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

User Guide
Version 5

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

About This Guide 9
- Guide Conventions 10
- NetX Data Types 11
- Customer Support Center 12

1 Introduction to NetX 15
- NetX Unique Features 16
- RFCs Supported by NetX 18
- Embedded Network Applications 19
- NetX Benefits 19

2 Installation and Use of NetX 23
- Host Considerations 24
- Target Considerations 25
- Product Distribution 25
- NetX Installation 26
- Using NetX 26
- Troubleshooting 27
- Configuration Options 28
- NetX Version ID 37
3 Functional Components of NetX 39
- Execution Overview 42
- Protocol Layering 48
- Packet Pools 49
- IP Protocol 57
- Address Resolution Protocol (ARP) in IP 69
- Reverse Address Resolution Protocol (RARP) in IP 75
- Internet Control Message Protocol (ICMP) 77
- Internet Group Management Protocol (IGMP) 80
- User Datagram Protocol (UDP) 84
- Transmission Control Protocol (TCP) 91

4 Description of NetX Services 107

5 NetX Network Drivers 353
- Driver Introduction 354
- Driver Entry 354
- Driver Requests 355
- Driver Output 371
- Driver Input 372
- Example RAM Ethernet Network Driver 374
A NetX Services 377

B NetX Constants 387

C NetX Data Types 407

D BSD-Compatible Socket API 415

E ASCII Character Codes 419

Index 421
# Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Protocol Layering</td>
<td>50</td>
</tr>
<tr>
<td>Figure 2</td>
<td>UDP Data Encapsulation</td>
<td>50</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Packet Header and Packet Pool Layout</td>
<td>53</td>
</tr>
<tr>
<td>Figure 4</td>
<td>IP Address Structure</td>
<td>58</td>
</tr>
<tr>
<td>Figure 5</td>
<td>IP Header Format</td>
<td>60</td>
</tr>
<tr>
<td>Figure 6</td>
<td>ARP Packet Format</td>
<td>73</td>
</tr>
<tr>
<td>Figure 7</td>
<td>ICMP Ping Message</td>
<td>79</td>
</tr>
<tr>
<td>Figure 8</td>
<td>IGMP Report Message</td>
<td>83</td>
</tr>
<tr>
<td>Figure 9</td>
<td>UDP Header</td>
<td>85</td>
</tr>
<tr>
<td>Figure 10</td>
<td>TCP Header</td>
<td>92</td>
</tr>
<tr>
<td>Figure 11</td>
<td>States of the TCP State Machine</td>
<td>97</td>
</tr>
</tbody>
</table>
About This Guide

This guide contains comprehensive information about NetX, the high-performance network stack from Express Logic.

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

Chapter 2  Gives the basic steps to install and use NetX with your ThreadX application.

Chapter 3  Provides a functional overview of the NetX system and basic information about the TCP/IP networking standards.

Chapter 4  Details the application’s interface to NetX.

Chapter 5  Describes network drivers for NetX.

Appendix A  NetX Services

Appendix B  NetX Constants

Appendix C  NetX 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 Data Types

In addition to the custom NetX control structure data types, there are several special data types that are used in NetX 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 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 source. They are located in either the `tx_port.h` or `nx_port.h` files.
Customer Support Center

<table>
<thead>
<tr>
<th>Support engineers</th>
<th>858.613.6640</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support fax</td>
<td>858.521.4259</td>
</tr>
<tr>
<td>Support email</td>
<td><a href="mailto:support@expresslogic.com">support@expresslogic.com</a></td>
</tr>
<tr>
<td>Web page</td>
<td><a href="http://www.expresslogic.com">http://www.expresslogic.com</a></td>
</tr>
</tbody>
</table>

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

5. A trace buffer captured immediately after the problem was detected. This is accomplished by building the ThreadX and NetX 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 User Guide” in the subject line.
NetX 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 and a description of its applications and benefits.

- **NetX Unique Features**
  - Piconet™ Architecture 16
  - Zero-copy Implementation 16
  - UDP Fast Path™ Technology 17
  - ANSI C Source Code 17
  - Not A Black Box 17
  - BSD-Compatibility Socket API 18

- **RFCs Supported by NetX** 18

- **Embedded Network Applications**
  - Real-time Network Software 19

- **NetX Benefits**
  - Improved Responsiveness 19
  - Software Maintenance 20
  - Increased Throughput 20
  - Processor Isolation 20
  - Ease of Use 20
  - Improve Time to Market 20

- **Protecting the Software Investment** 21
NetX Unique Features

Unlike other TCP/IP implementations, NetX 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 is Piconet, a software architecture especially designed for embedded systems. Piconet architecture maximizes scalability by implementing NetX 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 is completely determined by the application. For most applications, the instruction image requirements of NetX ranges between 5 KBytes and 30 KBytes in size.

NetX achieves superior network performance by layering internal component function calls only when it is absolutely necessary. In addition, much of NetX 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 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. 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 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 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 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 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 include the complete C source code. This eliminates the “black-box” problems that occur with many commercial network stacks. By using NetX, 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 is the ultimate network software for such developers.

**BSD-Compatible Socket API**

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

**RFCs Supported by NetX**

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

<table>
<thead>
<tr>
<th>RFC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFC 1112</td>
<td>Host Extensions for IP Multicasting (IGMPv1)</td>
</tr>
<tr>
<td>RFC 1122</td>
<td>Requirements for Internet Hosts - Communication Layers</td>
</tr>
<tr>
<td>RFC 2236</td>
<td>Internet Group Management Protocol, Version 2</td>
</tr>
<tr>
<td>RFC 768</td>
<td>User Datagram Protocol (UDP)</td>
</tr>
<tr>
<td>RFC 791</td>
<td>Internet Protocol (IP)</td>
</tr>
<tr>
<td>RFC 792</td>
<td>Internet Control Message Protocol (ICMP)</td>
</tr>
<tr>
<td>RFC 793</td>
<td>Transmission Control Protocol (TCP)</td>
</tr>
<tr>
<td>RFC 826</td>
<td>Ethernet Address Resolution Protocol (ARP)</td>
</tr>
<tr>
<td>RFC 903</td>
<td>Reverse Address Resolution Protocol (RARP)</td>
</tr>
</tbody>
</table>
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 Benefits

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

Improved Responsiveness

The high-performance NetX 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 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 provides the highest-performance networking available, which is achieved by minimal packet processing overhead. This also enables increased throughput.

**Processor Isolation**

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

**Improve Time to Market**

The powerful features of NetX accelerate the software development process. NetX 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!
NetX is written exclusively in ANSI C and is fully integrated with the ThreadX real-time operating system. This means NetX 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 ensures the application’s migration path and protects the original development investment.
Installation and Use of NetX

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

- Host Considerations 24
- Target Considerations 25
- Product Distribution 25
- NetX Installation 26
- Using NetX 26
- Troubleshooting 27
- Configuration Options 28
  - System Configuration Options 28
  - ARP Configuration Options 30
  - ICMP Configuration Options 31
  - IGMP Configuration Options 31
  - IP Configuration Options 31
  - Packet Configuration Options 32
  - RARP Configuration Options 33
  - TCP Configuration Options 33
  - UDP Configuration Options 37
- NetX Version ID 37
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 is delivered in ASCII format and requires approximately 1 Mbytes of space on the host computer’s hard disk.

Review the supplied readme_netx.txt file for additional host system considerations and options.
Target Considerations

NetX 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 thread stack and other global data structures.

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

Product Distribution

The exact content of the distribution disk depends on the target processor, development tools, and the NetX package purchased. However, the following is a list of several important files that are common to most product distributions:

**NetX_Express_Startup.pdf**
PDF that provides a simple, four-step procedure to get NetX running on a specific target processor/board and specific development tools.

**readme_netx.txt**
Text file containing specific information about the NetX port, including information about the target processor and the development tools.

**nx_api.h**
C header file containing all system equates, data structures, and service prototypes.
nx_port.h  C header file containing all development-tool and target-specific data definitions and structures.

demo_netx.c  C file containing a small demo application.

nx.a (or nx.lib)  Binary version of the NetX C library that is distributed with the standard package.

NetX Installation

Installation of NetX is straightforward. Refer to the NetX_Express_Startup.pdf file and the readme_netx.txt file for specific information on installing NetX for your specific environment.

Be sure to back up the NetX distribution disk and store it in a safe location.

Application software needs access to the NetX library file (usually 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

Using NetX is easy. Basically, the application code must include nx_api.h during compilation and link with the NetX library nx.a (or nx.lib).
The following are the four easy steps required to build a NetX application:

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

**Step 2:** Initialize the NetX 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 runtime library `nx.a` (or `nx.lib`). The resulting image can be downloaded to the target and executed!

---

**Troubleshooting**

Each NetX port is delivered with one or more demonstrations that execute on an actual network or via a simulated network driver. It is always a good idea to get the demonstration system running first.

See the `readme_netx.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 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 the network driver is supplying ARP and IP packets with its headers on 4-byte boundaries for applications requiring IP 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 library and the application using NetX. The configuration options 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 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. The following sections list the configuration options available in NetX.

System Configuration Options

- **NX_DEBUG**
  Defined, enables the optional print debug information available from the RAM Ethernet network driver.
NX_DEBUG_PACKET Defined, enables the optional debug packet dumping available in the RAM Ethernet network driver.

NX_DISABLE_ERROR_CHECKING Defined, removes the basic NetX error checking API and improves performance. 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.

NX_DRIVER_DEFERRED_PROCESSING Defined, enables deferred network driver packet handling. This allows the network driver to place a packet on the IP instance and have the real processing routine called from the NetX internal IP helper thread.

NX_ENABLE_EXTENDED_NOTIFY_SUPPORT Defined, enables more callback hooks in the stack. These callback functions are used by the BSD wrapper layer. By default this option is not defined.

NX_ENABLE_SOURCE_ADDRESS_CHECK Defined, enables the source address of incoming packet to be checked. By default this option is disabled.

NX_LITTLE_ENDIAN Defined, performs the necessary byte swapping on little endian environments to ensure the protocol headers are in proper big endian format. Note 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 header of the frame. 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, such as in nx_user.h.
ARP Configuration Options

NX_ARP_DEFEND_BY_REPLY
Defined, allows NetX to defend its IP address by sending an ARP response.

NX_ARP_DEFEND_INTERVAL
Defines the interval, in seconds, the ARP module sends out the next defend packet in response to an incoming ARP message that indicates an address in conflict.

NX_ARP_DISABLE_AUTO_ARP_ENTRY
Renamed to NX_DISABLE_ARP_AUTO_ENTRY. Although it is still being supported, new designs are encouraged to use NX_DISABLE_ARP_AUTO_ENTRY.

NX_ARP_EXPIRATION_RATE
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_MAC_CHANGE_NOTIFICATION_ENABLE
Renamed to NX_ENABLE_ARP_MAC_CHANGE_NOTIFICATION. Although it is still being supported, new designs are encouraged to use NX_ENABLE_ARP_MAC_CHANGE_NOTIFICATION.

NX_ARP_MAX_QUEUE_DEPTH
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
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
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.
NX_DISABLE_ARP_AUTO_ENTRY
Defined, disables entering ARP request information in the ARP cache.

NX_DISABLE_ARP_INFO
Defined, disables ARP information gathering.

NX_ENABLE_ARP_MAC_CHANGE_NOTIFICATION
Defined, allows ARP to invoke a callback notify function on detecting the MAC address is updated.

ICMP Configuration Options

NX_DISABLE_ICMP_INFO
Defined, disables ICMP information gathering.

NX_DISABLE_ICMP_RX_CHECKSUM
Defined, disables both ICMP checksum computation on received ICMP packets. This option is useful when the network interface driver is able to verify the ICMP checksum, and the application does not use the IP fragmentation feature. By default this option is not defined.

NX_DISABLE_ICMP_TX_CHECKSUM
Defined, disables both ICMP checksum computation on transmitted ICMP packets. This option is useful where the network interface driver is able to compute the ICMP checksum, and the application does not use the IP fragmentation feature. By default this option is not defined.

IGMP Configuration Options

NX_DISABLE_IGMP_INFO
Defined, disables IGMP information gathering.

NX_DISABLE_IGMPV2
Defined, disables IGMPv2 support, and NetX supports IGMPv1 only. By default this option is not set and is defined in nx_api.h.

NX_MAX_MULTICAST_GROUPS
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

NX_DISABLE_FRAGMENTATION
Defined, disables IP fragmentation and reassembly logic.
NX_DISABLE_IP_INFO Defined, disables IP information gathering.

NX_DISABLE_IP_RX_CHECKSUM Defined, disables checksum logic on received IP packets. This is useful if the network device is able to verify the IP header checksum, and the application does not expect to use IP fragmentation.

NX_DISABLE_IP_TX_CHECKSUM Defined, disables checksum logic on IP packets sent. This is useful in situations in which the underlying network device is capable of generating the IP header checksum, and the application does not expect to use IP fragmentation.

NX_DISABLE_LOOPBACK_INTERFACE Defined, disables NetX support for the loopback interface.

NX_DISABLE_RX_SIZE_CHECKING Defined, disables the size checking on received packets.

NX_ENABLE_IP_STATIC_ROUTING Defined, enables IP static routing in which a destination address can be assigned a specific next hop address. By default IP static routing is disabled.

NX_IP_PERIODIC_RATE Defined, specifies the number of ThreadX timer ticks in one second. The default value is derived from the ThreadX symbol TX_TIMER_TICKS_PER_SECOND, which by default is set to 100 (10ms timer). Applications shall exercise caution when modifying this value, as the rest of the NetX modules derive timing information from NX_IP_PERIODIC_RATE.

NX_IP_ROUTING_TABLE_SIZE Defined, sets the maximum number of entries in the IP static 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. This symbol is used only if NX_ENABLE_IP_STATIC_ROUTING is defined.

Packet Configuration Options
NX_DISABLE_PACKET_INFO
Defined, disables packet pool information gathering.

NX_PACKET_HEADER_PAD
Defined, enables padding towards the end of the
NX_PACKET control block. The number of ULONG words to pad is defined by `NX_PACKET_HEADER_PAD_SIZE`.

NX_PACKET_HEADER_PAD_SIZE
Sets the number of ULONG words to be padded to the
NX_PACKET structure, allowing the packet payload area to start at the desired alignment. This feature is useful when receive buffer descriptors point directly into NX_PACKET payload area, and the network interface receive logic or the cache operation logic expects the buffer starting address to meet certain alignment requirements. This value becomes valid only when `NX_PACKET_HEADER_PAD` is defined.

RARP Configuration Options

NX_DISABLE_RARP_INFO Defined, disables RARP information gathering.

TCP Configuration Options

NX_DISABLE_RESET_DISCONNECT
Defined, disables the reset processing during disconnect when the timeout value supplied is specified as `NX_NO_WAIT`.

NX_DISABLE_TCP_INFO Defined, disables TCP information gathering.

NX_DISABLE_TCP_RX_CHECKSUM
Defined, disables checksum logic on received TCP packets. This is only useful in situations in which the link-layer has reliable checksum or CRC processing, or the interface driver is able to verify TCP checksum in hardware.

NX_DISABLE_TCP_TX_CHECKSUM
Defined, 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_ENABLE_TCP_KEEPALIVE**
Defined, enables the optional TCP keepalive timer. The default settings is not enabled.

**NX_ENABLE_TCP_MSS_CHECKING**
Defined, enables the verification of minimum peer MSS before accepting a TCP connection. To use this feature, the symbol **NX_ENABLE_TCP_MSS_MINIMUM** must be defined. By default, this option is not enabled.

**NX_ENABLE_TCP_WINDOW_SCALING**
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_MAX_LISTEN_REQUESTS**
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**
Specifies the number of TCP packets to receive before sending an ACK. 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**
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**
Renamed to **NX_ENABLE_TCP_KEEPALIVE**. Although it is still being supported, new designs are encouraged to use **NX_ENABLE_TCP_KEEPALIVE**.
NX_TCP_ENABLE_WINDOW_SCALING

Renamed to **NX_ENABLE_TCP_WINDOW_SCALING**. Although it is still being supported, new designs are encouraged to use **NX_ENABLE_TCP_WINDOW_SCALING**.

NX_TCP_FAST_TIMER_RATE

Specifies how the number of NetX internal 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 assuming the ThreadX timer is running at 10ms. This value 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, 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

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_RETRIES

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_KEEPALIVE_RETRY

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_MAX_OUT_OF_ORDER_PACKETS

Symbol that defines the maximum number of out-of-order TCP packets can be kept in the TCP socket receive queue. This symbol can be used to limit the number of packets queued in
the TCP receive socket, preventing the packet pool from being starved. By default this symbol is not defined, thus there is no limit on the number of out of order packets being queued in the TCP socket.

**NX_TCP_MAXIMUM_RETRIES**

Specifies how many data 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**

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 packets stay in the transmit queue until an ACK that covers some or all of the packet data 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_MSS_CHECKING_ENABLED**

Renamed to `NX_ENABLE_TCP_MSS_CHECKING`. Although it is still being supported, new designs are encouraged to use `NX_ENABLE_TCP_MSS_CHECKING`.

**NX_TCP_MSS_MINIMUM**

Symbol that defines the minimal MSS value NetX TCP module accepts. This feature is enabled by `NX_ENABLE_TCP_MSS_CHECK`.

**NX_TCP_RETRY_SHIFT**

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**

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**

- **NX_DISABLE_UDP_INFO**
  Defined, disables UDP information gathering.

- **NX_DISABLE_UDP_RX_CHECKSUM**
  Defined, disables the UDP checksum computation on incoming UDP packets. This is useful if the network interface driver is able to verify UDP header checksum in hardware, and the application does not enable IP fragmentation logic.

- **NX_DISABLE_UDP_TX_CHECKSUM**
  Defined, disables the UDP checksum computation on outgoing UDP packets. This is useful if the network interface driver is able to compute UDP header checksum and insert the value in the IP head before transmitting the data, and the application does not enable IP fragmentation logic.

**NetX Version ID**

The current version of NetX is available to both the user and the application software during runtime. The programmer can find the NetX version in the `readme_netx_generic.txt` file. This file also contains a version history of the corresponding port. Application software can obtain the NetX 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.

```c
#define __PRODUCT_NETX__
#define __NETX_MAJOR_VERSION__
#define __NETX_MINOR_VERSION__
```
Functional Components of NetX

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

- Execution Overview 42
  - Initialization 42
  - Application Interface Calls 43
  - Internal IP Thread 44
  - IP Periodic Timers 45
  - Network Driver 45
  - Multihome Support 46
  - Loopback Interface 48
  - Interface Control Blocks 48

- Protocol Layering 48

- Packet Pools 49
  - Packet Pool Memory Area 52
  - Creating Packet Pools 52
  - Packet Header NX_PACKET 52
  - Packet Header Offsets 55
  - Pool Capacity 56
  - Thread Suspension 56
  - Pool Statistics and Errors 56
  - Packet Pool Control Block NX_PACKET_POOL 57

- IP Protocol 57
  - IP Addresses 58
  - IP Gateway_Address 59
  - IP Header 60
  - Creating IP Instances 62
  - IP Send 63
  - IP Receive 64
  - Raw IP Send 65
  - Raw IP Receive 65
  - Default Packet Pool 66
  - IP Helper Thread 66
Thread Suspension 66
IP Statistics and Errors 67
IP Control Block NX_IP 67
Static IP Routing 67
IP Fragmentation 68

- Address Resolution Protocol (ARP) in IP 69
  ARP Enable 69
  ARP Cache 70
  ARP Dynamic Entries 70
  ARP Static Entries 70
  Automatic ARP Entry 71
  ARP Messages 71
  ARP Aging 74
  ARP Defend 74
  ARP Statistics and Errors 74

- Reverse Address Resolution Protocol (RARP) in IP 75
  RARP Enable 75
  RARP Request 76
  RARP Reply 76
  RARP Statistics and Errors 77

- Internet Control Message Protocol (ICMP) 77
  ICMP Statistics and Errors 78
  ICMP Enable 78
  ICMP Echo Request 78
  ICMP Echo Response 80

- Internet Group Management Protocol (IGMP) 80
  IGMP Enable 81
  Multicast IP Addressing 81
  Physical Address Mapping in IP 81
  Multicast Group Join 81
  Multicast Group Leave 82
  Multicast Loopback 82
  IGMP Report Message 82
  IGMP Statistics and Errors 84

- User Datagram Protocol (UDP) 84
  UDP Header 85
  UDP Enable 86
  UDP Socket Create 86
  UDP Checksum 86
UDP Ports and Binding 87
UDP Fast Path™ 87
UDP Packet Send 88
UDP Packet Receive 89
UDP Receive Notify 89
Peer Address and Port 89
Thread Suspension 90
UDP Socket Statistics and Errors 90
UDP Socket Control Block NX_UDP_SOCKET 91

Transmission Control Protocol (TCP) 91
TCP Header 91
TCP Enable 94
TCP Socket Create 94
TCP Checksum 94
TCP Port 95
Client-Server Model 95
TCP Socket State Machine 96
TCP Client Connection 96
TCP Client Disconnection 96
TCP Server Connection 99
TCP Server Disconnection 100
MSS Validation 101
Stop Listening on a Server Port 102
TCP Window Size 102
TCP Packet Send 102
TCP Packet Retransmit 103
TCP Keepalive 103
TCP Packet Receive 104
TCP Receive Notify 104
Thread Suspension 104
TCP Socket Statistics and Errors 105

TCP Socket Control Block NX_TCP_SOCKET 106
Execution Overview

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

*NetX 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 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 packet pool must exist prior to creating an IP instance. Creating packet pools and IP instances are 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 network driver, supplied during the first part of IP creation, is first called. Calling the network driver from the internal IP thread enables the 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.
The NetX service *nx_ip_status_check* is available to obtain information on the IP instance and its 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 systems, see “Multihome Support” on page 46.

*nx_ip_interface_status_check* is available to obtain information on the specified interface.

**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 service can be called.

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, on transmission, the UDP checksum calculation is performed inside the *nx_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, executed in the context of the application thread. This helps prevent stalling network requests of higher-priority threads because of processing intensive checksum computation in lower-priority threads.

Values, such as IP addresses and port numbers, are passed to APIs in host byte order. Internally these values are stored in host byte order as well. This allows developers to easily view the values via a debugger. When these values are programmed into a
frame for transmission, they are converted to network byte order.

**Internal IP Thread**

As mentioned, each IP instance in NetX 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 internal 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 attach the network driver to the IP instance, followed by an initialization call, which allows the network driver to go through the initialization process. 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 the IP network.

*The NetX callback functions, including listen and disconnect callbacks, are called from the internal IP thread—not the original calling thread.*

![](image)
application must take care not to suspend inside any NetX 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 has a primary interface, which is identified by its device driver specified in the `nx_ip_create` service. The network driver is responsible for handling various NetX requests, including packet transmission, packet reception, and requests for status and control.

For a multi-home system, the IP instance has multiple interfaces, each with an associated network driver that performs these tasks for the respective 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 provides the network driver with several access functions to handle various events. These functions are designed to be called from the interrupt service routine portion of the network driver. For IP networks, the network driver should forward all ARP packets received to the `_nx_arp_packet_deferred_receive` internal function. All RARP packets should be forwarded to `_nx_rarp_packet_deferred_receive` internal function. 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 performance in handling IP 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. In this mode, the ISR shall save the necessary information, call the internal function `_nx_ip_driver_deferred_processing`, and acknowledge the interrupt controller. This service notifies IP thread to schedule a callback to the device driver to complete the process of the event that causes the interrupt.

Some network controllers are capable of performing TCP/IP header checksum computation and validation in hardware, without taking up valuable CPU resources. To take advantage of the hardware capability feature, NetX provides options to enable or disable various software checksum computation at compilation time, as well as turning on or off checksum computation at run time. See “NetX Network Drivers” on page 353 for more detailed information on writing NetX network drivers.

**Multihome Support**

NetX supports systems connected to multiple physical devices using a single IP instance. Each physical interface is assigned to an interface control block in the IP instance. Applications wishing to use a multihome system must define the value for `NX_MAX_PHYSICAL_INTERFACES` to the number of physical devices attached to the system, and rebuild NetX library. By default
**NX_MAX_PHYSICAL_INTERFACES** is set to one, creating one interface control block in the IP instance.

The NetX application creates a single IP instance for the primary device using the `nx_ip_create` service. For each additional network device, the application attaches the device to the IP instance using the `nx_ip_interface_attach` service.

Each network interface structure contains a subset of network information about the network interface that is contained in the IP control block, including interface IP address, subnet mask, IP MTU size, and MAC-layer address information.

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

The primary interface has index zero in the IP instance list. Each subsequent device 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, ICMP, and IGMP, are available to all the attached devices.

In most cases, NetX can determine the best source address to use when transmitting a packet. The source address selection is based on the destination address. NetX services are added to allow applications to specify a specific source address to use, in cases where the most suitable one cannot be determined by the destination address. An example would be in a multihome system, an application needs to send a packet to an IP broadcast or multicast destination addresses.

Services specifically for developing multihome applications include the following:
These services are explained in greater detail in “Description of NetX Services” on page 107.

**Loopback Interface**

The loopback interface is a special network interface without an physical link attached to. The loopback interface allows applications to communicate using the IP loopback address 127.0.0.1

To utilize a logical loopback interface, ensure the configurable option `NX_DISABLE_LOOPBACK_INTERFACE` is not set.

**Interface Control Blocks**

The number of interface control blocks in the IP instance is the number of physical interfaces (defined by `NX_MAX_PHYSICAL_INTERFACES`) plus the loopback interface if it is enabled. The total number of interfaces is defined in `NX_MAX_IP_INTERFACES`.

**Protocol Layering**

The TCP/IP implemented by NetX 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: 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 layering and Figure 2 shows the resulting data encapsulation for UDP data being sent.

### Packet Pools

Allocating packets in a fast and deterministic manner is always a challenge in real-time networking applications. With this in mind, NetX provides the ability to create and manage multiple pools of fixed-size network packets.
FIGURE 1. Protocol Layering

FIGURE 2. UDP Data Encapsulation
Because NetX packet pools consist of fixed-size memory blocks, there are never any internal fragmentation problems. Of course, fragmentation causes behavior that is inherently nondeterministic. In addition, the time required to allocate and free a NetX 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 packets address this problem with an optional feature called packet chaining. An actual network packet can be made of one or more NetX 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.

Since each packet payload size is fixed for a given packet pool, application data larger than the payload size would require multiple packets chained together. When filling a packet with user data, the application shall use the service `nx_packet_data_append`. This service moves application data into a packet. In situations where a packet is not enough to hold user data, additional packets are allocated to store user data. To use packet chaining, the driver must be able to receive into or transmit from chained packets.

Each NetX packet memory pool is a public resource. NetX places no constraints on how packet pools are used.
Packet Pool Memory Area

The memory area for the packet pool is specified during creation. Like other memory areas for ThreadX and NetX 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 packet memory pool.

Creating Packet Pools

Packet 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 application.

Packet Header

NX_PACKET

By default, NetX 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.

For network devices driver that are able to perform zero copy operations, typically the starting address of the packet payload area is programmed into the DMA logic. Certain DMA engines have alignment requirement on the payload area.

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.
FIGURE 3. Packet Header and Packet Pool Layout
The fields of the packet header are defined as follows. Note that this table is not a comprehensive list of all the members in the `NX_PACKET` structure.

<table>
<thead>
<tr>
<th>Packet header</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx_packet_pool_owner</code></td>
<td>This field points to the packet pool that owns 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.</td>
</tr>
<tr>
<td><code>nx_packet_next</code></td>
<td>This field points to the next packet within the same frame. If NULL, there are no additional packets that are part of the frame.</td>
</tr>
<tr>
<td><code>nx_packet_last</code></td>
<td>This field points to the last packet within the same network packet. If NULL, this packet represents the entire network packet.</td>
</tr>
<tr>
<td><code>nx_packet_length</code></td>
<td>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 <code>nx_packet_next</code> member.</td>
</tr>
<tr>
<td><code>nx_packet_ip_interface</code></td>
<td>This field is the interface control block which is assigned to the packet when it is received by the interface driver, and by NetX 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.</td>
</tr>
<tr>
<td><code>nx_packet_data_start</code></td>
<td>This field points to the start of the physical payload area of this packet. It does not have to be immediately following the <code>NX_PACKET</code> header, but that is the default for the <code>nx_packet_pool_create</code> service.</td>
</tr>
<tr>
<td><code>nx_packet_data_end</code></td>
<td>This field points to the end of the physical payload area of this packet. The difference between this field and the <code>nx_packet_data_start</code> field represents the payload size.</td>
</tr>
</tbody>
</table>
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 the offset required for inserting the protocol header (such as UDP, TCP, or ICMP) in front of the protocol data.

The following types are defined in NetX 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. IP packets are still defined in NetX for applications to allocate packets for IP networks. The following table shows symbols defined:

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_IP_PACKET</td>
<td>0x24</td>
</tr>
<tr>
<td>NX_UDP_PACKET</td>
<td>0x2c</td>
</tr>
<tr>
<td>NX_TCP_PACKET</td>
<td>0x38</td>
</tr>
</tbody>
</table>

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 proper alignment) into the total number of bytes in the supplied memory area.

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 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
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 library unless \texttt{NX\_DISABLE\_PACKET\_INFO} is defined. This data is available to the application with the \texttt{nx\_packet\_pool\_info\_get} service.

Packet Pool
Control Block
\texttt{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 \texttt{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.

IP Protocol

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

NetX supports IP protocol (RFC 791)
IP Addresses

Each host on the Internet has a unique 32-bit identifier called an IP address. There are five classes of IP addresses as described in Figure 4. The ranges of the five IP address classes are as follows:

<table>
<thead>
<tr>
<th>Class</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.0.0.0 to 127.255.255.255</td>
</tr>
<tr>
<td>B</td>
<td>128.0.0.0 to 191.255.255.255</td>
</tr>
<tr>
<td>C</td>
<td>192.0.0.0 to 223.255.255.255</td>
</tr>
<tr>
<td>D</td>
<td>224.0.0.0 to 239.255.255.255</td>
</tr>
<tr>
<td>E</td>
<td>240.0.0.0 to 247.255.255.255</td>
</tr>
</tbody>
</table>

**FIGURE 4. IP Address Structure**
There are also three types of address specifications: *unicast*, *broadcast*, 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. 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.

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

*The macro IP_ADDRESS is defined in nx_api.h. It allows easy specification of IP addresses using commas instead of periods. For example, IP_ADDRESS(128,0,0,0) specifies the first class B address shown in Figure 4.*

**IP Gateway Address**

Network gateways assist hosts on their networks to relay packets destined to destinations outside the local domain. Each node has some knowledge of which next hop to send to, either the destination one of its neighbors, or through a pre-programmed static routing table. However if these approaches fail, the node should forward the packet to its default gateway which has better knowledge on how to route the packet to its destination. Note that the default gateway must be directly accessible through one of the physical interfaces attached to the IP instance. The application calls `nx_ip_gateway_address_set` to configure IP default gateway address.
IP Header

For any IP packet to be sent on the Internet, it must have an IP header. When higher-level protocols (UDP, TCP, ICMP, or IGMP) call the IP component to send a packet, the IP transmit module places an IP header in front of the data. 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 5 shows the format of the IP 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. 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 IP header are defined as follows:

<table>
<thead>
<tr>
<th>IP Header Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit version</td>
<td>This field contains the version of IP this header represents. For IP version 4, which is what NetX supports, the value of this field is 4.</td>
</tr>
<tr>
<td>4-bit header length</td>
<td>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.</td>
</tr>
<tr>
<td>8-bit type of service (TOS)</td>
<td>This field specifies the type of service requested for this IP packet. Valid requests are as follows:</td>
</tr>
<tr>
<td></td>
<td><strong>TOS Request</strong></td>
</tr>
<tr>
<td></td>
<td>Normal</td>
</tr>
<tr>
<td></td>
<td>Minimum Delay</td>
</tr>
<tr>
<td></td>
<td>Maximum Data</td>
</tr>
<tr>
<td></td>
<td>Maximum Reliability</td>
</tr>
<tr>
<td></td>
<td>Minimum Cost</td>
</tr>
<tr>
<td>16-bit total length</td>
<td>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.</td>
</tr>
<tr>
<td>16-bit identification</td>
<td>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.</td>
</tr>
<tr>
<td>3-bit flags</td>
<td>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.</td>
</tr>
</tbody>
</table>
IP Header Field | Purpose
--- | ---
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:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICMP</td>
<td>0x01</td>
</tr>
<tr>
<td>IGMP</td>
<td>0x02</td>
</tr>
<tr>
<td>TCP</td>
<td>0x06</td>
</tr>
<tr>
<td>UDP</td>
<td>0x11</td>
</tr>
</tbody>
</table>

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.

Creating IP Instances

IP instances are created either during initialization or during runtime by application threads. The initial IP 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. If the application initializes the IP instance with its IP address set to an invalid address(0.0.0.0), it is assumed that the interface address is going to
resolved by manual configuration later, via RARP, or through DHCP or similar protocols.

