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

Network Address Translation (NAT)

User Guide

Express Logic, Inc.
858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259
www.expresslogic.com

Part Number: 000-1054
Revision 5.1

Contents

Chapter 1

Introduction to NetX NAT

The need for IP address translation arises when a network's internal (private) IP addresses cannot be used outside the network (e.g. the public Internet) either for privacy reasons or because they are invalid for use outside the network. Network topology outside local domain can change in many ways. Customers may change providers, company backbones may be reorganized, or providers may merge or split. Whenever external topology changes with time, address assignment for hosts within the local domain must also change to reflect the external changes. Changes of this type can be hidden from users within the domain by centralizing changes to a single address translation router. Therein lays the great benefit of network address translation.

Basic address translation in many cases allows hosts in a private network to transparently access the external network and enable access to selective local hosts from the outside. Organizations with a network setup predominantly for internal use, with a need for occasional external access are good candidates for this scheme.

The Network Address Translation (NAT) protocol is designed to provide a means for hosts on internal networks to share globally registered IP addresses for sending packets out onto the external network. It was originally created as a workaround to the IP address depletion problem.

The NAT server acts as a router. While administrating address translations it can also monitor traffic between private and external hosts. It picks up outbound packets from local hosts on its private interface, updates them for the public domain, and forwards these packets out on its external interface. Specifically, it modifies the packet IP header, replacing the private source IP address:port with a globally registered IP address:port. When the target host replies back to the local host, the NAT server knows to replace the inbound packet IP header’s destination IP address:port with the local host’s original private IP address:port before sending the packet out on its internal interface. To keep track of global and private address translations for all active connections between local and external hosts, the NAT server maintains a translation table with information about each connection including IP addresses and port number translation.

NetX NAT Packet Processing and the Translation Table

NetX NAT resides on a NetX enabled router. NetX is configured to forward each packet it receives on both internal and external networks to the NAT server thread running simultaneously on the same device. When the NAT server is done with the packet, it indicates to NetX if it consumed (attempted to forward) the packet. If it did not consume the packet, NetX will process the packet as if it was intended for the NAT router itself.

Connections across the NAT boundary are typically initiated by the private host sending outbound packets from the private network to an external host. However it is also possible to have connections initiated in the opposite direction if the private network has 'servers' e.g. HTTP or FTP servers that will accept packets from the external network. To do so, NAT must be configured to assign these local hosts static (permanent) IP address:ports in its translation table.

For each packet forwarded to it from NetX, the NAT server determines if the packet is inbound or outbound. For inbound packets, the NAT server checks the packet IP header source IP address and port if appropriate for the packet protocol. If the translation table does not contain an entry for a packet previously sent by this host for the same destination, it will create a new entry which will contain a unique global source IP address:port for that packet, and modify the packet IP header with this new IP address:port before sending it onto the external network. For inbound packets, the NAT server looks for a previous entry in its translation table with a local host with a global IP address:port matching the packet destination IP address:port. If no match is found, it will discard the packet unless the destination IP address:port is the global address for a local host acting as a server. That situation is described in greater detail further in this section. If it does find a match, it will replace the packet's global destination IP address:port with the private IP address:port in the matching translation table entry and then send the packet onto the local network to the intended private host.

In this manner a local host can exchange packets with an external host without the external host ever contacting the local host directly or knowing the local host private address.

Type	Protocol	Local Host IP	Local Port	Global Host IP	Global Port	Target Host IP	Target Port	Time Remaining
S	UDP	10.0.0.3	123	192.2.2.89	123	0	0	0
D	TCP	10.0.0.2	1031	192.2.2.89	4031	64.23.56.185	21	120
D	ICP	10.0.0.2	1032	192.2.2.89	4032	64.23.56.185	20	120
D	ICP	10.0.0.2	200	192.2.2.89	3200	63.105.88.135	67	30

Figure 1. Example Translation Table

In the example translation table in Figure 1, NAT has created one static entry that is an SNTP server for the local network for which will allow any inbound UDP packet destined for a local host with port 123. Before forwarding the packet to the local host on its internal network, the NAT server changes the target address from 192.2.2.89 to 10.0.0.03 (the port number in this case is not changed). Entries for local hosts acting as servers are usually added as permanent entries. Permanent entries are indicated by having Type set to "S" (static). Note that the time remaining is set to 0. The NAT server can therefore avoid accidentally deleting server entries from the table. Static entries can only be created by the NAT host application, and only when NetX NAT is not in operation.

