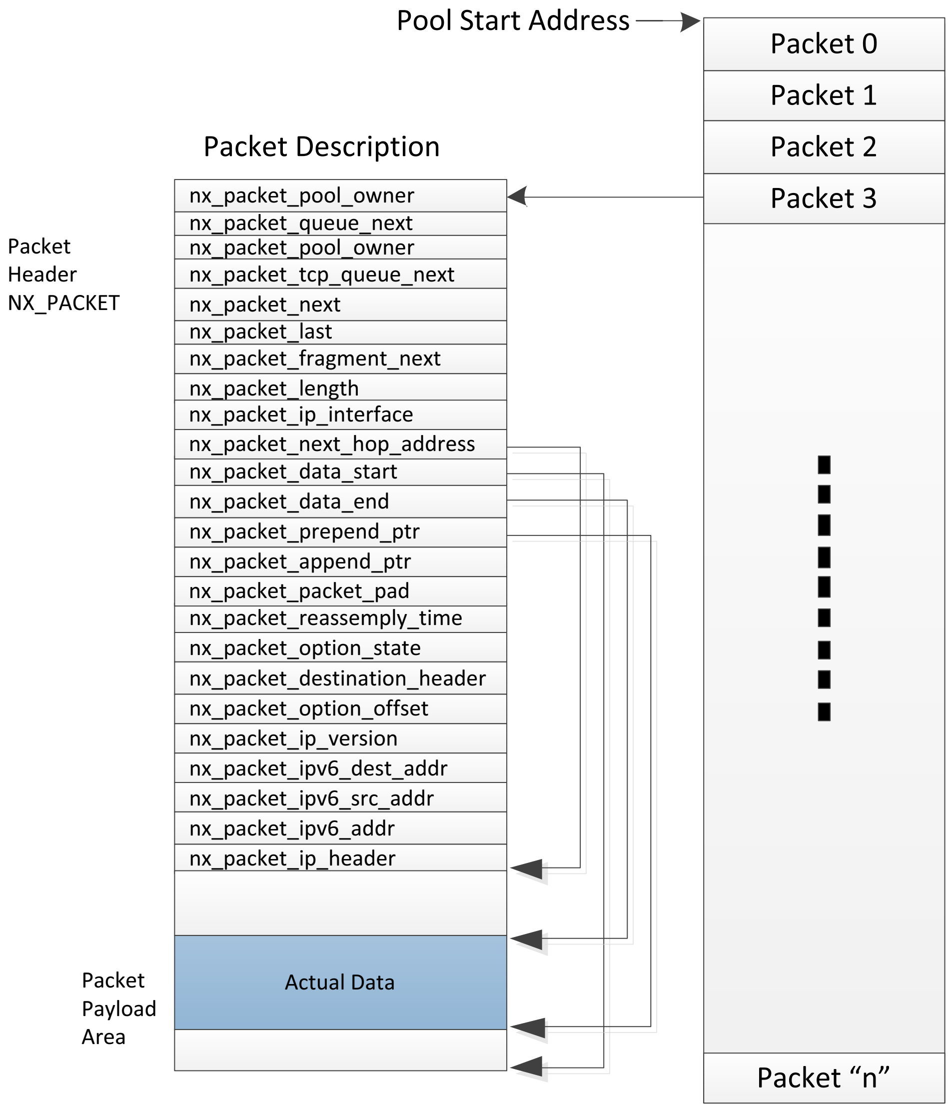


FIGURE 3. Packet Header and Packet Pool Layout

The fields of the packet header are defined as follows. Note there are additional fields added to support IPv6 operation, if IPv6 is enabled. These are included below:

*It is important for the network driver to use the **nx_packet_transmit_release** function when transmission of a packet is complete. This function checks to make sure the packet is not part of a TCP output queue before it is actually placed back in the available pool.*

Packet header	Purpose
nx_packet_pool_owner	This field points to the owner of this particular packet. When the packet is released, it is released to this particular pool. With the pool ownership inside each packet, it is possible for a datagram to span multiple packets from multiple packet pools.
nx_packet_queue_next	This field points to the first packet of the next separate network packet. If NULL, there is no next network packet. This field is used by NetX Duo to queue network packets, and it is also available to the network driver to queue packets for transmission.
nx_packet_tcp_queue_next	This field points to the first packet of the next separate TCP network packet on a specific socket's output queue. This requires a separate pointer because TCP packets are retransmitted if an ACK is not received from the connection prior to a specific timeout. If this field contains the constant NX_PACKET_FREE or NX_PACKET_ALLOCATED, then the network packet is not part of a TCP queue.
nx_packet_next	This field points to the next packet within the same network packet. If NULL, there are no additional packets that are part of the network packet. This field is also used to hold fragmented packets until the entire packet can be re-assembled.

Packet header	Purpose
nx_packet_last	This field points to the last packet within the same network packet. If NULL, this packet represents the entire network packet.
nx_packet_fragment_next	This field is used to hold different incoming IP packets in the process of being unfragmented.
nx_packet_length	This field contains the total number of bytes in the entire network packet, including the total of all bytes in all packets chained together by the *nx_packet_next* member.
nx_packet_ip_interface	This field is the interface control block which is assigned to the packet when it is received by the interface driver, and by NetX Duo for outgoing packets. An interface control block describes the interface e.g. network address, MAC address, IP address and interface status such as link enabled and physical mapping required.
nx_packet_next_hop_address	This field is used by the transmission logic. The next hop address determines how NetX Duo forwards the packet to the final destination. If the destination address is on the local network, the next hop address is the same as the destination address. Otherwise the next hop address would be the router that knows how to forward the packet to the destination.
nx_packet_data_start	This pointer field points to the start of the physical payload area of this packet. It does not have to be immediately following the NX_PACKET header, but that is the default for the ***nx_packet_pool_create*** service.
nx_packet_data_end	This pointer field points to the end of the physical payload area of this packet. The difference between this field and the *nx_packet_data_start* field represents the payload size.

Packet header	Purpose
nx_packet_prepend_ptr	This pointer field points to the location of where packet dat, either protocol header or actual data, is added in front of the existing packet data (if any) in the packet payload area. It must be greater than the *nx_packet_data_start* pointer location and less than or equal to the *nx_packet_append_ptr* pointer. *For performance reasons, NetX Duo assumes* nx_packet_prepend_ptr *always points to a long-word boundary. Hence, any manipulation of this field must maintain this long-word alignment.*
nx_packet_append_ptr	This pointer field points to the end of the data currently in the packet payload area. It must be less than or equal to the *nx_packet_data_end* pointer. The difference between this field and the *nx_packet_data_start* field represents the amount of data in this packet. *The remaining fields in the NX_PACKET structure enable NetX Duo to handle both IPv4 and IPv6 packets after* nx_packet_append_ptr*.*
nx_packet_packet_pad	This fields defines the length of padding in wordsto achieve16-byte alignment if necessary, if NX_PACKET_HEADER_PAD is defined.
nx_packet_reassembly_time	This is a time stamp for measuring the number of seconds a packet is in the reassembly logic. Once a timeout value is reached (the default is 60 seconds) packets of the same fragment ID are released.
nx_packet_option_state	This indicates the option currently being processed in an IPv6 packet.
nx_packet_destination_header	This indicates one or more destination options in an IPv6 packet header.
nx_packet_option_offset	This is reserved for internal use.

Packet header	Purpose
nx_packet_ip_version	This indicates if the packet is an IPv6 (NX_IP_VERSION_V6) or IPv4 (NX_IP_VERSION_V4) packet.
nx_packet_ipv6_dest_addr	Destination IPv6 address, in host byte order
nx_packet_ipv6_src_addr	Source IPv6 address, in host byte order
nx_packet_ipv6_addr	Pointer to the packet interface IPv6 address structure. On the transmit path, this is the address to use as source address. On the receive path, this is the address through which the packet is received.
nx_packet_ip_header	This pointer points to the beginning of the IPv4 or IPv6 header.

Figure 4 shows three network packets in a queue, in which the middle network packet is composed of two packet structures. This illustrates how NetX Duo achieves zero-copy performance by chaining together fixed-size packet structures. It also shows how NetX Duo packet queues are independent from individual packet chains.

Packet Header Offsets

The following types are defined in NetX Duo to take into account the IP header and physical layer (Ethernet) header in the packet. In the latter case, it is assumed to be 16 bytes taking the required 4-byte alignment into consideration. IPv4 packets are still defined in NetX Duo for applications to allocate packets for IPv4 networks. Note that if NetX Duo library is built with IPv6 enabled, the generic packet types (such as NX_IP_PACKET) are mapped to the IPv6 version. If the NetX Duo Library is built without IPv6 enabled, these generic packet types are mapped to the IPv4 version.

The following table shows symbols defined with IPv6 enabled:

Packet Type	Value
NX_IPv6_PACKET (NX_IP_PACKET)	0x38
NX_UDPv6_PACKET (NX_UDP_PACKET)	0x40
NX_TCPv6_PACKET (NX_TCP_PACKET)	0x4c

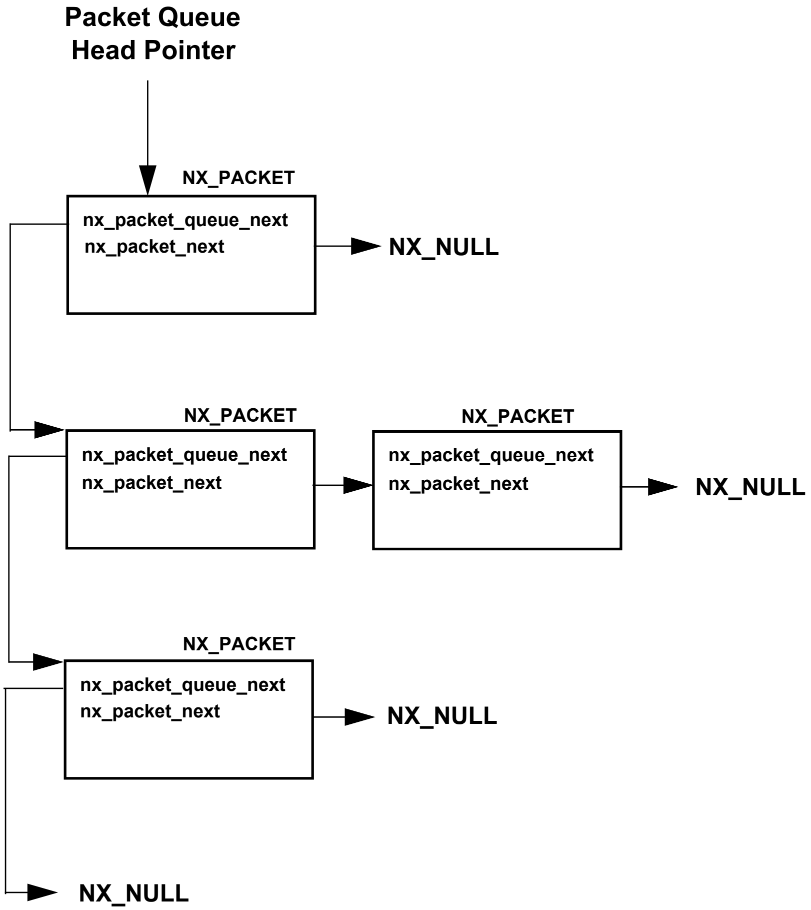


FIGURE 4. Network Packets and Chaining

Packet Type	Value
NX_IPv4_PACKET	0x24
NX_IPv4_UDP_PACKET	0x2c
NX_IPv4_TCP_PACKET	0x38

The following table shows symbols defined with IPv6 disabled:

Packet Type	Value
NX_IPv4_PACKET (NX_IP_PACKET)	0x24
NX_IPv4_UDP_PACKET (NX_UDP_PACKET)	0x2c
NX_IPv4_TCP_PACKET (NX_TCP_PACKET)	0x38

Packet header size is defined to allow enough room to accommodate the size of the header. The ***nx_packet_allocate*** service is used to allocate a packet and adjusts the prepend pointer in the packet according to the type of packet specified. The packet type tells NetX Duo the offset required for inserting the protocol header (such as UDP, TCP, or ICMP) in front of the protocol data.

Pool Capacity

The number of packets in a packet pool is a function of the payload size and the total number of bytes in the memory area supplied to the packet pool create service. The capacity of the pool is calculated by dividing the packet size (including the size of the NX_PACKET header, the payload size, and any necessary padding to keep long-word alignment) into the total number of bytes in the supplied memory area.

Although NetX Duo packet pool create function, ***nx_packet_pool_create***, allocates the payload area immediately following the packet header, it is possible for the application to create packet pools where the payload is in a separate memory area from the packet headers. The only complication with this technique is calculating the header pointer again given just the starting address of the payload. This situation typically occurs inside the receive packet interrupt processing of the network driver. If the packet payload immediately follows the header, the packet header is easily calculated by just moving backwards the size of the packet header. However, if the payload is in a different memory space from its header, the header would need to be calculated by examination of the relative offset of the payload and then applying that same offset to the start of the pool's packet header area.

Packet Pool Memory Area

The memory area for the packet pool is specified during creation. Like other memory areas for ThreadX and NetX Duo objects, it can be located anywhere in the target's address space.

This is an important feature because of the considerable flexibility it gives the application. For example, suppose that a communication product has a high-speed memory area for network buffers. This memory area is easily utilized by making it into a NetX Duo packet memory pool.

Thread Suspension

Application threads can suspend while waiting for a packet from an empty pool. When a packet is returned to the pool, the suspended thread is given this packet and resumed.

If multiple threads are suspended on the same packet pool, they are resumed in the order they were suspended (FIFO).

Pool Statistics and Errors

If enabled, the NetX Duo packet management software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for packet pools:

Total Packets in Pool
Free Packets in Pool
Total Packet Allocations
Pool Empty Allocation Requests
Pool Empty Allocation Suspensions
Invalid Packet Releases

All of these statistics and error reports, except for total and free packet count in pool, are built into NetX Duo library unless NX_DISABLE_PACKET_INFO is defined. This data is available to the application with the ***nx_packet_pool_info_get*** service.

Packet Pool Control Block NX_PACKET_POOL

The characteristics of each packet memory pool are found in its control block. It contains useful information such as the linked list of free packets, the number of free packets, and the payload size for packets in this pool. This structure is defined in the ***nx_api.h*** file.

Packet pool control blocks can be located anywhere in memory, but it is most common to make the control block a global structure by defining it outside the scope of any function.

IPv4 Protocol

The Internet Protocol (IP) component of NetX Duo is responsible for sending and receiving IPv4 packets on the Internet. In NetX Duo, it is the component ultimately responsible for sending and receiving TCP, UDP, ICMP, and IGMP messages, utilizing the underlying network driver.

NetX Duo supports both IPv4 protocol (RFC 791) and IPv6 protocol (RFC 2460). This section discusses IPv4. IPv6 is discussed in the next section.

IPv4 Addresses

Each host on the Internet has a unique 32-bit identifier called an IP address. There are five classes of IPv4 addresses as described in Figure 5. The

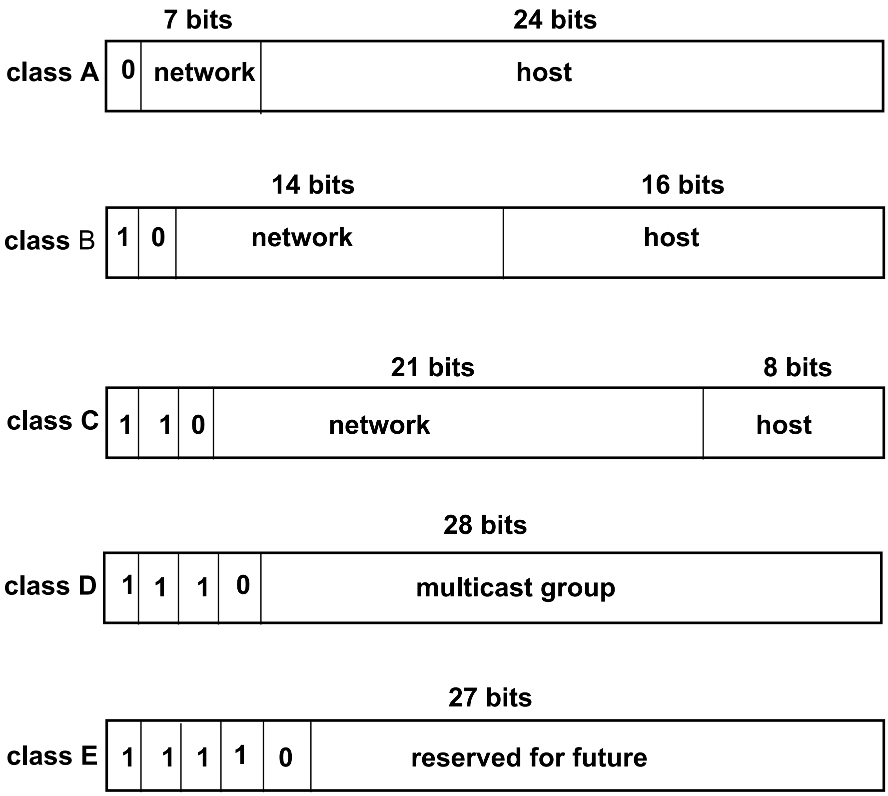


FIGURE 5. IPv4 Address Structure

ranges of the five IPv4 address classes are as follows:

Class	Range
A	0.0.0.0 to 127.255.255.255
B	128.0.0.0 to 191.255.255.255
C	192.0.0.0 to 223.255.255.255
D	224.0.0.0 to 239.255.255.255
E	240.0.0.0 to 247.255.255.255

There are also three types of address specifications: *unicast*, *broadcast*, and *multicast*. Unicast addresses are those IPv4 addresses that identify a specific host on the Internet. Unicast addresses can be either a source or a destination IPv4 address. A broadcast address identifies all hosts on a specific network or sub-network and can only be used as destination addresses. Broadcast addresses are specified by having the host ID portion of the address set to ones. Multicast addresses (Class D) specify a dynamic group of hosts on the Internet. Members of the multicast group may join and leave whenever they wish.

i *Only connectionless protocols like UDP over IPv4 can utilize broadcast and the limited broadcast capability of the multicast group.*

i *The macro* IP_ADDRESS *is defined in* **nx_api.h**. *It allows easy specification of IPv4 addresses using commas instead of a periods. For example,* IP_ADDRESS(128,0,0,0) *specifies the first class B address shown in Figure 5.*

Gateway IPv4 Address

In addition to the different types of network, loopback, and broadcast addresses, it is possible to set the IP instance gateway IPv4 address using the ***nx_ip_gateway_address_set*** service. A gateway resides on the local network, and its purpose is to provide a place ("next hop") to transmit packets whose destination lies outside the local network. Once set, all out-of network requests are routed by NetX Duo to the gateway. Note that the default gateway must be directly accessible through one of the physical interfaces.

IPv4 Header

For any packet to be sent on the Internet, it must have an IPv4 header. When higher-level protocols (UDP, TCP, ICMP, or IGMP) call the IP component to send a packet, an IP header is placed in front of the beginning of the packet. Conversely, when IP packets are received from the network, the IP component removes the IP header from the packet before delivery to the higher-level protocols. Figure 6 shows the format of the IP header.

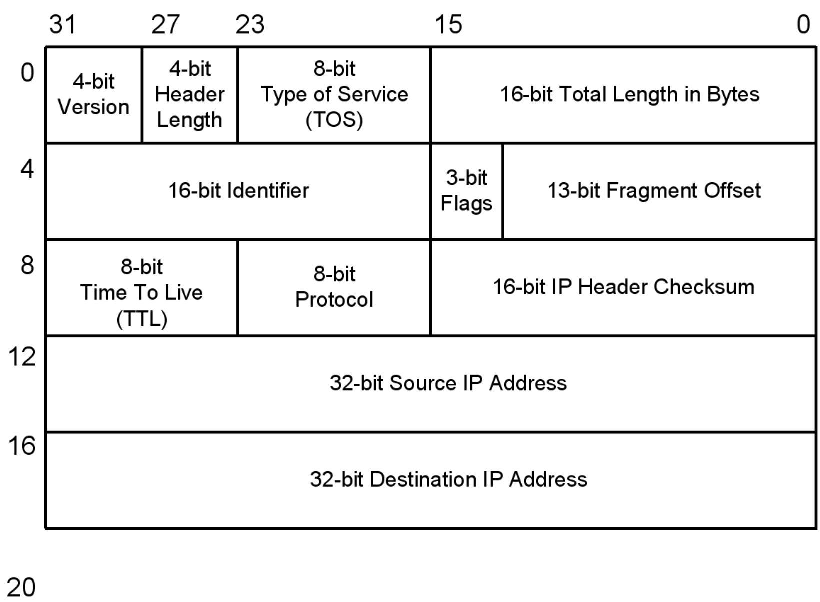


FIGURE 6. IPv4 Header Format

*All headers in the TCP/IP implementation are expected to be in **big endian** format. In this format, the most significant byte of the word resides at the lowest byte address. For example, the 4-bit version and the 4-bit header length of the IP header must be located on the first byte of the header.*

The fields of the IPv4 header are defined as follows:

IPv4 Header Field	Purpose
4-bit version	This field contains the version of IP this header represents. For IP version 4, which is what NetX Duo supports, the value of this field is 4.
4-bit header length	This field specifies the number of 32-bit words in the IP header. If no option words are present, the value for this field is 5.
8-bit type of service (TOS)	This field specifies the type of service requested for this IP packet. Valid requests are as follows:

TOS Request	Value
Normal	0x00
Minimum Delay	0x10
Maximum Data	0x08
Maximum Reliability	0x04
Minimum Cost	0x02

16-bit total length	This field contains the total length of the IP datagram in bytes, including the IP header. An IP datagram is the basic unit of information found on a TCP/IP Internet. It contains a destination and source address in addition to data. Because it is a 16-bit field, the maximum size of an IP datagram is 65,535 bytes.
16-bit identification	The field is a number used to uniquely identify each IP datagram sent from a host. This number is typically incremented after an IP datagram is sent. It is especially useful in assembling received IP packet fragments.

IPv4 Header Field	Purpose
3-bit flags	This field contains IP fragmentation information. Bit 14 is the "don't fragment" bit. If this bit is set, the outgoing IP datagram will not be fragmented. Bit 13 is the "more fragments" bit. If this bit is set, there are more fragments. If this bit is clear, this is the last fragment of the IP packet.
13-bit fragment offset	This field contains the upper 13-bits of the fragment offset. Because of this, fragment offsets are only allowed on 8-byte boundaries. The first fragment of a fragmented IP datagram will have the "more fragments" bit set and have an offset of 0.
8-bit time to live (TTL)	This field contains the number of routers this datagram can pass, which basically limits the lifetime of the datagram.
8-bit protocol	This field specifies which protocol is using the IP datagram. The following is a list of valid protocols and their values:

Protocol	Value
ICMP	0x01
IGMP	0x02
TCP	0X06
UDP	0X11

16-bit checksum	This field contains the 16-bit checksum that covers the IP header only. There are additional checksums in the higher level protocols that cover the IP payload.
32-bit source IP address	This field contains the IP address of the sender and is always a host address.
32-bit destination IP address	This field contains the IP address of the receiver or receivers if the address is a broadcast or multicast address.

IPv6 Protocol

IPv6 has three types of address specifications: unicast, anycast (not supported in NetX Duo), and multicast. Unicast addresses are those IP addresses that identify a specific host on the Internet. Unicast addresses can be either a source or a destination IP address. Multicast addresses specify a dynamic group of hosts on the Internet. Members of the multicast group may join and leave whenever they wish.

IPv6 does not use IP broadcast packets. The ability to send a packet to all hosts can be achieved by sending a packet to the link-local all hosts multicast group which is described below.

IPv6 utilizes multicast addresses to perform Neighbor Discovery, Router Discovery, and Stateless Address Auto Configuration procedures.

IPv6 Addresses

IPv6 addresses are 128 bits. The architecture of IPv6 address is described in RFC 4291. The address is divided into a prefix containing the most significant bits and a host address containing the lower bits. The prefix indicates the type of address and is roughly the equivalent of the network address in IPv4 network.

There are two types of IPv6 addresses: link local addresses, typically constructed by combining the well-known link local prefix with the interface MAC address, and global IP addresses, which also has the prefix portion and the host ID portion. A global address may be configured manually, or through the Stateless Address Auto Configuration. NetX Duo supports both types of IPv6 address configuration.

NetX Duo supports both IPv4 and IPv6 address formats as mentioned previously. To accommodate both IPv4 and IPv6 formats, NetX Duo provides a new data type, NXD_ADDRESS, for holding IPv4 and IPv6 addresses. The definition of this structure is shown below. The address field is a union of IPv4 and IPv6 addresses.

```
typedef struct NXD_ADDRESS_STRUCT
{
    ULONG     nxd_ip_version;
    union
    {
        ULONG v4;
        ULONG v6[4];
    } nxd_ip_address;
} NXD_ADDRESS;
```

In the NXD_ADDRESS structure, the first element, *nxd_ip_version*, indicates IPv4 or IPv6 version. Supported values are either NX_IP_VERSION_V4 or NX_IP_VERSION_V6. *nxd_ip_version* indicates which field in the *nxd_ip_address* union use as the IP address. NetX Duo API services typically take an NXD_ADDRESS input argument in lieu of the ULONG (32 bit) IP address.

Link Local Addresses

A link-local address is only valid on the local network. A device can send and receive packets to another device on the same network after a valid link local address is assigned to it. An application assigns a link-local address by calling the NetX Duo service ***nxd_ipv6_address_set***, with the prefix length parameter set to 10. The application may supply a link-local address to the service, or it may simply use NX_NULL as the link-local address and allow NetX Duo to construct a link-local address based on the device's MAC address.

The following example instructs NetX Duo to configure the link-local address with a prefix length of 10 on the primary interface (index 0) using its MAC address:

```
nxd_ipv6_address_set(ip_ptr, 0, NX_NULL, 10,
NX_NULL);
```

In the example above, if the MAC address of the interface is 54:32:10:1A:BC:67, the corresponding link-local address would be:

`FE80::5632:10FF:FE1A:BC67`

Note that the host ID portion of the IPv6 address (`5632:10FF:FE1A:BC67`) is made up of the 6-byte MAC address, with the following modifications:

- `0xFFFE` inserted between byte 3 and byte 4 of the MAC address
- Second lowest bit of the first byte of the MAC address (U/L bit) is set to 1

Refer to RFC 2464 (Transmission of IPv6 Packets over Ethernet Network) for more information on how to construct the host portion of an IPv6 address from its interface MAC address.

There are a few special multicast addresses for sending multicast messages to one or more hosts in IPv6:

All nodes group	`FF02::1`	All hosts on the local network
All routers group	`FF02::2`	All routers on the local network
Solicited-node	`FF02::1:FF00:0`	Explained below

A solicited-node multicast address targets specific hosts on the local link rather than all the IPv6 hosts. It consists of the prefix `FF02::1:FF00:0`, which is 104 bits and the last 24-bits of the target IPv6 address. For example, an IPv6 address `205B:209D:D028::F058:D1C8:1024` has a solicited-node multicast address of address `FF02::1:FFC8:1024`.

The double colon notation indicates the intervening bits are all zeroes. `FF02::1:FF00:0` *fully expanded looks like*
`FF02:0000:0000:0000:0000:0001:FF00:0000`

Global IPv6 Addresses

An example of an IPV6 global address is

`2001:0123:4567:89AB:CDEF::1`

NetX Duo stores IPv6 addresses in the NXD_ADDRESS structure. In the example below, the NXD_ADDRESS variable `global_ipv6_address` contains a unicast IPv6 address. The example demonstrates a NetX Duo device host creating a specific IPv6 global address for its primary interface:

```
NXD_ADDRESS global_ipv6_address;
UINT        primary_interface_index = 0;

global_ipv6_address.nxd_ip_version  = NX_IP_VERSION_V6;
global_ipv6_address.nxd_ip_address.v6[0]  = 0x20010123;
global_ipv6_address.nxd_ip_address.v6[1]  = 0x456789AB;
global_ipv6_address.nxd_ip_address.v6[2]  = 0xCDEF0000;
global_ipv6_address.nxd_ip_address.v6[3]  = 0x00000001;

status = nxd_ipv6_address_set(&ip_0,
primary_interface_index, global_ipv6_address,64,NX_NULL);
```

Note that the prefix of this IPv6 address is `2001:0123:4567:89AB`, which is 64 bits long and is a common prefix length for global unicast IPv6 addresses on Ethernet.

The NXD_ADDRESS structure also holds IPv4 addresses. An IP address of `192.1.168.10`

`(0xC001A80A)` would have the following memory layout:

Field	Value
global_ipv4_address.nxd_ip_version	NX_IP_VERSION_V4
global_ipv4_address.nxd_ip_address.v4	0xC001A80A

When an application passes an address to NetX Duo services, the *nxd_ip_version* field must specify the correct IP version for proper packet handling.

To be backward compatible with existing NetX applications, NetX Duo supports all existing NetX services. Internally, NetX Duo converts the IPv4 address type ULONG to an NXD_ADDRESS data type before forwarding it to the actual NetX Duo service.

Following is an example:

```
NXD_ADDRESS global_ipv4_address;
NX_TCP_SOCKET tcp_socket;
UINT            port_number = 80;

/* Make a connection to the destination IPv4 address
   192.1.168.12 through an already created TCP socket bound
   to the well known HTTP port number 80. */

global_ipv4_address.nxd_ip_version = NX_IP_VERSION_V4;
global_ipv4_address.nxd_ip_address.v4 = 0xC001A80C;

nxd_tcp_client_socket_connect(&tcp_socket,
                              &global_ipv4_address,
                              port_number,
                              NX_WAIT_FOREVER);
```

The following is the equivalent NetX API:

```
ULONG server_ip = 0xC001A80C;
NX_TCP_SOCKET tcp_socket;
UINT            port_number = 80;

nx_tcp_client_socket_connect(&tcp_socket,
server_ip,
port_number,
NX_WAIT_FOREVER);
```

Application developers are encouraged to use the nxd version of these APIs.

IPv6 Default Routers

IPv6 uses a default router to route packets to off-link destinations. The NetX Duo service ***nxd_ipv6_default_router_add*** enables an application to add a default router to the default router table. See Chapter 4 “Description of Services” for more default router services offered by NetX Duo.

When forwarding IPv6 packets, NetX Duo first checks if the packet destination is on-link. If not, NetX Duo checks the default routing table for a valid router to forward the off-link packet to.

IPv6 Header

The IPv6 header has been modified from the IPv4 header. When allocating a packet, the caller specifies the application protocol (e.g., UDP, TCP), buffer size in bytes, and hop limit.

The Figure 7 shows the format of the IPv6 header and the table lists the header components.

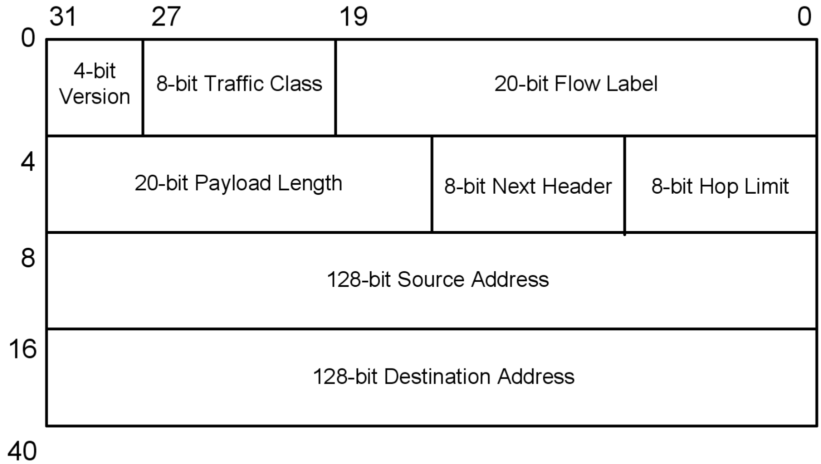


FIGURE 7. IPv6 Header Format

IP header	Purpose
Version	4-bit field for IP version. For IPv6 networks, the value in this field must be 6; For IPv4 networks it must be 4.
Traffic Class	8-bit field that stores the traffic class information. This field is not used by NetX Duo.
Flow Label	20-bit field to uniquely identify the flow, if any, that a packet is associated with. A value of zero indicates the packet does not belong to a particular flow. This replaces the *TOS* field in IPv4.
Payload Length	16-bit field indicating the amount of data in bytes of the IPv6 packet following the IPv6 base header. This includes all encapsulated protocol header and data.

IP header	Purpose
Next Header	8-bit field indicating the type of the extension header that follows the IPv6 base header. This replaces the *Protocol* field in IPv4.
Hop Limit	8-bit field that limits the number of routers the packet is allow to go through. This replaces the *TTL* field in IPv4.
Source Address	128-bit field that stores the IPv6 address of the sender.
Destination Address	128-bit field that sores the IPv6 address of the destination.

IP Fragmentation

The network driver may have limits on the size of outgoing packets. This physical limit is called the maximum transmission unit (MTU). The ***nx_ip_driver_mtu*** member of the NX_INTERFACE control block contains the MTU, which is initially set up by the device driver.

Although not recommended, the application may generate datagrams larger than the underlying network driver's MTU size. Before transmitting such IP datagram, the IP layer must fragment such packets. At the IP receiving end, the IP layer must collect and reassemble all the fragments before sending the packet to upper layer applications. In order to support IP fragmentation and reassembly operation, the system designer must enable the IP fragmentation feature in NetX Duo using the ***nx_ip_fragment_enable*** service. If this feature is not enabled, incoming fragmented IP packets are discarded, as well as packets that exceed the network driver's MTU.

The IP Fragmentation logic can be removed completely by defining ***NX_DISABLE_FRAGMENTATION*** *when building the NetX Duo library. Doing so helps reduce the code size of NetX Duo. Note that in this situation, both the IPv4 and IPv6 fragmentation/reassembly functions are disabled.*

In an IPv6 network, routers do not fragment a datagram if the size of the datagram exceeds its minimum MTU size. Therefore, it is up to the sending device to determine the minimum MTU between the source and the destination, and to ensure the IP datagram size does not exceed the path MTU.

IP Send

The IP send processing in NetX Duo is very streamlined. The prepend pointer in the packet is moved backwards to accommodate the IP header. The IP header is completed (with all the options specified by the calling protocol layer), the IP checksum is computed inline (for IPv4 packets only), and the packet is dispatched to the associated network driver. In addition, outgoing fragmentation is also coordinated from within the IP send processing.

For IPv4, NetX Duo initiates ARP requests if physical mapping is needed for the destination IP address. IPv6 uses Neighbor Discovery for IPv6-address-to-physical-address mapping.

For IPv4 connectivity, packets that require IP address resolution (i.e., physical mapping) are enqueued on the ARP queue until the number of packets queued exceeds the ARP queue depth (NX_ARP_MAX_QUEUE_DEPTH). If the queue depth is reached, NetX Duo will remove the oldest packet on the queue and continue waiting for address resolution for the remaining packets enqueued.

For devices with multiple network interfaces, NetX Duo chooses which interface based on the destination IP address. If a destination address is IPv4 broadcast or multicast, the application needs to specify the outgoing network interface to use, or NetX Duo drops the packet.

IP Receive

The IP receive processing is either called from the network driver or the internal IP thread (for processing packets on the deferred received packet queue). The IP receive processing examines the protocol field and attempts to dispatch the packet to the proper protocol component. Before the packet is actually dispatched, the IP header is removed by advancing the prepend pointer past the IP header.

IP receive processing also detects fragmented IP packets and performs the necessary steps to re-assemble them if fragmentation is enabled.

NetX Duo determines the appropriate interface based on the interface specified in the packet. If the packet interface is NULL, NetX Duo defaults to the primary interface. This is done to guarantee compatibility with legacy NetX Duo Ethernet drivers.

Raw IP Send

The application may send raw IP packets (packets with only an IP header and payload) directly using the ***nxd_ip_raw_packet_send*** service if raw IP packet processing has been enabled with the ***nx_ip_raw_packet_enabled*** service. When transmitting a unicast packet on an IPv6 network, NetX Duo automatically determines the best interface to send the packets out on, based on the destination address. If the destination address is a multicast (or broadcast for IPv4) address, however, NetX Duo will default to the first (primary) interface. Therefore, to send such packets out on non primary interfaces, the host application must use the ***nx_ip_raw_packet_interface_send*** service to specify the network interface.

Raw IP Receive

If raw IP packet processing is enabled, the application may receive raw IP packets through the ***nx_ip_raw_packet_receive*** service. All incoming packets are processed according to the protocol specified in the IP header. If the protocol specifies UDP, TCP, IGMP or ICMP, NetX Duo will process the packet using the appropriate handler for the packet protocol type. If the protocol is not one of these protocols, and raw IP receive is enabled, the packet will be processed by ***nx_ip_raw_packet_receive***. In addition, application threads may suspend with an optional timeout while waiting for a raw IP packet.

Creating IP Instances

IP instances are created either during initialization or during runtime by application threads. The initial IPv4 address, network mask, default packet pool, media driver, and memory and priority of the internal IP thread are defined by the ***nx_ip_create*** service even if the application intends to use IPv6 networks only. If the application initializes the IP instance with its IPv4 address set to an invalid address(0.0.0.0), it is assumed that through DHCP or similar protocol that the invalid address will be replaced by a dynamically assigned one.) For systems with multiple network interfaces, each interface can be attached to the same IP instance after the IP instance is created.

Each interface in an IP instance may have multiple IPv6 global addresses. In addition to using DHCPv6 for IPv6 address assignment, a device may also use Stateless Address Auto Configuration. More information is available in the IP Control Block and IPv6 Address Resolution sections later in this chapter.

A multihome device must associate additional network interfaces with the IP instance. It can do so by calling the ***nx_ip_interface_attach*** service. This services stores information about the network interface (such as IP address, network mask) in the interface control block. This service allows the driver to associate the driver instance with the IP interface instance. As the driver receives a data packet, it needs to store the interface information in NX_PACKET structure before forwarding it to the IP receive logic. Note an IP instance must already be created before attaching any interfaces.

More details on the NetX Duo multihome support are available in “Multiple Network Interface (Multihome) Support” on page 84.

Default Packet Pool

Each IP instance is given a default packet pool during creation. This packet pool is used to allocate packets for ARP, RARP, ICMP, IGMP, various TCP ACK and state changes, Neighbor Discovery, Router Discovery, and Duplicate Address Detection. If the default packet pool is empty, the underlying NetX Duo activity aborts the packet pool entirely, and returns an error message if possible.

IP Helper Thread

Each IP instance has a helper thread. This thread is responsible for handling all deferred packet processing and all periodic processing. The IP helper thread is created in ***nx_ip_create.*** This is where the thread is given its stack and priority. Note that the first processing in the IP helper thread is to finish the network driver initialization associated with the IP create service. After the network driver initialization is complete, the helper thread starts an endless loop to process packet and periodic requests.

If unexplained behavior is seen within the IP helper thread, increasing its stack size during the IP create service is the first debugging step. If the stack is too small, the IP helper thread could possibly be overwriting memory, which may cause unusual problems.

Thread Suspension

Application threads can suspend while attempting to receive raw IP packets. After a raw packet is received, the new packet is given to the first thread suspended and that thread is resumed. NetX Duo services for receiving packets all have an optional suspension timeout. When a packet is received or the timeout expires, the application thread is resumed with the appropriate completion status.

IP Statistics and Errors

If enabled, the NetX Duo keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP instance:

Total IP Packets Sent
Total IP Bytes Sent
Total IP Packets Received
Total IP Bytes Received
Total IP Invalid Packets
Total IP Receive Packets Dropped
Total IP Receive Checksum Errors
Total IP Send Packets Dropped
Total IP Fragments Sent
Total IP Fragments Received

All of these statistics and error reports are available to the application with the ***nx_ip_info_get*** service.

IP Control Block NX_IP

The characteristics of each IP instance are found in its control block. It contains useful information such as the IP address, network mask, and the linked list of destination IP and physical hardware address mapping. This structure is defined in the ***nx_api.h*** file. If IPv6 is enabled, it also contains an array of IPv6 address for its interfaces, the number of which is specified by the user configurable option NX_MAX_IPV6_ADDRESSES. The default value allows each physical interface to have three IPv6 addresses.

IP instance control blocks can be located anywhere in memory, but it is most common to make the control block a global structure by defining it outside the scope of any function.

Multiple Network Interface (Multihome) Support

NetX Duo supports hosts connected to multiple physical network interfaces using a single IP instance. To utilize the multihome feature, set the user configurable option NX_MAX_PHYSICAL_INTERFACES to the number of physical interfaces needed.

To utilize a logical loopback interface, ensure the configurable option NX_DISABLE_LOOPBACK_INTERFACE is not set. When the loopback interface is enabled (the default), the total number of interfaces defined by NX_MAX_IP_INTERFACES is automatically updated to NX_MAX_PHYSICAL_INTERFACES + 1.

The host application creates a single IP instance for the primary interface using the ***nx_ip_create*** service. For each additional interface, the host application attaches the interface to the IP instance using the ***nx_ip_interface_attach*** service.

Each interface structure contains a subset of network information about the network interface that is contained in the IP control block, including interface

IPv4 address, subnet mask, interface MTU size, and MAC-layer address information.

NetX Duo with multihome support is backward compatible with earlier versions of NetX Duo. Services that do not take explicit interface information default to the primary interface.

The primary interface has index zero in the IP instance list. Each subsequent interface attached to the IP instance is assigned the next index.

All upper layer protocol services for which the IP instance is enabled, including TCP, UDP and IGMP, are available to all the attached interfaces. These upper layer protocols are for the most part not directly involved with choosing or determining the packet interface when sending or receiving packets.

In most cases, NetX Duo can determine the best interface to send the packet out on from the packet destination IP address. NetX Duo services are added to allow applications to specify the outgoing interface to use, in cases where the best interface cannot be determined by the destination IP address. An example would be IPv4 broadcast or multicast destination addresses.

Services specifically for developing multihome applications include the following:

nx_igmp_multicast_interface_join
nx_ip_interface_address_get
nx_ip_interface_address_set
nx_ip_interface_attach
nx_ip_interface_info_get
nx_ip_interface_status_check
nx_ip_raw_packet_interface_send
nx_udp_socket_interface_send

These services are explained in greater detail in "Description of NetX Duo Services" on page 133.

Static IPv4 Routing

The static routing feature allows an application to specify an interface and next hop address for specific out of network destination IP addresses. If static routing is enabled, NetX Duo searches through the static routing table for an entry matching the destination address of the packet to send. If no match is found, NetX Duo searches through the list of physical interfaces and chooses an interface and next hop based on the destination IP address and network mask. If the destination does not match any of the network interfaces attached to the IP instance, NetX Duo chooses the IP instance default gateway.

Entries can be added and removed from the static routing table using the ***nx_ip_static_route_add*** and ***nx_ip_static_route_delete*** services, respectively. To use static routing, the host application must enable this feature by defining NX_ENABLE_IP_STATIC_ROUTING.

When adding an entry to the static routing table, NetX Duo checks for a matching entry for the specified destination address already in the table. If one exists, it gives preference to the entry with the smaller network (larger number of most significant bits) in the network mask.

The following sections describe the different protocols for IPv4 and IPv6 for establishing a host IP address.

Address Resolution Protocol (ARP) in IPv4

The Address Resolution Protocol (ARP) is responsible for dynamically mapping 32-bit IPv4 addresses to those of the underlying physical media

(RFC 826). Ethernet is the most typical physical media, and it supports 48-bit addresses. The need for ARP is determined by the IP network driver supplied to the ***nx_ip_create*** service. If physical mapping is required, the network driver must set the ***nx_interface_address_mapping_needed*** member of the associated NX_INTERFACE structure in the driver initialization procedure.

ARP Enable

For ARP to function properly, it must first be enabled by the application with the ***nx_arp_enable*** service. This service sets up various data structures for ARP processing, including the creation of an ARP cache area from the memory supplied to the ARP enable service.

ARP Cache

The ARP cache can be viewed as an array of internal ARP mapping data structures. Each internal structure is capable of maintaining the relationship between an IP address and a physical hardware address. In addition, each data structure has link pointers so it can be part of multiple linked lists.

ARP Dynamic Entries

By default, the ARP enable service places all entries in the ARP cache on the list of available dynamic ARP entries. A dynamic ARP entry is allocated from this list by the NetX Duo when a send request to an unmapped IP address is detected. After allocation, the ARP entry is set up and an ARP request is sent to the physical media.

If all dynamic ARP entries are in use, the least recently used ARP entry is replaced with a new mapping.

ARP Static Entries

The application can also set up static ARP mapping by using the ***nx_arp_static_entry_create*** service. This service allocates an ARP entry from the dynamic ARP entry list and places it on the static list with the mapping information supplied by the application. Static ARP entries are not subject to reuse or aging.

ARP Messages

As mentioned previously, an ARP request message is sent when the IP task detects that mapping is needed for an IP address. ARP requests are sent periodically (every **NX_ARP_UPDATE_RATE** seconds) until a corresponding ARP response is received. A total of **NX_ARP_MAXIMUM_RETRIES** ARP requests are made before the ARP attempt is abandoned. When an ARP response is received, the associated physical address information is stored in the ARP entry that is in the cache.

For multihome applications, NetX Duo determines which interface to send the ARP requests and responses based on information specified by the packet interface.

i

Outgoing IP packets are queued while NetX Duo waits for the ARP response. The number of outgoing IP packets queued is defined by the constant NX_ARP_MAX_QUEUE_DEPTH.

NetX Duo also responds to ARP requests from other nodes on the local IPv4 network. When an external ARP request is made that matches the current IP address, NetX Duo builds an ARP response message that contains the current physical address.

The formats of Ethernet ARP requests and responses are shown in Figure 8 and are described below .

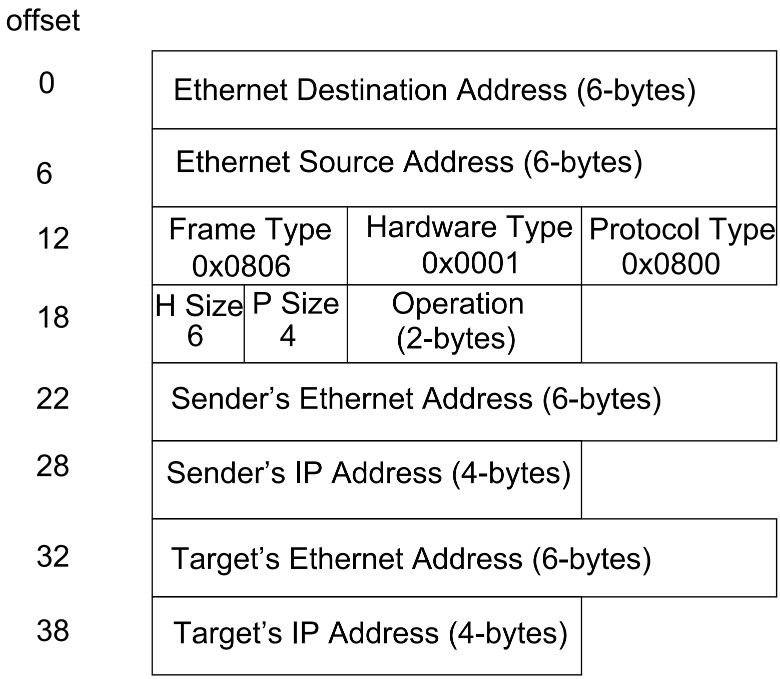


FIGURE 8. ARP Packet Format

Request/Response Field	**Purpose**
Ethernet Destination Address	This 6-byte field contains the destination address for the ARP response and is a broadcast (all ones) for ARP requests. This field is setup by the network driver.
Ethernet Source Address	This 6-byte field contains the address of the sender of the ARP request or response and is set up by the network driver.
Frame Type	This 2-byte field contains the type of Ethernet frame present and, for ARP requests and responses, this is equal to 0x0806. This is the last field the network driver is responsible for setting up.

Request/Response Field	Purpose
Hardware Type	This 2-byte field contains the hardware type, which is 0x0001 for Ethernet.
Protocol Type	This 2-byte field contains the protocol type, which is 0x0800 for IP addresses.
Hardware Size	This 1-byte field contains the hardware address size, which is 6 for Ethernet addresses.
Protocol Size	This 1-byte field contains the IP address size, which is 4 for IP addresses.
Operation Code	This 2-byte field contains the operation for this ARP packet. An ARP request is specified with the value of 0x0001, while an ARP response is represented by a value of 0x0002.
Sender Ethernet Address	This 6-byte field contains the sender's Ethernet address.
Sender IP Address	This 4-byte field contains the sender's IP address.
Target Ethernet Address	This 6-byte field contains the target's Ethernet address.
Target IP Address	This 4-byte field contains the target's IP address.

ARP requests and responses are Ethernet-level packets. All other TCP/IP packets are encapsulated by an IP packet header.

*All ARP messages in the TCP/IP implementation are expected to be in **big endian** format. In this format, the most significant byte of the word resides at the lowest byte address.*

ARP Aging

Automatic invalidation of dynamic ARP entries is supported. The constant **NX_ARP_EXPIRATION_RATE** specifies the number of seconds an established IP address to physical mapping stays valid. After expiration, the ARP entry is removed from the ARP cache. The next attempt to send to the corresponding IP address will result in a new ARP request. ARP aging is disabled when the NX_ARP_EXPIRATION_RATE is set to zero. The NX_ARP_EXPIRATION_RATE is defaulted zero in NetX Duo.

ARP Statistics and Errors

If enabled, the NetX Duo ARP software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP's ARP processing:

Total ARP Requests Sent
Total ARP Requests Received
Total ARP Responses Sent
Total ARP Responses Received
Total ARP Dynamic Entries
Total ARP Static Entries
Total ARP Aged Entries
Total ARP Invalid Messages

All these statistics and error reports are available to the application with the ***nx_arp_info_get*** service.

Reverse Address Resolution Protocol (RARP) in IPv4

The Reverse Address Resolution Protocol (RARP) is the protocol for requesting network assignment of the host's 32-bit IP addresses (RFC 903). This is done through an RARP request and continues periodically until a network member assigns an IP address to the host network interface in an RARP response. The IP create service ***nx_ip_create*** service with a zero IP address creates a need for RARP. If RARP is enabled by the application, it can use the RARP protocol to request an IP address from the network server for the IP network interface with a zero IP address.

RARP Enable

To use RARP, the application must create the IP instance with an IP address of zero, then enable RARP. For multihome hosts, at least one interface associated with the IP instance must have an IP address of zero. The RARP processing periodically sends RARP request messages for the NetX Duo host requiring an IP address until a valid RARP reply with the network designated IP address is received. At this point, RARP processing is complete.

RARP Request

The format of an RARP request packet is almost identical to the ARP packet shown in Figure 8 on page 89.The only difference is the frame type field is 0x8035 and the *Operation Code* field is 3, designating an RARP request. As mentioned previously, RARP requests will be sent periodically (every **NX_RARP_UPDATE_RATE** seconds) until a RARP reply with the network assigned IP address is received.

*All RARP messages in the TCP/IP implementation are expected to be in **big endian** format. In this format, the most significant byte of the word resides at the lowest byte address.*

RARP Reply

RARP reply messages are received from the network and contain the network assigned IP address for this host. The format of an RARP reply packet is almost identical to the ARP packet shown in Figure 8. The only difference is the frame type field is 0x8035 and the *Operation Code* field is 4, which designates an RARP reply. After received, the IP address is setup in the IP instance, the periodic RARP request is disabled, and the IP instance is now ready for normal network operation.

For multihome hosts, the IP address is applied to the requesting network interface. If there are other network interfaces still requesting an IP address assignment, the periodic RARP service continues until all interface IP address requests are resolved.

*The application should not use the IP instance until the RARP processing is complete. The **nx_ip_status_check** may be used by threads to wait for the RARP completion. For multihome hosts, the application should not use the requesting interface until the RARP processing is complete on that interface. Secondary interface IP address status can be checked with the **nx_ip_interface_status_check** service.*

RARP Statistics and Errors

If enabled, the NetX Duo RARP software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP's RARP processing:

Total RARP Requests Sent
Total RARP Responses Received
Total RARP Invalid Messages

All these statistics and error reports are available to the application with the ***nx_rarp_info_get*** service.

IPv6 in NetX Duo

Enabling IPv6 in NetX Duo

By default IPv6 is enabled in NetX Duo. IPv6 services are enabled in NetX Duo if the configurable option NX_DISABLE_IPV6 in *nx_user.h* is not defined. If NX_DISABLE_IPV6 is defined, NetX Duo will only offer IPv4 services.

The following service is provided for applications to configure the device IPv6 address:

nxd_ipv6_address_set

In addition to manually setting the device's IPv6 addresses, the system may also use Stateless Address Auto Configuration. To use this option, the application must call ***nxd_ipv6_enable*** to start IPv6 services on the device. In addition, ICMPv6 services must be started by calling ***nxd_icmp_enable***, which enables NetX Duo to perform services such as Router Solicitation, Neighbor Discovery, and Duplicate Address Detection. Note that ***nx_icmp_enable*** only starts ICMP for IPv4 services. ***nxd_icmp_enable*** starts ICMP services on both IPv4 and IPv6. If the system does not need ICMPv6 services, then ***nx_icmp_enable*** can be used so the ICMPv6 module is not linked into the system.

The following example shows a typical NetX Duo IPv6 initialization procedure.

```
/* Assume ip_0 has been created and IPv4 services (such as ARP,
ICMP, have been enabled. */

/* Enable IPv6 */
status = nxd_ipv6_enable(&ip_0);

if(status != NX_SUCCESS)
{
    /* nxd_ipv6_enable failed. */
}

/* Enable ICMPv6 */
status = nxd_icmp_enable(&ip_0);
if(status != NX_SUCCESS)
{
    /* nxd_icmp_enable failed. */
}

/* Configure the link local address on the primary interface. */
status = nxd_ipv6_address_set(&ip_0, 0, NX_NULL, 10, NX_NULL);

/* Configure ip_0 primary interface global address. */
ip_address.nxd_ip_version = NX_IP_VERSION_V6
ip_address.nxd_ip_address.v6[0] = 0x20010db8;
ip_address.nxd_ip_address.v6[1] = 0x0000f101;
ip_address.nxd_ip_address.v6[2] = 0;
ip_address.nxd_ip_address.v6[3] = 0x202;

/* Configure global address of the primary interface. */
status = nxd_ipv6_address_set(&ip_0, 0, &ip_address, 64, NX_NULL);
```

Upper layer protocols (such as TCP and UDP) can be enabled either before or after IPv6 starts.