For systems with multiple network interfaces, the primary interface is designated when calling `nx_ip_create`. Each additional interface can be attached to the same IP instance by calling `nx_ip_interface_attach`. This service stores information about the network interface (such as IP address, network mask) in the interface control block, and associates the driver instance with the interface control block in the IP instance. As the driver receives a data packet, it needs to store the interface information in the NX_PACKET structure before forwarding it to the IP receive logic. Note an IP instance must already be created before attaching any interfaces.

**IP Send**

The IP send processing in NetX 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 in-line, and the packet is dispatched to the associated network driver. In addition, outgoing fragmentation is also coordinated from within the IP send processing.

For IP, NetX initiates ARP requests if physical mapping is needed for the destination IP address.

*For IP 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 (defined by the symbol `NX_ARP_MAX_QUEUE_DEPTH`). If the queue depth is reached, NetX will remove the oldest packet on the queue and continue waiting for address resolution for the remaining packets enqueued. On*
the other hand, if an ARP entry is not resolved, the pending packets on the ARP entry are released upon ARP entry timeout.

For systems with multiple network interfaces, NetX chooses an interface based on the destination IP address. The following procedure applies to the selection process:

1. If a destination address is IP broadcast or multicast, and if a valid outgoing interface is specified, use that interface. Otherwise, the first physical interface is used.
2. If the destination address is found in the static routing table, the interface associated with the gateway is used.
3. If the destination is on-link, the on-link interface is used.
4. If the destination address is a loopback address 127.0.0.1, the loopback interface is used.
5. If the default gateway is properly configured, use the interface associated with the default gateway to transmit the packet.
6. The output packet is dropped if all the above fails.

**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. If fragmentation is needed but not enabled, the packet is dropped.
NetX determines the appropriate network interface based on the interface specified in the packet. If the packet interface is NULL, NetX defaults to the primary interface. This is done to guarantee compatibility with legacy NetX Ethernet drivers.

**Raw IP Send**

A raw IP packet is an IP frame that contains upper layer protocol payload not directly supported (and processed) by NetX. A raw packet allows developers to define their own IP-based applications. An application may send raw IP packets directly using the `nx_ip_raw_packet_send` service if raw IP packet processing has been enabled with the `nx_ip_raw_packet_enabled` service. If the destination address is a multicast or broadcast address, however, NetX will default to the first (primary) interface. Therefore, to send such packets out on secondary interfaces, the application must use the `nx_ip_raw_packet_interface_send` service to specify the source address to use for the outgoing packet.

**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 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 incoming packet will be put into the raw packet queue waiting for the application to receive it via the `nx_ip_raw_packet_receive` service. In addition, application threads may suspend with an optional timeout while waiting for a raw IP packet.
**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 control packets (such as SYN, ACK). If the default packet pool is empty when NetX needs to allocate a packet, NetX may have to abort the particular operation, and will return 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 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 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

The characteristics of each IP instance are found in its control block. It contains useful information such as the IP addresses and network masks of each network device, and a table of neighbor IP and physical hardware address mapping. This structure is defined in the `nx_api.h` file.

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.

Static IP Routing

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

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 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 (longer prefix) in the network mask.

**IP Fragmentation**

The network device may have limits on the size of outgoing packets. This limit is called the maximum transmission unit (MTU). IP MTU is the largest IP frame size a link layer driver is able to transmit without fragmenting the IP packet. During a device driver initialization phase, the driver module must configure its IP MTU size via the service `nx_ip_interface_mtu_set`.

Although not recommended, the application may generate datagrams larger than the underlying IP MTU supported by the device. Before transmitting such IP datagram, the IP layer must fragment these packets. On receiving fragmented IP frames, the receiving end must store all fragmented IP frames with the same fragmentation ID, and reassemble them in order. If the IP receive logic is unable to
collect all the fragments to restore the original IP frame in time, all the fragments are released. It is up to the upper layer protocol to detect such packet loss and recover from it.

In order to support IP fragmentation and reassembly operation, the system designer must enable the IP fragmentation feature in NetX 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 library. Doing so helps reduce the code size of NetX.

### Address Resolution Protocol (ARP) in IP

The Address Resolution Protocol (ARP) is responsible for dynamically mapping 32-bit IP 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 network driver supplied to the `nx_ip_create` service. If physical mapping is required, the network driver must set the flag `nx_interface_address_mapping_needed` in the interface structure.

### 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.

Application can look up an IP address from the ARP cache by supplying hardware MAC address using the service `nx_arp_ip_address_find` if the mapping exists in the ARP table. Similarly, the service `nx_arp_hardware_address_find` returns the MAC address for a given IP address.

**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 NetX 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.

A dynamic entry can also be created by the service `nx_arp_dynamic_entry_set`.

> *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. The application can delete a static entry by using the service `nx_arp_static_entry_delete`. To remove all static entries in the ARP table, the application may use the service `nx_arp_static_entries_delete`.

### Automatic ARP Entry

NetX records the peer’s IP/MAC mapping after the peer responses to the ARP request. NetX also implements the automatic ARP entry feature where it records peer IP/MAC address mapping based on unsolicited ARP requests from the network. This feature allows the ARP table to be populated with peer information, reducing the delay needed to go through the ARP request/response cycle. However, the downside with enabling automatic ARP is that the ARP table tends to fill up quickly on a busy network with many nodes on the local link, which would eventually lead to ARP entry replacement.

This feature is enabled by default. To disable it, the NetX library must be compiled with the symbol `NX_DISABLE_ARP_AUTO_ENTRY` defined.

### 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 systems, NetX determines which interface to send the ARP requests and responses based on destination address specified.

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

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

The formats of Ethernet ARP requests and responses are shown in Figure 6 and are described below:

<table>
<thead>
<tr>
<th>Request/Response Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ethernet Destination Address</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Ethernet Source Address</strong></td>
<td>This 6-byte field contains the address of the sender of the ARP request or response and is set up by the network driver.</td>
</tr>
<tr>
<td><strong>Frame Type</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Hardware Type</strong></td>
<td>This 2-byte field contains the hardware type, which is 0x0001 for Ethernet.</td>
</tr>
<tr>
<td><strong>Protocol Type</strong></td>
<td>This 2-byte field contains the protocol type, which is 0x0800 for IP addresses.</td>
</tr>
<tr>
<td><strong>Hardware Size</strong></td>
<td>This 1-byte field contains the hardware address size, which is 6 for Ethernet addresses.</td>
</tr>
</tbody>
</table>
### Functional Components of NetX

**Request/Response Field**

- **Protocol Size**
  - Purpose: This 1-byte field contains the IP address size, which is 4 for IP addresses.

- **Operation Code**
  - Purpose: 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**
  - Purpose: This 6-byte field contains the sender’s Ethernet address.

- **Sender IP Address**
  - Purpose: This 4-byte field contains the sender’s IP address.

- **Target Ethernet Address**
  - Purpose: This 6-byte field contains the target’s Ethernet address.

- **Target IP Address**
  - Purpose: This 4-byte field contains the target’s IP address.

---

**FIGURE 6. ARP Packet Format**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ethernet Destination Address (6-bytes)</td>
</tr>
<tr>
<td>6</td>
<td>Ethernet Source Address (6-bytes)</td>
</tr>
<tr>
<td>12</td>
<td>Frame Type 0x0806</td>
</tr>
<tr>
<td>12</td>
<td>Hardware Type 0x0001</td>
</tr>
<tr>
<td>12</td>
<td>Protocol Type 0x0800</td>
</tr>
<tr>
<td>18</td>
<td>H Size 6</td>
</tr>
<tr>
<td>18</td>
<td>P Size 4</td>
</tr>
<tr>
<td>18</td>
<td>Operation (2-bytes)</td>
</tr>
<tr>
<td>22</td>
<td>Sender’s Ethernet Address (6-bytes)</td>
</tr>
<tr>
<td>28</td>
<td>Sender’s IP Address (4-bytes)</td>
</tr>
<tr>
<td>32</td>
<td>Target’s Ethernet Address (6-bytes)</td>
</tr>
<tr>
<td>38</td>
<td>Target’s IP Address (4-bytes)</td>
</tr>
</tbody>
</table>
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

NetX supports automatic dynamic ARP entry invalidation. \texttt{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. Setting \texttt{NX\_ARP\_EXPIRATION\_RATE} to zero disables ARP aging, which is the default configuration.

ARP Defend

When an ARP request or ARP response packet is received and the sender has the same IP address, which conflicts with the IP address of this node, NetX sends an ARP request for that address as a defense. If the conflict ARP packet is received more than once in 10 seconds, NetX does not send more defend packets. The default interval 10 seconds can be redefined by \texttt{NX\_ARP\_DEFEND\_INTERVAL}. This behavior follows the policy specified in 2.4(c) of RFC5227. Since Windows XP ignores ARP announcement as a response for its ARP probe, user can define \texttt{NX\_ARP\_DEFEND\_BY\_REPLY} to send ARP response as additional defence.

ARP Statistics and Errors

If enabled, the NetX 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 IP**

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 application creates an IP instance by the service `nx_ip_create` with a zero IP address. If RARP is enabled by the application, it can use the RARP protocol to request an IP address from the network server accessible through the interface that has 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 using the service `nx_rarp_enable`. For multihome systems, at least one network device associated with the IP instance must have an IP address of zero. The RARP processing periodically
sends RARP request messages for the NetX system requiring an IP address until a valid RARP reply with the network designated IP address is received. At this point, RARP processing is complete.

After RARP has been enabled, it is disabled automatically after all interface addresses are resolved. The application may force RARP to terminate by using the service `nx_rarp_disable`.

RARP Request

The format of an RARP request packet is almost identical to the ARP packet shown in Figure 6 on page 73. 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 6. 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 applications to wait for the RARP completion. For multihome systems, the application should not use the requesting interface until the RARP processing is complete on that interface. Status of the IP address on the secondary device can be checked with the `nx_ip_interface_status_check` service.

### RARP Statistics and Errors

If enabled, the NetX 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.

### Internet Control Message Protocol (ICMP)

Internet Control Message Protocol for IP (ICMP) is limited to passing error and control information between IP network members.

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

If enabled, NetX 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.

ICMP Enable

Before ICMP messages can be processed by NetX, the application must call the `nx_icmp_enable` service to enable ICMP processing. After this is done, the application can issue ping requests and field incoming ping packets.

ICMP Echo Request

An echo request is one type of ICMP message that is typically used to check for the existence of a specific node on the network, as identified by its host IP address. The popular ping command is implemented using ICMP echo request/echo reply messages. If the specific host is present, its network stack processes the
ping request and responses with a ping response. Figure 7 details the ICMP ping message format.

All ICMP 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 table describes the ICMP header format:

<table>
<thead>
<tr>
<th>Header Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>This field specifies the ICMP message (bits 31-24). The most common are: 0 Echo Reply 8 Echo Request</td>
</tr>
<tr>
<td>Code</td>
<td>This field is context specific on the type field (bits 23-16). For an echo request or reply the code is set to zero.</td>
</tr>
</tbody>
</table>
ICMP Echo 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.

Internet Group Management Protocol (IGMP)

The Internet Group Management Protocol (IGMP) provides a device to communicate with its neighbors and its routers that it intends to receive, or join, an IP 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, the application must call the `nx_igmp_enable` service. This service performs basic IGMP initialization in preparation for multicast requests.

Multicast IP Addressing

As mentioned previously, multicast addresses are actually Class D IP addresses as shown in Figure 4 on page 58. 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 IP

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 primary 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.

In a multihome system, if the multicast group is accessible via a specific interface, application shall use the service `nx_igmp_multicast_interface_join`
instead of `nx_igmp_multicast_join`, which is limited to multicast groups on the primary network.

**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.

**Multicast Loopback**

An application may wish to receive multicast traffic originated from one of the sources on the same node. This requires the IP multicast component to have loopback enabled by using the service `nx_igmp_loopback_enable`.

**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 8. 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 8), the IGMP header contains a version/type field, maximum response time, 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 host’s Report type message as defined by the IGMPv2 protocol.

The following describes the IGMP header format:

<table>
<thead>
<tr>
<th>Header Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>This field specifies the IGMP version (bits 31-28).</td>
</tr>
<tr>
<td>Type</td>
<td>This field specifies the type of IGMP message (bits 27-24).</td>
</tr>
<tr>
<td>Maximum Response Time</td>
<td>Not used in IGMPv1. In IGMPv2 this field serves as the maximum response time.</td>
</tr>
<tr>
<td>Checksum</td>
<td>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)</td>
</tr>
<tr>
<td>Group Address</td>
<td>32-bit class D group IP address</td>
</tr>
</tbody>
</table>

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 8. 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 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.
UDP Header

UDP places a simple packet header in front of the application’s data on transmission, and removes a similar UDP header from the packet on reception 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 9 shows the format of the 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:

<table>
<thead>
<tr>
<th>Header Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit source port number</td>
<td>This field contains the port on which the UDP packet is being sent from. Valid UDP ports range from 1 through 0xFFFF.</td>
</tr>
<tr>
<td>16-bit destination port number</td>
<td>This field contains the UDP port to which the packet is being sent to. Valid UDP ports range from 1 through 0xFFFF.</td>
</tr>
</tbody>
</table>
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 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.

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. 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.
On the IP network, UDP checksum is optional. NetX allows an 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` service.

Certain Ethernet controllers are able to generate the UDP checksum on the fly. If the system is able to use hardware checksum computation feature, the NetX library can be built without the checksum logic. To disable UDP software checksum, the NetX library must be built with the following symbols defined: `NX_DISABLE_UDP_TX_CHECKSUM` and `NX_DISABLE_UDP_RX_CHECKSUM` (described in Chapter two). The configuration options remove UDP checksum logic from NetX entirely, while calling the `nx_udp_socket_checksum_disable` service allows the application to disable IP 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, 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 UDP implementation. Sending a UDP packet requires just a few function calls: `nx_udp_socket_send`, `nx_ip_packet_send`, and the eventual call to the network driver. `nx_udp_socket_send` is available in NetX for existing NetX applications and is only applicable for IP packets. The preferred method, however, is to use `nx_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 IP networks is easily accomplished by calling the `nx_udp_socket_send` function. The caller must set the IP version in the `IP address` field. NetX will determine the best source address for transmitted UDP packets based on the destination IP address. 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 or broadcast destinations, the application should specify the source IP address to use if the NetX device has multiple IP addresses to choose from. This can be done with the services `nx_udp_socket_interface_send`.

*If `nx_udp_socket_send` is used for transmitting multicast or broadcast packets, the IP address of the first interface is used as source address.*

*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 payload 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*
Functional Components of NetX

pointer for NetX to place the UDP, IP, and physical media headers.

UDP Packet Receive

Application threads may receive UDP packets from a particular socket by calling
\texttt{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 \texttt{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.

Peer Address and Port

On receiving a UDP packet, application may find the sender’s IP address and port number by using the service \texttt{nx_udp_packet_info_extract}. On
successful return, this service provides information on the sender’s IP address, sender’s port number, and the local interface through which the packet was received.

**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 services.

**UDP Socket Statistics and Errors**

If enabled, the NetX 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.
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.

TCP Header

On transmission, TCP header is placed in front of the data from the user. On reception, TCP header is removed from the incoming packet, leaving only the user data available 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 10 shows the format of the TCP header.
The following describes the TCP header format:

<table>
<thead>
<tr>
<th>Header Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>16-bit source port number</strong></td>
<td>This field contains the port the TCP packet is being sent out on. Valid TCP ports range from 1 through 0xFFFF.</td>
</tr>
<tr>
<td><strong>16-bit destination port number</strong></td>
<td>This field contains the TCP port the packet is being sent to. Valid TCP ports range from 1 through 0xFFFF.</td>
</tr>
<tr>
<td><strong>32-bit sequence number</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
Header Field | Purpose
--- | ---
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.

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:

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

16-bit window | This field is used for flow control. It contains the amount of bytes the socket 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.
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 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 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.

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.

Certain network controllers are able to perform TCP checksum computation and validation in hardware.

<table>
<thead>
<tr>
<th>Header Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit TCP checksum</td>
<td>This field contains the 16-bit checksum for the packet including the TCP header, the packet data area, and the pseudo IP header.</td>
</tr>
<tr>
<td>16-bit urgent pointer</td>
<td>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.</td>
</tr>
</tbody>
</table>
For such systems, applications may want to use hardware checksum logic as much as possible to reduce runtime overhead. Applications may disable TCP checksum computation logic from the NetX library altogether at build time by defining `NX_DISABLE_TCP_TX_CHECKSUM` and `NX_DISABLE_TCP_RX_CHECKSUM`. This way, the TCP checksum code is not compiled in.

TCP Port

A TCP port is a logical connection point in the TCP protocol. There are 65,535 valid ports in the TCP component of NetX, 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 automatically determines the source address to use for the connection, and the next hop address based on the destination IP address of the connection.
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. After the other side disconnects, eventually the TCP socket returns to the CLOSED state. This process repeats each time a TCP client and server establish and close a connection. Figure 11 on page 97 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 the `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 `nx_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 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 responds with an ACK packet and promotes the client socket to the ESTABLISHED state.

TCP Client Disconnection

Closing the connection is accomplished by calling `nx_tcp_socket_disconnect`. If no suspension is
FIGURE 11. States of the TCP State Machine
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 promotes the client 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 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 suspends for the specified timeout to allow the packets to be acknowledged. If the timeout expires, NetX 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., TIMED 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 to initiate connection request. To accept a client connection, TCP must first be enabled on the IP instance by calling the service `nx_tcp_enable`. 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. This is achieved by 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 socket.

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 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 is in the LISTEN state and 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 set up 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 `listen_queue_size`, passed into 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 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 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 suspends for the specified timeout to allow those packets to be acknowledged. If the timeout expires, NetX 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 \texttt{nx\_tcp\_socket\_disconnect} or \texttt{nx\_tcp\_server\_socket\_accept} return an error status. After the \texttt{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 \texttt{nx\_tcp\_server\_socket\_relisten} service should be called on this socket.

The following code segment illustrates the sequence of calls a typical TCP server uses:

```c
/* Set up a previously created TCP socket to listen on port 12 */
xq_tcp_server_socket_liston();

/* Loop to make a (another) connection. */
while(1)
{
    /* Wait for a client socket connection request for 100 ticks. */
    nxq_tcp_server_socket_accept();

    /* (Send and receive TCP messages with the TCP client) */

    /* Disconnect the server socket. */
    nxq_tcp_socket_disconnect();

    /* Remove this server socket from listening on the port. */
    nxq_tcp_server_socket_unaccept(&server_socket);

    /* Set up server socket to relisten on the same port for the next client. */
    nxq_tcp_server_socket_relisten();
}
```

### MSS Validation

The Maximum Segment Size (MSS) is the maximum amount of bytes a TCP host can receive without being fragmented by the underlying IP layer. During TCP connection establishment phase, both ends exchanges its own TCP MSS value, so that the sender does not send a TCP data segment that is larger than the receiver’s MSS. NetX TCP
module will optionally validate its peer’s advertised MSS value before establishing a connection. By default NetX does not enable such a check. Applications wishing to perform MSS validation shall define `NX_ENABLE_TCP_MSS_CHECKING` when building the NetX library, and the minimum value shall be defined in `NX_TCP_MSS_MINIMUM`. Incoming TCP connections with MSS values below `NX_TCP_MSS_MINIMUM` are dropped.

**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 fits into the receiver’s window. 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. If the size of the data being transmitted is larger than the MSS value of the socket or the current peer receive window size, whichever is smaller, TCP internal logic carves off the data that fits into min (MSS, peer receive Window) for transmission. This service then builds a TCP header in front of the packet (including the
checksum calculation). If the receiver’s window size is not zero, the caller will send as much data as it can to fill up the receiver window size. If the receive window becomes zero, 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, multiple threads may suspend while trying to send data through the same socket.

*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**

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

*Application shall not reuse the packet or alter the contents of the packet after nx_tcp_socket_send() returns with NX_SUCCESS. The transmitted packet is eventually released by NetX internal processing after the data is acknowledged by the other end.*

**TCP Keepalive**

TCP Keepalive feature allows a socket to detect whether or not its peer disconnects without proper termination (for example, the peer crashed), or to prevent certain network monitoring facilities to terminate a connection for long periods of idle. TCP Keepalive works by periodically sending a TCP frame with no data, and the sequence number set to one less than the current sequence number. On receiving
such TCP Keepalive frame, the recipient, if still alive, responses with an ACK for its current sequence number. This completes the keepalive transaction.

By default the keepalive feature is not enabled. To use this feature, NetX library must be built with `NX_ENABLE_TCP_KEEPALIVE` defined. The symbol `NX_TCP_KEEPALIVE_INITIAL` specifies the number of seconds of inactivity before the keepalive frame is initiated.

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.

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 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 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.
Description of NetX Services

This chapter contains a description of all NetX 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.

Note that a BSD-Compatible Socket API is available for legacy application code that cannot take full advantage of the high-performance NetX 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 service can be called.

```
nx_arp_dynamic_entries_invalidate 114
   Invalidate all dynamic entries in the ARP cache
nx_arp_dynamic_entry_set 116
   Set dynamic ARP entry
nx_arp_enable 118
   Enable Address Resolution Protocol (ARP)
nx_arp_gratuitous_send 120
   Send gratuitous ARP request
nx_arp_hardware_address_find 122
   Locate physical hardware address given an IP address
nx_arp_info_get 124
   Retrieve information about ARP activities
nx_arp_ip_address_find 126
   Locate IP address given a physical address
nx_arp_static_entries_delete 128
   Delete all static ARP entries
nx_arp_static_entry_create 130
   Create static IP to hardware mapping in ARP cache
```
nx_arp_static_entry_delete 132
   Delete static IP to hardware mapping in ARP cache

nx_icmp_enable 134
   Enable Internet Control Message Protocol (ICMP)

nx_icmp_info_get 136
   Retrieve information about ICMP activities

nx_icmp_ping 138
   Send ping request to specified IP address

nx_igmp_enable 140
   Enable Internet Group Management Protocol (IGMP)

nx_igmp_info_get 142
   Retrieve information about IGMP activities

nx_igmp_loopback_disable 144
   Disable IGMP loopback

nx_igmp_loopback_enable 146
   Enable IGMP loopback

nx_igmp_multicast_interface_join 148
   Join IP instance to specified multicast group via an interface

nx_igmp_multicast_join 150
   Join IP instance to specified multicast group

nx_igmp_multicast_leave 152
   Cause IP instance to leave specified multicast group

nx_ip_address_change_notify 154
   Notify application if IP address changes

nx_ip_address_get 156
   Retrieve IP address and network mask

nx_ip_address_set 158
   Set IP address and network mask

nx_ip_create 160
   Create an IP instance

nx_ip_delete 162
   Delete previously created IP instance

nx_ip_driver_direct_command 164
   Issue command to network driver

nx_ip_driver_interface_direct_command 166
   Issue command to network driver

nx_ip_forwarding_disable 168
   Disable IP packet forwarding
nx_ip_forwarding_enable 170
   Enable IP packet forwarding
nx_ip_fragment_disable 172
   Disable IP packet fragmenting
nx_ip_fragment_enable 174
   Enable IP packet fragmenting
nx_ip_gateway_address_set 176
   Set Gateway IP address
nx_ip_info_get 178
   Retrieve information about IP activities
nx_ip_interface_address_get 180
   Retrieve interface IP address
nx_ip_interface_address_set 182
   Set interface IP address and network mask
nx_ip_interface_attach 184
   Attach network interface to IP instance
nx_ip_interface_info_get 186
   Retrieve network interface parameters
nx_ip_interface_status_check 188
   Check status of an IP instance
nx_ip_link_status_change_notify_set 190
   Set the link status change notify callback function
nx_ip_raw_packet_disable 192
   Disable raw packet sending/receiving
nx_ip_raw_packet_enable 194
   Enable raw packet processing
nx_ip_raw_packet_interface_send 196
   Send raw IP packet through specified network interface
nx_ip_raw_packet_receive 198
   Receive raw IP packet
nx_ip_raw_packet_send 200
   Send raw IP packet
nx_ip_static_route_add 202
   Add static route to the routing table
nx_ip_static_route_delete 204
   Delete static route from routing table
nx_ip_status_check 206
   Check status of an IP instance
nx_packet_allocate 208
   Allocate packet from specified pool
nx_packet_copy 210
  Copy packet

nx_packet_data_append 212
  Append data to end of packet

nx_packet_data_extract_offset 214
  Extract data from packet via an offset

nx_packet_data_retrieve 216
  Retrieve data from packet

nx_packet_length_get 218
  Get length of packet data

nx_packet_pool_create 220
  Create packet pool in specified memory area

nx_packet_pool_delete 222
  Delete previously created packet pool

nx_packet_pool_info_get 224
  Retrieve information about a packet pool

nx_packet_release 226
  Release previously allocated packet

nx_packet_transmit_release 228
  Release a transmitted packet

nx_rarp_disable 230
  Disable Reverse Address Resolution Protocol (RARP)

nx_rarp_enable 232
  Enable Reverse Address Resolution Protocol (RARP)

nx_rarp_info_get 234
  Retrieve information about RARP activities

nx_system_initialize 236
  Initialize NetX System

nx_tcp_client_socket_bind 238
  Bind client TCP socket to TCP port

nx_tcp_client_socket_connect 240
  Connect client TCP socket

nx_tcp_client_socket_port_get 242
  Get port number bound to client TCP socket

nx_tcp_client_socket_unbind 244
  Unbind TCP client socket from TCP port

nx_tcp_enable 246
  Enable TCP component of NetX

nx_tcp_free_port_find 248
  Find next available TCP port
nx_tcp_info_get 250
  Retrieve information about TCP activities
nx_tcp_server_socket_accept 254
  Accept TCP connection
nx_tcp_server_socket_listen 258
  Enable listening for client connection on TCP port
nx_tcp_server_socket_relisten 262
  Re-listen for client connection on TCP port
nx_tcp_server_socket_unaccept 266
  Remove socket association with listening port
nx_tcp_server_socket_unlisten 270
  Disable listening for client connection on TCP port
nx_socket_bytes_available 274
  Retrieves number of bytes available for retrieval
nx_socket_create 276
  Create TCP client or server socket
nx_socket_delete 280
  Delete TCP socket
nx_socket_disconnect 282
  Disconnect client and server socket connections
nx_socket_disconnect_complete_notify 284
  Install TCP disconnect complete notify callback function
nx_socket_establish_notify 286
  Set TCP establish notify callback function
nx_socket_info_get 288
  Retrieve information about TCP socket activities
nx_socket_mss_get 292
  Get MSS of socket
nx_socket_mss_peer_get 294
  Get MSS of the peer TCP socket
nx_socket_mss_set 296
  Set MSS of socket
nx_socket_peer_info_get 298
  Retrieve information about peer TCP socket
nx_socket_receive 300
  Receive data from TCP socket
nx_socket_receive_notify 302
  Notify application of received packets
nx_socket_send 304
  Send data through a TCP socket
nx_tcp_socket_state_wait 308
   Wait for TCP socket to enter specific state
nx_tcp_socket_timed_wait_callback 310
   Install callback for timed wait state
nx_tcp_socket_transmit_configure 312
   Configure socket’s transmit parameters
nx_tcp_socket_window_update_notify_set 314
   Notify application of window size updates
nx_udp_enable 316
   Enable UDP component of NetX
nx_udp_free_port_find 318
   Find next available UDP port
nx_udp_info_get 320
   Retrieve information about UDP activities
nx_udp_packet_info_extract 322
   Extract network parameters from UDP packet
nx_udp_socket_bind 324
   Bind UDP socket to UDP port
nx_udp_socket_bytes_available 326
   Retrieves number of bytes available for retrieval
nx_udp_socket_checksum_disable 328
   Disable checksum for UDP socket
nx_udp_socket_checksum_enable 330
   Enable checksum for UDP socket
nx_udp_socket_create 332
   Create UDP socket
nx_udp_socket_delete 334
   Delete UDP socket
nx_udp_socket_info_get 336
   Retrieve information about UDP socket activities
nx_udp_socket_port_get 338
   Pick up port number bound to UDP socket
nx_udp_socket_receive 340
   Receive datagram from UDP socket
nx_udp_socket_receive_notify 342
   Notify application of each received packet
nx_udp_socket_send 344
   Send a UDP Datagram
nx_udp_socket_interface_send 346
   Send datagram through UDP socket
nx_udp_socket_unbind 348
   Unbind UDP socket from UDP port
nx_udp_source_extract 350
   Extract IP and sending port from UDP datagram
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**: Top 16 bits (47-32) of the physical address.
- **physical_lsw**: Lower 32 bits (31-0) 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_IP_ADDRESS_ERROR** (0x21): Invalid IP address.
- **NX_PTR_ERROR** (0x07): Invalid IP instance pointer.
- **NX_NOT_ENABLED** (0x14): This component has not been enabled.
- **NX_CALLER_ERROR** (0x11): Invalid caller of this service.
Allowed From
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),
0x1022, 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
10:22:00:00:12:34. */

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

```c
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 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 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): User supplied ARP cache memory is too small.
- **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 ip_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 goes through all the physical interfaces to transmit gratuitous ARP requests as long as the interface IP address is valid. If an ARP response is subsequently received, the supplied response handler is called to process the response to the gratuitous ARP.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip_ptr</td>
<td>Pointer to previously created IP instance.</td>
</tr>
<tr>
<td>response_handler</td>
<td>Pointer to response handling function. If NX_NULL is supplied, responses are ignored.</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>(0x00)</td>
<td>Successful gratuitous ARP send.</td>
</tr>
<tr>
<td>NX_NO_PACKET</td>
<td>(0x01)</td>
<td>No packet available.</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>(0x14)</td>
<td>ARP is not enabled.</td>
</tr>
<tr>
<td>NX_IP_ADDRESS_ERROR</td>
<td>(0x21)</td>
<td>Current IP address is invalid.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>(0x07)</td>
<td>Invalid IP pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>(0x11)</td>
<td>Caller is not a thread.</td>
</tr>
</tbody>
</table>
Allowed From  
Threads  

Preemption Possible  
No  

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,
ULONGLONG ip_address,
ULONGLONG *physical_msw,
ULONGLONG *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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip_ptr</td>
<td>Pointer to previously created IP instance.</td>
</tr>
<tr>
<td>ip_address</td>
<td>IP address to search for.</td>
</tr>
<tr>
<td>physical_msw</td>
<td>Pointer to the variable for returning the top 16 bits (47-32) of the physical address.</td>
</tr>
<tr>
<td>physical_lsw</td>
<td>Pointer to the variable for returning the lower 32 bits (31-0) of the physical address.</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>(0x00) Successful ARP hardware address find.</td>
</tr>
<tr>
<td>NX_ENTRY_NOT_FOUND</td>
<td>(0x16) Mapping was not found in the ARP cache.</td>
</tr>
<tr>
<td>NX_IP_ADDRESS_ERROR</td>
<td>(0x21) Invalid IP address.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>(0x07) Invalid IP or memory pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>(0x11) Invalid caller of this service.</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>(0x14) This component has not been enabled.</td>
</tr>
</tbody>
</table>
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.

```
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 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**

Threads

**Preemption Possible**

No

**Example**

```c
/* Pickup ARP information for ip_0. */
status = nx_arp_info_get(&ip_0, &arp_requests_sent,
                         &arp_requests_received,
                         &arpResponses_sent,
                         &arpResponses_received,
                         &arpDynamic_entries,
                         &arpStatic_entries,
                         &arp_aged_entries,
                         &arpInvalid_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`: Top 16 bits (47-32) of the physical address to search for.
- `physical_lsw`: Lower 32 bits (31-0) 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.
- **NX_INVALID_PARAMETERS** (0x4D): Physical_msw and physical_lsw are both 0.
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 ip_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 service 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_hardwire_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: Top 16 bits (47-32) of the physical address to map.
- physical_lsw: Lower 32 bits (31-0) 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.
- NX_INVALID_PARAMETERS (0x4D): Physical_msw and physical_lsw are both 0.
Allowed From
Initialization, threads

Preemption Possible
No

Example

```c
/* 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 00:00:00:00:12:34. */
```

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

```c
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**: Top 16 bits (47 - 32) of the physical address that was mapped statically.
- **physical_lsw**: Lower 32 bits (31 - 0) 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.
- **NX_INVALID_PARAMETERS** (0x4D): Physical_msw and physical_lsw are both 0.
Allowed From

Threads

Preemption Possible

No

Example

```c
/* Delete a static ARP entry on the previously created IP instance ip_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,
px_arp_enable, nx_arp_gratuitous_send,
nx_arp_hardwared_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.

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 ip_0. */
status = nx_icmp_enable(&ip_0);

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

See Also

nx_icmp_info_get, nx_icmp_ping
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

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>0x00</td>
<td>Successful ICMP information retrieval.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>0x11</td>
<td>Invalid caller of this service.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>0x07</td>
<td>Invalid IP pointer.</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>0x14</td>
<td>This component has not been enabled.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads

Preemption Possible

No

Example

/* Retrieve ICMP information from previously created IP instance ip_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_icmp_ping
nx_icmp_ping

Send ping request to specified IP address

Prototype

```c
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 is returned in the variable pointed to by `response_ptr`.