The remaining entries in Figure 1 are dynamically created entries, all for local host 10.0.0.2 connecting with two different external hosts. When the NAT server receives either of the two TCP packets from this local host, it changes the packet source IP address:port from 10.0.0.2 to 192.2.2.89 and the port number from 1031 and 1032 to 4031 and 4032 respectively. The choice of port translation in this case was made using NetX NAT's default set port callback. For more information about this and other NetX NAT callbacks see the **NetX NAT API Callbacks** section.

The last entry has the query ID in its ICMP changed from 200 to 3200. When NAT receives an inbound packet with target address 192.2.2.89 in response to these outbound packets, it will be able to identify which host the packet is bound for, and change the target IP address and port to the corresponding local host.

Whenever NAT changes data in a packet IP header and/or subprotocol header (TCP, UDP) it must update that header's checksum with a new checksum computation. An optimized checksum computation can be applied instead of computing the entire checksum because usually only a small amount of data is changed. This is a configurable NAT option and described more fully in the **Other NetX NAT Features** section. The specific configuration options are described in the **NetX NAT Configuration Options** section.

When NetX NAT creates a dynamic entry for the translation table, it also sets an expiration timeout. Each time a packet matching this entry is received by NAT, the timeout is refreshed. Otherwise, the entry timeout eventually expires. The NAT server has a background periodic task that continually searches the table for expired translation table timeouts and when it can acquire an exclusive lock on the translation table, removes that entry. Setting an entry timeout is a user configurable callback function and typically is protocol dependent. See the **NetX NAT API**

Callbacks section for more information about this API callback. If the host NAT application does not specify such a callback when NAT is created, NetX NAT applies a set of default time outs recommended by RFCs 1631, 3022 and 3235.

Most NAT solutions allow the host application to specify 'rules' limiting network traffic. NetX NAT provides this option using protocol specific packet filters for TCP, UDP and ICMP packets which it applies when it receives inbound and outbound packets. The NAT default filters allow all outbound packets to be forwarded to the external network, and all inbound packets to be forwarded onto the private network. However the intention of these filters is to give the host NAT application the opportunity to apply its own rules. For more details on these callback filter functions, see the section on **NetX NAT API Callbacks**.

Traditional NAT and Variations on Address Translation

With basic NAT, the NAT server 'owns' multiple globally registered IP addresses different from its own IP address which it can assign to local hosts either statically or temporarily. The NAT server's ARP table must be configured for the IP layer to trap all packets with these global destination IP addresses so that it can perform address translation back to the local host private IP address and forward the packet to that local host.

NAPT, network address port translation, is a variation of basic NAT, where network address translation is extended to include a transport identifier. Most typically this is the port number for TCP and UDP packets, and the Query ID for ICMP request packets. NAPT is one of the most common implementations of NAT. Typically a small network, either home or office, shares its own IP address with all the local hosts on the private network. No modification to the NAT server ARP table is necessary since it will of course receive all packets from external hosts intended for a local host, since they share the same global IP address.

NetX NAT will eventually accommodate both NAT and NAPT. However the current release implements NAPT for ease of configuration with NetX. This avoids having to interfere with the native NetX management of its own ARP table. This arrangement still permits servers to exist on the local network. It is important to note that NAPT deployments are based on the assumption of a client-server application model, with the clients in the private realm.

Application Specific Gateways and NAT

Network address translation is application independent. However, often IP address and port information is included in the application data and must also be translated to work with NAT. Application Specific (or Layer) Gateways (ALGs) are functions that perform payload monitoring and alterations so that packets sent by a certain applications can work with NAT. Applications that require ALG intervention must not have their payload encoded, as doing that would effectively disable an ALG to make the packet payload work after address translation, unless the ALG has the key to decrypt the payload.

The NAT solution has the disadvantage of taking away end-to-end usage of an IP address. NAT can partially compensate by creating an increased state in the network which is saved with the translation entry. For example, when NAT modifies the FTP PORT command with address:port translation, the packet payload and often packet length change. This causes NAT to have to track offsets in TCP packet sequence and acknowledgment numbers. NetX NAT internally handles this situation so there is no need for a host ALG for FTP. But the example demonstrates that one of the major difficulties with NAT: it is not strictly limited to the IP layer of network processing and must often extend into the application layer. This can make NAT difficult to debug and cause applications that send packets across the NAT boundary which don't have an ALG written for them to fail.