IPv6 services are available only after IP thread is initialized and the device is enabled.

After the interface is enabled (i.e.,the interface device driver is ready to send and receive data, and a valid link local address has been obtained), the device may obtain global IPv6 addresses by one of the two methods:

- Stateless Address Auto Configuration;
- Manual IPv6 address configuration;

Both of these methods are described below.

Stateless Address Auto Configuration Using Router Solicitation

NetX Duo devices can configure their interfaces automatically when connected to an IPv6 network with a router that supplies prefix information. Devices that require Stateless Address Auto Configuration send out router solicitation (RS) messages. Routers on the network respond with solicited router advertisement (RA) messages. RA messages advertise prefixes that identify the network addresses associated with a link. Devices then generate a unique identifier for the network the interface is attached to. The address is formed by combining the prefix and its unique identifier. In this manner on receiving the RA messages, hosts generate their IP address. Routers may also send periodic unsolicited RA messages.

Manual IPv6 Address Configuration

If a specific IPv6 address is needed, the application may use *nxd_ipv6_address_set* to manually configure an IPv6 address. An interface may have multiple IPv6 addresses. However keep in mind that the total number of IPv6 addresses in a system, either obtained through Stateless Address Auto Configuration, or through the Manual Configuration, cannot exceed *NX_MAX_IPV6_ADDRESSES*.

The following example illustrates how to manually configure a global address on the primary interface (interface 0) in ip_0:

```
NXD_ADDRESS global_address;
global_address.nxd_ip_version = NX_IP_VERSION_V6;
global_address.nxd_ip_address.v6[0] = 0x20010000;
global_address.nxd_ip_address.v6[1] = 0x00000000;
global_address.nxd_ip_address.v6[2] = 0x00000000;
global_address.nxd_ip_address.v6[3] = 0x0000ABCD;
```

The host then calls the following NetX Duo service to assign this address as its global IP address:

```
status = nxd_ipv6_address_set(&ip_0, 0,
                              &global_address, 64, NX_NULL);
```

Duplicate Address Detection (DAD)

After a device configures its IPv6 address, the state of the is marked as *TENTATIVE*. If Duplicate Address Detection (DAD), described in RFC 4862, is enabled, NetX Duo automatically sends neighbor solicitation (NS) messages with this tentative address as the destination. If no hosts on the network respond to these NS messages within a given period of time, the address is assumed to be unique on the local link, and the state transits to the VALID state where the application may start using this IP address for communication.

The DAD functionality is part of the ICMPv6 module. Therefore, the application must enable ICMPv6 services before a newly configured address can go through the DAD process. Alternatively, the DAD process may be turned off by defining *NXDUO_DISABLE_DAD* option in the NetX Duo library build environment (defined as ***nx_user.h***). During the DAD process, the *NX_DUP_ADDR_DETECT_TRANSMITS* parameter determines the number of NS messages sent by NetX Duo without receiving a response to determine that the address is unique. By default and recommended by RFC 4862, NX_DUP_ADDR_DETECT_TRANSMITS is set at 3. Setting this symbol to zero effectively disables DAD.

If ICMPv6 or DAD is not enabled at the time the application assigns an IPv6 address, DAD is not performed and NetX Duo sets the state of the IPv6 address to VALID immediately.

NetX Duo cannot communicate on the IPv6 network until its link local and/or global address is valid. After a valid address is obtained, NetX Duo attempts to match the destination address of an incoming packet against one of its interface addresses. If no matches are found, the packet is dropped.

During the DAD process, the number of DAD NS packets to be transmitted is defined by NX_DUP_ADDR_DETECT_TRANSMITS, which defaults to 3, and by default there is a one second delay between each DAD NS message is sent. Therefore, in a system with DAD enabled, after an IPv6 address is assigned (and assuming this is not a duplicated address), there is approximately 3 seconds delay before the IP address is in a VALID state and is ready for communication.

Multicast Support In NetX Duo

Multicast addresses specify a dynamic group of hosts on the Internet. Members of the multicast group may join and leave whenever they wish. NetX Duo implements several ICMPv6 protocols, including Duplicate Address Detection, Neighbor Discovery, and Router Discovery, which require IP multicast capability. Therefore, NetX Duo expects the underlying Ethernet driver to support multicast operations.

When NetX Duo needs to join or leave a multicast group (such as the all-node multicast address, and the *solicited-node* multicast address), it issues a driver command to the Ethernet driver to join or leave a multicast MAC address. The driver command for joining the multicast address is NX_LINK_MULTICAST_JOIN. To leave a multicast address, NetX Duo issues the driver command NX_LINK_MULTICAST_LEAVE. Ethernet driver must implement these two commands for ICMPv6 protocols to work properly.

Neighbor Discovery (ND)

Neighbor Discovery is a protocol in IPv6 networks for mapping physical addresses to the IPv6 addresses (global address or link-local address). This mapping is maintained in the Neighbor Discovery Cache (ND Cache). The ND process is the equivalent of the ARP process in IPv4, and the ND cache is similar to the ARP table. An IPv6 node can obtain its neighbor's MAC address using the Neighbor Discovery (ND) protocol. It sends out a neighbor solicitation (NS) message to the all-node solicited node multicast address, and waits for a corresponding neighbor advertisement (NA) message. The MAC address obtained through this process is stored in the ND Cache.

Each IP instance has one ND cache. The ND Cache is maintained as an array of entries. The size of the array is defined at compilation time by setting the option *NX_IPV6_NEIGHBOR_CACHE_SIZE* which in ***nx_nd_cache.h***. Note that all interfaces attached to an IP instance share the same ND cache.

The entire ND Cache is empty when a NetX Duo application starts up. As the system runs, NetX Duo automatically updates the ND Cache, adding and deleting entries as per ND protocol. However, an application may also update the ND Cache by manually adding and deleting cache entries using the following NetX Duo services:

nxd_nd_cache_entry_delete
nxd_nd_cache_entry_set
nxd_nd_cache_invalidate

When sending and receiving IPv6 packets, NetX Duo automatically updates the ND Cache table.

Internet Control Message Protocol (ICMP)

Internet Control Message Protocol for IPv4 (ICMP) is limited to passing error and control information between IP network members. Internet Control Message Protocol for IPv6 (ICMPv6) also handles error and control information and is required for address resolution protocols such as DAD and stateless Auto Configuration.

Like most other application layer (e.g., TCP/IP) messages, ICMP and ICMPv6 messages are encapsulated by an IP header with the ICMP (or ICMPv6) protocol designation.

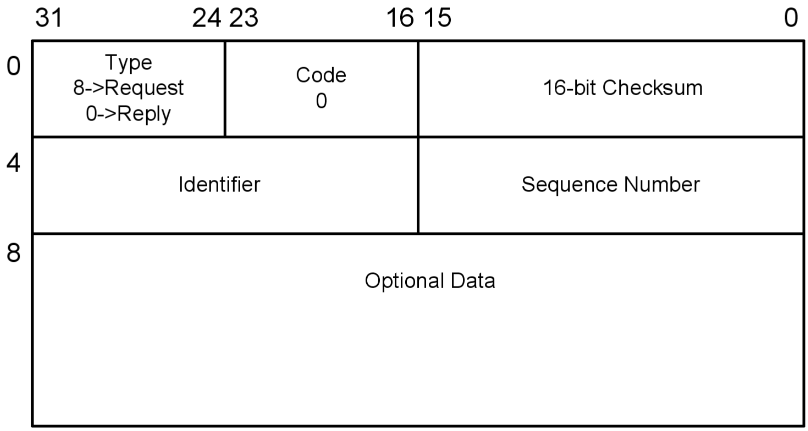
ICMPv4 Services in NetX Duo

ICMPv4 Enable

Before ICMPv4 messages can be processed by NetX Duo, the application must call the ***nx_icmp_enable*** service to enable ICMPv4 processing. After this is done, the application can issue ping requests and field ping responses.

Ping Request

A ping request is one type of ICMPv4 message that is typically used to check for the existence of a specific member on the network, as identified by a host IP address. If the specific host is present, its ICMPv4 component processes the ping request by issuing a ping response. Figure 9 details the ICMPv4 ping message format.



(Note: IP header is prepended)

FIGURE 9. ICMPv4 Ping Message

*All ICMPv4 messages in the TCP/IP implementation are expected to be in **big endian** format. In this format, the most significant byte of the word resides at the lowest byte address.*

The following describes the ICMPv4 header format:

Header Field	***Purpose***
Type	This field specifies the ICMPv4 message (bits 31-28). The most common are: 0 Echo Reply 3 Destination Unreachable 8 Echo Request
Code	This field is context specific on the type field (bits 27-24). For an echo request or reply the code is set to zero

Checksum	This field contains the 16-bit checksum of the one's complement sum of the ICMPv4 message including the entire the ICMPv4 header starting with the Type field. Before generating the checksum, the checksum field is cleared.
Identification	This field contains an ID value identifying the host; a host should use the ID extracted from an ECHO request in the ECHO REPLY (bits 31-16)
Sequence number	This field contains an ID value; a host should use the ID extracted from an ECHO request in the ECHO REPLY (bits 31-16). Unlike the identifier field, this value will change in a subsequent Echo request from the same host (bits 15-0)

Ping Response

A ping response is another type of ICMP message that is generated internally by the ICMP component in response to an external ping request. In addition to acknowledgement, the ping response also contains a copy of the user data supplied in the ping request.

ICMPv6 Services in NetX Duo

The role of ICMPv6 in IPv6 has been greatly expanded to support IPv6 address mapping and router discovery. In addition, NetX Duo ICMPv6 supports echo request and response, ICMPv6 error report, and ICMPv6 redirect messages.

ICMPv6 Enable

Before ICMPv6 messages can be processed by NetX Duo, the application must call the ***nxd_icmp_enable*** service to enable ICMPv6 processing as explained previously.

ICMPv6 Messages

The ICMPv6 header structure is similar to the ICMPv4 header structure. As shown below, the basic ICMPv6 header contains the three fields, type, code, and checksum, plus variable length of ICMPv6 option data

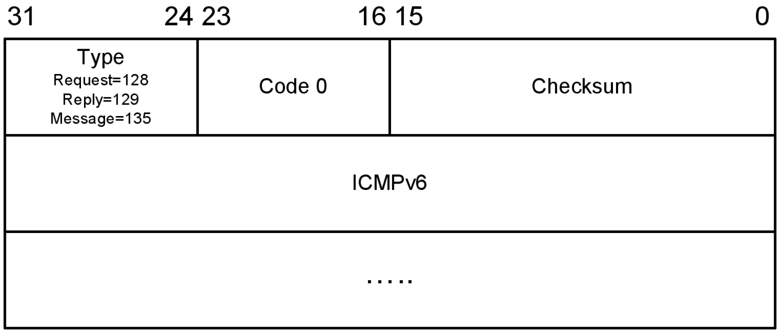


FIGURE 10. Basic ICMPv6 Header

Field	Size (bytes)	Description
Type	1	Identifies the ICMPv6 message type; e.g., either an echo request (128), an echo reply (129), or a Neighbor Solicitation message (135).
Code	1	Further qualifies the ICMPv6 message type. Generally used with error messages. If not used, it is set to zero. Echo request/reply and NS messages do not use it,
Checksum	2	16-bit checksum field for the ICMP Header. This is a 16-bit complement of the entire ICMPv6 message, including the ICMPv6 header. It also includes a pseudo-header of the IPv6 source address, destination address, and packet payload length.

An example Neighbor Solicitation header is shown below.

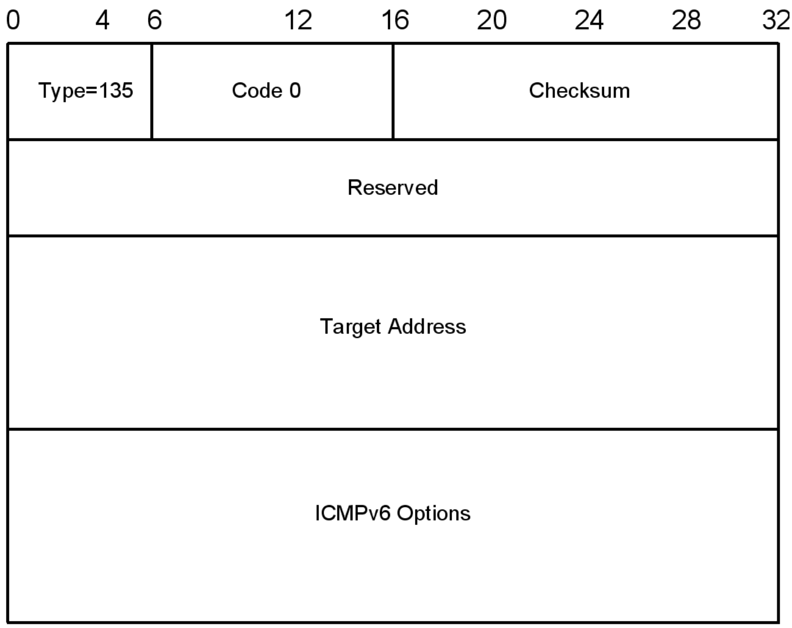


FIGURE 11. ICMPv6 Header for a Neighbor Solicitation Message

Field	Size (bytes)	Description
Type	1	Identifies the ICMPv6 message type for neighbor solicitation messages. Value is 135.
Code	1	Not used. Set to 0.
Checksum	2	16-bit checksum field for the ICMPv6 header.
Reserved	4	4 reserved bytes set to 0.

Field	Size (bytes)	Description
Target Address	16	IPv6 address of target of the solicitation. For IPv6 address resolution, this is the actual unicast IP address of the device whose link layer address needs to be resolved.
Options	Variable	Optional information specified by the Neighbor Discovery Protocol.

ICMPv6 Ping Request Message Type

In NetX Duo applications use ***nxd_icmp_ping*** to issue either IPv6 or IPv4 ping requests, based on the destination IP address specified in the parameters.

Thread Suspension

Application threads can suspend while attempting to ping another network member. After a ping response is received, the ping response message is given to the first thread suspended and that thread is resumed. Like all NetX Duo services, suspending on a ping request has an optional timeout.

ICMP Statistics and Errors

If enabled, NetX Duo keeps track of several ICMP statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP's ICMP processing:

Total ICMP Pings Sent
Total ICMP Ping Timeouts
Total ICMP Ping Threads Suspended
Total ICMP Ping Responses Received
Total ICMP Checksum Errors
Total ICMP Unhandled Messages

All these statistics and error reports are available to the application with the ***nx_icmp_info_get*** service.

Other ICMPv6 Message Types

ICMPv6 messages are required for the following features:

- Neighbor Discovery
- Stateless Address Auto Configuration
- Router Discovery
- Neighbor Unreachability Detection

Neighbor Unreachability, Router and Prefix Discovery

Neighbor Unreachability Detection, Router Discovery, and Prefix Discovery are based on the Neighbor Discovery protocol and are described below.

Neighbor Unreachability Detection: An IPv6 device searches its Neighbor (ND) Cache for the destination link layer address when it wishes to send a packet. The immediate destination, sometimes referred to as the 'next hop,' may be the actual destination on the same link or it may be a router if the destination is off link. An ND cache entry contains the status on a neighbor's reachability.

A REACHABLE status indicates the neighbor is considered reachable. A neighbor is reachable if it has recently received confirmation that packets sent to the neighbor have been received. Confirmation in NetX Duo take the form of receiving an NA message from the neighbor in response to an NS message sent by the NetX Duo device. NetX Duo will also change the state of the neighbor status to REACHABLE if the application calls the NetX Duo service ***_nxd_nd_cache_entry_set*** to manually enter a cache record.

Router Discovery: An IPv6 device uses a router to forward all packets intended for off link destinations. It may also use information sent by the router, such as router advertisement (RA) messages, to configure its global IPv6 addresses.

A device on the network may initiate the Router Discovery process by sending a router solicitation (RS) message to the all-router multicast address (FF01::2). Or it can wait on the all-node multicast address (FF::1) for a periodic RA from the routers.

An RA message contains the prefix information for configuring an IPv6 address for that network. In NetX Duo, router solicitation is by default enabled and can be disabled by setting the configuration option NX_DISABLE_ICMPV6_ROUTER_SOLICITATION in ***nx_user.h***. See Configuration Options in the “Installation and Use of NetX Duo” chapter for more details on setting Router Solicitation parameters.

Prefix Discovery: An IPv6 device uses prefix discovery to learn which target hosts are accessible directly without going through a router. This information is made available to the IPv6 device from RA messages from the router. The IPv6 device stores the prefix information in a prefix table. Prefix discovery is matching a prefix from the IPv6 device prefix table to a target address. A prefix matches a target address if all the bits in the prefix match the most significant bits of the target address. If more than one prefix covers an address, the longest prefix is selected.

Internet Group Management Protocol (IGMP)

The Internet Group Management Protocol (IGMP) provides UDP packet delivery to multiple network members that belong to the same multicast group (RFC 1112 and RFC 2236). A multicast group is basically a dynamic collection of network members and is represented by a Class D IP address. Members of the multicast group may leave at any time, and new members may join at any time. The coordination involved in joining and leaving the group is the responsibility of IGMP.

IGMP Enable

Before any multicasting activity can take place in NetX Duo, the application must call the ***nx_igmp_enable*** service. This service performs basic IGMP initialization in preparation for multicast requests.

Multicast IP Addressing in IPv4

As mentioned previously, multicast addresses are actually Class D IP addresses as shown in Figure 5 on page 66. The lower 28-bits of the Class D address correspond to the multicast group ID. There are a series of pre-defined multicast addresses; however, the *all hosts address* (244.0.0.1) is particularly important to IGMP processing. The *all hosts address* is used by routers to query all multicast members to report on which multicast groups they belong to.

Physical Address Mapping in IPv4

Class D multicast addresses map directly to physical Ethernet addresses ranging from 01.00.5e.00.00.00 through 01.00.5e.7f.ff.ff. The lower 23 bits of the IP multicast address map directly to the lower 23 bits of the Ethernet address.

Multicast Group Join

Applications that need to join a particular multicast group may do so by calling the ***nx_igmp_multicast_join*** service. This service keeps track of the number of requests to join this multicast group. If this is the first application request to join the multicast group, an IGMP report is sent out on the network indicating this host's intention to join the group. Next, the network driver is called to set up for listening for packets with the Ethernet address for this multicast group.

For multihome hosts, the ***nx_igmp_multicast_interface_join*** service should be used instead of *nx_igmp_multicast_join,* if the multicast group destination address is on a secondary network interface. The original service *nx_igmp_multicast_join* service is limited to multicast groups on the primary network and is included for backward compatibility.

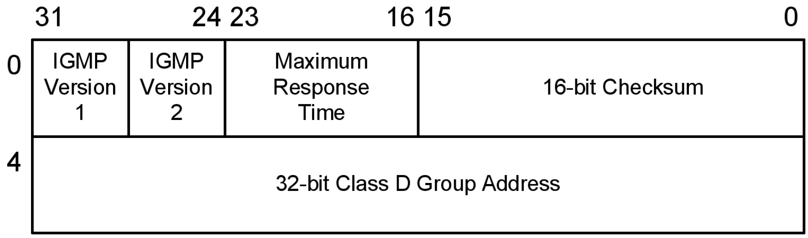
Multicast Group Leave

Applications that need to leave a previously joined multicast group may do so by calling the ***nx_igmp_multicast_leave*** service. This service reduces the internal count associated with how many times the group was joined. If there are no outstanding join requests for a group, the network driver is called to disable listening for packets with this multicast group's Ethernet address.

IGMP Report Message

When the application joins a multicast group, an IGMP report message is sent via the network to indicate the host's intention to join a particular multicast group. The format of the IGMP report message is shown in Figure 12. The multicast group address is used for both the group message in the IGMP report message and the destination IP address.

In the figure above (Figure 12), the IGMP header contains a version field, a type field, a checksum field, and a multicast group address field. For IGMPv1 messages, the Maximum Response Time field is always set to zero, as this is not part of the IGMPv1 protocol. The Maximum Response Time field is set when the host receives a Query type IGMP message and cleared when a host receives another



(Note: IP header is prepended)

FIGURE 12. IGMP Report Message

hosts Report type message as defined by the IGMPv2 protocol.

The following describes the IGMP header format:

Header Field	***Purpose***
Version	This field specifies the IGMP version (bits 31- 28)
Type	This field specifies the type of IGMP message (bits 27 -24)
Identifier	Not used in IGMP v1. In IGMP v2 this field serves as the maximum response time.
Checksum	This field contains the 16-bit checksum of the one's complement sum of the IGMP message starting with the IGMP version (bits 0-15)
Group Address	32-bit class D group IP address

IGMP report messages are also sent in response to IGMP query messages sent by a multicast router. Multicast routers periodically send query messages out to see which hosts still require group membership. Query messages have the same format as the IGMP report message shown in Figure 12.

The only differences are the IGMP type is equal to 1 and the group address field is set to 0. IGMP Query messages are sent to the *all hosts* IP address by the multicast router. A host that still wishes to maintain group membership responds by sending another IGMP report message.

*All messages in the TCP/IP implementation are expected to be in **big endian** format. In this format, the most significant byte of the word resides at the lowest byte address.*

IGMP Statistics and Errors

If enabled, the NetX Duo IGMP software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP's IGMP processing:

Total IGMP Reports Sent
Total IGMP Queries Received
Total IGMP Checksum Errors
Total IGMP Current Groups Joined

All these statistics and error reports are available to the application with the ***nx_igmp_info_get*** service.

User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) provides the simplest form of data transfer between network members (RFC 768). UDP data packets are sent from one network member to another in a best effort fashion; i.e., there is no built-in mechanism for acknowledgement by the packet recipient. In addition, sending a UDP packet does not require any connection to be established in advance. Because of this, UDP packet transmission is very efficient.

For developers migrating their NetX applications to NetX Duo there are only a few basic changes in UDP functionality between NetX and NetX Duo. This is because IPv6 is primarily concerned with the underlying IP layer. All NetX Duo UDP services can be used for either IPv4 or IPv6 connectivity.

UDP Enable

Before UDP packet transmission is possible, the application must first enable UDP by calling the ***nx_udp_enable*** service. After enabled, the application is free to send and receive UDP packets.

UDP Header

UDP places a simple packet header in front of the application's data when sending application data and removes a similar UDP header from the packet before delivering a received UDP packet to the application. UDP utilizes the IP protocol for sending and receiving packets, which means there is an IP header in front of the UDP header when the packet is on the network. Figure 13 shows the format of the UDP header.

31 16	15 0
16-bit Source Port	16-bit Destination Port
16-bit UDP Length	16-bit UDP Checksum

(Note: IP header is prepended)

FIGURE 13. UDP Header

*All headers in the UDP/IP implementation are expected to be in **big endian** format. In this format,* *the most significant byte of the word resides at the lowest byte address.*

The following describes the UDP header format:

Header Field	Purpose
16-bit source port number	This field contains the port on which the UDP packet is being sent. Valid UDP ports range from 1 through 0xFFFF.
16-bit destination port number	This field contains the UDP port to which the packet is being sent. Valid UDP ports range from 1 through 0xFFFF.
16-bit UDP length	This field contains the number of bytes in the UDP packet, including the size of the UDP header.
16-bit UDP checksum	This field contains the 16-bit checksum for the packet, including the UDP header, the packet data area, and the pseudo IP header.

UDP Checksum

IPv6 protocol requires a UDP header checksum computation on packet data, whereas in the IPv4 protocol it is optional.

UDP specifies a one's complement 16-bit checksum that covers the IP pseudo header (consisting of the source IP address, destination IP address, and the protocol/length IP word), the UDP header, and the UDP packet data. The only differences between IPv4 and IPv6 UDP packet header checksums is that the source and destination IP addresses are 32 bit in IPv4 while in IPv6 they are 128 bit. If the calculated UDP checksum is 0, it is stored as all ones (0xFFFF). If the sending socket has the UDP checksum logic disabled, a zero is placed in the UDP checksum field to indicate the checksum was not calculated.

If the UDP checksum does not match the computed checksum by the receiver, the UDP packet is simply discarded.

NetX Duo allows the application to enable or disable UDP checksum calculation on a per-socket basis. By default, the UDP socket checksum logic is enabled. The application can disable checksum logic for a particular UDP socket by calling the ***nx_udp_socket_checksum_disable***.

On IPv6 networks, a UDP packet checksum is mandatory. However, it need not be calculated by NetX Duo, for example if the network interface is able to compute the checksum in hardware. Checksum calculation can be disabled in NetX Duo by either using the ***nx_udp_socket_checksum_disable*** service or defining (enabling) the NX_DISABLE_UDP_TX_CHECKSUM and NX_DISABLE_UDP_RX_CHECKSUM configuration options (described Chapter two). The configuration options remove UDP checksum logic from NetX Duo entirely, while calling ***nx_udp_socket_checksum_disable*** allows the application to disable UDP checksum processing on a per socket basis.

UDP Ports and Binding

A UDP port is a logical end point in the UDP protocol. There are 65,535 valid ports in the UDP component of NetX Duo, ranging from 1 through 0xFFFF. To send or receive UDP data, the application must first create a UDP socket, then bind it to a desired port. After binding a UDP socket to a port, the application may send and receive data on that socket.

UDP Fast Path™

The UDP Fast Path™ is the name for a low packet overhead path through the NetX Duo UDP implementation. Sending a UDP packet requires just three function calls: ***nx_udp_socket_send***, ***nx_ip_socket_send***, and the eventual call to the network driver. ***nx_udp_socket_send*** is available in NetX Duo for existing NetX applications and is only applicable for IPv4 connections. The preferred method, however, is to use ***nxd_udp_socket_send*** service discussed below. On UDP packet reception, the UDP packet is either placed on the appropriate UDP socket receive queue or delivered to a suspended application thread in a single function call from the network driver's receive interrupt processing. This highly optimized logic for sending and receiving UDP packets is the essence of UDP Fast Path technology.

UDP Packet Send

Sending UDP data over IPv6 or IPv4 networks is easily accomplished by calling the ***nxd_udp_socket_send*** function. The caller must set the IP version in the *nx_ip_version* field of the NXD_ADDRESS pointer parameter. For IPv6 packets, NetX Duo will determine the correct source address for transmitted UDP packets based on the destination IPv6 address. For IPv4 networks, it will derive the source address from the packet interface. This service places a UDP header in front of the packet data and sends it out onto the network using an internal IP send routine. There is no thread suspension on sending UDP packets because all UDP packet transmissions are processed immediately.

For multicast destinations, the application must specify the source IP address to use if the NetX Duo device has multiple IP addresses to choose from. Also, sending a datagram to an IPv4 multicast or broadcast address from a multihome device would require the use the ***nxd_udp_socket_interface_send*** service. Similarly, a system with multiple IPv6 address should use the

nxd_udp_socket_interface_send service if there are more than one suitable source address to choose from.

If UDP checksum logic is enabled for this socket, the checksum operation is performed in the context of the calling thread, without blocking access to the UDP or IP data structures.

The UDP data residing in the NX_PACKET structure should reside on a long-word boundary. The application needs to leave sufficient space between the prepend pointer and the data start pointer for NetX Duo to place the UDP, IP, and physical media headers.

UDP Packet Receive

Application threads may receive UDP packets from a particular socket by calling ***nx_udp_socket_receive***. The socket receive function delivers the oldest packet on the socket's receive queue. If there are no packets on the receive queue, the calling thread can suspend (with an optional timeout) until a packet arrives.

The UDP receive packet processing (usually called from the network driver's receive interrupt handler) is responsible for either placing the packet on the UDP socket's receive queue or delivering it to the first suspended thread waiting for a packet. If the packet is queued, the receive processing also checks the maximum receive queue depth associated with the socket. If this newly queued packet exceeds the queue depth, the oldest packet in the queue is discarded.

UDP Receive Notify

If the application thread needs to process received data from more than one socket, the ***nx_udp_socket_receive_notify*** function should be used. This function registers a receive packet callback function for the socket. Whenever a packet is received on the socket, the callback function is executed.

The contents of the callback function is application-specific; however, it would most likely contain logic to inform the processing thread that a packet is now available on the corresponding socket.

UDP Socket Create

UDP sockets are created either during initialization or during runtime by application threads. The initial type of service, time to live, and receive queue depth are defined by the ***nx_udp_socket_create*** service. There are no limits on the number of UDP sockets in an application.

Thread Suspension

As mentioned previously, application threads can suspend while attempting to receive a UDP packet on a particular UDP port. After a packet is received on that port, it is given to the first thread suspended and that thread is then resumed. An optional timeout is available when suspending on a UDP receive packet, a feature available for most NetX Duo services.

UDP Socket Statistics and Errors

If enabled, the NetX Duo UDP socket software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP/UDP instance:

Total UDP Packets Sent
Total UDP Bytes Sent
Total UDP Packets Received
Total UDP Bytes Received
Total UDP Invalid Packets
Total UDP Receive Packets Dropped
Total UDP Receive Checksum Errors
UDP Socket Packets Sent

UDP Socket Bytes Sent
UDP Socket Packets Received
UDP Socket Bytes Received
UDP Socket Packets Queued
UDP Socket Receive Packets Dropped
UDP Socket Checksum Errors

All these statistics and error reports are available to the application with the ***nx_udp_info_get*** service for UDP statistics amassed over all UDP sockets, and the ***nx_udp_socket_info_get*** service for UDP statistics on the specified UDP socket.

UDP Socket Control Block NX_UDP_SOCKET

The characteristics of each UDP socket are found in the associated NX_UDP_SOCKET control block. It contains useful information such as the link to the IP data structure, the network interface for the sending and receiving paths, the bound port, and the receive packet queue. This structure is defined in the ***nx_api.h*** file.

Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) provides reliable stream data transfer between two network members (RFC 793). All data sent from one network member are verified and acknowledged by the receiving member. In addition, the two members must have established a connection prior to any data transfer. All this results in reliable data transfer; however, it does require substantially more overhead than the previously described UDP data transfer.

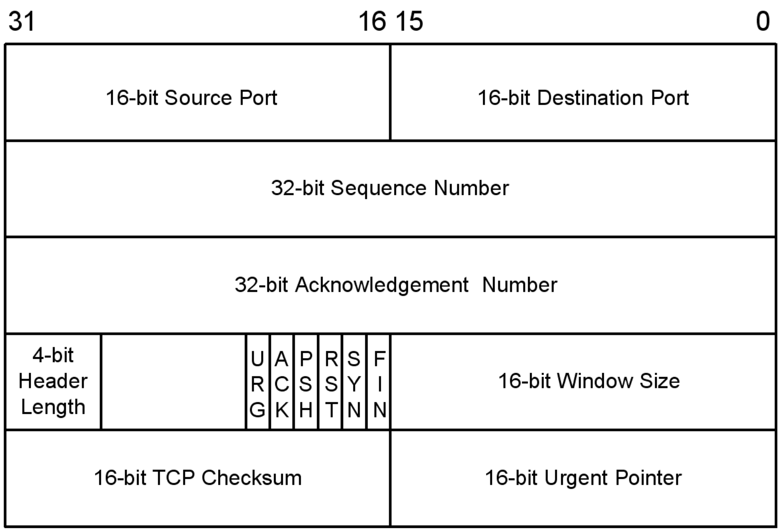
Except where noted, there are no changes in TCP protocol API services between NetX and NetX Duo because IPv6 is primarily concerned with the underlying IP layer. All NetX Duo TCP services can be used for either IPv4 or IPv6 connections.

TCP Enable

Before TCP connections and packet transmissions are possible, the application must first enable TCP by calling the ***nx_tcp_enable*** service. After enabled, the application is free to access all TCP services.

TCP Header

TCP places a somewhat complex packet header in front of the application's data when sending data and removes a similar TCP header from the packet before delivering a received TCP packet to the application. TCP utilizes the IP protocol to send and receive packets, which means there is an IP header in front of the TCP header when the packet is on the network. Figure 14 shows the format of the TCP header.



(Note: IP Header is prepended)

FIGURE 14. TCP Header

The following describes the TCP header format:

Header Field	Purpose
16-bit source port number	This field contains the port the TCP packet is being sent out on. Valid TCP ports range from 1 through 0xFFFF.
16-bit destination port number	This field contains the TCP port the packet is being sent to. Valid TCP ports range from 1 through 0xFFFF.
32-bit sequence number	This field contains the sequence number for data sent from this end of the connection. The original sequence is established during the initial connection sequence between two TCP nodes. Every data transfer from that point results in an increment of the sequence number by the amount bytes sent.
32-bit acknowledgement number	This field contains the sequence number corresponding to the last byte received by this side of the connection. This is used to determine whether or not data previously sent has successfully been received by the other end of the connection.
4-bit header length	This field contains the number of 32-bit words in the TCP header. If no options are present in the TCP header, this field is 5.

Header Field	Purpose
6-bit code bits	This field contains the six different code bits used to indicate various control information associated with the connection. The control bits are defined as follows:

Name	Bit	Meaning
URG	21	Urgent data present
ACK	20	Acknowledgement number is valid
PSH	19	Handle this data immediately
RST	18	Reset the connection
SYN	17	Synchronize sequence numbers (used to establish connection)
FIN	16	Sender is finished with transmit (used to close connection)

Header Field	Purpose
16-bit window	This field contains the amount of bytes the sender can currently receive. This basically is used for flow control. The sender is responsible for making sure the data to send will fit into the receiver's advertised window.
16-bit TCP checksum	This field contains the 16-bit checksum for the packet including the TCP header, the packet data area, and the pseudo IP header.
16-bit urgent pointer	This field contains the positive offset of the last byte of the urgent data. This field is only valid if the URG code bit is set in the header.

*All headers in the TCP/IP implementation are expected to be in **big endian** format. In this format, the most significant byte of the word resides at the lowest byte address.*

TCP Checksum

TCP specifies a one's complement 16-bit checksum that covers the IP pseudo header, (consisting of the source IP address, destination IP address, and the protocol/length IP word), the TCP header, and the TCP packet data. The only difference between IPv4 and IPv6 TCP packet header checksums is that the source and destination IP addresses are 32 bit in IPv4 and 128 bit in IPv6.

TCP Ports

A TCP port is a logical connection point in the TCP protocol. There are 65,535 valid ports in the TCP component of NetX Duo, ranging from 1 through 0xFFFF. Unlike UDP in which data from one port can be sent to any other destination port, a TCP port is connected to another specific TCP port, and only when this connection is established can any data transfer take place—and only between the two ports making up the connection.

TCP ports are completely separate from UDP ports; e.g., UDP port number 1 has no relation to TCP port number 1.

Client Server Model

To use TCP for data transfer, a connection must first be established between the two TCP sockets. The establishment of the connection is done in a client-server fashion. The client side of the connection is the side that initiates the connection, while the server side simply waits for client connection requests before any processing is done.

For multihome devices, NetX Duo automatically determines the interface and next hop address on the client side for transmitting packets based on the packet destination IP address. Because TCP is limited to sending packets to unicast (e.g.non-broadcast) destination addresses, NetX Duo does not require a "hint" for choosing the outgoing interface.

TCP Socket State Machine

The connection between two TCP sockets (one client and one server) is complex and is managed in a state machine manner. Each TCP socket starts in a CLOSED state. Through connection events each socket's state machine migrates into the ESTABLISHED state, which is where the bulk of the data transfer in TCP takes place. When one side of the connection no longer wishes to send data, it disconnects, and this action eventually causes both TCP sockets to return to the CLOSED state. This process repeats each time a TCP client and server establish and close a connection. Figure 15 on page 124 shows the various states of the TCP state machine.

TCP Client Connection

As mentioned previously, the client side of the TCP connection initiates a connection request to a TCP server. Before a connection request can be made, TCP must be enabled on the client IP instance. In addition, the client TCP socket must next be created with ***nx_tcp_socket_create*** service and bound to a port via the ***nx_tcp_client_socket_bind*** service.

After the client socket is bound, the ***nxd_tcp_client_socket_connect*** service is used to establish a connection with a TCP server. Note the socket must be in a CLOSED state to initiate a connection attempt. Establishing the connection starts with NetX Duo issuing a SYN packet and then waiting for a SYN ACK packet back from the server, which signifies acceptance of the connection request. After the SYN ACK is received, NetX Duo responds with an ACK packet and promotes the client socket to the ESTABLISHED state.

Applications should use ***nxd_tcp_client_socket_connect*** *for either IPv4 and IPv6 TCP connections. Applications can still use* ***nx_tcp_client_socket_connect*** *for IPv4 TCP*

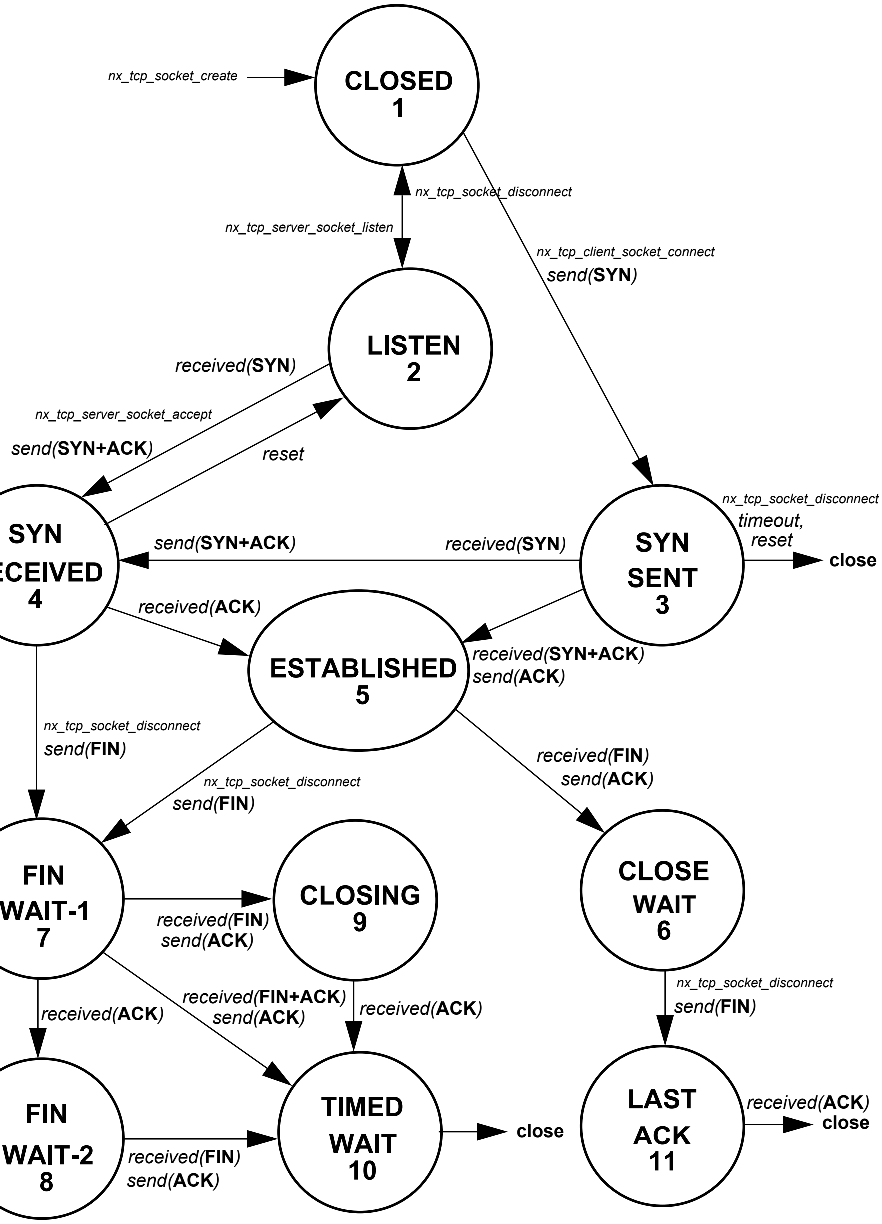


FIGURE 15. States of the TCP State Machine

connections, but developers are encouraged to use ***nxd_tcp_client_socket_connect*** *since* ***nx_tcp_client_socket_connect*** *will eventually be deprecated.*

Similarly, ***nxd_tcp_socket_peer_info_get*** *works with either IPv4 or IPv6 TCP connections. However,* ***nx_tcp_socket_peer_info_get*** *is still available for legacy applications. Developers are encouraged to use* ***nxd_tcp_socket_peer_info_get*** *since the IPv4*

TCP Client Disconnection

Closing the connection is accomplished by calling ***nx_tcp_socket_disconnect***. If no suspension is specified, the client socket sends a RST packet to the server socket and places the socket in the CLOSED state. Otherwise, if a suspension is requested, the full TCP disconnect protocol is performed, as follows:

- If the server previously initiated a disconnect request (the client socket has already received a FIN packet, responded with an ACK, and is in the CLOSE WAIT state), NetX Duo promotes the TCP socket state to the LAST ACK state and sends a FIN packet. It then waits for an ACK from the server before completing the disconnect and entering the CLOSED state.
- If on the other hand, the client is the first to initiate a disconnect request (the server has not disconnected and the socket is still in the ESTABLISHED state), NetX Duo sends a FIN packet to initiate the disconnect and waits to receive a FIN and an ACK from the server before completing the disconnect and placing the socket in a CLOSED state.

If there are still packets on the socket transmit queue, NetX Duo suspends for the specified timeout to allow the packets to be acknowledged. If the timeout expires, NetX Duo empties the transmit queue of the client socket.

To unbind the port from the client socket, the application calls ***nx_tcp_client_socket_unbind***. The socket must be in a CLOSED state or in the process of disconnecting (i.e., CLOSE WAIT state) before the port is released; otherwise, an error is returned.

Finally, if the application no longer needs the client socket, it calls ***nx_tcp_socket_delete*** to delete the socket.

TCP Server Connection

The server side of a TCP connection is passive; i.e., the server waits for a client connection request. To accept a client connection, TCP must first be enabled on the IP instance. Next, the application must create a TCP socket using the ***nx_tcp_socket_create*** service.

The server socket must also be set up for listening for connection requests using the ***nx_tcp_server_socket_listen*** service. This service places the server socket in the LISTEN state and binds the specified server port to the server socket. If the socket connection has already been established, the function simply returns a successful status.

i

To set a socket listen callback routine the application specifies the appropriate callback function for the tcp_listen_callback argument of the ***nx_tcp_server_socket_listen*** *service. This application callback function is then executed by NetX Duo whenever a new connection is requested on this server port. The processing in the callback is under application control.*

To accept client connection requests, the application calls the ***nx_tcp_server_socket_accept*** service. The server socket must either be in a LISTEN state or a SYN RECEIVED state (i.e., the server has received a SYN packet from a client requesting a connection) to call the accept service. A successful return status from ***nx_tcp_server_socket_accept*** indicates the connection has been established and the server socket is in the ESTABLISHED state.

After the server socket has a valid connection, additional client connection requests are queued up to the depth specified by the ***nx_tcp_server_socket_listen*** service. In order to process subsequent connections on a server port, the application must call ***nx_tcp_server_socket_relisten*** with an available socket (i.e., a socket in a CLOSED state). Note that the same server socket could be used if the previous connection associated with the socket is now finished and the socket is in the CLOSED state.

TCP Server Disconnection

Closing the connection is accomplished by calling ***nx_tcp_socket_disconnect***. If no suspension is specified, the server socket sends a RST packet to the client socket and places the socket in the CLOSED state. Otherwise, if a suspension is requested, the full TCP disconnect protocol is performed, as follows:

- If the client previously initiated a disconnect request (the server socket has already received a FIN packet, responded with an ACK, and is in the CLOSE WAIT state), NetX Duo promotes the TCP socket state to the LAST ACK state and sends a FIN packet. It then waits for an ACK from the client before completing the disconnect and entering the CLOSED state.
- If on the other hand, the server is the first to initiate a disconnect request (the client has not disconnected and the socket is still in the ESTABLISHED state), NetX Duo sends a FIN packet to initiate the disconnect and waits to

receive a FIN and an ACK from the client before completing the disconnect and placing the socket in a CLOSED state.

If there are still packets on the socket transmit queue, NetX Duo suspends for the specified timeout to allow those packets to be acknowledged. If the timeout expires, NetX Duo flushes the transmit queue of the server socket.

After the disconnect processing is complete and the server socket is in the CLOSED state, the application must call the ***nx_tcp_server_socket_unaccept*** service to end the association of this socket with the server port. Note this service must be called by the application even if ***nx_tcp_socket_disconnect*** or ***nx_tcp_server_socket_accept*** return an error status. After the ***nx_tcp_server_socket_unaccept*** returns, the socket can be used as a client or server socket, or even deleted if it is no longer needed. If accepting another client connection on the same server port is desired, the ***nx_tcp_server_socket_relisten*** service should be called on this socket.

Stop Listening on a Server Port

If the application no longer wishes to listen for client connection requests on a server port that was previously specified by a call to the ***nx_tcp_server_socket_listen*** service, the application simply calls the ***nx_tcp_server_socket_unlisten*** service. This service places any socket waiting for a connection back in the CLOSED state and releases any queued client connection request packets.

TCP Window Size

During both the setup and data transfer phases of the connection, each port reports the amount of data it can handle, which is called its window size. As data are received and processed, this window size is adjusted dynamically. In TCP, a sender can only send an amount of data that is less than or equal to the amount of data specified by the receiver's window size. In essence, the window size provides flow control for data transfer in each direction of the connection.

TCP Packet Send

Sending TCP data is easily accomplished by calling the ***nx_tcp_socket_send*** function. This service first builds a TCP header in front of the packet (including the checksum calculation). If the receiver's window size is larger than the data in this packet, the packet is sent on the Internet using the internal IP send routine. Otherwise, the caller may suspend and wait for the receiver's window size to increase enough for this packet to be sent. At any given time, only one sender may suspend while trying to send TCP data.

The TCP data residing in the NX_PACKET structure should reside on a long-word boundary. In addition, there needs to be sufficient space between the prepend pointer and the data start pointer to place the TCP, IP, and physical media headers.

TCP Packet Retransmit

TCP packets sent are actually stored internally until an ACK is returned from the other side of the connection. If an ACK is not received within the timeout period, the transmit packet is re-sent and the next timeout period is increased. When an ACK is received, all packets covered by the acknowledgement number in the internal transmit sent queue are finally released.

TCP Packet Receive

The TCP receive packet processing (called from the IP helper thread) is responsible for handling various connection and disconnection actions as well as transmit acknowledge processing. In addition, the TCP receive packet processing is responsible for placing packets with receive data on the appropriate TCP socket's receive queue or delivering the packet to the first suspended thread waiting for a packet.

TCP Receive Notify

If the application thread needs to process received data from more than one socket, the ***nx_tcp_socket_receive_notify*** function should be used. This function registers a receive packet callback function for the socket. Whenever a packet is received on the socket, the callback function is executed.

The contents of the callback function are application-specific; however, the function would most likely contain logic to inform the processing thread that a packet is available on the corresponding socket.

TCP Socket Create

TCP sockets are created either during initialization or during runtime by application threads. The initial type of service, time to live, and window size are defined by the ***nx_tcp_socket_create*** service. There are no limits on the number of TCP sockets in an application.

Thread Suspension

As mentioned previously, application threads can suspend while attempting to receive data from a particular TCP port. After a packet is received on that port, it is given to the first thread suspended and that thread is then resumed. An optional timeout is available when suspending on a TCP receive packet, a feature available for most NetX Duo services.

Thread suspension is also available for connection (both client and server), client binding, and disconnection services.

TCP Socket Statistics and Errors

If enabled, the NetX Duo TCP socket software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP/TCP instance:

Total TCP Packets Sent
Total TCP Bytes Sent
Total TCP Packets Received
Total TCP Bytes Received
Total TCP Invalid Packets
Total TCP Receive Packets Dropped
Total TCP Receive Checksum Errors
Total TCP Connections
Total TCP Disconnections
Total TCP Connections Dropped
Total TCP Packet Retransmits
TCP Socket Packets Sent
TCP Socket Bytes Sent
TCP Socket Packets Received
TCP Socket Bytes Received
TCP Socket Packet Retransmits
TCP Socket Packets Queued
TCP Socket Checksum Errors
TCP Socket State
TCP Socket Transmit Queue Depth
TCP Socket Transmit Window Size
TCP Socket Receive Window Size

All these statistics and error reports are available to the application with the ***nx_tcp_info_get*** service for total TCP statistics and the ***nx_tcp_socket_info_get*** service for TCP statistics per socket.

TCP Socket Control Block NX_TCP_SOCKET

The characteristics of each TCP socket are found in the associated NX_TCP_SOCKET control block, which contains useful information such as the link to the IP data structure, the network connection interface, the bound port, and the receive packet queue. This structure is defined in the ***nx_api.h*** file.

CHAPTER 4

Description of NetX Duo Services

This chapter contains a description of all NetX Duo services in alphabetic order. Service names are designed so all similar services are grouped together. For example, all ARP services are found at the beginning of this chapter.

There are numerous new services in NetX Duo introduced to support IPv6-based protocols and operations. IPv6-enabled services in Net Duo have the prefix ***nxd**,* indicating that they are designed for IPv4 and IPv6 dual stack operation.

Existing services in NetX are fully supported in NetX Duo. NetX applications can be migrated to NetX Duo with minimal porting effort.

Note that a BSD-Compatible Socket API is available for legacy application code that cannot take full advantage of the high-performance NetX Duo API. Refer to Appendix D for more information on the BSD-Compatible Socket API.

In the “Return Values” section of each description, values in **BOLD** are not affected by the NX_DISABLE_ERROR_CHECKING option used to disable the API error checking, while values in non-bold are completely disabled. The “Allowed From” sections indicate from which each NetX Duo service can be called.

nx_arp_dynamic_entries_invalidate

Invalidate all dynamic entries in the ARP cache

Prototype

```
UINT  nx_arp_dynamic_entries_invalidate(NX_IP *ip_ptr);
```

Description

This service invalidates all dynamic ARP entries currently in the ARP cache.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful ARP cache invalidate.
NX_NOT_ENABLED	(0x14)	ARP is not enabled.
NX_PTR_ERROR	(0x07)	Invalid IP address.
NX_CALLER_ERROR	(0x11)	Caller is not a thread.

Allowed From

Threads

Preemption Possible

No

Example

```
/*  Invalidate all dynamic entries in the ARP cache. */
status =  nx_arp_dynamic_entries_invalidate(&ip_0);

/*  If status is NX_SUCCESS the dynamic ARP entries were
    successfully invalidated.  */
```

See Also

nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_dynamic_entry_set

Set dynamic ARP entry

Prototype

```
UINT  nx_arp_dynamic_entry_set(NX_IP *ip_ptr,
                               ULONG ip_address,
                               ULONG physical_msw,
                               ULONG physical_lsw);
```

Description

This service allocates a dynamic entry from the ARP cache and sets up the specified IP to physical address mapping. If a zero physical address is specified, an actual ARP request is sent to the network in order to have the physical address resolved. Also note that this entry will be removed if ARP aging is active or if the ARP cache is exhausted and this is the least recently used ARP entry.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	IP address to map.
physical_msw	Most significant word of the physical address.
physical_lsw	Least significant word of the physical address.

Return Values

NX_SUCCESS	(0x00)	Successful ARP dynamic entry set.
NX_NO_MORE_ENTRIES	(0x17)	No more ARP entries are available in the ARP cache.
NX_PTR_ERROR	(0x07)	Invalid IP instance pointer.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/*  Setup a dynamic ARP entry on the previously created IP
    Instance 0. */
status = nx_arp_dynamic_entry_set(&ip_0, IP_ADDRESS(1,2,3,4),
                                  0x0, 0x1234);

/*  If status is NX_SUCCESS, there is now a dynamic mapping between
    the IP address of 1.2.3.4 and the physical hardware address of
    0x0:0x1234. */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_enable

Enable Address Resolution Protocol (ARP)

Prototype

```
UINT nx_arp_enable(NX_IP *ip_ptr, VOID *arp_cache_memory,
                   ULONG arp_cache_size);
```

Description

This service initializes the ARP component of NetX Duo for the specific IP instance. ARP initialization includes setting up the ARP cache and various ARP processing routines necessary for sending and receiving ARP messages.

Parameters

ip_ptr	Pointer to previously created IP instance.
arp_cache_memory	Pointer to memory area to place ARP cache.
arp_cache_size	Each ARP entry is approximately 52 bytes, the total number of ARP entries is, therefore, the size divided by 52.

Return Values

NX_SUCCESS	(0x00)	Successful ARP enable.
NX_PTR_ERROR	(0x07)	Invalid IP or cache memory pointer.
NX_SIZE_ERROR	(0x09)	Invalid size of ARP cache memory.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_ALREADY_ENABLED	(0x15)	This component has already been enabled.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/*  Enable ARP and supply 1024 bytes of ARP cache memory for
    previously created IP Instance 0.  */
status =  nx_arp_enable(&ip_0, (void *) pointer, 1024);

/*  If status is NX_SUCCESS, ARP was successfully enabled for this IP
    instance.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_gratuitous_send

Send gratuitous ARP request

Prototype

```
UINT  nx_arp_gratuitous_send(NX_IP *ip_ptr,
                             VOID (*response_handler)
                             (NX_IP *ip_ptr,
                             NX_PACKET *packet_ptr));
```

Description

This service sends a gratuitous ARP request. If an ARP response is subsequently received, the supplied response handler is called to process the error.

Parameters

ip_ptr	Pointer to previously created IP instance.
response_handler	Pointer to response handling function. If NX_NULL is supplied, responses are ignored.

Return Values

NX_SUCCESS	(0x00)	Successful gratuitous ARP send.
NX_NO_PACKET	(0x01)	No packet available.
NX_NOT_ENABLED	(0x14)	ARP is not enabled.
NX_IP_ADDRESS_ERROR	(0x21)	Current IP address is invalid.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Caller is not a thread.