*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, in host byte order, 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. Wait options are defined as follows:
  - `NX_NO_WAIT` (0x00000000)
  - `NX_WAIT_FOREVER` (0xFFFFFFFF)
  - timeout value (0x00000001 through 0xFFFFFFFF)

Return Values

- **NX_SUCCESS** (0x00) Successful ping. Response message pointer was placed in
Internet Control Message Protocol (ICMP)

The variable pointed to by response_ptr.

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

Allowed From

Threads

Preemption Possible

No

Example

/* Issue a ping to IP address 1.2.3.5 from the previously created IP Instance ip_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
nx_igmp_enable

Enable Internet Group Management Protocol (IGMP)

Prototype

```c
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 ip_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

```c
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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>0x00</td>
<td>Successful IGMP information retrieval.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>0x07</td>
<td>Invalid IP pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>0x11</td>
<td>Invalid caller of this service.</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>0x14</td>
<td>This component has not been enabled.</td>
</tr>
</tbody>
</table>
Allowed From
Initialization, threads

Preemption Possible
No

Example

```c
/* Retrieve IGMP information from previously created IP Instance ip_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

Prototype

```c
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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>0x00</td>
<td>Successful IGMP loopback disable.</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>0x14</td>
<td>IGMP is not enabled.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>0x07</td>
<td>Invalid IP pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>0x11</td>
<td>Caller is not a thread or initialization.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads

Preemption Possible

No
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

Preemption Possible

No
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 instance to specified multicast group via an interface

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 joining the multicast group, 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 Index of the Interface attached to the NetX 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_PTR_ERROR (0x07) Invalid IP pointer.
- NX_INVALID_INTERFACE (0x4C) Device 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.
Internet Group Management Protocol (IGMP)

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 interface list. */
#define INTERFACE_INDEX 1
status = nx_igmp_multicast_interface_join(&ip, IP_ADDRESS(244,0,0,200), INTERFACE_INDEX);

/* 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

Express Logic
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.

To join a multicast group on a non-primary device, use the service nx_igmp_multicast_interface_join.

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_INVALID_INTERFACE (0x4C) Device index points to an invalid network interface.
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

/* Previously created IP Instance ip_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_INVALID_INTERFACE** (0x4C) Device index points to an invalid network interface.
- **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_notify**  
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.

- **NX_CALLER_ERROR**  
  (0x11) Invalid caller of this service.

**Allowed From**

Initialization, threads

**Preemption Possible**

No
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_driver_interface_direct_command,
nx_ip_forwarding_disable, nx_ip_forwarding_enable,
nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_info_get,
nx_ip_status_check, nx_system_initialize
nx_ip_address_get

Retrieve IP address and network mask

Prototype

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

Description

This service retrieves IP address and its subnet mask of the primary network interface.

To obtain information of the secondary device, 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

Preemption Possible

- No
Example

/* Get the IP address and network mask from the previously created IP Instance ip_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_driver_interface_direct_command, nx_ip_forwarding_disable,
nx_ip_forwarding_enable, nx_ip_fragment_disable,
nx_ip_fragment_enable, nx_ip_info_get, nx_ip_status_check,
nx_system_initialize
nx_ip_address_set

Set IP address and network mask

Prototype

```c
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 device, 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 0xFFFFFFFF00 for the previously created IP Instance ip_0. */
status = nx_ip_address_set(&ip_0, IP_ADDRESS(1,2,3,4), 0xFFFFFFFF00UL);

/* If status is NX_SUCCESS, the IP instance now has an IP address of 1.2.3.4 and a network mask of 0xFFFFFFFF00. */

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command,
nx_ip_driver_interface_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable,
nx_ip_fragment_enable, nx_ip_info_get, nx_ip_status_check, nx_system_initialize
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 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.
**Internet Protocol (IP)**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_NOT_IMPLEMENTED (0x4A)</td>
<td>NetX library is configured incorrectly.</td>
</tr>
<tr>
<td>NX_PTR_ERROR (0x07)</td>
<td>Invalid IP, network driver function pointer, packet pool, or memory pointer.</td>
</tr>
<tr>
<td>NX_SIZE_ERROR (0x09)</td>
<td>The supplied stack size is too small.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR (0x11)</td>
<td>Invalid caller of this service.</td>
</tr>
<tr>
<td>NX_IP_ADDRESS_ERROR (0x21)</td>
<td>The supplied IP address is invalid.</td>
</tr>
<tr>
<td>NX_OPTION_ERROR (0x21)</td>
<td>The supplied IP thread priority is invalid.</td>
</tr>
</tbody>
</table>

**Allowed From**

Initialization, threads

**Preemption Possible**

No

**Example**

```c
/* Create an IP instance with an IP address of 1.2.3.4 and a network mask of 0xFFFFFFFF00UL. 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 ip_0", IP_ADDRESS(1, 2, 3, 4), 0xFFFFFFFF00UL, &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_driver_interface_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_info_get, nx_ip_status_check, nx_system_initialize
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_driver_interface_direct_command, nx_ip_forwarding_disable,
    nx_ip_forwarding_enable, nx_ip_fragment_disable,
    nx_ip_fragment_enable, nx_ip_info_get, nx_ip_status_check,
    nx_system_initialize
nx_ip_driver_direct_command

Issue command to network driver

Prototype

```c
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 command for the secondary device, 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.

NX_INVALID_INTERFACE (0x4C) Invalid interface index.

Allowed From

Threads

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_driver_interface_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_info_get, nx_ip_status_check, nx_system_initialize
nx_ip_driver_interface_direct_command

Issue command to network driver

Prototype

```c
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 command to the application’s network device driver in the IP instance. Application-specific commands can be used providing their numeric value is greater than or equal to

`NX_LINK_USER_COMMAND`.

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` Index of the network interface the command should be sent to.
- `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

Threads

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 device. */
UINT interface_index = 0;

status = nx_ip_driver_interface_direct_command(&ip_0,
    NX_LINK_GET_STATUS,
    interface_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_driver_direct_command,
nx_ip_forwarding_disable, nx_ip_forwarding_enable,
nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_info_get,
nx_ip_status_check, nx_system_initialize
nx_ip_forwarding_disable

Description
This service disables forwarding IP packets inside the NetX 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_driver_interface_direct_command, nx_ip_forwarding_enable,
    nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_info_get,
    nx_ip_status_check, nx_system_initialize
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 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_driver_interface_direct_command, nx_ip_forwarding_disable,
nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_info_get, nx_ip_status_check, nx_system_initialize
nx_ip_fragment_disable

Disable IP packet fragmenting

Prototype

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

Description

This service disables IP packet fragmenting and reassembling functionality. For packets waiting to be reassembled, this service releases these packets. On creation of the IP task, this service is automatically disabled.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip_ptr</td>
<td>Pointer to previously created IP instance.</td>
</tr>
</tbody>
</table>

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.
- **NX_NOT_ENABLED** (0x14) IP Fragmentation is not enabled on the IP instance.

Allowed From

- Initialization, threads

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_address_change_notify, nx_ip_address_get, nx_ip_address_set,
nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command,
nx_ip_driver_interface_direct_command, nx_ip_forwarding_disable,
nx_ip_forwarding_enable, nx_ip_fragment_enable, nx_ip_info_get,
nx_ip_status_check, nx_system_initialize
nx_ip_fragment_enable

Enable IP packet fragmenting

Prototype

```c
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.
- `NX_NOT_ENABLED` (0x14) IP Fragmentation features is not compiled into NetX.

Allowed From

- Initialization, threads

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_driver_interface_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_info_get,
nx_ip_status_check, nx_system_initialize
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 IP gateway IP address. All out-of-network traffic are routed to this gateway 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_IP_ADDRESS_ERROR** (0x21): Invalid IP address.
- **NX_CALLER_ERROR** (0x11): Invalid caller of this service.

Allowed From

- Initialization, thread

Preemption Possible

- No
Example

/* Setup the Gateway address for previously created IP Instance ip_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_info_get, 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

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_driver_interface_direct_command, nx_ip_forwarding_disable,
nx_ip_forwarding_enable, nx_ip_fragment_disable,
nx_ip_fragment_enable, nx_ip_status_check, nx_system_initialize
nx_ip_interface_address_get

Retrieve interface IP address

Prototype

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

Description

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

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

Parameters

- **ip_ptr**: Pointer to previously created IP instance.
- **interface_index**: Interface index, the same value as the index to the network interface attached to the IP 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

Preemption Possible

No
Example

```c
#define INTERFACE_INDEX 1
/* 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, INTERFACE_INDEX,
 &ip_address,
 &network_mask);

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

See Also

nx_ip_interface_address_set, nx_ip_interface_attach,
nx_ip_interface_info_get, nx_ip_interface_status_check,
nx_ip_link_status_change_notify_set
nx_ip_interface_address_set

Set interface IP address and network mask

Prototype

UINT nx_ip_interface_address_set(NX_IP *ip_ptr,
                                   UINT interface_index,
                                   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_index Index of the interface attached to the NetX 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.
- NX_IP_ADDRESS_ERROR (0x21) Invalid IP address

Allowed From

Initialization, threads

Preemption Possible

No
Example

```c
#define INTERFACE_INDEX 1
/* 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, INTERFACE_INDEX,
    ip_address,
    network_mask);

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

See Also

nx_ip_interface_address_get, nx_ip_interface_attach,
nx_ip_interface_info_get, nx_ip_interface_status_check,
nx_ip_link_status_change_notify_set
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 service adds a physical network interface to the IP interface. Note the
IP instance 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.

If the IP thread has not been running yet, the secondary interfaces will be
initialized as part of the IP thread startup process that initializes all
physical interfaces.

If the IP thread is not running yet, the secondary interface is initialized as
part of the nx_ip_interface_attach service.

\[ ip_ptr \] must point to a valid NetX 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip_ptr</td>
<td>Pointer to previously created IP instance.</td>
</tr>
<tr>
<td>interface_name</td>
<td>Pointer to interface name string.</td>
</tr>
<tr>
<td>ip_address</td>
<td>Device IP address in host byte order.</td>
</tr>
<tr>
<td>network_mask</td>
<td>Device network mask in host byte order.</td>
</tr>
<tr>
<td>ip_link_driver</td>
<td>Ethernet driver for the interface.</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>(0x00) Entry is added to static routing table.</td>
</tr>
</tbody>
</table>
NX_NO_MORE_ENTRIES (0x17)  Max number of interfaces. NX_MAX_PHYSICAL_INTERFACES is exceeded.

NX_DUPLICATED_ENTRY (0x52)  The supplied IP address is already used on this IP instance.

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 device 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),
        0xFFFFFFFFUL,
        nx_etherDriver);

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

See Also
nx_ip_interface_address_get, nx_ip_interface_address_set,
nx_ip_interface_info_get, nx_ip_interface_status_check,
nx_ip_link_status_change_notify_set
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 service retrieves information on network parameters for the specified network interface. All data are retrieved in host byte order.

`ip_ptr` must point to a valid NetX 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 the buffer that holds the name of the network interface.
- **ip_address**: Pointer to the destination for the IP address of the interface.
- **network_mask**: Pointer to destination for network mask.
- **mtu_size**: Pointer to destination for maximum transfer unit for this interface.
- **physical_address_msw**: Pointer to destination for top 16 bits of the device MAC address.
- **physical_address_lsw**: Pointer to destination for lower 32 bits of the device MAC address.

Return Values

- **NX_SUCCESS** (0x00): Interface information has been obtained.
- **NX_PTR_ERROR** (0x07): Invalid pointer input.
NX_INVALID_INTERFACE (0x4C)  Invalid IP pointer.
NX_CALLER_ERROR          (0x11)          Service is not called from system initialization or thread context.

Allowed From
Initialization, threads

Preemption Possible
No

Example

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

/*  If status is NX_SUCCESS the interface information is successfully retrieved.  */

See Also

nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_status_check, nx_ip_link_status_change_notify_set
nx_ip_interface_status_check

Check status of an IP instance

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 network interface of a previously created IP instance.

Parameters

- **ip_ptr**: Pointer to previously created IP instance.
- **interface_index**: Interface index number
- **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)
  - NX_WAIT_FOREVER (0xFFFFFFFF)
  - timeout value (0x00000001 through 0xFFFFFFFF)
Return Values

- **NX_SUCCESS** (0x00): Successful IP status check.
- **NX_NOT_SUCCESSFUL** (0x43): Status request was not satisfied within the timeout specified.
- **NX_PTR_ERROR** (0x07): IP pointer is or has become invalid, or actual status pointer is invalid.
- **NX_OPTION_ERROR** (0x0a): Invalid needed status option.
- **NX_CALLER_ERROR** (0x11): Invalid caller of this service.
- **NX_INVALID_INTERFACE** (0x4C): Interface_index is out of range or the interface is not valid.

Allowed From

Threads

Preemption Possible

No

Example

```c
/* Wait 10 ticks for the link up status on the previously created IP instance. */
status = nx_ip_interface_status_check(&ip_0, 1, NX_IP_LINK_ENABLED, &actual_status, 10);

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

See Also

- `nx_ip_interface_address_get`, `nx_ip_interface_address_set`,
- `nx_ip_interface_attach`, `nx_ip_interface_info_get`,
- `nx_ip_link_status_change_notify_set`
nx_ip_link_status_change_notify_set

Set the link status change notify callback function

Prototype

UINT nx_ip_link_status_change_notify_set(NX_IP *ip_ptr,
VOID(*link_status_change_notify(NX_IP *ip_ptr,
UINT interface_index, UINT link_up))

Description

This service configures the link status change notify callback function. The user-supplied *link_status_change_notify* routine is invoked when either the primary or secondary interface status is changed (such as IP address is changed.) If *link_status_change_notify* is NULL, the link status change notify callback feature is disabled.

Parameters

- **ip_ptr**: IP control block pointer
- **link_status_change_notify**: User-supplied callback function to be called upon a change to the physical interface.

Return Values

- **NX_SUCCESS** (0x00) Successful set
- **NX_PTR_ERROR** (0x07) Invalid IP control block pointer or new physical address pointer
- **NX_CALLER_ERROR** (0x11) Service is not called from system initialization or thread context.

Allowed From

Initialization, threads

Preemption Possible

No
Example

/* Configure a callback function to be used when the physical
 interface status is changed. */
status = nx_ip_link_status_change_notify_set(&ip_0,
                my_change_cb);

/* If status == NX_SUCCESS, the link status change notify function
is set. */

See Also

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

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. If the raw packet service was previously enabled, and there are raw packets in the receive queue, this service will release any received raw packets.

Parameters

ip_ptr Pointer to previously created IP instance.

Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>(0x00) Successful IP raw packet disable.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>(0x07) Invalid IP pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>(0x11) Invalid caller of this service.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads

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_raw_packet_enable, nx_ip_raw_packet_receive,
nx_ip_raw_packet_send, nx_ip_raw_packet_interface_send
nx_ip_raw_packet_enable

Enable raw packet processing

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. Incoming TCP, UDP, ICMP, and IGMP packets are still processed by NetX. Packets with unknown upper layer protocol types are processed by raw packet reception routine.

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

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_raw_packet_disable, nx_ip_raw_packet_receive,
nx_ip_raw_packet_send, nx_ip_raw_packet_interface_send
nx_ip_raw_packet_interface_send

Send raw IP packet through specified network interface

Prototype

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

Description

This service sends a raw IP packet to the destination IP address using the specified local IP address as the source address, and through the associated 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.

\begin{itemize}
  \item \textit{Note that raw IP processing must be enabled.}
\end{itemize}

\begin{itemize}
  \item This service is similar to \textit{nx_ip_raw_packet_send}, except that this service allows an application to send raw IP packet from a specified physical interfaces.
\end{itemize}

Parameters

\begin{itemize}
  \item ip_ptr Pointer to previously created IP task.
  \item packet_ptr Pointer to packet to transmit.
  \item destination_ip IP address to send packet.
  \item address_index Index of the address of the interface to send packet out on.
  \item type_of_service Type of service for packet.
\end{itemize}
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

- No

Example

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

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

See Also

- `nx_ip_raw_packet_disable`, `nx_ip_raw_packet_enable`,
- `nx_ip_raw_packet_receive`, `nx_ip_raw_packet_send`
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 through 0xFFFFFFFF)

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
No

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_raw_packet_disable, nx_ip_raw_packet_enable,
nx_ip_raw_packet_send, nx_ip_raw_packet_interface_send
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 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.

For a multihome system, NetX uses the destination IP address to find an
appropriate network interface and uses the IP address of the interface as
the source address. If the destination IP address is broadcast or multicast,
the first valid interface is used. 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip_ptr</td>
<td>Pointer to previously created IP instance.</td>
</tr>
<tr>
<td>packet_ptr</td>
<td>Pointer to the raw IP packet to send.</td>
</tr>
<tr>
<td>destination_ip</td>
<td>Destination IP address, which can be a specific host IP address, a network broadcast, an internal loop-back, or a multicast address.</td>
</tr>
<tr>
<td>type_of_service</td>
<td>Defines the type of service for the transmission, legal values are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>NX_IP_NORMAL</td>
<td>(0x00000000)</td>
</tr>
<tr>
<td>NX_IP_MIN_DELAY</td>
<td>(0x00100000)</td>
</tr>
<tr>
<td>NX_IP_MAX_DATA</td>
<td>(0x00080000)</td>
</tr>
<tr>
<td>NX_IP_MAX_RELIABLE</td>
<td>(0x00040000)</td>
</tr>
<tr>
<td>NX_IP_MIN_COST</td>
<td>(0x00020000)</td>
</tr>
</tbody>
</table>
Return Values

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

Allowed From

Threads

Preemption Possible

No

Example

```c
/* 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_raw_packet_disable, nx_ip_raw_packet_enable,
nx_ip_raw_packet_receive, nx_ip_raw_packet_send,
nx_ip_raw_packet_interface_send
nx_ip_static_route_add

Add static route to the routing table

Prototype

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

Description

This service adds an entry to the static routing table. Note that the `next_hop` address must be directly accessible from one of the local network devices.

Note that `ip_ptr` must point to a valid NetX IP structure and the NetX library must be built with `NX_ENABLE_IP_STATIC_ROUTING` defined to use this service. By default NetX is built without `NX_ENABLE_IP_STATIC_ROUTING` defined.

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 the 192.168.10.0 through the gateway 192.168.1.1. */
status = nx_ip_static_route_add(ip_ptr, IP_ADDRESS(192,168,10,0),
                               0xFFFFFFFFUL,
                               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_gateway_address_set, nx_ip_info_get, nx_ip_static_route_delete`
nx_ip_static_route_delete

Delete static route from routing table

Prototype

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

Description

This service deletes an entry from the static routing table.

Note that ip_ptr must point to a valid NetX IP structure and the NetX library must be built with \textit{NX\_ENABLE\_IP\_STATIC\_ROUTING} defined to use this service. By default NetX is built without \textit{NX\_ENABLE\_IP\_STATIC\_ROUTING} defined.

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

```c
/* Remove the static route for 192.168.10.0 from the routing table. */
status = nx_ip_static_route_delete(ip_ptr,
    IP_ADDRESS(192,168,10,0),
    0xFFFFFFFFUL,);

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

See Also

nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_static_route_add
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. To obtain status on secondary interfaces, applications shall use the service nx_ip_interface_status_check.

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)
  - `NX_WAIT_FOREVER` (0xFFFFFFFF)
  - timeout value (0x00000001 through 0xFFFFFFFF)
Return Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>(0x00)</td>
<td>Successful IP status check.</td>
</tr>
<tr>
<td>NX_NOT_SUCCESSFUL</td>
<td>(0x43)</td>
<td>Status request was not satisfied within the timeout specified.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>(0x07)</td>
<td>IP pointer is or has become invalid, or actual status pointer is invalid.</td>
</tr>
<tr>
<td>NX_OPTION_ERROR</td>
<td>(0x0a)</td>
<td>Invalid needed status option.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>(0x11)</td>
<td>Invalid caller of this service.</td>
</tr>
</tbody>
</table>

Allowed From
Threads

Preemption Possible
No

Example

```c
/* 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_driver_interface_direct_command, nx_ip_forwarding_disable,
nx_ip_forwarding_enable, nx_ip_fragment_disable,
nx_ip_fragment_enable, nx_ip_info_get, nx_system_initialize
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 Pools” on page 49 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 0xFFFFFFFF)
Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>0x00</td>
<td>Successful packet allocate.</td>
</tr>
<tr>
<td>NX_NO_PACKET</td>
<td>0x01</td>
<td>No packet available.</td>
</tr>
<tr>
<td>NX_WAIT_ABORTED</td>
<td>0x1A</td>
<td>Requested suspension was aborted by a call to <code>tx_thread_wait_abort</code>.</td>
</tr>
<tr>
<td>NX_INVALID_PARAMETERS</td>
<td>0x4D</td>
<td>Packet size cannot support protocol.</td>
</tr>
<tr>
<td>NX_OPTION_ERROR</td>
<td>0x0A</td>
<td>Invalid packet type.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>0x07</td>
<td>Invalid pool or packet return pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>0x11</td>
<td>Invalid wait option from non-thread.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads, timers, and ISRs (application network drivers). Wait option must be `NX_NO_WAIT` when used in ISR or in timer context.

Preemption Possible

No

Example

```c
/* 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
0xFFFFFFFFE)

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.
### Packet Management

**NX_INVALID_PARAMETERS** (0x4D)  
Packet size cannot support protocol.

**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
No

#### 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, and the chained packet feature is enabled, one or more packets will be allocated to satisfy the request. If the chained packet feature is not enabled, NX_SIZE_ERROR is returned.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>packet_ptr</td>
<td>Packet pointer.</td>
</tr>
<tr>
<td>data_start</td>
<td>Pointer to the start of the user’s data area to append to the packet.</td>
</tr>
<tr>
<td>data_size</td>
<td>Size of user’s data area.</td>
</tr>
<tr>
<td>pool_ptr</td>
<td>Pointer to packet pool from which to allocate another packet if there is not enough room in the current packet.</td>
</tr>
<tr>
<td>wait_option</td>
<td>Defines how the service behaves if there are no packets available. The wait options are defined as follows:</td>
</tr>
<tr>
<td></td>
<td>NX_NO_WAIT     (0x00000000)</td>
</tr>
<tr>
<td></td>
<td>NX_WAIT_FOREVER (0xFFFFFFF)</td>
</tr>
<tr>
<td></td>
<td>timeout value  (0x00000001 through 0xFFFFFFF)</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>(0x00)</td>
<td>Successful packet append.</td>
</tr>
<tr>
<td>NX_NO_PACKET</td>
<td>(0x01)</td>
<td>No packet available.</td>
</tr>
<tr>
<td>NX_WAIT_ABORTED</td>
<td>(0x1A)</td>
<td>Requested suspension was aborted by a call to tx_thread_wait_abort.</td>
</tr>
</tbody>
</table>
**NX_INVALID_PARAMETERS**  
(0x4D) Packet size cannot support protocol.

**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**
No

**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

```c
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 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 or other internal state information.

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_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

```c
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`.

Note that this service does not change internal state of the packet. The data being retrieved is still available in the packet.

```
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

```c
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

```c
/* 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 created packet pool. NetX 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

Yes
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

```c
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

```c
/* 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 chained
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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Successful packet release.</td>
</tr>
<tr>
<td>0x07</td>
<td>Invalid packet pointer.</td>
</tr>
<tr>
<td>0x02</td>
<td>Prepend pointer is less than payload start.</td>
</tr>
<tr>
<td>0x03</td>
<td>Append pointer is greater than payload end.</td>
</tr>
</tbody>
</table>

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

```c
UINT nx_packet_transmit_release(NX_PACKET *packet_ptr);
```

Description

For non-TCP packets, this service releases a transmitted packet, including any additional packets chained 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 after a packet is transmitted.

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.
- **NX_UNDERFLOW** (0x02) Prepend pointer is less than payload start.
- **NX_OVERFLOW** (0x03) Append pointer is greater than payload end.

Allowed From

Initialization, threads, timers, 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

\[
\text{UINT \ nx_rarp_disable(NX_IP *ip_ptr);}
\]

Description

This service disables the RARP component of NetX for the specific IP instance. For a multihome system, this service disables RARP on all interfaces.

Parameters

\[
ip\_ptr \quad \text{Pointer to previously created IP instance.}
\]

Return Values

\[
\begin{align*}
\text{NX_SUCCESS} \quad (0x00) & \quad \text{Successful RARP disable.} \\
\text{NX_NOT_ENABLED} \quad (0x14) & \quad \text{RARP was not enabled.} \\
\text{NX_PTR_ERROR} \quad (0x07) & \quad \text{Invalid IP pointer.} \\
\text{NX_CALLER_ERROR} \quad (0x11) & \quad \text{Invalid caller of this service.}
\end{align*}
\]

Allowed From

Initialization, threads

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

```c
UINT nx_rarp_enable(NX_IP *ip_ptr);
```

Description

This service enables the RARP component of NetX for the specific IP instance. The RARP components searches through all attached network interfaces for zero IP address. A zero IP address indicates the interface does not have IP address assignment yet. RARP attempts to resolve the IP address by enabling RARP process on 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
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 System

Prototype

VOID nx_system_initialize(VOID);

Description

This service initializes the basic NetX system resources in preparation for use. It should be called by the application during initialization and before any other NetX call are made.

Parameters

None

Return Values

None

Allowed From

Initialization, threads, timers, ISRs

Preemption Possible

No
Example

/* Initialize NetX for operation. */

\texttt{nx\_system\_initialize();}

/* At this point, NetX is ready for IP creation and all subsequent network operations. */

See Also

\texttt{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\_driver\_interface\_direct\_command, nx\_ip\_forwarding\_disable, nx\_ip\_forwarding\_enable, nx\_ip\_fragment\_disable, nx\_ip\_fragment\_enable, nx\_ip\_info\_get, nx\_ip\_status\_check}
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. If
the specified TCP port is unavailable, the service suspends according to
the supplied wait option.

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
                    0xFFFFFFFF)

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.
**Transmission Control Protocol (TCP)**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1A</td>
<td>Requested suspension was aborted by a call to tx_thread_wait_abort.</td>
</tr>
<tr>
<td>0x46</td>
<td>Invalid port.</td>
</tr>
<tr>
<td>0x07</td>
<td>Invalid socket pointer.</td>
</tr>
<tr>
<td>0x11</td>
<td>Invalid caller of this service.</td>
</tr>
<tr>
<td>0x14</td>
<td>This component has not been enabled.</td>
</tr>
</tbody>
</table>

**Allowed From**

Threads

**Preemption Possible**

No

**Example**

```c
/* 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_receive`
- `nx_tcp_socket_receive_queue_max_set`
- `nx_tcp_socket_send`
- `nx_tcp_socket_state_wait`
nx_tcp_client_socket_connect

Connect client TCP socket

Prototype

UINT nx_tcp_client_socket_connect(NX_TCP_SOCKET *socket_ptr,
    ULONG server_ip,
    UINT server_port,
    ULONG wait_option);

Description

This service connects the previously created and bound TCP client socket to the specified server's port. Valid TCP server ports range from 0 through 0xFFFF. If the connection does not complete immediately, the service suspends according to the supplied wait option.

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 0xFFFFFFFF)

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.
Transmission Control Protocol (TCP)

**Allowed From**
Threads

**Preemption Possible**
No

**Example**

```c
/* 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_disconnect`,
- `nx_tcp_socket_receive_queue_max_set`, `nx_tcp_socket_send`,
- `nx_tcp_socket_state_wait`
nx_tcp_client_socket_port_get

Get port number bound to client TCP socket

Prototype

```c
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 in situations where the
NX_ANY_PORT was specified at the time the socket was bound.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket_ptr</td>
<td>Pointer to previously created TCP socket instance.</td>
</tr>
<tr>
<td>port_ptr</td>
<td>Pointer to destination for the return port number. Valid port numbers are (1 through 0xFFFF).</td>
</tr>
</tbody>
</table>

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_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_receive,
nx_tcp_socket_receive_queue_max_set, nx_tcp_socket_send,
nx_tcp_socket_state_wait
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 same port number, the first suspended thread is then bound to this 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_receive, 
nx_tcp_socket_receive_queue_max_set, nx_tcp_socket_send, 
nx_tcp_socket_state_wait
nx_tcp_enable

Enable TCP component of NetX

Prototype

UINT nx_tcp_enable(NX_IP *ip_ptr);

Description

This service enables the Transmission Control Protocol (TCP) component of NetX. After enabled, TCP connections may be established 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 ip_0. */
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_receive,
nx_tcp_socket_receive_queue_max_set, nx_tcp_socket_send,
nx_tcp_socket_state_wait
nx_tcp_free_port_find

Find next available TCP port

Prototype

```c
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_receive,
nx_tcp_socket_receive_queue_max_set, nx_tcp_socket_send,
nx_tcp_socket_state_wait
nx_tlp_info_get

Retrieve information about TCP activities

Prototype

```c
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.

**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.
- **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.
Transmission Control Protocol (TCP)

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

Preemption Possible
No
Example

/* Retrieve TCP information from previously created IP Instance ip_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,
x_tcp_client_socket_port_get, nx_tcp_client_socket_unbind,
x_tcp_enable, nx_tcp_free_port_find, nx_tcp_server_socket_accept,
x_tcp_server_socket_listen, nx_tcp_server_socket_relisten,
x_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten,
x_tcp_socket_bytes_available, nx_tcp_socket_create,
x_tcp_socket_delete, nx_tcp_socket_disconnect,
x_tcp_socket_info_get, nx_tcp_socket_receive,
x_tcp_socket_receive_queue_max_set, nx_tcp_socket_send,
x_tcp_socket_state_wait
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. If a connection cannot not be established
right away, the service suspends according to the supplied wait option.

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
0xFFFFFFFF)

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.
Transmission Control Protocol (TCP)

**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**

Initialization, threads

**Preemption Possible**

No

**Example**

```c
    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 wake up 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. */
x_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. */
x_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. */
x_nx_tcp_server_socket_unlisten(&my_ip, 12);

/* Delete the server socket. */
x_nx_tcp_socket_delete(&server_socket);

See Also

nx_nx_tcp_client_socket_bind, nx_nx_tcp_client_socket_connect,
x_nx_tcp_client_socket_port_get, nx_nx_tcp_client_socket_unbind,
x_nx_tcp_enable, nx_nx_tcp_free_port_find, nx_nx_tcp_info_get,
x_nx_tcp_server_socket_listen, nx_nx_tcp_server_socket_relisten,
x_nx_tcp_server_socket_unaccept, nx_nx_tcp_server_socket_unlisten,
x_nx_tcp_socket_bytes_available, nx_nx_tcp_socket_create,
x_nx_tcp_socket_delete, nx_nx_tcp_socket_disconnect,
x_nx_tcp_socket_info_get, nx_nx_tcp_socket_receive,
x_nx_tcp_socket_receive_queue_max_set, nx_nx_tcp_socket_send,
x_nx_tcp_socket_state_wait
nx_tcp_server_socket_listen

Enable listening for client connection on TCP port

Prototype

```c
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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>0x00</td>
<td>Successful TCP port listen enable.</td>
</tr>
<tr>
<td>NX_MAX_LISTEN</td>
<td>0x33</td>
<td>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.</td>
</tr>
<tr>
<td>NX_NOT_CLOSED</td>
<td>0x35</td>
<td>The supplied server socket is not in a closed state.</td>
</tr>
<tr>
<td>NX_ALREADY_BOUND</td>
<td>0x22</td>
<td>The supplied server socket is already bound to a port.</td>
</tr>
<tr>
<td>NX_DUPLICATE_LISTEN</td>
<td>0x34</td>
<td>There is already an active listen request for this port.</td>
</tr>
<tr>
<td>NX_INVALID_PORT</td>
<td>0x46</td>
<td>Invalid port specified.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>0x07</td>
<td>Invalid IP or socket pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>0x11</td>
<td>Invalid caller of this service.</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>0x14</td>
<td>This component has not been enabled.</td>
</tr>
</tbody>
</table>

Allowed From

Threads

Preemption Possible

No

Example

```c
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 wake up 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_receive,
        nx_tcp_socket_receive_queue_max_set, nx_tcp_socket_send,
        nx_tcp_socket_state_wait
nx_tcp_server_socket_relisten

Re-listen for client connection on TCP port

Prototype

```c
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
Transmission Control Protocol (TCP)  

**Allowed From**

Threads

**Preemption Possible**

No

**Example**

```c
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 wake up 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,

    /* ...
```
/* Setup server listening on port 12. */
nx_tcp_server_socket_listen(&my_ip, 12, &server_socket);

/* 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_receive,  
nx_tcp_socket_receive_queue_max_set, nx_tcp_socket_send,  
nx_tcp_socket_state_wait
nx_tcp_server_socket_unaccept

Remove socket association with listening port

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

```c
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 wake up 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. */
x_packet_data_append(my_packet,
  "Hello_and_Goodbye",sizeof("Hello_and_Goodbye"),
  &my_pool, NX_WAIT_FOREVER);

/* Send "Hello_and_Goodbye" to client. */
x_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. */
x_tcp_socket_disconnect(&server_socket, 200);

/* Unaccept the server socket. Note that unaccept is called even
   if disconnect or accept fails. */
x_tcp_server_socket_unaccept(&server_socket);

/* Setup server socket for listening with this socket again. */
x_tcp_server_socket_relisten(&my_ip, 12, &server_socket);

/* We are now done so unlisten on server port 12. */
x_tcp_server_socket_unlisten(&my_ip, 12);

/* Delete the server socket. */
x_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_receive,
nx_tcp_socket_receive_queue_max_set, nx_tcp_socket_send,
nx_tcp_socket_state_wait
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 the specified 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

```c
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 wake up 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. */
x_nx_packet_data_append(my_packet, "Hello_and_Goodbye",
    sizeof("Hello_and_Goodbye"), &my_pool,
    NX_WAIT_FOREVER);

/* Send "Hello_and_Goodbye" to client. */
x_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. */
x_nx_tcp_socket_disconnect(&server_socket, 200);

/* Unaccept the server socket. Note that unaccept is called even if
disconnect or accept fails. */
x_nx_tcp_server_socket_unaccept(&server_socket);

/* Setup server socket for listening with this socket again. */
x_nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);

/* We are now done so unlisten on server port 12. */
x_nx_tcp_server_socket_unlisten(&my_ip, 12);

/* Delete the server socket. */
x_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_receive,
nx_tcp_socket_receive_queue_max_set, nx_tcp_socket_send,
nx_tcp_socket_state_wait
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 service obtains the 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_receive,
nx_tcp_socket_receive_queue_max_set, nx_tcp_socket_send,
nx_tcp_socket_state_wait
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 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS (0x00)</td>
<td>Successful TCP client socket create.</td>
</tr>
<tr>
<td>NX_OPTION_ERROR (0x0A)</td>
<td>Invalid type-of-service, fragment, invalid window size, or time-to-live option.</td>
</tr>
<tr>
<td>NX_PTR_ERROR (0x07)</td>
<td>Invalid IP or socket pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR (0x11)</td>
<td>Invalid caller of this service.</td>
</tr>
<tr>
<td>NX_NOT_ENABLED (0x14)</td>
<td>This component has not been enabled.</td>
</tr>
</tbody>
</table>

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_disconnect, 
nx_tcp_socket_receive, 
nx_tcp_socket_receive_queue_max_set, nx_tcp_socket_send, 
nx_tcp_socket_state_wait
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. If the socket is still bound or connected, the service returns an error code.