In NetX NAT, it is up to the host application to write its own ALG that fall into this category. It can build these ALGs into the user defined packet filters described in the **NetX NAT API Callback** section.

NetX NAT Requirements and Constraints

The NetX NAT API requires ThreadX 5.1 or later, and NetX 5.1 or later.

IP Thread Task Requirements

The NetX NAT API requires a creation of two NetX IP instances, one for the internal and external interface each.

- ARP must be enabled on each interface with the *nx_arp_enable* call prior to using the NetX NAT API.

Packet Pool Requirements

NAT also needs its own packet pool for sending its own ICMP error messages. The size of the NAT packet pool in terms of packet payload and number of packets available is user configurable, and depends on the anticipated volume of ICMP messages NAT will need to send when it is unable to process a packet.

- The packet pool used for receiving packets on either private or public interface is handled by the network driver assigned during the *nx_ip_create* call.

- If NAT attempts to forward a packet but is unable to do so, it will release the packet back to the packet pool from which it was allocated. If NAT does not consume a packet, it is left to NetX to release the packet.

- To display packet pool reserves for both the NAT packet pool and the driver packet pool during NAT operation, NetX NAT has a diagnostic utility, *_nx_nat_utility_display_packetpool_reserves,* for troubleshooting purposes. See the section on **Description of NAT Services** for more details.

Other Requirements

The NetX NAT requires the creation of a translation table to create the NAT instance itself. The size of this table depends on the anticipated number of translation tables at peak traffic periods. There is no logical limit on the size other than stack allocation and dynamic memory resources, plus the larger the table the longer each table search takes.

The NAT translation table ***must*** be created with IP address overloading enabled. This is because the current NetX NAT release is designed for the NAT server to share its IP address with its local hosts as their global source IP address.

The NetX NAT API has a requirement for a ThreadX byte pool. It uses this byte pool for dynamic translation table entry creation. It also uses it for creating packet fragment queues to receive fragmented packets. Even if NAT is not configured to either accept packet fragments or dynamic translation table creation, this byte pool is needed for TCP and UDP port assignment where NAT creates a temporary socket to reserve the port number in the native NetX port tables, so that NAT does not accidentally step on a port number being used by NetX itself. It is not unusual for the NAT server to also act as an HTTP server for example on a local host. So care must be taken to reserve port numbers accordingly.

Constraints

The current release of NAT is limited to supporting TCP, UDP and ICMP (IGMP is not supported). If the NAT host application plans to allow these subprotocols, it must accordingly enable NetX support through the *nx_tcp_enable, nx_udp_enable* and *nx_icmp_enable* calls respectively.

NetX NAT does not support IP6 addressing.

NetX NAT does not include a DNS or DHCP services, although the NAT host application can take integrate those services with NAT operation.

There can be no overlap between the private and global network domains. The *nx_nat_create* service call will check for overlap and return an error status if this condition is detected.

NetX NAT API Callbacks

The NetX NAT API has a default service for each callback function if the host application does not define its own. Example callback functions can be found in the demo files on the installation CD**.**

nx_nat_get_available_port

This function finds an unused port to use with a local host global source IP address to create a unique global source IP address:port combination. The port must not be in use in the translation table or in the NAT server's own port tables. This function takes as its arguments packet protocol, global source IP address which in this case is always NAT's global IP address, and private source port (or for ICMP packets the query ID). It then applies an algorithm to come up with an available global source port. The NAT server then insures that NetX itself does not accidentally attempt to claim the same port by updating the native NetX port tables using NetX services *_nx_udp_socket_bind* or *_nx_tcp_client_socket_bind*.