Allowed From

Threads

Example

```
/*  Send gratuitous ARP without any response handler. */
status =  nx_arp_gratuitous_send(&ip_0, NX_NULL);

/*  If status is NX_SUCCESS the gratuitous ARP was successfully
    sent.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_hardware_address_find

Locate physical hardware address given an IP address

Prototype

```
UINT nx_arp_hardware_address_find(NX_IP *ip_ptr,
                                  ULONG ip_address,
                                  ULONG *physical_msw,
                                  ULONG *physical_lsw);
```

Description

This service attempts to find a physical hardware address in the ARP cache that is associated with the supplied IP address.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	IP address to search for.
physical_msw	Pointer to the variable for returning the most significant word of the physical address.
physical_lsw	Pointer to the variable for returning the least significant word of the physical address.

Return Values

NX_SUCCESS	(0x00)	Successful ARP hardware address find.
NX_ENTRY_NOT_FOUND	(0x16)	Mapping was not found in the ARP cache.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_PTR_ERROR	(0x07)	Invalid IP or memory pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/*  Search for the hardware address associated with the IP address of
    1.2.3.4 in the ARP cache of the previously created IP
    Instance 0.   */
status = nx_arp_hardware_address_find(&ip_0, IP_ADDRESS(1,2,3,4),
                                        &physical_msw,
                                        &physical_lsw);

/*  If status is NX_SUCCESS, the variables physical_msw and
    physical_lsw contain the hardware address.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_info_get

Retrieve information about ARP activities

Prototype

```
UINT nx_arp_info_get(NX_IP *ip_ptr,
                     ULONG *arp_requests_sent,
                     ULONG *arp_requests_received,
                     ULONG *arp_responses_sent,
                     ULONG *arp_responses_received,
                     ULONG *arp_dynamic_entries,
                     ULONG *arp_static_entries,
                     ULONG *arp_aged_entries,
                     ULONG *arp_invalid_messages);
```

Description

This service retrieves information about ARP activities for the associated IP instance.

i *If a destination pointer is NX_NULL, that particular information is not returned to the caller.*

Parameters

ip_ptr	Pointer to previously created IP instance.
arp_requests_sent	Pointer to destination for the total ARP requests sent from this IP instance.
arp_requests_received	Pointer to destination for the total ARP requests received from the network.
arp_responses_sent	Pointer to destination for the total ARP responses sent from this IP instance.
arp_responses_received	Pointer to the destination for the total ARP responses received from the network.
arp_dynamic_entries	Pointer to the destination for the current number of dynamic ARP entries.
arp_static_entries	Pointer to the destination for the current number of static ARP entries.

arp_aged_entries	Pointer to the destination of the total number of ARP entries that have aged and became invalid.
arp_invalid_messages	Pointer to the destination of the total invalid ARP messages received.

Return Values

NX_SUCCESS	(0x00)	Successful ARP information retrieval.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Pickup ARP information for ip_0.  */
status =  nx_arp_info_get(&ip_0, &arp_requests_sent,
                          &arp_requests_received,
                          &arp_responses_sent,
                          &arp_responses_received,
                          &arp_dynamic_entries,
                          &arp_static_entries,
                          &arp_aged_entries,
                          &arp_invalid_messages);

/*  If status is NX_SUCCESS, the ARP information has been stored in
    the supplied variables.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_ip_address_find

Locate IP address given a physical address

Prototype

```
UINT nx_arp_ip_address_find(NX_IP *ip_ptr, ULONG *ip_address,
                            ULONG physical_msw, ULONG physical_lsw);
```

Description

This service attempts to find an IP address in the ARP cache that is associated with the supplied physical address.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	Pointer to return IP address, if one is found that has been mapped.
physical_msw	Most significant word of the physical address to search for.
physical_lsw	Least significant word of the physical address to search for.

Return Values

NX_SUCCESS	(0x00)	Successful ARP IP address find
NX_ENTRY_NOT_FOUND	(0x16)	Mapping was not found in the ARP cache.
NX_PTR_ERROR	(0x07)	Invalid IP or memory pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/*  Search for the IP address associated with the hardware address of
    0x0:0x01234 in the ARP cache of the previously created IP
    Instance 0.  */
status =  nx_arp_ip_address_find(&ip_0, &ip_address, 0x0, 0x1234);

/*  If status is NX_SUCCESS, the variables ip_address contains the
    associated IP address.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_static_entries_delete

Delete all static ARP entries

Prototype

```
UINT nx_arp_static_entries_delete(NX_IP *ip_ptr);
```

Description

This function deletes all static entries in the ARP cache.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Static entries are deleted.
NX_PTR_ERROR	(0x07)	Invalid *ip_ptr* pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/*  Delete all the static ARP entries for IP Instance 0, assuming
    "ip_0" is the NX_IP structure for IP Instance 0.  */
status =  nx_arp_static_entries_delete(&ip_0);

/*  If status is NX_SUCCESS all static ARP entries in the ARP cache
    have been deleted.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_static_entry_create

Create static IP to hardware mapping in ARP cache

Prototype

```
UINT nx_arp_static_entry_create(NX_IP *ip_ptr,
                                ULONG ip_address,
                                ULONG physical_msw,
                                ULONG physical_lsw);
```

Description

This service creates a static IP-to-physical address mapping in the ARP cache for the specified IP instance. Static ARP entries are not subject to ARP periodic updates.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	IP address to map.
physical_msw	Most significant word of the physical address to map.
physical_lsw	Least significant word of the physical address to map.

Return Values

NX_SUCCESS	(0x00)	Successful ARP static entry create.
NX_NO_MORE_ENTRIES	(0x17)	No more ARP entries are available in the ARP cache.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/*  Create a static ARP entry on the previously created IP
    Instance 0.  */
status =  nx_arp_static_entry_create(&ip_0, IP_ADDRESS(1,2,3,4),
                                          0x0, 0x1234);

/*  If status is NX_SUCCESS, there is now a static mapping between
    the IP address of 1.2.3.4 and the physical hardware address of
    0x0:0x1234.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_delete

nx_arp_static_entry_delete

Delete static IP to hardware mapping in ARP cache

Prototype

```
UINT nx_arp_static_entry_delete(NX_IP *ip_ptr,
                                ULONG ip_address,
                                ULONG physical_msw,
                                ULONG physical_lsw);
```

Description

This service finds and deletes a previously created static IP-to-physical address mapping in the ARP cache for the specified IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	IP address that was mapped statically.
physical_msw	Most significant word of the physical address that was mapped statically.
physical_lsw	Least significant word of the physical address that was mapped statically.

Return Values

NX_SUCCESS	(0x00)	Successful ARP static entry delete.
NX_ENTRY_NOT_FOUND	(0x16)	Static ARP entry was not found in the ARP cache.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.

Allowed From

Threads

Preemption Possible

No

Example

```
/*  Delete a static ARP entry on the previously created IP
    Instance 0.  */
status =  nx_arp_static_entry_delete(&ip_0, IP_ADDRESS(1,2,3,4),
                                     0x0, 0x1234);

/*  If status is NX_SUCCESS, the previously created static ARP entry
    was successfully deleted.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create

nx_icmp_enable

Enable Internet Control Message Protocol (ICMP)

Prototype

```
UINT nx_icmp_enable(NX_IP *ip_ptr);
```

Description

This service enables the ICMP component for the specified IP instance. The ICMP component is responsible for handling Internet error messages and ping requests and replies.

*This service only enables ICMP for IPv4 service. To enable both ICMPv4 and ICMPv6, applications shall use the **nxd_icmp_enable** service.*

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful ICMP enable.
NX_ALREADY_ENABLED	(0x15)	ICMP is already enabled.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Enable ICMP on the previously created IP Instance 0.  */
status =  nx_icmp_enable(&ip_0);

/* If status is NX_SUCCESS, ICMP is enabled.  */
```

See Also

nx_icmp_info_get, nxd_icmp_enable

nx_icmp_info_get

Retrieve information about ICMP activities

Prototype

```
UINT nx_icmp_info_get(NX_IP *ip_ptr,
                      ULONG *pings_sent,
                      ULONG *ping_timeouts,
                      ULONG *ping_threads_suspended,
                      ULONG *ping_responses_received,
                      ULONG *icmp_checksum_errors,
                      ULONG *icmp_unhandled_messages);
```

Description

This service retrieves information about ICMP activities for the specified IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
pings_sent	Pointer to destination for the total number of pings sent.
ping_timeouts	Pointer to destination for the total number of ping timeouts.
ping_threads_suspended	Pointer to destination of the total number of threads suspended on ping requests.
ping_responses_received	Pointer to destination of the total number of ping responses received.
icmp_checksum_errors	Pointer to destination of the total number of ICMP checksum errors.
icmp_unhandled_messages	Pointer to destination of the total number of un-handled ICMP messages.

Return Values

NX_SUCCESS	(0x00)	Successful ICMP information retrieval.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/*  Retrieve ICMP information from previously created IP
    Instance 0. */
status =  nx_icmp_info_get(&ip_0, &pings_sent, &ping_timeouts,
                            &ping_threads_suspended,
                            &ping_responses_received,
                            &icmp_checksum_errors,
                            &icmp_unhandled_messages);

/*  If status is NX_SUCCESS, ICMP information was retrieved.  */
```

See Also

nx_icmp_enable, nx_igmp_loopback_disable,
nx_igmp_loopback_enable, nx_icmp_ping

nx_icmp_ping

Send ping request to specified IP address

Prototype

```
UINT nx_icmp_ping(NX_IP *ip_ptr,
                  ULONG ip_address,
                  CHAR *data, ULONG data_size,
                  NX_PACKET **response_ptr,
                  ULONG wait_option);
```

Description

This service sends a ping request to the specified IP address and waits for the specified amount of time for a ping response message. If no response is received, an error is returned. Otherwise, the entire response message, including the ICMP header, is returned in the variable pointed to by response_ptr.

To send a ping request to an IPv6 destination, applications shall use the ***nxd_icmp_ping*** or ***nxd_icmp_interface_ping*** service.

If NX_SUCCESS is returned, the application is responsible for releasing the received packet after it is no longer needed.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	IP address to ping.
data	Pointer to data area for ping message.
data_size	Number of bytes in the ping data
response_ptr	Pointer to packet pointer to return the ping response message in.
wait_option	Defines how long to wait for a ping response. Legal values are: 1 through 0xFFFFFFFE.

Return Values

NX_SUCCESS	(0x00)	Successful ping. Response message pointer was placed in the variable pointed to by response_ptr.

NX_NO_PACKET	(0x01)	Unable to allocate a ping request packet.
NX_OVERFLOW	(0x03)	Specified data area exceeds the default packet size for this IP instance.
NX_NO_RESPONSE	(0x29)	Requested IP did not respond.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_PTR_ERROR	(0x07)	Invalid IP or response pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/*  Issue a ping to IP address 1.2.3.5 from the previously created IP
    Instance 0.  */
status =  nx_icmp_ping(&ip_0, IP_ADDRESS(1,2,3,5), "abcd", 4,
                       &response_ptr, 10);

/*  If status is NX_SUCCESS, a ping response was received from IP
    address 1.2.3.5 and the response packet is contained in the
    packet pointed to by response_ptr. It should have the same "abcd"
    four bytes of data.  */
```

See Also

nx_icmp_enable, nx_icmp_info_get, nxd_icmp_interface_ping, nxd_icmp_ping

nx_igmp_enable

Enable Internet Group Management Protocol (IGMP)

Prototype

```
UINT nx_igmp_enable(NX_IP *ip_ptr);
```

Description

This service enables the IGMP component on the specified IP instance. The IGMP component is responsible for providing support for IP multicast group management operations.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IGMP enable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_ALREADY_ENABLED	(0x15)	This component has already been enabled.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Enable IGMP on the previously created IP Instance 0.  */
status =  nx_igmp_enable(&ip_0);

/* If status is NX_SUCCESS, IGMP is enabled.  */
```

See Also

nx_igmp_info_get,nx_igmp_loopback_disable, nx_igmp_loopback_enable, nx_igmp_multicast_interface_join, nx_igmp_multicast_join, nx_igmp_multicast_leave

nx_igmp_info_get

Retrieve information about IGMP activities

Prototype

```
UINT nx_igmp_info_get(NX_IP *ip_ptr,
                      ULONG *igmp_reports_sent,
                      ULONG *igmp_queries_received,
                      ULONG *igmp_checksum_errors,
                      ULONG *current_groups_joined);
```

Description

This service retrieves information about IGMP activities for the specified IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
igmp_reports_sent	Pointer to destination for the total number of ICMP reports sent.
igmp_queries_received	Pointer to destination for the total number of queries received by multicast router.
igmp_checksum_errors	Pointer to destination of the total number of IGMP checksum errors on receive packets.
current_groups_joined	Pointer to destination of the current number of groups joined through this IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IGMP information retrieval.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve IGMP information from previously created IP Instance 0.  */
status =  nx_igmp_info_get(&ip_0, &igmp_reports_sent,
                           &igmp_queries_received,
                           &igmp_checksum_errors,
                           &current_groups_joined);

/* If status is NX_SUCCESS, IGMP information was retrieved.  */
```

See Also

nx_igmp_enable, nx_igmp_loopback_disable, nx_igmp_loopback_enable, nx_igmp_multicast_interface_join, nx_igmp_multicast_join, nx_igmp_multicast_leave

nx_igmp_loopback_disable

Disable IGMP loopback

Prototype

```
UINT  nx_igmp_loopback_disable(NX_IP *ip_ptr);
```

Description

This service disables IGMP loopback for all subsequent multicast groups joined.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IGMP loopback disable.
NX_NOT_ENABLED	(0x14)	IGMP is not enabled.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Caller is not a thread or initialization.

Allowed From

Initialization, threads

Example

```
/*  Disable IGMP loopback for all subsequent multicast groups
    joined. */
status =  nx_igmp_loopback_disable(&ip_0);

/*  If status is NX_SUCCESS IGMP loopback is disabled.  */
```

See Also

nx_igmp_enable, nx_igmp_info_get, nx_igmp_loopback_enable, nx_igmp_multicast_interface_join, nx_igmp_multicast_join, nx_igmp_multicast_leave

nx_igmp_loopback_enable

Enable IGMP loopback

Prototype

```
UINT  nx_igmp_loopback_enable(NX_IP *ip_ptr);
```

Description

This service enables IGMP loopback for all subsequent multicast groups joined.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IGMP loopback disable.
NX_NOT_ENABLED	(0x14)	IGMP is not enabled.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Caller is not a thread or initialization.

Allowed From

Initialization, threads

Example

```
/* Enable IGMP loopback for all subsequent multicast
   groups joined. */
status =  nx_igmp_loopback_enable(&ip_0);

/* If status is NX_SUCCESS IGMP loopback is enabled.  */
```

See Also

nx_igmp_enable, nx_igmp_info_get, nx_igmp_loopback_disable, nx_igmp_multicast_interface_join, nx_igmp_multicast_join, nx_igmp_multicast_leave

nx_igmp_multicast_interface_join

Join IP interface to specified multicast group

Prototype

```
UINT nx_igmp_multicast_interface_join(NX_IP *ip_ptr,
                                      ULONG group_address,
                                      UINT interface_index)
```

Description

This service joins an IP instance to the specified multicast group via a specified network interface. An internal counter is maintained to keep track of the number of times the same group has been joined. After joined, the IGMP component will allow reception of IP packets with this group address via the specified network interface and also report to routers that this IP is a member of this multicast group. The IGMP membership join, report, and leave messages are also sent via the specified network interface.

Parameters

ip_ptr	Pointer to previously created IP instance.
group_address	Class D IP multicast group address to join in host byte order.
interface_index	Interface index attached to NetX Duo instance.

Return Values

NX_SUCCESS	(0x00)	Successful multicast group join.
NX_NO_MORE_ENTRIES	(0x17)	No more multicast groups can be joined, maximum exceeded.
NX_INVALID_INTERFACE	(0x4C	Interface index points to an invalid network interface.
NX_IP_ADDRESS_ERROR	(0x21)	Multicast group address provided is not a valid class D address.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	IP multicast support is not enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
    /*  Previously created IP Instance joins the multicast group
        244.0.0.200, via the interface at index 1 in the IP task
        interface list.   */
   status = nx_igmp_multicast_interface_join
                                    (&ip IP_ADDRESS(244,0,0,200), 1);

    /*  If status is NX_SUCCESS, the IP instance has successfully joined
        the multicast group. */
```

See Also

nx_igmp_enable, nx_igmp_info_get, nx_igmp_loopback_disable, nx_igmp_loopback_enable, nx_igmp_multicast_join, nx_igmp_multicast_leave

nx_igmp_multicast_join

Join IP instance to specified multicast group

Prototype

```
UINT nx_igmp_multicast_join(NX_IP *ip_ptr, ULONG group_address);
```

Description

This service joins an IP instance to the specified multicast group. An internal counter is maintained to keep track of the number of times the same group has been joined. The driver is commanded to send an IGMP report if this is the first join request out on the network indicating the host's intention to join the group. After joining, the IGMP component will allow reception of IP packets with this group address and report to routers that this IP is a member of this multicast group.

Parameters

ip_ptr	Pointer to previously created IP instance.
group_address	Class D IP multicast group address to join.

Return Values

NX_SUCCESS	(0x00)	Successful multicast group join.
NX_NO_MORE_ENTRIES	(0x17)	No more multicast groups can be joined, maximum exceeded.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP group address.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/*  Previously created IP Instance 0 joins the multicast group
    224.0.0.200. */
status =  nx_igmp_multicast_join(&ip_0, IP_ADDRESS(224,0,0,200);

/*  If status is NX_SUCCESS, this IP instance has successfully
    joined the multicast group 224.0.0.200.  */
```

See Also

nx_igmp_enable, nx_igmp_info_get, nx_igmp_loopback_disable, nx_igmp_loopback_enable, nx_igmp_multicast_interface_join, nx_igmp_multicast_leave

nx_igmp_multicast_leave

Cause IP instance to leave specified multicast group

Prototype

```
UINT nx_igmp_multicast_leave(NX_IP *ip_ptr, ULONG group_address);
```

Description

This service causes an IP instance to leave the specified multicast group, if the number of leave requests matches the number of join requests. Otherwise, the internal join count is simply decremented.

Parameters

ip_ptr	Pointer to previously created IP instance.
group_address	Multicast group to leave.

Return Values

NX_SUCCESS	(0x00)	Successful multicast group join.
NX_ENTRY_NOT_FOUND	(0x16)	Previous join request was not found.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP group address.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/*  Cause IP instance to leave the multicast group 224.0.0.200.  */
status =  nx_igmp_multicast_leave(&ip_0, IP_ADDRESS(224,0,0,200);

/*  If status is NX_SUCCESS, this IP instance has successfully left
    the multicast group 224.0.0.200.  */
```

See Also

nx_igmp_enable, nx_igmp_info_get, nx_igmp_loopback_disable, nx_igmp_loopback_enable, nx_igmp_multicast_interface_join,, nx_igmp_multicast_join

nx_ip_address_change_notifiy

Notify application if IP address changes

Prototype

```
UINT  nx_ip_address_change_notify(NX_IP *ip_ptr,
                                  VOID(*change_notify)(NX_IP *, VOID *),
                                  VOID *additional_info);
```

Description

This service registers an application notification function that is called whenever the IP address is changed.

Parameters

ip_ptr	Pointer to previously created IP instance.
change_notify	Pointer to IP change notification function. If this parameter is NX_NULL, IP address change notification is disabled.
additional_info	Pointer to optional additional information that is also supplied to the notification function when the IP address is changed.

Return Values

NX_SUCCESS	(0x00)	Successful IP address change notification.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.

Allowed From

Initialization, threads, timers

Example

```
/*  Register the function "my_ip_changed" to be called whenever the
    IP address is changed. */
status =  nx_ip_address_change_notify(&ip_0, my_ip_changed,
                                      NX_NULL);

/*  If status is NX_SUCCESS, the "my_ip_changed" function will be
    called whenever the IP address changes.  */
```

See Also

nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_address_get

Retrieve IP address and network mask

Prototype

```
UINT nx_ip_address_get(NX_IP *ip_ptr,
                       ULONG *ip_address,
                       ULONG *network_mask);
```

Description

This service retrieves information of the primary network interface.

*To obtain information of the secondary interface, use the service **nx_ip_interface_address_get**.*

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	Pointer to destination for IP address.
network_mask	Pointer to destination for network mask.

Return Values

NX_SUCCESS	(0x00)	Successful IP address get.
NX_PTR_ERROR	(0x07)	Invalid IP or return variable pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Get the IP address and network mask from the previously created
    IP instance 0.  */
status =  nx_ip_address_get(&ip_0, &ip_address, &network_mask);

/*  If status is NX_SUCCESS, the variables ip_address and
    network_mask contain the IP and network mask respectively.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_address_set

Set IP address and network mask

Prototype

```
UINT nx_ip_address_set(NX_IP *ip_ptr,
                       ULONG ip_address,
                       ULONG network_mask);
```

Description

This service sets IP address and network mask for the primary network interface.

*To set IP address and network mask for the secondary interface, use the service **nx_ip_interface_address_set**.*

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	New IP address.
network_mask	New network mask.

Return Values

NX_SUCCESS	(0x00)	Successful IP address set.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/*  Set the IP address and network mask to 1.2.3.4 and 0xFF for the
    previously created IP instance 0.  */
status =  nx_ip_address_set(&ip_0, IP_ADDRESS(1,2,3,4),
                            0xFFFFFF00UL);

/*  If status is NX_SUCCESS, the IP instance now has an IP address of
    1.2.3.4 and a network mask of 0xFF.   */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_info_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_receive, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_create

Create an IP instance

Prototype

```
UINT nx_ip_create(NX_IP *ip_ptr, CHAR *name, ULONG ip_address,
                  ULONG network_mask, NX_PACKET_POOL *default_pool,
                  VOID (*ip_network_driver)(NX_IP_DRIVER *),
                  VOID *memory_ptr, ULONG memory_size,
                  UINT priority);
```

Description

This service creates an IP instance with the user supplied IP address and network driver. In addition, the application must supply a previously created packet pool for the IP instance to use for internal packet allocation. Note that the supplied application network driver is not called until this IP's thread executes.

Parameters

ip_ptr	Pointer to control block to create a new IP instance.
name	Name of this new IP instance.
ip_address	IP address for this new IP instance.
network_mask	Mask to delineate the network portion of the IP address for sub-netting and super-netting uses.
default_pool	Pointer to control block of previously created NetX Duo packet pool.
ip_network_driver	User-supplied network driver used to send and receive IP packets.
memory_ptr	Pointer to memory area for the IP helper thread's stack area.
memory_size	Number of bytes in the memory area for the IP helper thread's stack.
priority	Priority of IP helper thread.

Return Values

NX_SUCCESS	(0x00)	Successful IP instance creation.
NX_IP_INTERNAL_ERROR	(0x20)	An internal IP system resource was not able to be created

		causing the IP create service to fail.
NX_PTR_ERROR	(0x07)	Invalid IP, network driver address, packet pool, or memory pointer.
NX_SIZE_ERROR	(0x09)	The supplied stack size is too small.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_IP_ADDRESS_ERROR	(0x21)	The supplied IP address is invalid.

Allowed From

Initialization, threads

Preemption Possible

Yes

Example

```
/*  Create an IP instance with an IP address of 1.2.3.4 and a network
    mask of 0xFFFFFF00UL. The "ethernet_driver" specifies the entry
    point of the application specific network driver and the
    "stack_memory_ptr" specifies the start of a 1024 byte memory
    area that is used for this IP instance's helper thread.  */
status = nx_ip_create(&ip_0, "NetX IP Instance 0",
                      IP_ADDRESS(1, 2, 3, 4),
                      0xFFFFFF00UL, &pool_0, ethernet_driver,
                      stack_memory_ptr, 1024, 1);

/*  If status is NX_SUCCESS, the IP instance has been created.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_raw_packet_interface_send, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_delete

Delete previously created IP instance

Prototype

```
UINT nx_ip_delete(NX_IP *ip_ptr);
```

Description

This service deletes a previously created IP instance and releases all of the system resources owned by the IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP deletion.
NX_SOCKETS_BOUND	(0x28)	This IP instance still has UDP or TCP sockets bound to it. All sockets must be unbound and deleted prior to deleting the IP instance.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/*  Delete a previously created IP instance. */
status = nx_ip_delete(&ip_0);

/*  If status is NX_SUCCESS, the IP instance has been deleted.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check, nx_ip_driver_direct_command

nx_ip_driver_interface_direct_command

Issue command to network driver

Prototype

```
UINT nx_ip_driver_interface_direct_command(NX_IP *ip_ptr,
                                           UINT command,
                                           UINT interface_index,
                                           ULONG *return_value_ptr);
```

Description

This service provides a direct interface to the application's primary network interface driver specified during the ***nx_ip_create*** call. Application-specific commands can be used providing their numeric value is greater than or equal to NX_LINK_USER_COMMAND.

*The **nx_ip_driver_direct_command** service sends all commands on the primary interface.*

Parameters

ip_ptr	Pointer to previously created IP instance.
command	Numeric command code. Standard commands are defined as follows: NX_LINK_GET_STATUS (10) NX_LINK_GET_SPEED (11) NX_LINK_GET_DUPLEX_TYPE (12) NX_LINK_GET_ERROR_COUNT (13) NX_LINK_GET_RX_COUNT (14) NX_LINK_GET_TX_COUNT (15) NX_LINK_GET_ALLOC_ERRORS (16) NX_LINK_USER_COMMAND (50)
interface_index	Network interface index to send command.
return_value_ptr	Pointer to return variable in the caller.

Return Values

NX_SUCCESS	(0x00)	Successful network driver direct command.
NX_UNHANDLED_COMMAND	(0x44)	Unhandled or unimplemented network driver command.

NX_INVALID_INTERFACE	(0x4C)	Invalid interface index
NX_PTR_ERROR	(0x07)	Invalid IP or return value pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Make a direct call to the application-specific network driver
    for the previously created IP instance. For this example, the
    network driver is interrogated for the link status.  */

/*  Set the interface index to the primary interface. */
UINT iface_index = 0;

status = nx_ip_driver_interface_direct_command(&ip_0,
                                               NX_LINK_GET_STATUS,
                                               iface_index,
                                               &link_status);

/*  If status is NX_SUCCESS, the link_status variable contains a
    NX_TRUE or NX_FALSE value representing the status of the
    physical link.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check, nx_ip_driver_direct_command

nx_ip_driver_direct_command

Issue command to network driver

Prototype

```
UINT nx_ip_driver_direct_command(NX_IP *ip_ptr,
                                 UINT command,
                                 ULONG *return_value_ptr);
```

Description

This service provides a direct interface to the application's primary network interface driver specified during the ***nx_ip_create*** call. Application-specific commands can be used providing their numeric value is greater than or equal to NX_LINK_USER_COMMAND.

*To issue commands for the secondary interface, use the **nx_ip_driver_interface_direct_command** service.*

Parameters

ip_ptr	Pointer to previously created IP instance.
command	Numeric command code. Standard commands are defined as follows: NX_LINK_GET_STATUS (10) NX_LINK_GET_SPEED (11) NX_LINK_GET_DUPLEX_TYPE (12) NX_LINK_GET_ERROR_COUNT (13) NX_LINK_GET_RX_COUNT (14) NX_LINK_GET_TX_COUNT (15) NX_LINK_GET_ALLOC_ERRORS (16) NX_LINK_USER_COMMAND (50)
return_value_ptr	Pointer to return variable in the caller.

Return Values

NX_SUCCESS	(0x00)	Successful network driver direct command.
NX_UNHANDLED_COMMAND	(0x44)	Unhandled or unimplemented network driver command.

NX_PTR_ERROR	(0x07)	Invalid IP or return value pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Make a direct call to the application-specific network driver
    for the previously created IP instance. For this example, the
    network driver is interrogated for the link status.  */
status = nx_ip_driver_direct_command(&ip_0, NX_LINK_GET_STATUS,
                                     &link_status);

/*  If status is NX_SUCCESS, the link_status variable contains a
    NX_TRUE or NX_FALSE value representing the status of the
    physical link.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_forwarding_disable

Disable IP packet forwarding

Prototype

```
UINT nx_ip_forwarding_disable(NX_IP *ip_ptr);
```

Description

This service disables forwarding IP packets inside the NetX Duo IP component. On creation of the IP task, this service is automatically disabled.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP forwarding disable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Disable IP forwarding on this IP instance.  */
status = nx_ip_forwarding_disable(&ip_0);

/*  If status is NX_SUCCESS, IP forwarding has been disabled on the
    previously created IP instance.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_forwarding_enable

Enable IP packet forwarding

Prototype

```
UINT nx_ip_forwarding_enable(NX_IP *ip_ptr);
```

Description

This service enables forwarding IP packets inside the NetX Duo IP component. On creation of the IP task, this service is automatically disabled.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP forwarding enable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Enable IP forwarding on this IP instance.  */
status = nx_ip_forwarding_enable(&ip_0);

/*  If status is NX_SUCCESS, IP forwarding has been enabled on the
    previously created IP instance.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_fragment_disable

Disable IP packet fragmenting

Prototype

```
UINT nx_ip_fragment_disable(NX_IP *ip_ptr);
```

Description

This service disables IP packet fragmenting and reassembling functionality. On creation of the IP task, this service is automatically disabled.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP fragment disable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Disable IP fragmenting on this IP instance.  */
status = nx_ip_fragment_disable(&ip_0);

/*  If status is NX_SUCCESS, disables IP fragmenting on the
    previously created IP instance.  */
```

See Also

nx_ip_create, nx_ip_delete, nx_ip_fragment_enable,

nx_ip_fragment_enable

Enable IP packet fragmenting

Prototype

```
UINT nx_ip_fragment_enable(NX_IP *ip_ptr);
```

Description

This service enables IP packet fragmenting and reassembling functionality. On creation of the IP task, this service is automatically disabled.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP fragment enable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Enable IP fragmenting on this IP instance.  */
status = nx_ip_fragment_enable(&ip_0);

/*  If status is NX_SUCCESS, IP fragmenting has been enabled on the
    previously created IP instance.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_gateway_address_set

Set Gateway IP address

Prototype

```
UINT  nx_ip_gateway_address_set(NX_IP *ip_ptr, ULONG ip_address);
```

Description

This service sets the Gateway IP address to the specified IP address. All out-of-network IP addresses are routed to this IP address for transmission. The gateway must be directly accessible through one of the network interfaces.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	IP address of the Gateway.

Return Values

NX_SUCCESS	(0x00)	Successful Gateway IP address set.
NX_PTR_ERROR	(0x07)	Invalid IP instance pointer.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

No

Example

```
/*  Setup the Gateway address for previously created IP
    Instance 0. */
status = nx_ip_gateway_address_set(&ip_0, IP_ADDRESS(1,2,3,99);

/*  If status is NX_SUCCESS, all out-of-network send requests are
    routed to 1.2.3.99. */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_static_route_add, nx_ip_static_route_delete

nx_ip_info_get

Retrieve information about IP activities

Prototype

```
UINT nx_ip_info_get(NX_IP *ip_ptr,
                    ULONG *ip_total_packets_sent,
                    ULONG *ip_total_bytes_sent,
                    ULONG *ip_total_packets_received,
                    ULONG *ip_total_bytes_received,
                    ULONG *ip_invalid_packets,
                    ULONG *ip_receive_packets_dropped,
                    ULONG *ip_receive_checksum_errors,
                    ULONG *ip_send_packets_dropped,
                    ULONG *ip_total_fragments_sent,
                    ULONG *ip_total_fragments_received);
```

Description

This service retrieves information about IP activities for the specified IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_total_packets_sent	Pointer to destination for the total number of IP packets sent.
ip_total_bytes_sent	Pointer to destination for the total number of bytes sent.
ip_total_packets_received	Pointer to destination of the total number of IP receive packets.
ip_total_bytes_received	Pointer to destination of the total number of IP bytes received.
ip_invalid_packets	Pointer to destination of the total number of invalid IP packets.
ip_receive_packets_dropped	Pointer to destination of the total number of receive packets dropped.
ip_receive_checksum_errors	Pointer to destination of the total number of checksum errors in receive packets.
ip_send_packets_dropped	Pointer to destination of the total number of send packets dropped.

ip_total_fragments_sent	Pointer to destination of the total number of fragments sent.
ip_total_fragments_received	Pointer to destination of the total number of fragments received.

Return Values

NX_SUCCESS	(0x00)	Successful IP information retrieval.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/*  Retrieve IP information from previously created IP
    Instance 0.  */
status =  nx_ip_info_get(&ip_0,
                         &ip_total_packets_sent,
                         &ip_total_bytes_sent,
                         &ip_total_packets_received,
                         &ip_total_bytes_received,
                         &ip_invalid_packets,
                         &ip_receive_packets_dropped,
                         &ip_receive_checksum_errors,
                         &ip_send_packets_dropped,
                         &ip_total_fragments_sent,
                         &ip_total_fragments_received);

/* If status is NX_SUCCESS, IP information was retrieved.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_interface_address_get

Retrieve interface IP address

Prototype

```
UINT nx_ip_interface_address_get (NX_IP *ip_ptr,
                                  ULONG interface_id,
                                  ULONG *ip_address,
                                  ULONG *network_mask)
```

Description

This service retrieves the IP address of a specified network interface.

The specified interface, if not the primary interface, must be previously attached to the IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
interface_id	Interface index attached to NetX Duo instance.
ip_address	Pointer to destination for the device interface IP address.
network_mask	Pointer to destination for the device interface network mask.

Return Values

NX_SUCCESS	(0x00)	Successful IP address get.
NX_INVALID_INTERFACE	(0x4C)	Specified network interface is invalid.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Get device IP address and network mask for the specified
    interface index 1 in IP instance list of interfaces).  */
status =  nx_ip_interface_address_get(ip_ptr,1, &ip_address,
                                               &network_mask);

/*  If status is NX_SUCCESS the interface address was successfully
    retrieved.  */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check

nx_ip_interface_address_set

Set interface IP address and network mask

Prototype

```
UINT nx_ip_interface_address_set (NX_IP *ip_ptr,
                                   ULONG interface_id,
                                   ULONG ip_address,
                                   ULONG network_mask)
```

Description

This service sets the IP address and network mask for the specified IP interface.

The specified interface must be previously attached to the IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
interface_id	Interface index attached to NetX Duo instance.
ip_address	New network interface IP address.
network_mask	New interface network mask.

Return Values

NX_SUCCESS	(0x00)	Successful IP address set.
NX_INVALID_INTERFACE	(0x4C)	Specified network interface is invalid.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid pointers.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/*  Set device IP address and network mask for the specified
    interface index 1 in IP instance list of interfaces).  */
status =  nx_ip_interface_address_set(ip_ptr,1, ip_address,
                                      network_mask);

/*  If status is NX_SUCCESS the interface IP address and mask was
    successfully set.  */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check

nx_ip_interface_attach

Attach network interface to IP instance

Prototype

```
UINT nx_ip_interface_attach(NX_IP *ip_ptr, CHAR *interface_name,
                            ULONG ip_address,
                            ULONG network_mask,
                            VOID(*ip_link_driver)
                                 (struct NX_IP_DRIVER_STRUCT *));
```

Description

This function adds a physical network interface to the IP interface table. Note the IP task is created with the primary interface so each additional interface is secondary to the primary interface. The total number of network interfaces attached to the IP instance (including the primary interface) cannot exceed NX_MAX_PHYSICAL_INTERFACES.

ip_ptr must point to a valid NetX Duo IP structure. NX_MAX_PHYSICAL_INTERFACES must be configured for the number of network interfaces for the IP instance. The default value is one.

Parameters

ip_ptr	Pointer to previously created IP instance.
interface_name	Pointer to device name buffer.
ip_address	Device IP address in host byte order.
network_mask	Device network mask in host byte order.
ip_link_driver	Ethernet driver for the interface.

Return Values

NX_SUCCESS	**(0x00)**	Entry is added to static routing table.
NX_NO_MORE_ENTRIES	**(0x17)**	Max number of interfaces. NX_MAX_PHYSICAL_INTERFACES is exceeded.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid pointer input.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address input.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/*  Attach secondary interface for device IP address 192.168.1.68
    with the specified ethernet driver. */
status = nx_ip_interface_attach(ip_ptr, "secondary_port",
                                 IP_ADDRESS(192,168,1,68),
                                 0xFFFFFF00UL,
                                 nx_etherDriver_mcf5485);

/*  If status is NX_SUCCESS the interface was successfully added to
    the IP task interface table.  */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command

nx_ip_interface_info_get

Retrieve network interface parameters

Prototype

```
UINT nx_ip_interface_info_get(NX_IP *ip_ptr,
                              UINT interface_index,
                              CHAR **interface_name,
                              ULONG *ip_address,
                              ULONG *network_mask,
                              ULONG *mtu_size,
                              ULONG *physical_address_msw,
                              ULONG *physical_address_lsw);
```

Description

This function retrieves information on network parameters for the specified interface. All network data is retrieved in host byte order.

ip_ptr must point to a valid NetX Duo IP structure. The specified interface, if not the primary interface, must be previously attached to the IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
interface_index	Index specifying network interface.
interface_name	Pointer to destination for interface name.
ip_address	Pointer to destination for network interface IP address.
network_mask	Pointer to destination for network interface mask.
mtu_size	Pointer to destination for maximum transfer unit for the IP task. Differs from the driver MTU by the additional size for the Ethernet header.
physical_address_msw	Pointer to destination for MSB of interface MAC address.
physical_address_lsw	Pointer to destination for LSB of interface MAC address.

Return Values

NX_SUCCESS	(0x00)	Entry is added to static routing table.
NX_PTR_ERROR	(0x07)	Invalid pointer input.
NX_INVALID_INTERFACE	(0x4C)	Invalid IP pointer.

Allowed From

Initialization, threads, timers, ISRs

Preemption Possible

No

Example

```
/*  Retrieve interface parameters for the specified interface (index
    1 in IP instance list of interfaces).  */
status =  nx_ip_interface_info_get(ip_ptr, 1, &name_ptr,
                                   &ip_address,
                                   &network_mask,
                                   &mtu_size,
                                   &physical_address_msw,
                                   &physical_address_lsw);

/*  If status is NX_SUCCESS the interface was successfully added to
    the IP task interface table.  */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_status_check

nx_ip_interface_status_check

Check status of attached IP interface

Prototype

```
UINT nx_ip_interface_status_check (NX_IP *ip_ptr,
                                   UINT interface_index,
                                   ULONG needed_status,
                                   ULONG *actual_status,
                                   ULONG wait_option);
```

Description

This service checks and optionally waits for the specified status of the interface corresponding to the interface index attached to the IP instance. Note: the *nx_ip_status_check* service can provide the same information but defaults to the primary interface on the IP instance.

ip_ptr must point to a valid NetX Duo IP structure. The specified interface, if not the primary interface, must be previously attached to the IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
interface_index	Index specifying network interface.
needed_status	IP status requested, defined in bit-map form as follows: NX_IP_INITIALIZE_DONE (0x0001) NX_IP_ADDRESS_RESOLVED (0x0002) NX_IP_LINK_ENABLED (0x0004) NX_IP_ARP_ENABLED (0x0008) NX_IP_UDP_ENABLED (0x0010) NX_IP_TCP_ENABLED (0x0020) NX_IP_IGMP_ENABLED (0x0040) NX_IP_RARP_COMPLETE (0x0080) NX_IP_INTERFACE_LINK_ENABLED (0x0100)
actual_status	Pointer to the actual bits set.
wait_option	Defines how the service behaves if the requested status bits are not available. The wait options are defined as follows: NX_NO_WAIT 0x00000000) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful IP status check.
NX_PTR_ERROR	(0x07)	IP pointer is or has become invalid or actual status pointer is invalid.
NX_NOT_SUCCESSFUL	(0x43)	Status request was not satisfied within the timeout specified.
NX_INVALID_INTERFACE	(0x4C)	Invalid interface.
NX_OPTION_ERROR	(0x0A)	Invalid needed status option.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/*  Wait 10 ticks for the link up status on the specified interface
    index 1 in IP instance list of interfaces). */
status = nx_ip_interface_status_check(&ip_0, 1, NX_IP_LINK_ENABLED,
                                      &actual_status, 10);

/*  If status is NX_SUCCESS, the link for the specified interface is
    up. */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get,

nx_ip_max_payload_size_find

Compute maximum packet data payload

Prototype

```
UINT nx_ip_max_payload_size_find(NX_IP *ip_ptr,
                                 NXD_ADDRESS *dest_address,
                                 UINT if_index,
                                 UINT src_port,
                                 UINT dest_port,
                                 ULONG protocol,
                                 ULONG *start_offset_ptr,
                                 ULONG *payload_length_ptr)
```

Description

This function finds the maximum payload size that will not require IP fragmentation to reach the destination; e.g., payload is at or below the local interface MTU size. (or the Path MTU value obtained via IPv6 Path MTU discovery). IP header and upper application header size (TCP or UDP) are subtracted from the total payload. If NetX Duo IPsec Security Policy applies to this end-points, the IPsec headers (ESP/AH) and associated overhead, such as Initial Vector, is also subtracted from the MTU. This service is applicable for both IPv4 and IPv6 packets.

The parameter *if_index* specifies the interface to use for sending out the packet. For a multihome system, the caller needs to specify the *if_index* parameter if the destination is a broadcast (IPv4 only), multicast, or IPv6 link-local address.

There is no equivalent NetX service.

Restrictions

The IP instance must be previously created.

Parameters

ip_ptr	Pointer to IP instance
dest_address	Pointer to packet destination address
if_index	Indicates the index of the interface to use
src_port	Source port number
dest_port	Destination port number
start_offset_ptr	Pointer to the start of data for maximum packet payload
payload_length_ptr	Pointer to payload size excluding headers

Return Values

NX_SUCCESS	(0x00)	Payload successfully computed
NX_PTR_ERROR	(0x07)	Invalid IP pointer
NX_IP_ADDRESS_ERROR	(0x21)	Invalid address supplied
NX_NOT_SUPPORTED	(0x4B)	Invalid protocol (not UDP or TCP)

Example

```
/*  The following example determines the maximum payload for UDP
    packet to remote host. */

status = nx_ip_max_payload_size_find(&ip_0,  0,
                                     &dest_ipv6_address,
                                     source_port,
                                     dest_port, NX_PROTOCOL_UDP,
                                     &start_offset,
                                     &payload_length);

/*  A return value of NX_SUCCESS indicates the packet payload
    payload_length starting at the offset start_offset is
    successfully computed. */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release

nx_ip_raw_packet_disable

Disable raw packet sending/receiving

Prototype

```
UINT nx_ip_raw_packet_disable(NX_IP *ip_ptr);
```

Description

This service disables transmission and reception of raw IP packets for this IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP raw packet disable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Disable raw packet sending/receiving for this IP instance.  */
status = nx_ip_raw_packet_disable(&ip_0);

/*  If status is NX_SUCCESS, raw IP packet sending/receiving has
    been disabled for the previously created IP instance.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_raw_packet_enable

Enable raw packet sending/receiving

Prototype

```
UINT nx_ip_raw_packet_enable(NX_IP *ip_ptr);
```

Description

This service enables transmission and reception of raw IP packets for this IP instance. By enabling raw IP facilities, the protocol field in the IP header is not examined on reception. Instead, all incoming IP packets are placed on the raw IP receive queue.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP raw packet enable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Enable raw packet sending/receiving for this IP instance.  */
status = nx_ip_raw_packet_enable(&ip_0);

/*  If status is NX_SUCCESS, raw IP packet sending/receiving has
    been enabled for the previously created IP instance.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_raw_packet_interface_send

Send raw IP packet out specified network interface

Prototype

```
UINT  nx_ip_raw_packet_interface_send(NX_IP *ip_ptr,
                                      NX_PACKET *packet_ptr,
                                      ULONG destination_ip,
                                      UINT interface_index,
                                      ULONG type_of_service);
```

Description

This service sends a raw IP packet to the specified destination IP address from the specified network interface. Note that this routine returns immediately, and it is, therefore, not known if the IP packet has actually been sent. The network driver will be responsible for releasing the packet when the transmission is complete. This service differs from other services in that there is no way of knowing if the packet was actually sent. It could get lost on the Internet.

Note that raw IP processing must be enabled.

Parameters

ip_ptr	Pointer to previously created IP task.
packet_ptr	Pointer to packet to transmit.
destination_ip	IP address to send packet.
interface_index	Index of interface to send packet out on.
type_of_service	Type of service for packet.

Return Values

NX_SUCCESS	(0x00)	Packet successfully transmitted.
NX_IP_ADDRESS_ERROR	(0x21)	No suitable outgoing interface available.
NX_NOT_ENABLED	(0x14)	Raw IP packet processing not enabled.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid pointer input.
NX_OPTION_ERROR	(0x0A)	Invalid type of service specified.

NX_OVERFLOW	(0x03)	Invalid packet prepend pointer.
NX_UNDERFLOW	(0x02)	Invalid packet prepend pointer.
NX_INVALID_INTERFACE	(0x4C)	Invalid interface index specified.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/*  Send packet out on interface 1 with normal type of service.  */
status = nx_ip_raw_packet_interface_send (ip_ptr, packet_ptr,
                                          destination_ip, 1,
                                          NX_IP_NORMAL);

/*  If status is NX_SUCCESS the packet was successfully
    transmitted. */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_raw_packet_receive

Receive raw IP packet

Prototype

```
UINT nx_ip_raw_packet_receive(NX_IP *ip_ptr,
                              NX_PACKET **packet_ptr,
                              ULONG wait_option);
```

Description

This service receives a raw IP packet from the specified IP instance. If there are IP packets on the raw packet receive queue, the first (oldest) packet is returned to the caller. Otherwise, if no packets are available, the caller may suspend as specified by the wait option.

If NX_SUCCESS, is returned, the application is responsible for releasing the received packet when it is no longer needed.

Parameters

ip_ptr	Pointer to previously created IP instance.
packet_ptr	Pointer to pointer to place the received raw IP packet in.
wait_option	Defines how the service behaves if there are no raw IP packets available. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 – 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful IP raw packet receive.
NX_NO_PACKET	(0x01)	No packet was available.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

NX_PTR_ERROR	(0x07)	Invalid IP or return packet pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Preemption Possible

Yes

Example

```
/*  Receive a raw IP packet for this IP instance, wait for a maximum
    of 4 timer ticks.  */
status = nx_ip_raw_packet_receive(&ip_0, &packet_ptr, 4);

/*  If status is NX_SUCCESS, the raw IP packet pointer is in the
    variable packet_ptr.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_enable, nx_ip_fragment_disable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_raw_packet_send

Send raw IP packet

Prototype

```
UINT nx_ip_raw_packet_send(NX_IP *ip_ptr,
                           NX_PACKET *packet_ptr,
                           ULONG destination_ip,
                           ULONG type_of_service);
```

Description

This service sends a raw IP packet to the specified destination IP address. Note that this routine returns immediately, and it is therefore not known whether the IP packet has actually been sent. The network driver will be responsible for releasing the packet when the transmission is complete. This service differs from other services in that there is no way of knowing if the packet was actually sent. It could get lost on the Internet.

For a multihome system, NetX Duo uses the destination IP address to find an appropriate network interface. If the destination IP address is broadcast or multicast, this service would return failure. Applications use the *nx_ip_raw_packet_interface_send* in this case.

! *Unless an error is returned, the application should not release the packet after this call. Doing so will cause unpredictable results because the network driver will release the packet after transmission.*

Parameters

ip_ptr	Pointer to previously created IP instance.
packet_ptr	Pointer to the raw IP packet to send.
destination_ip	Destination IP address, which can be a specific host IP address, a network broadcast, an internal loop-back, or a multicast address.
type_of_service	Defines the type of service for the transmission, legal values are as follows:

NX_IP_NORMAL	(0x00000000)
NX_IP_MIN_DELAY	(0x00100000)
NX_IP_MAX_DATA	(0x00080000)
NX_IP_MAX_RELIABLE	(0x00040000)
NX_IP_MIN_COST	(0x00020000)

Return Values

NX_SUCCESS	(0x00)	Successful IP raw packet send initiated.
NX_NOT_ENABLED	(0x14)	Raw IP feature is not enabled.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_OPTION_ERROR	(0x0A)	Invalid type of service.
NX_UNDERFLOW	(0x02)	Not enough room to prepend an IP header on the packet.
NX_OVERFLOW	(0x03)	Packet append pointer is invalid.
NX_PTR_ERROR	(0x07)	Invalid IP or packet pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/*  Send a raw IP packet to IP address 1.2.3.5.  */
status = nx_ip_raw_packet_send(&ip_0, packet_ptr,
                               IP_ADDRESS(1,2,3,5),
                               NX_IP_NORMAL);

/*  If status is NX_SUCCESS, the raw IP packet pointed to by
    packet_ptr has been sent.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_static_route_add

Add static route

Prototype

```
UINT nx_ip_static_route_add(NX_IP *ip_ptr,
                            ULONG network_address,
                            ULONG net_mask,
                            ULONG next_hop);
```

Description

This function adds an entry to the static routing table. Note that the *next_hop* address must be directly accessible from the local interface.

Note that ip_ptr must point to a valid NetX Duo IP structure and static routing must be enabled via NX_ENABLE_IP_STATIC_ROUTING to use this service.

Parameters

ip_ptr	Pointer to previously created IP instance.
network_address	Target network address, in host byte order
net_mask	Target network mask, in host byte order
next_hop	Next hop address for the target network, in host byte order

Return Values

NX_SUCCESS	(0x00)	Entry is added to the static routing table.
NX_OVERFLOW	(0x03)	Static routing table is full.
NX_NOT_SUPPORTED	(0x4B)	This feature is not compiled in.
NX_IP_ADDRESS_ERROR	(0x21)	Next hop is not directly accessible via local interfaces.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid *ip_ptr* pointer.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/*  Specify the next hop for 192.168.1.68 through the gateway
    192.168.1.1.  */
status =  nx_ip_static_route_add(ip_ptr, IP_ADDRESS(192,168,1,68),
                                 0xFFFFFF00UL,
                                 IP_ADDRESS(192,168,1,1));

/* If status is NX_SUCCESS the route was successfully added to the
    static routing table.  */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable

nx_ip_static_route_delete

Delete static route

Prototype

```
UINT nx_ip_static_route_delete(NX_IP *ip_ptr,
                               ULONG network_address,
                               ULONG net_mask);
```

Description

This function deletes an entry from the static routing table.

Note that ip_ptr must point to a valid NetX Duo IP structure and static routing must be enabled via NX_ENABLE_IP_STATIC_ROUTING to use this service.

Parameters

ip_ptr	Pointer to previously created IP instance.
network_address	Target network address, in host byte order.
net_mask	Target network mask, in host byte order.

Return Values

NX_SUCCESS	(0x00)	Successful deletion from the static routing table.
NX_NOT_SUCCESSFUL	(0x43)	Entry cannot be found in the routing table.
NX_NOT_SUPPORTED	(0x4B)	This feature is not compiled in.
NX_PTR_ERROR	(0x07)	Invalid *ip_ptr* pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization,threads

Preemption Possible

No

Example

```
/*  Remove the static route for 192.168.1.68 from the routing
    table.*/
status =  nx_ip_static_route_delete(ip_ptr,
                                    IP_ADDRESS(192,168,1,68),
                                    0xFFFFFF00UL,);

/*  If status is NX_SUCCESS the route was successfully removed from
    the static routing table.  */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_static_route_add, nx_ip_status_check

nx_ip_status_check

Check status of an IP instance

Prototype

```
UINT nx_ip_status_check(NX_IP *ip_ptr,
                        ULONG needed_status,
                        ULONG *actual_status,
                        ULONG wait_option);
```

Description

This service checks and optionally waits for the specified status of the primary network interface of a previously created IP instance.

Parameters

ip_ptr — Pointer to previously created IP instance.

needed_status — IP status requested, defined in bit-map form as follows:

NX_IP_INITIALIZE_DONE	(0x0001)
NX_IP_ADDRESS_RESOLVED	(0x0002)
NX_IP_LINK_ENABLED	(0x0004)
NX_IP_ARP_ENABLED	(0x0008)
NX_IP_UDP_ENABLED	(0x0010)
NX_IP_TCP_ENABLED	(0x0020)
NX_IP_IGMP_ENABLED	(0x0040)
NX_IP_RARP_COMPLETE	(0x0080)
NX_IP_INTERFACE_LINK_ENABLED	(0x0100)

actual_status — Pointer to destination of actual bits set.

wait_option — Defines how the service behaves if the requested status bits are not available. The wait options are defined as follows:

NX_NO_WAIT	0x00000000)
timeout value	(0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful IP status check.
NX_PTR_ERROR	(0x07)	IP pointer is or has become invalid, or actual status pointer is invalid.
NX_NOT_SUCCESSFUL	(0x43)	Status request was not satisfied within the timeout specified.
NX_OPTION_ERROR	(0x0a)	Invalid needed status option.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/*  Wait 10 ticks for the link up status on the previously created IP
    instance.   */
status = nx_ip_status_check(&ip_0, NX_IP_LINK_ENABLED,
                              &actual_status, 10);

/*  If status is NX_SUCCESS, the link for the specified IP instance
    is up.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete

nx_packet_allocate

Allocate packet from specified pool

Prototype

```
UINT nx_packet_allocate(NX_PACKET_POOL *pool_ptr,
                        NX_PACKET **packet_ptr,
                        ULONG packet_type,
                        ULONG wait_option);
```

Description

This service allocates a packet from the specified pool and adjusts the prepend pointer in the packet according to the type of packet specified. If no packet is available, the service suspends according to the supplied wait option.

Parameters

pool_ptr	Pointer to previously created packet pool.
packet_ptr	Pointer to the pointer of the allocated packet pointer.
packet_type	Defines the type of packet requested. See "Packet Memory Pools" on page 55 in Chapter 3 for a list of supported packet types.
wait_option	Defines the wait time in ticks if there are no packets available in the packet pool. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful packet allocate.
NX_NO_PACKET	(0x01)	No packet available.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_OPTION_ERROR	(0x0A)	Invalid packet type.
NX_PTR_ERROR	(0x07)	Invalid pool or packet return pointer.
NX_INVALID_PARAMETERS	(0x4D)	Packet size cannot support protocol.
NX_CALLER_ERROR	(0x11)	Invalid wait option from non-thread.

Allowed From

Initialization, threads, timers, and ISRs (application network drivers)

Preemption Possible

Yes

Example

```
/*  Allocate a new UDP packet from the previously created packet pool
    and suspend for a maximum of 5 timer ticks if the pool is
    empty. */
status = nx_packet_allocate(&pool_0, &packet_ptr,
                              NX_UDP_PACKET, 5);

/*  If status is NX_SUCCESS, the newly allocated packet pointer is
    found in the variable packet_ptr.  */
```

See Also

nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_copy

Copy packet

Prototype

```
UINT  nx_packet_copy(NX_PACKET *packet_ptr,
                     NX_PACKET **new_packet_ptr,
                     NX_PACKET_POOL *pool_ptr,
                     ULONG wait_option);
```

Description

This service copies the information in the supplied packet to one or more new packets that are allocated from the supplied packet pool. If successful, the pointer to the new packet is returned in destination pointed to by **new_packet_ptr**.

Parameters

packet_ptr	Pointer to the source packet.
new_packet_ptr	Pointer to the destination of where to return the pointer to the new copy of the packet.
pool_ptr	Pointer to the previously created packet pool that is used to allocate one or more packets for the copy.
wait_option	Defines how the service waits if there are no packets available. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful packet copy.
NX_NO_PACKET	(0x01)	Packet not available for copy.
NX_INVALID_PACKET	(0x12)	Empty source packet or copy failed.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to tx_thread_wait_abort.

NX_PTR_ERROR	(0x07)	Invalid pool, packet, or destination pointer.
NX_UNDERFLOW	(0x02)	Invalid packet prepend pointer.
NX_OVERFLOW	(0x03)	Invalid packet append pointer.
NX_CALLER_ERROR	(0x11)	A wait option was specified in initialization or in an ISR.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
NX_PACKET *new_copy_ptr;

/* Copy packet pointed to by "old_packet_ptr" using packets from
   previously created packet pool_0. */
status = nx_packet_copy(old_packet, &new_copy_ptr, &pool_0, 20);

/* If status is NX_SUCCESS, new_copy_ptr points to the packet copy. */
```

See Also

nx_packet_allocate, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_data_append

Append data to end of packet

Prototype

```
UINT nx_packet_data_append(NX_PACKET *packet_ptr,
                           VOID *data_start, ULONG data_size,
                           NX_PACKET_POOL *pool_ptr,
                           ULONG wait_option);
```

Description

This service appends data to the end of the specified packet. The supplied data area is copied into the packet. If there is not enough memory available, one or more packets will be allocated to satisfy the request.