Parameters

socket_ptr
Previously created TCP socket

Return Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>(0x00) Successful socket delete.</td>
</tr>
<tr>
<td>NX_NOT_CREATED</td>
<td>(0x27) Socket was not created.</td>
</tr>
<tr>
<td>NX_STILL_BOUND</td>
<td>(0x42) Socket is still bound.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>(0x07) Invalid socket pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>(0x11) Invalid caller of this service.</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>(0x14) This component has not been enabled.</td>
</tr>
</tbody>
</table>

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_server_socket_unlisten,
    nx_tcp_socket_bytes_available,
    nx_tcp_socket_create, nx_tcp_socket_disconnect,
    nx_tcp_socket_info_get, nx_tcp_socket_receive,
    nx_tcp_socket_receive_queue_max_set, nx_tcp_socket_send,
    nx_tcp_socket_state_wait
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 unaccept request, while a client socket that is disconnected is left in a state ready for another connection request. If the disconnect process cannot finish immediately, the service suspends according to the supplied wait option.

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 0xFFFFFFFF)

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_receive, nx_tcp_socket_receive_queue_max_set,
nx_tcp_socket_send, nx_tcp_socket_state_wait
nx_tcp_socket_disconnect_complete_notify

Install TCP disconnect complete notify callback function

Prototype

```c
UINT nx_tcp_socket_disconnect_complete_notify(
    NX_TCP_SOCKET *socket_ptr,
    VOID (*tcp_disconnect_complete_notify)(NX_TCP_SOCKET *socket_ptr))
```

Description

This service registers a callback function which is invoked after a socket disconnect operation is completed. The TCP socket disconnect complete callback function is available if NetX is built with the option `NX_ENABLE_EXTENDED_NOTIFY_SUPPORT` defined.

Parameters

- **socket_ptr**: Pointer to previously connected client or server socket instance.
- **tcp_disconnect_complete_notify**: The callback function to be installed.

Return Values

- **NX_SUCCESS**: (0x00) Successfully registered the callback function.
- **NX_NOT_SUPPORTED**: (0x4B) The extended notify feature is not built into the NetX library.
- **NX_PTR_ERROR**: (0x07) Invalid socket pointer.
- **NX_CALLER_ERROR**: (0x11) Invalid caller of this service.
- **NX_NOT_ENABLED**: (0x14) TCP feature is not enabled.

Allowed From

- Initialization, threads

Preemption Possible

- No
Example

/* Install the disconnect complete notify callback function. */
status = nx_tcp_socket_disconnect_complete_notify(&client_socket, callback);

See Also

nx_tcp_enable, nx_tcp_socket_create, nx_tcp_socket_establish_notify,
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_timed_wait_callback,
nx_tcp_socket_transmit_configure,
nx_tcp_socket_window_update_notify_set
nx_tcp_socket_establish_notify

Set TCP establish notify callback function

Prototype

UINT nx_tcp_socket_establish_notify(NX_TCP_SOCKET *socket_ptr,
                                    VOID (*tcp_establish_notify)(NX_TCP_SOCKET *socket_ptr))

Description

This service registers a callback function, which is called after a TCP socket makes a connection. The TCP socket establish callback function is available if NetX is built with the option

NX_ENABLE_EXTENDED_NOTIFY_SUPPORT defined.

Parameters

socket_ptr Pointer to previously connected client or server socket instance.
tcp_establish_notify Callback function invoked after a TCP connection is established.

Return Values

NX_SUCCESS (0x00) Successfully sets the notify function.

NX_NOT_SUPPORTED (0x4B) The extended notify feature is not built into the NetX library

NX_PTR_ERROR (0x07) Invalid socket pointer.

NX_CALLER_ERROR (0x11) Invalid caller of this service.

NX_NOT_ENABLED (0x14) TCP has not been enabled by the application.

Allowed From

Threads

Preemption Possible

No
Example

/* Set the function pointer "callback" as the notify function NetX will call when the connection is in the established state. */
status = nx_tcp_socket-establish_notify(&client_socket, callback);

See Also

nx_tcp_enable, nx_tcp_socket_create,
nx_tcp_socket_disconnect_complete_notify, 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_timed_wait_callback, 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket_ptr</td>
<td>Pointer to previously created TCP socket instance.</td>
</tr>
<tr>
<td>tcp_packets_sent</td>
<td>Pointer to destination for the total number of TCP packets sent on socket.</td>
</tr>
<tr>
<td>tcp_bytes_sent</td>
<td>Pointer to destination for the total number of TCP bytes sent on socket.</td>
</tr>
<tr>
<td>tcp_packets_received</td>
<td>Pointer to destination of the total number of TCP packets received on socket.</td>
</tr>
<tr>
<td>tcp_bytes_received</td>
<td>Pointer to destination of the total number of TCP bytes received on socket.</td>
</tr>
<tr>
<td>tcp_retransmit_packets</td>
<td>Pointer to destination of the total number of TCP packet retransmissions.</td>
</tr>
<tr>
<td>tcp_packets_queued</td>
<td>Pointer to destination of the total number of queued TCP packets on socket.</td>
</tr>
<tr>
<td>tcp_checksum_errors</td>
<td>Pointer to destination of the total number of TCP packets with checksum errors on socket.</td>
</tr>
<tr>
<td>tcp_socket_state</td>
<td>Pointer to destination of the socket's current state.</td>
</tr>
<tr>
<td>tcp_transmit_queue_depth</td>
<td>Pointer to destination of the total number of transmit packets still queued waiting for ACK.</td>
</tr>
<tr>
<td>tcp_transmit_window</td>
<td>Pointer to destination of the current transmit window size.</td>
</tr>
<tr>
<td>tcp_receive_window</td>
<td>Pointer to destination of the current receive window size.</td>
</tr>
</tbody>
</table>
Return Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS (0x00)</td>
<td>Successful TCP socket information retrieval.</td>
</tr>
<tr>
<td>NX_PTR_ERROR (0x07)</td>
<td>Invalid socket pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR (0x11)</td>
<td>Invalid caller of this service.</td>
</tr>
<tr>
<td>NX_NOT_ENABLED (0x14)</td>
<td>This component has not been enabled.</td>
</tr>
</tbody>
</table>

Allowed From
Initialization, threads

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_disconnect,
nx_tcp_socket_receive, nx_tcp_socket_receive_queue_max_set,
nx_tcp_socket_send, nx_tcp_socket_state_wait
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 local 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

Preemption Possible
No

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_enable, nx_tcp_socket_create,
nx_tcp_socket_disconnect_complete_notify,
nx_tcp_socket_establish_notify, 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_timed_wait_callback,
nx_tcp_socket_transmit_configure,
nx_tcp_socket_window_update_notify_set
nx_tcp_socket_mss_peer_get

Get MSS of the peer TCP socket

Prototype

```
UINT nx_tcp_socket_mss_peer_get(NX_TCP_SOCKET *socket_ptr,
                                ULONG *mss);
```

Description

This service retrieves the Maximum Segment Size (MSS) advertised by the peer socket.

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.
Transmission Control Protocol (TCP)

Allowed From
Threads

Preemption Possible
No

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_enable, nx_tcp_socket_create,
nx_tcp_socket_disconnect_complete_notify,
nx_tcp_socket-establish_notify, nx_tcp_socket_mss_get,
nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get,
nx_tcp_socket_receive_notify, nx_tcp_socket_timed_wait_callback,
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 IP MTU, allowing room for IP and TCP headers.

This service should be used before a TCP socket starts the connection process. If the service is used after a TCP connection is established, the new value has no effect on the connection.

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_NOT_CONNECTED (0x38) TCP connection has not been established
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

Preemption Possible
No

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_enable, nx_tcp_socket_create,
nx_tcp_socket_disconnect_complete_notify,
nx_tcp_socket_establish_notify, nx_tcp_socket_mss_get,
nx_tcp_socket_mss_peer_get, nx_tcp_socket_peer_info_get,
nx_tcp_socket_receive_notify, nx_tcp_socket_timed_wait_callback,
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 peer IP address and port information for the connected TCP socket over IP network.

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

No
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_enable, nx_tcp_socket_create,
nx_tcp_socket_disconnect_complete_notify,
nx_tcp_socket_establish_notify, nx_tcp_socket_mss_get,
nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set,
nx_tcp_socket_receive_notify, nx_tcp_socket_timed_wait_callback,
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 no 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 0xFFFFFFFFE)

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

No

### 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_receive_queue_max_set`
- `nx_tcp_socket_send`, `nx_tcp_socket_state_wait`
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 configures the receive notify function pointer with 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.
- **NX_CALLER_ERROR**: (0x11) Invalid caller of this service.
- **NX_NOT_ENABLED**: (0x14) TCP feature is not enabled.

Allowed From

Initialization, threads

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 will call the function named "my_receive_notify" whenever one or more packets are received for "client_socket". */

See Also

nx_tcp_enable, nx_tcp_socket_create,
nx_tcp_socket_disconnect_complete_notify,
nx_tcp_socket_establish_notify, 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_timed_wait_callback,
nx_tcp_socket_transmit_configure,
nx_tcp_socket_window_update_notify_set
nx_tcp_socket_send

Send data through a TCP socket

Prototype

```c
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 option 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 0xFFFFFFFFF)
### Return Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NX_SUCCESS</strong></td>
<td>(0x00) Successful socket send.</td>
</tr>
<tr>
<td><strong>NX_NOT_BOUND</strong></td>
<td>(0x24) Socket was not bound to any port.</td>
</tr>
<tr>
<td><strong>NX_NO_INTERFACE_ADDRESS</strong></td>
<td>(0x50) No suitable outgoing interface found.</td>
</tr>
<tr>
<td><strong>NX_NOT_CONNECTED</strong></td>
<td>(0x38) Socket is no longer connected.</td>
</tr>
<tr>
<td><strong>NX_WINDOW_OVERFLOW</strong></td>
<td>(0x39) Request is greater than receiver’s advertised window size in bytes.</td>
</tr>
<tr>
<td><strong>NX_WAIT_ABORTED</strong></td>
<td>(0x1A) Requested suspension was aborted by a call to <code>tx_thread_wait_abort</code>.</td>
</tr>
<tr>
<td><strong>NX_INVALID_PACKET</strong></td>
<td>(0x12) Packet is not allocated.</td>
</tr>
<tr>
<td><strong>NX_TX_QUEUE_DEPTH</strong></td>
<td>(0x49) Maximum transmit queue depth has been reached.</td>
</tr>
<tr>
<td><strong>NX_OVERFLOW</strong></td>
<td>(0x03) Packet append pointer is invalid.</td>
</tr>
<tr>
<td><strong>NX_PTR_ERROR</strong></td>
<td>(0x07) Invalid socket pointer.</td>
</tr>
<tr>
<td><strong>NX_CALLER_ERROR</strong></td>
<td>(0x11) Invalid caller of this service.</td>
</tr>
<tr>
<td><strong>NX_NOT_ENABLED</strong></td>
<td>(0x14) This component has not been enabled.</td>
</tr>
<tr>
<td><strong>NX_UNDERFLOW</strong></td>
<td>(0x02) Packet prepend pointer is invalid.</td>
</tr>
</tbody>
</table>
**Allowed From**

Threads

**Preemption Possible**

No

**Example**

```c
/* 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_receive,
- nx_tcp_socket_receive_queue_max_set, nx_tcp_socket_state_wait
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. If the socket is
not in the desired state, the service suspends according to the supplied
wait option.

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)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- timeout value (0x00000001 through
  0xFFFFFFFF)

Return Values

NX_SUCCESS (0x00) Successful state wait.

NX_PTR_ERROR (0x07) Invalid socket pointer.
**Transmission Control Protocol (TCP)**

**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**

No

**Example**

```c
/* 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_receive`,  
`nx_tcp_socket_receive_queue_max_set`, `nx_tcp_socket_send`
nx_tcp_socket_timed_wait_callback

Install callback for timed wait state

Prototype

```c
UINT nx_tcp_socket_timed_wait_callback(NX_TCP_SOCKET *socket_ptr,
                                          VOID (*tcp_timed_wait_callback)(NX_TCP_SOCKET *socket_ptr))
```

Description

This service registers a callback function which is invoked when the TCP socket is in timed wait state. To use this service, the NetX library must be built with the option `NX_ENABLE_EXTENDED_NOTIFY` defined.

Parameters

- `socket_ptr` Pointer to previously connected client or server socket instance.
- `tcp_timed_wait_callback` The timed wait callback function

Return Values

- `NX_SUCCESS` (0x00) Successfully registers the callback function socket
- `NX_NOT_SUPPORTED` (0x4B) NetX library is built without the extended notify feature enabled.
- `NX_PTR_ERROR` (0x07) Invalid socket pointer.
- `NX_CALLER_ERROR` (0x11) Invalid caller of this service.
- `NX_NOT_ENABLED` (0x14) TCP feature is not enabled.

Allowed From

Initialization, threads

Preemption Possible

No
Example

/* Install the timed wait callback function */
#include "nx/tcp/socket.h"

nx_tcp_socket_timed_wait_callback(client_socket, callback);

See Also

nx_tcp_enable, nx_tcp_socket_create,
x_tcp_socket_disconnect_complete_notify,
x_tcp_socket_establish_notify, nx_tcp_socket_mss_get,
x_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set,
x_tcp_socket_peer_info_get, nx_tcp_socket_receive_notify,
x_tcp_socket_transmit_configure,
x_tcp_socket_window_update_notify_set
nx_tcp_socket_transmit_configure

Configure socket’s transmit parameters

Prototype

UINT \texttt{nx\_tcp\_socket\_transmit\_configure}(NX\_TCP\_SOCKET *socket\_ptr, 
\hspace{1em} ULONG max\_queue\_depth, 
\hspace{1em} ULONG timeout, 
\hspace{1em} ULONG max\_retries, 
\hspace{1em} ULONG timeout\_shift);

Description

This service configures various transmit parameters of the specified TCP socket.

Parameters

\begin{itemize}
  \item \texttt{socket\_ptr} Pointer to the TCP socket.
  \item \texttt{max\_queue\_depth} Maximum number of packets allowed to be queued for transmission.
  \item \texttt{timeout} Number of ThreadX timer ticks an ACK is waited for before the packet is sent again.
  \item \texttt{max\_retries} Maximum number of retries allowed.
  \item \texttt{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.
\end{itemize}

Return Values

\begin{itemize}
  \item \texttt{NX\_SUCCESS} (0x00) Successful transmit socket configure.
  \item \texttt{NX\_PTR\_ERROR} (0x07) Invalid socket pointer.
  \item \texttt{NX\_OPTION\_ERROR} (0x0a) Invalid queue depth option.
  \item \texttt{NX\_CALLER\_ERROR} (0x1a) Invalid caller of this service.
  \item \texttt{NX\_NOT\_ENABLED} (0x14) TCP feature is not enabled.
\end{itemize}

Allowed From

Initialization, threads
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_enable, nx_tcp_socket_create,
nx_tcp_socket_disconnect_complete_notify,
nx_tcp_socket_establish_notify, 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_timed_wait_callback,
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

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(NX_TCP_SCOCKET *data_socket)
{
    /* Process update on increase TCP transmit socket window size. */
    return;
}

See Also

nx_tcp_enable, nx_tcp_socket_create,
nx_tcp_socket_disconnect_complete_notify,
nx_tcp_socket_establish_notify, 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_timed_wait_callback, nx_tcp_socket_transmit_configure
nx_udp_enable

Enable UDP component of NetX

Prototype

UINT nx_udp_enable(NX_IP *ip_ptr);

Description

This service enables the User Datagram Protocol (UDP) component of NetX. 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_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_port_get, nx_udp_socket_receive,
nx_udp_socket_receive_notify, nx_udp_socket_send,
nx_udp_socket_interface_send, nx_udp_socket_unbind,
nx_udp_source_extract
nx_udp_free_port_find

Find next available UDP port

Prototype

```c
UINT nx_udp_free_port_find(NX_IP *ip_ptr, UINT port,
                           UINT *free_port_ptr);
```

Description

This service looks for a free UDP port (unbound) starting from the
application supplied port number. The search logic will wrap around if the
search reaches 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

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_port_get, nx_udp_socket_receive,
nx_udp_socket_receive_notify, nx_udp_socket_send,
nx_udp_socket_interface_send, nx_udp_socket_unbind,
nx_udp_source_extract
nx_udp_info_get

Retrieve information about UDP activities

Prototype

```c
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.

**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.
- **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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>0x00</td>
<td>Successful UDP information retrieval.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>0x07</td>
<td>Invalid IP pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>0x11</td>
<td>Invalid caller of this service.</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>0x14</td>
<td>This component has not been enabled.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```c
/* Retrieve UDP information from previously created IP Instance ip_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_port_get, nx_udp_socket_receive,
nx_udp_socket_receive_notify, nx_udp_socket_send,
nx_udp_socket_interface_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 service extracts network parameters, such as IP address, peer port number, protocol type (this service always returns UDP type) 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 IP frame.
- NX_PTR_ERROR (0x07): Invalid pointer input
- NX_CALLER_ERROR (0x11): Invalid caller of this service.

Allowed From

Threads

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_port_get, nx_udp_socket_receive,
nx_udp_socket_receive_notify, nx_udp_socket_send,
nx_udp_socket_interface_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. If the
requested port number is bound to another socket, this service waits for
specified period of time for the socket to unbind from the port number.

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
0xFFFFFFFF)

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
No

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_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_interface_send, nx_udp_socket_unbind, nx_udp_source_extract
nx_udp_socket_bytes_available

Retrieves number of bytes available for retrieval

Prototype

```c
UINT nx_udp_socket_bytes_available(NX_UDP_SOCKET *socket_ptr,
ULONG *bytes_available);
```

Description

This service retrieves number of bytes available for reception 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_port_get, nx_udp_socket_receive,
nx_udp_socket_receive_notify, nx_udp_socket_send,
nx_udp_socket_interface_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 value in the UDP header signals the receiver that checksum is not computed for this packet.

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>(0x00) Successful socket checksum disable.</td>
</tr>
<tr>
<td>NX_NOT_BOUND</td>
<td>(0x24) Socket is not bound.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>(0x07) Invalid socket pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>(0x11) Invalid caller of this service.</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>(0x14) This component has not been enabled.</td>
</tr>
</tbody>
</table>

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_port_get, nx_udp_socket_receive,
nx_udp_socket_receive_notify, nx_udp_socket_send,
nx_udp_socket_interface_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.

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,
x_udp_packet_info_extract, nx_udp_socket_bind,
x_udp_socket_bytes_available, nx_udp_socket_checksum_disable,
x_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get,
x_udp_socket_port_get, nx_udp_socket_receive,
x_udp_socket_receive_notify, nx_udp_socket_send,
x_udp_socket_interface_send, nx_udp_socket_unbind,
x_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)
- 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.
- 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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_SUCCESS</td>
<td>0x00</td>
<td>Successful UDP socket create.</td>
</tr>
<tr>
<td>NX_OPTION_ERROR</td>
<td>0x0A</td>
<td>Invalid type-of-service, fragment, or time-to-live option.</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>0x07</td>
<td>Invalid IP or socket pointer.</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>0x11</td>
<td>Invalid caller of this service.</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>0x14</td>
<td>This component has not been enabled.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization and Threads

Preemption Possible

No

Example

```c
/* 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_port_get,
nx_udp_socket_receive, nx_udp_socket_receive_notify,
nx_udp_socket_send, nx_udp_socket_interface_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. If the socket was bound to a port, the socket must be unbound first.

Parameters

socket_ptr Pointer to previously created UDP socket instance.

Return Values

NX_SUCCESS (0x00) Successful socket delete.
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_port_get,
nx_udp_socket_receive, nx_udp_socket_receive_notify,
nx_udp_socket_send, nx_udp_socket_interface_send,
nx_udp_socket_unbind, nx_udp_source_extract
nx_udp_socket_info_get

Retrieve information about UDP socket activities

Prototype

```c
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 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,
rx_udp_socket_bytes_available, nx_udp_socket_checksum_disable,
rx_udp_socket_checksum_enable, nx_udp_socket_create,
rx_udp_socket_delete, nx_udp_socket_port_get,
rx_udp_socket_receive, nx_udp_socket_receive_notify,
rx_udp_socket_send, nx_udp_socket_interface_send,
rx_udp_socket_unbind, nx_udp_source_extract
nx_udp_socket_port_get

Pick up port number bound to UDP socket

Prototype

```c
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 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**

Preemption Possible

- **No**
Example

/* Get the port number of 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_receive, nx_udp_socket_receive_notify,
nx_udp_socket_send, nx_udp_socket_interface_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
0xFFFFFFFFF)

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.
User Datagram Protocol (UDP)

7

Allowed From
Threads

Preemption Possible
No

Example

/* Receive a packet from a previously created and bound UDP socket. */
if (status == NX_SUCCESS)

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_port_get, nx_udp_socket_receive_notify,
nx_udp_socket_send, nx_udp_socket_interface_send,
nx_udp_socket_unbind, nx_udp_source_extract
nx_udp_socket_receive_notify

Notify application of each received packet

Prototype

```c
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) Successfully set socket receive notify function.
- **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 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_port_get, nx_udp_socket_receive, nx_udp_socket_send,
nx_udp_socket_interface_send, nx_udp_socket_unbind,
nx_udp_source_extract
nx_udp_socket_send

Send a UDP Datagram

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. NetX finds a suitable local IP address as source
address based on the destination IP address. To specify a specific
interface and source IP address, the application should use the
nx_udp_socket_interface_send service.

Note that this service returns immediately regardless of whether the UDP
datagram was successfully sent.

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**: Destination IP address
- **port**: Valid destination port number between 1 and
  0xFFFF), in host byte order

Return Values

- **NX_SUCCESS** (0x00): Successful UDP socket send
- **NX_NOT_BOUND** (0x24): Socket not bound to any port
- **NX_NO_INTERFACE_ADDRESS** (0x50): No suitable outgoing interface
can be found.
- **NX_IP_ADDRESS_ERROR** (0x21): Invalid server IP address
- **NX_UNDERFLOW** (0x02): Not enough room for UDP header in the packet
User Datagram Protocol (UDP)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_OVERFLOW</td>
<td>(0x03) Packet append pointer is invalid</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>(0x07) Invalid socket pointer</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>(0x11) Invalid caller of this service</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>(0x14) UDP has not been enabled</td>
</tr>
<tr>
<td>NX_INVALID_PORT</td>
<td>(0x46) Port number is not within a valid range</td>
</tr>
</tbody>
</table>

**Allowed From**

Threads

**Preemption Possible**

No

**Example**

```c
ULONG server_address;

/* Set the UDP Client IP address. */
server_address = IP_ADDRESS(1,2,3,5);

/* Send a packet to the UDP server at server_address on port 12. */
status = nx_udp_socket_send(&client_socket, packet_ptr,
                            server_address, 12);

/* If status == NX_SUCCESS, the application successfully transmitted
the packet out the UDP socket to its peer. */
```

**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_port_get, nx_udp_socket_receive,
nx_udp_socket_receive_notify, nx_udp_socket_interface_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 address_index);

Description

This service sends a UDP datagram through a previously created and bound UDP socket through the network interface with the specified IP address as the source address. 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.
- address_index: Index of the address associated with the 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 address index.
NX_INVALID_PORT (0x46) Port number exceeds maximum port number.

Allowed From
Threads

Preemption Possible
No

Example

#define ADDRESS_INDEX 1

/* 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, ADDRESS_INDEX);

/* 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_bytes_available, nx_udp_socketChecksumDisable,
nx_udp_socketChecksumEnable, nx_udp_socketCreate,
nx_udp_socket_delete, nx udp_socketInfoGet,
nx_udp_socket_port_get, nx_udp_socket_receive,
nx_udp_socket_receive_notify, nx_udp_socket_send,
nx_udp_socket_unbind
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. Any received packets stored in the receive queue are released as part of the unbind operation.

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_port_get, nx_udp_socket_receive,
nx_udp_socket_receive_notify, nx_udp_socket_send,
nx_udp_socket_interface_send, nx_udp_source_extract
nx_udp_source_extract

Extract IP and sending port from UDP datagram

Prototype

```c
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`: Valid pointer to the return IP address variable.
- `port`: Valid 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

Initialization, threads, timers, ISR

Preemption Possible

No
Example

/* Extract the IP and port information from the sender of the UDP packet. */
status = nx_udp_source_extract(packet_ptr, &sender_ip_address, &sender_port);

/* If status is NX_SUCCESS, the sender 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_port_get, nx_udp_socket_receive,
nx_udp_socket_receive_notify, nx_udp_socket_send,
nx_udp_socket_interface_send, nx_udp_socket_unbind
NetX Network Drivers

This chapter contains a description of network drivers for NetX. The information presented is designed to help developers write application-specific network drivers for NetX. The following topics are covered:

- Driver Introduction 354
- Driver Entry 354
- Driver Requests 355
  - Driver Initialization 355
  - Enable Link 357
  - Disable Link 357
  - Uninitialize Link 358
  - Packet Send 359
  - Packet Broadcast 360
  - ARP Send 361
  - ARP Response Send 361
  - RARP Send 362
  - Multicast Group Join 363
  - Multicast Group Leave 364
  - Attach Interface 364
  - Detach Interface 365
  - Get Link Status 365
  - Get Link Speed 366
  - Get Duplex Type 367
  - Get Error Count 367
  - Get Receive Packet Count 368
  - Get Transmit Packet Count 369
  - Get Allocation Errors 369
  - Driver Deferred Processing 370
  - User Commands 370
  - Unimplemented Commands 371
- Driver Output 371
- Driver Input 372
  - Deferred Receive Packet Handling 374
- Example RAM Ethernet Network Driver 374
Driver Introduction

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 \texttt{nx_ip_create} service. Additional devices may be added to the IP instance via the \texttt{nx_ip_interface_attach} service.

Communication between NetX and the application’s network driver is accomplished through the \texttt{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:

```c
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 invokes the network driver entry function for driver initialization and for sending packets and for various control and status operations, including initializing and enabling the network device. NetX issues commands to the network driver by setting the \texttt{nx_ip_driver_command} field in the \texttt{NX_IP_DRIVER} request structure. The driver entry function has the following format:

```c
VOID my_driver_entry(NX_IP_DRIVER *request);
```
Driver Requests

NetX creates the driver request with a specific command and invokes the driver entry function to execute the command. Because each network driver has a single entry function, NetX 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 serializes all access to the driver. Therefore, the driver does not need to handle multiple threads asynchronously calling the entry function. Note that the device driver function executes with the IP mutex locked. Therefore the device driver internal function shall not block itself.

Typically the device driver also handles interrupts. Therefore, all driver functions need to be interrupt-safe.

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 for driver initialization is the IP Maximum Transmission Unit (MTU), which is the number of bytes available to the IP-layer payload, including IP header) and if the physical interface needs logical-to-physical mapping. The driver needs to configure the interface MTU by setting the value in *nx_interface_ip_mtu_size* in the **NX_INTERFACE** structure.

The device driver also needs to set up the value in *nx_ip_interface_address_mapping_needed* in **NX_INTERFACE** to inform NetX whether or not interface address mapping is required. If address
mapping is needed, the driver is responsible for configuring the interface with valid MAC address, and supply the MAC address to NetX.

When the network driver receives the NX_LINK INITIALIZE request from NetX, it receives a pointer to the IP control block as part of the NX_IP_DRIVER request control block shown above.

After the application calls `nx_ip_create`, the IP helper thread sends a driver request 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.

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td>NX_LINK_INITIALIZE</td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to the IP instance. This value should be saved by the driver so that the driver function can find the IP instance to operate on.</td>
</tr>
<tr>
<td>nx_ip_driver_interface</td>
<td>Pointer to the network interface structure within the IP instance. This information should be saved by the driver. On receiving packets, the driver shall use the interface structure information when sending the packet up the stack.</td>
</tr>
<tr>
<td>nx_ip_driver_status</td>
<td>Completion status. If the driver is not able to initialize the specified interface to the IP instance, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>

Most of the driver commands are called from the IP helper thread that was created for the IP instance. Therefore the driver routine should avoid performing blocking operations, or the IP helper thread could...
stall, causing unbounded delays to applications that rely on the IP thread.

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:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td>NX_LINK_ENABLE</td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td>nx_ip_driver_interface</td>
<td>Pointer to the interface instance</td>
</tr>
<tr>
<td>nx_ip_driver_status</td>
<td>Completion status. If the driver is not able to enable the specified interface, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>

Disable Link

This request is made by NetX during the deletion of an IP instance by the nx_ip_delete service. Or an application may issue this command in order to temporarily disable the link in order to save power. This service disables the 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. Typically it is a reverse operation of the
Enable Link operation. After the link is disabled, the application request Enable Link operation to enable the interface.

The following NX_IP_DRIVER members are used for the disable link request:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td>NX_LINK_DISABLE</td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td>nx_ip_driver_interface</td>
<td>Pointer to the interface instance</td>
</tr>
<tr>
<td>nx_ip_driver_status</td>
<td>Completion status. If the driver is not able to disable the specified interface in the IP instance, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>

Uninitialize Link

This request is made by NetX during the deletion of an IP instance by the nx_ip_delete service. This request uninitialize the interface, and release any resources created during initialization phase. Typically it is a reverse operation of the Initialize Link operation. After the interface is uninitialize, the device cannot be used until the interface is initialized again.

The following NX_IP_DRIVER members are used for the disable link request:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td>NX_LINK_UNINITIALIZE</td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to IP instance</td>
</tr>
</tbody>
</table>
Packet Send

This request is made during internal IP send processing, which all NetX protocols use to transmit packets (except for ARP, RARP). On receiving the packet send command, the `nx_packet_prepend_ptr` points to the beginning of the packet to be sent, which is the beginning of the IP header. `nx_packet_length` indicates the total size (in bytes) of the data being transmitted. If `nx_packet_next` is valid, the outgoing IP datagram is stored in multiple packets, the driver is required to follow the chained packet and transmit the entire frame. Note that valid data area in each chained packet is stored between `nx_packet_prepend_ptr` and `nx_packet_append_ptr`.

The driver is responsible for constructing physical header. If physical address to IP address mapping is required (such as Ethernet), the IP layer already resolved the MAC address. The destination MAC address is passed from the IP instance, stored in `nx_ip_driver_physical_address_msw` and `nx_ip_driver_physical_address_lsw`.

After adding the physical header, the packet send processing then calls the driver’s output function to transmit the packet.