If the host NAT application does not define this callback, NetX NAT will use *_nx_udp_socket_bind* or *_nx_tcp_client_socket_bind* specifying that NetX find an available port number. For ICMP packets, NetX NAT will use its own *nx_nat_table_find_available_query_ID* to find an unused query ID and create a unique source IP address:query ID combination.

nx_nat_set_entry_timeout

NetX NAT calls this callback function when creating a translation table entry to assign a timeout value to a new table entry. Its arguments include the packet pointer and packet direction (inbound or outbound). The host application can parse packet information (e.g. protocol, destination IP address, and other relevant packet information) to determine the entry timeout.

nx_nat_set_filters

The host NAT application calls this function to set up NetX NAT with its own TCP, UDP and ICMP packet filter callback functions. These callback functions are required by NetX NAT to include a flag indicating if the packet is inbound or outbound. This gives the host application the ability to design its own filters to monitor and limit packet traffic. It can also use these filters for implementing ALGs described in the **Application Specific Gateways and NAT** section.

nx_nat_get_global_IP_address

[*Not in use*] The current NetX NAT release does not support multiple global IP address assignment by NAT. In future releases, the function of this callback will be to obtain an available global IP address for a local host among NAT's list of reserved IP addresses. NetX NAT is designed to store reserved IP addresses for when this configuration of NAT becomes available.

Dynamic Memory Allocation during NetX NAT Operation

The NetX NAT API requires dynamic memory allocation for internal operations such as creating entries in the NAT translation table, creating packet queues for receiving packet fragments and creating temporary sockets for binding port numbers used in address translation.

Other than the requirement for creating the byte pool, none of the NetX NAT API services deal directly with this byte pool. All memory allocation and deallocation is handled by internal NetX NAT functions.

To display byte pool reserves during NAT operation, NetX NAT has a diagnostic utility, *_nx_nat_utility_display_bytepool_reserves,* for troubleshooting purposes. See the section on **Description of NAT Services** for more details.

NetX NAT Event Logging

The NetX NAT API includes an event logging service **NX_NAT_EVENT_LOG** to log events and data during NAT operation. NetX NAT event logging macro is defined in *nx_nat.h*. This macro uses the *printf* statement for displaying output. However, the application code can define its own macro in lieu of the system library *printf*. Note that the macro makes use of the TX_INTERRUPT_SAVE_AREA for disabling interrupts during event logging. This is because *printf* handling is comparatively slow on embedded systems and the output is unpredictable unless interrupts are disabled while the output is displayed.

There are four levels of event logging, defined in *nx_nat.h*, ranging from NONE to ALL. The level of debug output allows the placement of event logging calls throughout the source code and NAT host application with a means to disable some or all of the output without removing the event logging calls themselves.

- NONE: No messages are displayed.
- LOG: This is associated with table entry additions or deletions or various NAT utility APIs such as services for displaying packet pool or byte pool reserves.
- SEVERE: only events of 'severe' consequences are logged. This would include memory allocation failure, failure to allocate packets, or failed packet transmission. Generally NAT operation can not continue as a result of a severe event.
- MODERATE: events of moderate or severe level are logged. This typically means a packet cannot be received or forwarded, e.g. the protocol header cannot be parsed. NAT operation can normally continue after the occurance of moderate events.
- ALL: all events are logged. In addition to event categories described above, these include events such as starting or stopping NAT operation, receiving packets with unknown protocol or packets that were rejected by the NAT host application packet filter and so one.

Below are examples of NX_NAT_EVENT_LOG logging call:

```
NX_NAT_EVENT_LOG(MODERATE, ("Packet payload exceeds driver MTU
                           while NAT is not configured for
                           fragmenting packets.\r\n"));

NX_NAT_EVENT_LOG(SEVERE, ("Error allocating memory for creating a
                           packet fragment queue. Status 0x%x. r\n",
                           status));


NX_NAT_EVENT_LOG(ALL, ("Resuming the NAT server thread\r\n"));

NX_NAT_EVENT_LOG(LOG, ("NAT byte pool status: Total memory %u
                         Available memory (bytes) %u
                         Number of fragments %ur\n",
                        byte_pool_size, available, fragments));
```

In the first event logging call, the packet is too large for NAT's Ethernet controller whose MTU (maximum transfer unit) is specified in the driver header file. This does not obviously prevent NAT from continuing operation handling other packets. But this particular packet cannot be forwarded.

In the second event logging call, the message includes the error ID. All NAT error codes are listed in *nx_nat.h.* NAT is unable to allocate memory to create a packet queue (in this case NAT is configured to at least receive fragmented packets). This is a serious error because it indicates that the memory pool is depleted or there is some other internal error with memory handling. In this case, NAT cannot continue with reliable operation and should be stopped for trouble shooting.