Parameters

packet_ptr	Packet pointer.
data_start	Pointer to the start of the user's data area to append to the packet.
data_size	Size of user's data area.
pool_ptr	Pointer to packet pool from which to allocate another packet if there is not enough room in the current packet.
wait_option	Defines how the service behaves if there are no packets available. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful packet append.
NX_NO_PACKET	(0x01)	No packet available.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_UNDERFLOW	(0x02)	Prepend pointer is less than payload start.
NX_OVERFLOW	(0x03)	Append pointer is greater than payload end.
NX_PTR_ERROR	(0x07)	Invalid pool, packet, or data Pointer.
NX_SIZE_ERROR	(0x09)	Invalid data size.
NX_CALLER_ERROR	(0x11)	Invalid wait option from non-thread.

Allowed From

Initialization, threads, timers, and ISRs (application network drivers)

Preemption Possible

Yes

Example

```
/*  Append "abcd" to the specified packet.  */
status = nx_packet_data_append(packet_ptr, "abcd", 4, &pool_0, 5);

/*  If status is NX_SUCCESS, the additional four bytes "abcd" have
    been appended to the packet.  */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_data_extract_offset

Extract data from packet via an offset

Prototype

```
UINT nx_packet_data_extract_offset(NX_PACKET *packet_ptr,
                                   ULONG offset, VOID
                                   *buffer_start,
                                   ULONG buffer_length,
                                   ULONG *bytes_copied);
```

Description

This service copies data from a NetX Duo packet (or packet chain) starting at the specified offset from the packet prepend pointer of the specified size in bytes into the specified buffer. The number of bytes actually copied is returned in *bytes_copied.* This service does not remove data from the packet, nor does it adjust the prepend pointer.

Parameters

packet_ptr	Pointer to packet to extract
offset	Offset from the current prepend pointer.
buffer_start	Pointer to start of save buffer
buffer_length	Number of bytes to copy
bytes_copied	Number of bytes actually copied

Return Values

NX_SUCCESS	(0x00)	Successful packet copy
NX_PACKET_OFFSET_ERROR	(0x53)	Invalid offset value was supplied
NX_PTR_ERROR	(0x07)	Invalid packet pointer or buffer pointer

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
/*  Extract 10 bytes from the start of the received packet buffer
    into the specified memory area. */
status = nx_packet_data_extract_offset(my_packet, 0, &data[0], 10,
                                       &bytes_copied) ;

/*  If status is NX_SUCCESS, 10 bytes were successfully copied into
    the data buffer. */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_data_retrieve

Retrieve data from packet

Prototype

```
UINT  nx_packet_data_retrieve(NX_PACKET *packet_ptr,
                              VOID *buffer_start,
                              ULONG *bytes_copied);
```

Description

This service copies data from the supplied packet into the supplied buffer. The actual number of bytes copied is returned in the destination pointed to by **bytes_copied**.

The destination buffer must be large enough to hold the packet's contents. If not, memory will be corrupted causing unpredictable results.

Parameters

packet_ptr	Pointer to the source packet.
buffer_start	Pointer to the start of the buffer area.
bytes_copied	Pointer to the destination for the number of bytes copied.

Return Values

NX_SUCCESS	(0x00)	Successful packet data retrieve.
NX_INVALID_PACKET	(0x12)	Invalid packet.
NX_PTR_ERROR	(0x07)	Invalid packet, buffer start, or bytes copied pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
UCHAR                    buffer[512];
ULONG                    bytes_copied;

/*  Retrieve data from packet pointed to by "packet_ptr". */
status = nx_packet_data_retrieve(packet_ptr, buffer, &bytes_copied);

/*  If status is NX_SUCCESS, buffer contains the contents of the
    packet, the size of which is contained in "bytes_copied." */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_length_get

Get length of packet data

Prototype

```
UINT nx_packet_length_get (NX_PACKET *packet_ptr, ULONG *length);
```

Description

This service gets the length of the data in the specified packet.

Parameters

packet_ptr	Pointer to the packet.
length	Destination for the packet length.

Return Values

NX_SUCCESS	(0x00)	Successful packet length get.
NX_PTR_ERROR	(0x07)	Invalid packet pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
/*  Get the length of the data in "my_packet." */
status = nx_packet_length_get(my_packet, &my_length);

/*  If status is NX_SUCCESS, data length is in "my_length". */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_pool_create

Create packet pool in specified memory area

Prototype

```
UINT nx_packet_pool_create(NX_PACKET_POOL *pool_ptr,
                           CHAR *name,
                           ULONG payload_size,
                           VOID *memory_ptr,
                           ULONG memory_size);
```

Description

This service creates a packet pool of the specified packet size in the memory area supplied by the user.

Parameters

pool_ptr	Pointer to packet pool control block.
name	Pointer to application's name for the packet pool.
payload_size	Number of bytes in each packet in the pool. This value must be at least 40 bytes and must also be evenly divisible by 4.
memory_ptr	Pointer to the memory area to place the packet pool in. The pointer should be aligned on an ULONG boundary.
memory_size	Size of the pool memory area.

Return Values

NX_SUCCESS	(0x00)	Successful packet pool create.
NX_PTR_ERROR	(0x07)	Invalid pool or memory pointer.
NX_SIZE_ERROR	(0x09)	Invalid block or memory size.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/*  Create a packet pool of 32000 bytes starting at physical
    address 0x10000000.  */
status = nx_packet_pool_create(&pool_0, "Default Pool", 128,
                               (void *) 0x10000000, 32000);

/*  If status is NX_SUCCESS, the packet pool has been successfully
    created.  */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_pool_delete

Delete previously created packet pool

Prototype

```
UINT nx_packet_pool_delete(NX_PACKET_POOL *pool_ptr);
```

Description

This service deletes a previously create packet pool. NetX Duo checks for any threads currently suspended on packets in the packet pool and clears the suspension.

Parameters

pool_ptr	Packet pool control block pointer.

Return Values

NX_SUCCESS	(0x00)	Successful packet pool delete.
NX_PTR_ERROR	(0x07)	Invalid pool pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

No

Example

```
/*  Delete a previously created packet pool.  */
status = nx_packet_pool_delete(&pool_0);

/*  If status is NX_SUCCESS, the packet pool has been successfully
    deleted.  */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_pool_info_get

Retrieve information about a packet pool

Prototype

```
UINT nx_packet_pool_info_get(NX_PACKET_POOL *pool_ptr,
                             ULONG *total_packets,
                             ULONG *free_packets,
                             ULONG *empty_pool_requests,
                             ULONG *empty_pool_suspensions,
                             ULONG *invalid_packet_releases);
```

Description

This service retrieves information about the specified packet pool.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

pool_ptr	Pointer to previously created packet pool.
total_packets	Pointer to destination for the total number of packets in the pool.
free_packets	Pointer to destination for the total number of currently free packets.
empty_pool_requests	Pointer to destination of the total number of allocation requests when the pool was empty.
empty_pool_suspensions	Pointer to destination of the total number of empty pool suspensions.
invalid_packet_releases	Pointer to destination of the total number of invalid packet releases.

Return Values

NX_SUCCESS	(0x00)	Successful packet pool information retrieval.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/*  Retrieve packet pool information.  */
status = nx_packet_pool_info_get(&pool_0,
                                  &total_packets,
                                  &free_packets,
                                  &empty_pool_requests,
                                  &empty_pool_suspensions,
                                  &invalid_packet_releases);

/*  If status is NX_SUCCESS, packet pool information was
    retrieved. */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_release, nx_packet_transmit_release

nx_packet_release

Release previously allocated packet

Prototype

```
UINT nx_packet_release(NX_PACKET *packet_ptr);
```

Description

This service releases a packet, including any additional packets linked to the specified packet. If another thread is blocked on packet allocation, it is given the packet and resumed.

The application must prevent releasing a packet more than once, because doing so will cause unpredictable results.

Parameters

packet_ptr	Packet pointer.

Return Values

NX_SUCCESS	(0x00)	Successful packet release.
NX_PTR_ERROR	(0x07)	Invalid packet pointer.

Allowed From

Initialization, threads, timers, and ISRs (application network drivers)

Preemption Possible

Yes

Example

```
/*  Release a previously allocated packet.  */
status = nx_packet_release(packet_ptr);

/*  If status is NX_SUCCESS, the packet has been returned to the
    packet pool it was allocated from.  */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_transmit_release

nx_packet_transmit_release

Release a transmitted packet

Prototype

```
UINT nx_packet_transmit_release(NX_PACKET *packet_ptr);
```

Description

For non-TCP packets, this service releases a transmitted packet, including any additional packets linked to the specified packet. If another thread is blocked on packet allocation, it is given the packet and resumed. For a transmitted TCP packet, the packet is marked as being transmitted but not released till the packet is acknowledged. This service is typically called from the application's network driver.

The network driver should remove the physical media header and adjust the length of the packet before calling this service.

Parameters

packet_ptr	Packet pointer.

Return Values

NX_SUCCESS	(0x00)	Successful transmit packet release.
NX_PTR_ERROR	(0x07)	Invalid packet pointer.

Allowed From

Application network drivers (including ISRs)

Preemption Possible

Yes

Example

```
/*  Release a previously allocated packet that was just transmitted
    from the application network driver.  */
status = nx_packet_transmit_release(packet_ptr);

/*  If status is NX_SUCCESS, the transmitted packet has been
    returned to the packet pool it was allocated from.  */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release

nx_rarp_disable

Disable Reverse Address Resolution Protocol (RARP)

Prototype

```
UINT nx_rarp_disable(NX_IP *ip_ptr);
```

Description

This service disables the RARP component of NetX Duo for the specific IP instance. For a multihome system, this service disables RARP on all interfaces.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful RARP disable.
NX_NOT_ENABLED	(0x14)	RARP was not enabled.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Disable RARP on the previously created IP instance.  */
status = nx_rarp_disable(&ip_0);

/*  If status is NX_SUCCESS, RARP is disabled.  */
```

See Also

nx_rarp_enable, nx_rarp_info_get

nx_rarp_enable

Enable Reverse Address Resolution Protocol (RARP)

Prototype

```
UINT nx_rarp_enable(NX_IP *ip_ptr);
```

Description

This service enables the RARP component of NetX Duo for the specific IP instance. Note that the IP instance must be created with an IP address of zero in order to use RARP. A none-zero IP address implies the interface already has valid IP address and thus RARP is not enabled for that interface.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful RARP enable.
NX_IP_ADDRESS_ERROR	(0x21)	IP address is already valid.
NX_ALREADY_ENABLED	(0x15)	RARP was already enabled.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Enable RARP on the previously created IP instance.  */
status = nx_rarp_enable(&ip_0);

/*  If status is NX_SUCCESS, RARP is enabled and is attempting to
    resolve this IP instance's address by querying the network.  */
```

See Also

nx_rarp_disable, nx_rarp_info_get

nx_rarp_info_get

Retrieve information about RARP activities

Prototype

```
UINT nx_rarp_info_get(NX_IP *ip_ptr,
                      ULONG *rarp_requests_sent,
                      ULONG *rarp_responses_received,
                      ULONG *rarp_invalid_messages);
```

Description

This service retrieves information about RARP activities for the specified IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
rarp_requests_sent	Pointer to destination for the total number of RARP requests sent.
rarp_responses_received	Pointer to destination for the total number of RARP responses received.
rarp_invalid_messages	Pointer to destination of the total number of invalid messages.

Return Values

NX_SUCCESS	(0x00)	Successful RARP information retrieval.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/*  Retrieve RARP information from previously created IP
    Instance 0. */
status =  nx_rarp_info_get(&ip_0,
                           &rarp_requests_sent,
                           &rarp_responses_received,
                           &rarp_invalid_messages);

/*  If status is NX_SUCCESS, RARP information was retrieved.  */
```

See Also

nx_rarp_disable, nx_rarp_enable

nx_system_initialize

Initialize NetX Duo System

Prototype

```
VOID nx_system_initialize(VOID);
```

Description

This service initializes the basic NetX Duo system resources in preparation for use. It should be called by the application during initialization and before any other NetX Duo call are made.

Parameters

None

Return Values

None

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/*  Initialize NetX Duo for operation.  */
nx_system_initialize();

/*  At this point, NetX Duo is ready for IP creation and all
    subsequent network operations.  */
```

See Also

None

nx_tcp_client_socket_bind

Bind client TCP socket to TCP port

Prototype

```
UINT nx_tcp_client_socket_bind(NX_TCP_SOCKET *socket_ptr,
                               UINT port,
                               ULONG wait_option);
```

Description

This service binds the previously created TCP client socket to the specified TCP port. Valid TCP sockets range from 0 through 0xFFFF.

Parameters

socket_ptr	Pointer to previously created TCP socket instance.
port	Port number to bind (1 through 0xFFFF). If port number is NX_ANY_PORT (0x0000), the IP instance will search for the next free port and use that for the binding.
wait_option	Defines how the service behaves if the port is already bound to another socket. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket bind.
NX_ALREADY_BOUND	(0x22)	This socket is already bound to another TCP port.
NX_PORT_UNAVAILABLE	(0x23)	Port is already bound to a different socket.
NX_NO_FREE_PORTS	(0x45)	No free port.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.

NX_INVALID_PORT	(0x46)	Invalid port.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/*  Bind a previously created client socket to port 12 and wait for 7
    timer ticks for the bind to complete.  */
status = nx_tcp_client_socket_bind(&client_socket, 12, 7);

/*  If status is NX_SUCCESS, the previously created client_socket is
    bound to port 12 on the associated IP instance.  */
```

See Also

nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_client_socket_connect

Connect client TCP socket

Prototype

```
UINT nx_tcp_client_socket_connect(NX_TCP_SOCKET *socket_ptr,
                                  UINT server_ip,
                                  UINT server_port,
                                  ULONG wait_option)
```

Description

This service connects the previously created TCP client socket to the specified server's port. Valid TCP server ports range from 0 through 0xFFFF.

Parameters

socket_ptr	Pointer to previously created TCP socket instance.
server_ip	Server's IP address.
server_port	Server port number to connect to (1 through 0xFFFF).
wait_option	Defines how the service behaves while the connection is being established. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket connect.
NX_NOT_BOUND	(0x24)	Socket is not bound.
NX_NOT_CLOSED	(0x35)	Socket is not in a closed state.
NX_IN_PROGRESS	(0x37)	No wait was specified, the connection attempt is in progress.
NX_INVALID_INTERFACE	(0x4C)	Invalid interface supplied.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.

NX_IP_ADDRESS_ERROR	(0x21)	Invalid server IP address.
NX_INVALID_PORT	(0x46)	Invalid port.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/*  Initiate a TCP connection from a previously created and bound
    client socket. The connection requested in this example is to
    port 12 on the server with the IP address of 1.2.3.5. This
    service will wait 300 timer ticks for the connection to take
    place before giving up.  */
status = nx_tcp_client_socket_connect(&client_socket,
                                       IP_ADDRESS(1,2,3,5),
                                       12, 300);

/*  If status is NX_SUCCESS, the previously created and bound
    client_socket is connected to port 12 on IP 1.2.3.5.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_client_socket_port_get

Get port number bound to client TCP socket

Prototype

```
UINT nx_tcp_client_socket_port_get(NX_TCP_SOCKET *socket_ptr,
                                   UINT *port_ptr);
```

Description

This service retrieves the port number associated with the socket, which is useful to find the port allocated by NetX Duo in situations where the NX_ANY_PORT was specified at the time the socket was bound.

Parameters

socket_ptr	Pointer to previously created TCP socket instance.
port_ptr	Pointer to destination for the return port number. Valid port numbers are (1 through 0xFFFF).

Return Values

NX_SUCCESS	(0x00)	Successful socket bind.
NX_NOT_BOUND	(0x24)	This socket is not bound to a port.
NX_PTR_ERROR	(0x07)	Invalid socket pointer or port return pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/*  Get the port number of previously created and bound client
    socket.  */
status = nx_tcp_client_socket_port_get(&client_socket, &port);

/*  If status is NX_SUCCESS, the port variable contains the port this
    socket is bound to.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_unbind, nx_tcp_enable,nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, , nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_client_socket_unbind

Unbind TCP client socket from TCP port

Prototype

```
UINT nx_tcp_client_socket_unbind(NX_TCP_SOCKET *socket_ptr);
```

Description

This service releases the binding between the TCP client socket and a TCP port. If there are other threads waiting to bind another socket to the unbound port, the first suspended thread is then bound to the newly unbound port.

Parameters

socket_ptr	Pointer to previously created TCP socket instance.

Return Values

NX_SUCCESS	(0x00)	Successful socket unbind.
NX_NOT_BOUND	(0x24)	Socket was not bound to any port.
NX_NOT_CLOSED	(0x35)	Socket has not been disconnected.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/*  Unbind a previously created and bound client TCP socket.
status = nx_tcp_client_socket_unbind(&client_socket);

/*  If status is NX_SUCCESS, the client socket is no longer
    bound.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_enable

Enable TCP component of NetX Duo

Prototype

```
UINT nx_tcp_enable(NX_IP *ip_ptr);
```

Description

This service enables the Transmission Control Protocol (TCP) component of NetX Duo. After enabled, TCP data may be sent and received by the application.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful TCP enable.
NX_ALREADY_ENABLED	(0x15)	TCP is already enabled.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/*  Enable TCP on a previously created IP instance. /*
status = nx_tcp_enable(&ip_0);

/*  If status is NX_SUCCESS, TCP is enabled on the IP instance.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_free_port_find

Find next available TCP port

Prototype

```
UINT nx_tcp_free_port_find(NX_IP *ip_ptr,
                           UINT port,
                           UINT *free_port_ptr);
```

Description

This service attempts to locate a free TCP port (unbound) starting from the application supplied port. The search logic will wrap around if the search happens to reach the maximum port value of 0xFFFF. If the search is successful, the free port is returned in the variable pointed to by *free_port_ptr*.

This service can be called from another thread and have the same port returned. To prevent this race condition, the application may wish to place this service and the actual client socket bind under the protection of a mutex.

Parameters

ip_ptr	Pointer to previously created IP instance.
port	Port number to start search at (1 through 0xFFFF).
free_port_ptr	Pointer to the destination free port return value.

Return Values

NX_SUCCESS	(0x00)	Successful free port find.
NX_NO_FREE_PORTS	(0x45)	No free ports found.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_INVALID_PORT	(0x46)	The specified port number is invalid.

Allowed From

Threads

Preemption Possible

No

Example

```
/*  Locate a free TCP port, starting at port 12, on a previously
    created IP instance.  */
status = nx_tcp_free_port_find(&ip_0, 12, &free_port);

/*  If status is NX_SUCCESS, "free_port" contains the next free port
    on the IP instance.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_info_get

Retrieve information about TCP activities

Prototype

```
UINT nx_tcp_info_get(NX_IP *ip_ptr,
                          ULONG *tcp_packets_sent,
                          ULONG *tcp_bytes_sent,
                          ULONG *tcp_packets_received,
                          ULONG *tcp_bytes_received,
                          ULONG *tcp_invalid_packets,
                          ULONG *tcp_receive_packets_dropped,
                          ULONG *tcp_checksum_errors,
                          ULONG *tcp_connections,
                          ULONG *tcp_disconnections,
                          ULONG *tcp_connections_dropped,
                          ULONG *tcp_retransmit_packets);
```

Description

This service retrieves information about TCP activities for the specified IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
tcp_packets_sent	Pointer to destination for the total number of TCP packets sent.
tcp_bytes_sent	Pointer to destination for the total number of TCP bytes sent.
tcp_packets_received	Pointer to destination of the total number of TCP packets received.
tcp_bytes_received	Pointer to destination of the total number of TCP bytes received.
tcp_invalid_packets	Pointer to destination of the total number of invalid TCP packets.
tcp_receive_packets_dropped	Pointer to destination of the total number of TCP receive packets dropped.
tcp_checksum_errors	Pointer to destination of the total number of TCP packets with checksum errors.

tcp_connections	Pointer to destination of the total number of TCP connections.
tcp_disconnections	Pointer to destination of the total number of TCP disconnections.
tcp_connections_dropped	Pointer to destination of the total number of TCP connections dropped.
tcp_retransmit_packets	Pointer to destination of the total number of TCP packets retransmitted.

Return Values

NX_SUCCESS	(0x00)	Successful TCP information retrieval.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/*  Retrieve TCP information from previously created IP Instance 0. */
status =  nx_tcp_info_get(&ip_0,
                          &tcp_packets_sent,
                          &tcp_bytes_sent,
                          &tcp_packets_received,
                          &tcp_bytes_received,
                          &tcp_invalid_packets,
                          &tcp_receive_packets_dropped,
                          &tcp_checksum_errors,
                          &tcp_connections,
                          &tcp_disconnections
                          &tcp_connections_dropped,
                          &tcp_retransmit_packets);

/*  If status is NX_SUCCESS, TCP information was retrieved.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_server_socket_accept

Accept TCP connection

Prototype

```
UINT nx_tcp_server_socket_accept(NX_TCP_SOCKET *socket_ptr,
                                  ULONG wait_option);
```

Description

This service accepts (or prepares to accept) a TCP client socket connection request for a port that was previously set up for listening. This service may be called immediately after the application calls the listen or re-listen service or after the listen callback routine is called when the client connection is actually present.

*The application must call **nx_tcp_server_socket_unaccept** after the connection is no longer needed to remove the server socket's binding to the server port.*

Application callback routines are called from within the IP's helper thread.

Parameters

socket_ptr	Pointer to the TCP server socket control block.
wait_option	Defines how the service behaves while the connection is being established. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful TCP server socket accept (passive connect).
NX_NOT_LISTEN_STATE	(0x36)	The server socket supplied is not in a listen state.
NX_IN_PROGRESS	(0x37)	No wait was specified, the connection attempt is in progress.

NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_PTR_ERROR	(0x07)	Socket pointer error.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
NX_PACKET_POOL          my_pool;
NX_IP                   my_ip;
NX_TCP_SOCKET           server_socket;

void   port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{

    /* Simply set the semaphore to wakeup the server thread.  */
    tx_semaphore_put(&port_12_semaphore);
}

void  port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This
       example doesn't use this callback.  */
}

void   port_12_server_thread_entry(ULONG id)
{

NX_PACKET   *my_packet;
UINT        status, i;

    /* Assuming that:
         "port_12_semaphore" has already been created with an
         initial count of 0 "my_ip" has already been created and the
         link is enabled "my_pool" packet pool has already been
         created
    */

    /* Create the server socket.  */
    nx_tcp_socket_create(&my_ip, &server_socket,
                         "Port 12 Server Socket",
                         NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                         NX_IP_TIME_TO_LIVE, 100,
                         NX_NULL, port_12_disconnect_request);
```

```
    /* Setup server listening on port 12.  */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
                                port_12_connect_request);

    /* Loop to process 5 server connections, sending "Hello_and_Goodbye" to
       each client and then disconnecting.  */
    for (i = 0; i < 5; i++)
    {

        /* Get the semaphore that indicates a client connection request is
           present.  */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

        /* Wait for 200 ticks for the client socket connection to complete.*/
        status =  nx_tcp_server_socket_accept(&server_socket, 200);

        /* Check for a successful connection.  */
        if (status == NX_SUCCESS)
        {

            /* Allocate a packet for the "Hello_and_Goodbye" message.  */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                                         NX_WAIT_FOREVER);

            /* Place "Hello_and_Goodbye" in the packet.  */
            nx_packet_data_append(my_packet, "Hello_and_Goodbye",
                                  sizeof("Hello_and_Goodbye"),
                                  &my_pool, NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client.  */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error.  */
            if (status)
            {

                /* Error, release the packet.  */
                nx_packet_release(my_packet);
            }

             /* Now disconnect the server socket from the client.  */
             nx_tcp_socket_disconnect(&server_socket, 200);
        }

        /* Unaccept the server socket.  Note that unaccept is called even if
           disconnect or accept fails.  */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket again.  */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }

    /* We are now done so unlisten on server port 12.  */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket.  */
    nx_tcp_socket_delete(&server_socket);
}
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_server_socket_listen

Enable listening for client connection on TCP port

Prototype

```
UINT nx_tcp_server_socket_listen(NX_IP *ip_ptr, UINT port,
                                 NX_TCP_SOCKET *socket_ptr,
                                 UINT listen_queue_size,
                                 VOID (*listen_callback)
                                 (NX_TCP_SOCKET *socket_ptr,
                                 UINT port));
```

Description

This service enables listening for a client connection request on the specified TCP port. When a client connection request is received, the supplied server socket is bound to the specified port and the supplied listen callback function is called.

The listen callback routine's processing is completely up to the application. It may contain logic to wake up an application thread that subsequently performs an accept operation. If the application already has a thread suspended on accept processing for this socket, the listen callback routine may not be needed.

If the application wishes to handle additional client connections on the same port, the ***nx_tcp_server_socket_relisten*** must be called with an available socket (a socket in the CLOSED state) for the next connection. Until the re-listen service is called, additional client connections are queued. When the maximum queue depth is exceeded, the oldest connection request is dropped in favor of queuing the new connection request. The maximum queue depth is specified by this service.

Application callback routines are called from the internal IP helper thread.

Parameters

ip_ptr	Pointer to previously created IP instance.
port	Port number to listen on (1 through 0xFFFF).
socket_ptr	Pointer to socket to use for the connection.
listen_queue_size	Number of client connection requests that can be queued.
listen_callback	Application function to call when the connection is received. If a NULL is specified, the listen callback feature is disabled.

Return Values

NX_SUCCESS	(0x00)	Successful TCP port listen enable.
NX_MAX_LISTEN	(0x33)	No more listen request structures are available. The constant NX_MAX_LISTEN_REQUESTS in ***nx_api.h*** defines how many active listen requests are possible.
NX_NOT_CLOSED	(0x35)	The supplied server socket is not in a closed state.
NX_ALREADY_BOUND	(0x22)	The supplied server socket is already bound to a port.
NX_DUPLICATE_LISTEN	(0x34)	There is already an active listen request for this port.
NX_INVALID_PORT	(0x46)	Invalid port specified.
NX_PTR_ERROR	(0x07)	Invalid IP or socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
NX_PACKET_POOL          my_pool;
NX_IP                   my_ip;
NX_TCP_SOCKET           server_socket;

void   port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{

    /* Simply set the semaphore to wakeup the server thread.  */
    tx_semaphore_put(&port_12_semaphore);
}

void  port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This example
       doesn't use this callback.  */
}

void   port_12_server_thread_entry(ULONG id)
{

NX_PACKET   *my_packet;
UINT        status, i;

    /* Assuming that:
         "port_12_semaphore" has already been created with an initial count
         of 0 "my_ip" has already been created and the link is enabled
         "my_pool" packet pool has already been created
    */

    /* Create the server socket.  */
    nx_tcp_socket_create(&my_ip, &server_socket, "Port 12 Server Socket",
                         NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                         NX_IP_TIME_TO_LIVE, 100,
                         NX_NULL, port_12_disconnect_request);

    /* Setup server listening on port 12.  */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
                                port_12_connect_request);

    /* Loop to process 5 server connections, sending "Hello_and_Goodbye" to
       each client and then disconnecting.  */
    for (i = 0; i < 5; i++)
    {

         /* Get the semaphore that indicates a client connection request is
            present.  */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

        /* Wait for 200 ticks for the client socket connection to complete.*/
        status =  nx_tcp_server_socket_accept(&server_socket, 200);

        /* Check for a successful connection.  */
        if (status == NX_SUCCESS)
        {

            /* Allocate a packet for the "Hello_and_Goodbye" message.  */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                               NX_WAIT_FOREVER);

            /* Place "Hello_and_Goodbye" in the packet.  */
```

```
            nx_packet_data_append(my_packet, "Hello_and_Goodbye",
                                  sizeof("Hello_and_Goodbye"),
                                  &my_pool,
                                  NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client.  */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error.  */
            if (status)
            {

                /* Error, release the packet.  */
                nx_packet_release(my_packet);
            }

            /* Now disconnect the server socket from the client.  */
            nx_tcp_socket_disconnect(&server_socket, 200);
        }

        /* Unaccept the server socket.  Note that unaccept is called
           even if disconnect or accept fails.  */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket
           again.  */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }

    /* We are now done so unlisten on server port 12.  */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket.  */
    nx_tcp_socket_delete(&server_socket);
}
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get,nx_tcp_server_socket_accept, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_server_socket_relisten

Re-listen for client connection on TCP port

Prototype

```
UINT nx_tcp_server_socket_relisten(NX_IP *ip_ptr, UINT port,
                                   NX_TCP_SOCKET *socket_ptr);
```

Description

This service is called after a connection has been received on a port that was setup previously for listening. The main purpose of this service is to provide a new server socket for the next client connection. If a connection request is queued, the connection will be processed immediately during this service call.

The same callback routine specified by the original listen request is also called when a connection is present for this new server socket.

Parameters

ip_ptr	Pointer to previously created IP instance.
port	Port number to re-listen on (1 through 0xFFFF).
socket_ptr	Socket to use for the next client connection.

Return Values

NX_SUCCESS	(0x00)	Successful TCP port re-listen.
NX_NOT_CLOSED	(0x35)	The supplied server socket is not in a closed state.
NX_ALREADY_BOUND	(0x22)	The supplied server socket is already bound to a port.
NX_INVALID_RELISTEN	(0x47)	There is already a valid socket pointer for this port or the port specified does not have a listen request active.
NX_CONNECTION_PENDING	(0x48)	Same as NX_SUCCESS, except there was a queued connection request and it was processed during this call.

NX_INVALID_PORT	(0x46)	Invalid port specified.
NX_PTR_ERROR	(0x07)	Invalid IP or listen callback pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
NX_PACKET_POOL          my_pool;
NX_IP                   my_ip;
NX_TCP_SOCKET           server_socket;

void   port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{

    /* Simply set the semaphore to wakeup the server thread.  */
    tx_semaphore_put(&port_12_semaphore);
}

void  port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This
       example doesn't use this callback.  */
}

void   port_12_server_thread_entry(ULONG id)
{

NX_PACKET   *my_packet;
UINT        status, i;

    /* Assuming that:
       "port_12_semaphore" has already been created with an initial count
        of 0.
        "my_ip" has already been created and the link is enabled.
        "my_pool" packet pool has already been created.   */

    /* Create the server socket.  */
    nx_tcp_socket_create(&my_ip, &server_socket, "Port 12 Server Socket",
                                 NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                                 NX_IP_TIME_TO_LIVE, 100,
                                 NX_NULL, port_12_disconnect_request);

    /* Setup server listening on port 12.  */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
                                port_12_connect_request);
```

```
/* Loop to process 5 server connections, sending
   "Hello_and_Goodbye" to each client then disconnecting.  */
for (i = 0; i < 5; i++)
{

   /* Get the semaphore that indicates a client connection
      request is present.  */
   tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

    /* Wait for 200 ticks for the client socket connection to
       complete.  */
   status =  nx_tcp_server_socket_accept(&server_socket, 200);

   /* Check for a successful connection.  */
   if (status == NX_SUCCESS)
   {

        /* Allocate a packet for the "Hello_and_Goodbye"
           message.  */
       nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                          NX_WAIT_FOREVER);

        /* Place "Hello_and_Goodbye" in the packet.  */
        nx_packet_data_append(my_packet, "Hello_and_Goodbye",
                              sizeof("Hello_and_Goodbye"),
                              &my_pool, NX_WAIT_FOREVER);

        /* Send "Hello_and_Goodbye" to client.  */
        nx_tcp_socket_send(&server_socket, my_packet, 200);

        /* Check for an error.  */
        if (status)
        {

            /* Error, release the packet.  */
            nx_packet_release(my_packet);
        }

        /* Now disconnect the server socket from the client. */
        nx_tcp_socket_disconnect(&server_socket, 200);
    }

    /* Unaccept the server socket.  Note that unaccept is
       called even if disconnect or accept fails.  */
    nx_tcp_server_socket_unaccept(&server_socket);

    /* Setup server socket for listening with this socket
       again.  */
    nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
}

/* We are now done so unlisten on server port 12.  */
nx_tcp_server_socket_unlisten(&my_ip, 12);

/* Delete the server socket.  */
nx_tcp_socket_delete(&server_socket);
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_server_socket_unaccept

Unaccept previous server socket connection

Prototype

```
UINT nx_tcp_server_socket_unaccept(NX_TCP_SOCKET *socket_ptr);
```

Description

This service removes the association between this server socket and the specified server port. The application must call this service after a disconnection or after an unsuccessful accept call.

Parameters

socket_ptr	Pointer to previously setup server socket instance.

Return Values

NX_SUCCESS	(0x00)	Successful server socket unaccept.
NX_NOT_LISTEN_STATE	(0x36)	Server socket is in an improper state, and is probably not disconnected.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
NX_PACKET_POOL          my_pool;
NX_IP                   my_ip;
NX_TCP_SOCKET           server_socket;

void   port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{

    /* Simply set the semaphore to wakeup the server thread.  */
    tx_semaphore_put(&port_12_semaphore);
}

void  port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This example
       doesn't use this callback.  */
}

void   port_12_server_thread_entry(ULONG id)
{

NX_PACKET   *my_packet;
UINT        status, i;

    /* Assuming that:
       "port_12_semaphore" has already been created with an initial count
       of 0 "my_ip" has already been created and the link is enabled
       "my_pool" packet pool has already been created
    */

    /* Create the server socket.  */
    nx_tcp_socket_create(&my_ip, &server_socket, "Port 12 Server Socket",
                         NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                         NX_IP_TIME_TO_LIVE, 100,
                         NX_NULL, port_12_disconnect_request);

    /* Setup server listening on port 12.  */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
                                port_12_connect_request);

    /* Loop to process 5 server connections, sending "Hello_and_Goodbye" to
       each client and then disconnecting.  */
    for (i = 0; i < 5; i++)
    {

        /* Get the semaphore that indicates a client connection request is
           present.  */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

        /* Wait for 200 ticks for the client socket connection to
           complete.*/
        status =  nx_tcp_server_socket_accept(&server_socket, 200);

        /* Check for a successful connection.  */
        if (status == NX_SUCCESS)
        {

            /* Allocate a packet for the "Hello_and_Goodbye" message.  */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                                             NX_WAIT_FOREVER);
```

```
            /* Place "Hello_and_Goodbye" in the packet.  */
            nx_packet_data_append(my_packet,
                       "Hello_and_Goodbye",sizeof("Hello_and_Goodbye"),
                       &my_pool, NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client.  */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error.  */
            if (status)
            {

                /* Error, release the packet.  */
                nx_packet_release(my_packet);
            }

             /* Now disconnect the server socket from the client.  */
             nx_tcp_socket_disconnect(&server_socket, 200);
        }

        /* Unaccept the server socket.  Note that unaccept is called even
           if disconnect or accept fails.  */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket again.  */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }

    /* We are now done so unlisten on server port 12.  */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket.  */
    nx_tcp_socket_delete(&server_socket);
}
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get,nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_server_socket_unlisten

Disable listening for client connection on TCP port

Prototype

```
UINT nx_tcp_server_socket_unlisten(NX_IP *ip_ptr, UINT port);
```

Description

This service disables listening for a client connection request on the specified TCP port.

Parameters

ip_ptr	Pointer to previously created IP instance.
port	Number of port to disable listening (0 through 0xFFFF).

Return Values

NX_SUCCESS	(0x00)	Successful TCP listen disable.
NX_ENTRY_NOT_FOUND	(0x16)	Listening was not enabled for thespecified port.
NX_INVALID_PORT	(0x46)	Invalid port specified.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
NX_PACKET_POOL          my_pool;
NX_IP                   my_ip;
NX_TCP_SOCKET           server_socket;

void   port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{

    /* Simply set the semaphore to wakeup the server thread.  */
    tx_semaphore_put(&port_12_semaphore);
}

void  port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This exmaple
       doesn't use this callback.  */
}

void   port_12_server_thread_entry(ULONG id)
{

NX_PACKET   *my_packet;
UINT        status, i;

    /* Assuming that:
       "port_12_semaphore" has already been created with an initial count
       of 0 "my_ip" has already been created and the link is enabled
       "my_pool" packet pool has already been created
    */

    /* Create the server socket.  */
    nx_tcp_socket_create(&my_ip, &server_socket, "Port 12 Server Socket",
                         NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                         NX_IP_TIME_TO_LIVE, 100,
                         NX_NULL, port_12_disconnect_request);

    /* Setup server listening on port 12.  */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
                                port_12_connect_request);

    /* Loop to process 5 server connections, sending "Hello_and_Goodbye" to
       each client and then disconnecting.  */
    for (i = 0; i < 5; i++)
    {

       /* Get the semaphore that indicates a client connection request is
          present.  */
       tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

       /* Wait for 200 ticks for the client socket connection to complete.*/
       status =  nx_tcp_server_socket_accept(&server_socket, 200);

       /* Check for a successful connection.  */
       if (status == NX_SUCCESS)
       {

            /* Allocate a packet for the "Hello_and_Goodbye" message.  */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                               NX_WAIT_FOREVER);
```

```
            /* Place "Hello_and_Goodbye" in the packet.  */
            nx_packet_data_append(my_packet, "Hello_and_Goodbye",
                                  sizeof("Hello_and_Goodbye"), &my_pool,
                                  NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client.  */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error.  */
            if (status)
            {

                /* Error, release the packet.  */
                nx_packet_release(my_packet);
            }

             /* Now disconnect the server socket from the client.  */
             nx_tcp_socket_disconnect(&server_socket, 200);
        }

        /* Unaccept the server socket.  Note that unaccept is called even if
           disconnect or accept fails.  */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket again.  */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }

    /* We are now done so unlisten on server port 12.  */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket.  */
    nx_tcp_socket_delete(&server_socket);
}
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_bytes_available

Retrieves number of bytes available for retrieval

Prototype

```
UINT nx_tcp_socket_bytes_available(NX_TCP_SOCKET *socket_ptr,
                                   ULONG *bytes_available);
```

Description

This retrieves number of bytes available for retrieval in the specified TCP socket. Note that the TCP socket must already be connected.

Parameters

socket_ptr	Pointer to previously created and connected TCP socket.
bytes_available	Pointer to destination for bytes available.

Return Values

NX_SUCCESS	(0x00)	Service executes successfully. Number of bytes available for read is returned to the caller.
NX_NOT_CONNECTED	(0x38)	Socket is not in a connected state.
NX_PTR_ERROR	(0x07)	Invalid pointers.
NX_NOT_ENABLED	(0x14)	TCP is not enabled.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Get the bytes available for retrieval on the specified socket. */
status = nx_tcp_socket_bytes_available(&my_socket,&bytes_available);

/* Is status = NX_SUCCESS, the available bytes is returned in
   bytes_available. */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get,nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_create

Create TCP client or server socket

Prototype

```
UINT nx_tcp_socket_create(NX_IP *ip_ptr, NX_TCP_SOCKET *socket_ptr, CHAR *name,
                          ULONG type_of_service, ULONG fragment,
                          UINT time_to_live, ULONG window_size,
                          VOID (*urgent_data_callback)(NX_TCP_SOCKET *socket_ptr),
                          VOID (*disconnect_callback)(NX_TCP_SOCKET *socket_ptr));
```

Description

This service creates a TCP client or server socket for the specified IP instance.

Application callback routines are called from the thread associated with this IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
socket_ptr	Pointer to new TCP client socket control block.
name	Application name for this TCP socket.
type_of_service	Defines the type of service for the transmission, legal values are as follows: NX_IP_NORMAL (0x00000000) NX_IP_MIN_DELAY (0x00100000) NX_IP_MAX_DATA (0x00080000) NX_IP_MAX_RELIABLE (0x00040000) NX_IP_MIN_COST (0x00020000)
fragment	Specifies whether or not IP fragmenting is allowed. If NX_FRAGMENT_OKAY (0x0) is specified, IP fragmenting is allowed. If NX_DONT_FRAGMENT (0x4000) is specified, IP fragmenting is disabled.
time_to_live	Specifies the 8-bit value that defines how many routers this packet can pass before being thrown away. The default value is specified by NX_IP_TIME_TO_LIVE.

window_size	Defines the maximum number of bytes allowed in the receive queue for this socket
urgent_data_callback	Application function that is called whenever urgent data is detected in the receive stream. If this value is NX_NULL, urgent data is ignored.
disconnect_callback	Application function that is called whenever a disconnect is issued by the socket at the other end of the connection. If this value is NX_NULL, the disconnect callback function is disabled.

Return Values

NX_SUCCESS	(0x00)	Successful TCP client socket create.
NX_OPTION_ERROR	(0x0A)	Invalid type-of-service, fragment, or time-to-live option.
NX_PTR_ERROR	(0x07)	Invalid IP or socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization and Threads

Preemption Possible

No

Example

```
/* Create a TCP client socket on the previously created IP instance,
   with normal delivery, IP fragmentation enabled, 0x80 time to
   live, a 200-byte receive window, no urgent callback routine, and
   the "client_disconnect" routine to handle disconnection initiated
   from the other end of the connection.  */
status =  nx_tcp_socket_create(&ip_0, &client_socket,
                               "Client Socket",
                               NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                               0x80, 200, NX_NULL
                               client_disconnect);

/* If status is NX_SUCCESS, the client socket is created and ready
   to be bound.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_delete

Delete TCP socket

Prototype

```
UINT nx_tcp_socket_delete(NX_TCP_SOCKET *socket_ptr);
```

Description

This service deletes a previously created TCP socket.

Parameters

socket_ptr	Previously created TCP socket

Return Values

NX_SUCCESS	(0x00)	Successful socket delete.
NX_NOT_CREATED	(0x27)	Socket was not created.
NX_STILL_BOUND	(0x42)	Socket is still bound.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Delete a previously created TCP client socket.  */
status =  nx_tcp_socket_delete(&client_socket);

/* If status is NX_SUCCESS, the client socket is deleted.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_disconnect

Disconnect client and server socket connections

Prototype

```
UINT nx_tcp_socket_disconnect(NX_TCP_SOCKET *socket_ptr,
                              ULONG wait_option);
```

Description

This service disconnects an established client or server socket connection. A disconnect of a server socket should be followed by an un-accept request, while a client socket that is disconnected is left in a state ready for another connection request.

Parameters

socket_ptr	Pointer to previously connected client or server socket instance.
wait_option	Defines how the service behaves while the disconnection is in progress. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket disconnect.
NX_NOT_CONNECTED	(0x38)	Specified socket is not connected.
NX_IN_PROGRESS	(0x37)	Disconnect is in progress, no wait was specified.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.

NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Disconnect from a previously established connection and wait a
   maximum of 400 timer ticks.  */
status =  nx_tcp_socket_disconnect(&client_socket, 400);

/* If status is NX_SUCCESS, the previously connected socket (either
   as a result of the client socket connect or the server accept) is
   disconnected.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get,nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_info_get

Retrieve information about TCP socket activities

Prototype

```
UINT nx_tcp_socket_info_get(NX_TCP_SOCKET *socket_ptr,
                            ULONG *tcp_packets_sent,
                            ULONG *tcp_bytes_sent,
                            ULONG *tcp_packets_received,
                            ULONG *tcp_bytes_received,
                            ULONG *tcp_retransmit_packets,
                            ULONG *tcp_packets_queued,
                            ULONG *tcp_checksum_errors,
                            ULONG *tcp_socket_state,
                            ULONG *tcp_transmit_queue_depth,
                            ULONG *tcp_transmit_window,
                            ULONG *tcp_receive_window);
```

Description

This service retrieves information about TCP socket activities for the specified TCP socket instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

socket_ptr	Pointer to previously created TCP socket instance.
tcp_packets_sent	Pointer to destination for the total number of TCP packets sent on socket.
tcp_bytes_sent	Pointer to destination for the total number of TCP bytes sent on socket.
tcp_packets_received	Pointer to destination of the total number of TCP packets received on socket.
tcp_bytes_received	Pointer to destination of the total number of TCP bytes received on socket.
tcp_retransmit_packets	Pointer to destination of the total number of TCP packet retransmissions.
tcp_packets_queued	Pointer to destination of the total number of queued TCP packets on socket.

tcp_checksum_errors	Pointer to destination of the total number of TCP packets with checksum errors on socket.
tcp_socket_state	Pointer to destination of the socket's current state.
tcp_transmit_queue_depth	Pointer to destination of the total number of transmit packets still queued waiting for ACK.
tcp_transmit_window	Pointer to destination of the current transmit window size.
tcp_receive_window	Pointer to destination of the current receive window size.

Return Values

NX_SUCCESS	(0x00)	Successful TCP socket information retrieval.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve TCP socket information from previously created socket 0.  */
status =  nx_tcp_socket_info_get(&socket_0,
                                 &tcp_packets_sent,
                                 &tcp_bytes_sent,
                                 &tcp_packets_received,
                                 &tcp_bytes_received,
                                 &tcp_retransmit_packets,
                                 &tcp_packets_queued,
                                 &tcp_checksum_errors,
                                 &tcp_socket_state,
                                 &tcp_transmit_queue_depth,
                                 &tcp_transmit_window,
                                 &tcp_receive_window);

/* If status is NX_SUCCESS, TCP socket information was retrieved.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_mss_get

Get MSS of socket

Prototype

```
UINT  nx_tcp_socket_mss_get(NX_TCP_SOCKET *socket_ptr, ULONG *mss);
```

Description

This service retrieves the specified socket's current Maximum Segment Size (MSS).

Parameters

socket_ptr	Pointer to previously created socket.
mss	Destination for returning MSS.

Return Values

NX_SUCCESS	(0x00)	Successful MSS get.
NX_PTR_ERROR	(0x07)	Invalid socket or MSS destination pointer.
NX_NOT_ENABLED	(0x14)	TCP is not enabled.
NX_CALLER_ERROR	(0x11)	Caller is not a thread or initialization.

Allowed From

Initialization and threads

Example

```
/* Get the MSS for the socket "my_socket". */
status =  nx_tcp_socket_mss_get(&my_socket, &mss_value);

/* If status is NX_SUCCESS, the "mss_value" variable contains the
   socket's current MSS value.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_receive, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_mss_peer_get

Get MSS of socket peer

Prototype

```
UINT  nx_tcp_socket_mss_peer_get(NX_TCP_SOCKET *socket_ptr,
                                 ULONG *mss);
```

Description

This service retrieves the specified socket connected peer's advertised Maximum Segment Size (MSS).

Parameters

socket_ptr	Pointer to previously created and connected socket.
mss	Destination for returning the MSS.

Return Values

NX_SUCCESS	(0x00)	Successful peer MSS get.
NX_PTR_ERROR	(0x07)	Invalid socket or MSS destination pointer.
NX_NOT_ENABLED	(0x14)	TCP is not enabled.
NX_CALLER_ERROR	(0x11)	Caller is not a thread or initialization.

Allowed From

Initialization and threads

Example

```
/* Get the MSS of the connected peer to the socket "my_socket". */
status =  nx_tcp_socket_mss_peer_get(&my_socket, &mss_value);

/* If status is NX_SUCCESS, the "mss_value" variable contains the socket
   peer's advertised MSS value.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_set, nx_tcp_socket_receive, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_mss_set

Set MSS of socket

Prototype

```
UINT  nx_tcp_socket_mss_set(NX_TCP_SOCKET *socket_ptr, ULONG mss);
```

Description

This service sets the specified socket's Maximum Segment Size (MSS). Note the MSS value must be within the network interface MTU, allowing room for IP and TCP headers.

Parameters

socket_ptr	Pointer to previously created socket.
mss	Value of MSS to set.

Return Values

NX_SUCCESS	(0x00)	Successful MSS set.
NX_SIZE_ERROR	(0x09)	Specified MSS value is too large.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_NOT_ENABLED	(0x14)	TCP is not enabled.
NX_CALLER_ERROR	(0x11)	Caller is not a thread or initialization.

Allowed From

Initialization and threads

Example

```
/* Set the MSS of the socket "my_socket" to 1000 bytes. */
status =  nx_tcp_socket_mss_set(&my_socket, 1000);

/* If status is NX_SUCCESS, the MSS of "my_socket" is 1000 bytes. */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_peer_info_get

Retrieve information about peer TCP socket

Prototype

```
UINT nx_tcp_socket_peer_info_get(NX_TCP_SOCKET *socket_ptr,
                                 ULONG *peer_ip_address,
                                 ULONG *peer_port);
```

Description

This service retrieves IP address and port number of the peer socket for a connection.

Parameters

socket_ptr	Pointer to previously created TCP socket.
peer_ip_address	Pointer to destination for peer IP address, in host byte order.
peer_port	Pointer to destination for peer port number, in host byte order.

Return Values

NX_SUCCESS	(0x00)	Service executes successfully. Peer IP address and port number are returned to the caller.
NX_NOT_CONNECTED	(0x38)	Socket is not in a connected state.
NX_PTR_ERROR	(0x07)	Invalid pointers.
NX_NOT_ENABLED	(0x14)	TCP is not enabled.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Obtain peer IP address and port on the specified TCP socket. */
status = nx_tcp_socket_peer_info_get(&my_socket, &peer_ip_address,
                                     &peer_port);

/* If status = NX_SUCCESS, the data was successfully obtained. */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_receive

Receive data from TCP socket

Prototype

```
UINT nx_tcp_socket_receive(NX_TCP_SOCKET *socket_ptr,
                           NX_PACKET **packet_ptr,
                           ULONG wait_option);
```

Description

This service receives TCP data from the specified socket. If no data is queued on the specified socket, the caller suspends based on the supplied wait option.

If NX_SUCCESS is returned, the application is responsible for releasing the received packet when it is no longer needed.

Parameters

socket_ptr	Pointer to previously created TCP socket instance.
packet_ptr	Pointer to TCP packet pointer.
wait_option	Defines how the service behaves if do data are currently queued on this socket. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket data receive.
NX_NOT_BOUND	(0x24)	Socket is not bound yet.
NX_NO_PACKET	(0x01)	No data received.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.

NX_NOT_CONNECTED	(0x38)	The socket is no longer connected.
NX_PTR_ERROR	(0x07)	Invalid socket or return packet pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Receive a packet from the previously created and connected TCP client
   socket. If no packet is available, wait for 200 timer ticks before
   giving up.  */
status =  nx_tcp_socket_receive(&client_socket, &packet_ptr, 200);

/* If status is NX_SUCCESS, the received packet is pointed to by
   "packet_ptr".  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_receive_notify

Notify application of received packets

Prototype

```
UINT nx_tcp_socket_receive_notify(NX_TCP_SOCKET *socket_ptr, VOID
                                  (*tcp_receive_notify)
                                  (NX_TCP_SOCKET *socket_ptr));
```

Description

This service sets the receive notify function pointer to the callback function specified by the application. This callback function is then called whenever one or more packets are received on the socket. If a NX_NULL pointer is supplied, the notify function is disabled.

Parameters

socket_ptr	Pointer to the TCP socket.
tcp_receive_notify	Application callback function pointer that is called when one or more packets are received on the socket.

Return Values

NX_SUCCESS	(0x00)	Successful socket receive notify.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
/* Setup a receive packet callback function for the "client_socket"
   socket. */
status = nx_tcp_socket_receive_notify(&client_socket,
                                      my_receive_notify);

/* If status is NX_SUCCESS, NetX Duo will call the function named
   "my_receive_notify" whenever one or more packets are received for
   "client_socket". */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_send

Send data through a TCP socket

Prototype

```
UINT nx_tcp_socket_send(NX_TCP_SOCKET *socket_ptr,
                        NX_PACKET *packet_ptr,
                        ULONG wait_option);
```

Description

This service sends TCP data through a previously connected TCP socket. If the receiver's last advertised window size is less than this request, the service optionally suspends based on the wait options specified. This service guarantees that no packet data larger than MSS is sent to the IP layer.

! *Unless an error is returned, the application should not release the packet after this call. Doing so will cause unpredictable results because the network driver will also try to release the packet after transmission.*

Parameters

socket_ptr	Pointer to previously connected TCP socket instance.
packet_ptr	TCP data packet pointer.
wait_option	Defines how the service behaves if the request is greater than the window size of the receiver. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket send.
NX_NOT_BOUND	(0x24)	Socket was not bound to any port.
NX_NO_INTERFACE_ADDRESS		

	(0x50)	No suitable outgoing interface found.
NX_NOT_CONNECTED	(0x38)	Socket is no longer connected.
NX_ALREADY_SUSPENDED	(0x40)	Another thread is already suspended trying to send data on this socket. Only one thread is allowed.
NX_WINDOW_OVERFLOW	(0x39)	Request is greater than receiver's advertised window size in bytes.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_INVALID_PACKET	(0x12)	Packet is not allocated.
NX_TX_QUEUE_DEPTH	(0x49)	Maximum transmit queue depth has been reached.
NX_OVERFLOW	(0x03)	Packet append pointer is invalid.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_UNDERFLOW	(0x02)	Packet prepend pointer is invalid.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Send a packet out on the previously created and connected TCP
   socket. If the receive window on the other side of the connection
   is less than the packet size, wait 200 timer ticks before giving
   up.  */
status =  nx_tcp_socket_send(&client_socket, packet_ptr, 200);

/* If status is NX_SUCCESS, the packet has been sent!  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_state_wait

Wait for TCP socket to enter specific state

Prototype

```
UINT nx_tcp_socket_state_wait(NX_TCP_SOCKET *socket_ptr,
                              UINT desired_state,
                              ULONG wait_option);
```

Description

This service waits for the socket to enter the desired state.