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx_ip_driver_interface</code></td>
<td>Pointer to the interface instance</td>
</tr>
<tr>
<td><code>nx_ip_driver_status</code></td>
<td>Completion status. If the driver is not able to uninitialize the specified interface to the IP instance, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>
The following NX_IP_DRIVER members are used for the packet send request:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td>NX_LINK_PACKET_SEND</td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td>nx_ip_driver_packet</td>
<td>Pointer to the packet to send</td>
</tr>
<tr>
<td>nx_ip_driver_interface</td>
<td>Pointer to the interface instance.</td>
</tr>
<tr>
<td>nx_ip_driver_physical_address_msw</td>
<td>Most significant 32-bits of physical address (only if physical mapping needed)</td>
</tr>
<tr>
<td>nx_ip_driver_physical_address_lsw</td>
<td>Least significant 32-bits of physical address (only if physical mapping needed)</td>
</tr>
<tr>
<td>nx_ip_driver_status</td>
<td>Completion status. If the driver is not able to send the packet, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>

Packet Broadcast

This request is almost identical to the send packet request. The only difference is that the destination physical address fields are set to the Ethernet broadcast MAC address. The following NX_IP_DRIVER members are used for the packet broadcast request:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td>NX_LINK_PACKET_BROADCAST</td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td>nx_ip_driver_packet</td>
<td>Pointer to the packet to send</td>
</tr>
<tr>
<td>nx_ip_driver_physical_address_msw</td>
<td>0x0000FFFF (broadcast)</td>
</tr>
<tr>
<td>nx_ip_driver_physical_address_lsw</td>
<td>0xFFFFFFFF (broadcast)</td>
</tr>
</tbody>
</table>
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 destination physical address fields are set to MAC broadcast address. The following `NX_IP_DRIVER` members are used for the ARP send request:

<table>
<thead>
<tr>
<th><code>NX_IP_DRIVER</code> member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx_ip_driver_command</code></td>
<td><code>NX_LINK_ARP_SEND</code></td>
</tr>
<tr>
<td><code>nx_ip_driver_ptr</code></td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td><code>nx_ip_driver_packet</code></td>
<td>Pointer to the packet to send</td>
</tr>
<tr>
<td><code>nx_ip_driver_physical_address_msw</code></td>
<td>0x0000FFFF (broadcast)</td>
</tr>
<tr>
<td><code>nx_ip_driver_physical_address_lsw</code></td>
<td>0xFFFFFFFF (broadcast)</td>
</tr>
<tr>
<td><code>nx_ip_driver_interface</code></td>
<td>Pointer to the interface instance</td>
</tr>
<tr>
<td><code>nx_ip_driver_status</code></td>
<td>Completion status. If the driver is not able to send the ARP packet, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>

If physical mapping is not needed, implementation of this request is not required.

ARP Response Send

This request is almost identical to the ARP send packet request. The only difference is the destination physical address fields are passed from the IP instance. The following `NX_IP_DRIVER` members
are used for the ARP response send request:

**NX_IP_DRIVER member**

nx_ip_driver_command | Meaning
---------------------|-------------------
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_physical_address_lsw | Least significant 32-bits of physical address

nx_ip_driver_interface | Pointer to the interface instance

nx_ip_driver_status | Completion status. If the driver is not able to send the ARP packet, it will return a non-zero error status.

*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 address.

The following NX_IP_DRIVER members are used for the RARP send request:

**NX_IP_DRIVER member**

nx_ip_driver_command | Meaning
---------------------|-------------------
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)
Applications that require RARP service must implement this command.

Multicast Group Join

This request is made with the \texttt{nx\_igmp\_multicast\_interface\_join} service. The network driver takes the supplied multicast group address and sets up the physical media to accept incoming packets from that multicast group address. Note that for drivers that don’t support multicast filter, the driver receive logic may have to be in promiscuous mode. In this case, the driver may need to filter incoming frames based on destination MAC address, thus reducing the amount of traffic passed into the IP instance. The following NX\_IP\_DRIVER members are used for the multicast group join request.

\begin{itemize}
\item \texttt{nx\_ip\_driver\_command} - NX\_LINK\_MULTICAST\_JOIN
\item \texttt{nx\_ip\_driver\_ptr} - Pointer to IP instance
\item \texttt{nx\_ip\_driver\_physical\_address\_msw} - Most significant 32-bits of physical multicast address
\item \texttt{nx\_ip\_driver\_physical\_address\_lsw} - Least significant 32-bits of physical multicast address
\item \texttt{nx\_ip\_driver\_interface} - Pointer to the interface instance
\item \texttt{nx\_ip\_driver\_status} - Completion status. If the driver is not able to join the multicast group, it will return a non-zero error status.
\end{itemize}
Multicast Group Leave

This request is invoked by explicitly calling the `nx_igmp_multicast_leave` service. The driver removes the supplied Ethernet multicast address from the multicast 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:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx_ip_driver_command</code></td>
<td><code>NX_LINK_MULTICAST_LEAVE</code></td>
</tr>
<tr>
<td><code>nx_ip_driver_ptr</code></td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td><code>nx_ip_driver_physical_address_msw</code></td>
<td>Most significant 32 bits of physical multicast address</td>
</tr>
<tr>
<td><code>nx_ip_driver_physical_address_lsw</code></td>
<td>Least significant 32 bits of physical multicast address</td>
</tr>
<tr>
<td><code>nx_ip_driver_interface</code></td>
<td>Pointer to the interface instance</td>
</tr>
<tr>
<td><code>nx_ip_driver_status</code></td>
<td>Completion status. If the driver is not able to leave the multicast group, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>

Attach Interface

This request is invoked from the NetX 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:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx_ip_driver_command</code></td>
<td><code>NX_LINK_INTERFACE_ATTACH</code></td>
</tr>
<tr>
<td><code>nx_ip_driver_ptr</code></td>
<td>Pointer to IP instance</td>
</tr>
</tbody>
</table>
Detach Interface

This request is invoked by NetX to the device driver, allowing the driver to disassociate 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:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td>NX_LINK_INTERFACE_DETACH</td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td>nx_ip_driver_interface</td>
<td>Pointer to the interface instance. Completion status. If the driver is not able to detach the specified interface to the IP instance, it will return a non-zero error status.</td>
</tr>
<tr>
<td>nx_ip_driver_status</td>
<td>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.</td>
</tr>
</tbody>
</table>

Get Link Status

The application can query the network interface link status using the NetX service `nx_ip_interface_status_check` service for any interface on the host. See Chapter 4, “Description of NetX Services” on page 107, for more details on these services.

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:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td>NX_LINK_GET_STATUS</td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td>nx_ip_driver_return_ptr</td>
<td>Pointer to the destination to place the status.</td>
</tr>
<tr>
<td>nx_ip_driver_interface</td>
<td>Pointer to the interface instance</td>
</tr>
<tr>
<td>nx_ip_driver_status</td>
<td>Completion status. If the driver is not able to get specific status, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>

\[
\text{nx_ip_status_check is still available for checking the status of the primary interface. However, application developers are encouraged to use the interface specific service: nx_ip_interface_status_check.}
\]

Get Link Speed

This request is made from within the \text{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:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td>NX_LINK_GET_SPEED</td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td>nx_ip_driver_return_ptr</td>
<td>Pointer to the destination to place the line speed</td>
</tr>
</tbody>
</table>
**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:

<table>
<thead>
<tr>
<th>NX_IP.Driver member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td><code>NX_LINK_GET_DUPLEX_TYPE</code></td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td>nx_ip_driver_return_ptr</td>
<td>Pointer to the destination to place the duplex type</td>
</tr>
<tr>
<td>nx_ip_driver_interface</td>
<td>Pointer to the interface instance</td>
</tr>
<tr>
<td>nx_ip_driver_status</td>
<td>Completion status. If the driver is not able to get duplex information, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>

This request is not used internally by NetX 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

---

*Express Logic*
destination. To support this feature, the driver needs to track operation errors. The following NX_IP_DRIVER members are used for the link error count request:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td>NX_LINK_GET_ERROR_COUNT</td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td>nx_ip_driver_return_ptr</td>
<td>Pointer to the destination to place the error count</td>
</tr>
<tr>
<td>nx_ip_driver_interface</td>
<td>Pointer to the interface instance</td>
</tr>
<tr>
<td>nx_ip_driver_status</td>
<td>Completion status. If the driver is not able to get error count, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>

This request is not used internally by NetX 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. To support this feature, the driver needs to keep track of the number of packets received. The following NX_IP_DRIVER members are used for the link receive packet count request:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td>NX_LINK_GET_RX_COUNT</td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td>nx_ip_driver_return_ptr</td>
<td>Pointer to the destination to place the receive packet count</td>
</tr>
<tr>
<td>nx_ip_driver_interface</td>
<td>Pointer to the physical network interface</td>
</tr>
<tr>
<td>nx_ip_driver_status</td>
<td>Completion status. If the driver is not able to get receive count, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>
This request is not used internally by NetX 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. To support this feature, the driver needs to keep track of each packet it transmits on each interface. The following `NX_IP_DRIVER` members are used for the link transmit packet count request:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx_ip_driver_command</code></td>
<td><code>NX_LINK_GET_TX_COUNT</code></td>
</tr>
<tr>
<td><code>nx_ip_driver_ptr</code></td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td><code>nx_ip_driver_return_ptr</code></td>
<td>Pointer to the destination to place the transmit packet count</td>
</tr>
<tr>
<td><code>nx_ip_driver_interface</code></td>
<td>Pointer to the interface instance</td>
</tr>
<tr>
<td><code>nx_ip_driver_status</code></td>
<td>Completion status. If the driver is not able to get transmit count, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>

Get Allocation Errors

This request is made from within the `nx_ip_driver_direct_command` service. The driver stores the link’s packet pool allocation error count in the supplied destination. The following `NX_IP_DRIVER` members are used for the link allocation error count request:

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx_ip_driver_command</code></td>
<td><code>NX_LINK_GET_ALLOC_ERRORS</code></td>
</tr>
<tr>
<td><code>nx_ip_driver_ptr</code></td>
<td>Pointer to IP instance</td>
</tr>
</tbody>
</table>

This request is not used internally by NetX 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 process 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.

<table>
<thead>
<tr>
<th><code>NX_IP_DRIVER</code> member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx_ip_driver_command</code></td>
<td>NX_LINK_DEFERRED_PROCESSING</td>
</tr>
<tr>
<td><code>nx_ip_driver_ptr</code></td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td><code>nx_ip_driver_interface</code></td>
<td>Pointer to the interface instance</td>
</tr>
</tbody>
</table>

### User Commands

This request is made from within the `nx_ip_driver_direct_command` service. The driver processes the application specific user commands.

<table>
<thead>
<tr>
<th><code>NX_IP_DRIVER</code> member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx_ip_driver_return_ptr</code></td>
<td>Pointer to the destination to place the allocation error count</td>
</tr>
<tr>
<td><code>nx_ip_driver_interface</code></td>
<td>Pointer to the interface instance</td>
</tr>
<tr>
<td><code>nx_ip_driver_status</code></td>
<td>Completion status. If the driver is not able to get allocation errors, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>
The following NX_IP_DRIVER members are used for the user command request.

<table>
<thead>
<tr>
<th>NX_IP_DRIVER member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_command</td>
<td>NX_LINK_USER_COMMAND</td>
</tr>
<tr>
<td>nx_ip_driver_ptr</td>
<td>Pointer to IP instance</td>
</tr>
<tr>
<td>nx_ip_driver_return_ptr</td>
<td>User defined</td>
</tr>
<tr>
<td>nx_ip_driver_interface</td>
<td>Pointer to the interface instance</td>
</tr>
<tr>
<td>nx_ip_driver_status</td>
<td>Completion status. If the driver is not able to execute user commands, it will return a non-zero error status.</td>
</tr>
</tbody>
</table>

*This request is not used internally by NetX 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_prepending_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 packet. Building a
valid NetX packet involves setting up the appropriate length field and chaining together multiple packets if the incoming packet’s size is greater than a single packet payload. Once properly built, the prepend_ptr is moved after the physical layer header and the receive packet is dispatched to NetX.

NetX assumes that the IP and ARP headers are aligned on a ULONG boundary. The NetX 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 \texttt{nx\_packet\_prepend\_ptr} is moved beyond the Ethernet header, the underlying IP or ARP header is 4-byte aligned.

There are several receive packet functions available in NetX. If the received packet is an ARP packet, \texttt{\_nx\_arp\_packet\_deferred\_receive} is called. If the received packet is an RARP packet, \texttt{\_nx\_rarp\_packet\_deferred\_receive} is called. There are several options for handling incoming IP packets. For the fastest handling of IP packets, \texttt{\_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 \texttt{\_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 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.

\textit{The initial receive buffers are setup during driver initialization.}
Deferred Receive Packet Handling

The driver may defer receive packet processing to the NetX 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 library must first be compiled with `NX_DRIVER_DEFERRED_PROCESSING` defined. This adds the deferred packet logic to the NetX 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 receive functions:

```
_nx_ip_packet_deferred_receive
_nx_arp_packet_deferred_receive
_nx_rarp_packet_deferred_receive
```

Example RAM Ethernet Network Driver

The NetX demonstration system is delivered with a small RAM-based 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 (MAC addresses) to each device instance as they are created. This file provides a good example of the basic structure of NetX physical network drivers. Users may develop their own network drivers using the driver framework presented in this example.
The entry function of the network driver is \_nx\_ram\_network\_driver(), which is passed to the IP instance create call. Entry functions for additional network interfaces can be passed into the \nx\_ip\_interface\_attach() service. After the IP instance starts to run, the driver entry function is invoked to initialize and enable the device (refer to the case \nx\_link\_initialize and \nx\_link\_enable). After the \nx\_link\_enable command is issued, the device should be ready to transmit and receive packets.

The IP instance transmits network packets via one of these commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\nx_link_packet_send</td>
<td>An IP packet is being transmitted,</td>
</tr>
<tr>
<td>\nx_link_arp_send</td>
<td>An ARP request or ARP response packet is being transmitted,</td>
</tr>
<tr>
<td>\nx_link_arp_rarp_send</td>
<td>A Reverse ARP request or response packet is being transmitted,</td>
</tr>
</tbody>
</table>

On processing these commands, the network driver needs to prepend the appropriate Ethernet frame header, and then send it to the underlying hardware for transmission. During the transmission process, the network driver has the exclusive ownership of the packet buffer area. Therefore once the data are being transmitted (or once the data has been copied into the driver internal transfer buffer), the network driver is responsible for releasing the packet buffer by first moving the prepend pointer past the Ethernet header to the IP header (and adjust packet length accordingly), and then by calling the \nx\_packet\_transmit\_release() service to release the packet. Not releasing the packet after data transmission will cause packets to leak.

The network device driver is also responsible for handling incoming data packets. In the RAM driver example, the received packet is processed by the
function \_nx\_ram\_network\_driver\_receive()\.
Once the device receives an Ethernet frame, the
driver is responsible for storing the data in
NX\_PACKET structure. Note that NetX assumes the
IP header starts from 4-byte aligned address. Since
the length of Ethernet header is 14-byte, the driver
needs to store the starting of the Ethernet header at
2-byte aligned address to guarantee that the IP
header starts at 4-byte aligned address.
NetX Services

- Address Resolution Protocol (ARP) 378
- Internet Control Message Protocol (ICMP) 378
- Internet Group Management Protocol (IGMP) 379
- Internet Protocol (IP) 379
- Packet Management 381
- Reverse Address Resolution Protocol (RARP) 381
- System Management 382
- Transmission Control Protocol (TCP) 382
- User Datagram Protocol (UDP) 384
### Address Resolution Protocol (ARP)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx_arp_dynamic_entries_invalidate</code></td>
<td>Invalidates dynamic ARP entries</td>
</tr>
<tr>
<td><code>nx_arp_dynamic_entry_set</code></td>
<td>Sets a dynamic ARP entry</td>
</tr>
<tr>
<td><code>nx_arp_enable</code></td>
<td>Enables ARP on the specified IP address</td>
</tr>
<tr>
<td><code>nx_arp GRATUITOUS.Send</code></td>
<td>Sends gratuitous ARP requests</td>
</tr>
<tr>
<td><code>nx_arp_hardware_address_find</code></td>
<td>Finds the hardware address of the specified IP address</td>
</tr>
<tr>
<td><code>nx_arp_info_get</code></td>
<td>Gets ARP information</td>
</tr>
<tr>
<td><code>nx_arp_ip_address_find</code></td>
<td>Finds the IP address associated with the specified hardware address</td>
</tr>
<tr>
<td><code>nx_arp_static_entries_delete</code></td>
<td>Deletes a static ARP entry</td>
</tr>
<tr>
<td><code>nx_arp_static_entry_create</code></td>
<td>Creates a static ARP entry</td>
</tr>
<tr>
<td><code>nx_arp_static_entry_delete</code></td>
<td>Deletes a static ARP entry</td>
</tr>
</tbody>
</table>

### Internet Control Message Protocol (ICMP)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx_icmp_enable</code></td>
<td>Enables ICMP on the specified IP address</td>
</tr>
<tr>
<td><code>nx_icmp_info_get</code></td>
<td>Gets ICMP information</td>
</tr>
<tr>
<td><code>nx_icmp_ping</code></td>
<td>Sends an ICMP echo request response packet</td>
</tr>
</tbody>
</table>

The functions provided in the NetX User Guide for ARP and ICMP operations are designed to handle various aspects of IP address resolution and network control, including setting up, enabling, and managing ARP and ICMP entries.
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_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 *physical_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_link_status_change_notify_set(NX_IP *ip_ptr,
VOID (*link_status_change_notify)(NX_IP *ip_ptr,
UINT interface_index, UINT link_up));

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);

Packet Management

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

UINT  nx_packetCopy(NX_PACKET *packet_ptr, 
    NX_PACKET **new_packet_ptr, NX_PACKET_POOL 
    *pool_ptr, 
    ULONG wait_option);

UINT  nx_packetDataAppend(NX_PACKET *packet_ptr, 
    VOID *data_start, ULONG data_size, 
    NX_PACKET_POOL *pool_ptr, ULONG wait_option);

UINT  nx_packetDataExtractOffset(NX_PACKET *packet_ptr, 
    ULONG offset, VOID *buffer_start, ULONG 
    buffer_length, ULONG *bytes_copied);

UINT  nx_packetDataRetrieve(NX_PACKET *packet_ptr, 
    VOID *buffer_start, ULONG *bytes_copied);

UINT  nx_packetLengthGet(NX_PACKET *packet_ptr, ULONG 
    *length);

UINT  nx_packetPoolCreate(NX_PACKET_POOL *pool_ptr, 
    CHAR *name, ULONG block_size, VOID *memory_ptr, 
    ULONG memory_size);

UINT  nx_packetPoolDelete(NX_PACKET_POOL *pool_ptr);

UINT  nx_packetPoolInfoGet(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_packetRelease(NX_PACKET *packet_ptr);

UINT  nx_packetTransmitRelease(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

Transmission Control Protocol (TCP)

VOID nx_system_initialize(VOID);

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, UINT port));
\*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_establish_notify(NX_TCP_SOCKET
*socket_ptr, VOID
(*tcp_establish_notify)(NX_TCP_SOCKET
*socket_ptr));

UINT nx_tcp_socket_disconnect_complete_notify(NX_TCP_SOCKET
*socket_ptr, VOID
(*tcp_disconnect_complete_notify)(NX_TCP_SOCKET
*socket_ptr));

UINT nx_tcp_socket_timed_wait_callback(NX_TCP_SOCKET
*socket_ptr, VOID
(*tcp_timed_wait_callback)(NX_TCP_SOCKET
*socket_ptr));

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);
User Datagram Protocol (UDP)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx_tcp_socket_transmit_configure</code></td>
<td>Configures the transmit function</td>
</tr>
<tr>
<td><code>nx_tcp_socket_window_update_notify_set</code></td>
<td>Sets the window update function</td>
</tr>
<tr>
<td><code>nx_udp_enable</code></td>
<td>Enables the UDP protocol</td>
</tr>
<tr>
<td><code>nx_udp_free_port_find</code></td>
<td>Finds a free UDP port</td>
</tr>
<tr>
<td><code>nx_udp_info_get</code></td>
<td>Returns information about the UDP packet</td>
</tr>
<tr>
<td><code>nx_udp_packet_info_extract</code></td>
<td>Extracts information from the UDP packet</td>
</tr>
<tr>
<td><code>nx_udp_socket_bind</code></td>
<td>Binds the UDP socket to a specific port</td>
</tr>
<tr>
<td><code>nx_udp_socket_bytes_available</code></td>
<td>Gets the number of bytes available on the socket</td>
</tr>
<tr>
<td><code>nx_udp_socket_checksum_disable</code></td>
<td>Disables the checksum in the UDP packet</td>
</tr>
<tr>
<td><code>nx_udp_socket_checksum_enable</code></td>
<td>Enables the checksum in the UDP packet</td>
</tr>
<tr>
<td><code>nx_udp_socket_create</code></td>
<td>Creates a new UDP socket</td>
</tr>
<tr>
<td><code>nx_udp_socket_delete</code></td>
<td>Deletes the UDP socket</td>
</tr>
<tr>
<td><code>nx_udp_socket_info_get</code></td>
<td>Gets information about the UDP socket</td>
</tr>
<tr>
<td><code>nx_udp_socket_interface_send</code></td>
<td>Sends a UDP packet over an interface</td>
</tr>
<tr>
<td><code>nx_udp_socket_window_update_notify_set</code></td>
<td>Sets the window update function</td>
</tr>
<tr>
<td><code>nx_udp_socket_transmit_configure</code></td>
<td>Configures the transmit function</td>
</tr>
</tbody>
</table>
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);
NetX Constants

- Alphabetic Listing 388
- Listings by Value 397
# Alphabetic Listing

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_ALL_HOSTS_ADDRESS</td>
<td>0xFE000001</td>
</tr>
<tr>
<td>NX_ALL_ROUTERS_ADDRESS</td>
<td>0xFE000002</td>
</tr>
<tr>
<td>NX_ALREADY_BOUND</td>
<td>0x22</td>
</tr>
<tr>
<td>NX_ALREADY_ENABLED</td>
<td>0x15</td>
</tr>
<tr>
<td>NX_ALREADY_RELEASED</td>
<td>0x31</td>
</tr>
<tr>
<td>NX_ALREADY_SUSPENDED</td>
<td>0x40</td>
</tr>
<tr>
<td>NX_ANY_PORT</td>
<td>0</td>
</tr>
<tr>
<td>NX_ARP_EXPIRATION_RATE</td>
<td>0</td>
</tr>
<tr>
<td>NX_ARP_HARDWARE_SIZE</td>
<td>0x06</td>
</tr>
<tr>
<td>NX_ARP_HARDWARE_TYPE</td>
<td>0x0001</td>
</tr>
<tr>
<td>NX_ARP_MAX_QUEUE_DEPTH</td>
<td>4</td>
</tr>
<tr>
<td>NX_ARP_MAXIMUM_RETRIES</td>
<td>18</td>
</tr>
<tr>
<td>NX_ARP_MESSAGE_SIZE</td>
<td>28</td>
</tr>
<tr>
<td>NX_ARP_OPTION_REQUEST</td>
<td>0x0001</td>
</tr>
<tr>
<td>NX_ARP_OPTION_RESPONSE</td>
<td>0x0002</td>
</tr>
<tr>
<td>NX_ARP_PROTOCOL_SIZE</td>
<td>0x04</td>
</tr>
<tr>
<td>NX_ARP_PROTOCOL_TYPE</td>
<td>0x0800</td>
</tr>
<tr>
<td>NX_ARP_TIMER_ERROR</td>
<td>0x18</td>
</tr>
<tr>
<td>NX_ARP_UPDATE_RATE</td>
<td>0x10</td>
</tr>
<tr>
<td>NX_ARP_TABLE_SIZE</td>
<td>0x2F</td>
</tr>
<tr>
<td>NX_ARP_TABLE_MASK</td>
<td>0x1F</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>0x11</td>
</tr>
<tr>
<td>NX_CARRY_BIT</td>
<td>0x10000</td>
</tr>
<tr>
<td>NX_CONNECTION_PENDING</td>
<td>0x48</td>
</tr>
<tr>
<td>NX_DELETE_ERROR</td>
<td>0x10</td>
</tr>
<tr>
<td>NX_DELETED</td>
<td>0x05</td>
</tr>
<tr>
<td>NX_DISCONNECT_FAILED</td>
<td>0x41</td>
</tr>
<tr>
<td>NX_DONT_FRAGMENT</td>
<td>0x00004000</td>
</tr>
<tr>
<td>NX_DRIVER_TX_DONE</td>
<td>0xDDDDDDDD</td>
</tr>
<tr>
<td>NX_DUPLICATE_LISTEN</td>
<td>0x34</td>
</tr>
<tr>
<td>NX_ENTRY_NOT_FOUND</td>
<td>0x16</td>
</tr>
<tr>
<td>NX_FALSE</td>
<td>0</td>
</tr>
<tr>
<td>NX_FOREVER</td>
<td>1</td>
</tr>
</tbody>
</table>
### NetX Constants

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_FRAG_OFFSET_MASK</td>
<td>0x00001FFF</td>
</tr>
<tr>
<td>NX_FRAGMENT_OKAY</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NX_ICMP_ADDRESS_MASK_REP_TYPE</td>
<td>18</td>
</tr>
<tr>
<td>NX_ICMP_ADDRESS_MASK_REQ_TYPE</td>
<td>17</td>
</tr>
<tr>
<td>NX_ICMP_DEST_UNREACHABLE_TYPE</td>
<td>3</td>
</tr>
<tr>
<td>NX_ICMP_ECHO_REPLY_TYPE</td>
<td>0</td>
</tr>
<tr>
<td>NX_ICMP_ECHO_REQUEST_TYPE</td>
<td>8</td>
</tr>
<tr>
<td>NX_ICMP_FRAMENT_NEEDED_CODE</td>
<td>4</td>
</tr>
<tr>
<td>NX_ICMP_HOST_PROHIBIT_CODE</td>
<td>10</td>
</tr>
<tr>
<td>NX_ICMP_HOST_SERVICE_CODE</td>
<td>12</td>
</tr>
<tr>
<td>NX_ICMP_HOSTUNKNOWN_CODE</td>
<td>7</td>
</tr>
<tr>
<td>NX_ICMP_HOST_UNREACH_CODE</td>
<td>1</td>
</tr>
<tr>
<td>NX_ICMP_NETWORK_PROHIBIT_CODE</td>
<td>9</td>
</tr>
<tr>
<td>NX_ICMP_NETWORK_SERVICE_CODE</td>
<td>11</td>
</tr>
<tr>
<td>NX_ICMP_NETWORKUNKNOWN_CODE</td>
<td>6</td>
</tr>
<tr>
<td>NX_ICMP_NETWORKUNREACH_CODE</td>
<td>0</td>
</tr>
<tr>
<td>NX_ICMP_PACKET</td>
<td>36</td>
</tr>
<tr>
<td>NX_ICMP_PARAMETERPROBTYPE</td>
<td>12</td>
</tr>
<tr>
<td>NX_ICMP_PORTUNREACH_CODE</td>
<td>3</td>
</tr>
<tr>
<td>NX_ICMP_PROTOCOLUNREACH_CODE</td>
<td>2</td>
</tr>
<tr>
<td>NX_ICMP_REDIRECT_TYPE</td>
<td>5</td>
</tr>
<tr>
<td>NX_ICMP_SOURCEISOLATEDCODE</td>
<td>8</td>
</tr>
<tr>
<td>NX_ICMP_SOURCEQUEENCH_TYPE</td>
<td>4</td>
</tr>
<tr>
<td>NX_ICMP_SOURCEROUTECODE</td>
<td>5</td>
</tr>
<tr>
<td>NX_ICMP_TIMEEXCEEDED_TYPE</td>
<td>11</td>
</tr>
<tr>
<td>NX_ICMP_TIMESTAMPREP_TYPE</td>
<td>14</td>
</tr>
<tr>
<td>NX_ICMP_TIMESTAMPREQ_TYPE</td>
<td>13</td>
</tr>
<tr>
<td>NX_IGMP_HEADER_SIZE</td>
<td>8</td>
</tr>
<tr>
<td>NX_IGMP_HOSTRESPONSETYPE</td>
<td>0x02000000</td>
</tr>
<tr>
<td>NX_IGMP_HOSTV2JOIN_TYPE</td>
<td>0x16000000</td>
</tr>
<tr>
<td>NX_IGMP_HOSTV2LEAVE_TYPE</td>
<td>0x17000000</td>
</tr>
<tr>
<td>NX_IGMP_HOSTVERSION_1</td>
<td>1</td>
</tr>
<tr>
<td>NX_IGMP_HOSTVERSION_2</td>
<td>2</td>
</tr>
<tr>
<td>NX_IGMP_MAXRESPTIME_MASK</td>
<td>0x00FF0000</td>
</tr>
<tr>
<td>NX_IGMP_MAXUPDATE_TIME</td>
<td>10</td>
</tr>
</tbody>
</table>
### NetX User Guide