Parameters

socket_ptr	Pointer to previously connected TCP socket instance.
desired_state	Desired TCP state. Valid TCP socket states are defined as follows: NX_TCP_CLOSED (0x01) NX_TCP_LISTEN_STATE (0x02) NX_TCP_SYN_SENT (0x03) NX_TCP_SYN_RECEIVED (0x04) NX_TCP_ESTABLISHED (0x05) NX_TCP_CLOSE_WAIT (0x06) NX_TCP_FIN_WAIT_1 (0x07) NX_TCP_FIN_WAIT_2 (0x08) NX_TCP_CLOSING (0x09) NX_TCP_TIMED_WAIT (0x0A) NX_TCP_LAST_ACK (0x0B)
wait_option	Defines how the service behaves if the requested state is not present. The wait options are defined as follows: NX_NO_WAIT (0x00000000) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful state wait.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_NOT_SUCCESSFUL	(0x43)	State not present within the specified wait time.

NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_OPTION_ERROR	(0x0A)	The desired socket state is invalid.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Wait 300 timer ticks for the previously created socket to enter the
   established state in the TCP state machine. */
status =  nx_tcp_socket_state_wait(&client_socket,
                                   NX_TCP_ESTABLISHED, 300);

/* If status is NX_SUCCESS, the socket is now in the established
   state! */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_transmit_configure

Configure socket's transmit parameters

Prototype

```
UINT  nx_tcp_socket_transmit_configure(NX_TCP_SOCKET *socket_ptr,
                                        ULONG max_queue_depth,
                                        ULONG timeout,
                                        ULONG max_retries,
                                        ULONG timeout_shift);
```

Description

This service configures various transmit parameters of the specified TCP socket.

Parameters

socket_ptr	Pointer to the TCP socket.
max_queue_depth	Maximum number of packets allowed to be queued for transmission.
timeout	Number of ThreadX timer ticks an ACK is waited for before the packet is sent again.
max_retries	Maximum number of retries allowed.
timeout_shift	Value to shift the timeout for each subsequent retry. A value of 0, results in the same timeout between successive retries. A value of 1, doubles the timeout between retries.

Return Values

NX_SUCCESS	(0x00)	Successful transmit socket configure.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_OPTION_ERROR	(0x0a)	Invalid queue depth option.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
/* Configure the "client_socket" for a maximum transmit queue depth of 12, 100
   tick timeouts, a maximum of 20 retries, and a timeout double on each
   successive retry. */
status = nx_tcp_socket_transmit_configure(&client_socket,12,100,20,1);

/* If status is NX_SUCCESS, the socket's transmit retry has been configured.
   */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_window_update_notify_set

Notify application of window size updates

Prototype

```
UINT nx_tcp_socket_window_update_notify_set(NX_TCP_SOCKET
                                            *socket_ptr,
                                            VOID
                                            (*tcp_window_update_notify)
                                            (NX_TCP_SOCKET *socket_ptr))
```

Description

This service installs a socket window update callback routine. This routine is called automatically whenever the specified socket receives a packet indicating an increase in the window size of the remote host.

Parameters

socket_ptr	Pointer to previously created TCP socket.
tcp_window_update_notify	Callback routine to be called when the window size changes. A value of NULL disables the window change update.

Return Values

NX_SUCCESS	(0x00)	Callback routine is installed on the socket.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid pointers.
NX_NOT_ENABLED	(0x14)	TCP feature is not enabled.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Set the function pointer to the windows update callback after creating the
   socket. */
status = nx_tcp_socket_window_update_notify_set(&data_socket,
                                                 my_windows_update_callback);
/* Define the window callback function in the host application. */
void    my_windows_update_callback(&data_socket)
{

    /* Process update on increase TCP transmit socket window size. */
    return;
}
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get,nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure

nx_udp_enable

Enable UDP component of NetX Duo

Prototype

```
UINT nx_udp_enable(NX_IP *ip_ptr);
```

Description

This service enables the User Datagram Protocol (UDP) component of NetX Duo. After enabled, UDP datagrams may be sent and received by the application.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful UDP enable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_ALREADY_ENABLED	(0x15)	This component has already been enabled.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Enable UDP on the previously created IP instance.  */
status =  nx_udp_enable(&ip_0);

/* If status is NX_SUCCESS, UDP is now enabled on the specified IP
   instance.  */
```

See Also

nx_udp_free_port_find, nx_udp_info_get, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_free_port_find

Find next available UDP port

Prototype

```
UINT nx_udp_free_port_find(NX_IP *ip_ptr, UINT port,
                           UINT *free_port_ptr);
```

Description

This service starts looking for a free UDP port (unbound) starting from the application supplied port. The search logic will wrap around if the search happens to reach the maximum port value of 0xFFFF. If the search is successful, the free port is returned in the variable pointed to by free_port_ptr.

This service can be called from another thread and can have the same port returned. To prevent this race condition, the application may wish to place this service and the actual socket bind under the protection of a mutex.

Parameters

ip_ptr	Pointer to previously created IP instance.
port	Port number to start search (1 through 0xFFFF).
free_port_ptr	Pointer to the destination free port return variable.

Return Values

NX_SUCCESS	(0x00)	Successful free port find.
NX_NO_FREE_PORTS	(0x45)	No free ports found.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_INVALID_PORT	(0x46)	Specified port number is invalid.

Allowed From

Threads, timers

Preemption Possible

No

Example

```
/* Locate a free UDP port, starting at port 12, on a previously
   created IP instance.  */
status = nx_udp_free_port_find(&ip_0, 12, &free_port);

/* If status is NX_SUCCESS pointer, "free_port" identifies the next
   free UDP port on the IP instance.  */
```

See Also

nx_udp_enable, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_info_get

Retrieve information about UDP activities

Prototype

```
UINT nx_udp_info_get(NX_IP *ip_ptr,
                     ULONG *udp_packets_sent,
                     ULONG *udp_bytes_sent,
                     ULONG *udp_packets_received,
                     ULONG *udp_bytes_received,
                     ULONG *udp_invalid_packets,
                     ULONG *udp_receive_packets_dropped,
                     ULONG *udp_checksum_errors);
```

Description

This service retrieves information about UDP activities for the specified IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
udp_packets_sent	Pointer to destination for the total number of UDP packets sent.
udp_bytes_sent	Pointer to destination for the total number of UDP bytes sent.
udp_packets_received	Pointer to destination of the total number of UDP packets received.
udp_bytes_received	Pointer to destination of the total number of UDP bytes received.
udp_invalid_packets	Pointer to destination of the total number of invalid UDP packets.
udp_receive_packets_dropped	Pointer to destination of the total number of UDP receive packets dropped.
udp_checksum_errors	Pointer to destination of the total number of UDP packets with checksum errors.

Return Values

NX_SUCCESS	(0x00)	Successful UDP information retrieval.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve UDP information from previously created IP Instance 0.  */
status =  nx_udp_info_get(&ip_0, &udp_packets_sent,
                                 &udp_bytes_sent,
                                 &udp_packets_received,
                                 &udp_bytes_received,
                                 &udp_invalid_packets,
                                 &udp_receive_packets_dropped,
                                 &udp_checksum_errors);

/* If status is NX_SUCCESS, UDP information was retrieved.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_packet_info_extract

Extract network parameters from UDP packet

Prototype

```
UINT nx_udp_packet_info_extract(NX_PACKET *packet_ptr,
                                ULONG *ip_address,
                                UINT *protocol,
                                UINT *port,
                                UINT *interface_index);
```

Description

This function extracts network parameters from a packet received on an incoming interface.

Parameters

packet_ptr	Pointer to packet.
ip_address	Pointer to sender IP address.
protocol	Pointer to protocol (UDP).
port	Pointer to sender's port number.
interface_index	Pointer to receiving interface index.

Return Values

NX_SUCCESS	(0x00)	Packet interface data successfully extracted.
NX_INVALID_PACKET	(0x12)	Packet does not contain IPv4 frame.
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Initialization, threads, timers, ISRs

Preemption Possible

No

Example

```
/* Extract network data from UDP packet interface.  */
status = nx_udp_packet_info_extract( packet_ptr, &ip_address,
                                                 &protocol, &port,
                                                 &interface_index)

/* If status is NX_SUCCESS packet data was successfully retrieved. */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_bind

Bind UDP socket to UDP port

Prototype

```
UINT nx_udp_socket_bind(NX_UDP_SOCKET *socket_ptr, UINT port,
                        ULONG wait_option);
```

Description

This service binds the previously created UDP socket to the specified UDP port. Valid UDP sockets range from 0 through 0xFFFF.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.
port	Port number to bind to (1 through 0xFFFF). If port number is NX_ANY_PORT (0x0000), the IP instance will search for the next free port and use that for the binding.
wait_option	Defines how the service behaves if the port is already bound to another socket. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket bind.
NX_ALREADY_BOUND	(0x22)	This socket is already bound to another port.
NX_PORT_UNAVAILABLE	(0x23)	Port is already bound to a different socket.
NX_NO_FREE_PORTS	(0x45)	No free port.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_INVALID_PORT	(0x46)	Invalid port specified.

NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Bind the previously created UDP socket to port 12 on the previously
   created IP instance. If the port is already bound, wait for 300 timer
   ticks before giving up. */
status = nx_udp_socket_bind(&udp_socket, 12, 300);

/* If status is NX_SUCCESS, the UDP socket is now bound to port 12.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create,nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_bytes_available

Retrieves number of bytes available for retrieval

Prototype

```
UINT nx_udp_socket_bytes_available(NX_UDP_SOCKET *socket_ptr,
                                   ULONG *bytes_available);
```

Description

This service retrieves number of bytes available for retrieval in the specified UDP socket.

Parameters

socket_ptr	Pointer to previously created UDP socket.
bytes_available	Pointer to destination for bytes available.

Return Values

NX_SUCCESS	(0x00)	Successful bytes available retrieval.
NX_NOT_SUCCESSFUL	(0x43)	Socket not bound to a port.
NX_PTR_ERROR	(0x07)	Invalid pointers.
NX_NOT_ENABLED	(0x14)	UDP feature is not enabled.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Get the bytes available for retrieval from the UDP socket. */
status = nx_udp_socket_bytes_available(&my_socket,
                                        &bytes_available);

/* If status == NX_SUCCESS, the number of bytes was successfully
   retrieved.*/
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_checksum_disable

Disable checksum for UDP socket

Prototype

```
UINT nx_udp_socket_checksum_disable(NX_UDP_SOCKET *socket_ptr);
```

Description

This service disables the checksum logic for sending and receiving packets on the specified UDP socket. When the checksum logic is disabled, a value of zero is loaded into the UDP header's checksum field for all packets sent through this socket. A zero-value checksum signals the receiver that checksum is not computed for this socket. Note that this service has no effect on packets on the IPv6 network since UDP checksum is mandatory in IPv6.

Also note that this has no effect if NX_DISABLE_UDP_RX_CHECKSUM and NX_DISABLE_UDP_TX_CHECKSUM are defined when receiving and sending UDP packets respectively,

Parameters

socket_ptr	Pointer to previously created UDP socket instance.

Return Values

NX_SUCCESS	(0x00)	Successful socket checksum disable.
NX_NOT_BOUND	(0x24)	Socket is not bound.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, timer

Preemption Possible

No

Example

```
/* Disable the UDP checksum logic for packets sent on this socket.  */
status = nx_udp_socket_checksum_disable(&udp_socket);

/* If status is NX_SUCCESS, outgoing packets will not have a checksum
   calculated.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_checksum_enable

Enable checksum for UDP socket

Prototype

```
UINT nx_udp_socket_checksum_enable(NX_UDP_SOCKET *socket_ptr);
```

Description

This service enables the checksum logic for sending and receiving packets on the specified UDP socket. The checksum covers the entire UDP data area as well as a pseudo IP header. Note that this service has no effect on packets on the IPv6 network. UDP checksum is mandatory in IPv6.

Also note that this has no effect if NX_DISABLE_UDP_RX_CHECKSUM and NX_DISABLE_UDP_TX_CHECKSUM are defined when receiving and sending UDP packets respectively,

Parameters

socket_ptr	Pointer to previously created UDP socket instance.

Return Values

NX_SUCCESS	(0x00)	Successful socket checksum enable.
NX_NOT_BOUND	(0x24)	Socket is not bound.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, timer

Preemption Possible

No

Example

```
/* Enable the UDP checksum logic for packets sent on this socket.  */
status = nx_udp_socket_checksum_enable(&udp_socket);

/* If status is NX_SUCCESS, outgoing packets will have a checksum
   calculated.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_create

Create UDP socket

Prototype

```
UINT nx_udp_socket_create(NX_IP *ip_ptr,
                          NX_UDP_SOCKET *socket_ptr, CHAR *name,
                          ULONG type_of_service, ULONG fragment,
                          UINT time_to_live, ULONG queue_maximum);
```

Description

This service creates a UDP socket for the specified IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
socket_ptr	Pointer to new UDP socket control bloc.
name	Application name for this UDP socket.
type_of_service	Defines the type of service for the transmission, legal values are as follows: NX_IP_NORMAL (0x00000000) NX_IP_MIN_DELAY (0x00100000) NX_IP_MAX_DATA (0x00080000) NX_IP_MAX_RELIABLE (0x00040000) NX_IP_MIN_COST (0x00020000)
	fragmentSpecifies whether or not IP fragmenting is allowed. If NX_FRAGMENT_OKAY (0x0) is specified, IP fragmenting is allowed. If NX_DONT_FRAGMENT (0x4000) is specified, IP fragmenting is disabled.
time_to_live	Specifies the 8-bit value that defines how many routers this packet can pass before being thrown away. The default value is specified by NX_IP_TIME_TO_LIVE.
queue_maximum	Defines the maximum number of UDP datagrams that can be queued for this socket. After the queue limit is reached, for every new packet received the oldest UDP packet is released.

Return Values

NX_SUCCESS	(0x00)	Successful UDP socket create.
NX_OPTION_ERROR	(0x0A)	Invalid type-of-service, fragment, or time-to-live option.
NX_PTR_ERROR	(0x07)	Invalid IP or socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization and Threads

Preemption Possible

No

Example

```
/* Create a UDP socket with a maximum receive queue of 30 packets.  */
status = nx_udp_socket_create(&ip_0, &udp_socket, "Sample UDP Socket",
                              NX_IP_NORMAL, NX_FRAGMENT_OKAY, 0x80, 30);

/* If status is NX_SUCCESS, the new UDP socket has been created and is
   ready for binding.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_delete

Delete UDP socket

Prototype

```
UINT nx_udp_socket_delete(NX_UDP_SOCKET *socket_ptr);
```

Description

This service deletes a previously created UDP socket.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.

Return Values

NX_SUCCESS	(0x00)	Successful socket delete.
NX_NOT_CREATED	(0x27)	Socket was not created.
NX_STILL_BOUND	(0x42)	Socket is still bound.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Delete a previously created UDP socket.  */
status = nx_udp_socket_delete(&udp_socket);

/* If status is NX_SUCCESS, the previously created UDP socket has
   been deleted.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_info_get

Retrieve information about UDP socket activities

Prototype

```
UINT nx_udp_socket_info_get(NX_UDP_SOCKET *socket_ptr,
                            ULONG *udp_packets_sent,
                            ULONG *udp_bytes_sent,
                            ULONG *udp_packets_received,
                            ULONG *udp_bytes_received,
                            ULONG *udp_packets_queued,
                            ULONG *udp_receive_packets_dropped,
                            ULONG *udp_checksum_errors);
```

Description

This service retrieves information about UDP socket activities for the specified UDP socket instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.
udp_packets_sent	Pointer to destination for the total number of UDP packets sent on socket.
udp_bytes_sent	Pointer to destination for the total number of UDP bytes sent on socket.
udp_packets_received	Pointer to destination of the total number of UDP packets received on socket.
udp_bytes_received	Pointer to destination of the total number of UDP bytes received on socket.
udp_packets_queued	Pointer to destination of the total number of queued UDP packets on socket.
udp_receive_packets_dropped	Pointer to destination of the total number of UDP receive packets dropped for socket due to queue size being exceeded.
udp_checksum_errors	Pointer to destination of the total number of UDP packets with checksum errors on socket.

Return Values

NX_SUCCESS	(0x00)	Successful UDP socket information retrieval.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve UDP socket information from previously created socket 0.  */
status =  nx_udp_socket_info_get(&socket_0,
                                 &udp_packets_sent,
                                 &udp_bytes_sent,
                                 &udp_packets_received,
                                 &udp_bytes_received,
                                 &udp_queued_packets,
                                 &udp_receive_packets_dropped,
                                 &udp_checksum_errors);

/* If status is NX_SUCCESS, UDP socket information was retrieved.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_interface_send

Send datagram through UDP socket

Prototype

```
UINT nx_udp_socket_interface_send(NX_UDP_SOCKET *socket_ptr,
                                  NX_PACKET *packet_ptr,
                                  ULONG ip_address,
                                  UINT port,
                                  UINT interface_index);
```

Description

This service sends a UDP datagram through a previously created and bound UDP socket from the specified network interface. Note that service returns immediately, regardless of whether or not the UDP datagram was successfully sent.

Parameters

socket_ptr	Socket to transmit the packet out on.
packet_ptr	Pointer to packet to transmit.
ip_address	Destination IP address to send packet.
port	Destination port.
interface_index	Index of interface to send packet on.

Return Values

NX_SUCCESS	(0x00)	Packet successfully sent.
NX_NOT_BOUND	(0x24)	Socket not bound to a port.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_NOT_ENABLED	(0x14)	UDP processing not enabled.
NX_PTR_ERROR	(0x07)	Invalid pointer.
NX_OVERFLOW	(0x03)	Invalid packet append pointer.
NX_UNDERFLOW	(0x02)	Invalid packet prepend pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_INVALID_INTERFACE	(0x4C)	Invalid interface index.
NX_INVALID_PORT	(0x46)	Port number exceeds maximum port number.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Send packet out on port 80 to the specified destination IP on the
   interface at index 1 in the IP task interface list.  */
status = nx_udp_packet_interface_send(socket_ptr, packet_ptr,
                                      destination_ip, 80, 1);

/* If status is NX_SUCCESS packet was successfully transmitted. */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_bytes_available, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_port_get

Pick up port number bound to UDP socket

Prototype

```
UINT nx_udp_socket_port_get(NX_UDP_SOCKET *socket_ptr,
                            UINT *port_ptr);
```

Description

This service retrieves the port number associated with the socket, which is useful to find the port allocated by NetX Duo in situations where the NX_ANY_PORT was specified at the time the socket was bound.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.
port_ptr	Pointer to destination for the return port number. Valid port numbers are (1- 0xFFFF).

Return Values

NX_SUCCESS	(0x00)	Successful socket bind.
NX_NOT_BOUND	(0x24)	This socket is not bound to a port.
NX_PTR_ERROR	(0x07)	Invalid socket pointer or port return pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads, timers

Preemption Possible

No

Example

```
/* Get the port number of previously created and bound UDP socket. */
status = nx_udp_socket_port_get(&udp_socket, &port);

/* If status is NX_SUCCESS, the port variable contains the port this
   socket is bound to. */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_receive

Receive datagram from UDP socket

Prototype

```
UINT nx_udp_socket_receive(NX_UDP_SOCKET *socket_ptr,
                           NX_PACKET **packet_ptr,
                           ULONG wait_option);
```

Description

This service receives an UDP datagram from the specified socket. If no datagram is queued on the specified socket, the caller suspends based on the supplied wait option.

If NX_SUCCESS is returned, the application is responsible for releasing the received packet when it is no longer needed.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.
packet_ptr	Pointer to UDP datagram packet pointer.
wait_option	Defines how the service behaves if a datagram is not currently queued on this socket. The wait options are defined as follows:

NX_NO_WAIT	(0x00000000)
NX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket receive.
NX_NOT_BOUND	(0x24)	Socket was not bound to any port.
NX_NO_PACKET	(0x01)	There was no UDP datagram to receive.

NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_PTR_ERROR	(0x07)	Invalid socket or packet return pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Receive a packet from a previously created and bound UDP socket.
   If no packets are currently available, wait for 500 timer ticks
   before giving up. */
status = nx_udp_socket_receive(&udp_socket, &packet_ptr, 500);

/* If status is NX_SUCCESS, the received UDP packet is pointed to by
   packet_ptr. */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_receive_notify

Notify application of each received packet

Prototype

```
UINT nx_udp_socket_receive_notify(NX_UDP_SOCKET *socket_ptr,
                                   VOID (*udp_receive_notify)
                                         (NX_UDP_SOCKET *socket_ptr));
```

Description

This service sets the receive notify function pointer to the callback function specified by the application. This callback function is then called whenever a packet is received on the socket. If a NX_NULL pointer is supplied, the receive notify function is disabled.

Parameters

socket_ptr	Pointer to the UDP socket.
udp_receive_notify	Application callback function pointer that is called when a packet is received on the socket.

Return Values

NX_SUCCESS	(0x00)	Successful socket receive notify.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
/* Setup a receive packet callback function for the "udp_socket"
   socket. */
status = nx_udp_socket_receive_notify(&udp_socket,
                                       my_receive_notify);

/* If status is NX_SUCCESS, NetX Duo will call the function named
   "my_receive_notify" whenever a packet is received for
   "udp_socket". */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_socket_extract

nx_udp_socket_send

Send datagram through UDP socket

Prototype

```
UINT nx_udp_socket_send(NX_UDP_SOCKET *socket_ptr,
                        NX_PACKET *packet_ptr,
                        ULONG ip_address, UINT port);
```

Description

This service sends a UDP datagram through a previously created and bound UDP socket. Note that the service returns immediately, regardless of whether or not the UDP datagram was successfully sent.

Unless an error is returned, the application should not release the packet after this call. Doing so will cause unpredictable results because the network driver will release the packet after transmission.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.
packet_ptr	UDP datagram packet pointer.
ip_address	Destination IP address, which can be a specific host IP address, a network broadcast, an internal loopback, or a multicast address.
port	Destination port number, legal values range between 1 and 0xFFFF.

Return Values

NX_SUCCESS	(0x00)	Successful socket send.
NX_NOT_BOUND	(0x24)	Socket was not bound to any port.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_NO_INTERFACE_ADDRESS	(0x50)	No suitable outgoing interface found.

NX_UNDERFLOW	(0x02)	Not enough room to prepend the UPD header in the packet structure.
NX_OVERFLOW	(0x03)	Packet append pointer is invalid.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_INVALID_PORT	(0x46)	Invalid port specified.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Send a packet through a previously created and bound UDP socket
   to port 12 on IP 1.2.3.5.  */
status = nx_udp_socket_send(&udp_socket, packet_ptr,
                            IP_ADDRESS(1,2,3,5), 12);

/* If status is NX_SUCCESS, the UDP packet was sent.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_unbind

Unbind UDP socket from UDP port

Prototype

```
UINT nx_udp_socket_unbind(NX_UDP_SOCKET *socket_ptr);
```

Description

This service releases the binding between the UDP socket and a UDP port. If there are other threads waiting to bind another socket to the unbound port, the first suspended thread is then bound to the newly unbound port.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.

Return Values

NX_SUCCESS	(0x00)	Successful socket unbind.
NX_NOT_BOUND	(0x24)	Socket was not bound to any port.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Unbind the previously bound UDP socket.  */
status = nx_udp_socket_unbind(&udp_socket);

/* If status is NX_SUCCESS, the previously bound socket is now
   unbound.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_source_extract

nx_udp_source_extract

Extract IP and sending port from UDP datagram

Prototype

```
UINT nx_udp_source_extract(NX_PACKET *packet_ptr,
                           ULONG *ip_address, UINT *port);
```

Description

This service extracts the sender's IP and port number from the IP and UDP headers of the supplied UDP datagram.

Parameters

packet_ptr	UDP datagram packet pointer.
ip_address	Pointer to the return IP address variable.
port	Pointer to the return port variable.

Return Values

NX_SUCCESS	(0x00)	Successful source IP/port extraction.
NX_INVALID_PACKET	(0x12)	The supplied packet is invalid.
NX_PTR_ERROR	(0x07)	Invalid packet or IP or port destination.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Extract the IP and port information from the sender of the UPD
   packet.  */
status = nx_udp_source_extract(packet_ptr, &sender_ip_address,
                                           &sender_port);

/* If status is NX_SUCCESS, the sending IP and port information has been
   stored in sender_ip_address and sender_port respectively.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind

nxd_icmp_enable

Enable ICMPv4 and ICMPv6 Services

Prototype

```
UINT nxd_icmp_enable(NX_IP *ip_ptr);
```

Description

This function enables both ICMPv4 and ICMPv6 services and can only be called after the IP instance has been created. The service can be enabled either before or after IPv6 is enabled (see *nxd_ipv6_enable*). ICMPv4 services include Echo Request/Reply. ICMPv6 services include Echo Request/Reply, Neighbor Discovery, Duplicate Address Detection, Router Discovery, and Stateless Address Auto-configuration. The IPv4 equivalent in NetX is *nx_icmp_enable*.

nx_icmp_enable starts ICMP services for IPv4 operations only. Applications using ICMPv6 services must use *nxd_icmp_enable* instead of *nx_icmp_enable*.

To utilize IPv6 router solicitation and IPv6 stateless auto-address configuration, ICMPv6 must be enabled.

Parameters

ip_ptr	Pointer to previously created IP instance

Return Values

NX_SUCCESS	(0x00)	ICMP services are successfully enabled
NX_PTR_ERROR	(0x07)	Invalid IP pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Initialization, Threads

Preemption Possible

No

Example

```
/* Enable ICMP on the IP instance.  */
status = nxd_icmp_enable(&ip_0);

/* A status return of NX_SUCCESS indicates that the IP instance is
   enabled for ICMP services. */
```

See Also

nx_icmp_enable, nx_ip_create, nxd_ipv6_enable, nx_icmp_info_get, nxd_icmp_ping

nxd_icmp_interface_ping

Perform ICMPv4 and ICMPv6 Echo Requests

Prototype

```
UINT nxd_icmp_interface_ping(NX_IP *ip_ptr, NXD_ADDRESS *ip_address,
                             UINT if_index,
                             CHAR *data_ptr, ULONG data_size,
                             NX_PACKET *response_ptr,
                             ULONG wait_option);
```

Description

This function sends out an ICMP Echo Request packet through an appropriate interface and waits for an Echo Reply from the destination host. This service works with both IPv4 and IPv6 addresses. For an IPv4 destination address, the parameter *if_index* indicates the source IP address to use. For IPv6, the *if_index* indicates the entry in the IPv6 address table to use as source address.

The IP instance must have been created, and the ICMPv4 and ICMPv6 services must be enabled (see *nxd_icmp_enable*). On a multihome system, the *if_index* must be valid if the destination IP address is an IPv6 link local address.

If NX_SUCCESS is returned, the application is responsible for releasing the received packet after it is no longer needed.

Parameters

ip_ptr	Pointer to IP instance
ip_address	Destination IP address to ping, in host byte order
if_index	Indicates the IP address to use as source address
data_ptr	Pointer to ping packet data area
data_size	Number of bytes of ping data
response_ptr	Pointer to response packet pointer
wait_option	Time to wait for a reply

Return Values

NX_SUCCESS	(0x00)	Successful sent and received ping
NX_NOT_SUPPORTED	(0x4B)	IPv6 is not enabled
NX_OVERFLOW	(0x03)	Ping data exceeds packet payload
NX_NO_RESPONSE	(0x29)	Destination host did not respond
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by *tx_thread_wait_abort*
NX_NO_INTERFACE_ADDRESS	(0x50)	No suitable outgoing interface can be found
NX_PTR_ERROR	(0x07)	Invalid IP or response pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_NOT_ENABLED	(0x14)	IP or ICMP component is not enabled
NX_IP_ADDRESS_ERROR	(0x21)	Input IP address is invalid

Allowed From

Threads

Preemption Possible

No

Example

```
/* The following two examples illustrate how to use this API to send ping
   packets to IPv6 or IPv4 destinations.  */

/* The first example:  Send a ping packet to an IPv6 host
   FE80::411:7B23:40dc:f181 */

/* Declare variable address to hold the destination address. */

#define PRIMARY_INTERFACE 0

NXD_ADDRESS  ip_address;
char *buffer = "abcd";
UINT  prefix_length = 10;

/* Set the IPv6 address. */
ip_address.nxd_ip_address_version   = NX_IP_VERSION_V6;
ip_address.nxd_ip_address.v6[0]     = 0xFE800000;
ip_address.nxd_ip_address.v6[1]     = 0x00000000;
ip_address.nxd_ip_address.v6[2]     = 0x04117B23;
ip_address.nxd_ip_address.v6[3]     = 0x40DCF181;

status = nxd_icmp_interface_ping(&ip_0, &ip_address,
                                 PRIMARY_INTERFACE,
                                 buffer,
                                 strlen(buffer),
                                 &response_ptr,
                                 prefix_length);

/* A return value of NX_SUCCESS indicates a ping reply has been received
   from IP address FE80::411:7B23:40dc:f181 and the response packet is
   contained in the packet pointed to by response_ptr.  It should have the
   same "abcd" four bytes of data. */

/* The second example:  Send a ping packet to an IPv4 host 1.2.3.4 */

/* Program the IPv4 address. */
ip_address.nxd_ip_address_version   = NX_IP_VERSION_V4;
ip_address.nxd_ip_address.v4        = 0x01020304;

status = nxd_icmp_interface_ping(&ip_0, &ip_address,
                                 PRIMARY_INTERFACE,
                                 buffer,
                                 strlen(buffer),
                                 &response_ptr,
                                 prefix_length);

/* A return value of NX_SUCCESS indicates a ping reply was received from
   IP address 1.2.3.4 and the response packet is contained in the packet
   pointed to by response_ptr. It should have the same "abcd" four bytes
   of data. */
```

See also

nx_icmp_enable, nx_ip_create, nx_icmp_info_get, nx_icmp_ping, nxd_icmp_enable

nxd_icmp_ping

Perform ICMPv4 and ICMPv6 Echo Requests

Prototype

```
UINT nxd_icmp_ping(NX_IP *ip_ptr, NXD_ADDRESS *ip_address,
                   CHAR *data_ptr, ULONG data_size,
                   NX_PACKET **response_ptr, ULONG wait_option)
```

Description

This function sends out an ICMP Echo Request packet through an appropriate physical interface and waits for an Echo Reply from the destination host. For the IPv4 network, the primary interface is used for sending the ICMP echo request. For the IPv6 network, NetX Duo determines the appropriate interface based on the destination address for determining the source address and sending the echo request. Applications use the service ***nxd_icmp_interface_ping*** to specify the physical interface and precise source IP address to use for packet transmission.

The IP instance must have been created, and the ICMPv4/ICMPv6 services must be enabled (see ***nxd_icmp_enable***).

If NX_SUCCESS is returned, the application is responsible for releasing the received packet after it is no longer needed.

Parameters

ip_ptr	Pointer to IP instance
ip_address	Destination IP address to ping, in host byte order
data_ptr	Pointer to ping packet data area
data_size	Number of bytes of ping data
response_ptr	Pointer to response packet pointer
wait_option	Time to wait for a reply

Return Values

NX_SUCCESS	(0x00)	Successful sent and received ping
NX_NOT_SUPPORTED	(0x4B)	IPv6 is not enabled
NX_OVERFLOW	(0x03)	Ping data exceeds packet payload
NX_NO_RESPONSE	(0x29)	Destination host did not respond
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by tx_thread_wait_abort
NX_NO_INTERFACE_ADDRESS	(0x50)	No suitable outgoing interface can be found.
NX_PTR_ERROR	(0x07)	Invalid IP or response pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_NOT_ENABLED	(0x14)	IP or ICMP component is not enabled
NX_IP_ADDRESS_ERROR	(0x21)	Input IP address is invalid

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* The following two examples illustrate how to use this API to send ping
   packets to IPv6 or IPv4 destinations.  */

/* The first example:  Send a ping packet to an IPv6 host
   2001:1234:5678::1 */

/* Declare variable address to hold the destination address. */
NXD_ADDRESS  ip_address;

char *buffer = "abcd";
UINT  prefix_length = 10;

/* Set the IPv6 address. */
ip_address.nxd_ip_address_version    = NX_IP_VERSION_V6;
ip_address.nxd_ip_address.v6[0]      = 0x20011234;
```

```
ip_address.nxd_ip_address.v6[1]     = 0x56780000;
ip_address.nxd_ip_address.v6[2]     = 0;
ip_address.nxd_ip_address.v6[3]     = 1;

status = nxd_icmp_ping(&ip_0, &ip_address, buffer,
                       strlen(buffer),&response_ptr, prefix_length);

/* A return value of NX_SUCCESS indicates a ping reply has been received
   from IP address 2001:1234:5678::1 and the response packet is
   contained in the packet pointed to by response_ptr.  It should have
   the same "abcd" four bytes of data. */

/* The second example:  Send a ping packet to an IPv4 host 1.2.3.4 */

/* Program the IPv4 address. */
ip_address.nxd_ip_address_version   = NX_IP_VERSION_V4;
ip_address.nxd_ip_address.v4[0]     = 0x01020304;

status = nxd_icmp_ping(&ip_0, &ip_address, buffer,
                        strlen(buffer),&response_ptr, 10);

/* A return value of NX_SUCCESS indicates a ping reply was received from
   IP address 1.2.3.4 and the response packet is contained in the packet
   pointed to by response_ptr. It should have the same "abcd" four bytes
   of data. */
```

See also

nx_icmp_enable, nx_ip_create, nx_icmp_info_get, nx_icmp_ping, nxd_icmp_enable

nxd_ip_raw_packet_send

Send Raw IP Packet

Prototype

```
UINT nxd_ip_raw_packet_send(NX_IP *ip_ptr, NX_PACKET *packet_ptr,
                            NXD_ADDRESS *destination_ip,
                            ULONG protocol)
```

Description

This function sends a raw IP packet (no transport-layer protocol headers) through a suitable interface based on destination IP address. On a multihome system, if the system is unable to determine an appropriate interface (for example, if the destination IP address is IPv4 broadcast or IPv6 link local address), the primary interface is selected. The service *nxd_ip_raw_packet_interface_send* can be used to specify an outgoing interface. The NetX equivalent is *nx_ip_raw_packet_send*.

The IP instance must be previously created and raw IP packet handling must be enabled using the *nx_ip_raw_packet_enable* service.

Parameters

ip_ptr	Pointer to the previously created IP instance
packet_ptr	Pointer to packet to transmit
destination_ip	Pointer to destination address
protocol	Packet protocol stored to the IP header

Return Value

NX_SUCCESS	(0x00)	Raw IP packet successfully sent
NX_NOT_ENABLED	(0x14)	Raw IP handling not enabled
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IPv4 or IPv6 address
NX_PTR_ERROR	(0x07)	Invalid IP pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_INVALID_INTERFACE	(0x4C)	Invalid interface index

Allowed From

Threads

Preemption Possible

Yes

Example

```
NXD_ADDRESS  dest_address;

/* Set the destination address,in this case an IPv6 address. */
     dest_address.nxd_ip_address_version   = NX_IP_VERSION_V6;
     dest_address.nxd_ip_address.v6[0]     = 0x20011234;
     dest_address.nxd_ip_address.v6[1]     = 0x56780000;
     dest_address.nxd_ip_address.v6[2]     = 0;
     dest_address.nxd_ip_address.v6[3]     = 1;

/* Enable RAW IP handling on the previously created IP instance.  */
status = nx_raw_ip_packet_enable(&ip_0);

/* Allocate a packet pointed to by packet_ptr from the IP packet pool. */
/* Then transmit the packet to the destination address. */

status = nxd_ip_raw_packet_send(&ip_0, packet_ptr,  dest_address,
                                NX_PROTOCOL_UDP);

/* A status return of NX_SUCCESS indicates the packet was successfully
   transmitted. */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_receive, nx_ip_raw_packet_send

nxd_ip_raw_packet_interface_send

Send Raw IP Packet

Prototype

```
UINT nxd_ip_raw_packet_interface_send(NX_IP *ip_ptr, NX_PACKET
                                      *packet_ptr, NXD_ADDRESS
                                      *destination_ip,
                                      UINT if_index,
                                      ULONG protocol);
```

Description

This function sends a raw IP packet (no transport-layer protocol headers prepended) through the specified interface index to the destination IP address.

The IP instance must be previously created and raw IP packet handling must be enabled using the *nx_ip_raw_packet_enable* service.

Parameters

ip_ptr	Pointer to the previously created IP instance
packet_ptr	Pointer to packet to transmit
destination_ip	Pointer to destination address
if_index	Index specifying the outgoing physical network
protocol	Packet protocol stored to the IP header

Return Value

NX_SUCCESS	(0x00)	Raw IP packet successfully sent
NX_NOT_ENABLED	(0x14)	Raw IP handling not enabled
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IPv4 or IPv6 address
NX_PTR_ERROR	(0x07)	Invalid IP pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_INVALID_INTERFACE	(0x4C)	Invalid interface_index

Allowed From

Threads

Preemption Possible

Yes

Example

```
#define PRIMARY_INTERFACE 0

NXD_ADDRESS  dest_address;

/* Set the destination address, in this case the destination IP address is
   FE80::411:7B23:40dc:f181. */
dest_address.nxd_ip_address_version   = NX_IP_VERSION_V6;
dest_address.nxd_ip_address.v6[0]     = 0xFE800000;
dest_address.nxd_ip_address.v6[1]     = 0x00000000;
dest_address.nxd_ip_address.v6[2]     = 0x04117B23;
dest_address.nxd_ip_address.v6[3]     = 0x40DCF181;

/* Enable RAW IP handling on the previously created IP instance.  */
status = nx_ip_raw_packet_enable(&ip_0);

/* Allocate a packet pointed to by packet_ptr from the IP packet pool. */
/* Then transmit the packet to the destination address. */

status = nxd_ip_raw_packet_interface_send(&ip_0, packet_ptr,
                                          PRIMARY_INTERFACE,
                                          dest_address,
                                          NX_PROTOCOL_UDP);

/* A status return of NX_SUCCESS indicates the packet was successfully
   transmitted. */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_receive, nx_ip_raw_packet_send

nxd_ipv6_address_delete

Delete IPv6 Address

Prototype

```
UINT nxd_ipv6_address_delete(NX_IP *ip_ptr, UINT address_index);
```

Description

This function deletes the IPv6 address at the specified index in the address table of the specified IP instance. There is no NetX equivalent.

Parameters

ip_ptr	Pointer to the previously created IP instance
address_index	Index to IP instance address table

Return Values

NX_SUCCESS	(0x00)	Address successfully deleted
NX_NO_INTERFACE_ADDRESS	(0x50)	No suitable outgoing interface can be found
NX_PTR_ERROR	(0x07)	Invalid IP pointer

Allowed From

Initialization, Threads

Preemption Possible

Yes

Example

```
NXD_ADDRESS ip_address;
UINT        address_index;

/* Delete the IPv6 address at the specified address table index.  */

address_index = 1;
status = nxd_ipv6_address_delete(&ip_0, address_index);

/* A status return of NX_SUCCESS indicates that the IP instance address
   is successfully deleted. */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_create,nx_ip_delete, nx_ip_interface_address_get, nx_ip_info_get, nx_ip_interface_address_set, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check, nxd_ipv6_address_get, nxd_ipv6_address_set

nxd_ipv6_address_get

Retrieve IPv6 Address and Prefix

Prototype

```
UINT nxd_ipv6_address_get(NX_IP *ip_ptr, UINT address_index,
                          NXD_ADDRESS *ip_address,
                          ULONG prefix_length, UINT *if_index);
```

Description

This function retrieves the IPv6 address and prefix at the specified index in the address table of the specified IP instance. The address network interface is returned in the *if_index* pointer. The NetX equivalent ***nx_ip_address_get*** and for multihome hosts ***nx_ip_interface_address_get***.

Parameters

ip_ptr	Pointer to the previously created IP instance
address_index	Index to IP instance address table
ip_address	Pointer to the address to set
prefix_length	Length of the address prefix (subnet mask)
if_index	Pointer to the address interface index

Return Values

NX_SUCCESS	(0x00)	IPv6 is successfully enabled
NX_NO_INTERFACE_ADDRESS	(0x50)	No suitable outgoing interface can be found
NX_PTR_ERROR	(0x07)	Invalid IP pointer

Allowed From

Initialization, Threads

Preemption Possible

Yes

Example

```
NXD_ADDRESS ip_address;
UINT        address_index;
UINT        prefix_length;
UINT        interface_id;

ip_address.nxd_ip_version = NX_IP_VERSION_V6;
ip_address.nxd_ip_address.v6[0] = 0x20010000;
ip_address.nxd_ip_address.v6[1] = 0;
ip_address.nxd_ip_address.v6[2] = 0;
ip_address.nxd_ip_address.v6[3] = 1;


/* Get the IPv6 address at the specified address table index. If found,
   the address network interface is returned in the interface_id input,
   as well as the address prefix in the prefix_length input.  */

address_index = 1;
status = nxd_ipv6_address_get(&ip_0, address_index, &ip_address,
                              &prefix_length, &interface_id);

/* A status return of NX_SUCCESS indicates that the IP instance address
   is successfully retrieved. */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_create,nx_ip_delete, nx_ip_interface_address_get, nx_ip_info_get, nx_ip_interface_address_set, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check, nxd_ipv6_address_deleted, nxd_ipv6_address_set

nxd_ipv6_address_set

Set IPv6 Address and Prefix

Prototype

```
UINT nxd_ipv6_address_set(NX_IP *ip_ptr, UINT if_index, NXD_ADDRESS
                          *ip_address,
                          ULONG prefix_length,
                          UINT *address_index);
```

Description

This function sets the supplied IPv6 address and prefix to the specified IP instance. If the *address_index* argument is not null, the index into the IP address table where the address is inserted is returned. The NetX equivalent ***nx_ip_address_set*** and for multihome hosts ***nx_ip_interface_address_set***.

Parameters

ip_ptr	Pointer to the previously created IP instance
if_index	Index to interface to set the address
ip_address	Pointer to the address to set
prefix_length	Length of the address prefix (subnet mask)
address_index	Pointer to the index into the address table where the address is inserted

Return Values

NX_SUCCESS	(0x00)	IPv6 is successfully enabled
NX_NO_MORE_ENTRIES	(0x15)	IP address table is full
NX_PTR_ERROR	(0x07)	Invalid IP pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IPv6 address
NX_INVALID_INTERFACE	(0x4C)	Interface points to an invalid network interface

Allowed From

Initialization, Threads

Preemption Possible

Yes

Example

```
NXD_ADDRESS ip_address;
UINT        address_index;
UINT        interface_id;

ip_address.nxd_ip_version = NX_IP_VERSION_V6;
ip_address.nxd_ip_address.v6[0] = 0x20010000;
ip_address.nxd_ip_address.v6[1] = 0;
ip_address.nxd_ip_address.v6[2] = 0;
ip_address.nxd_ip_address.v6[3] = 1;

/* First create an IP instance with packet pool, source address, and
   driver.*/
status = nx_ip_create(&ip_0, "NetX IP Instance 0",
                      IP_ADDRESS(1,2,3,4),
                      0xFFFFFF00UL, &pool_0,_nx_ram_network_driver,
                      pointer, 2048, 1);

/* Then enable IPv6 on the IP instance. */
status = nxd_ipv6_enable(&ip_0);

/* Set the IPv6 address (a global address as indicated by the 64 bit
   prefix) using the IPv6 address just created on the primary interface
   (index zero). The index into the address table is returned in
   address_index. */
interface_id = 0;
status = nxd_ipv6_address_set(&ip_0, interface_id, &ip_address, 64,
                              &address_index);

/* A status return of NX_SUCCESS indicates that the IP instance address
   is successfully registered with the IP instance. */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_create,nx_ip_delete, nx_ip_interface_address_get, nx_ip_info_get, nx_ip_interface_address_set, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check, nxd_ipv6_address_get, nxd_ipv6_address_delete

nxd_ipv6_enable

Enable IPv6 Services

Prototype

```
UINT nxd_ipv6_enable(NX_IP *ip_ptr);
```

Description

This function enables IPv6 services. When the IPv6 services are enabled, the IP instance joins the all-node multicast group (FF02::1). This function does not set the link local address or global address. Applications should use *nxd_ipv6_address_set* to configure the device network addresses. There is no NetX equivalent.

Parameters

ip_ptr	Pointer to the previously created IP instance

Return Values

NX_SUCCESS	(0x00)	IPv6 is successfully enabled
NX_ALREADY_ENABLED	(0x15)	IPv6 is already enabled
NX_NOT_SUPPORTED	(0x4B)	IPv6 not enabled
NX_PTR_ERROR	(0x07)	Invalid IP pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Initialization, Threads

Preemption Possible

Yes

Example

```
/* First create an IP instance with packet pool, source address, and
   driver.*/
status = nx_ip_create(&ip_0, "NetX IP Instance 0",
                      IP_ADDRESS(1,2,3,4),
                      0xFFFFFF00UL, &pool_0,_nx_ram_network_driver,
                      pointer, 2048, 1);

/* Then enable IPv6 on the IP instance. */
status = nxd_ipv6_enable(&ip_0);

/* A status return of NX_SUCCESS indicates that the IP instance is
   enabled for IPv6 services. */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_create,nx_ip_delete, nx_ip_interface_address_get, nx_ip_info_get, nx_ip_interface_address_set, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check, nxd_ipv6_address_get, nxd_ipv6_address_set

nxd_nd_cache_entry_set

Set IPv6 Address to MAC Mapping

Prototype

```
UINT nxd_nd_cache_entry_set(NX_IP *ip_ptr, NXD_ADDRESS *dest_ip,
                            UINT if_index, char *mac);
```

Description

This function adds an entry to the neighbor discovery cache for the specified IP instance for the IP address *ip_address* mapped to the hardware address mac on the specified interface index *if_index*. The equivalent NetX IPv4 service is *nx_arp_static_entry_create*.

Parameters

ip_ptr	Pointer to previously created IP instance
dest_ip	Pointer to IPv6 address instance
if_index	Index specifying physical interface where the destination IPv6 address can be reached
mac	Pointer to hardware address, host byte order

Return Values

NX_SUCCESS	(0x00)	Entry successfully added
NX_NOT_SUCCESSFUL	(0x43)	Invalid cache or no neighbor cache entries available
NX_PTR_ERROR	(0x07)	Invalid IP instance or storage space

Allowed From

Initialization, Threads

Preemption Possible

No

Example

```
/* This example adds an entry on the primary network interface to the
   neighbor cache table. */

#define PRIMARY_INTERFACE 0

NXD_ADDRESS ip_address;
UCHAR      hw_address[6] = {0x0, 0xcf,0x01,0x02, 0x03, 0x04};
char       *mac;

mac = (char *)&hw_address[0];

ip_address.nxd_ip_address_version = NX_IP_VERSION_V6;
ip_address.nxd_ip_address.v6[0]   = 0x20011234;
ip_address.nxd_ip_address.v6[1]   = 0x56780000;
ip_address.nxd_ip_address.v6[2]   = 0;
ip_address.nxd_ip_address.v6[3]   = 1;


/* Create an entry in the neighbor cache table with the specified IPv6
   address and hardware address. */
status = nxd_nd_cache_entry_set(&ip_0,
                                &ip_address.nxd_ip_address.v6[0],
                                PRIMARY_INTERFACE, mac);

/* If status == NX_SUCCESS, the entry was added to the neighbor cache
   table. */
```

See Also

nxd_nd_cache_entry_delete, nxd_nd_cache_invalidate, nxd_nd_cache_hardware_address_find, nxd_nd_cache_ip_address_find

nxd_nd_cache_entry_delete

Delete IPv6 Address entry in the Neighbor Cache

Prototype

```
UINT nxd_nd_cache_entry_delete(NX_IP ip_ptr, ULONG *ip_address)
```

Description

This function deletes an IPv6 neighbor discovery cache entry for the supplied IP address. The equivalient NetX IPv4 function is *nx_arp_static_entry_delete*.

Parameters

ip_ptr	Pointer to previously created IP instance
ip_address	Pointer to IPv6 address to delete, in host byte order

Return Values

NX_SUCCESS	(0x00)	Successfully deleted the address
NX_ENTRY_NOT_FOUND	(0x16)	Address not found in the IPv6 neighbor cache
NX_PTR_ERROR	(0x07)	Invalid IP instance or storage space

Allowed From

Threads

Preemption Possible

No

Example

```
/* This example deletes an entry from the neighbor cache table. */

NXD_ADDRESS ip_address;

ip_address.nxd_ip_address_version = NX_IP_VERSION_V6;
ip_address.nxd_ip_address.v6[0]   = 0x20011234;
ip_address.nxd_ip_address.v6[1]   = 0x56780000;
ip_address.nxd_ip_address.v6[2]   = 0;
ip_address.nxd_ip_address.v6[3]   = 1;


/* Delete an entry in the neighbor cache table with the specified IPv6
   address and hardware address. */
status = nxd_nd_cache_entry_delete(&ip_0,
                                   &ip_address.nxd_ip_address.v6[0]);

/* If status == NX_SUCCESS, the entry was deleted from the neighbor
   cache table. */
```

See Also

nxd_nd_cache_entry_set, nxd_nd_cache_invalidate, nxd_nd_cache_hardware_address_find, nxd_nd_cache_ip_address_find

nxd_nd_cache_hardware_address_find

Locate Hardware Address for an IPv6 Address

Prototype

```
UINT nxd_nd_cache_hardware_address_find(NX_IP *ip_ptr,
                                        NXD_ADDRESS *ip_address,
                                        ULONG *physical_msw,
                                        ULONG *physical_lsw
                                        UINT *if_index);
```

Description

This function attempts to find a physical hardware address in the IPv6 neighbor discovery cache that is associated with the supplied IPv6 address on the specified interface index. The equivalent NetX IPv4 service is *nx_arp_hardware_address_find*.

Parameters

ip_ptr	Pointer to previously created IP instance
ip_address	Pointer to IP address to find, host byte order
physical_msw	Pointer to the most significant word of the physical address, in host byte order
physical_lsw	Pointer to the least significant word of the physical address in host byte order
if_index	Pointer to the valid memory location for the interface index specifying the physical interface on which the IPv6 address can be reached.

Return Values

NX_SUCCESS	(0x00)	Successfully found the address
NX_ENTRY_NOT_FOUND	(0x16)	Mapping not in the neighbor cache
NX_INVALID_PARAMETERS	(0x4D)	Invalid non pointer input
NX_PTR_ERROR	(0x07)	Invalid IP instance or storage space

Allowed From

Threads

Preemption Possible

No

Example

```
/* This example inputs an IP address on the primary network in order to
   obtain the hardware address it is mapped to in the neighbor cache
   table. */

#define PRIMARY_INTERFACE 0

NXD_ADDRESS  ip_address;
ULONG physical_msw, physical_lsw;
UINT  if_index = PRIMARY_INTERFACE;

ip_address.nxd_ip_address_version = NX_IP_VERSION_V6;
ip_address.nxd_ip_address.v6[0]   = 0x20011234;
ip_address.nxd_ip_address.v6[1]   = 0x56780000;
ip_address.nxd_ip_address.v6[2]   = 0;
ip_address.nxd_ip_address.v6[3]   = 1;

/* Obtain the hardware address mapped to the supplied global IPv6
   address. */
status = nxd_nd_cache_hardware_address_find(&ip_0, &ip_address,
                                            &physical_msw,
                                            &physical_lsw
                                            if_index);

/* If status == NX_SUCCESS, a matching entry was found in the neighbor
   cache table and the hardware address returned in variables
   physical_msw and physical_lsw. */
```

See Also

nxd_nd_cache_entry_delete, nxd_nd_cache_entry_set, nxd_nd_cache_invalidate, nxd_nd_cache_ip_address_find

nxd_nd_cache_invalidate

Invalidate the Neighbor Discovery Cache

Prototype

```
UINT nxd_nd_cache_invalidate(NX_IP *ip_ptr);
```

Description

This function invalidates the entire IPv6 neighbor discovery cache. This function can be invoked either before or after ICMPv6 has been enabled. This service is not applicable to IPv4 connectivity, so there is no NetX equivalent service.

Parameters

ip_ptr	Pointer to IP instance

Return Values

NX_SUCCESS	(0x00)	Cache successfully invalidated
NX_PTR_ERROR	(0x07)	Invalid IP instance or storage space

Allowed From

Threads

Preemption Possible

No

Example

```
/* This example invalidates the host neighbor cache table. */

/* Invalidate the cache table bound to the IP instance. */
status = nxd_nd_cache_invalidate (&ip_0);

/* If status == NX_SUCCESS, all entries in the neighbor cache table
   are invalidated. */
```

See also

nxd_nd_cache_entry_delete, nxd_nd_cache_entry_set,
nxd_nd_cache_ip_address_find, nxd_nd_cache_hardware_address_find

nxd_nd_cache_ip_address_find

Retrieve IPv6 Address for a Physical Address

Prototype

```
UINT nxd_nd_cache_ip_address_find(NX_IP *ip_ptr,
                                  NXD_ADDRESS *ip_address,
                                  ULONG physical_msw,
                                  ULONG physical_lsw,
                                  UINT  *if_index);
```

Description

This function attempts to find an IPv6 address on the specified physical interface (*if_index*) in the IPv6 neighbor discovery cache that is associated with the supplied physical address. The equivalent NetX IPv4 service is *nx_arp_ip_address_find*.

Parameters

ip_ptr	Pointer to previously created IP instance
ip_address	Pointer to valid NXD_ADDRESS structure
physical_msw	Most significant word of the physical address to find, host byte order
physical_lsw	Least significant word of the physical address to find, host byte order
if_index	Pointer to the physical interface index through which the IPv6 address can be reached

Return Values

NX_SUCCESS	(0x00)	Successfully found the address
NX_ENTRY_NOT_FOUND	(0x16)	Physical address not found in the neighbor cache
NX_INVALID_PARAMETERS	(0x4D)	Invalid non pointer input
NX_PTR_ERROR	(0x07)	Invalid IP instance or storage space

Allowed From

Threads

Preemption Possible

No

Example

```
/* This example inputs a hardware address to search on for the matching
   IPv6 global address in the neighbor cache table. */

#define PRIMARY_INTERFACE 0

NXD_ADDRESS ip_address;
ULONG   physical_msw = 0xcf;
ULONG   physical_lsw = 0x01020304;
UINT        if_index = PRIMARY_INTERFACE;


/* Obtain the IPv6 address mapped to the supplied hardware
   Address on the primary interface. */
status = _nxd_nd_cache_ip_address_find(&ip_0, &ip_address,
                                       physical_msw, physical_lsw,
                                       &if_index);

/* If status == NX_SUCCESS, a matching entry was found in the neighbor
   cache table and the global IPv6 address returned in variable
   ip_address. */
```

See Also

nxd_nd_cache_entry_delete, nxd_nd_cache_entry_set, nxd_nd_cache_invalidate, nxd_nd_cache_hardware_address_find

nxd_ipv6_default_router_add

Add an IPv6 Router to Default Router Table

Prototype

```
UINT nxd_ipv6_default_router_add(NX_IP *ip_ptr,
                                 NXD_ADDRESS *router_address,
                                 ULONG router_lifetime,
                                 UINT if_index);
```

Description

This function adds an IPv6 default router on the specified physical interface to the default router table. The equivalent NetX IPv4 service is *nx_ip_gateway_address_set*.

router_address must point to a valid IP address, and the router must be directly accessible from the specified physical interface.

Parameters

ip_ptr	Pointer to previously created IP instance
router_address	Pointer to the default router address, in host byte order
router_lifetime	Default router life time, in seconds. Valid values are: 0xFFFF: No time out 0-0xFFFE: Timeout value, in seconds
if_index	Pointer to the valid memory location for the interface index through which the router can be reached

Return Values

NX_SUCCESS	(0x00)	Default router is successfully added
NX_INVALID_PARAMETERS	(0x4D)	Not valid IPv6 address input
NX_PTR_ERROR	(0x07)	Invalid IP instance or storage space

Allowed From

Initialization, Threads

Preemption Possible

No

Example

```
/* This example adds a default router for the primary interrface at
   fe80::1219:B9FF:FE37:ac to the default router table. */

#define PRIMARY_INTERFACE 0

NXD_ADDRESS  router_address;

/* Set the router address version to IPv6 */
router_address.nxd_ip_version    = NX_IP_VERSION_V6;

/* Set the IPv6 address, in host byte order. */
router_address.nxd_ip_address[0] = 0xfe800000;
router_address.nxd_ip_address[1] = 0x0;
router_address.nxd_ip_address[2] = 0x1219B9FF;
router_address.nxd_ip_address[3] = 0xFE3700AC;

/* Set IPv6 default router. */
status = nxd_ipv6_default_router_add(ip_ptr, &router_address, 0xFFFF,
                                     PRIMARY_INTERFACE );

/* Unless invalid pointer input is detected by the error checking
   Service for adding routers, status return is always NX_SUCCESS. */
```

See also

nxd_ipv6_default_router_delete, nxd_ipv6_default_router_get

nxd_ipv6_default_router_delete

Remove IPv6 Router from Default Router Table

Prototype

```
UINT nxd_ipv6_default_router_delete (NX_IP *ip_ptr,
                                     NXD_ADDRESS *router_address);
```

Description

This function deletes an IPv6 default router from the default router table. The equivalent NetX IPv4 service is *nx_ip_gateway_address_set* with a null gateway address specified.

Restrictions

The IP instance has been created. *router_address* must point to valid information.

Parameters

ip_ptr	Pointer to a previously created IP instance
router_address	Pointer to the IPv6 default gateway address

Return Values

NX_SUCCESS	(0x00)	Router successfully deleted
NX_PTR_ERROR	(0x07)	Invalid IP instance or storage space
NX_INVALID_PARAMETERS	(0x82)	Invalid non pointer input

Allowed From

Initialization, Threads

Preemption Possible

No

Example

```
/*This example removes a default router:fe80::1219:B9FF:FE37:ac */

NXD_ADDRESS  router_address;

/* Set the router_address version to IPv6 */
router_address.nxd_ip_version    = NX_IP_VERSION_V6;

/* Program the IPv6 address, in host byte order. */
router_address.nxd_ip_address[0] = 0xfe800000;
router_address.nxd_ip_address[1] = 0x0;
router_address.nxd_ip_address[2] = 0x1219B9FF;
router_address.nxd_ip_address[3] = 0xFE3700AC;

/* Delete the IPv6 default router. */
nxd_ipv6_default_router_delete(ip_ptr, &router_address);

/* Unless invalid pointer input is detected by the error checking
   Service for deleting routers, status return is always NX_SUCCESS.
   */
```

See also

nxd_ipv6_default_router_add, nxd_ipv6_default_router_get

nxd_ipv6_default_router_get

Retrieve an IPv6 Router from Default Router Table

Prototype

```
UINT nxd_ipv6_default_router_get(NX_IP *ip_ptr, UINT if_index
                                 NXD_ADDRESS *router_address,
                                 ULONG *router_lifetime,
                                 UINT *prefix_length);
```

Description

This function retrieves an IPv6 default router address, lifetime and prefix length on the specified physical interface matching the input router address from the default router table. The equivalent NetX IPv4 service is *nx_ip_gateway_address_get*.

router_address must point to a valid IP address, and the router must be directly accessible from the specified physical interface.

Parameters

ip_ptr	Pointer to previously created IP instance
if_index	Router network interface index
router_address	Pointer to the default router address, in host byte order
router_lifetime	Pointer to the router life time
prefix_length	Pointer to the router address prefix length

Return Values

NX_SUCCESS	(0x00)	Default router is successfully added
NX_NOT_FOUND	(0x4E)	Default router not found
NX_INVALID_INTERFACE	(0x4C)	Invalid router interface index
NX_PTR_ERROR	(0x07)	Invalid IP instance or storage space

Allowed From

Initialization, Threads

Preemption Possible

No

Example

```
/* This example retrieves a default router for the primary interface
   from the default router table. */

#define PRIMARY_INTERFACE 0


NXD_ADDRESS  router_address;
ULONG        router_lifetime;
ULONG        prefix_length;

/* Get IPv6 default router. */
status = nxd_ipv6_default_router_get(ip_ptr, PRIMARY_INTERFACE,
   &router_address, &router_lifetime, &prefix_length);

/* If status returns NX_SUCCESS, the router address and related
   information is returned successfully. */
```

See also

nxd_ipv6_default_router_delete, nxd_ipv6_default_router_add

nxd_tcp_socket_peer_info_get

Retrieves Peer TCP Socket IP Address and Port Number

Prototype

```
UINT nxd_tcp_socket_peer_info_get(NX_TCP_SOCKET *socket_ptr,
                                  NXD_ADDRESS *peer_ip_address,
                                  ULONG *peer_port);
```

Description

This function retrieves IP address and port information for TCP sockets with for the connected TCP peer socket over IPv4 or IPv6 connectivity. The equivalent NetX IPv4 service is *nx_tcp_socket_peer_info_get*.

Note that *socket_ptr* must point to a TCP socket that is already in the connected state.

Parameters

socket_ptr	Pointer to TCP socket connected to peer host
peer_ip_address	Pointer to IPv4 or IPv6 peer address, host byte order
peer_port	Pointer to peer port number, host byte order

Return Values

NX_SUCCESS	(0x00)	Socket information successfully retrieved
NX_NOT_CONNECTED	(0x38)	Socket not connected to peer
NX_NOT_ENABLED	(0x14)	TCP not enabled
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Preemption Possible

Yes

Example

```
NXD_ADDRESS  peer_ip_address;
ULONG        peer_port;


/* Get TCP socket information. */
status = nxd_tcp_socket_peer_info_get(socket_ptr, &peer_ip_address,
                                       &peer_port);

/* If status == NX_SUCCESS, the service returns valid peer info: */
if(peer_ip_address.nxd_ip_version == NX_IP_VERSION_V4)
    /* Peer IP address is stored in
       peer_ip_address.nxd_ip_address.v4 */

if(peer_ip_address.nxd_ip_version == NX_IP_VERSION_V6)
    /* Peer IP address is stored in
       peer_ip_address.nxd_ip_address.v6 */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get,nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nxd_tcp_client_socket_connect

nxd_tcp_client_socket_connect

Make a TCP Connection

Prototype

```
UINT nxd_tcp_client_socket_connect(NX_TCP_SOCKET *socket_ptr
                                   NXD_ADDRESSS *server_ip,
                                   UINT  server_port,
                                   ULONG wait_option)
```

Description

This function makes TCP connection using a previously created TCP client socket to the specified server's port for either IPv6 or IPv4 networks. Valid TCP server ports range from 0 through 0xFFFF. NetX Duo determines the appropriate physical interface based on the server IP address. The NetX IPv4 equivalent is *nx_tcp_client_socket_connect*.

The socket must be bound to a local port.

Parameters

socket_ptr	Pointer to previously created TCP socket
server_ip	Pointer to IPv4 or IPv6 destination address, in host byte order
server_port	Server port number to connect to (1 through 0xFFFF), in host byte order
wait_option	Wait option while the connection is being established. Valid wait options are: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) Timeout value (0x01 - 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket connect
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to tx_thread_wait_abort
NX_NOT_BOUND	(0x24)	Socket is not bound
NX_NOT_CLOSED	(0x35)	Socket is not in a closed state

NX_IN_PROGRESS	(0x37)	No wait was specified, connection attempt is in progress
NX_INVALID_INTERFACE	(0x4C)	Invalid interface index.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid server IPv4 or IPv6 address
NX_NOT_ENABLED	(0x14)	TCP not enabled
NX_INVALID_PORT	(0x46)	Invalid port
NX_PTR_ERROR	(0x07)	Invalid socket pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Preemption Possible

Yes

Example

```
NXD_ADDRESS  ip_address, server_address;

/* Set the host IPv6 address and set the version to IPv6. If we were
   connecting to an IPv4 host we would set the 32 bit IP address in
   ip_address.nxd_ip_address.v4 and set the version to
   NX_IP_VERSION_V4. */

ip_address.nxd_ip_version = NX_IP_VERSION_V6;
ip_address.nxd_ip_address.v6[0] = 0x20010000;
ip_address.nxd_ip_address.v6[1] = 0;
ip_address.nxd_ip_address.v6[2] = 0;
ip_address.nxd_ip_address.v6[3] = 1;

/* Set the TCP server IPv6 address to connect to. */
server_address.nxd_ip_version = NX_IP_VERSION_V6;
server_address.nxd_ip_address.v6[0] = 0x20010000;
server_address.nxd_ip_address.v6[1] = 0;
server_address.nxd_ip_address.v6[2] = 0;
server_address.nxd_ip_address.v6[3] = 2;

/* Set the global address (indicated by the 64 bit prefix) using the IPv6
   address just created on the primary interface. We don't need the index
   into the address table, so the last argument is set to null. *
interface_id = 0;
status = nxd_ipv6_address_set(&ip_0, interface_id,
                              &ip_address, 64, NX_NULL);

/* Create the TCP socket client_socket with the ip_address. */

/* Connect to the TCP server on port 12 with a 100 tick wait option. */
status = nxd_tcp_client_socket_connect(&client_socket,
                                       &server_address, 12, 100);

/* If status == NX_SUCCESS, the TCP client has successfully connected
   with the server with the socket in the ESTABLISHED state. */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect,
nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind,
nx_tcp_enable, nx_tcp_free_port_find,
nx_tcp_info_get,nx_tcp_server_socket_accept,
nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten,
nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten,
nx_tcp_socket_bytes_available, nx_tcp_socket_create,
nx_tcp_socket_delete, nx_tcp_socket_disconnect,
nx_tcp_socket_info_get, nx_tcp_socket_mss_get,
nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set,
nx_tcp_socket_peer_info_get, nx_tcp_socket_receive,
nx_tcp_socket_receive_notify, nx_tcp_socket_send,
nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure,
nxd_tcp_socket_peer_information_get

nxd_udp_packet_info_extract

Extract network parameters from UDP packet

Prototype

```
UINT nxd_udp_packet_info_extract(NX_PACKET *packet_ptr,
                                NXD_ADDRESS *ip_address,
                                UINT *protocol,
                                UINT *port,
                                UINT *interface_index);
```

Description

This function extracts network parameters from a packet received on an incoming interface for either IPv4 or IPv6 UDP networks. The NetX equivalent is ***nx_udp_packet_info_extract***

Parameters

packet_ptr	Pointer to packet.
ip_address	Pointer to sender IP address.
protocol	Pointer to protocol (UDP).
port	Pointer to sender's port number.
interface_index	Pointer to network interface index.

Return Values

NX_SUCCESS	(0x00)	Packet interface data successfully extracted.
NX_INVALID_PACKET	(0x12)	Packet does not contain IPv4 frame.
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Initialization, threads, timers, ISRs

Preemption Possible

No

Example

```
/* Extract network data from UDP packet interface.  */
status = nxd_udp_packet_info_extract(packet_ptr, &ip_address,
                                     &protocol, &port,
                                     &interface_index)

/* If status is NX_SUCCESS packet data was successfully retrieved. */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract, nx_udp_packet_info_extract

nxd_udp_socket_send

Send a UDP Datagram

Prototype

```
UINT nxd_udp_socket_send(NX_UDP_SOCKET *socket_ptr,
                         NX_PACKET *packet_ptr,
                         NXD_ADDRESS *ip_address,
                         UINT port);
```

Description

This function sends a UDP datagram through a previously created and bound UDP socket for either IPv4 or IPv6 networks. NetX Duo finds a suitable network interface based on the destination IP address. To specify a specific interface and source IP address, the application developer should use the nxd_ipv6_udp_socket_interface_send service. Note that the function returns immediately regardless of whether the UDP datagram was successfully sent. The NetX (IPv4) equivalent service is *nx_udp_socket_send*.

The socket must be bound to a local port.

Parameters

socket_ptr	Pointer to previously created UDP socket instance
packet_ptr	UDP datagram packet pointer
ip_address	Pointer to destination IPv4 or IPv6 address
port	Valid destination port number between 1 and 0xFFFF), in host byte order

Return Values

NX_SUCCESS	(0x00)	Successful socket connect
NX_NOT_BOUND	(0x24)	Socket not bound to any port
NX_NO_INTERFACE_ADDRESS	(0x50)	No suitable outgoing interface can be found.
NX_UNDERFLOW	(0x02)	Not enough room for UDP header in the packet
NX_OVERFLOW	(0x07)	Packet append pointer is invalid
NX_PTR_ERROR	(0x07)	Invalid socket pointer

NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_NOT_ENABLED	(0x14)	UDP has not been enabled
NX_IP_ADDRESS_ERROR	(0x21)	Invalid server IPv4 or IPv6 address
NX_INVALID_PORT	(0x46)	Port number is not within a valid range

Allowed From

Threads

Preemption Possible

No

Example

```
NXD_ADDRESS  ip_address, server_address;

/* Set the UDP Client IPv6 address. */
ip_address.nxd_ip_version = NX_IP_VERSION_V6;
ip_address.nxd_ip_address.v6[0] = 0x20010000;
ip_address.nxd_ip_address.v6[1] = 0;
ip_address.nxd_ip_address.v6[2] = 0;
ip_address.nxd_ip_address.v6[3] = 1;

/* Set the UDP server IPv6 address to send to. */
server_address.nxd_ip_version = NX_IP_VERSION_V6;
server_address.nxd_ip_address.v6[0] = 0x20010000;
server_address.nxd_ip_address.v6[1] = 0;
server_address.nxd_ip_address.v6[2] = 0;
server_address.nxd_ip_address.v6[3] = 2;

/* Set the global address (indicated by the 64 bit prefix) using the IPv6
   address just created on the primary interface (index 0). We don't need
   the index into the address table, so the last argument is set to null. */

interface_id = 0;
status = nxd_ipv6_address_set(&client_ip, interface_id,
                              &server_address, 64, NX_NULL);

/* Create the UDP socket client_socket with the ip_address and */
/* allocate a packet pointed to by packet_ptr (not shown). */

/* Send a packet to the UDP server at server_address on port 12. */
status = nxd_udp_socket_send(&client_socket, packet_ptr,
                             &server_address, 12);

/* If status == NX_SUCCESS, the UDP host successfully transmitted the
   packet out the UDP socket to the server. */
```

See Also

nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available,= nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract, nxd_udp_socket_extract

nxd_udp_socket_interface_send

Send a UDP Datagram

Prototype

```
UINT nxd_udp_socket_interface_send(NX_UDP_SOCKET *socket_ptr,
                                   NX_PACKET *packet_ptr,
                                   NXD_ADDRESS *ip_address,
                                   UINT port, UINT address_index);
```

Description

This function sends a UDP datagram through a previously created and bound UDP socket for either IPv4 or IPv6 networks. The parameter *address_index* specifies the source IP address to use for the outgoing packet. Note that the function returns immediately regardless of whether the UDP datagram was successfully sent.

The socket must be bound to a local port.

The NetX (IPv4) equivalent service is *nx_udp_socket_interface_send*.

Parameters

socket_ptr	Pointer to previously created UDP socket instance
packet_ptr	UDP datagram packet pointer
ip_address	Pointer to destination IPv4 or IPv6 address
port	Valid destination port number between 1 and 0xFFFF), in host byte order
address_index	Index specifying the address interface

Return Values

NX_SUCCESS	(0x00)	Successful socket connect
NX_NOT_BOUND	(0x24)	Socket not bound to any port
NX_NO_INTERFACE_ADDRESS	(0x50)	No suitable outgoing interface can be found.

NX_NOT_FOUND	(0x4E)	No suitable interface can be found
NX_PTR_ERROR	(0x07)	Invalid socket pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_NOT_ENABLED	(0x14)	UDP has not been enabled
NX_IP_ADDRESS_ERROR	(0x21)	Invalid server IPv4 or IPv6 address
NX_INVALID_PORT	(0x46)	Port number is not within valid range.
NX_INVALID_INTERFACE	(0x4C)	Specified network interface is valid

Allowed From

Threads

Preemption Possible

No

Example

```
NXD_ADDRESS  ip_address, server_address;

/* Set the UDP Client IPv6 address. */
ip_address.nxd_ip_version = NX_IP_VERSION_V6;
ip_address.nxd_ip_address.v6[0] = 0x20010000;
ip_address.nxd_ip_address.v6[1] = 0;
ip_address.nxd_ip_address.v6[2] = 0;
ip_address.nxd_ip_address.v6[3] = 1;

/* Set the UDP server IPv6 address to send to. */
server_address.nxd_ip_version = NX_IP_VERSION_V6;
server_address.nxd_ip_address.v6[0] = 0x20010000;
server_address.nxd_ip_address.v6[1] = 0;
server_address.nxd_ip_address.v6[2] = 0;
server_address.nxd_ip_address.v6[3] = 2;

 /* Set the global address (indicated by the 64 bit prefix) using the IPv6
    address just created on the primary interface (index 0). We don't need
    the index into the address table, so the last argument is set to null.
    */

 status = nxd_ipv6_address_set(&client_ip, 0,
                               &server_address, 64, NX_NULL);

/* Create the UDP socket client_socket with the ip_address and */
/* allocate a packet pointed to by packet_ptr (not shown). */

/* Send a packet to the UDP server at server_address on port 12. */
status = nxd_udp_socket_interface_send(&client_socket, packet_ptr,
                                       &server_address, 12, address_index);

/* If status == NX_SUCCESS, the UDP host successfully transmitted the
   packet out the UDP socket to the server. */
```

See Also

nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract, nxd_udp_socket_extract, nxd_udp_socket_set_interface

nxd_udp_source_extract

Retrieve UPD Packet Source Information

Prototype

```
UINT nxd_udp_source_extract(NX_PACKET *packet_ptr,
                            NXD_ADDRESS *ip_address, UINT *port)
```

Description

This function extracts the source IP address and port number from a UDP packet received through the host UDP socket. The NetX (IPv4) equivalent is *nx_udp_source_extract*.

Parameters

packet_ptr	Pointer to received UDP packet
ip_address	Pointer to NXD_ADDRESS to store packet source IP address
port	Pointer to UDP socket port number

Return Values

NX_SUCCESS	(0x00)	Successful source extract
NX_INVALID_PACKET	(0x12)	Packet is not valid
NX_PTR_ERROR	(0x07)	Invalid socket pointer

Allowed From

Threads

Preemption Possible

No

Example

```
NXD_ADDRESS  ip_address;
UINT          port;

/* Create the UDP socket client_socket and */
/* allocate the packet pointed to by packet_ptr (not shown). */

/* Extract the IP address and port of the packet received on the UDP
   socket specified in the packet interface. */
status = nxd_udp_source_extract(&packet_ptr, &ip_address, &port);

/* If status == NX_SUCCESS, the source IP address and port of the
   packet received on the UDP socket was successfully extracted. */
```

See Also

nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract, nxd_udp_socket_send

CHAPTER 5

NetX Duo Network Drivers

This chapter contains a description of network drivers for NetX Duo. The information presented is designed to help developers write application-specific network drivers for NetX Duo. The following topics are covered:

Driver Introduction

NetX Duo supports multiple IP instances. The NX_IP structure contains everything to manage a single IP instance. This includes general TCP/IP protocol information as well as the application-specific physical network driver's entry routine. The driver's entry routine is defined during the ***nx_ip_create*** service.

Communication between NetX Duo and the application's network driver is accomplished through the **NX_IP_DRIVER** request structure. This structure is most often defined locally on the caller's stack and is therefore released after the driver and calling function return. The structure is defined as follows:

```
typedef struct NX_IP_DRIVER_STRUCT
{
      UINT            nx_ip_driver_command;
      UINT            nx_ip_driver_status;
      ULONG           nx_ip_driver_physical_address_msw;
      ULONG           nx_ip_driver_physical_address_lsw;
      NX_PACKET       *nx_ip_driver_packet;
      ULONG           *nx_ip_driver_return_ptr;
      NX_IP           *nx_ip_driver_ptr;
      NX_INTERFACE      *nx_ip_driver_interface;
} NX_IP_DRIVER;
```

Driver Entry

NetX Duo invokes the network driver entry function for sending packets and for various control and status operations, including initializing and enabling the network interface. NetX Duo issues commands to the network driver by setting the ***nx_ip_driver_command*** field in the **NX_IP_DRIVER** request structure. As mentioned previously, the network driver is specified in the

nx_ip_create service call. The driver entry function has the following format:

```
VOID my_driver_entry(NX_IP_DRIVER *request);
```

Driver Requests

NetX Duo creates the driver request with a send command and invokes the driver entry function to execute the command. Because each network driver has a single entry function, NetX Duo makes all requests through the driver request data structure. The ***nx_ip_driver_command*** member of the driver request data structure (**NX_IP_DRIVER**) defines the request. Status information is reported back to the caller in the member ***nx_ip_driver_status***. If this field is **NX_SUCCESS**, the driver request was completed successfully.

NetX Duo serializes all access to the driver. Therefore, the driver does not need to handle multiple threads asynchronously calling the entry function.

Driver Initialization

Although the actual driver initialization processing is application specific, it usually consists of data structure and physical hardware initialization. The information required from NetX Duo for driver initialization is the network Maximum Transmission Unit (MTU) and whether the physical interface needs logical-to-physical mapping. When the network driver receives the NX_LINK INITIALIZE request from NetX Duo, it receives a pointer to the IP control block as part of the NX_IP_DRIVER request control block shown above. The driver then updates the following fields of the interface instance control block:

nx_interface_ip_mtu_size
nx_interface_address_mapping_needed
nx_interface_physical_address_msw

nx_interface_physical_address_lsw

After the application calls ***nx_ip_create***, the IP helper thread sends a driver request driver with the command set to NX_LINK_INITIALIZE to the driver to initialize its physical network interface. The following NX_IP_DRIVER members are used for the initialize request.

NX_IP_DRIVER member	**Meaning**
nx_ip_driver_command	NX_LINK_INITIALIZE
nx_ip_driver_ptr	Pointer to the IP instance. This should be saved for processing receive packets.
nx_ip_driver_interface	Pointer to the interface instance within the IP instance. This should be saved for processing receive packets.

i

The driver is actually called from the IP helper thread that was created for the IP instance. Because of this, it may suspend during the initialization request, if the physical media initialization requires it.

Enable Link

Next, the IP helper thread enables the physical network by setting the driver command to NX_LINK_ENABLE in the driver request and sending the request to the network driver. This happens shortly after the IP helper thread completes the initialization request. Enabling the link may be as simple as setting the *nx_interface_link_up* field in the interface instance. But it may also involve manipulation of the physical hardware. The following

NX_IP_DRIVER members are used for the enable link request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_ENABLE
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the interface instance

Disable Link

This request is made by NetX Duo during the deletion of an IP instance by the ***nx_ip_delete*** service. This service disables each physical network interface on the IP instance. The processing to disable the link may be as simple as clearing the *nx_interface_link_up* flag in the interface instance. But it may also involve manipulation of the physical hardware. The following NX_IP_DRIVER members are used for the disable link request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_DISABLE
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the interface instance

Packet Send

This request is made during internal IP send processing, which all NetX Duo protocols use to transmit packets (except for ARP and RARP). The packet send processing places a physical media header on the front of the packet and then calls the driver's output function to transmit the packet. The following NX_IP_DRIVER members are used for the packet send request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_PACKET_SEND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_interface	Pointer to the interface instance.
nx_ip_driver_physical_address_msw	Most significant 32-bits of physical address (only if physical mapping needed)
nx_ip_driver_physical_address_lsw	Least significant 32-bits of physical address (only if physical mapping needed)

Packet Broadcast (IPv4 packets only)

This request is almost identical to the send packet request. The only difference is that the physical address fields are set to the Ethernet broadcast MAC address. The following NX_IP_DRIVER members are used for the packet broadcast request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_PACKET_BROADCAST
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_physical_address_msw	0x0000FFFF (broadcast)
nx_ip_driver_physical_address_lsw	0xFFFFFFFF (broadcast)
nx_ip_driver_interface	Pointer to the interface instance.

ARP Send

This request is also similar to the IP packet send request. The only difference is that the Ethernet header specifies an ARP packet instead of an IP packet, and physical address fields are must be set to broadcast address. The following NX_IP_DRIVER members are used for the ARP send request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_ARP_SEND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_physical_address_msw	0x0000FFFF (broadcast)
nx_ip_driver_physical_address_lsw	0xFFFFFFFF (broadcast)
nx_ip_driver_interface	Pointer to the interface instance

If physical mapping is not needed, implementation of this request is not required.

Although ARP has been replaced with the Neighbor Discovery Protocol and the Router Discovery Protocol in IPv6, Ethernet network drivers must still be compatible with IPv4 peers and routers. Therefore, drivers must still handle ARP packets.

ARP Response Send

This request is almost identical to the ARP send packet request. The only difference is the physical address fields are required. The following NX_IP_DRIVER members are used for the ARP response send request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_ARP_RESPONSE_SEND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_physical_address_msw	Most significant 32-bits of physical address

NX_IP_DRIVER member	Meaning
nx_ip_driver_physical_address_lsw	Least significant 32-bits of physical address
nx_ip_driver_interface	Pointer to the interface instance

If physical mapping is not needed, implementation of this request is not required.

RARP Send

This request is almost identical to the ARP send packet request. The only differences are the type of packet header and the physical address fields are not required because the physical destination is always a broadcast. The following NX_IP_DRIVER members are used for the RARP send request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_RARP_SEND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_physical_address_msw	0x0000FFFF (broadcast)
nx_ip_driver_physical_address_lsw	0xFFFFFFFF (broadcast)
nx_ip_driver_interface	Pointer to the interface instance

Applications that require RARP service must implement this command.

Multicast Group Join

This request is made with the ***nx_igmp_multicast_join*** service in IPv4, and various operation required by IPv6. The network driver takes the supplied multicast group address and sets up the physical media to accept incoming packets from that address.

The following NX_IP_DRIVER members are used for the multicast group join request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_MULTICAST_JOIN
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_physical_address_msw	Most significant 32-bits of physical multicast address
nx_ip_driver_physical_address_lsw	Least significant 32-bits of physical multicast address
nx_ip_driver_interface	Pointer to the interface instance

IPv6 applications will require multicast to be implemented in the driver for ICMPv6 based protocols such as address configuration. However, for IPv4 applications, implementation of this request is not necessary if multicast capabilities are not required.

If IPv6 is not enabled, and multicast capabilities are not required by IPv4, implementation of this request is not required.

Multicast Group Leave

This request is invoked by explicitly calling the ***nx_igmp_multicast_leave*** service in IPv4, or by various internal NetX Duo operations required for IPv6. The driver removes the supplied Ethernet multicast address from the multicast join list. After a host has left a multicast group, packets on the network with this Ethernet multicast address are no longer received by this IP instance. The following

NX_IP_DRIVER members are used for the multicast group leave request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_MULTICAST_LEAVE
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_physical_address_msw	Most significant 32-bits of physical multicast address
nx_ip_driver_physical_address_lsw	Least significant 32-bits of physical multicast address
nx_ip_driver_interface	Pointer to the interface instance

If multicast capabilities are not required by either IPv4 or IPv6, implementation of this request is not required.

Attach Interface

This request is invoked from the NetX Duo to the device driver, allowing the driver to associate the driver instance with the corresponding IP instance and the physical interface instance within the IP. The following NX_IP_DRIVER members are used for the attach interface request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_INTERFACE_ATTACH
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the interface instance.
nx_ip_driver_status	Completion status. If the driver is not able to attach the specified interface to the IP instance, it will return a non-zero error status.

Get Link Status

The host application can query the primary interface link status using the NetX Duo service

nx_ip_interface_status_check service for any interface on the host. See Chapter 4, “Description of NetX Duo Services” on page 133, for more details on these services.

The the link status is contained in the *nx_interface_link_up* field in the NX_INTERFACE structure pointed to by *nx_ip_driver_interface* pointer.The following NX_IP_DRIVER members are used for the link status request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_STATUS
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the status.
nx_ip_driver_interface	Pointer to the interface instance

i

nx_ip_status_check *is still available for checking the status of the primary interface. However, application developers are encouraged to use the interface specific service:* ***nx_ip_interface_status_check.***

Get Link Speed

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link’s line speed in the supplied destination. The following NX_IP_DRIVER members are used for the link line speed request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_SPEED
nx_ip_driver_ptr	Pointer to IP instance

NX_IP_DRIVER member	Meaning
nx_ip_driver_return_ptr	Pointer to the destination to place the line speed
nx_ip_driver_interface	Pointer to the interface instance

This request is not used internally by NetX Duo so its implementation is optional.

Get Duplex Type

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's duplex type in the supplied destination. The following NX_IP_DRIVER members are used for the duplex type request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_DUPLEX_TYPE
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the duplex type
nx_ip_driver_interface	Pointer to the interface instance

This request is not used internally by NetX Duo so its implementation is optional.

Get Error Count

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's error count in the supplied destination. The following NX_IP_DRIVER members are used for the link error count request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_ERROR_COUNT
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the error count
nx_ip_driver_interface	Pointer to the interface instance

This request is not used internally by NetX Duo so its implementation is optional.

Get Receive Packet Count

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's receive packet count in the supplied destination. The following NX_IP_DRIVER members are used for the link receive packet count request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_RX_COUNT
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the receive packet count
nx_ip_driver_interface	Pointer to the physical network interface

This request is not used internally by NetX Duo so its implementation is optional.

Get Transmit Packet Count

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's transmit packet count in the supplied destination. The following NX_IP_DRIVER members are used for the link transmit packet count request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_TX_COUNT
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the transmit packet count
nx_ip_driver_interface	Pointer to the interface instance

This request is not used internally by NetX Duo so its implementation is optional.

Get Allocation Errors

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's allocation error count in the supplied destination. The following NX_IP_DRIVER members are used for the link allocation error count request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_ALLOC_ERRORS
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the allocation error count
nx_ip_driver_interface	Pointer to the interface instance

This request is not used internally by NetX Duo so its implementation is optional.

Driver Deferred Processing

This request is made from the IP helper thread in response to the driver calling the ***_nx_ip_driver_deferred_processing*** routine from a transmit or receive ISR. This allows the driver ISR to defer the packet receive and transmit processing to the IP helper thread and thus reduce the amount to processing in the ISR. The *nx_interface_additional_link_info* field in the NX_INTERFACE structure pointed to by *nx_ip_driver_interface* may be used by the driver to store information about the deferred processing event from the IP helper thread context. The following NX_IP_DRIVER members are used for the deferred processing event.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_DEFERRED_PROCESSING
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the interface instance

User Commands

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver processes the application specific user commands. The following NX_IP_DRIVER members are used for the user command request.

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_USER_COMMAND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	User defined
nx_ip_driver_interface	Pointer to the interface instance

This request is not used internally by NetX Duo so its implementation is optional.

Unimplemented Commands

Commands unimplemented by the network driver must have the return status field set to NX_UNHANDLED_COMMAND.

Driver Output

All previously mentioned packet transmit requests require an output function implemented in the driver. Specific transmit logic is hardware specific, but it usually consists of checking for hardware capacity to send the packet immediately. If possible, the packet payload (and additional payloads in the packet chain) are loaded into one or more of the hardware transmit buffers and a transmit operation is initiated. If the packet won't fit in the available transmit buffers, the packet should be queued, and be transmitted when the transmission buffers become available.

The recommended transmit queue is a singly linked list, having both head and tail pointers. New packets are added to the end of the queue, keeping the oldest packet at the front. The *nx_packet_queue_next* field is used as the packet's next link in the queue. The driver defines the head and tail pointers of the transmit queue.

Because this queue is accessed from thread and interrupt portions of the driver, interrupt protection must be placed around the queue manipulations.

Most physical hardware implementations generate an interrupt upon packet transmit completion. When the driver receives such an interrupt, it typically releases the resources associated with the packet just being transmitted. In case the transmit logic reads data directly from the NX_PACKET buffer, the driver should use the ***nx_packet_transmit_release*** service to release the packet associated with the transmit complete interrupt back to the available packet pool. Next, the driver examines the transmit queue for additional packets waiting to be sent. As many of the queued transmit packets that fit into the hardware transmit buffer(s) are de-queued and loaded into the buffers. This is followed by initiation of another send operation.

As soon as the data in the NX_PACKET has been moved into the transmitter FIFO (or in case a driver supports zero-copy operation, the data in NX_PACKET has been transmitted), the driver must move the nx_packet_prepend_ptr to the beginning of the IP header before calling ***nx_packet_transmit_release.*** Remember to adjust *nx_packet_length* field accordingly. If an IP frame is made up of multiple packets, only the head of the packet chain needs to be released.

Driver Input

Upon reception of a received packet interrupt, the network driver retrieves the packet from the physical hardware receive buffers and builds a valid NetX Duo packet. Building a valid NetX Duo packet involves setting up the appropriate length field and chaining together multiple packets if the incoming packet's size was greater than a single packet payload. Once properly built, the physical layer header is removed and the receive packet is dispatched to NetX Duo.

NetX Duo assumes that the IP (IPv4 and IPv6) and ARP headers are aligned on a ULONG boundary. The NetX Duo driver must, therefore, ensure this alignment. In Ethernet environments this is done by starting the Ethernet header two bytes from the beginning of the packet. When the *nx_packet_prepend_ptr* is moved beyond the Ethernet header, the underlying IP (IPv4 and IPv6) or ARP header is 4-byte aligned.

!

See the section Ethernet Headers below for important differences between IPv6 and IPv6 Ethernet headers.

There are several receive packet functions available in NetX Duo. If the received packet is an ARP packet, ***_nx_arp_packet_deferred_receive*** is called. If the received packet is an RARP packet,

_nx_rarp_packet_deferred_receive is called. There are several options for handling incoming IP packets. For the fastest handling of IP packets, ***_nx_ip_packet_receive*** is called. This approach has the least overhead, but requires more processing in the driver's receive interrupt service handler (ISR). For minimal ISR processing ***_nx_ip_packet_deferred_receive*** is called.

After the new receive packet is properly built, the physical hardware's receive buffers are setup to receive more data. This might require allocating NetX Duo packets and placing the payload address in the hardware receive buffer or it may simply amount to changing a setting in the hardware receive buffer. To minimize overrun possibilities, it is important that the hardware's receive buffers have available buffers as soon as possible after a packet is received.

i

The initial receive buffers are setup during driver initialization.

Deferred Receive Packet Handling

The driver may defer receive packet processing to the NetX Duo IP helper thread. For some applications this may be necessary to minimize ISR processing as well as dropped packets.

To use deferred packet handling, the NetX Duo library must first be compiled with *NX_DRIVER_DEFERRED_PROCESSING* defined. This adds the deferred packet logic to the NetX Duo IP helper thread. Next, on receiving a data packet, the driver must call *_nx_ip_packet_deferred_receive():*

```
_nx_ip_packet_deferred_receive(ip_ptr, packet_ptr);
```

The deferred receive function places the receive packet represented by *packet_ptr* on a FIFO (linked list) and notifies the IP helper thread. After executing, the IP helper repetitively calls the deferred handling function to process each deferred packet. The deferred handler processing typically includes removing the packet's physical layer header (usually Ethernet) and dispatching it to one of these NetX Duo receive functions:

_nx_ip_packet_receive
_nx_arp_packet_deferred_receive
_nx_rarp_packet_deferred_receive

Ethernet Headers

One of the most significant differences between IPv6 and IPv4 for Ethernet Headers is the frame type setting. When sending out packets, the Ethernet driver is responsible for setting the Ethernet frame type in outgoing packets. For IPv6 packets, the frame type should be 0x86DD; for IPv4 packets, the frame type should be 0x800.

The following code segment illustrates this process:

```
NX_PACKET *packet_ptr;
packet_ptr = driver_req_ptr -> nx_ip_driver_packet;
if (packet_ptr -> nx_packet_ip_version == NX_IP_VERSION_V4)
{
    /* Set Ethernet frame type to IPv4  /*
    ethernet_frame_ptr -> frame_type = 0x0800;

    /* Swap endian-ness for little endian targets.*/
    NX_CHANGE_USHORT_ENDIAN(ethernet_frame_ptr -> frame_type);
}
else if (packet_ptr -> nx_packet_ip_version == NX_IP_VERSION_V6)
{
    /* Set Ethernet frame type to IPv6. /*
    ethernet_frame_ptr -> frame_type = 0x86DD;

    /* Swap endian-ness for little endian targets.*/
    NX_CHANGE_USHORT_ENDIAN(ethernet_frame_ptr -> frame_type);
}
else
{
    /* Unknown IP version. Free the packet we will not send. */
    nx_packet_transmit_release(packet_ptr);
}
```

Similarly, for incoming packets, the Ethernet driver should determine the packet type from the Ethernet frame type. It should be implemented to accept IPv6 (0x86DD), IPv4 (0x0800), ARP (0x0806), and RARP (0x8035) frame types.

Example RAM Ethernet Network Driver

The NetX Duo demonstration system is delivered with a small RAM network driver, defined in the file ***nx_ram_network_driver.c***. This driver assumes the IP instances are all on the same network and simply assigns virtual hardware addresses to each IP instance as they are created. This file provides a good example of the basic structure for NetX Duo physical network drivers and is listed on the following page.

For multihome hosts, this driver assumes each IP instance interface exchanges packets with another IP instance on the same network interface. The RAM driver assigns virtual hardware addresses to each interface as they are created.

```
/**************************************************************************/
/*                                                                        */
/*            Copyright (c) 1996-2011 by Express Logic Inc.               */
/*                                                                        */
/*  This software is copyrighted by and is the sole property of Express   */
/*  Logic, Inc.  All rights, title, ownership, or other interests         */
/*  in the software remain the property of Express Logic, Inc.  This      */
/*  software may only be used in accordance with the corresponding        */
/*  license agreement.  Any unauthorized use, duplication, transmission,  */
/*  distribution, or disclosure of this software is expressly forbidden.  */
/*                                                                        */
/*  This Copyright notice may not be removed or modified without prior    */
/*  written consent of Express Logic, Inc.                                */
/*                                                                        */
/*  Express Logic, Inc. reserves the right to modify this software        */
/*  without notice.                                                       */
/*                                                                        */
/*  Express Logic, Inc.                     info@expresslogic.com         */
/*  11423 West Bernardo Court               http://www.expresslogic.com   */
/*  San Diego, CA  92127                                                  */
/*                                                                        */
/**************************************************************************/


/**************************************************************************/
/**************************************************************************/
/**                                                                       */
/** NetX Duo Component                                                    */
/**                                                                       */
/**   RAM Network (RAM)                                                   */
/**                                                                       */
/**************************************************************************/
/**************************************************************************/


/* Include necessary system files.  */

#include "nx_api.h"


     /* Define the Link MTU. Note this is not the same as the IP MTU. The Link MTU
        includes the addition of the Physical Network header (usually Ethernet). This
        should be larger than the IP instance MTU by the size of the physical header. */
#define NX_LINK_MTU     1514


/* Define Ethernet address format.  This is prepended to the incoming IP
   and ARP/RARP messages.  The frame beginning is 14 bytes, but for speed
   purposes, we are going to assume there are 16 bytes free in front of the
   prepend pointer and that the prepend pointer is 32-bit aligned.

    Byte Offset     Size            Meaning

        0           6           Destination Ethernet Address
        6           6           Source Ethernet Address
        12          2           Ethernet Frame Type, where:

                                        0x0800 -> IP Datagram
                                        0x0806 -> ARP Request/Reply
                                        0x0835 -> RARP request reply

        42          18          Padding on ARP and RARP messages only.  */

#define NX_ETHERNET_IP      0x0800
#define NX_ETHERNET_ARP     0x0806
#define NX_ETHERNET_RARP    0x8035
#define NX_ETHERNET_IPV6    0x86DD
#define NX_ETHERNET_SIZE    14

/* For the simulated ethernet driver, physical addresses are allocated starting
   at the preset value and then incremented before the next allocation.  */

ULONG   simulated_address_msw =  0x0011;
ULONG   simulated_address_lsw =  0x22334456;
```

```
/* Define driver prototypes.  */

VOID    _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr);
void    _nx_ram_network_driver_output(NX_IP *ip_ptr, NX_PACKET *packet_ptr, UINT device_instance_id);
void    _nx_ram_network_driver_receive(NX_IP *ip_ptr, NX_PACKET *packet_ptr, UINT
device_instance_id);

#define NX_MAX_RAM_INTERFACES 4
#define NX_RAM_DRIVER_MAX_MCAST_ADDRESSES 2
typedef struct MAC_ADDRESS_STRUCT
{
    ULONG nx_mac_address_msw;
    ULONG nx_mac_address_lsw;


} MAC_ADDRESS;


/* Define an application-specific data structure that holds internal
   data (such as the state information) of a device driver.

   The example below applies to the simulated RAM driver.
   The user shall replace its content with information related to
   the actual driver being used. */
typedef struct _nx_ram_network_driver_instance_type
{
    UINT                 nx_ram_network_driver_in_use;

    UINT                 nx_ram_network_driver_id;

    NX_INTERFACE        *nx_ram_driver_interface_ptr;

    NX_IP               *nx_ram_driver_ip_ptr;

    MAC_ADDRESS          nx_ram_driver_mac_address;

    MAC_ADDRESS          nx_ram_driver_mcast_address[NX_RAM_DRIVER_MAX_MCAST_ADDRESSES];

} _nx_ram_network_driver_instance_type;


/* In this example, there are four instances of the simulated RAM driver.
   Therefore an array of four driver instances are created to keep track of
   the device information of each driver. */
static _nx_ram_network_driver_instance_type nx_ram_driver[NX_MAX_RAM_INTERFACES];


/**************************************************************************/
/*                                                                        */
/*  FUNCTION                                               RELEASE        */
/*                                                                        */
/*    _nx_ram_network_driver                              PORTABLE C      */
/*                                                                        */
/*  AUTHOR                                                                */
/*                                                                        */
/*    Express Logic, Inc.                                                 */
/*                                                                        */
/*  DESCRIPTION                                                           */
/*                                                                        */
/*    This function acts as a virtual network for testing the NetX Duo    */
/*    source and driver concepts.   User application may use this routine */
/*    as a template for the actual network driver.  Note that this driver */
/*    simulates Ethernet operation.  Some of the parameters don't apply   */
/*    for non-Ethernet devices.                                           */
/*                                                                        */
/*  INPUT                                                                 */
/*                                                                        */
/*    ip_ptr                                Pointer to IP protocol block  */
```

```
/*                                                                        */
/*  OUTPUT                                                                */
/*                                                                        */
/*    None                                                                */
/*                                                                        */
/*  CALLS                                                                 */
/*                                                                        */
/*    _nx_ram_network_driver_output         Send physical packet out      */
/*                                                                        */
/*  CALLED BY                                                             */
/*                                                                        */
/*    NetX Duo IP processing                                              */
/*                                                                        */
/*                                                                        */
/**************************************************************************/
VOID  _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr)
{
UINT             i;
NX_IP           *ip_ptr;
NX_PACKET       *packet_ptr;
ULONG           *ethernet_frame_ptr;
NX_INTERFACE    *interface_ptr;


    /* Setup the IP pointer from the driver request.  */
    ip_ptr =  driver_req_ptr -> nx_ip_driver_ptr;

    /* Default to successful return.  */
    driver_req_ptr -> nx_ip_driver_status =  NX_SUCCESS;

    /* Setup interface pointer.  */
    interface_ptr = driver_req_ptr -> nx_ip_driver_interface;


    /* Find out the driver interface if the driver command is not ATTACH. */
    if(driver_req_ptr -> nx_ip_driver_command != NX_LINK_INTERFACE_ATTACH)
    {
        for(i = 0; i < NX_MAX_RAM_INTERFACES;i++)
        {
            if(nx_ram_driver[i].nx_ram_network_driver_in_use == 0)
                continue;

            if(nx_ram_driver[i].nx_ram_driver_ip_ptr != ip_ptr)
                continue;

            if(nx_ram_driver[i].nx_ram_driver_interface_ptr != driver_req_ptr -> nx_ip_driver_interface)
                continue;

            break;
        }

        if(i == NX_MAX_RAM_INTERFACES)
        {
            driver_req_ptr -> nx_ip_driver_status =  NX_INVALID_INTERFACE;
            return;
        }
    }


    /* Process according to the driver request type in the IP control
       block.  */
    switch (driver_req_ptr -> nx_ip_driver_command)
    {

        case NX_LINK_INTERFACE_ATTACH:
            /* Find an available driver instance to attach the interface. */
            for(i = 0; i < NX_MAX_RAM_INTERFACES;i++)
            {
                if(nx_ram_driver[i].nx_ram_network_driver_in_use == 0)
                    break;
            }
            /* An available entry is found. */
```

```
            if(i < NX_MAX_RAM_INTERFACES)
            {
                /* Set the IN USE flag.*/
                nx_ram_driver[i].nx_ram_network_driver_in_use  = 1;

                nx_ram_driver[i].nx_ram_network_driver_id = i;

                /* Record the interface attached to the IP instance. */
                nx_ram_driver[i].nx_ram_driver_interface_ptr = driver_req_ptr -> nx_ip_driver_interface;

                /* Record the IP instance. */
                nx_ram_driver[i].nx_ram_driver_ip_ptr = ip_ptr;

                nx_ram_driver[i].nx_ram_driver_mac_address.nx_mac_address_msw = simulated_address_msw;
                nx_ram_driver[i].nx_ram_driver_mac_address.nx_mac_address_lsw = simulated_address_lsw +
                              i;
            }
            else
                driver_req_ptr -> nx_ip_driver_status =  NX_INVALID_INTERFACE;

            break;

        case NX_LINK_INITIALIZE:
        {

            /* Device driver shall initialize the Ethernet Controller here. */

#ifdef NX_DEBUG
            printf("NetX Duo RAM Driver Initialization - %s\n", ip_ptr -> nx_ip_name);
            printf("  IP Address =%08X\n", ip_ptr -> nx_ip_address);
#endif

            /* Once the Ethernet controller is initialized, the driver needs to
               configure the NetX Duo Interface Control block, as outlined below. */

            /* The nx_interface_ip_mtu_size should be the MTU for the IP payload.
               For regular Ethernet, the IP MTU is 1500. */
            interface_ptr -> nx_interface_ip_mtu_size =  (NX_LINK_MTU - NX_ETHERNET_SIZE);

            /* Set the physical address (MAC address) of this IP instance.  */
            /* For this simulated RAM driver, the MAC address is contructed by
               incrementing a base lsw value, to simulate multiple nodes hanging on the
               ethernet.  */
            interface_ptr -> nx_interface_physical_address_msw =
                              nx_ram_driver[i].nx_ram_driver_mac_address.nx_mac_address_msw;
            interface_ptr -> nx_interface_physical_address_lsw =
                              nx_ram_driver[i].nx_ram_driver_mac_address.nx_mac_address_lsw;

            /* Indicate to the IP software that IP to physical mapping is required.  */
            interface_ptr -> nx_interface_address_mapping_needed =  NX_TRUE;

            break;
        }

        case NX_LINK_ENABLE:
        {

            /* Process driver link enable.  An Ethernet driver shall enable the
               transmit and reception logic.  Once the IP stack issures the
               LINK_ENABLE command, the stack may start transmitting IP packets. */


            /* In the RAM driver, just set the enabled flag.  */
            interface_ptr -> nx_interface_link_up =  NX_TRUE;

#ifdef NX_DEBUG
            printf("NetX Duo RAM Driver Link Enabled - %s\n", ip_ptr -> nx_ip_name);
#endif
            break;
        }
```

```
        case NX_LINK_DISABLE:
        {

            /* Process driver link disable.  This command indicates the IP layer
               is not going to transmit any IP datagrams, nor does it expect any
               IP datagrams from the device.  Therefore after processing this command,
               the device driver shall not send any incoming packets to the IP
               layer.  Optionally the device driver may turn off the device. */

            /* In the RAM driver, just clear the enabled flag.  */
            interface_ptr -> nx_interface_link_up =  NX_FALSE;

#ifdef NX_DEBUG
            printf("NetX Duo RAM Driver Link Disabled - %s\n", ip_ptr -> nx_ip_name);
#endif
            break;
        }

        case NX_LINK_PACKET_SEND:
        case NX_LINK_PACKET_BROADCAST:
        case NX_LINK_ARP_SEND:
        case NX_LINK_ARP_RESPONSE_SEND:
        case NX_LINK_RARP_SEND:
        {

            /*
               The IP stack sends down a data packet for transmission.
               The device driver needs to prepend a MAC header, and fill in the
               Ethernet frame type (assuming Ethernet protocol for network transmission)
               based on the type of packet being transmitted.

               The following sequence illustrates this process.
            */


            /* Place the ethernet frame at the front of the packet.  */
            packet_ptr =  driver_req_ptr -> nx_ip_driver_packet;

            /* Adjust the prepend pointer.  */
            packet_ptr -> nx_packet_prepend_ptr =  packet_ptr -> nx_packet_prepend_ptr - NX_ETHERNET_SIZE;

            /* Adjust the packet length.  */
            packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length + NX_ETHERNET_SIZE;

            /* Setup the ethernet frame pointer to build the ethernet frame.  Backup another 2
               bytes to get 32-bit word alignment.  */
            ethernet_frame_ptr =  (ULONG *) (packet_ptr -> nx_packet_prepend_ptr - 2);

            /* Build the ethernet frame.  */
            *ethernet_frame_ptr     =  driver_req_ptr -> nx_ip_driver_physical_address_msw;
            *(ethernet_frame_ptr+1) =  driver_req_ptr -> nx_ip_driver_physical_address_lsw;
            *(ethernet_frame_ptr+2) =  (interface_ptr -> nx_interface_physical_address_msw << 16) |
                                       (interface_ptr -> nx_interface_physical_address_lsw >> 16);
            *(ethernet_frame_ptr+3) =  (interface_ptr -> nx_interface_physical_address_lsw << 16);

            if(driver_req_ptr -> nx_ip_driver_command == NX_LINK_ARP_SEND)
                *(ethernet_frame_ptr+3) |= NX_ETHERNET_ARP;
            else if(driver_req_ptr -> nx_ip_driver_command == NX_LINK_ARP_RESPONSE_SEND)
                *(ethernet_frame_ptr+3) |= NX_ETHERNET_ARP;
            else if(driver_req_ptr -> nx_ip_driver_command == NX_LINK_RARP_SEND)
                *(ethernet_frame_ptr+3) |= NX_ETHERNET_RARP;
            else if(packet_ptr -> nx_packet_ip_version == 4)
                *(ethernet_frame_ptr+3) |= NX_ETHERNET_IP;
            else
                *(ethernet_frame_ptr+3) |= NX_ETHERNET_IPV6;


            /* Endian swapping if NX_LITTLE_ENDIAN is defined.  */
```

```
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+1));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+2));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+3));
#ifdef NX_DEBUG_PACKET
            printf("NetX Duo RAM Driver Packet Send - %s\n", ip_ptr -> nx_ip_name);
#endif

            /* At this point, the packet is a complete Ethernet frame, ready to be transmitted.
               The driver shall call the actual Ethernet transmit routine and put the packet
               on the wire.

               In this example, the simulated RAM network transmit routine is called. */
            _nx_ram_network_driver_output(ip_ptr, packet_ptr, i);
            break;
        }


        case NX_LINK_MULTICAST_JOIN:
        {
            UINT          mcast_index;

            /* The IP layer issues this command to join a multicast group.  Note that
               multicast opertion is required for IPv6.

               On a typically Ethernet controller, the driver computes a hash value based
               on MAC address, and programs the hash table.

               It is likely the driver also needs to maintain an internal MAC address table.
               Later if a multicast address is removed, the driver needs
               to reprogram the hash table based on the remaining multicast MAC addresses. */


            /* The following procedure only applies to our simulated RAM network driver, which manages
               multicast MAC addresses by a simple look up table. */
            for(mcast_index = 0; mcast_index < NX_RAM_DRIVER_MAX_MCAST_ADDRESSES; mcast_index++)
            {
                if(nx_ram_driver[i].nx_ram_driver_mcast_address[mcast_index].nx_mac_address_msw == 0 &&
                   nx_ram_driver[i].nx_ram_driver_mcast_address[mcast_index].nx_mac_address_lsw == 0 )
                {
                    nx_ram_driver[i].nx_ram_driver_mcast_address[mcast_index].nx_mac_address_msw = driver_req_ptr ->
                             nx_ip_driver_physical_address_msw;
                    nx_ram_driver[i].nx_ram_driver_mcast_address[mcast_index].nx_mac_address_lsw = driver_req_ptr ->
                             nx_ip_driver_physical_address_lsw;
                    break;
                }
            }
            if(mcast_index == NX_RAM_DRIVER_MAX_MCAST_ADDRESSES)
                driver_req_ptr -> nx_ip_driver_status =  NX_NO_MORE_ENTRIES;

            break;
        }


        case NX_LINK_MULTICAST_LEAVE:
        {

            UINT  mcast_index;

            /* The IP layer issues this command to remove a multicast MAC address from the
               receiving list.  A device driver shall properly remove the multicast address
               from the hash table, so the hardware does not receive such traffic.  Note that
               in order to reprogram the hash table, the device driver may have to keep track of
               current active multicast MAC addresses. */

            /* The following procedure only applies to our simulated RAM network driver, which manages
               multicast MAC addresses by a simple look up table. */
            for(mcast_index = 0; mcast_index < NX_RAM_DRIVER_MAX_MCAST_ADDRESSES; mcast_index++)
            {
                if(nx_ram_driver[i].nx_ram_driver_mcast_address[mcast_index].nx_mac_address_msw ==
                             driver_req_ptr ->
                             nx_ip_driver_physical_address_msw &&
```

```
                nx_ram_driver[i].nx_ram_driver_mcast_address[mcast_index].nx_mac_address_lsw ==
                          driver_req_ptr -> nx_ip_driver_physical_address_lsw)
            {
                nx_ram_driver[i].nx_ram_driver_mcast_address[mcast_index].nx_mac_address_msw = 0;
                nx_ram_driver[i].nx_ram_driver_mcast_address[mcast_index].nx_mac_address_lsw = 0;
                break;
            }
        }
        if(mcast_index == NX_RAM_DRIVER_MAX_MCAST_ADDRESSES)
            driver_req_ptr -> nx_ip_driver_status =  NX_ENTRY_NOT_FOUND;

        break;
    }

    case NX_LINK_GET_STATUS:
    {

        /* Return the link status in the supplied return pointer.  */
        *(driver_req_ptr -> nx_ip_driver_return_ptr) =  ip_ptr-> nx_ip_interface[0].nx_interface_link_up;
        break;
    }

    case NX_LINK_DEFERRED_PROCESSING:
    {

        /* Driver defined deferred processing. This is typically used to defer interrupt
           processing to the thread level.

           A typical use case of this command is:
           On receiving an Ethernet frame, the RX ISR does not process the received frame,
           but instead records such an event in its internal data structure, and issues
           a notification to the IP stack (the driver sends the notification to the IP
           helping thread by calling "_nx_ip_driver_deferred_processing()".  When the IP stack
           gets a notification of a pending driver deferred process, it calls the
           driver with the NX_LINK_DEFERRED_PROCESSING command.  The driver shall complete
           the pending receive process.
        */

        /* The simulated RAM driver doesn't require a deferred process so it breaks out of
           the switch case. */


        break;
    }

    default:
    {

        /* Invalid driver request.  */

        /* Return the unhandled command status.  */
        driver_req_ptr -> nx_ip_driver_status =  NX_UNHANDLED_COMMAND;

#ifdef NX_DEBUG
        printf("NetX Duo RAM Driver Received invalid request - %s\n", ip_ptr -> nx_ip_name);
#endif
    }
  }
}


/**************************************************************************/
/*                                                                        */
/*  FUNCTION                                               RELEASE        */
/*                                                                        */
/*    _nx_ram_network_driver_output                       PORTABLE C      */
/*                                                                        */
/*  AUTHOR                                                                */
/*                                                                        */
/*    Express Logic, Inc.                                                 */
/*                                                                        */
/*  DESCRIPTION                                                           */
/*                                                                        */
```

```
/*    This function simply sends the packet to the IP instance on the     */
/*    created IP list that matches the physical destination specified in  */
/*    the Ethernet packet.  In a real hardware setting, this routine      */
/*    would simply put the packet out on the wire.                        */
/*                                                                        */
/*  INPUT                                                                 */
/*                                                                        */
/*    ip_ptr                              Pointer to IP protocol block    */
/*    packet_ptr                          Packet pointer                  */
/*                                                                        */
/*  OUTPUT                                                                */
/*                                                                        */
/*    None                                                                */
/*                                                                        */
/*  CALLS                                                                 */
/*                                                                        */
/*    nx_packet_copy                      Copy a packet                   */
/*    nx_packet_transmit_release          Release a packet                */
/*    _nx_ram_network_driver_receive      RAM driver receive processing */
/*                                                                        */
/*  CALLED BY                                                             */
/*                                                                        */
/*    NetX Duo IP processing                                              */
/*                                                                        */
/*                                                                        */
/**************************************************************************/
void  _nx_ram_network_driver_output(NX_IP *ip_ptr, NX_PACKET *packet_ptr, UINT device_instance_id)
{

NX_IP        *next_ip;
NX_PACKET    *packet_copy;
ULONG         destination_address_msw;
ULONG         destination_address_lsw;
UINT          old_threshold;
UINT          i;
UINT          mcast_index;

#ifdef NX_DEBUG_PACKET
UCHAR       *ptr;
UINT        j;

    ptr =  packet_ptr -> nx_packet_prepend_ptr;
    printf("Ethernet Packet: ");
    for (j = 0; j < 6; j++)
        printf("%02X", *ptr++);
    printf(" ");
    for (j = 0; j < 6; j++)
        printf("%02X", *ptr++);
    printf(" %02X", *ptr++);
    printf("%02X ", *ptr++);

    i = 0;
    for (j = 0; j < (packet_ptr -> nx_packet_length - NX_ETHERNET_SIZE); j++)
    {
        printf("%02X", *ptr++);
        i++;
        if (i > 3)
        {
            i = 0;
            printf(" ");
        }
    }
    printf("\n");


#endif

    /* Pickup the destination IP address from the packet_ptr.  */
    destination_address_msw =  (ULONG) *(packet_ptr -> nx_packet_prepend_ptr);
```

```
destination_address_msw =  (destination_address_msw << 8) | (ULONG) *(packet_ptr -> nx_packet_prepend_ptr+1);
destination_address_lsw =  (ULONG) *(packet_ptr -> nx_packet_prepend_ptr+2);
destination_address_lsw =  (destination_address_lsw << 8) | (ULONG) *(packet_ptr -> nx_packet_prepend_ptr+3);
destination_address_lsw =  (destination_address_lsw << 8) | (ULONG) *(packet_ptr -> nx_packet_prepend_ptr+4);
destination_address_lsw =  (destination_address_lsw << 8) | (ULONG) *(packet_ptr -> nx_packet_prepend_ptr+5);


/* Disable preemption.  */
tx_thread_preemption_change(tx_thread_identify(), 0, &old_threshold);

/* Loop through all instances of created IPs to see who gets the packet.  */
next_ip =  ip_ptr -> nx_ip_created_next;

for(i = 0; i < NX_MAX_RAM_INTERFACES; i++)
{

    /* Skip the interface from which the packet was sent. */
    if(i == device_instance_id)
        continue;

    /* Skip the instance that has not been initialized. */
    if(nx_ram_driver[i].nx_ram_network_driver_in_use == 0)
        continue;


    /* If the destination MAC address is broadcast or the destination matches the interface MAC,
       accept the packet. */
    if(((destination_address_msw == ((ULONG) 0x0000FFFF)) && (destination_address_lsw == ((ULONG)
                      0xFFFFFFFF))) || /* Broadcast match */
       ((destination_address_msw == nx_ram_driver[i].nx_ram_driver_mac_address.nx_mac_address_msw) &&
        (destination_address_lsw == nx_ram_driver[i].nx_ram_driver_mac_address.nx_mac_address_lsw)))
    {

        /* Make a copy of packet for the forwarding.  */
        if (nx_packet_copy(packet_ptr, &packet_copy, next_ip -> nx_ip_default_packet_pool, NX_NO_WAIT))
        {

            /* Remove the Ethernet header.  */
            packet_ptr -> nx_packet_prepend_ptr =  packet_ptr -> nx_packet_prepend_ptr + NX_ETHERNET_SIZE;

            /* Adjust the packet length.  */
            packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length - NX_ETHERNET_SIZE;

            /* Error, no point in continuing, just release the packet.  */
            nx_packet_transmit_release(packet_ptr);
            return;
        }

        _nx_ram_network_driver_receive(next_ip, packet_copy, i);
    }
    else
    {
        for(mcast_index = 0; mcast_index < NX_RAM_DRIVER_MAX_MCAST_ADDRESSES; mcast_index++)
        {

            if(destination_address_msw ==
                         nx_ram_driver[i].nx_ram_driver_mcast_address[mcast_index].nx_mac_address_msw &&
               destination_address_lsw ==
                         nx_ram_driver[i].nx_ram_driver_mcast_address[mcast_index].nx_mac_address_lsw)
            {

                /* Make a copy of packet for the forwarding.  */
                if (nx_packet_copy(packet_ptr, &packet_copy, next_ip -> nx_ip_default_packet_pool,
                         NX_NO_WAIT))
                {

                    /* Remove the Ethernet header.  */
                    packet_ptr -> nx_packet_prepend_ptr =  packet_ptr -> nx_packet_prepend_ptr +
                         NX_ETHERNET_SIZE;
```

```
                        /* Adjust the packet length.  */
                        packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length - NX_ETHERNET_SIZE;

                        /* Error, no point in continuing, just release the packet.  */
                        nx_packet_transmit_release(packet_ptr);
                        return;
                    }

                    _nx_ram_network_driver_receive(next_ip, packet_copy, i);


                }


            }
        }
    }

    /* Remove the Ethernet header.  In real hardware environments, this is typically
       done after a transmit complete interrupt.  */
    packet_ptr -> nx_packet_prepend_ptr =  packet_ptr -> nx_packet_prepend_ptr + NX_ETHERNET_SIZE;

    /* Adjust the packet length.  */
    packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length - NX_ETHERNET_SIZE;

    /* Now that the Ethernet frame has been removed, release the packet.  */
    nx_packet_transmit_release(packet_ptr);

    /* Restore preemption.  */
    tx_thread_preemption_change(tx_thread_identify(), old_threshold, &old_threshold);
}


/**************************************************************************/
/*                                                                        */
/*  FUNCTION                                               RELEASE        */
/*                                                                        */
/*    _nx_ram_network_driver_receive                      PORTABLE C      */
/*                                                                        */
/*  AUTHOR                                                                */
/*    Express Logic, Inc.                                                 */
/*                                                                        */
/*  DESCRIPTION                                                           */
/*                                                                        */
/*    This function processing incoming packets.  In the RAM network      */
/*    driver, the incoming packets are coming from the RAM driver output  */
/*    routine.  In real hardware settings, this routine would be called   */
/*    from the receive packet ISR.                                        */
/*                                                                        */
/*  INPUT                                                                 */
/*                                                                        */
/*    ip_ptr                                Pointer to IP protocol block  */
/*    packet_ptr                            Packet pointer                */
/*    device_instance_id                    The device ID the packet is   */
/*                                            destined for                */
/*                                                                        */
/*  OUTPUT                                                                */
/*                                                                        */
/*    None                                                                */
/*                                                                        */
/*  CALLS                                                                 */
/*                                                                        */
/*    _nx_ip_packet_receive                 IP receive packet processing  */
/*    _nx_ip_packet_deferred_receive        IP deferred receive packet    */
/*                                            processing                  */
/*    _nx_arp_packet_deferred_receive       ARP receive processing        */
/*    _nx_rarp_packet_deferred_receive      RARP receive processing       */
/*    nx_packet_release                     Packet release                */
/*                                                                        */
/*  CALLED BY                                                             */
```

```
/*                                                                          */
/*    NetX Duo IP processing                                                */
/*                                                                          */
/*                                                                          */
/**************************************************************************/
void  _nx_ram_network_driver_receive(NX_IP *ip_ptr, NX_PACKET *packet_ptr, UINT device_instance_id)
{

UINT    packet_type;

    /* Pickup the packet header to determine where the packet needs to be
       sent.  */
    packet_type =  (((UINT) (*(packet_ptr -> nx_packet_prepend_ptr+12))) << 8) |
                    ((UINT) (*(packet_ptr -> nx_packet_prepend_ptr+13)));


    /* Setup interface pointer.  */
    packet_ptr -> nx_packet_ip_interface =
nx_ram_driver[device_instance_id].nx_ram_driver_interface_ptr;


    /* Route the incoming packet according to its ethernet type.  */
    /* The RAM driver accepts both IPv4 and IPv6 frames. */
    if ((packet_type == NX_ETHERNET_IP) || (packet_type == NX_ETHERNET_IPV6))
    {

        /* Note:  The length reported by some Ethernet hardware includes bytes after the packet
           as well as the Ethernet header.  In some cases, the actual packet length after the
           Ethernet header should be derived from the length in the IP header (lower 16 bits of
           the first 32-bit word).  */

        /* Clean off the Ethernet header.  */
        packet_ptr -> nx_packet_prepend_ptr =  packet_ptr -> nx_packet_prepend_ptr +
NX_ETHERNET_SIZE;

        /* Adjust the packet length.  */
        packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length - NX_ETHERNET_SIZE;

        /* Route to the ip receive function.  */
#ifdef NX_DEBUG_PACKET
        printf("NetX Duo RAM Driver IP Packet Receive - %s\n", ip_ptr -> nx_ip_name);
#endif

#ifdef NX_DIRECT_ISR_CALL
        _nx_ip_packet_receive(ip_ptr, packet_ptr);
#else
        _nx_ip_packet_deferred_receive(ip_ptr, packet_ptr);
#endif
    }
    else if (packet_type == NX_ETHERNET_ARP)
    {

        /* Clean off the Ethernet header.  */
        packet_ptr -> nx_packet_prepend_ptr =  packet_ptr -> nx_packet_prepend_ptr +
NX_ETHERNET_SIZE;

        /* Adjust the packet length.  */
        packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length - NX_ETHERNET_SIZE;

        /* Route to the ARP receive function.  */
#ifdef NX_DEBUG
        printf("NetX Duo RAM Driver ARP Receive - %s\n", ip_ptr -> nx_ip_name);
#endif
        _nx_arp_packet_deferred_receive(ip_ptr, packet_ptr);

    }
    else if (packet_type == NX_ETHERNET_RARP)
    {

        /* Clean off the Ethernet header.  */
```

```
        packet_ptr -> nx_packet_prepend_ptr =  packet_ptr -> nx_packet_prepend_ptr + NX_ETHERNET_SIZE;

        /* Adjust the packet length.  */
        packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length - NX_ETHERNET_SIZE;

        /* Route to the RARP receive function.  */
#ifdef NX_DEBUG
        printf("NetX Duo RAM Driver RARP Receive - %s\n", ip_ptr -> nx_ip_name);
#endif
        _nx_rarp_packet_deferred_receive(ip_ptr, packet_ptr);
    }
    else
    {

        /* Invalid ethernet header... release the packet.  */
        nx_packet_release(packet_ptr);
    }
}
```