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_IGMP_PACKET</td>
<td>36</td>
</tr>
<tr>
<td>NX_IGMP_ROUTER_QUERY_TYPE</td>
<td>0x01000000</td>
</tr>
<tr>
<td>NX_IGMP_TTL</td>
<td>1</td>
</tr>
<tr>
<td>NX_IGMP_TYPE_MASK</td>
<td>0x0F000000</td>
</tr>
<tr>
<td>NX_IGMP_VERSION</td>
<td>0x10000000</td>
</tr>
<tr>
<td>NX_IGMPV2_TYPE_MASK</td>
<td>0xFF000000</td>
</tr>
<tr>
<td>NX_IN_PROGRESS</td>
<td>0x37</td>
</tr>
<tr>
<td>NX_INIT_PACKET_ID</td>
<td>1</td>
</tr>
<tr>
<td>NX_NOT_IMPLEMENTED</td>
<td>0x4A</td>
</tr>
<tr>
<td>NX_NOT_SUPPORTED</td>
<td>0x4B</td>
</tr>
<tr>
<td>NX_INVALID_INTERFACE</td>
<td>0x4C</td>
</tr>
<tr>
<td>NX_INVALID_PACKET</td>
<td>0x12</td>
</tr>
<tr>
<td>NX_INVALID_PORT</td>
<td>0x46</td>
</tr>
<tr>
<td>NX_INVALID_RELISTEN</td>
<td>0x47</td>
</tr>
<tr>
<td>NX_INVALID_SOCKET</td>
<td>0x13</td>
</tr>
<tr>
<td>NX_IP_ADDRESS_ERROR</td>
<td>0x21</td>
</tr>
<tr>
<td>NX_IP_ADDRESS_RESOLVED</td>
<td>0x0002</td>
</tr>
<tr>
<td>NX_IP_ALIGN_FRAGS</td>
<td>8</td>
</tr>
<tr>
<td>NX_IP_ALL_EVENTS</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>NX_IP_ARP_ENABLED</td>
<td>0x0008</td>
</tr>
<tr>
<td>NX_IP_ARP_REC_EVENT</td>
<td>0x00000010</td>
</tr>
<tr>
<td>NX_IP_CLASS_A_HOSTID</td>
<td>0x00FFFFFF</td>
</tr>
<tr>
<td>NX_IP_CLASS_A_MASK</td>
<td>0x80000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_A_NETID</td>
<td>0x7F000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_A_TYPE</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_B_HOSTID</td>
<td>0x0000FFFF</td>
</tr>
<tr>
<td>NX_IP_CLASS_B_MASK</td>
<td>0xC0000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_B_NETID</td>
<td>0xF0000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_B_TYPE</td>
<td>0x80000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_C_HOSTID</td>
<td>0x000000FF</td>
</tr>
<tr>
<td>NX_IP_CLASS_C_MASK</td>
<td>0xE0000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_C_NETID</td>
<td>0x1FFFFFFF</td>
</tr>
<tr>
<td>NX_IP_CLASS_C_TYPE</td>
<td>0xC0000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_D_GROUP</td>
<td>0x0FFFFFFFF</td>
</tr>
<tr>
<td>NX_IP_CLASS_D_HOSTID</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_D_MASK</td>
<td>0xF0000000</td>
</tr>
<tr>
<td>NetX Constants</td>
<td>Value</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td>NX_IP_CLASS_D_TYPE</td>
<td>0xE0000000</td>
</tr>
<tr>
<td>NX_IP_DEBUG_LOG_SIZE</td>
<td>100</td>
</tr>
<tr>
<td>NX_IP_DONT_FRAGMENT</td>
<td>0x00004000</td>
</tr>
<tr>
<td>NX_IP_DRIVER_DEFERRED_EVENT</td>
<td>0x00000800</td>
</tr>
<tr>
<td>NX_IP_DRIVER_PACKET_EVENT</td>
<td>0x00000200</td>
</tr>
<tr>
<td>NX_IP_FRAGMENT_MASK</td>
<td>0x00003FFF</td>
</tr>
<tr>
<td>NX_IP_ICMP</td>
<td>0x00010000</td>
</tr>
<tr>
<td>NX_IP_ICMP_EVENT</td>
<td>0x00000004</td>
</tr>
<tr>
<td>NX_IP_ID</td>
<td>0x49502020</td>
</tr>
<tr>
<td>NX_IP_IPM</td>
<td>0x00020000</td>
</tr>
<tr>
<td>NX_IP_ICMP_ENABLE_EVENT</td>
<td>0x00000400</td>
</tr>
<tr>
<td>NX_IP_ICMP_ENABLED</td>
<td>0x0040</td>
</tr>
<tr>
<td>NX_IP_ICMP_EVENT</td>
<td>0x00000040</td>
</tr>
<tr>
<td>NX_IP_INITIALIZE_DONE</td>
<td>0x0001</td>
</tr>
<tr>
<td>NX_IP_INTERNAL_ERROR</td>
<td>0x20</td>
</tr>
<tr>
<td>NX_IP_LENGTH_MASK</td>
<td>0xF0000000</td>
</tr>
<tr>
<td>NX_IP_LIMITIED_BROADCAST</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>NX_IP_LINK_ENABLED</td>
<td>0x0004</td>
</tr>
<tr>
<td>NX_IP_LOOPBACK_FIRST</td>
<td>0x7F000000</td>
</tr>
<tr>
<td>NX_IP_LOOPBACK_LAST</td>
<td>0x7FFFFFFF</td>
</tr>
<tr>
<td>NX_IP_MAX_DATA</td>
<td>0x00080000</td>
</tr>
<tr>
<td>NX_IP_MAX_RELIABLE</td>
<td>0x00040000</td>
</tr>
<tr>
<td>NX_IP_MIN_COST</td>
<td>0x00020000</td>
</tr>
<tr>
<td>NX_IP_MIN_DELAY</td>
<td>0x00100000</td>
</tr>
<tr>
<td>NX_IP_MORE_FRAGMENT</td>
<td>0x00002000</td>
</tr>
<tr>
<td>NX_IP_MULTICAST_LOWER</td>
<td>0x05E000000</td>
</tr>
<tr>
<td>NX_IP_MULTICAST_MASK</td>
<td>0x007FFFFFFF</td>
</tr>
<tr>
<td>NX_IP_MULTICAST_UPPER</td>
<td>0x00000100</td>
</tr>
<tr>
<td>NX_IP_NORMAL</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NX_IP_NORMAL_LENGTH</td>
<td>5</td>
</tr>
<tr>
<td>NX_IP_OFFSET_MASK</td>
<td>0x00001FFF</td>
</tr>
<tr>
<td>NX_IP_PACKET</td>
<td>36</td>
</tr>
<tr>
<td>NX_IP_PACKET_SIZE_MASK</td>
<td>0x0000FFFF</td>
</tr>
<tr>
<td>NX_IP_PERIODIC_EVENT</td>
<td>0x00000001</td>
</tr>
<tr>
<td>NX_IP_PERIODIC_RATE</td>
<td>100</td>
</tr>
<tr>
<td>Variable</td>
<td>Value</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>NX_IP_PROTOCOL_MASK</td>
<td>0x00FF0000</td>
</tr>
<tr>
<td>NX_IP_RARP_COMPLETE</td>
<td>0x0080</td>
</tr>
<tr>
<td>NX_IP_RARP_REC_EVENT</td>
<td>0x00000020</td>
</tr>
<tr>
<td>NX_IP_RECEIVE_EVENT</td>
<td>0x00000008</td>
</tr>
<tr>
<td>NX_IP_TCP</td>
<td>0x00060000</td>
</tr>
<tr>
<td>NX_IP_TCP_CLEANUP_DEFERRED</td>
<td>0x00010000</td>
</tr>
<tr>
<td>NX_IP_TCP_ENABLED</td>
<td>0x0020</td>
</tr>
<tr>
<td>NX_IP_TCP_EVENT</td>
<td>0x00000080</td>
</tr>
<tr>
<td>NX_IP_TCP_FAST_EVENT</td>
<td>0x00000100</td>
</tr>
<tr>
<td>NX_IP_TIME_TO_LIVE</td>
<td>0x00000080</td>
</tr>
<tr>
<td>NX_IP_TIME_TO_LIVE_MASK</td>
<td>0xFF000000</td>
</tr>
<tr>
<td>NX_IP_TIME_TO_LIVE_SHIFT</td>
<td>24</td>
</tr>
<tr>
<td>NX_IP_TOS_MASK</td>
<td>0x00FF0000</td>
</tr>
<tr>
<td>NX_IP_UDP</td>
<td>0x00110000</td>
</tr>
<tr>
<td>NX_IP_UDP_ENABLED</td>
<td>0x0010</td>
</tr>
<tr>
<td>NX_IP_UNFRAG_EVENT</td>
<td>0x00000002</td>
</tr>
<tr>
<td>NX_IP_VERSION</td>
<td>0x4500000000x80</td>
</tr>
<tr>
<td>NX_LINK_ARP_RESPONSE_SEND</td>
<td>6</td>
</tr>
<tr>
<td>NX_LINK_ARP_SEND</td>
<td>5</td>
</tr>
<tr>
<td>NX_LINK_DEFERRED_PROCESSING</td>
<td>18</td>
</tr>
<tr>
<td>NX_LINK_DISABLE</td>
<td>3</td>
</tr>
<tr>
<td>NX_LINK_ENABLE</td>
<td>2</td>
</tr>
<tr>
<td>NX_LINK_GET_ALLOC_ERRORS</td>
<td>16</td>
</tr>
<tr>
<td>NX_LINK_GET_DUPLEX_TYPE</td>
<td>12</td>
</tr>
<tr>
<td>NX_LINK_GET_ERROR_COUNT</td>
<td>13</td>
</tr>
<tr>
<td>NX_LINK_GET_RX_COUNT</td>
<td>14</td>
</tr>
<tr>
<td>NX_LINK_GET_SPEED</td>
<td>11</td>
</tr>
<tr>
<td>NX_LINK_GET_STATUS</td>
<td>10</td>
</tr>
<tr>
<td>NX_LINK_GET_TX_COUNT</td>
<td>15</td>
</tr>
<tr>
<td>NX_LINK_INITIALIZE</td>
<td>1</td>
</tr>
<tr>
<td>NX_LINK_INTERFACE_ATTACH</td>
<td>19</td>
</tr>
<tr>
<td>NX_LINK_MULTICAST_JOIN</td>
<td>8</td>
</tr>
<tr>
<td>NX_LINK_MULTICAST_LEAVE</td>
<td>9</td>
</tr>
<tr>
<td>NX_LINK_PACKET_BROADCAST</td>
<td>4</td>
</tr>
<tr>
<td>NX_LINK_PACKET_SEND</td>
<td>0</td>
</tr>
<tr>
<td>Constant Name</td>
<td>Value</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
</tr>
<tr>
<td>NX_LINK_RARP_SEND</td>
<td>7</td>
</tr>
<tr>
<td>NX_LINK_UNINITIALIZE</td>
<td>17</td>
</tr>
<tr>
<td>NX_LINK_USER_COMMAND</td>
<td>50</td>
</tr>
<tr>
<td>NX_LOWER_16_MASK</td>
<td>0x0000FFFF</td>
</tr>
<tr>
<td>NX_MAX_LISTEN</td>
<td>0x33</td>
</tr>
<tr>
<td>NX_MAX_LISTEN_REQUESTS</td>
<td>10</td>
</tr>
<tr>
<td>NX_MAX_MULTICAST_GROUPS</td>
<td>7</td>
</tr>
<tr>
<td>NX_MAX_PORT</td>
<td>0xFFFF</td>
</tr>
<tr>
<td>NX_MORE_FRAGMENTS</td>
<td>0x00002000</td>
</tr>
<tr>
<td>NX_NO_FREE_PORTS</td>
<td>0x45</td>
</tr>
<tr>
<td>NX_NO_MAPPING</td>
<td>0x04</td>
</tr>
<tr>
<td>NX_NO_MORE_ENTRIES</td>
<td>0x17</td>
</tr>
<tr>
<td>NX_NO_PACKET</td>
<td>0x01</td>
</tr>
<tr>
<td>NX_NO_RESPONSE</td>
<td>0x29</td>
</tr>
<tr>
<td>NX_NO_WAIT</td>
<td>0</td>
</tr>
<tr>
<td>NX_NOT_BOUND</td>
<td>0x24</td>
</tr>
<tr>
<td>NX_NOT_CLOSED</td>
<td>0x35</td>
</tr>
<tr>
<td>NX_NOT_CONNECTED</td>
<td>0x38</td>
</tr>
<tr>
<td>NX_NOT_CREATED</td>
<td>0x27</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>0x14</td>
</tr>
<tr>
<td>NX_NOT_IMPLEMENTED</td>
<td>0x4A</td>
</tr>
<tr>
<td>NX_NOT_LISTEN_STATE</td>
<td>0x36</td>
</tr>
<tr>
<td>NX_NOT_SUCCESSFUL</td>
<td>0x43</td>
</tr>
<tr>
<td>NX_NULL</td>
<td>0</td>
</tr>
<tr>
<td>NX_OPTION_ERROR</td>
<td>0x0a</td>
</tr>
<tr>
<td>NX_OVERFLOW</td>
<td>0x03</td>
</tr>
<tr>
<td>NX_PACKET_ALLOCATED</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>NX_PACKET_DEBUG_LOG_SIZE</td>
<td>100</td>
</tr>
<tr>
<td>NX_PACKET_ENQUEUED</td>
<td>0xEEEEEEEE</td>
</tr>
<tr>
<td>NX_PACKET_FREE</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>NX_PACKET_POOL_ID</td>
<td>0x5041434B</td>
</tr>
<tr>
<td>NX_PACKET_READY</td>
<td>0xBBB BBBBB</td>
</tr>
<tr>
<td>NX_PHYSICAL_HEADER</td>
<td>16</td>
</tr>
<tr>
<td>NX_PHYSICAL_TRAILER</td>
<td>4</td>
</tr>
<tr>
<td>NX_POOL_DELETED</td>
<td>0x30</td>
</tr>
<tr>
<td>Symbol</td>
<td>Value</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>NX_POOL_ERROR</td>
<td>0x06</td>
</tr>
<tr>
<td>NX_PORT_UNAVAILABLE</td>
<td>0x23</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>0x07</td>
</tr>
<tr>
<td>NX_RARP_HARDWARE_SIZE</td>
<td>0x06</td>
</tr>
<tr>
<td>NX_RARP_HARDWARE_TYPE</td>
<td>0x0001</td>
</tr>
<tr>
<td>NX_RARP_MESSAGE_SIZE</td>
<td>28</td>
</tr>
<tr>
<td>NX_RARP_OPTION_REQUEST</td>
<td>0x0003</td>
</tr>
<tr>
<td>NX_RARP_OPTION_RESPONSE</td>
<td>0x0004</td>
</tr>
<tr>
<td>NX_RARP_PROTOCOL_SIZE</td>
<td>0x04</td>
</tr>
<tr>
<td>NX_RARP_PROTOCOL_TYPE</td>
<td>0x0800</td>
</tr>
<tr>
<td>NX_RECEIVE_PACKET</td>
<td>0</td>
</tr>
<tr>
<td>NX_RESERVED_CODE0</td>
<td>0x19</td>
</tr>
<tr>
<td>NX_RESERVED_CODE1</td>
<td>0x25</td>
</tr>
<tr>
<td>NX_RESERVED_CODE2</td>
<td>0x32</td>
</tr>
<tr>
<td>NX_ROUTE_TABLE_MASK</td>
<td>0x1F</td>
</tr>
<tr>
<td>NX_ROUTE_TABLE_SIZE</td>
<td>32</td>
</tr>
<tr>
<td>NX_SEARCH_PORT_START</td>
<td>30000</td>
</tr>
<tr>
<td>NX_SHIFT_BY_16</td>
<td>16</td>
</tr>
<tr>
<td>NX_SIZE_ERROR</td>
<td>0x09</td>
</tr>
<tr>
<td>NX_SOCKET_UNBOUND</td>
<td>0x26</td>
</tr>
<tr>
<td>NX_SOCKETS_BOUND</td>
<td>0x28</td>
</tr>
<tr>
<td>NX_STILL_BOUND</td>
<td>0x42</td>
</tr>
<tr>
<td>NX_SUCCESS</td>
<td>0x00</td>
</tr>
<tr>
<td>NX_TCP_ACK_BIT</td>
<td>0x00100000</td>
</tr>
<tr>
<td>NX_TCP_ACK_TIMER_RATE</td>
<td>5</td>
</tr>
<tr>
<td>NX_TCP_CLIENT</td>
<td>1</td>
</tr>
<tr>
<td>NX_TCP_CLOSE_WAIT</td>
<td>6</td>
</tr>
<tr>
<td>NX_TCP_CLOSED</td>
<td>1</td>
</tr>
<tr>
<td>NX_TCP_CLOSING</td>
<td>9</td>
</tr>
<tr>
<td>NX_TCP_CONTROL_MASK</td>
<td>0x00170000</td>
</tr>
<tr>
<td>NX_TCP_EOL_KIND</td>
<td>0x00</td>
</tr>
<tr>
<td>NX_TCP_ESTABLISHED</td>
<td>5</td>
</tr>
<tr>
<td>NX_TCP_FAST_TIMER_RATE</td>
<td>10</td>
</tr>
<tr>
<td>NX_TCP_FIN_BIT</td>
<td>0x00010000</td>
</tr>
<tr>
<td>NX_TCP_FIN_WAIT_1</td>
<td>7</td>
</tr>
<tr>
<td>Constant</td>
<td>Value</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>NX_TCP_FIN_WAIT_2</td>
<td>8</td>
</tr>
<tr>
<td>NX_TCP_HEADER_MASK</td>
<td>0xF0000000</td>
</tr>
<tr>
<td>NX_TCP_HEADER_SHIFT</td>
<td>28</td>
</tr>
<tr>
<td>NX_TCP_HEADER_SIZE</td>
<td>0x50000000</td>
</tr>
<tr>
<td>NX_TCP_ID</td>
<td>0x54435020</td>
</tr>
<tr>
<td>NX_TCP_KEEPALIVE_INITIAL</td>
<td>7200</td>
</tr>
<tr>
<td>NX_TCP_KEEPALIVE_RETRIES</td>
<td>10</td>
</tr>
<tr>
<td>NX_TCP_KEEPALIVE_RETRY</td>
<td>75</td>
</tr>
<tr>
<td>NX_TCP_LAST_ACK</td>
<td>11</td>
</tr>
<tr>
<td>NX_TCP_LISTEN_STATE</td>
<td>2</td>
</tr>
<tr>
<td>NX_TCP_MAXIMUM_RETRIES</td>
<td>10</td>
</tr>
<tr>
<td>NX_TCP_MAXIMUM_TX_QUEUE</td>
<td>20</td>
</tr>
<tr>
<td>NX_TCP_MSS_KIND</td>
<td>0x02</td>
</tr>
<tr>
<td>NX_TCP_MSS_OPTION</td>
<td>0x02040000</td>
</tr>
<tr>
<td>NX_TCP_MSS_SIZE</td>
<td>1460</td>
</tr>
<tr>
<td>NX_TCP_NOP_KIND</td>
<td>0x01</td>
</tr>
<tr>
<td>NX_TCP_OPTION_END</td>
<td>0x01010100</td>
</tr>
<tr>
<td>NX_TCP_PACKET</td>
<td>56</td>
</tr>
<tr>
<td>NX_TCP_PORT_TABLE_MASK</td>
<td>0x1F</td>
</tr>
<tr>
<td>NX_TCP_PORT_TABLE_SIZE</td>
<td>32</td>
</tr>
<tr>
<td>NX_TCP_PSH_BIT</td>
<td>0x00080000</td>
</tr>
<tr>
<td>NX_TCP_RETRY_SHIFT</td>
<td>0</td>
</tr>
<tr>
<td>NX_TCP_RST_BIT</td>
<td>0x00040000</td>
</tr>
<tr>
<td>NX_TCP_SERVER</td>
<td>2</td>
</tr>
<tr>
<td>NX_TCP_SYN_BIT</td>
<td>0x00020000</td>
</tr>
<tr>
<td>NX_TCP_SYN_HEADER</td>
<td>0x70000000</td>
</tr>
<tr>
<td>NX_TCP_SYN_RECEIVED</td>
<td>4</td>
</tr>
<tr>
<td>NX_TCP_SYN_SENT</td>
<td>3</td>
</tr>
<tr>
<td>NX_TCP_TIMED_WAIT</td>
<td>10</td>
</tr>
<tr>
<td>NX_TCP_TRANSMIT_TIMER_RATE</td>
<td>1</td>
</tr>
<tr>
<td>NX_TCP URG_BIT</td>
<td>0x00200000</td>
</tr>
<tr>
<td>NX_TRUE</td>
<td>1</td>
</tr>
<tr>
<td>NX_TX_QUEUE_DEPTH</td>
<td>0x49</td>
</tr>
<tr>
<td>NX_UDP_ID</td>
<td>0x55445020</td>
</tr>
<tr>
<td>NX_UDP_PACKET</td>
<td>44</td>
</tr>
<tr>
<td>Symbol</td>
<td>Value</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>NX_UDP_PORT_TABLE_MASK</td>
<td>0x1F</td>
</tr>
<tr>
<td>NX_UDP_PORT_TABLE_SIZE</td>
<td>32</td>
</tr>
<tr>
<td>NX_UNDERFLOW</td>
<td>0x02</td>
</tr>
<tr>
<td>NX_UNHANDLED_COMMAND</td>
<td>0x44</td>
</tr>
<tr>
<td>NX_WAIT_ABORTED</td>
<td>0x1A</td>
</tr>
<tr>
<td>NX_WAIT_ERROR</td>
<td>0x08</td>
</tr>
<tr>
<td>NX_WAIT_FOREVER</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>NX_WINDOW_OVERFLOW</td>
<td>0x39</td>
</tr>
</tbody>
</table>
# Listings by Value