NetX Services

Address Resolution Protocol (ARP)

```
UINT    nx_arp_dynamic_entries_invalidate(NX_IP *ip_ptr);

UINT    nx_arp_dynamic_entry_set(NX_IP *ip_ptr, ULONG
           ip_address, ULONG physical_msw, ULONG physical_lsw);

UINT    nx_arp_enable(NX_IP *ip_ptr, VOID *arp_cache_memory,
           ULONG arp_cache_size);

UINT    nx_arp_gratuitous_send(NX_IP *ip_ptr,
           VOID (*response_handler)(NX_IP *ip_ptr,
           NX_PACKET *packet_ptr));

UINT    nx_arp_hardware_address_find(NX_IP *ip_ptr,
           ULONG ip_address, ULONG*physical_msw,
           ULONG *physical_lsw);

UINT    nx_arp_info_get(NX_IP *ip_ptr, ULONG
           *arp_requests_sent, ULONG*arp_requests_received,
           ULONG *arp_responses_sent,
           ULONG*arp_responses_received,
           ULONG *arp_dynamic_entries,
           ULONG *arp_static_entries,
           ULONG *arp_aged_entries,
           ULONG *arp_invalid_messages);

UINT    nx_arp_ip_address_find(NX_IP *ip_ptr,
           ULONG *ip_address, ULONG physical_msw,
           ULONG physical_lsw);

UINT    nx_arp_static_entries_delete(NX_IP *ip_ptr);

UINT    nx_arp_static_entry_create(NX_IP *ip_ptr,
           ULONG ip_address,
           ULONG physical_msw, ULONG physical_lsw);

UINT    nx_arp_static_entry_delete(NX_IP *ip_ptr,
           ULONG ip_address, ULONG physical_msw,
           ULONG physical_lsw);
```

Internet Control Message Protocol (ICMP)

```
UINT    nx_icmp_enable(NX_IP *ip_ptr);

UINT    nx_icmp_info_get(NX_IP *ip_ptr, ULONG *pings_sent,
           ULONG *ping_timeouts, ULONG *ping_threads_suspended,
           ULONG *ping_responses_received,
           ULONG *icmp_checksum_errors,
           ULONG *icmp_unhandled_messages);

UINT    nx_icmp_ping(NX_IP *ip_ptr,
           ULONG ip_address, CHAR *data,
           ULONG data_size, NX_PACKET **response_ptr,
           ULONG wait_option);

UINT    nxd_icmp_enable(NX_IP *ip_ptr)

UINT    nxd_icmp_ping(NX_IP *ip_ptr, NXD_ADDRESS *ip_address,
           CHAR *data_ptr, ULONG data_size, NX_PACKET
           **response_ptr, ULONG wait_option)
```

```
UINT      nxd_icmp_interface_ping(NX_IP *ip_ptr, NXD_ADDRESS
             *ip_address, UINT source_index, CHAR *data_ptr,
             ULONG data_size, NX_PACKET **response_ptr, ULONG
             wait_option);
```

Internet Group Management Protocol (IGMP)

```
UINT      nx_igmp_enable(NX_IP *ip_ptr);

UINT      nx_igmp_info_get(NX_IP *ip_ptr, ULONG
             *igmp_reports_sent, ULONG *igmp_queries_received,
             ULONG *igmp_checksum_errors,
             ULONG *current_groups_joined);

UINT      nx_igmp_loopback_disable(NX_IP *ip_ptr);

UINT      nx_igmp_loopback_enable(NX_IP *ip_ptr);

UINT      nx_igmp_multicast_interface_join(NX_IP *ip_ptr,
             ULONG group_address, UINT interface_index);

UINT      nx_igmp_multicast_join(NX_IP *ip_ptr,
             ULONG group_address);

UINT      nx_igmp_multicast_leave(NX_IP *ip_ptr,
             ULONG group_address);
```

Internet Protocol (IP)

```
UINT      nx_ip_address_change_notify(NX_IP *ip_ptr,
             VOID (*change_notify)(NX_IP *, VOID *),
             VOID *additional_info);

UINT      nx_ip_address_get(NX_IP *ip_ptr, ULONG *ip_address,
             ULONG *network_mask);

UINT      nx_ip_address_set(NX_IP *ip_ptr, ULONG ip_address,
             ULONG network_mask);

UINT      nx_ip_create(NX_IP *ip_ptr, CHAR *name,
             ULONG ip_address,
             ULONG network_mask, NX_PACKET_POOL *default_pool,
             VOID (*ip_network_driver)(NX_IP_DRIVER *),
             VOID *memory_ptr, ULONG memory_size, UINT priority);

UINT      nx_ip_delete(NX_IP *ip_ptr);

UINT      nx_ip_driver_direct_command(NX_IP *ip_ptr, UINT
             command, ULONG *return_value_ptr);

UINT      nx_ip_driver_interface_direct_command(NX_IP *ip_ptr,
             UINT command, UINT interface_index, ULONG
             *return_value_ptr);

UINT      nx_ip_forwarding_disable(NX_IP *ip_ptr);

UINT      nx_ip_forwarding_enable(NX_IP *ip_ptr);

UINT      nx_ip_fragment_disable(NX_IP *ip_ptr);
```

```
UINT      nx_ip_fragment_enable(NX_IP *ip_ptr);

UINT      nx_ip_gateway_address_set(NX_IP *ip_ptr,
             ULONG ip_address);

UINT      nx_ip_info_get(NX_IP *ip_ptr,
             ULONG *ip_total_packets_sent,
             ULONG *ip_total_bytes_sent,
             ULONG *ip_total_packets_received,
             ULONG *ip_total_bytes_received,
             ULONG *ip_invalid_packets,
             ULONG *ip_receive_packets_dropped,
             ULONG *ip_receive_checksum_errors,
             ULONG *ip_send_packets_dropped,
             ULONG *ip_total_fragments_sent,
             ULONG *ip_total_fragments_received);

UINT      nx_ip_interface_address_get(NX_IP *ip_ptr,
             ULONG interface_index,
             ULONG *ip_address,
             ULONG *network_mask);

UINT      nx_ip_interface_address_set(NX_IP *ip_ptr,
             ULONG interface_index, ULONG ip_address, ULONG network_mask);

UINT      nx_ip_interface_attach(NX_IP *ip_ptr, CHAR* interface_name,
             ULONG ip_address, ULONG network_mask,
             VOID (*ip_link_driver)(struct NX_IP_DRIVER_STRUCT *));

UINT      nx_ip_interface_info_get(NX_IP *ip_ptr, UINT interface_index,
             CHAR **interface_name, ULONG *ip_address,
             ULONG *network_mask, ULONG *mtu_size,
             ULONG *phsyical_address_msw, ULONG *physical_address_lsw);

UINT      nx_ip_interface_status_check(NX_IP *ip_ptr,
             UINT interface_index, ULONG needed_status,
             ULONG *actual_status, ULONG wait_option);

UINT      _nx_ip_max_payload_size_find(NX_IP *ip_ptr,
             NXD_ADDRESS *dest_address, UINT if_index,
             UINT src_port,
             UINT dest_port, ULONG protocol,
             ULONG *start_offset_ptr,
             ULONG *payload_length_ptr)

UINT      nx_ip_raw_packet_disable(NX_IP *ip_ptr);

UINT      nx_ip_raw_packet_enable(NX_IP *ip_ptr);

UINT      nx_ip_raw_packet_interface_send(NX_IP *ip_ptr,
             NX_PACKET *packet_ptr, ULONG destination_ip,
             UINT interface_index, ULONG type_of_service);

UINT      nx_ip_raw_packet_receive(NX_IP *ip_ptr,
             NX_PACKET **packet_ptr,
             ULONG wait_option);

UINT      nx_ip_raw_packet_send(NX_IP *ip_ptr,
             NX_PACKET *packet_ptr,
             ULONG destination_ip, ULONG type_of_service);
```

```
UINT    nx_ip_static_route_add(NX_IP *ip_ptr, ULONG network_address,
           ULONG net_mask, ULONG next_hop);

UINT    nx_ip_static_route_delete(NX_IP *ip_ptr, ULONG
           network_address, ULONG net_mask);

UINT    nx_ip_status_check(NX_IP *ip_ptr, ULONG needed_status, ULONG
           *actual_status, ULONG wait_option);

UINT    nxd_ipv6_default_router_add(NX_IP *ip_ptr, NXD_ADDRESS
           *router_address, ULONG router_lifetime, UINT if_index)

UINT    nxd_ipv6_default_router_delete(NX_IP *ip_ptr, NXD_ADDRESS
           *router_address)

UINT    nxd_ipv6_default_router_get(NX_IP *ip_ptr, UINT if_index,
           NXD_ADDRESS *router_address, ULONG *router_lifetime, ULONG
           *prefix_length)

UINT    nxd_ipv6_enable(NX_IP *ip_ptr)

UINT    nxd_ip_raw_packet_send(NX_IP *ip_ptr, NX_PACKET *packet_ptr,
           NXD_ADDRESS *destination_ip, ULONG protocol)

UINT    nxd_ip_raw_packet_interface_send(NX_IP *ip_ptr, NX_PACKET
           *packet_ptr, NXD_ADDRESS *destination_ip, UINT if_index,
           ULONG protocol);

UINT    nxd_ipv6_address_delete(NX_IP *ip_ptr, UINT address_index);

UINT    nxd_ipv6_address_get(NX_IP *ip_ptr, UINT address_index,
           NXD_ADDRESS *ip_address, ULONG *prefix_length, UINT
           *if_index);

UINT    nxd_ipv6_address_set(UINT nxd_ipv6_address_set(NX_IP *ip_ptr,
           UINT address_index);
```

Neighbor Discovery

```
UINT    nxd_nd_cache_entry_delete(NX_IP ip_ptr, ULONG *ip_address)

UINT    nxd_nd_cache_entry_set(NX_IP *ip_ptr, ULONG *ip_address, char
           *mac)

UINT    nxd_nd_cache_hardware_address_find(NX_IP *ip_ptr, NXD_ADDRESS
           *ip_address, ULONG *physical_msw,
           ULONG *physical_lsw)

UINT    nxd_nd_cache_invalidate(NX_IP *ip_ptr)

UINT    nxd_nd_cache_ip_address_find(NX_IP *ip_ptr, NXD_ADDRESS
           *ip_address,ULONG physical_msw, ULONG physical_lsw, UINT
           *if_index)
```

Packet Management

```
UINT    nx_packet_allocate(NX_PACKET_POOL *pool_ptr,
           NX_PACKET **packet_ptr, ULONG packet_type,
           ULONG wait_option);

UINT    nx_packet_copy(NX_PACKET *packet_ptr,
           NX_PACKET **new_packet_ptr, NX_PACKET_POOL
           *pool_ptr,
           ULONG wait_option);

UINT    nx_packet_data_append(NX_PACKET *packet_ptr,
           VOID *data_start, ULONG data_size,
           NX_PACKET_POOL *pool_ptr, ULONG wait_option);

UINT    nx_packet_data_extract_offset(NX_PACKET *packet_ptr,
           ULONG offset, VOID *buffer_start, ULONG
           buffer_length, ULONG *bytes_copied);

UINT    nx_packet_data_retrieve(NX_PACKET *packet_ptr,
           VOID *buffer_start, ULONG *bytes_copied);

UINT    nx_packet_length_get(NX_PACKET *packet_ptr, ULONG
           *length);

UINT    nx_packet_pool_create(NX_PACKET_POOL *pool_ptr,
           CHAR *name, ULONG block_size, VOID *memory_ptr,
           ULONG memory_size);

UINT    nx_packet_pool_delete(NX_PACKET_POOL *pool_ptr);

UINT    nx_packet_pool_info_get(NX_PACKET_POOL *pool_ptr, ULONG
           *total_packets, ULONG *free_packets,
           ULONG *empty_pool_requests,
           ULONG *empty_pool_suspensions,
           ULONG *invalid_packet_releases);

UINT    nx_packet_release(NX_PACKET *packet_ptr);

UINT    nx_packet_transmit_release(NX_PACKET *packet_ptr);
```

Reverse Address Resolution Protocol (RARP)

```
UINT    nx_rarp_disable(NX_IP *ip_ptr);

UINT    nx_rarp_enable(NX_IP *ip_ptr);

UINT    nx_rarp_info_get(NX_IP *ip_ptr,
           ULONG *rarp_requests_sent,
           ULONG *rarp_responses_received,
           ULONG *rarp_invalid_messages);
```

System Management

```
VOID    nx_system_initialize(VOID);
```

Transmission Control Protocol (TCP)

```
UINT      nx_tcp_client_socket_bind(NX_TCP_SOCKET *socket_ptr,
             UINT port, ULONG wait_option);

UINT      nx_tcp_client_socket_connect(NX_TCP_SOCKET
             *socket_ptr, ULONG server_ip, UINT server_port,
             ULONG wait_option);

UINT      nx_tcp_client_socket_port_get(NX_TCP_SOCKET
             *socket_ptr, UINT *port_ptr);

UINT      nx_tcp_client_socket_unbind(NX_TCP_SOCKET
             *socket_ptr);

UINT      nx_tcp_enable(NX_IP *ip_ptr);

UINT      nx_tcp_free_port_find(NX_IP *ip_ptr, UINT port,
             UINT *free_port_ptr);

UINT      nx_tcp_info_get(NX_IP *ip_ptr, ULONG *tcp_packets_sent,
             ULONG *tcp_bytes_sent, ULONG *tcp_packets_received,
             ULONG *tcp_bytes_received, ULONG
             *tcp_invalid_packets, ULONG
             *tcp_receive_packets_dropped,
             ULONG *tcp_checksum_errors,ULONG *tcp_connections,
             ULONG *tcp_disconnections,
             ULONG *tcp_connections_dropped,
             ULONG*tcp_retransmit_packets);

UINT      nx_tcp_server_socket_accept(NX_TCP_SOCKET *socket_ptr,
             ULONG wait_option);

UINT      nx_tcp_server_socket_listen(NX_IP *ip_ptr,
             UINT port, NX_TCP_SOCKET *socket_ptr,
             UINT listen_queue_size,
             VOID (*tcp_listen_callback)(NX_TCP_SOCKET
             *socket_ptr, UINT port));

UINT      nx_tcp_server_socket_relisten(NX_IP *ip_ptr,
             UINT port, NX_TCP_SOCKET *socket_ptr);

UINT      nx_tcp_server_socket_unaccept(NX_TCP_SOCKET
             *socket_ptr);

UINT      nx_tcp_server_socket_unlisten(NX_IP *ip_ptr, UINT
             port);

UINT      nx_tcp_socket_bytes_available(NX_TCP_SOCKET
             *socket_ptr, ULONG *bytes_available);

UINT      nx_tcp_socket_create(NX_IP *ip_ptr,
             NX_TCP_SOCKET *socket_ptr, CHAR *name,
             ULONG type_of_service, ULONG fragment,
             UINT time_to_live, ULONG window_size,
             VOID (*tcp_urgent_data_callback)(NX_TCP_SOCKET
             *socket_ptr),
             VOID (*tcp_disconnect_callback)(NX_TCP_SOCKET
             *socket_ptr));

UINT      nx_tcp_socket_delete(NX_TCP_SOCKET *socket_ptr);
```

```
UINT      nx_tcp_socket_disconnect(NX_TCP_SOCKET *socket_ptr,
             ULONG wait_option);

UINT      nx_tcp_socket_info_get(NX_TCP_SOCKET *socket_ptr,
             ULONG *tcp_packets_sent, ULONG *tcp_bytes_sent,
             ULONG *tcp_packets_received, ULONG
             *tcp_bytes_received,
             ULONG *tcp_retransmit_packets, ULONG
             *tcp_packets_queued,
             ULONG *tcp_checksum_errors, ULONG *tcp_socket_state,
             ULONG *tcp_transmit_queue_depth, ULONG
             *tcp_transmit_window,
             ULONG *tcp_receive_window);

UINT      nx_tcp_socket_mss_get(NX_TCP_SOCKET *socket_ptr,
             ULONG *mss);

UINT      nx_tcp_socket_mss_peer_get(NX_TCP_SOCKET *socket_ptr,
             ULONG *peer_mss);

UINT      nx_tcp_socket_mss_set(NX_TCP_SOCKET *socket_ptr,
             ULONG mss);

UINT      nx_tcp_socket_peer_info_get(NX_TCP_SOCKET *socket_ptr,
             ULONG *peer_ip_address, ULONG *peer_port);

UINT      nx_tcp_socket_receive(NX_TCP_SOCKET *socket_ptr,
             NX_PACKET **packet_ptr, ULONG wait_option);

UINT      nx_tcp_socket_receive_notify(NX_TCP_SOCKET
             *socket_ptr, VOID
             (*tcp_receive_notify)(NX_TCP_SOCKET *socket_ptr));

UINT      nx_tcp_socket_send(NX_TCP_SOCKET *socket_ptr,
             NX_PACKET *packet_ptr, ULONG wait_option);

UINT      nx_tcp_socket_state_wait(NX_TCP_SOCKET *socket_ptr,
             UINT desired_state, ULONG wait_option);

UINT      nx_tcp_socket_transmit_configure(NX_TCP_SOCKET
             *socket_ptr, ULONG max_queue_depth, ULONG timeout,
             ULONG max_retries, ULONG timeout_shift);

UINT      nx_tcp_socket_window_update_notify_set
             (NX_TCP_SOCKET *socket_ptr,
             VOID (*tcp_window_update_notify)
             (NX_TCP_SOCKET *socket_ptr));

UINT      nxd_tcp_client_socket_connect(NX_TCP_SOCKET
             *socket_ptr, NXD_ADDRESS *server_ip,  UINT
             server_port, ULONG wait_option)

UINT      nxd_tcp_socket_peer_info_get(NX_TCP_SOCKET
             *socket_ptr, NXD_ADDRESS *peer_ip_address,
             ULONG *peer_port)
```

User Datagram Protocol (UDP)

```
UINT    nx_udp_enable(NX_IP *ip_ptr);

UINT    nx_udp_free_port_find(NX_IP *ip_ptr, UINT port,
           UINT *free_port_ptr);

UINT    nx_udp_info_get(NX_IP *ip_ptr, ULONG *udp_packets_sent,
           ULONG *udp_bytes_sent, ULONG *udp_packets_received,
           ULONG *udp_bytes_received,
           ULONG *udp_invalid_packets,
           ULONG *udp_receive_packets_dropped,
           ULONG *udp_checksum_errors);

UINT    nx_udp_packet_info_extract(NX_PACKET *packet_ptr,
           ULONG *ip_address, UINT *protocol, UINT *port,
           UINT *interface_index);

UINT    nx_udp_socket_bind(NX_UDP_SOCKET *socket_ptr,
           UINT  port, ULONG wait_option);

UINT    nx_udp_socket_bytes_available(NX_UDP_SOCKET
           *socket_ptr, ULONG *bytes_available);

UINT    nx_udp_socket_checksum_disable(NX_UDP_SOCKET
           *socket_ptr);

UINT    nx_udp_socket_checksum_enable(NX_UDP_SOCKET
           *socket_ptr);

UINT    nx_udp_socket_create(NX_IP *ip_ptr, NX_UDP_SOCKET
           *socket_ptr, CHAR *name, ULONG type_of_service,
           ULONG fragment,
           UINT time_to_live, ULONG queue_maximum);

UINT    nx_udp_socket_delete(NX_UDP_SOCKET *socket_ptr);

UINT    nx_udp_socket_info_get(NX_UDP_SOCKET *socket_ptr,
           ULONG *udp_packets_sent, ULONG *udp_bytes_sent,
           ULONG *udp_packets_received, ULONG
           *udp_bytes_received,
           ULONG *udp_packets_queued,
           ULONG *udp_receive_packets_dropped,
           ULONG *udp_checksum_errors);

UINT    nx_udp_socket_interface_send(NX_UDP_SOCKET
           *socket_ptr, NX_PACKET *packet_ptr, ULONG
           ip_address, UINT port, UINT address_index);

UINT    nx_udp_socket_port_get(NX_UDP_SOCKET *socket_ptr,
           UINT *port_ptr);

UINT    nx_udp_socket_receive(NX_UDP_SOCKET *socket_ptr,
           NX_PACKET **packet_ptr, ULONG wait_option);

UINT    nx_udp_socket_receive_notify(NX_UDP_SOCKET
           *socket_ptr, VOID
           (*udp_receive_notify)(NX_UDP_SOCKET *socket_ptr));

UINT    nx_udp_socket_send(NX_UDP_SOCKET *socket_ptr,
           NX_PACKET *packet_ptr, ULONG ip_address, UINT port);

UINT    nx_udp_socket_unbind(NX_UDP_SOCKET *socket_ptr);
```

```
UINT     nx_udp_source_extract(NX_PACKET *packet_ptr,
            ULONG *ip_address, UINT *port);

UINT     nxd_udp_packet_info_extract(NX_PACKET *packet_ptr,
            NXD_ADDRESS *ip_address, UINT *protocol, UINT *port,
            UINT *interface_index);

UINT     nxd_udp_source_extract (NX_PACKET *packet_ptr,
            NXD_ADDRESS *ip_address, UINT *port)

UINT     nxd_udp_socket_interface_send(NX_UDP_SOCKET
            *socket_ptr, NX_PACKET *packet_ptr, NXD_ADDRESS
            *ip_address, UINT port, UINT address_index)

UINT     nxd_udp_socket_send(NX_UDP_SOCKET *socket_ptr,
            NX_PACKET *packet_ptr, NXD_ADDRESS *ip_address,
            UINT port)
```

NetX Constants

Alphabetic Listing

NX_ALL_HOSTS_ADDRESS	0xFE000001
NX_ALL_ROUTERS_ADDRESS	0xFE000002
NX_ALREADY_BOUND	0x22
NX_ALREADY_ENABLED	0x15
NX_ALREADY_RELEASED	0x31
NX_ALREADY_SUSPENDED	0x40
NX_ANY_PORT	0
NX_ARP_EXPIRATION_RATE	0
NX_ARP_HARDWARE_SIZE	0x06
NX_ARP_HARDWARE_TYPE	0x0001
NX_ARP_MAX_QUEUE_DEPTH	4
NX_ARP_MAXIMUM_RETRIES	18
NX_ARP_MESSAGE_SIZE	28
NX_ARP_OPTION_REQUEST	0x0001
NX_ARP_OPTION_RESPONSE	0x0002
NX_ARP_PROTOCOL_SIZE	0x04
NX_ARP_PROTOCOL_TYPE	0x0800
NX_ARP_TIMER_ERROR	0x18
NX_ARP_UPDATE_RATE	10
NX_ARP_TABLE_SIZE	0x2F
NX_ARP_TABLE_MASK	0x1F
NX_CALLER_ERROR	0x11
NX_CARRY_BIT	0x10000
NX_CONNECTION_PENDING	0x48
NX_DELETE_ERROR	0x10
NX_DELETED	0x05
NX_DISCONNECT_FAILED	0x41
NX_DONT_FRAGMENT	0x00004000
NX_DRIVER_TX_DONE	0xDDDDDDDD
NX_DUPLICATE_LISTEN	0x34
NX_ENTRY_NOT_FOUND	0x16
NX_FALSE	0
NX_FOREVER	1

NX_FRAG_OFFSET_MASK	0x00001FFF
NX_FRAGMENT_OKAY	0x00000000
NX_ICMP_ADDRESS_MASK_REP_TYPE	18
NX_ICMP_ADDRESS_MASK_REQ_TYPE	17
NX_ICMP_DEBUG_LOG_SIZE	100
NX_ICMP_DEST_UNREACHABLE_TYPE	3
NX_ICMP_ECHO_REPLY_TYPE	0
NX_ICMP_ECHO_REQUEST_TYPE	8
NX_ICMP_FRAMENT_NEEDED_CODE	4
NX_ICMP_HOST_PROHIBIT_CODE	10
NX_ICMP_HOST_SERVICE_CODE	12
NX_ICMP_HOST_UNKNOWN_CODE	7
NX_ICMP_HOST_UNREACH_CODE	1
NX_ICMP_NETWORK_PROHIBIT_CODE	9
NX_ICMP_NETWORK_SERVICE_CODE	11
NX_ICMP_NETWORK_UNKNOWN_CODE	6
NX_ICMP_NETWORK_UNREACH_CODE	0
NX_ICMP_PACKET (IPv6 enabled	56
NX_ICMP_PACKET (IPv6 disabled	36
NX_ICMP_PARAMETER_PROB_TYPE	12
NX_ICMP_PORT_UNREACH_CODE	3
NX_ICMP_PROTOCOL_UNREACH_CODE	2
NX_ICMP_REDIRECT_TYPE	5
NX_ICMP_SOURCE_ISOLATED_CODE	8
NX_ICMP_SOURCE_QUENCH_TYPE	4
NX_ICMP_SOURCE_ROUTE_CODE	5
NX_ICMP_TIME_EXCEEDED_TYPE	11
NX_ICMP_TIMESTAMP_REP_TYPE	14
NX_ICMP_TIMESTAMP_REQ_TYPE	13
NX_ICMPV6_ADDRESS_UNREACHABLE_CODE	3
NX_ICMPV6_BEYOND_SCOPE_OF_SOURCE_ADDRESS_CODE	2
NX_ICMPV6_COMMUNICATION_WITH_DESTINATION_PROHIBITED_CODE	1
NX_ICMPV6_DEST_UNREACHABLE_CODE	4
NX_ICMPV6_DEST_UNREACHABLE_TYPE	1
NX_ICMPV6_ECHO_REPLY_TYPE	129

NX_ICMPV6_ECHO_REQUEST_TYPE	128
NX_ICMPV6_MINIMUM_IPV4_PATH_MTU	576
NX_ICMPV6_MINIMUM_IPV6_PATH_MTU	1280
NX_ICMPV6_NEIGHBOR_ADVERTISEMENT_TYPE	136
NX_ICMPV6_NEIGHBOR_SOLICITATION_TYPE	135
NX_ICMPV6_NO_ROUTE_TO_DESTINATION_CODE	0
NX_ICMPV6_NO_SLLA	1
NX_ICMPV6_OPTION_TYPE_PREFIX_INFO	3
NX_ICMPV6_OPTION_REDIRECTED_HEADER	4
NX_ICMPV6_OPTION_TYPE_MTU	5
NX_ICMPV6_OPTION_TYPE_SRC_LINK_ADDR	1
NX_ICMPV6_OPTION_TYPE_TRG_LINK_ADDR	2
NX_ICMPV6_PACKET_TOO_BIG_TYPE	2
NX_ICMPV6_PARAMETER_PROBLEM_TYPE	4
NX_ICMPV6_PATH_MTU_INFINITE_TIMEOUT	0xFFFFFFFF
NX_ICMPV6_REDIRECT_MESSAGE_TYPE	137
NX_ICMPV6_REJECT_ROUTE_TO_DESTINATION_CODE	6
NX_ICMPV6_ROUTER_ADVERTISEMENT_TYPE	134
NX_ICMPV6_ROUTER_SOLICITATION_TYPE	133
NX_ICMPV6_SOURCE_ADDRESS_FAILED_I_E_POLICY_CODE	5
NX_ICMPV6_TIME_EXCEED_TYPE	3
NX_IGMP_HEADER_SIZE sizeof(NX_IGMP_HEADER)	
NX_IGMP_HOST_RESPONSE_TYPE	0x02000000
NX_IGMP_HOST_V2_JOIN_TYPE	0x16000000
NX_IGMP_HOST_V2_LEAVE_TYPE	0x17000000
NX_IGMP_HOST_VERSION_1	1
NX_IGMP_HOST_VERSION_2	2
NX_IGMP_MAX_RESP_TIME_MASK	0x00FF0000
NX_IGMP_MAX_UPDATE_TIME	10
NX_IGMP_PACKET	36
NX_IGMP_ROUTER_QUERY_TYPE	0x01000000
NX_IGMP_TTL	1
NX_IGMP_TYPE_MASK	0x0F000000
NX_IGMP_VERSION	0x10000000
NX_IGMPV2_TYPE_MASK	0xFF000000
NX_IN_PROGRESS	0x37

NX_INIT_PACKET_ID	1
NX_NOT_IMPLEMENTED	0x4A
NX_NOT_SUPPORTED	0x4B
NX_INVALID_INTERFACE	0x4C
NX_INVALID_PACKET	0x12
NX_INVALID_PORT	0x46
NX_INVALID_RELISTEN	0x47
NX_INVALID_SOCKET	0x13
NX_IP_ADDRESS_ERROR	0x21
NX_IP_ADDRESS_RESOLVED	0x0002
NX_IP_ALIGN_FRAGS	8
NX_IP_ALL_EVENTS	0xFFFFFFFF
NX_IP_ARP_ENABLED	0x0008
NX_IP_ARP_REC_EVENT	0x00000010
NX_IP_CLASS_A_HOSTID	0x00FFFFFF
NX_IP_CLASS_A_MASK	0x80000000
NX_IP_CLASS_A_NETID	0x7F000000
NX_IP_CLASS_A_TYPE	0x00000000
NX_IP_CLASS_B_HOSTID	0x0000FFFF
NX_IP_CLASS_B_MASK	0xC0000000
NX_IP_CLASS_B_NETID	0x3FFF0000
NX_IP_CLASS_B_TYPE	0x80000000
NX_IP_CLASS_C_HOSTID	0x000000FF
NX_IP_CLASS_C_MASK	0xE0000000
NX_IP_CLASS_C_NETID	0x1FFFFF00
NX_IP_CLASS_C_TYPE	0xC0000000
NX_IP_CLASS_D_GROUP	0x0FFFFFFF
NX_IP_CLASS_D_HOSTID	0x00000000
NX_IP_CLASS_D_MASK	0xF0000000
NX_IP_CLASS_D_TYPE	0xE0000000
NX_IP_DEBUG_LOG_SIZE	100
NX_IP_DONT_FRAGMENT	0x00004000
NX_IP_DRIVER_DEFERRED_EVENT	0x00000800
NX_IP_DRIVER_PACKET_EVENT	0x00000200
NX_IP_FRAGMENT_MASK	0x00003FFF

NX_IP_ICMP	0x00010000
NX_IP_ICMP_EVENT	0x00000004
NX_IP_ID	0x49502020
NX_IP_IGMP	0x00020000
NX_IP_IGMP_ENABLE_EVENT	0x00000400
NX_IP_IGMP_ENABLED	0x0040
NX_IP_IGMP_EVENT	0x00000040
NX_IP_INITIALIZE_DONE	0x0001
NX_IP_INTERNAL_ERROR	0x20
NX_IP_LENGTH_MASK	0x0F000000
NX_IP_LIMITIED_BROADCAST	0xFFFFFFFF
NX_IP_LINK_ENABLED	0x0004
NX_IP_LOOPBACK_FIRST	0x7F000000
NX_IP_LOOPBACK_LAST	0x7FFFFFFF
NX_IP_MAX_DATA	0x00080000
NX_IP_MAX_RELIABLE	0x00040000
NX_IP_MIN_COST	0x00020000
NX_IP_MIN_DELAY	0x00100000
NX_IP_MORE_FRAGMENT	0x00002000
NX_IP_MULTICAST_LOWER	0x5E000000
NX_IP_MULTICAST_MASK	0x007FFFFF
NX_IP_MULTICAST_UPPER	0x00000100
NX_IP_NORMAL	0x00000000
NX_IP_NORMAL_LENGTH	5
NX_IP_OFFSET_MASK	0x00001FFF
NX_IP_PACKET (IPv6 enabled	56
NX_IP_PACKET (IPv6 disabled	36
NX_IP_PACKET_SIZE_MASK	0x0000FFFF
NX_IP_PERIODIC_EVENT	0x00000001
NX_IP_PERIODIC_RATE	100
NX_IP_PROTOCOL_MASK	0x00FF0000
NX_IP_RARP_COMPLETE	0x0080
NX_IP_RARP_REC_EVENT	0x00000020
NX_IP_RECEIVE_EVENT	0x00000008
NX_IP_TCP	0x00060000

NX_IP_TCP_CLEANUP_DEFERRED	0x00001000
NX_IP_TCP_ENABLED	0x0020
NX_IP_TCP_EVENT	0x00000080
NX_IP_TCP_FAST_EVENT	0x00000100
NX_IP_TIME_TO_LIVE	0x00000080
NX_IP_TIME_TO_LIVE_MASK	0xFF000000
NX_IP_TIME_TO_LIVE_SHIFT	24
NX_IP_TOS_MASK	0x00FF0000
NX_IP_UDP	0x00110000
NX_IP_UDP_ENABLED	0x0010
NX_IP_UNFRAG_EVENT	0x00000002
NX_IP_VERSION	0x45000000
NX_IPV6_ADDRESS_INVALID	0
NX_IPV6_ADDRESS_LINKLOCAL	0x00000001
NX_IPV6_ADDRESS_SITELOCAL	0x00000002
NX_IPV6_ADDRESS_GLOBAL	0x00000004
NX_IPV6_ALL_NODE_MCAST	0x00000010
NX_IPV6_ALL_ROUTER_MCAST	0x00000020
NX_IPV6_SOLICITED_NODE_MCAST	0x00000040
NX_IPV6_ADDRESS_UNICAST	0x80000000
NX_IPV6_ADDRESS_MULTICAST	0x40000000
NX_IPV6_ADDRESS_UNSPECIFIED	0x20000000
NX_IPV6_ADDRESS_LOOPBACK	0x10000000
NX_IPV4_ICMP_PACKET	36
NX_IPV4_IGMP_PACKET	36
NX_IPV4_TCP_PACKET	56
NX_IPV4_UDP_PACKET	44
NX_IPV6_ICMP_PACKET	56
NX_IPV6_IGMP_PACKET	56
NX_IPV6_TCP_PACKET	76
NX_IPV6_UDP_PACKET	64
NX_IPV6_PROTOCOL_NEXT_HEADER_HOP_BY_HOP	0
NX_IPV6_PROTOCOL_NEXT_HEADER_ROUTING	43
NX_IPV6_PROTOCOL_NEXT_HEADER_FRAGMENT	44
NX_IPV6_PROTOCOL_NEXT_HEADER_ICMPV6	58

NX_IPV6_PROTOCOL_NO_NEXT_HEADER	59
NX_IPV6_PROTOCOL_NEXT_HEADER_DESTINATION	60
NX_IPV6_PROTOCOL_TCP	6
NX_IPV6_PROTOCOL_UDP	17
NX_IPV6_PROTOCOL_ICMPV6	58
NX_IPV6_PROTOCOL_ICMP	1
NX_IPV6_PROTOCOL_IPV4	4
NX_IPV6_PROTOCOL_IPV6	41
NX_IPV6_ADDR_STATE_UNKNOWN	0x00
NX_IPV6_ADDR_STATE_TENTATIVE	0x01
NX_IPV6_ADDR_STATE_PREFERRED	0x02
NX_IPV6_ADDR_STATE_DEPRECATED	0x03
NX_IPV6_ADDR_STATE_VALID	0x04
NX_IPV6_ROUTE_TYPE_NOT_ROUTER	0x00
NX_IPV6_ROUTE_TYPE_SOLICITATED	0x01
NX_IPV6_ROUTE_TYPE_UNSOLICITATED	0x02
NX_IPV6_ROUTE_TYPE_STATIC	0x04
NX_IPV6_ROUTE_TYPE_DEFAULT	0x40
NX_IPV6_ROUTE_TYPE_VALID	0x80
NX_LINK_ARP_RESPONSE_SEND	6
NX_LINK_ARP_SEND	5
NX_LINK_DEFERRED_PROCESSING	18
NX_LINK_DISABLE	3
NX_LINK_ENABLE	2
NX_LINK_GET_ALLOC_ERRORS	16
NX_LINK_GET_DUPLEX_TYPE	12
NX_LINK_GET_ERROR_COUNT	13
NX_LINK_GET_RX_COUNT	14
NX_LINK_GET_SPEED	11
NX_LINK_GET_STATUS	10
NX_LINK_GET_TX_COUNT	15
NX_LINK_INITIALIZE	1
NX_LINK_INTERFACE_ATTACH	19
NX_LINK_MULTICAST_JOIN	8
NX_LINK_MULTICAST_LEAVE	9

NX_LINK_PACKET_BROADCAST	4
NX_LINK_PACKET_SEND	0
NX_LINK_RARP_SEND	7
NX_LINK_UNINITIALIZE	17
NX_LINK_USER_COMMAND	50
NX_LOWER_16_MASK	0x0000FFFF
NX_MAX_LISTEN	0x33
NX_MAX_LISTEN_REQUESTS	10
NX_MAX_MULTICAST_GROUPS	7
NX_MAX_PORT	0xFFFF
NX_MORE_FRAGMENTS	0x00002000
NX_NO_FREE_PORTS	0x45
NX_NO_MAPPING	0x04
NX_NO_MORE_ENTRIES	0x17
NX_NO_PACKET	0x01
NX_NO_RESPONSE	0x29
NX_NO_WAIT	0
NX_NOT_BOUND	0x24
NX_NOT_CLOSED	0x35
NX_NOT_CONNECTED	0x38
NX_NOT_CREATED	0x27
NX_NOT_ENABLED	0x14
NX_NOT_IMPLEMENTED	0x4A
NX_NOT_LISTEN_STATE	0x36
NX_NOT_SUCCESSFUL	0x43
NX_NULL	0
NX_OPTION_ERROR	0x0a
NX_OVERFLOW	0x03
NX_PACKET_ALLOCATED	0xAAAAAAAA
NX_PACKET_DEBUG_LOG_SIZE	100
NX_PACKET_ENQUEUED	0xEEEEEEEE
NX_PACKET_FREE	0xFFFFFFFF
NX_PACKET_POOL_ID	0x5041434B
NX_PACKET_READY	0xBBBBBBBB
NX_PHYSICAL_HEADER	16

NX_PHYSICAL_TRAILER	4
NX_POOL_DELETED	0x30
NX_POOL_ERROR	0x06
NX_PORT_UNAVAILABLE	0x23
NX_PTR_ERROR	0x07
NX_RARP_HARDWARE_SIZE	0x06
NX_RARP_HARDWARE_TYPE	0x0001
NX_RARP_MESSAGE_SIZE	28
NX_RARP_OPTION_REQUEST	0x0003
NX_RARP_OPTION_RESPONSE	0x0004
NX_RARP_PROTOCOL_SIZE	0x04
NX_RARP_PROTOCOL_TYPE	0x0800
NX_RECEIVE_PACKET	0
NX_RESERVED_CODE0	0x19
NX_RESERVED_CODE1	0x25
NX_RESERVED_CODE2	0x32
NX_ROUTE_TABLE_MASK	0x1F
NX_ROUTE_TABLE_SIZE	32
NX_SEARCH_PORT_START	30000
NX_SHIFT_BY_16	16
NX_SIZE_ERROR	0x09
NX_SOCKET_UNBOUND	0x26
NX_SOCKETS_BOUND	0x28
NX_STILL_BOUND	0x42
NX_SUCCESS	0x00
NX_TCP_ACK_BIT	0x00100000
NX_TCP_ACK_TIMER_RATE	5
NX_TCP_CLIENT	1
NX_TCP_CLOSE_WAIT	6
NX_TCP_CLOSED	1
NX_TCP_CLOSING	9
NX_TCP_CONTROL_MASK	0x00170000
NX_TCP_EOL_KIND	0x00
NX_TCP_ESTABLISHED	5
NX_TCP_FAST_TIMER_RATE	10