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX_ANY_PORT</td>
<td>0</td>
</tr>
<tr>
<td>NX_ARP_EXPIRATION_RATE</td>
<td>0</td>
</tr>
<tr>
<td>NX_FALSE</td>
<td>0</td>
</tr>
<tr>
<td>NX_ICMP_ECHO_REPLY_TYPE</td>
<td>0</td>
</tr>
<tr>
<td>NX_ICMP_NETWORK_UNREACH_CODE</td>
<td>0</td>
</tr>
<tr>
<td>NX_LINK_PACKET_SEND</td>
<td>0</td>
</tr>
<tr>
<td>NX_NO_WAIT</td>
<td>0</td>
</tr>
<tr>
<td>NX_NULL</td>
<td>0</td>
</tr>
<tr>
<td>NX_RECEIVE_PACKET</td>
<td>0</td>
</tr>
<tr>
<td>NX_TCP_RETRY_SHIFT</td>
<td>0</td>
</tr>
<tr>
<td>NX_SUCCESS</td>
<td>0x00</td>
</tr>
<tr>
<td>NX_TCP_EOL_KIND</td>
<td>0x00</td>
</tr>
<tr>
<td>NX_FRAGMENT_OKAY</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_A_TYPE</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_D_HOSTID</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NX_IP_NORMAL</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NX_FOREVER</td>
<td>1</td>
</tr>
<tr>
<td>NX_ICMP_HOST_UNREACH_CODE</td>
<td>1</td>
</tr>
<tr>
<td>NX_IGMP_HOST_VERSION_1</td>
<td>1</td>
</tr>
<tr>
<td>NX_IGMP_TTL</td>
<td>1</td>
</tr>
<tr>
<td>NX_INIT_PACKET_ID</td>
<td>1</td>
</tr>
<tr>
<td>NX_LINK_INITIALIZE</td>
<td>1</td>
</tr>
<tr>
<td>NX_TCP_CLIENT</td>
<td>1</td>
</tr>
<tr>
<td>NX_TCP_CLOSED</td>
<td>1</td>
</tr>
<tr>
<td>NX_TCP_TRANSMIT_TIMER_RATE</td>
<td>1</td>
</tr>
<tr>
<td>NX_TRUE</td>
<td>1</td>
</tr>
<tr>
<td>NX_IP_PERIODIC_EVENT</td>
<td>0x00000001</td>
</tr>
<tr>
<td>NX_ARP_HARDWARE_TYPE</td>
<td>0x0001</td>
</tr>
<tr>
<td>NX_ARP_OPTION_REQUEST</td>
<td>0x0001</td>
</tr>
<tr>
<td>NX_IP_INITIALIZE_DONE</td>
<td>0x0001</td>
</tr>
<tr>
<td>NX_RARP_HARDWARE_TYPE</td>
<td>0x0001</td>
</tr>
<tr>
<td>NX_NO_PACKET</td>
<td>0x01</td>
</tr>
<tr>
<td>NX_TCP_NOP_KIND</td>
<td>0x01</td>
</tr>
<tr>
<td>Variable</td>
<td>Value</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>NX_ICMP_PROTOCOL_UNREACH_CODE</td>
<td>2</td>
</tr>
<tr>
<td>NX_IGMP_HOST_VERSION_2</td>
<td>2</td>
</tr>
<tr>
<td>NX_LINK_ENABLE</td>
<td>2</td>
</tr>
<tr>
<td>NX_TCP_LISTEN_STATE</td>
<td>2</td>
</tr>
<tr>
<td>NX_TCP_SERVER</td>
<td>2</td>
</tr>
<tr>
<td>NX_IP_UNFRAG_EVENT</td>
<td>0x00000002</td>
</tr>
<tr>
<td>NX_ARP_OPTION_RESPONSE</td>
<td>0x0002</td>
</tr>
<tr>
<td>NX_IP_ADDRESS_RESOLVED</td>
<td>0x0002</td>
</tr>
<tr>
<td>NX_TCP_MSS_KIND</td>
<td>0x02</td>
</tr>
<tr>
<td>NX_UNDERFLOW</td>
<td>0x02</td>
</tr>
<tr>
<td>NX_ICMP_DEST_UNREACHABLE_TYPE</td>
<td>3</td>
</tr>
<tr>
<td>NX_ICMP_PORT_UNREACHABLE_CODE</td>
<td>3</td>
</tr>
<tr>
<td>NX_LINK_DISABLE</td>
<td>3</td>
</tr>
<tr>
<td>NX_TCP_SYN_SENT</td>
<td>3</td>
</tr>
<tr>
<td>NX_RARP_OPTION_REQUEST</td>
<td>0x0003</td>
</tr>
<tr>
<td>NX_OVERFLOW</td>
<td>0x03</td>
</tr>
<tr>
<td>NX_ARP_MAX_QUEUE_DEPTH</td>
<td>4</td>
</tr>
<tr>
<td>NX_ICMP_FRAMENT_NEEDED_CODE</td>
<td>4</td>
</tr>
<tr>
<td>NX_ICMP_SOURCE_QUENCH_TYPE</td>
<td>4</td>
</tr>
<tr>
<td>NX_LINK_PACKET_BROADCAST</td>
<td>4</td>
</tr>
<tr>
<td>NX_PHYSICAL_TRAILER</td>
<td>4</td>
</tr>
<tr>
<td>NX_TCP_SYN_RECEIVED</td>
<td>4</td>
</tr>
<tr>
<td>NX_IP_ICMP_EVENT</td>
<td>0x00000004</td>
</tr>
<tr>
<td>NX_IP_LINK_ENABLED</td>
<td>0x0004</td>
</tr>
<tr>
<td>NX_RARP_OPTION_RESPONSE</td>
<td>0x0004</td>
</tr>
<tr>
<td>NX_ARP_PROTOCOL_SIZE</td>
<td>0x04</td>
</tr>
<tr>
<td>NX_NO_MAPPING</td>
<td>0x04</td>
</tr>
<tr>
<td>NX_RARP_PROTOCOL_SIZE</td>
<td>0x04</td>
</tr>
<tr>
<td>NX_ICMP_REDIRECT_TYPE</td>
<td>5</td>
</tr>
<tr>
<td>NX_ICMP_SOURCE_ROUTE_CODE</td>
<td>5</td>
</tr>
<tr>
<td>NX_IP_NORMAL_LENGTH</td>
<td>5</td>
</tr>
<tr>
<td>NX_LINK_ARP_SEND</td>
<td>5</td>
</tr>
<tr>
<td>NX_TCP_ACK_TIMER_RATE</td>
<td>5</td>
</tr>
<tr>
<td>NX_TCP_ESTABLISHED</td>
<td>5</td>
</tr>
<tr>
<td>NX_DELETED</td>
<td>0x05</td>
</tr>
<tr>
<td>Constant</td>
<td>Value</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>NX_ICMP_NETWORK_UNKNOWN_CODE</td>
<td>6</td>
</tr>
<tr>
<td>NX_LINK_ARP_RESPONSE_SEND</td>
<td>6</td>
</tr>
<tr>
<td>NX_TCP_CLOSE_WAIT</td>
<td>6</td>
</tr>
<tr>
<td>NX_ARP_HARDWARE_SIZE</td>
<td>0x06</td>
</tr>
<tr>
<td>NX_POOL_ERROR</td>
<td>0x06</td>
</tr>
<tr>
<td>NX_RARP_HARDWARE_SIZE</td>
<td>0x06</td>
</tr>
<tr>
<td>NX_ICMP_HOST_UNKNOWN_CODE</td>
<td>7</td>
</tr>
<tr>
<td>NX_LINK_RARP_SEND</td>
<td>7</td>
</tr>
<tr>
<td>NX_MAX_MULTICAST_GROUPS</td>
<td>7</td>
</tr>
<tr>
<td>NX_TCP_FIN_WAIT_1</td>
<td>7</td>
</tr>
<tr>
<td>NX_PTR_ERROR</td>
<td>0x07</td>
</tr>
<tr>
<td>NX_ICMP_ECHO_REQUEST_TYPE</td>
<td>8</td>
</tr>
<tr>
<td>NX_ICMP_SOURCE_ISOLATED_CODE</td>
<td>8</td>
</tr>
<tr>
<td>NX_IP_ALIGN_FRAGS</td>
<td>8</td>
</tr>
<tr>
<td>NX_LINK_MULTICAST_JOIN</td>
<td>8</td>
</tr>
<tr>
<td>NX_TCP_FIN_WAIT_2</td>
<td>8</td>
</tr>
<tr>
<td>NX_IGMP_HEADER_SIZE</td>
<td>8</td>
</tr>
<tr>
<td>NX_IP_RECEIVE_EVENT</td>
<td>0x00000008</td>
</tr>
<tr>
<td>NX_IP_ARP_ENABLED</td>
<td>0x0008</td>
</tr>
<tr>
<td>NX_WAIT_ERROR</td>
<td>0x08</td>
</tr>
<tr>
<td>NX_ICMP_NETWORK_PROHIBIT_CODE</td>
<td>9</td>
</tr>
<tr>
<td>NX_LINK_MULTICAST_LEAVE</td>
<td>9</td>
</tr>
<tr>
<td>NX_TCP_CLOSING</td>
<td>9</td>
</tr>
<tr>
<td>NX_SIZE_ERROR</td>
<td>0x09</td>
</tr>
<tr>
<td>NX_ARP_UPDATE_RATE</td>
<td>10</td>
</tr>
<tr>
<td>NX_ICMP_HOST_PROHIBIT_CODE</td>
<td>10</td>
</tr>
<tr>
<td>NX_IGMP_MAX_UPDATE_TIME</td>
<td>10</td>
</tr>
<tr>
<td>NX_LINK_GET_STATUS</td>
<td>10</td>
</tr>
<tr>
<td>NX_MAX_LISTEN_REQUESTS</td>
<td>10</td>
</tr>
<tr>
<td>NX_TCP_FAST_TIMER_RATE</td>
<td>10</td>
</tr>
<tr>
<td>NX_TCP_KEEPALIVE_RETRIES</td>
<td>10</td>
</tr>
<tr>
<td>NX_TCP_MAXIMUM_RETRIES</td>
<td>10</td>
</tr>
<tr>
<td>NX_TCP_TIMED_WAIT</td>
<td>10</td>
</tr>
<tr>
<td>NX_OPTION_ERROR</td>
<td>0x0A</td>
</tr>
<tr>
<td>NX_ICMP_NETWORK_SERVICE_CODE</td>
<td>11</td>
</tr>
<tr>
<td>Code Name</td>
<td>Value</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>NX_ICMP_TIME_EXCEEDED_TYPE</td>
<td>11</td>
</tr>
<tr>
<td>NX_LINK_GET_SPEED</td>
<td>11</td>
</tr>
<tr>
<td>NX_TCP_LAST_ACK</td>
<td>11</td>
</tr>
<tr>
<td>NX_ICMP_HOST_SERVICE_CODE</td>
<td>12</td>
</tr>
<tr>
<td>NX_ICMP_PARAMETER_PROB_TYPE</td>
<td>12</td>
</tr>
<tr>
<td>NX_LINK_GET_DUPLEX_TYPE</td>
<td>12</td>
</tr>
<tr>
<td>NX_ICMP_TIMESTAMP_REQ_TYPE</td>
<td>13</td>
</tr>
<tr>
<td>NX_LINK_GET_ERROR_COUNT</td>
<td>13</td>
</tr>
<tr>
<td>NX_ICMP_TIMESTAMP_REP_TYPE</td>
<td>14</td>
</tr>
<tr>
<td>NX_LINK_GET_RX_COUNT</td>
<td>14</td>
</tr>
<tr>
<td>NX_LINK_GET_TX_COUNT</td>
<td>15</td>
</tr>
<tr>
<td>NX_LINK_GET_ALLOC_ERRORS</td>
<td>16</td>
</tr>
<tr>
<td>NX_PHYSICAL_HEADER</td>
<td>16</td>
</tr>
<tr>
<td>NX_SHIFT_BY_16</td>
<td>16</td>
</tr>
<tr>
<td>NX_IP_ARP_REC_EVENT</td>
<td>0x00000010</td>
</tr>
<tr>
<td>NX_IP_UDP_ENABLED</td>
<td>0x0010</td>
</tr>
<tr>
<td>NX_DELETE_ERROR</td>
<td>0x10</td>
</tr>
<tr>
<td>NX_ICMP_ADDRESS_MASK_REQ_TYPE</td>
<td>17</td>
</tr>
<tr>
<td>NX_LINK_UNINITIALIZE</td>
<td>17</td>
</tr>
<tr>
<td>NX_CALLER_ERROR</td>
<td>0x11</td>
</tr>
<tr>
<td>NX_ARP_MAXIMUM_RETRIES</td>
<td>18</td>
</tr>
<tr>
<td>NX_ICMP_ADDRESS_MASK_REP_TYPE</td>
<td>18</td>
</tr>
<tr>
<td>NX_LINK_DEFERRED_PROCESSING</td>
<td>18</td>
</tr>
<tr>
<td>NX_INVALID_PACKET</td>
<td>0x12</td>
</tr>
<tr>
<td>NX_INVALID_SOCKET</td>
<td>0x13</td>
</tr>
<tr>
<td>NX_LINK_INTERFACE_ATTACH</td>
<td>19</td>
</tr>
<tr>
<td>NX_TCP_MAXIMUM_TX_QUEUE</td>
<td>20</td>
</tr>
<tr>
<td>NX_NOT_ENABLED</td>
<td>0x14</td>
</tr>
<tr>
<td>NX_ALREADY_ENABLED</td>
<td>0x15</td>
</tr>
<tr>
<td>NX_ENTRY_NOT_FOUND</td>
<td>0x16</td>
</tr>
<tr>
<td>NX_NO_MORE_ENTRIES</td>
<td>0x17</td>
</tr>
<tr>
<td>NX_IP_TIME_TO_LIVE_SHIFT</td>
<td>24</td>
</tr>
<tr>
<td>NX_ARP_TIMER_ERROR</td>
<td>0x18</td>
</tr>
<tr>
<td>NX_RESERVED_CODE0</td>
<td>0x19</td>
</tr>
<tr>
<td>NX_WAIT_ABORTED</td>
<td>0x1A</td>
</tr>
<tr>
<td>Constant Name</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>NX_ARP_MESSAGE_SIZE</td>
<td>28</td>
</tr>
<tr>
<td>NX_RARP_MESSAGE_SIZE</td>
<td>28</td>
</tr>
<tr>
<td>NX_TCP_HEADER_SHIFT</td>
<td>28</td>
</tr>
<tr>
<td>NX_ROUTE_TABLE_MASK</td>
<td>0x1F</td>
</tr>
<tr>
<td>NX_TCP_PORT_TABLE_MASK</td>
<td>0x1F</td>
</tr>
<tr>
<td>NX_UDP_PORT_TABLE_MASK</td>
<td>0x1F</td>
</tr>
<tr>
<td>NX_ROUTE_TABLE_SIZE</td>
<td>32</td>
</tr>
<tr>
<td>NX_TCP_PORT_TABLE_SIZE</td>
<td>32</td>
</tr>
<tr>
<td>NX_UDP_PORT_TABLE_SIZE</td>
<td>32</td>
</tr>
<tr>
<td>NX_IP_RARP_REC_EVENT</td>
<td>0x00000020</td>
</tr>
<tr>
<td>NX_IP_TCP_ENABLED</td>
<td>0x0020</td>
</tr>
<tr>
<td>NX_IP_INTERNAL_ERROR</td>
<td>0x20</td>
</tr>
<tr>
<td>NX_IP_ADDRESS_ERROR</td>
<td>0x21</td>
</tr>
<tr>
<td>NX_ALREADY_BOUND</td>
<td>0x22</td>
</tr>
<tr>
<td>NX_PORT_UNAVAILABLE</td>
<td>0x23</td>
</tr>
<tr>
<td>NX_ICMP_PACKET</td>
<td>36</td>
</tr>
<tr>
<td>NX_IGMP_PACKET</td>
<td>36</td>
</tr>
<tr>
<td>NX_IP_PACKET</td>
<td>36</td>
</tr>
<tr>
<td>NX_IPV4_ICMP_PACKET</td>
<td>36</td>
</tr>
<tr>
<td>NX_IPV4_IGMP_PACKET</td>
<td>36</td>
</tr>
<tr>
<td>NX_NOT_BOUND</td>
<td>0x24</td>
</tr>
<tr>
<td>NX_RESERVED_CODE1</td>
<td>0x25</td>
</tr>
<tr>
<td>NX_SOCKET_UNBOUND</td>
<td>0x26</td>
</tr>
<tr>
<td>NX_NOT_CREATED</td>
<td>0x27</td>
</tr>
<tr>
<td>NX_SOCKETS_BOUND</td>
<td>0x28</td>
</tr>
<tr>
<td>NX_NO_RESPONSE</td>
<td>0x29</td>
</tr>
<tr>
<td>NX_IPV4_UDP_PACKET</td>
<td>44</td>
</tr>
<tr>
<td>NX_UDP_PACKET</td>
<td>44</td>
</tr>
<tr>
<td>NX_POOL_DELETED</td>
<td>0x30</td>
</tr>
<tr>
<td>NX_ALREADY_RELEASED</td>
<td>0x31</td>
</tr>
<tr>
<td>NX_LINK_USER_COMMAND</td>
<td>50</td>
</tr>
<tr>
<td>NX_RESERVED_CODE2</td>
<td>0x32</td>
</tr>
<tr>
<td>NX_MAX_LISTEN</td>
<td>0x33</td>
</tr>
<tr>
<td>NX_DUPLICATE_LISTEN</td>
<td>0x34</td>
</tr>
<tr>
<td>NX_NOT_CLOSED</td>
<td>0x35</td>
</tr>
<tr>
<td>Symbol</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>NX_NOT_LISTEN_STATE</td>
<td>0x36</td>
</tr>
<tr>
<td>NX_IN_PROGRESS</td>
<td>0x37</td>
</tr>
<tr>
<td>NX_NOT_CONNECTED</td>
<td>0x38</td>
</tr>
<tr>
<td>NX_IPV4_TCP_PACKET</td>
<td>56</td>
</tr>
<tr>
<td>NX_TCP_PACKET</td>
<td>56</td>
</tr>
<tr>
<td>NX_WINDOW_OVERFLOW</td>
<td>0x39</td>
</tr>
<tr>
<td>NX_IP_IGMP_EVENT</td>
<td>0x00000040</td>
</tr>
<tr>
<td>NX_IP_IGMP_ENABLED</td>
<td>0x0040</td>
</tr>
<tr>
<td>NX_ALREADY_SUSPENDED</td>
<td>0x40</td>
</tr>
<tr>
<td>NX_DISCONNECT_FAILED</td>
<td>0x41</td>
</tr>
<tr>
<td>NX_STILL_BOUND</td>
<td>0x42</td>
</tr>
<tr>
<td>NX_NOT_SUCCESSFUL</td>
<td>0x43</td>
</tr>
<tr>
<td>NX_UNHANDLED_COMMAND</td>
<td>0x44</td>
</tr>
<tr>
<td>NX_NO_FREE_PORTS</td>
<td>0x45</td>
</tr>
<tr>
<td>NX_INVALID_PORT</td>
<td>0x46</td>
</tr>
<tr>
<td>NX_INVALID_RELISTEN</td>
<td>0x47</td>
</tr>
<tr>
<td>NX_CONNECTION_PENDING</td>
<td>0x48</td>
</tr>
<tr>
<td>NX_TX_QUEUE_DEPTH</td>
<td>0x49</td>
</tr>
<tr>
<td>NX_NOT_IMPLEMENTED</td>
<td>0x4A</td>
</tr>
<tr>
<td>NX_NOT_SUPPORTED</td>
<td>0x4B</td>
</tr>
<tr>
<td>NX_TCP_KEEPALIVE_RETRY</td>
<td>75</td>
</tr>
<tr>
<td>NX_INVALID_INTERFACE</td>
<td>0x4C</td>
</tr>
<tr>
<td>NX_ARP_DEBUG_LOG_SIZE</td>
<td>100</td>
</tr>
<tr>
<td>NX_ICMP_DEBUG_LOG_SIZE</td>
<td>100</td>
</tr>
<tr>
<td>NX_IGMP_DEBUG_LOG_SIZE</td>
<td>100</td>
</tr>
<tr>
<td>NX_IP_DEBUG_LOG_SIZE</td>
<td>100</td>
</tr>
<tr>
<td>NX_IP_PERIODIC_RATE</td>
<td>100</td>
</tr>
<tr>
<td>NX_PACKET_DEBUG_LOG_SIZE</td>
<td>100</td>
</tr>
<tr>
<td>NX_RARP_DEBUG_LOG_SIZE</td>
<td>100</td>
</tr>
<tr>
<td>NX_TCP_DEBUG_LOG_SIZE</td>
<td>100</td>
</tr>
<tr>
<td>NX_UDP_DEBUG_LOG_SIZE</td>
<td>100</td>
</tr>
<tr>
<td>NX_IP_TCP_EVENT</td>
<td>0x00000080</td>
</tr>
<tr>
<td>NX_IP_TIME_TO_LIVE</td>
<td>0x00000080</td>
</tr>
<tr>
<td>NX_IP_RARP_COMPLETE</td>
<td>0x0080</td>
</tr>
<tr>
<td>NX_NOT_IMPLEMENTED</td>
<td>0x4A</td>
</tr>
<tr>
<td>Symbol</td>
<td>Value</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>NX_IP_CLASS_C_HOSTID</td>
<td>0x000000FF</td>
</tr>
<tr>
<td>NX_IP_MULTICAST_UPPER</td>
<td>0x00000100</td>
</tr>
<tr>
<td>NX_IP_TCP_FAST_EVENT</td>
<td>0x00000100</td>
</tr>
<tr>
<td>NX_IP_DRIVER_PACKET_EVENT</td>
<td>0x00000200</td>
</tr>
<tr>
<td>NX_IP_IGMP_ENABLE_EVENT</td>
<td>0x00000400</td>
</tr>
<tr>
<td>NX_IP_DRIVER_DEFERRED_EVENT</td>
<td>0x00000800</td>
</tr>
<tr>
<td>NX_ARP_PROTOCOL_TYPE</td>
<td>0x0800</td>
</tr>
<tr>
<td>NX_RARP_PROTOCOL_TYPE</td>
<td>0x0800</td>
</tr>
<tr>
<td>NX_IP_TCP_CLEANUP_DEFERRED</td>
<td>0x00001000</td>
</tr>
<tr>
<td>NX_TCP_KEEPALIVE_INITIAL</td>
<td>7200</td>
</tr>
<tr>
<td>NX_FRAG_OFFSET_MASK</td>
<td>0x00001FFFF</td>
</tr>
<tr>
<td>NX_IP_OFFSET_MASK</td>
<td>0x00001FFFF</td>
</tr>
<tr>
<td>NX_IP_MORE_FRAGMENT</td>
<td>0x00002000</td>
</tr>
<tr>
<td>NX_MORE_FRAGMENTS</td>
<td>0x00002000</td>
</tr>
<tr>
<td>NX_IP_FRAGMENT_MASK</td>
<td>0x00003FFF</td>
</tr>
<tr>
<td>NX_TCP_MSS_SIZE</td>
<td>16384</td>
</tr>
<tr>
<td>NX_DONT_FRAGMENT</td>
<td>0x00004000</td>
</tr>
<tr>
<td>NX_IP_DONT_FRAGMENT</td>
<td>0x00004000</td>
</tr>
<tr>
<td>NX_SEARCH_PORT_START</td>
<td>30000</td>
</tr>
<tr>
<td>NX_IP_CLASS_B_HOSTID</td>
<td>0x0000FFFF</td>
</tr>
<tr>
<td>NX_IP_PACKET_SIZE_MASK</td>
<td>0x0000FFFF</td>
</tr>
<tr>
<td>NX_LOWER_16_MASK</td>
<td>0x0000FFFF</td>
</tr>
<tr>
<td>NX_MAX_PORT</td>
<td>0xFFFF</td>
</tr>
<tr>
<td>NX_IP_ICMP</td>
<td>0x00010000</td>
</tr>
<tr>
<td>NX_TCP_FIN_BIT</td>
<td>0x00010000</td>
</tr>
<tr>
<td>NX_CARRY_BIT</td>
<td>0x10000</td>
</tr>
<tr>
<td>NX_IP_IGMP</td>
<td>0x00020000</td>
</tr>
<tr>
<td>NX_IP_MIN_COST</td>
<td>0x00020000</td>
</tr>
<tr>
<td>NX_TCP_SYN_BIT</td>
<td>0x00020000</td>
</tr>
<tr>
<td>NX_IP_MAX_RELIABLE</td>
<td>0x00040000</td>
</tr>
<tr>
<td>NX_TCP_RST_BIT</td>
<td>0x00040000</td>
</tr>
<tr>
<td>NX_IP_TCP</td>
<td>0x00060000</td>
</tr>
<tr>
<td>NX_IP_MAX_DATA</td>
<td>0x00080000</td>
</tr>
<tr>
<td>NX_TCP_PSH_BIT</td>
<td>0x00080000</td>
</tr>
<tr>
<td>NX_IP_MIN_DELAY</td>
<td>0x00100000</td>
</tr>
<tr>
<td>Symbol</td>
<td>Value</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>NX_TCP_ACK_BIT</td>
<td>0x00100000</td>
</tr>
<tr>
<td>NX_IP_UDP</td>
<td>0x00110000</td>
</tr>
<tr>
<td>NX_TCP_CONTROL_MASK</td>
<td>0x00170000</td>
</tr>
<tr>
<td>NX_TCP_URG_BIT6</td>
<td>0x00200000</td>
</tr>
<tr>
<td>NX_IP_MULTICAST_MASK</td>
<td>0x007FFFFFFF</td>
</tr>
<tr>
<td>NX_IP_PROTOCOL_MASK</td>
<td>0x00FF0000</td>
</tr>
<tr>
<td>NX_IP_TOS_MASK</td>
<td>0x00FFE0000</td>
</tr>
<tr>
<td>NX_IP_MULTICAST_LOWER</td>
<td>0x5E000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_A_HOSTID</td>
<td>0x00FFFFFF</td>
</tr>
<tr>
<td>NX_IP_CLASS_D_GROUP</td>
<td>0x0FFFFFFF</td>
</tr>
<tr>
<td>NX_IGMP_ROUTER_QUERY_TYPE</td>
<td>0x01000000</td>
</tr>
<tr>
<td>NX_TCP_OPTION_END</td>
<td>0x01010402</td>
</tr>
<tr>
<td>NX_TCP_MSS_OPTION</td>
<td>0x02000000</td>
</tr>
<tr>
<td>NX_TCP_MSS_MASK</td>
<td>0x0F000000</td>
</tr>
<tr>
<td>NX_IP_PROTOCOL_MASK</td>
<td>0x02040000</td>
</tr>
<tr>
<td>NX_IP_CLASS_D_GROUP</td>
<td>0x0F000000</td>
</tr>
<tr>
<td>NX_IGMP_MAX_RESP_TIME_MASK</td>
<td>0x00FFE0000</td>
</tr>
<tr>
<td>NX_IP_CLASS_A_HOSTID</td>
<td>0x00FFFFFF</td>
</tr>
<tr>
<td>NX_IP_CLASS_D_GROUP</td>
<td>0x0FFFFFFF</td>
</tr>
<tr>
<td>NX_IGMP_VERSION</td>
<td>0x10000000</td>
</tr>
<tr>
<td>NX_IGMP_HOST_V2_JOIN_TYPE</td>
<td>0x16000000</td>
</tr>
<tr>
<td>NX_IGMP_HOST_V2_LEAVE_TYPE</td>
<td>0x17000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_C_NETID</td>
<td>0x1FFFFFFF</td>
</tr>
<tr>
<td>NX_IP_CLASS_B_NETID</td>
<td>0x3FFF0000</td>
</tr>
<tr>
<td>NX_IGMP_MAX_RESP_TIME_MASK</td>
<td>0x00FFE0000</td>
</tr>
<tr>
<td>NX_IP_CLASS_A_NETID</td>
<td>0x7F000000</td>
</tr>
<tr>
<td>NX_TCP_SYN_HEADER</td>
<td>0x70000000</td>
</tr>
<tr>
<td>NX_TCP_HEADER_SIZE</td>
<td>0x7F000000</td>
</tr>
<tr>
<td>NX_TCP_HEADER_SIZE</td>
<td>0x7F000000</td>
</tr>
<tr>
<td>NX_PACKET_POOL_ID</td>
<td>0x5041434B</td>
</tr>
<tr>
<td>NX_TCP_ID</td>
<td>0x54435020</td>
</tr>
<tr>
<td>NX_UDP_ID</td>
<td>0x55445020</td>
</tr>
<tr>
<td>NX_IP_MULTICAST_LOWER</td>
<td>0x5E000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_A_NETID</td>
<td>0x7F000000</td>
</tr>
<tr>
<td>NX_TCP_SYN_HEADER</td>
<td>0x70000000</td>
</tr>
<tr>
<td>NX_IP_LOOPBACK_FIRST</td>
<td>0x7F000000</td>
</tr>
<tr>
<td>NX_IP_LOOPBACK_LAST</td>
<td>0x7FFFFFFF</td>
</tr>
<tr>
<td>NX_IP_CLASS_A_MASK</td>
<td>0x80000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_B_TYPE</td>
<td>0x80000000</td>
</tr>
<tr>
<td>NX_PACKET_ALLOCATED</td>
<td>0xAAAAA9AAA</td>
</tr>
<tr>
<td>Constant</td>
<td>Value</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>NX_PACKET_READY</td>
<td>0xBBBBBBBB</td>
</tr>
<tr>
<td>NX_IP_CLASS_B_MASK</td>
<td>0xC0000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_C_TYPE</td>
<td>0xC0000000</td>
</tr>
<tr>
<td>NX_DRIVER_TX_DONE</td>
<td>0xDDDDDDDD</td>
</tr>
<tr>
<td>NX_IP_CLASS_C_MASK</td>
<td>0xE0000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_D_TYPE</td>
<td>0xE0000000</td>
</tr>
<tr>
<td>NX_PACKET_ENQUEUED</td>
<td>0xEEEEEEEE</td>
</tr>
<tr>
<td>NX_IGMP_VERSION_MASK</td>
<td>0xF0000000</td>
</tr>
<tr>
<td>NX_IP_CLASS_D_MASK</td>
<td>0xF0000000</td>
</tr>
<tr>
<td>NX_TCP_HEADER_MASK</td>
<td>0xF0000000</td>
</tr>
<tr>
<td>NX_ALL_HOSTS_ADDRESS</td>
<td>0xFE000001</td>
</tr>
<tr>
<td>NX_IGMPV2_TYPE_MASK</td>
<td>0xFF000000</td>
</tr>
<tr>
<td>NX_IP_TIME_TO_LIVE_MASK</td>
<td>0xFF000000</td>
</tr>
<tr>
<td>NX_IP_ALL_EVENTS</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>NX_IP_LIMITIED_BROADCAST</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>NX_PACKET_FREE</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>NX_WAIT_FOREVER</td>
<td>0xFFFFFFFF</td>
</tr>
</tbody>
</table>
NetX Data Types

- NX_ARP 408
- NX_INTERFACE 408
- NX_IP 411
- NX_IP_DRIVER 411
- NX_IP_ROUTING_ENTRY 412
- NX_PACKET 412
- NX_PACKET_POOL 412
- NX_TCP_LISTEN 412
- NX_TCP_SOCKET 414
- NX_UDP_SOCKET 414
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       nx_interface_link_status_change;
    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;
    ULONG       nx_interface_ip_mtu_size;
    VOID        (*nx_interface_link_driver_entry)
    (struct NX_IP_DRIVER_STRUCT *);
    VOID       (*nx_interface_additional_link_info);
    ULONG       nx_interface_arp_defend_timeout;
} NX_INTERFACE;

typedef struct NX_IP_STRUCT
{
    ULONG       nx_ip_id;
    CHAR        *nx_ip_name;
#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_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;
    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_packetsreassembled;
    ULONG       nx_ip_reassembly_failures;
    ULONG       nx_ip_invalid_packets;
    ULONG       nx_ip_invalid_transmit_packets;
    ULONG       nx_ip_invalid_receive_address;
} NX_IP;
NetX Data Types

**ULONG**

- nx_ip_unknown_protocols_received;
- nx_ip_transmit_resource_errors;
- nx_ip_transmit_no_route_errors;
- nx_ip_receive_packets_dropped;
- nx_ip_receive_checksum_errors;
- nx_ip_send_packets_dropped;
- nx_ip_total_fragment_requests;
- nx_ip_successful_fragment_requests;
- nx_ip_fragment_failures;
- nx_ip_total_fragments_sent;
- nx_ip_total_fragments_received;
- nx_ip_unknown_protocols_received;
- nx_ip_transmit_resource_errors;
- nx_ip_transmit_no_route_errors;
- nx_ip_receive_packets_dropped;
- nx_ip_receive_checksum_errors;
- nx_ip_send_packets_dropped;
- nx_ip_total_fragment_requests;
- nx_ip_successful_fragment_requests;
- nx_ip_fragment_failures;
- nx_ip_total_fragments_sent;
- nx_ip_total_fragments_received;
- nx_ip_arp_requests_sent;
- nx_ip_arp_requests_received;
- nx_ip_arp_responses_sent;
- nx_ip_arp_responses_received;
- nx_ip_arp_aged_entries;
- nx_ip_arp_invalid_messages;
- nx_ip_arp_static_entries;
- nx_ip_udp_packets_sent;
- nx_ip_udp_bytes_sent;
- nx_ip_udp_packets_received;
- nx_ip_udp_bytes_received;
- nx_ip_udp_invalid_packets;
- nx_ip_udp_no_port_for_delivery;
- nx_ip_udp_receive_packets_dropped;
- nx_ip_udp_checksum_errors;
- nx_ip_UDP_packets_sent;
- nx_ip_UDP_bytes_sent;
- nx_ip_UDP_packets_received;
- nx_ip_UDP_bytes_received;
- nx_ip_UDP_invalid_packets;
- nx_ip_UDP_no_port_for_delivery;
- nx_ip_UDP_receive_packets_dropped;
- nx_ip_UDP_checksum_errors;
- nx_ip_TCP_packets_sent;
- nx_ip_TCP_bytes_sent;
- nx_ip_TCP_packets_received;
- nx_ip_TCP_bytes_received;
- nx_ip_TCP_invalid_packets;
- nx_ip_TCP_no_port_for_delivery;
- nx_ip_TCP_receive_packets_dropped;
- nx_ip_TCP_checksum_errors;
- nx_ip_TCP_connections;
- nx_ip_TCP_active_connections;
- nx_ip_TCP_disconnections;
- nx_ip_TCP_connections_dropped;
- nx_ip_TCP_retransmit_packets;
- nx_ip_TCP_resets_received;
- nx_ip_TCP_resets_sent;
- nx_ip_ICMP_total_messages_received;
- nx_ip_ICMP_checksum_errors;
- nx_ip_ICMP_unhandled_messages;
- nx_ip_ICMP_invalid_packets;
- nx_ip_ICMP_ignored_messages;
- nx_ip_ICMP_reports_sent;
- nx_ip_ICMP_queries_received;
- nx_ip_ICMP_check_sum_errors;
- nx_ip_ICMP_groups_joined;
- nx_ip_ICMP_router_version;
- nx_ip_rarp_requests_sent;
- nx_ip_rarp_responses_received;
- nx_ip_rarp_invalid_messages;

**VOID**

- void (*nx_ip_forward_packet_process)(struct NX_IP_STRUCT *, NX_PACKET *);
- void (*nx_ip_packet_id);
- struct NX_PACKET_POOL_STRUCT *nx_ip_default_packet_pool;
- TX_MUTEX nx_ip_protection;
- UINT nx_ip_initialize_done;
- void (*nx_ip_driver_deferred_packet_head);
- void (*nx_ip_driver_deferred_packet_tail);

**Express Logic**
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;
UINT (*nx_ip_raw_ip_processing)(struct NX_IP_STRUCT *,
                                ULONG, NX_PACKET *);
#ifdef NX_ENABLE_IP_RAW_PACKET_FILTER
UINT (*nx_ip_raw_packet_filter)(struct NX_IP_STRUCT *
                                , ULONG, NX_PACKET *);
#endif /* NX_ENABLE_IP_RAW_PACKET_FILTER */
NX_PACKET *nx_ip_raw_received_packet_head,
    *nx_ip_raw_received_packet_tail;
ULONG nx_ip_raw_received_packet_count,
    nx_ip_raw_received_packet_max;
TX_THREAD nx_ip_raw_packet_suspension_list;
ULONG nx_ip_raw_packet_suspended_count;
xip_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 *
                                         , VOID *);
void (*nx_ip_FRAGMENT_timeout_fragment)(struct
                                         NX_IP_STRUCT *
                                         , VOID *);
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_interfacejoin_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];
ULONG nx_ip_igmp_global_loopback_enable;
void (*nx_ip_igmp_group_loopback_enable[NX_MAX_MULTICAST_GROUPS]
       , void (*nx_ip_igmp_packet_receive)(struct NX_IP_STRUCT *,
                                             struct NX_PACKET_STRUCT *);
void (*nx_ip_igmp_periodic_processing)(struct NX_IP_STRUCT *
                                        , struct NX_PACKET_STRUCT *);
void (*nx_ip_igmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *atical_processing);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *atical_processing);
void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *
                                            , struct NX_PACKET_STRUCT *
                                            , struct NX_PACKET_STRUCT *atical_processing);
void (*nx_ip_tcp_queue_process)(struct NX_IP_STRUCT *);
void (*nx_ip_tcp_queue_head,
    struct NX_TCP_LISTEN_STRUCT *);
void (*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_LISTENSTRUCT nx_ip_tcp_available_listens_requests;
struct NX_TCP_LISTEN_STRUCT nx_ip_tcp_active_listens_requests;
UINT nx_ip_tcp_port_search_start;
TX_TIMER nx_ip_fast_periodic_timer,
    nx_ip_fast_periodic_timer_created;
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;
void (*nx_ip_arp_allocate)(struct NX_IP_STRUCT *,
    struct NX_ARP_STRUCT **, UINT);
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];
#define NX_ENABLE_IP_STATIC_ROUTING
    NX_IP_ROUTING_ENTRY nx_ip_routing_table[NX_IP_ROUTING_TABLE_SIZE];
    nx_ip_routing_table_entry_count;
#endif /* NX_ENABLE_IP_STATIC_ROUTING */
void (*nx_ip_link_status_change_callback)(struct NX_IP_STRUCT *,
    UINT, UINT);
#define NX_ENABLE_IP_PACKET_FILTER
    UX_PACKET (void *, UINT);
    (void *);
#endif /* NX_ENABLE_IP_PACKET_FILTER */
} 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;
    struct NX_IP_STRUCT nx_ip_driver_packet;
    struct NX_IP_STRUCT nx_ip_driver_return_ptr;
    struct NX_IP_STRUCT nx_ip_driver_ptr;
    struct NX_IP_STRUCT nx_ip_driver_interface;
} NX_IP_DRIVER;

typedef struct NX_IP_ROUTING_ENTRY_STRUCT
{    ULONG nx_ip_routing_dist_ip;
    ULONG nx_ip_routing_net_mask;
}
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_pad[NX_PACKET_HEADER_PAD_SIZE];
  #endif
} 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;
}
struct NX_INTERFACE_STRUCT
struct NX_IP_STRUCT

void (*nx_tcp_urgent_data_callback)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);

#endif /* NX_DISABLE_EXTENDED_NOTIFY_SUPPORT */

#endif /* NX_ENABLE_TCP_WINDOW_SCALING */

#define NX_ENABLE_TCP_WINDOW_SCALING

ulong nx_tcp_socket_rx_window_maximum;
ulong nx_tcp_rcv_win_scale_value;
ulong nx_tcp_snd_win_scale_value;

ulong nx_tcp_socket_keepalive_timeout;
ulong nx_tcp_socket_keepalive_retries;

struct NX_TCP_SOCKET_STRUCT

void

#ifdef NX_DISABLE_EXTENDED_NOTIFY_SUPPORT

#endif

#define NX_DISABLE_EXTENDED_NOTIFY_SUPPORT

struct NX_TCP_SOCKET_STRUCT

void

#ifdef NX_ENABLE_TCP_WINDOW_SCALING

#endif

ulong nx_tcp_socket_timeout_shift;

ulong nx_tcp_socket_timeout_max_retries;
ulong nx_tcp_socket_timeout_rate;
ulong nx_tcp_socket_timeout_retries;
ulong nx_tcp_socket_timeout_sent_count;
ulong nx_tcp_socket_timeout_current;
ulong nx_tcp_socket_timeout_default;
ulong nx_tcp_socket_timeout_window;
ulong nx_tcp_socket_timeout_send_window;
ulong nx_tcp_socket_timeout Şi;
typedef struct NX_TCP_SOCKET_STRUCT
{
    UINT (*nx_tcp_socket_syn_received_notify)(struct NX_TCP_SOCKET_STRUCT *socket_ptr,
                                              NX_PACKET *packet_ptr);
    VOID (*nx_tcp_establish_notify)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    VOID (*nx_tcp_disconnect_complete_notify)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    VOID (*nx_tcp_timed_wait_callback)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    #endif
    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;
    UINT    nx_tcp_socket_keepalive_enabled;
} 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;
    struct NX_INTERFACE_STRUCT *nx_udp_socket_ip_interface;
} NX_UDP_SOCKET;
BSD-Compatible Socket API
BSD-Compatible Socket API

The BSD-Compatible Socket API supports a subset of the BSD Sockets API calls (with some limitations) by utilizing NetX Duo® primitives underneath. Both IPv6 and IPv4 protocols and network addressing are supported. This BSD-Compatible Sockets API layer should perform as fast or slightly faster than typical BSD implementations because this API utilizes internal NetX Duo primitives and bypasses unnecessary NetX error checking.

Configurable options allow the host application to define the maximum number of sockets, TCP maximum window size, and depth of listen queue.

Due to performance and architecture constraints, this BSD-Compatible Sockets API does not support all BSD Sockets calls. In addition, not all BSD options are available for the BSD services, specifically the following:

- `select()` call works with only `fd_set *readfds`, other arguments in this call e.g., `writefds`, `exceptfds` are not supported.
- The “int flags” argument is not supported for `send()`, `recv()`, `sendto()`, and `recvfrom()` calls.
- The BSD-Compatible Socket API supports only limited set of BSD Sockets calls.

The source code is designed for simplicity and is comprised of only two files, `nxd BSD.c` and `nxd BSD.h`. Installation requires adding these two files to the build project (not the NetX library) and creating the host application which will use BSD Socket service calls. The `nxd BSD.h` file must also be included in your application source. Sample demo files for both IPv4 and IPv6 based applications are included with the distribution which is freely available with NetX Duo. Further details are available in the
help and Readme files bundled with the BSD-Compatible Socket API package.

The BSD-Compatible Sockets API supports the following BSD Sockets API calls:

```c
INT bsd_initialize (NX_IP *default_ip, NX_PACKET_POOL *default_pool, CHAR *bsd_memory_not_used);
INT getpeername( INT sockID, struct sockaddr *remoteAddress, INT *addressLength);
INT getsockname( INT sockID, struct sockaddr *localAddress, INT *addressLength);
INT recvfrom(INT sockID, CHAR *buffer, INT buffersize, INT flags, struct sockaddr *fromAddr, INT *fromAddrLen);
INT recv(INT sockID, VOID *rcvBuffer, INT bufferLength, INT flags);
INT sendto(INT sockID, CHAR *msg, INT msgLength, INT flags, struct sockaddr *destAddr, INT destAddrLen);
INT send(INT sockID, const CHAR *msg, INT msgLength, INT flags);
INT accept(INT sockID, struct sockaddr *ClientAddress, INT *addressLength);
INT listen(INT sockID, INT backlog);
INT bind (INT sockID, struct sockaddr *localAddress, INT addressLength);
INT connect(INT sockID, struct sockaddr *remoteAddress, INT addressLength);
INT socket( INT protocolFamily, INT type, INT protocol);
INT soc_close ( INT sockID);
INT select(INT nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
VOID FD_SET(INT fd, fd_set *fdset);
VOID FD_CLR(INT fd, fd_set *fdset);
INT FD_ISSET(INT fd, fd_set *fdset);
VOID FD_ZERO(fd_set *fdset);
```
# ASCII Character Codes in HEX

*most significant nibble*

<table>
<thead>
<tr>
<th></th>
<th>0_</th>
<th>1_</th>
<th>2_</th>
<th>3_</th>
<th>4_</th>
<th>5_</th>
<th>6_</th>
<th>7_</th>
</tr>
</thead>
<tbody>
<tr>
<td>_0</td>
<td>NUL</td>
<td>DLE</td>
<td>SP</td>
<td>0</td>
<td>@</td>
<td>P</td>
<td>'</td>
<td>p</td>
</tr>
<tr>
<td>_1</td>
<td>SOH</td>
<td>DC1</td>
<td>!</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
</tr>
<tr>
<td>_2</td>
<td>STX</td>
<td>DC2</td>
<td>&quot;</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
</tr>
<tr>
<td>_3</td>
<td>ETX</td>
<td>DC3</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
</tr>
<tr>
<td>_4</td>
<td>EOT</td>
<td>DC4</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
</tr>
<tr>
<td>_5</td>
<td>ENQ</td>
<td>NAK</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
</tr>
<tr>
<td>_6</td>
<td>ACK</td>
<td>SYN</td>
<td>&amp;</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
</tr>
<tr>
<td>_7</td>
<td>BEL</td>
<td>ETB</td>
<td>'</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
</tr>
<tr>
<td>_8</td>
<td>BS</td>
<td>CAN</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
</tr>
<tr>
<td>_9</td>
<td>HT</td>
<td>EM</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
</tr>
<tr>
<td>_A</td>
<td>LF</td>
<td>SUB</td>
<td>*</td>
<td>:</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
</tr>
<tr>
<td>_B</td>
<td>VT</td>
<td>ESC</td>
<td>+</td>
<td>;</td>
<td>K</td>
<td>[</td>
<td>K</td>
<td>}</td>
</tr>
<tr>
<td>_C</td>
<td>FF</td>
<td>FS</td>
<td>,</td>
<td>&lt;</td>
<td>L</td>
<td>\</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>_D</td>
<td>CR</td>
<td>GS</td>
<td>-</td>
<td>=</td>
<td>M</td>
<td>]</td>
<td>m</td>
<td>}</td>
</tr>
<tr>
<td>_E</td>
<td>SO</td>
<td>RS</td>
<td>.</td>
<td>&gt;</td>
<td>N</td>
<td>^</td>
<td>n</td>
<td>~</td>
</tr>
<tr>
<td>_F</td>
<td>SI</td>
<td>US</td>
<td>/</td>
<td>?</td>
<td>O</td>
<td>_</td>
<td>o</td>
<td>DEL</td>
</tr>
</tbody>
</table>

**least significant nibble**
Index

Symbols

_\_nx_\_arp_\_packet_\_deferred_\_receive\_ 45, 373, 374
_\_nx_\_ip_\_driver_\_deferred_\_processing\_ 46, 370
_\_nx_\_ip_\_packet_\_deferred_\_receive\_ 46, 373
_\_nx_\_ip_\_packet_\_receive\_ 45, 373, 374
_\_nx_\_ip_\_thread_\_entry\_ 44
_\_nx_\_rarp_\_packet_\_deferred_\_receive\_ 45, 373, 374
_\_nx_\_version_\_id\_ 37

Numerics

16-bit checksum that covers the IP header only 62
48-bit address support 69

A

accelerated software development process 20
accepting a TCP server connection 254
access functions 45
ACK
    returned 103
adding deferred packet logic to the NetX IP helper thread 374
adding static route 202
address resolution activities 44
Address Resolution Protocol (see ARP) in IPv4 69
address specifications
    broadcast 59
    multicast 59
    unicast 59
    all hosts address 81
allocating a packet from specified pool 208
allocating memory packets 49
ANSI C 17, 21
appending data to end of packet 212
application downloaded to target hardware 24
application interface calls 43
application source and link 27
application specific modifications 17
application threads 27, 42
ARP 27
    processing 69
ARP aging 74
        disabled 74
ARP cache 69, 70
ARP dynamic entries 70
ARP Enable 69
ARP enable service 70
ARP entry from dynamic ARP entry list 70
ARP entry setup 70
ARP information gathering
    disabling 31
ARP messages 71
    Ethernet destination address 72
    Ethernet source address 72
    frame type 72
    hardware size 72
    hardware type 72
    operation code 73
    protocol size 73
    protocol type 72
    sender Ethernet address 73
    sender IP address 73
    target Ethernet address 73
    target IP address 73
ARP packet processing 44
ARP packets
    format 73
ARP periodic processing in IPv4 44
ARP request information in the ARP cache 31
ARP request message 71
ARP requests 70, 71
ARP response 71
ARP response send 361
ARP response send request 362
ARP send 361
ARP send packet request 361
ARP static entries 70
ARP statistics and errors 74
array of internal ARP mapping data structures 70
ASCII character codes in HEX 420
asynchronous events 45
attach interface 364
attach interface request 364, 365
attaching network interface to IP instance 184
automatic invalidation of dynamic ARP entries 74

B
big endian 60, 74, 76, 79, 84, 85, 94
binding client TCP socket to TCP port 238
binding UDP socket to UDP port 324
black box 17
broadcast addresses 59
BSD-compatible socket API 18
building a NetX application 27
building a TCP header 102
building a valid NetX packet 372
bypassing changes to see if problem changes 28
byte swapping on little endian environments 29

C
C compilers 11
calculation of capacity of pool 56
callback function 45
calling thread’s context 43
causng IP instance to leave specified multicast group 152
characteristics of packet memory pool 57
checking status of an IP instance 188, 206
checksum 33, 43
checksum calculation 103
checksum logic
   disabling 33
checksum logic on IP packets sent
   disabling 32
checksum logic on received TCP packets
   disabling 33
checksum processing in lower-priority threads 43
Class D IP address 80
Class D multicast addresses 81
classes of IP addresses 58
client binding 105
client connection requests 102
commercial network stacks 17
compatibility with legacy NetX Ethernet drivers 65
compilation and link with NetX library 26
complex protocols 48
configuration 28
configuring socket’s transmit parameters 312
connecting a client TCP socket 240
connection events 96
connection management 49
connection request to a TCP server 96
connection service 105
connectionless protocols 59
connectionless sending and receiving of data 49
copying packet 210
CRC processing 33
create services 43
creating a packet pool in specified memory area 220
creating a static IP to hardware mapping in ARP cache 130
creating a TCP client or server socket 276
creating a UDP socket 332
creating an IP instance 160
creating IP instance with IP address of zero 75
creating IP instances 66
Customer Support Center 12

D

data encapsulation 49
data transfer between network members 84
datagram definition 61
datagrams larger than underlying network driver's MTU size 68
debug packet dumping 29
debugging 24
default packet pool 42, 66
deferred driver packet handling 29
deferred IP packet reception 44
deferred processing event 370
deferred processing queue 46
deferred receive packet handling 374
deferring interrupt processing 46
deleting a previously created IP instance 162
deleting a previously created packet pool 222
deleting a static IP to hardware mapping in the ARP cache 132
deleting a TCP socket 280
deleting a UDP socket 334
deleting all static ARP entries 128
deleting static route 204
deletion of an IP instance 357, 358
delivering packet to first suspended thread 104
demo_threadx.c 26
demonstration system 27
destination address of the packet 67
disabling checksum for the UDP socket 328
disabling checksum logic on received IP packets 32
disabling error checking 29
disabling IGMP information gathering 31
disabling IGMP loopback 144
disabling IGMP v2 support 31
disabling IP packet forwarding 168
disabling IP packet fragmenting 172
disabling link 357
disabling listening for client connection on TCP port 270
disabling NetX support on the 127.0.0.1 loopback interface 32
disabling raw packet sending/receiving 192
disabling reset processing during disconnect 33
disabling Reverse Address Resolution Protocol (RARP) 230
disabling the UDP checksum logic 87
disconnect callbacks 44
disconnect processing 100
disconnecting client and server socket connections 282, 284, 286, 310
disconnection services 105
driver deferred processing 370
driver entry 354
driver entry function 355
driver initialization 44, 66, 355
driver input 372
driver introduction 354
driver output 371
driver output function 359
driver request data structure 355
driver requests 355
duplex type request 367
dynamic ARP entries 70
dynamically mapping 32-bit IP addresses 69

E

ease of use 20
easy-to-use interface 20
embedded development on Windows or Linux 24
embedded network applications 19
enable link 357
enable services 43
enabling Address Resolution Protocol (ARP) 118
enabling checksum for the UDP socket 330
enabling ICMP processing 78
enabling IGMP loopback 146
enabling Internet Control Message Protocol (ICMP) component 134
enabling Internet Group Management Protocol (IGMP) component 140
enabling IP packet forwarding 170
enabling IP packet fragmenting 174
enabling listening for client connection on TCP port 258
enabling raw packet sending/receiving 194
enabling Reverse Address Resolution Protocol (RARP) 232
enabling static routing 32
enabling TCP component of NetX 246
enabling UDP component of NetX 316
ensuring driver supplies ARP and IP packets 28
entry point of internal IP thread 44
Ethernet 69
Ethernet ARP requests formats 72
examining default packet pool 28
examining NX_IP structure 28
external ping request 80
extracting data from packet via an offset 214
extracting IP and sending port from UDP datagram 350
extracting network parameters from UDP packet 322

fast response 19
fields of the IPv4 header 61
finding next available TCP port 248
finding next available UDP port 318
fixed-size memory blocks 51
fixed-size packet pools 51
flow control for data transfer 102
fragmentation 51
fragmented IP packets 64
freeing up processor cycles 16
functional components of NetX 39

gateway IPv4 address 59
getting allocation errors 369
getting duplex type 367
getting error count 367
getting length of packet data 218
getting link speed 366
getting link status 365
getting MSS of socket 292
getting MSS of socket peer 294
getting port number bound to client TCP socket 242
getting receive packet count 368
getting transmit packet count 369
global data structures 25
guide conventions 10

handling
periodic processing 66
handling connection and disconnection actions 104
handling deferred packet processing 66
head and tail pointers of the transmit queue 372
headers 49
headers in the TCP/IP implementation 60, 94
higher-level protocols 60
host system considerations 24

I/O 42
IBM-PC hosts 24
ICMP 49
ICMP header format 79
ICMP information gathering disabling 31
ICMP ping message format 79
ICMP ping processing 44
ICMP statistics and errors 78
ICMPv4 enable 78
ICMPv6 Services in NetX Duo 80
IGMP 49
IGMP enable 81
IGMP header 82
IGMP initialization 81
IGMP periodic processing 44
IGMP processing 44, 81
IGMP query messages 83
  format 83
IGMP report 81
IGMP report message 82
IGMP report message format 82
IGMP statistics and errors 84
image download to target 27
implemented as a C library 16
incoming IP packets 45
increased throughput 20
increasing stack size during the IP create 66
initial execution 42
initialization 42, 43, 94
  NetX system 27
  of driver 44
initializing NetX system 236
in-line processing 16
installation of ThreadX 26
instruction image requirements 16
interface and next hop address 95
interface control block 63
interface control block assigned to the packet 54
internal component function calls 16
internal IP send processing 359
internal IP thread 42, 44, 46, 64
internal IP thread calls 44
internal transmit sent queue 103
Internet Control Message Protocol (see ICMP) 77
Internet Group Management Protocol (see IGMP) 80, 379
Internet Protocol v4 57
interrupt service routine 45
invalidating all dynamic entries in ARP cache 114
IP address 58
IP address of the receiver or receivers 62
IP address of the sender 62
IP address structure 58
IP checksum 63
IP control block
  NX_IP 67
IP create call 44
IP create service 66
IP data structure 42
IP datagram 61
IP fragment assembly timeouts 44
IP fragment reassemble processing 45
IP fragmentation
  disabling 69
IP fragmentation information 61
IP header format 60
IP helper thread 66, 104, 356
IP information gathering
  disabling 32
IP instance 27, 66
  control blocks 67
  creation 42
IP instances 62
IP packet fragment assembly 44
IP packets 45, 46
IP periodic timers 45
IP receive 64
IP receive processing 64
IP resources 42
IP send 62
IP send function 43
IP statistics and errors 67
IP version 4 61
IP_ADDRESS 59
IPv4 16-bit identification 61
IPv4 address
  13-bit fragment offset 62
  16-bit checksum 62
  16-bit total length 61
  32-bit destination IP address 62

Express Logic
joined multicast groups 82
joining a multicast group 82
joining IP interface to specified multicast group 148
joining the specified multicast group 150

K
keeping track of statistics and errors 90

L
last packet within the same network packet 54
layering 49
least significant 32-bits of physical address 360, 362
least significant 32-bits of physical multicast address 363, 364
line speed request 366
link allocation error count request 369
link enable call 44
link error count request 368
LINK INITIALIZE request 356
link level 48
link receive packet count request 368
link status request 366
link transmit packet count request 369
linked-list manipulation 51
linked-list processing 51
listen callbacks 44
listening for packets with the Ethernet address 81
locating a physical hardware address given an IP address 122
locating an IP address given a physical address 126
logical connection point in the TCP protocol 95
logical loopback interface 48
long-word boundary 55
low packet overhead path 87
lowest layer protocol 48

M
maintaining relationship between IP address and physical hardware address 70
management
Internet Control Message Protocol (ICMP) 378
Internet Protocol (IP) 379
Reverse Address Resolution Protocol (RARP) 381
Transmission Control Protocol (TCP) 382
management-type protocols 49
managing the flow of data 49
maximum number of ARP retries without ARP response 30
maximum number of entries in routing table 32
maximum number of multicast groups that can be joined 31
Maximum Transmission Unit (MTU) 355
memory areas
NetX objects 52
Index

Index

ThreadX 52
microprocessors 19
minimizing dropped packets 374
minimizing ISR processing 374
most significant 32-bits of physical address 360, 362
most significant 32-bits of physical multicast address 363, 364
multicast addresses 59
multicast group 80
multicast group join 81, 363
multicast group leave 364
multicast groups on the primary network 82
multicast IP addresses 81
multicast routers 83
multihome devices 95
multihome hosts 45, 77
multihome support service
  nx_igmp_multicast_interface_join 48
  nx_ip_interface_address_get 48
  nx_ip_interface_address_set 48
  nx_ip_interface_attach 48
  nx_ip_interface_info_get 48
  nx_ip_interface_status_check 48
  nx_ip_raw_packet_interface_send 48
multiple linked lists 70
multiple network interfaces 63
multiple physical network interfaces 46
multiple pools of fixed-size network packets 49
multiple thread suspension 56

N

network data packets 49
network driver 17, 42, 45, 46, 68
network driver entry function 354
network driver’s entry routine 354
network hardware 20
network layer 49
network stack 17
NetX ARP software 74
NetX benefits 19
  application migration path 21
development investment protection 21
development process 20
high-speed Internet connectivity 19
improved responsiveness 19
integrated with ThreadX 19, 21
network traffic 20
NetX architecture easy to use 20
new processor architecture 21
processing requirements on a single packet 20
processor-independent interface 20
protecting software investment 21
small memory requirements 19
ThreadX supported processors 21
NetX callback functions 44
NetX constants 387
  alphabetic listings 388
NetX data structures 27
NetX data types 407
NetX error checking API removal 29
NetX IGMP software 84
NetX IP send routine 17
NetX IP software 67
NetX packet management software 56
NetX physical media drivers 353
NetX protocol stack 19, 25
NetX RARP software 77
NetX runtime library 27
NetX services 27, 107, 377
NetX source code 24
NetX system initialization 27
NetX unique features 16
NetX Version ID 37
new application threads 27
next packet within same network packet 54
notifying application if IP address changes 154
notifying application of each received packet 342
notifying application of received packets 302
notifying application of window size updates 314
number of 32-bit words in the IP header 61

Express Logic
number of bytes in entire network packet 54
number of bytes in the memory area 56
number of keepalive retries before connection is broken 35
number of packets queued while waiting for an ARP response 30
number of routers this datagram can pass 62
number of seconds ARP entries remain valid 30
number of seconds between ARP retries 30
number of ThreadX timer ticks in one second 32
nx_api.h 26, 27, 29, 30, 31, 34, 35, 36, 37, 57, 59, 67, 91, 106
NX_ARP_DISABLE_AUTO_ARP_ENTRY 30
nx_arp_dynamic_entries_invalidate 114
nx_arp_dynamic_entry_set 116
nx_arp_enable 69, 118
NX_ARP_EXPIRATION_RATE 30, 74
nx_arp.hardware_address_find 122
nx_arp.info_get 75, 124
nx_arp.ip_address_find 126
NX_ARP_MAX_QUEUE_DEPTH 30, 63
NX_ARP_MAXIMUM_RETRIES 30, 71
nx_arp.static_entries_delete 128
nx_arp.static_entry_create 70, 130
nx_arp.static_entry_delete 132
NX_ARP_UPDATE_RATE 30, 71
NX_DEBUG 28
NX_DEBUG_PACKET 29
NX_DISABLE_ARP_INFO 31
NX_DISABLE_ERROR_CHECKING 29
NX_DISABLE_FRAGMENTATION 31, 69
NX_DISABLE_ICMP_INFO 31
NX_DISABLE_IGMP_INFO 31
NX_DISABLE_IGMPV2 31
NX_DISABLE_IP_INFO 32
NX_DISABLE_IP_RX_CHECKSUM 32
NX_DISABLE_IP_TX_CHECKSUM 32
NX_DISABLE_LOOPBACK_INTERFACE 32, 48
NX_DISABLE_PACKET_INFO 33
NX_DISABLE_RARP_INFO 33
NX_DISABLE_RESET_DISCONNECT 33
NX_DISABLE_RX_SIZE_CHECKING 32
NX_DISABLE_TCP_INFO 33
NX_DISABLE_TCP_RX_CHECKSUM 33
NX_DISABLE_TCP_TX_CHECKSUM 33
NX_DISABLE_UDP_INFO 33, 35, 36, 37
NX_DRIVER_DEFERRED_PROCESSING 29, 374
NX_ENABLE_IP_STATIC_ROUTING 32, 68
nx_icmp_enable 78, 134
nx_icmp.info_get 78, 136
nx_icmp.ping 138
nx_igmp_enable 81, 140
nx_igmp.info_get 84, 142
nx_igmp.loopback_disable 144
nx_igmp.loopback.enable 146
nx_igmp.multicast 363
nx_igmp.multicast_interface_join 148
nx_igmp.multicast_join 81, 82
nx_igmp.multicast.leave 82, 152, 364
NX_INCLUDE_USER_DEFINE_FILE 28
NX_INTERFACE 370
nx.interface.additional_link_info 370
nx.interface.link_up 365
nx.ip.address.change_notify 154
nx.ip.address.get 154, 156
nx.ip.address.set 158
nx.ip.create 44, 45, 47, 62, 66, 69, 75, 354, 356
nx.ip.delete 162, 357, 358
nx.ip.driver.direct_command 366, 367, 368, 369, 370
nx.ip.driver.link_up 357
<table>
<thead>
<tr>
<th>Function/Variable</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx_ip_driver_packet</td>
<td>360, 361, 362</td>
</tr>
<tr>
<td>nx_ip_driver_physical_address_lsw</td>
<td>360, 361, 362, 363, 364</td>
</tr>
<tr>
<td>nx_ip_driver_physical_address_msw</td>
<td>360, 361, 362, 363, 364</td>
</tr>
<tr>
<td>nx_ip_driver_return_ptr</td>
<td>366, 367, 368, 369, 370, 371</td>
</tr>
<tr>
<td>nx_ip_fragment_enable</td>
<td>170</td>
</tr>
<tr>
<td>nx_ip_fragment_disable</td>
<td>172</td>
</tr>
<tr>
<td>nx_ip_fragment_enable</td>
<td>174</td>
</tr>
<tr>
<td>nx_ip_gateway_address_set</td>
<td>176</td>
</tr>
<tr>
<td>nx_ip_info_get</td>
<td>67, 178</td>
</tr>
<tr>
<td>nx_ip_interface</td>
<td>54</td>
</tr>
<tr>
<td>nx_ip_interface_address_get</td>
<td>180</td>
</tr>
<tr>
<td>nx_ip_interface_address_set</td>
<td>182</td>
</tr>
<tr>
<td>nx_ip_interface_attach</td>
<td>47, 364, 365</td>
</tr>
<tr>
<td>nx_ip_interface_status_check</td>
<td>43, 77, 365</td>
</tr>
<tr>
<td>NX_IP_PERIODIC_RATE</td>
<td>32, 34, 35</td>
</tr>
<tr>
<td>nx_ip_raw_packet_disable</td>
<td>192</td>
</tr>
<tr>
<td>nx_ip_raw_packet_enable</td>
<td>194</td>
</tr>
<tr>
<td>nx_ip_raw_packet_enabled</td>
<td>65</td>
</tr>
<tr>
<td>nx_ip_raw_packet_receive</td>
<td>65, 196</td>
</tr>
<tr>
<td>nx_ip_raw_packet_send</td>
<td>200</td>
</tr>
<tr>
<td>NX_IP_ROUTING_TABLE_SIZE</td>
<td>32</td>
</tr>
<tr>
<td>nx_ip_socket_send</td>
<td>87</td>
</tr>
<tr>
<td>nx_ip_static_route_add</td>
<td>68</td>
</tr>
<tr>
<td>nx_ip_static_route_delete</td>
<td>68, 204</td>
</tr>
<tr>
<td>nx_ip_status_check</td>
<td>43, 77, 206, 366</td>
</tr>
<tr>
<td>NX_LINK_ARP_RESPONSE_SEND</td>
<td>362</td>
</tr>
<tr>
<td>NX_LINK_ARPSEND</td>
<td>361</td>
</tr>
<tr>
<td>NX_LINK_DISABLE</td>
<td>358</td>
</tr>
<tr>
<td>NX_LINK_ENABLE</td>
<td>357</td>
</tr>
<tr>
<td>NX_LINK_GET_ALLOC_ERRORS</td>
<td>369</td>
</tr>
<tr>
<td>NX_LINK_GET_DUPLEX_TYPE</td>
<td>367</td>
</tr>
<tr>
<td>NX_LINK_GET_ERROR_COUNT</td>
<td>368</td>
</tr>
<tr>
<td>NX_LINK_GET_RX_COUNT</td>
<td>368</td>
</tr>
<tr>
<td>NX_LINK_GET_SPEED</td>
<td>366</td>
</tr>
<tr>
<td>NX_LINK_GET_STATUS</td>
<td>366</td>
</tr>
<tr>
<td>NX_LINK_GET_TX_COUNT</td>
<td>369</td>
</tr>
</tbody>
</table>

**Index**
nx_tcp_client_socket_port_get 242
nx_tcp_client_socket_unbind 98, 244
nx_tcp_enable 94, 246
NX_TCP_ENABLE_KEEPALIVE 34, 36
NX_TCP_ENABLE_WINDOW_SCALING 35
NX_TCP_FAST_TIMER_RATE 35
nx_tcp_free_port_find 248
NX_TCP_IMMEDIATE_ACK 34, 35
nx_tcp_info_get 105, 250
NX_TCP_KEEPALIVE_INITIAL 35
NX_TCP_KEEPALIVE_RETRIES 35
NX_TCP_KEEPALIVE_RETRY 35
NX_TCP_MAXIMUM_RETRIES 36
NX_TCP_MAXIMUM_TX_QUEUE 36
NX_TCP_RETRY_SHIFT 36
nx_tcp_server_socket_accept 99, 101, 254
nx_tcp_server_socket_listen 99, 102, 258
nx_tcp_server_socket_relisten 99, 101, 262
nx_tcp_server_socket_unaccept 100, 101, 266
nx_tcp_server_socket_unlisten 102, 270
NX_TCP_SOCKET 106
nx_tcp_socket_bytes_available 274
nx_tcp_socket_create 94, 96, 99, 276
nx_tcp_socket_delete 98, 280
nx_tcp_socket_disconnect 96, 100, 101, 282, 286, 288
nx_tcp_socket_info_get 105, 286
nx_tcp_socket_mss_get 292
nx_tcp_socket_mss_peer_get 294
nx_tcp_socket_mss_set 296
nx_tcp_socket_peer_info_get 298
nx_tcp_socket_receive 300
nx_tcp_socket_receive_notify 104, 302
nx_tcp_socket_send 102, 304
nx_tcp_socket_state_wait 308
nx_tcp_socket_transmit_configure 310
nx_tcp_socket_window_update_notify 314
NX_TCP_TRANSMIT_TIMER_RATE 36
nx_tcp.h 34, 35, 36
nx_udp_enable 86, 316
nx_udp_free_port_find 318

nx_udp_info_get 90, 320
nx_udp_packet_info_extract 322
NX_UDP_SOCKET 91
nx_udp_socket_bind 324
nx_udp_socket_bytes_available 326
nx_udp_socket_checksum_disable 87, 328
nx_udp_socket_checksum_enable 330
nx_udp_socket_create 86, 332
nx_udp_socket_delete 334
nx_udp_socket_info_get 90, 336
nx_udp_socket_interface_send 338
nx_udp_socket_port_get 338
nx_udp_socket_receive_notify 89
nx_udp_socket_receive 43, 89, 342
nx_udp_socket_receive_notify 342
nx_udp_socket_send 17, 87, 348
nx_udp_socket_unbind 348
nx_udp_source_extract 350
NX_UNHANDLED_COMMAND 371
nx_user.h 28
nx.a (or nx.lib) 26, 27
nx.duo.lib 26, 27
nxd_ip_raw_packet_send 65
nxd_tcp_client_socket_connect 96
nxd_udp_socket_extract 197
nxd_udp_socket_send 43, 88

O

optimal packet payload size 51
outgoing fragmentation 63
overwriting memory
  IP helper thread 66

P

packet allocation 51
packet broadcast 360
packet broadcast request 360
packet deallocation 51
packet destination IP address 47
packet header and packet pool layout 53
packet memory pool 52
packet memory pools 49
Index

packet pool control block 57
  NX_PACKET_POOL 57
packet pool control blocks 57
packet pool creation 42
packet pool information gathering
  disabling 33
packet pool memory area 52
packet pools 51
packet reception 45
packet send processing 359
packet size 56
packet transmission 45, 52
packet transmission completion 45
packet_ptr 374
packet-receive processing 17
packets requiring IP address resolution 63
partitioning network aspect 20
passing error and control information
  between IP network members 77
payload size 54, 56
payload size for packets in pool 57
performance advantages 16
periodic RARP request 76
physical address mapping in IPv4 81
physical address mapping in IPv6 81
physical Ethernet addresses 81
physical layer header removed 373
physical media 69
physical packet header size 29
picking up port number bound to UDP socket 338
Piconet™ architecture 16
ping request 79
ping response 79
placing a raw packet on an IP instance 29
placing packets with receive data on TCP socket receive queue 104
pointer to the destination to place the allocation error count 370
pointer to the destination to place the duplex type 367
pointer to the destination to place the error count 368
pointer to the destination to place the line speed 366
pointer to the destination to place the receive packet count 368
pointer to the destination to place the status 366
pointer to the destination to place the transmit packet count 369
Pointer to the packet to send 361
pointer to the packet to send 360, 362
pointer to the physical network interface 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371
points to the end of the data currently in the packet payload area 55
points to the location of where packet data is added 55
pool capacity 56
pool statistics and errors 56
portability 11, 17
pre-defined multicast addresses 81
preemption 44
prepend pointer 63, 64
prevention of stalling network requests 43
primary interface 47
print debug information 28
priority and stack size of internal IP thread 44
processing needs 25
processing packet and periodic requests 66
processor isolation 20
processors
  DSP 16
  RISC 16
product distribution 25
product release by name and the product major and minor version 37
program execution overview 42
protocol layering 50
protocol using the IP datagram 62
public domain network stacks 18
Q
queued client connection request packets 102

R
RAM driver example 374
RARP enable 75
RARP information gathering disabling 33
RARP reply 76
RARP reply messages 76
RARP reply packet 76
RARP request 76
RARP request packet format 76
RARP send 362
RARP send request 362
RARP statistics and errors 77
raw IP packet processing 65
raw IP packets 65
raw IP receive 65
raw IP send 65
readme_netx_duo_generic.txt 24, 27, 28, 37
readme_threadx.txt 25
ready-to-execute mode 44
real-time applications 19
real-time network software 19
receive functions 374
receive packet callback 89
receive packet callback function 104
receive packet dispatched 373
receive packet processing 374
received packet 43
receiving a raw IP packet 198
receiving data from a TCP socket 300
receiving datagram from UDP socket 340
recompiling NetX library with debug options 27
recompiling the NetX library 27
releasing a previously allocated packet 226
releasing a transmitted packet 228
reliable data path 49
re-listening for client connection on TCP port 262
removing association between server socket and server port 266
retransmit timeout period changes between retries 36
retrieving data from packet 216
retrieving information
ARP activities 124
ICMP activities 136
IGMP activities 142
retrieving information about IP activities 178
retrieving information about packet pool 224
retrieving information about peer TCP socket 298
retrieving information about RARP activities 234
retrieving information about TCP activities 250
retrieving information about TCP socket activities 288
retrieving information about UDP activities 320
retrieving information about UDP socket activities 336
retrieving interface IP address 180
retrieving network interface parameters 186
retrieving number of bytes available for retrieval 274, 326
retrieving the IP address and network mask 156
Reverse Address Resolution Protocol (see RARP) in IPv4 75
RFC 1112 80
768 84
793 91
826 69
RFC 791 57
RFC 903 75
RFCs Supported by NetX
RFC 1112 18
RFC 2236 18
RFC 768 18
RFC 791 18
RFC 792 18
RFC 793 18
RFC 826 18
RFC 903 18
RFCs supported by NetX
  basic network protocols 18
runtime image 16

S

scaling 16
seconds between retries of the keepalive
timer 35
seconds of inactivity before the keepalive
timer activates
defining 35
send packet request 360, 362
sending a raw IP packet 200
sending a UDP packet 84
sending and receiving of data 49
sending and receiving simple packets 49
sending and receiving UDP packets 86
sending data through a TCP socket 304
sending datagram through UDP socket 346
sending gratuitous ARP request 120
sending or receiving UDP data 87
sending ping request to specified IP
address 138
sending raw IP packet out specified
network interface 196
sending request to unmapped IP address
70
server listen requests
defining 34
service call data type 11
  CHAR 11
  UINT 11
  ULONG 11
  VOID 11
service call interface 11, 20
setting dynamic ARP entry 116
setting Gateway IP address 176
setting interface IP address and network
mask 182
setting MSS of socket 296
setting the IP address and network mask
158
setup and data transfer phase of a
connection 102
size of
  NetX 16
socket receive function 89
socket receive queue 89
socket transmit queue 100
socket waiting for a connection 102
sockets 27
software maintenance 20
source code
  ANSI C 17
  ASCII format 24
specification of IP addresses 59
stack sizes 27
start of the physical payload area 54
static ARP mapping 70
static IPv4 routing 67
static routing table 67, 68
statistics 56
  free packets in pool 57
  invalid packet releases 57
  pool empty allocation requests 57
  pool empty allocation suspensions 57
  TCP socket bytes received 105
  TCP socket bytes sent 105
  TCP socket checksum errors 105
  TCP socket packet retransmits 105
  TCP socket packets queued 105
  TCP socket packets received 105
  TCP socket packets sent 105
  TCP socket receive window size 105
  TCP socket state 105
  TCP socket transmit queue depth 105
  TCP socket transmit window size 105
  total ARP aged entries 75
  total ARP dynamic entries 75
  total ARP invalid messages 75
  total ARP requests received 75
  total ARP requests sent 75
  total ARP responses received 75
total ARP responses sent  75
total ARP static entries  75
total ICMP checksum errors  78
total ICMP ping responses received  78
total ICMP ping threads suspended  78
total ICMP ping timeouts  78
total ICMP pings sent  78
total ICMP unhandled messages  78
total IGMP checksum errors  84
total IGMP current groups joined  84
total IGMP queries received  84
total IGMP reports sent  84
total IP bytes received  67
total IP bytes sent  67
total IP fragments received  67
total IP fragments sent  67
total IP invalid packets  67
total IP packets received  67
total IP packets sent  67
total IP receive checksum errors  67
total IP receive packets dropped  67
total IP send packets dropped  67
total packet allocations  57
total packets in pool  57
total RARP invalid messages  77
total RARP requests sent  77
total RARP responses received  77
total TCP bytes received  105
total TCP bytes sent  105
total TCP connections  105
total TCP connections dropped  105
total TCP disconnections  105
total TCP invalid packets  105
total TCP packet retransmits  105
total TCP packets received  105
total TCP packets sent  105
total TCP receive checksum errors  105
total TCP receive packets dropped  105
UDP receive checksum Errors  90
UDP receive packets dropped  90
UDP socket bytes received  90
UDP socket bytes sent  90
UDP socket checksum errors  90
UDP socket packets queued  90
UDP socket packets received  90
UDP socket packets sent  90
UDP socket receive packets dropped  90
status and control requests  45
status changes  45
stop listening on a server port  102
stream data transfer between two network members  91
suspend while attempting to receive a UDP packet  90
system configuration options  28
system initialization  42
system management  382
system tic division to calculate
fast TCP timer rate  35
timer rate for TCP transmit retry processing  36
timer rate for TCP-delayed ACK processing  34

T

target address space  52
target considerations  25
target RAM  25
target ROM  25
TCP  49
TCP checksum  94
TCP checksum logic  33
TCP client connection  96
TCP client disconnection  96
TCP disconnect protocol  98,  100
TCP enable  94
TCP for data transfer  95
TCP header  91
16-bit destination port number  92
16-bit source port number  92
16-bit TCP checksum  94
Index

16-bit urgent pointer 94
16-bit window 93
32-bit acknowledgement number 93
32-bit sequence number 92
4-bit header length 93
6-bit code bits 93
TCP header control bits 93
TCP header format 91, 92
TCP immediate ACK response processing enabling 35
TCP information gathering disabling 33
TCP keepalive timer enabling 34
TCP output queue 52
TCP packet queue processing 44
TCP packet receive 104
TCP packet retransmit 103
TCP packet send 102
TCP packets to receive before sending an ACK 34
TCP periodic processing 44
TCP receive notify 104
TCP receive packet processing 104
TCP retransmission timeout 45
TCP server connection 99
TCP server disconnection 100
TCP socket control block

NX_TCP_SOCKET 106
TCP socket create 91
TCP socket state machine 96
TCP socket statistics and errors 105
TCP sockets

number of in application 94
TCP transmit queue depth before suspended or rejected TCP send request 36
TCP window size 102
thread protection 25
thread stack and priority 66
thread stack requirements 25
thread suspension 56, 66, 80, 90, 105
ThreadX 11, 19, 42
distribution contents 25
ThreadX context switches 17

ThreadX mutex object 25
ThreadX periodic timers 45
ThreadX RTOS 43
ThreadX support 17
ThreadX timer 25
ThreadX_Express_Startup.pdf 25
time constraints on network applications 19
time-to-market improvement 20
total length of the IP datagram in bytes— including the IP header 61
total number of physical network interfaces on the device 29
Transmission Control Protocol (TCP) 91
transmit acknowledge processing 104
transmit retries allowed before connection is broken 36
transmitting packets 359
transport layer 49
troubleshooting 27
tx_api.h 25, 26
tx_application_define 27, 42, 43
tx_port.h 11, 26
txA 26
tx.lib 26
type of ICMP message

ping request 78
ping response 80
type of service requested for this IP packet 61

U

UDP 49
UDP checksum 17, 86
UDP checksum calculation 43, 87
UDP data encapsulation 50
UDP enable 86
UDP Fast Path 87
UDP Fast Path Technology 17
UDP Fast Path technology 88
UDP header 85

16-bit destination port number 85
16-bit source port number 85
16-bit UDP checksum 86
16-bit UDP length 86
UDP header format 85
UDP information gathering
disabling 37
UDP packet delivery to multiple network members 80
UDP packet receive 89
UDP packet reception 88
UDP packet send 88
UDP packet transmission 84
UDP ports and binding 87
UDP receive notify 89
UDP receive packet processing 89
UDP socket 17, 87
UDP socket characteristics 91
UDP socket checksum logic 87
UDP socket control block
   TX_UDP_SOCKET 91
UDP socket create 89
UDP socket receive queue 17
UDP socket statistics and errors 90
UDP socket's receive queue 89
UDP utilization of IP protocol for sending and receiving packets 85
unbinding a TCP client socket from a TCP port 244
unbinding UDP socket from UDP port 348
unicast addresses 59
unimplemented commands 371
unique 32-bit Internet address 58
Unix host 24
upper 13-bits of the fragment offset 62
upper layer protocol services 47
user command request 371
user-defined pointer 371
using deferred packet handling 374
using NetX 26

V

version history 37

W

waiting for TCP socket to enter specific state 308

window scaling option for TCP applications
   enabling 34
   window size 102
   window size adjusted dynamically 102

Z

zero copy implementation 16