NX_TCP_FIN_BIT	0x00010000
NX_TCP_FIN_WAIT_1	7
NX_TCP_FIN_WAIT_2	8
NX_TCP_HEADER_MASK	0xF0000000
NX_TCP_HEADER_SHIFT	28
NX_TCP_HEADER_SIZE	0x50000000
NX_TCP_ID	0x54435020
NX_TCP_KEEPALIVE_INITIAL	7200
NX_TCP_KEEPALIVE_RETRIES	10
NX_TCP_KEEPALIVE_RETRY	75
NX_TCP_LAST_ACK	11
NX_TCP_LISTEN_STATE	2
NX_TCP_MAXIMUM_RETRIES	10
NX_TCP_MAXIMUM_TX_QUEUE	20
NX_TCP_MSS_KIND	0x02
NX_TCP_MSS_OPTION	0x02040000
NX_TCP_MSS_SIZE	1460
NX_TCP_NOP_KIND	0x01
NX_TCP_OPTION_END	0x01010100
NX_TCP_PACKET (IPv6 enabled)	76
NX_TCP_PACKET (IPv6 disabled)	56
NX_TCP_PORT_TABLE_MASK	0x1F
NX_TCP_PORT_TABLE_SIZE	32
NX_TCP_PSH_BIT	0x00080000
NX_TCP_RETRY_SHIFT	0
NX_TCP_RST_BIT	0x00040000
NX_TCP_SERVER	2
NX_TCP_SYN_BIT	0x00020000
NX_TCP_SYN_HEADER	0x70000000
NX_TCP_SYN_RECEIVED	4
NX_TCP_SYN_SENT	3
NX_TCP_TIMED_WAIT	10
NX_TCP_TRANSMIT_TIMER_RATE	1
NX_TCP_URG_BIT	0x00200000
NX_TRUE	1

NX_TX_QUEUE_DEPTH	0x49
NX_UDP_ID	0x55445020
NX_UDP_PACKET (IPv6 enabled)	64
NX_UDP_PACKET (IPv6 disabled)	44
NX_UDP_PORT_TABLE_MASK	0x1F
NX_UDP_PORT_TABLE_SIZE	32
NX_UNDERFLOW	0x02
NX_UNHANDLED_COMMAND	0x44
NX_WAIT_ABORTED	0x1A
NX_WAIT_ERROR	0x08
NX_WAIT_FOREVER	0xFFFFFFFF
NX_WINDOW_OVERFLOW	0x39

Listings by Value

NX_ANY_PORT	0
NX_ARP_EXPIRATION_RATE	0
NX_FALSE	0
NX_ICMP_ECHO_REPLY_TYPE	0
NX_ICMP_NETWORK_UNREACH_CODE	0
NX_ICMPV6_NO_ROUTE_TO_DESTINATION_CODE	0
NX_IPV6_ADDRESS_INVALID	0
NX_IPV6_PROTOCOL_NEXT_HEADER_HOP_BY_HOP	0
NX_LINK_PACKET_SEND	0
NX_NO_WAIT	0
NX_NULL	0
NX_RECEIVE_PACKET	0
NX_TCP_RETRY_SHIFT	0
NX_IPV6_ADDR_STATE_UNKNOWN	0x00
NX_IPV6_ROUTE_TYPE_NOT_ROUTER	0x00
NX_SUCCESS	0x00
NX_TCP_EOL_KIND	0x00
NX_FRAGMENT_OKAY	0x00000000
NX_IP_CLASS_A_TYPE	0x00000000
NX_IP_CLASS_D_HOSTID	0x00000000
NX_IP_NORMAL	0x00000000
NX_FOREVER	1
NX_ICMP_HOST_UNREACH_CODE	1
NX_ICMPV6_COMMUNICATION_WITH_DESTINATION_PROHIBITED_CODE	1
NX_ICMPV6_DEST_UNREACHABLE_TYPE	1
NX_ICMPV6_NO_SLLA	1
NX_ICMPV6_OPTION_TYPE_SRC_LINK_ADDR	1
NX_IGMP_HOST_VERSION_1	1
NX_IGMP_TTL	1
NX_INIT_PACKET_ID	1
NX_IPV6_PROTOCOL_ICMP	1
NX_LINK_INITIALIZE	1
NX_TCP_CLIENT	1

NX_TCP_CLOSED	1
NX_TCP_TRANSMIT_TIMER_RATE	1
NX_TRUE	1
NX_IP_PERIODIC_EVENT	0x00000001
NX_IPV6_ADDRESS_LINKLOCAL	0x00000001
NX_ARP_HARDWARE_TYPE	0x0001
NX_ARP_OPTION_REQUEST	0x0001
NX_IP_INITIALIZE_DONE	0x0001
NX_RARP_HARDWARE_TYPE	0x0001
NX_IPV6_ADDR_STATE_TENTATIVE	0x01
NX_IPV6_ROUTE_TYPE_SOLICITATED	0x01
NX_NO_PACKET	0x01
NX_TCP_NOP_KIND	0x01
NX_ICMP_PROTOCOL_UNREACH_CODE	2
NX_ICMPV6_BEYOND_SCOPE_OF_SOURCE_ADDRESS_CODE	2
NX_ICMPV6_OPTION_TYPE_TRG_LINK_ADDR	2
NX_ICMPV6_PACKET_TOO_BIG_TYPE	2
NX_IGMP_HOST_VERSION_2	2
NX_LINK_ENABLE	2
NX_TCP_LISTEN_STATE	2
NX_TCP_SERVER	2
NX_IP_UNFRAG_EVENT	0x00000002
NX_IPV6_ADDRESS_SITELOCAL	0x00000002
NX_ARP_OPTION_RESPONSE	0x0002
NX_IP_ADDRESS_RESOLVED	0x0002
NX_IPV6_ADDR_STATE_PREFERRED	0x02
NX_IPV6_ROUTE_TYPE_UNSOLICITATED	0x02
NX_TCP_MSS_KIND	0x02
NX_UNDERFLOW	0x02
NX_ICMP_DEST_UNREACHABLE_TYPE	3
NX_ICMP_PORT_UNREACH_CODE	3
NX_ICMPV6_ADDRESS_UNREACHABLE_CODE	3
NX_ICMPV6_OPTION_TYPE_PREFIX_INFO	3
NX_ICMPV6_TIME_EXCEED_TYPE	3
NX_LINK_DISABLE	3

NX_TCP_SYN_SENT	3
NX_RARP_OPTION_REQUEST	0x0003
NX_IPV6_ADDR_STATE_DEPRECATED	0x03
NX_OVERFLOW	0x03
NX_ARP_MAX_QUEUE_DEPTH	4
NX_ICMP_FRAMENT_NEEDED_CODE	4
NX_ICMP_SOURCE_QUENCH_TYPE	4
NX_ICMPV6_DEST_UNREACHABLE_CODE	4
NX_ICMPV6_OPTION_REDIRECTED_HEADER	4
NX_ICMPV6_PARAMETER_PROBLEM_TYPE	4
NX_IPV6_PROTOCOL_IPV4	4
NX_LINK_PACKET_BROADCAST	4
NX_PHYSICAL_TRAILER	4
NX_TCP_SYN_RECEIVED	4
NX_IP_ICMP_EVENT	0x00000004
NX_IPV6_ADDRESS_GLOBAL	0x00000004
NX_IP_LINK_ENABLED	0x0004
NX_RARP_OPTION_RESPONSE	0x0004
NX_ARP_PROTOCOL_SIZE	0x04
NX_IPV6_ADDR_STATE_VALID	0x04
NX_IPV6_ROUTE_TYPE_STATIC	0x04
NX_NO_MAPPING	0x04
NX_RARP_PROTOCOL_SIZE	0x04
NX_NOT_IMPLEMENTED	0x4A
NX_NOT_SUPPORTED	0x4B
NX_INVALID_INTERFACE	0x4C
NX_ICMP_REDIRECT_TYPE	5
NX_ICMP_SOURCE_ROUTE_CODE	5
NX_ICMPV6_OPTION_TYPE_MTU	5
NX_ICMPV6_SOURCE_ADDRESS_FAILED_I_E_POLICY_CODE	5
NX_IP_NORMAL_LENGTH	5
NX_LINK_ARP_SEND	5
NX_TCP_ACK_TIMER_RATE	5
NX_TCP_ESTABLISHED	5
NX_DELETED	0x05

NX_ICMP_NETWORK_UNKNOWN_CODE	6
NX_ICMPV6_REJECT_ROUTE_TO_DESTINATION_CODE	6
NX_IPV6_PROTOCOL_TCP	6
NX_LINK_ARP_RESPONSE_SEND	6
NX_TCP_CLOSE_WAIT	6
NX_ARP_HARDWARE_SIZE	0x06
NX_POOL_ERROR	0x06
NX_RARP_HARDWARE_SIZE	0x06
NX_ICMP_HOST_UNKNOWN_CODE	7
NX_LINK_RARP_SEND	7
NX_MAX_MULTICAST_GROUPS	7
NX_TCP_FIN_WAIT_1	7
NX_PTR_ERROR	0x07
NX_ICMP_ECHO_REQUEST_TYPE	8
NX_ICMP_SOURCE_ISOLATED_CODE	8
NX_IP_ALIGN_FRAGS	8
NX_LINK_MULTICAST_JOIN	8
NX_TCP_FIN_WAIT_2	8
NX_IP_RECEIVE_EVENT	0x00000008
NX_IP_ARP_ENABLED	0x0008
NX_WAIT_ERROR	0x08
NX_ICMP_NETWORK_PROHIBIT_CODE	9
NX_LINK_MULTICAST_LEAVE	9
NX_TCP_CLOSING	9
NX_SIZE_ERROR	0x09
NX_ARP_UPDATE_RATE	10
NX_ICMP_HOST_PROHIBIT_CODE	10
NX_IGMP_MAX_UPDATE_TIME	10
NX_LINK_GET_STATUS	10
NX_MAX_LISTEN_REQUESTS	10
NX_TCP_FAST_TIMER_RATE	10
NX_TCP_KEEPALIVE_RETRIES	10
NX_TCP_MAXIMUM_RETRIES	10
NX_TCP_TIMED_WAIT	10
NX_IPV6_ALL_NODE_MCAST	0x00000010

NX_OPTION_ERROR	0x0a
NX_ICMP_NETWORK_SERVICE_CODE	11
NX_ICMP_TIME_EXCEEDED_TYPE	11
NX_LINK_GET_SPEED	11
NX_TCP_LAST_ACK	11
NX_ICMP_HOST_SERVICE_CODE	12
NX_ICMP_PARAMETER_PROB_TYPE	12
NX_LINK_GET_DUPLEX_TYPE	12
NX_ICMP_TIMESTAMP_REQ_TYPE	13
NX_LINK_GET_ERROR_COUNT	13
NX_ICMP_TIMESTAMP_REP_TYPE	14
NX_LINK_GET_RX_COUNT	14
NX_LINK_GET_TX_COUNT	15
NX_LINK_GET_ALLOC_ERRORS	16
NX_PHYSICAL_HEADER	16
NX_SHIFT_BY_16	16
NX_IP_ARP_REC_EVENT	0x00000010
NX_IP_UDP_ENABLED	0x0010
NX_DELETE_ERROR	0x10
NX_ICMP_ADDRESS_MASK_REQ_TYPE	17
NX_IPV6_PROTOCOL_UDP	17
NX_LINK_UNINITIALIZE	17
NX_CALLER_ERROR	0x11
NX_ARP_MAXIMUM_RETRIES	18
NX_ICMP_ADDRESS_MASK_REP_TYPE	18
NX_LINK_DEFERRED_PROCESSING	18
NX_INVALID_PACKET	0x12
NX_INVALID_SOCKET	0x13
NX_LINK_INTERFACE_ATTACH	19
NX_TCP_MAXIMUM_TX_QUEUE	20
NX_IPV6_ALL_ROUTER_MCAST	0x00000020
NX_NOT_ENABLED	0x14
NX_ALREADY_ENABLED	0x15
NX_ENTRY_NOT_FOUND	0x16
NX_NO_MORE_ENTRIES	0x17

NX_IP_TIME_TO_LIVE_SHIFT	24
NX_ARP_TIMER_ERROR	0x18
NX_RESERVED_CODE0	0x19
NX_WAIT_ABORTED	0x1A
NX_ARP_MESSAGE_SIZE	28
NX_RARP_MESSAGE_SIZE	28
NX_TCP_HEADER_SHIFT	28
NX_ROUTE_TABLE_MASK	0x1F
NX_TCP_PORT_TABLE_MASK	0x1F
NX_UDP_PORT_TABLE_MASK	0x1F
NX_ROUTE_TABLE_SIZE	32
NX_TCP_PORT_TABLE_SIZE	32
NX_UDP_PORT_TABLE_SIZE	32
NX_IP_RARP_REC_EVENT	0x00000020
NX_IP_TCP_ENABLED	0x0020
NX_IP_INTERNAL_ERROR	0x20
NX_IP_ADDRESS_ERROR	0x21
NX_ALREADY_BOUND	0x22
NX_PORT_UNAVAILABLE	0x23
NX_ICMP_PACKET	36
NX_IGMP_PACKET	36
NX_IP_PACKET	36
NX_IPV4_ICMP_PACKET	36
NX_IPV4_IGMP_PACKET	36
NX_NOT_BOUND	0x24
NX_RESERVED_CODE1	0x25
NX_SOCKET_UNBOUND	0x26
NX_NOT_CREATED	0x27
NX_SOCKETS_BOUND	0x28
NX_NO_RESPONSE	0x29
NX_IPV6_SOLICITED_NODE_MCAST	0x00000040
NX_IPV6_PROTOCOL_IPV6	41
NX_IPV6_PROTOCOL_NEXT_HEADER_ROUTING	43
NX_IPV4_UDP_PACKET	44
NX_IPV6_PROTOCOL_NEXT_HEADER_FRAGMENT	44

NX_UDP_PACKET	44
NX_POOL_DELETED	0x30
NX_ALREADY_RELEASED	0x31
NX_LINK_USER_COMMAND	50
NX_RESERVED_CODE2	0x32
NX_MAX_LISTEN	0x33
NX_DUPLICATE_LISTEN	0x34
NX_NOT_CLOSED	0x35
NX_NOT_LISTEN_STATE	0x36
NX_IN_PROGRESS	0x37
NX_NOT_CONNECTED	0x38
NX_WINDOW_OVERFLOW	0x39
NX_IP_IGMP_EVENT	0x00000040
NX_IP_IGMP_ENABLED	0x0040
NX_ALREADY_SUSPENDED	0x40
NX_IPV6_ROUTE_TYPE_DEFAULT	0x40
NX_DISCONNECT_FAILED	0x41
NX_STILL_BOUND	0x42
NX_NOT_SUCCESSFUL	0x43
NX_UNHANDLED_COMMAND	0x44
NX_NO_FREE_PORTS	0x45
NX_INVALID_PORT	0x46
NX_INVALID_RELISTEN	0x47
NX_CONNECTION_PENDING	0x48
NX_TX_QUEUE_DEPTH	0x49
NX_IPV4_TCP_PACKET	56
NX_IPV6_ICMP_PACKET	56
NX_IPV6_IGMP_PACKET	56
NX_TCP_PACKET	56
NX_IPV6_PROTOCOL_NEXT_HEADER_ICMPV6	58
NX_IPV6_PROTOCOL_ICMPV6	58
NX_IPV6_PROTOCOL_NO_NEXT_HEADER	59
NX_IPV6_PROTOCOL_NEXT_HEADER_DESTINATION	60
NX_IPV6_UDP_PACKET	64
NX_TCP_KEEPALIVE_RETRY	75

NX_IPV6_TCP_PACKET	76
NX_ARP_DEBUG_LOG_SIZE	100
NX_ICMP_DEBUG_LOG_SIZE	100
NX_IGMP_DEBUG_LOG_SIZE	100
NX_IP_DEBUG_LOG_SIZE	100
NX_IP_PERIODIC_RATE	100
NX_PACKET_DEBUG_LOG_SIZE	100
NX_RARP_DEBUG_LOG_SIZE	100
NX_TCP_DEBUG_LOG_SIZE	100
NX_UDP_DEBUG_LOG_SIZE	100
NX_IP_TCP_EVENT	0x00000080
NX_IP_TIME_TO_LIVE	0x00000080
NX_IP_RARP_COMPLETE	0x0080
NX_IPV6_ROUTE_TYPE_VALID	0x80
NX_NOT_IMPLEMENTED	0x4A
NX_IP_CLASS_C_HOSTID	0x000000FF
NX_IP_MULTICAST_UPPER	0x00000100
NX_IP_TCP_FAST_EVENT	0x00000100
NX_IP_DRIVER_PACKET_EVENT	0x00000200
NX_IP_IGMP_ENABLE_EVENT	0x00000400
NX_IP_DRIVER_DEFERRED_EVENT	0x00000800
NX_ARP_PROTOCOL_TYPE	0x0800
NX_RARP_PROTOCOL_TYPE	0x0800
NX_IP_TCP_CLEANUP_DEFERRED	0x00001000
NX_ICMPV6_ECHO_REQUEST_TYPE	128
NX_ICMPV6_ECHO_REPLY_TYPE	129
NX_ICMPV6_ROUTER_SOLICITATION_TYPE	133
NX_ICMPV6_ROUTER_ADVERTISEMENT_TYPE	134
NX_ICMPV6_NEIGHBOR_SOLICITATION_TYPE	135
NX_ICMPV6_NEIGHBOR_ADVERTISEMENT_TYPE	136
NX_ICMPV6_REDIRECT_MESSAGE_TYPE	137
NX_ICMPV6_MINIMUM_IPV4_PATH_MTU	576
NX_ICMPV6_MINIMUM_IPV6_PATH_MTU	1280
NX_TCP_KEEPALIVE_INITIAL	7200
NX_FRAG_OFFSET_MASK	0x00001FFF

NX_IP_OFFSET_MASK	0x00001FFF
NX_IP_MORE_FRAGMENT	0x00002000
NX_MORE_FRAGMENTS	0x00002000
NX_IP_FRAGMENT_MASK	0x00003FFF
NX_TCP_MSS_SIZE	16384
NX_DONT_FRAGMENT	0x00004000
NX_IP_DONT_FRAGMENT	0x00004000
NX_SEARCH_PORT_START	30000
NX_IP_CLASS_B_HOSTID	0x0000FFFF
NX_IP_PACKET_SIZE_MASK	0x0000FFFF
NX_LOWER_16_MASK	0x0000FFFF
NX_MAX_PORT	0xFFFF
NX_IP_ICMP	0x00010000
NX_TCP_FIN_BIT	0x00010000
NX_CARRY_BIT	0x10000
NX_IP_IGMP	0x00020000
NX_IP_MIN_COST	0x00020000
NX_TCP_SYN_BIT	0x00020000
NX_IP_MAX_RELIABLE	0x00040000
NX_TCP_RST_BIT	0x00040000
NX_IP_TCP	0x00060000
NX_IP_MAX_DATA	0x00080000
NX_TCP_PSH_BIT	0x00080000
NX_IP_MIN_DELAY	0x00100000
NX_TCP_ACK_BIT	0x00100000
NX_IP_UDP	0x00110000
NX_TCP_CONTROL_MASK	0x00170000
NX_TCP_URG_BIT	0x00200000
NX_IP_MULTICAST_MASK	0x007FFFFF
NX_IP_PROTOCOL_MASK	0x00FF0000
NX_IP_TOS_MASK	0x00FF0000
NX_IGMP_ROUTER_QUERY_TYPE	0x01000000
NX_TCP_OPTION_END	0x01010402
NX_IGMP_HOST_RESPONSE_TYPE	0x02000000
NX_TCP_MSS_OPTION	0x02040000

NX_IGMP_TYPE_MASK	0x0F000000
NX_IP_LENGTH_MASK	0x0F000000
NX_IGMP_MAX_RESP_TIME_MASK	0x00FF0000
NX_IP_CLASS_A_HOSTID	0x00FFFFFF
NX_IP_CLASS_D_GROUP	0x0FFFFFFF
NX_IGMP_VERSION	0x10000000
NX_IPV6_ADDRESS_LOOPBACK	0x10000000
NX_IGMP_HOST_V2_JOIN_TYPE	0x16000000
NX_IGMP_HOST_V2_LEAVE_TYPE	0x17000000
NX_IPV6_ADDRESS_UNSPECIFIED	0x20000000
NX_IP_CLASS_C_NETID	0x1FFFFF00
NX_IP_CLASS_B_NETID	0x3FFF0000
NX_IPV6_ADDRESS_MULTICAST	0x40000000
NX_IP_VERSION	0x45000000
NX_IP_ID	0x49502020
NX_TCP_HEADER_SIZE	0x50000000
NX_PACKET_POOL_ID	0x5041434B
NX_TCP_ID	0x54435020
NX_UDP_ID	0x55445020
NX_IP_MULTICAST_LOWER	0x5E000000
NX_IP_CLASS_A_NETID	0x7F000000
NX_TCP_SYN_HEADER	0x70000000
NX_IP_LOOPBACK_FIRST	0x7F000000
NX_IP_LOOPBACK_LAST	0x7FFFFFFF
NX_IP_CLASS_A_MASK	0x80000000
NX_IP_CLASS_B_TYPE	0x80000000
NX_IPV6_ADDRESS_UNICAST	0x80000000
NX_PACKET_ALLOCATED	0xAAAAAAAA
NX_PACKET_READY	0xBBBBBBBB
NX_IP_CLASS_B_MASK	0xC0000000
NX_IP_CLASS_C_TYPE	0xC0000000
NX_DRIVER_TX_DONE	0xDDDDDDDD
NX_IP_CLASS_C_MASK	0xE0000000
NX_IP_CLASS_D_TYPE	0xE0000000
NX_PACKET_ENQUEUED	0xEEEEEEEE

NX_IGMP_VERSION_MASK	0xF0000000
NX_IP_CLASS_D_MASK	0xF0000000
NX_TCP_HEADER_MASK	0xF0000000
NX_ALL_HOSTS_ADDRESS	0xFE000001
NX_IGMPV2_TYPE_MASK	0xFF000000
NX_IP_TIME_TO_LIVE_MASK	0xFF000000
NX_ICMPV6_PATH_MTU_INFINITE_TIMEOUT	0xFFFFFFFF
NX_IP_ALL_EVENTS	0xFFFFFFFF
NX_IP_LIMITIED_BROADCAST	0xFFFFFFFF
NX_PACKET_FREE	0xFFFFFFFF
NX_WAIT_FOREVER	0xFFFFFFFF
NX_IGMP_HEADER_SIZE sizeof(NX_IGMP_HEADER)	

NetX Data Types

```
typedef struct NX_ARP_STRUCT
{

    UINT                                        nx_arp_route_static;
    UINT                                     nx_arp_entry_next_update;
    UINT                                     nx_arp_retries;
    struct NX_ARP_STRUCT              *nx_arp_pool_next,
            *nx_arp_pool_previous;
    struct NX_ARP_STRUCT              *nx_arp_active_next,
                                             *nx_arp_active_previous,
                                             **nx_arp_active_list_head;
    ULONG                                    nx_arp_ip_address;
    ULONG                                    nx_arp_physical_address_msw;
    ULONG                                    nx_arp_physical_address_lsw;
    struct NX_INTERFACE_STRUCT *nx_arp_ip_interface;
    struct NX_PACKET_STRUCT       *nx_arp_packets_waiting;
} NX_ARP;


typedef struct NX_INTERFACE_STRUCT
{
    CHAR         *nx_interface_name;
    UCHAR       nx_interface_valid;
    UCHAR       nx_interface_address_mapping_needed;
    UCHAR       nx_interface_link_up;
    UCHAR       reserved;
    struct           NX_IP_STRUCT *nx_interface_ip_instance;
    ULONG       nx_interface_physical_address_msw;
    ULONG       nx_interface_physical_address_lsw;
    ULONG       nx_interface_ip_address;
    ULONG       nx_interface_ip_network_mask;
    ULONG       nx_interface_ip_network;
    struct          NXD_IPV6_ADDRESS_STRUCT *nxd_interface_ipv6_address_list_head;
    ULONG       nx_interface_ip_mtu_size;
    VOID           *nx_interface_additional_link_info;
    VOID           (*nx_interface_link_driver_entry)(struct NX_IP_DRIVER_STRUCT *);
} NX_INTERFACE;


typedef struct NX_IP_STRUCT
{
    ULONG     nx_ip_id;
    CHAR        *nx_ip_name;

/* Existing fields specific to the NX_IP struct intended for single homed hosts before multihome
support
   was added are defined as follows:
*/
#define nx_ip_address                  nx_ip_interface[0].nx_interface_ip_address
#define nx_ip_driver_mtu              nx_ip_interface[0].nx_interface_ip_mtu_size
#define nx_ip_driver_mapping_needed   nx_ip_interface[0].nx_interface_address_mapping_needed
#define nx_ip_network_mask             nx_ip_interface[0].nx_interface_ip_network_mask
#define nx_ip_network                  nx_ip_interface[0].nx_interface_ip_network
#define nx_ip_arp_physical_address_msw
nx_ip_interface[0].nx_interface_physical_address_msw
#define nx_ip_arp_physical_address_lsw
nx_ip_interface[0].nx_interface_physical_address_lsw
#define nx_ip_driver_link_up           nx_ip_interface[0].nx_interface_link_up
#define nx_ip_link_driver_entry        nx_ip_interface[0].nx_interface_link_driver_entry
#define nx_ip_additional_link_info     nx_ip_interface[0].nx_interface_additional_link_info

    ULONG                                    nx_ip_gateway_address;
   struct NX_INTERFACE_STRUCT       *nx_ip_gateway_interface;

#ifdef FEATURE_NX_IPV6
    struct NXD_IPV6_ADDRESS_STRUCT
nx_ipv6_address[NX_MAX_IPV6_ADDRESSES];
#endif /* FEATURE_NX_IPV6 */

    ULONG         nx_ip_total_packet_send_requests;
```

```
    ULONG         nx_ip_total_packets_sent;
    ULONG         nx_ip_total_bytes_sent;
    ULONG         nx_ip_total_packets_received;
    ULONG         nx_ip_total_packets_delivered;
    ULONG         nx_ip_total_bytes_received;
    ULONG         nx_ip_packets_forwarded;
    ULONG         nx_ip_packets_reassembled;
    ULONG         nx_ip_reassembly_failures;
    ULONG         nx_ip_invalid_packets;
    ULONG         nx_ip_invalid_transmit_packets;
    ULONG         nx_ip_invalid_receive_address;
    ULONG         nx_ip_unknown_protocols_received;
    ULONG         nx_ip_transmit_resource_errors;
    ULONG         nx_ip_transmit_no_route_errors;
    ULONG         nx_ip_receive_packets_dropped;
    ULONG         nx_ip_receive_checksum_errors;
    ULONG         nx_ip_send_packets_dropped;
    ULONG         nx_ip_total_fragment_requests;
    ULONG         nx_ip_successful_fragment_requests;
    ULONG         nx_ip_fragment_failures;
    ULONG         nx_ip_total_fragments_sent;
    ULONG         nx_ip_total_fragments_received;
    ULONG         nx_ip_arp_requests_sent;
    ULONG         nx_ip_arp_requests_received;
    ULONG         nx_ip_arp_responses_sent;
    ULONG         nx_ip_arp_responses_received;
    ULONG         nx_ip_arp_aged_entries;
    ULONG         nx_ip_arp_invalid_messages;
    ULONG         nx_ip_arp_static_entries;
    ULONG         nx_ip_udp_packets_sent;
    ULONG         nx_ip_udp_bytes_sent;
    ULONG         nx_ip_udp_packets_received;
    ULONG         nx_ip_udp_bytes_received;
    ULONG         nx_ip_udp_invalid_packets;
    ULONG         nx_ip_udp_no_port_for_delivery;
    ULONG         nx_ip_udp_receive_packets_dropped;
    ULONG         nx_ip_udp_checksum_errors;
    ULONG         nx_ip_tcp_packets_sent;
    ULONG         nx_ip_tcp_bytes_sent;
    ULONG         nx_ip_tcp_packets_received;
    ULONG         nx_ip_tcp_bytes_received;
    ULONG       nx_ip_tcp_invalid_packets;
    ULONG       nx_ip_tcp_receive_packets_dropped;
    ULONG       nx_ip_tcp_checksum_errors;
    ULONG       nx_ip_tcp_connections;
    ULONG      nx_ip_tcp_passive_connections;
    ULONG      nx_ip_tcp_active_connections;
    ULONG       nx_ip_tcp_disconnections;
    ULONG       nx_ip_tcp_connections_dropped;
    ULONG       nx_ip_tcp_retransmit_packets;
    ULONG       nx_ip_tcp_resets_received;
    ULONG       nx_ip_tcp_resets_sent;
    ULONG       nx_ip_icmp_total_messages_received;
    ULONG       nx_ip_icmp_checksum_errors;
    ULONG       nx_ip_icmp_invalid_packets;
    ULONG      nx_ip_icmp_unhandled_messages;
    ULONG      nx_ip_pings_sent;
    ULONG       nx_ip_ping_timeouts;
    ULONG       nx_ip_ping_threads_suspended;
    ULONG       nx_ip_ping_responses_received;
    ULONG       nx_ip_pings_received;
    ULONG       nx_ip_pings_responded_to;
    ULONG       nx_ip_igmp_invalid_packets;
    ULONG       nx_ip_igmp_reports_sent;
    ULONG       nx_ip_igmp_queries_received;
    ULONG       nx_ip_igmp_checksum_errors;
    ULONG       nx_ip_igmp_groups_joined;
#ifndef NX_DISABLE_IGMPV2
    ULONG       nx_ip_igmp_router_version;
#endif
```

```
    ULONG       nx_ip_rarp_requests_sent;
    ULONG       nx_ip_rarp_responses_received;
    ULONG       nx_ip_rarp_invalid_messages;
    VOID          (*nx_ip_forward_packet_process)(struct NX_IP_STRUCT *, NX_PACKET *);
    ULONG       nx_ip_packet_id;
    struct NX_PACKET_POOL_STRUCT                *nx_ip_default_packet_pool;
    TX_MUTEX    nx_ip_protection;
    UINT           nx_ip_initialize_done;
    UINT           nx_ip_driver_mapping_needed;
    NX_PACKET *nx_ip_driver_deferred_packet_head,
                *nx_ip_driver_deferred_packet_tail;
    VOID          (*nx_ip_driver_deferred_packet_handler)(struct NX_IP_STRUCT *, NX_PACKET *);
    NX_PACKET *nx_ip_deferred_received_packet_head,
               *nx_ip_deferred_received_packet_tail;
    VOID          (*nx_ip_raw_ip_processing)(struct NX_IP_STRUCT *, NX_PACKET *);
    NX_PACKET *nx_ip_raw_received_packet_head,
               *nx_ip_raw_received_packet_tail;
    ULONG       nx_ip_raw_received_packet_count;
    TX_THREAD *nx_ip_raw_packet_suspension_list;
    ULONG       nx_ip_raw_packet_suspended_count;
    TX_THREAD   nx_ip_thread;
    TX_EVENT_FLAGS_GROUP             nx_ip_events;
    TX_TIMER    nx_ip_periodic_timer;
    VOID        (*nx_ip_fragment_processing)(struct NX_IP_DRIVER_STRUCT *);
    VOID        (*nx_ip_fragment_assembly)(struct NX_IP_STRUCT *);
    VOID        (*nx_ip_fragment_timeout_check)(struct NX_IP_STRUCT *);
    NX_PACKET *nx_ip_timeout_fragment;
    NX_PACKET *nx_ip_received_fragment_head,
               *nx_ip_received_fragment_tail;
    NX_PACKET *nx_ip_fragment_assembly_head,
                *nx_ip_fragment_assembly_tail;
    VOID        (*nx_ip_address_change_notify)(struct NX_IP_STRUCT *, VOID *);
    VOID        *nx_ip_address_change_notify_additional_info;
    ULONG       nx_ip_igmp_join_list[NX_MAX_MULTICAST_GROUPS];
    NX_INTERFACE
*nx_ip_igmp_join_interface_list[NX_MAX_MULTICAST_GROUPS];
    ULONG       nx_ip_igmp_join_count[NX_MAX_MULTICAST_GROUPS];
    ULONG       nx_ip_igmp_update_time[NX_MAX_MULTICAST_GROUPS];
    UINT        nx_ip_igmp_group_loopback_enable[NX_MAX_MULTICAST_GROUPS];
    UINT        nx_ip_igmp_global_loopback_enable;
    void        (*nx_ip_igmp_packet_receive)(struct NX_IP_STRUCT *, struct NX_PACKET_STRUCT *);
    void        (*nx_ip_igmp_periodic_processing)(struct NX_IP_STRUCT *);
    void        (*nx_ip_igmp_queue_process)(struct NX_IP_STRUCT *);
    NX_PACKET  *nx_ip_igmp_queue_head;
    ULONG       nx_ip_icmp_sequence;
    void        (*nx_ip_icmp_packet_receive)(struct NX_IP_STRUCT *, struct NX_PACKET_STRUCT *);
    void        (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *);
    void        (*nx_ip_icmpv4_packet_process)(struct NX_IP_STRUCT *, NX_PACKET *);
#ifdef FEATURE_NX_IPV6
    void        (*nx_ip_icmpv6_packet_process)(struct NX_IP_STRUCT *, NX_PACKET *);
    void        (*nx_icmpv6_process_router_advertisement)(struct NX_IP_STRUCT *, NX_PACKET *);
    void        (*nx_nd_cache_fast_periodic_update)(struct NX_IP_STRUCT *);
    void        (*nx_nd_cache_slow_periodic_update)(struct NX_IP_STRUCT *);
#ifndef NX_DISABLE_IPV6_PATH_MTU_DISCOVERY
    void        (*nx_destination_table_periodic_update)(struct NX_IP_STRUCT *);
#endif
#endif /* FEATURE_NX_IPV6 */

    NX_PACKET *nx_ip_icmp_queue_head;
    TX_THREAD *nx_ip_icmp_ping_suspension_list;
    ULONG        nx_ip_icmp_ping_suspended_count;
    struct NX_UDP_SOCKET_STRUCT   *nx_ip_udp_port_table[NX_UDP_PORT_TABLE_SIZE];
    struct NX_UDP_SOCKET_STRUCT    *nx_ip_udp_created_sockets_ptr;
    ULONG       nx_ip_udp_created_sockets_count;
    void        (*nx_ip_udp_packet_receive)(struct NX_IP_STRUCT *, struct NX_PACKET_STRUCT *);
    UINT        nx_ip_udp_port_search_start;
    struct NX_TCP_SOCKET_STRUCT    *nx_ip_tcp_port_table[NX_TCP_PORT_TABLE_SIZE];
    struct NX_TCP_SOCKET_STRUCT    *nx_ip_tcp_created_sockets_ptr;
    ULONG       nx_ip_tcp_created_sockets_count;
    void        (*nx_ip_tcp_packet_receive)(struct NX_IP_STRUCT *, struct NX_PACKET_STRUCT *);
    void        (*nx_ip_tcp_periodic_processing)(struct NX_IP_STRUCT *);
    void        (*nx_ip_tcp_fast_periodic_processing)(struct NX_IP_STRUCT *);
```

```
    void        (*nx_ip_tcp_queue_process)(struct NX_IP_STRUCT *);
    NX_PACKET *nx_ip_tcp_queue_head,
                *nx_ip_tcp_queue_tail;
    ULONG       nx_ip_tcp_received_packet_count;
    struct NX_TCP_LISTEN_STRUCT   nx_ip_tcp_server_listen_reqs[NX_MAX_LISTEN_REQUESTS];
    struct NX_TCP_LISTEN_STRUCT      *nx_ip_tcp_available_listen_requests;
    struct NX_TCP_LISTEN_STRUCT      *nx_ip_tcp_active_listen_requests;
    UINT        nx_ip_tcp_port_search_start;
    UINT        nx_ip_fast_periodic_timer_created;
    TX_TIMER    nx_ip_fast_periodic_timer;
    struct NX_ARP_STRUCT                 *nx_ip_arp_table[NX_ARP_TABLE_SIZE];
    struct NX_ARP_STRUCT                 *nx_ip_arp_static_list;
    struct NX_ARP_STRUCT                    *nx_ip_arp_dynamic_list;
    ULONG       nx_ip_arp_dynamic_active_count;
    NX_PACKET *nx_ip_arp_deferred_received_packet_head,
                *nx_ip_arp_deferred_received_packet_tail;
    UINT        (*nx_ip_arp_allocate)(struct NX_IP_STRUCT *, struct NX_ARP_STRUCT **);
    void        (*nx_ip_arp_periodic_update)(struct NX_IP_STRUCT *);
    void        (*nx_ip_arp_queue_process)(struct NX_IP_STRUCT *);
    void        (*nx_ip_arp_packet_send)(struct NX_IP_STRUCT *, ULONG destination_ip, ,
   NX_INTERFACE *nx_interface);
    void        (*nx_ip_arp_gratuitous_response_handler)(struct NX_IP_STRUCT *, NX_PACKET *);
    void        (*nx_ip_arp_collision_notify_response_handler)(void *);
    void        *nx_ip_arp_collision_notify_parameter;
    ULONG       nx_ip_arp_collision_notify_ip_address;
    struct NX_ARP_STRUCT                 *nx_ip_arp_cache_memory;
    ULONG       nx_ip_arp_total_entries;
    void        (*nx_ip_rarp_periodic_update)(struct NX_IP_STRUCT *);
    void        (*nx_ip_rarp_queue_process)(struct NX_IP_STRUCT *);
    NX_PACKET *nx_ip_rarp_deferred_received_packet_head,
               *nx_ip_rarp_deferred_received_packet_tail;
    struct NX_IP_STRUCT
               *nx_ip_created_next,
                *nx_ip_created_previous;
    void        *nx_ip_reserved_ptr;
    void        (*nx_tcp_deferred_cleanup_check)(struct NX_IP_STRUCT *);
    NX_INTERFACE
nx_ip_interface[NX_MAX_IP_INTERFACES];
    void        (*nx_ipv4_packet_receive)(struct NX_IP_STRUCT *, NX_PACKET *);

#ifdef NX_ENABLE_IP_STATIC_ROUTING
    NX_IP_ROUTING_ENTRY                        nx_ip_routing_table[NX_IP_ROUTING_TABLE_SIZE];
    ULONG       nx_ip_routing_table_entry_count;
    ULONG       (*nx_ip_find_route_process)(struct NX_IP_STRUCT*, ULONG);
#endif /* NX_ENABLE_IP_STATIC_ROUTING */

#ifdef FEATURE_NX_IPV6
    USHORT  nx_ipv6_default_router_table_size;
    NX_IPV6_DEFAULT_ROUTER_ENTRY
 nx_ipv6_default_router_table[NX_IPV6_DEFAULT_ROUTER_TABLE_SIZE];
    UINT  nx_ipv6_default_router_table_round_robin_index;
    NX_IPV6_PREFIX_ENTRY nx_ipv6_prefix_list_table[NX_IPV6_PREFIX_LIST_TABLE_SIZE];
    NX_IPV6_PREFIX_ENTRY *nx_ipv6_prefix_list_ptr;
    NX_IPV6_PREFIX_ENTRY *nx_ipv6_prefix_entry_free_list;
    void        (*nx_ipv6_packet_receive)(struct NX_IP_STRUCT *, NX_PACKET *);
    ULONG       nx_ipv6_retrans_timer_ticks;
    ULONG       nx_ipv6_reachable_timer;
    ULONG       nx_ipv6_hop_limit;
#ifndef NXDUO_DISABLE_ICMPV6_ROUTER_SOLICITATION
    ULONG       nx_ipv6_rtr_solicitation_max;
    ULONG       nx_ipv6_rtr_solicitation_count;
    ULONG       nx_ipv6_rtr_solicitation_interval;
    ULONG       nx_ipv6_rtr_solicitation_timer;
#endif /* NXDUO_DISABLE_ICMPV6_ROUTER_SOLICITATION */
#endif /* FEATURE_NX_IPV6 */

} NX_IP;

typedef struct NX_IP_DRIVER_STRUCT
```

```
{
    UINT        nx_ip_driver_command;
    UINT        nx_ip_driver_status;
    ULONG       nx_ip_driver_physical_address_msw;
    ULONG       nx_ip_driver_physical_address_lsw;
    NX_PACKET *nx_ip_driver_packet;
    ULONG       *nx_ip_driver_return_ptr;
    struct NX_IP_STRUCT  *nx_ip_driver_ptr;
    NX_INTERFACE
                                                 *nx_ip_driver_interface;
} NX_IP_DRIVER;

#ifdef NX_ENABLE_IP_STATIC_ROUTING
typedef struct NX_IP_ROUTING_ENTRY_STRUCT
{
    ULONG     nx_ip_routing_dest_ip;
    ULONG     nx_ip_routing_net_mask;
    ULONG     nx_ip_routing_next_hop_address;
    NX_INTERFACE
                                                 *nx_ip_routing_entry_ip_interface;
} NX_IP_ROUTING_ENTRY;
#endif /* FEATURE_NX_IPV6 */

#ifdef FEATURE_NX_IPV6
typedef struct NX_IPV6_DEFAULT_ROUTER_ENTRY_STRUCT
{
    UCHAR  nx_ipv6_default_router_entry_flag;
    UCHAR  nx_ipv6_default_router_entry_reserved;
    USHORT nx_ipv6_default_router_entry_life_time;
    ULONG  nx_ipv6_default_router_entry_router_address[4];
    VOID  *nx_ipv6_default_router_entry_neighbor_cache_ptr;
} NX_IPV6_DEFAULT_ROUTER_ENTRY;
#endif /* FEATURE_NX_IPV6 */

#ifdef FEATURE_NX_IPV6
typedef struct NX_IPV6_PREFIX_ENTRY_STRUCT
{
    ULONG  nx_ipv6_prefix_entry_network_address[4];
    ULONG  nx_ipv6_prefix_entry_prefix_length;
    ULONG  nx_ipv6_prefix_entry_valid_lifetime;
    struct NX_IPV6_PREFIX_ENTRY_STRUCT * nx_ipv6_prefix_entry_prev;
    struct NX_IPV6_PREFIX_ENTRY_STRUCT * nx_ipv6_prefix_entry_next;
} NX_IPV6_PREFIX_ENTRY;

typedef  struct NX_PACKET_STRUCT
{
    struct NX_PACKET_POOL_STRUCT    *nx_packet_pool_owner;
    struct NX_PACKET_STRUCT                *nx_packet_queue_next;
    struct NX_PACKET_STRUCT                *nx_packet_tcp_queue_next;
    struct NX_PACKET_STRUCT                *nx_packet_next;
    struct NX_PACKET_STRUCT                *nx_packet_last;
    struct NX_PACKET_STRUCT                *nx_packet_fragment_next;
    ULONG   nx_packet_length;
    struct NX_INTERFACE_STRUCT          *nx_packet_ip_interface;
    ULONG   nx_packet_next_hop_address;
    UCHAR   *nx_packet_data_start;
    UCHAR   *nx_packet_data_end;
    UCHAR   *nx_packet_prepend_ptr;
    UCHAR   *nx_packet_append_ptr;
#ifdef NX_PACKET_HEADER_PAD
    ULONG    nx_packet_packet_pad;
#endif
    ULONG    nx_packet_reassembly_time;
    UCHAR    nx_packet_option_state;
    UCHAR    nx_packet_destination_header;
    USHORT nx_packet_option_offset;
    ULONG    nx_packet_ip_version;
#ifdef FEATURE_NX_IPV6
    ULONG    nx_packet_ipv6_dest_addr[4];
    ULONG    nx_packet_ipv6_src_addr[4];
    struct       NXD_IPV6_ADDRESS_STRUCT *nx_packet_interface;
```

```
#endif /* FEATURE_NX_IPV6 */
    UCHAR    *nx_packet_ip_header;
} NX_PACKET;

typedef struct NX_PACKET_POOL_STRUCT
{
    ULONG       nx_packet_pool_id;
    CHAR        *nx_packet_pool_name;
    ULONG      nx_packet_pool_available;
    ULONG       nx_packet_pool_total;
    ULONG       nx_packet_pool_empty_requests;
    ULONG       nx_packet_pool_empty_suspensions;
    ULONG       nx_packet_pool_invalid_releases;
    struct NX_PACKET_STRUCT                  *nx_packet_pool_available_list;
    CHAR        *nx_packet_pool_start;
    ULONG       nx_packet_pool_size;
    ULONG       nx_packet_pool_payload_size;
    TX_THREAD *nx_packet_pool_suspension_list;
    ULONG       nx_packet_pool_suspended_count;
    struct NX_PACKET_POOL_STRUCT         *nx_packet_pool_created_next,
                                             *nx_packet_pool_created_previous;
} NX_PACKET_POOL;

typedef struct NX_TCP_LISTEN_STRUCT
{
    UINT        nx_tcp_listen_port;
    VOID        (*nx_tcp_listen_callback)(NX_TCP_SOCKET *socket_ptr, UINT port);
    NX_TCP_SOCKET  *nx_tcp_listen_socket_ptr;
    ULONG       nx_tcp_listen_queue_maximum;
    ULONG       nx_tcp_listen_queue_current;
    NX_PACKET *nx_tcp_listen_queue_head,
                *nx_tcp_listen_queue_tail;
    struct NX_TCP_LISTEN_STRUCT
                *nx_tcp_listen_next,
                *nx_tcp_listen_previous;
} NX_TCP_LISTEN;

typedef struct NX_TCP_SOCKET_STRUCT
{
    ULONG       nx_tcp_socket_id;
    CHAR        *nx_tcp_socket_name;
    UINT        nx_tcp_socket_client_type;
    UINT        nx_tcp_socket_port;
    ULONG       nx_tcp_socket_mss;
    NXD_ADDRESS
nx_tcp_socket_connect_ip;
    UINT        nx_tcp_socket_connect_port;
    ULONG       nx_tcp_socket_connect_mss;
    NX_INTERFACE
                                             *nx_tcp_socket_connect_interface;
    ULONG  nx_tcp_socket_next_hop_address;
    ULONG       nx_tcp_socket_connect_mss2;

    ULONG       nx_tcp_socket_tx_slow_start_threshold;
    UINT        nx_tcp_socket_state;
    ULONG       nx_tcp_socket_tx_sequence;
    ULONG       nx_tcp_socket_rx_sequence;
    ULONG       nx_tcp_socket_rx_sequence_acked;
    ULONG       nx_tcp_socket_delayed_ack_timeout;
    ULONG       nx_tcp_socket_fin_sequence;
    ULONG       nx_tcp_socket_fin_received;
    ULONG       nx_tcp_socket_tx_window_advertised;
    ULONG       nx_tcp_socket_tx_window_congestion;
    ULONG       nx_tcp_socket_tx_outstanding_bytes;
    ULONG       nx_tcp_socket_ack_n_packet_counter;
    UINT        nx_tcp_socket_duplicated_ack_received;
    UINT        nx_tcp_socket_need_fast_retransmit;
    ULONG       nx_tcp_socket_rx_window_default;
    ULONG       nx_tcp_socket_rx_window_current;
    ULONG       nx_tcp_socket_rx_window_last_sent;
```

```
    ULONG       nx_tcp_socket_packets_sent;
    ULONG       nx_tcp_socket_bytes_sent;
    ULONG       nx_tcp_socket_packets_received;
    ULONG       nx_tcp_socket_bytes_received;
    ULONG       nx_tcp_socket_retransmit_packets;
    ULONG       nx_tcp_socket_checksum_errors;
    struct NX_IP_STRUCT    *nx_tcp_socket_ip_ptr;
    ULONG       nx_tcp_socket_type_of_service;
    UINT        nx_tcp_socket_time_to_live;
    ULONG       nx_tcp_socket_fragment_enable;
    ULONG   nx_tcp_socket_receive_queue_count;
    NX_PACKET *nx_tcp_socket_receive_queue_head,
                *nx_tcp_socket_receive_queue_tail;
    ULONG       nx_tcp_socket_transmit_queue_maximum;
    ULONG       nx_tcp_socket_transmit_sent_count;
    NX_PACKET *nx_tcp_socket_transmit_sent_head,
                *nx_tcp_socket_transmit_sent_tail;
    ULONG       nx_tcp_socket_timeout;
    ULONG       nx_tcp_socket_timeout_rate;
    ULONG       nx_tcp_socket_timeout_retries;
    ULONG       nx_tcp_socket_timeout_max_retries;
    ULONG       nx_tcp_socket_timeout_shift;
#ifdef NX_TCP_ENABLE_WINDOW_SCALING
    ULONG         nx_tcp_socket_rx_window_maximum;
    ULONG         nx_tcp_rcv_win_scale_value;
    ULONG         nx_tcp_snd_win_scale_value;
#endif
    ULONG       nx_tcp_socket_keepalive_timeout;
    ULONG       nx_tcp_socket_keepalive_retries;
    struct NX_TCP_SOCKET_STRUCT
                *nx_tcp_socket_bound_next,
                *nx_tcp_socket_bound_previous;
    TX_THREAD *nx_tcp_socket_bind_in_progress;
    TX_THREAD *nx_tcp_socket_receive_suspension_list;
    ULONG       nx_tcp_socket_receive_suspended_count;
    TX_THREAD *nx_tcp_socket_transmit_suspension_list;
    ULONG       nx_tcp_socket_transmit_suspended_count;
    TX_THREAD *nx_tcp_socket_connect_suspended_thread;
    TX_THREAD *nx_tcp_socket_disconnect_suspended_thread;
    TX_THREAD *nx_tcp_socket_bind_suspension_list;
    ULONG       nx_tcp_socket_bind_suspended_count;
    struct NX_TCP_SOCKET_STRUCT
                                             *nx_tcp_socket_created_next,
                                             *nx_tcp_socket_created_previous;
    VOID (*nx_tcp_urgent_data_callback)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    VOID (*nx_tcp_disconnect_callback)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    VOID (*nx_tcp_receive_callback)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    VOID (*nx_tcp_socket_window_update_notify)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    void    *nx_tcp_socket_reserved_ptr;
    ULONG   nx_tcp_socket_transmit_queue_maximum_default;

#ifdef FEATURE_NX_IPV6
    NXD_IPV6_ADDRESS
*nx_tcp_socket_outgoing_interface;
#endif /* FEATURE_NX_IPV6 */
} NX_TCP_SOCKET;

typedef struct NX_UDP_SOCKET_STRUCT
{
    ULONG       nx_udp_socket_id;
    CHAR        *nx_udp_socket_name;
    UINT        nx_udp_socket_port;
    struct NX_IP_STRUCT    *nx_udp_socket_ip_ptr;
    ULONG       nx_udp_socket_packets_sent;
    ULONG       nx_udp_socket_bytes_sent;
    ULONG       nx_udp_socket_packets_received;
    ULONG       nx_udp_socket_bytes_received;
    ULONG       nx_udp_socket_invalid_packets;
    ULONG       nx_udp_socket_packets_dropped;
    ULONG       nx_udp_socket_checksum_errors;
```

```
    ULONG       nx_udp_socket_type_of_service;
    UINT        nx_udp_socket_time_to_live;
    ULONG       nx_udp_socket_fragment_enable;
    UINT        nx_udp_socket_disable_checksum;
    ULONG       nx_udp_socket_receive_count;
    ULONG       nx_udp_socket_queue_maximum;
    NX_PACKET *nx_udp_socket_receive_head,
                *nx_udp_socket_receive_tail;
    struct NX_UDP_SOCKET_STRUCT
                *nx_udp_socket_bound_next,
                *nx_udp_socket_bound_previous;
    TX_THREAD *nx_udp_socket_bind_in_progress;
    TX_THREAD *nx_udp_socket_receive_suspension_list;
    ULONG       nx_udp_socket_receive_suspended_count;
    TX_THREAD *nx_udp_socket_bind_suspension_list;
    ULONG       nx_udp_socket_bind_suspended_count;
    struct NX_UDP_SOCKET_STRUCT
                *nx_udp_socket_created_next,
                *nx_udp_socket_created_previous;
    VOID (*nx_udp_receive_callback)(struct NX_UDP_SOCKET_STRUCT *socket_ptr);
    void        *nx_udp_socket_reserved_ptr;
#ifdef FEATURE_NX_IPV6
    NXD_IPV6_ADDRESS
*nx_udp_socket_outgoing_interface;
#endif /* FEATURE_NX_IPV6 */
} NX_UDP_SOCKET;

#ifdef FEATURE_NX_IPV6
typedef struct NXD_IPV6_ADDRESS_STRUCT
{
    UCHAR  nxd_interface_address_valid;
    UCHAR  nxd_interface_address_index;
    UCHAR  nxd_interface_address_type;
    UCHAR  nxd_interface_address_reserved;
    struct     NX_INTERFACE_STRUCT *nxd_interface_address_attached;
    ULONG  nxd_interface_ipv6_address[4];
    CHAR    nxd_interface_ipv6_address_prefix_length;
    CHAR    nxd_interface_ipv6_address_state;
    CHAR    nxd_interface_ipv6_address_DupAddrDetectTransmit;
    CHAR    nxd_interface_ipv6_address_ConfigurationMethod;
    struct     NXD_IPV6_ADDRESS_STRUCT *nxd_interface_ipv6_address_next;
} NXD_IPV6_ADDRESS;
#endif


typedef struct NXD_ADDRESS_STRUCT
{
    ULONG                                    nxd_ip_version;
    union
    {
        ULONG                                v4;
#ifdef FEATURE_NX_IPV6
        ULONG                                v6[4];
#endif
    } nxd_ip_address;
} NXD_ADDRESS;
```

APPENDIX D

ASCII Character Codes

ASCII Character Codes in HEX

most significant nibble

least signigicant nibble

	0_	1_	2_	3_	4_	5_	6_	7_
_0	NUL	DLE	SP	0	@	P	'	p
_1	SOH	DC1	!	1	A	Q	a	q
_2	STX	DC2	"	2	B	R	b	r
_3	ETX	DC3	#	3	C	S	c	s
_4	EOT	DC4	$	4	D	T	d	t
_5	ENQ	NAK	%	5	E	U	e	u
_6	ACK	SYN	&	6	F	V	f	v
_7	BEL	ETB	'	7	G	W	g	w
_8	BS	CAN	(	8	H	X	h	x
_9	HT	EM	)	9	I	Y	i	y
_A	LF	SUB	*	:	J	Z	j	z
_B	VT	ESC	+	;	K	[	K	}
_C	FF	FS	,	<	L	\	l	|
_D	CR	GS	-	=	M	]	m	}
_E	SO	RS	.	>	N	^	n	~
_F	SI	US	/	?	O	_	o	DEL

Index

B

C

D

E

F

N

Q

R

U

V

W

Z