The third event logging call is called from the NetX NAT API service *_nx_nat_server_resume* which starts up NAT operation, and displays a message that NAT is starting up.

The last event logging call is used in the byte pool reserves utility, *_nx_nat_utility_display_bytepool_reserves*.

Other NetX NAT Features

Packet checksums

The native NetX IP task receives each packet and performs the IP header checksum and validates the packet. When the NATserver receives the packet from NetX, it validates the subprotocol (TCP, UDP or ICMP) header checksum unless the checksum computation option for that packet is disabled. For example,.if NX_NAT_DISABLE_WHOLE_TCP_TX_CHECKSUM is disabled, the TCP header checksum of outbound TCP packets from the local network are not validated.

NetX NAT has configurable options for TCP, UDP and ICMP inbound and outbound packets for the host application to enable or disable ‘whole’ checksum computation for each type of packet received in nx_nat.h.

Note: These options are distinct from the native NetX options described in *nx_system.h* when it computes checksums for packets intended for the NAT device itself.

Regardless if the NAT server is configured to validate packet checksums after receiving the packet from NetX, it must at least update the IP and subprotocol header checksums after address translation. If the ‘whole’ checksum option is not disabled, NAT will recompute the whole header checksum. Otherwise, NAT performs a checksum adjustment on the changed protocol header data. For example for an inbound UDP packet, NAT will adjust the UDP header checksum for just the destination IP address and port if the NX_NAT_DISABLE_WHOLE_UDP_RX_CHECKSUM is disabled.

There are some cases where NAT processing requires changing the packet payload as well and the entire checksum must be recomputed regardless how the above mentioned configuration options are set. In particular, packets containing the FTP “PORT” command from a local host require NAT to compute the entire checksum because the packet length and actual data are changed for the arguments of the PORT command.

Note: If NAT receives packets from a local Windows PC host, a packet snooper on the Windows PC may detect bad TCP checksums. This is most likely because the Windows operating system may be deferring the TCP checksum to the Ethernet controller and therefore has nothing to do with NAT address translation.

Suspending and Resuming NAT Operation

The NetX NAT API gives the host application the ability to suspend and resume the NAT operation throught the use of the *nx_nat_server_suspend* and *nx_nat_server_resume* service calls respectively. Reasons for suspending NAT operation might be major translation table operations, or network servicing, etc.

Note: To delete the translation table, it is recommended the application code suspend the NAT server first and allow time for current local host sessions with external hosts to complete.

NAT Network Statistics

NetX NAT provides some basic network information statistics such as the number of packets sent, dropped, total received by NAT and total number of bytes processed. These data are approximate numbers and should not be regarded as precise statistics. There are some cases when NAT cannot track packets after attempting to forward them. Examples would be when a fragmented packet is only partially received (e.g. one or more packets do not arrive or are invalid) or if a packet is dropped from an ARP queue.

NAT and Packet Fragmentation

NetX NAT creates a queue for storing packet fragments when it detects that a packet is part of a multipacket datagram and if the user configurable option NX_NAT_ENABLE_FRAGMENT_RECEIVE is enabled. Each packet received by NAT that is part of the same datagram, as determined by the IP header identification field, is stored in the same packet queue. The packet queue has a user configurable expiration timeout, during which if it has not received all the packets for the complete datagram, the NAT server will delete the packet queue and its packet fragments. This is necessary to conserve NAT and NetX resources, as well as foil denial-of-service attacks.

When NAT receives a packet that it must fragment to forward across the NAT boundary, it will check the IP header DONT_FRAGMENT bit. If it is not set, NAT will fragment the packet if the user configurable option NX_NAT_ENABLE_FRAGMENTATION is enabled.

Note: packet fragmentation for outbound packets across the NAT boundary carries the risk of packet corruption if multiple local hosts are sending packets to the same destination host. This is because only the first packet of a fragmented datagram carries the subprotocol (e.g. port) information. Port number distinguishes which local hosts are sending which packets when they are using the same source IP address. The destination host would not be able to know which packets belong to which session for the subsequent packet headers that only carry the IP header. On a small network, the chances of this situation occurring might be remote but this is left for the user to determine when it creates the NAT instance.

RFCs Supported by NetX NAT

NetX NAT API is compliant with RFCs 2663, 3022, 3235, and 4787.

Chapter 2 Installation and Use of NetX NAT

This chapter contains a description how to install, set up, and use the NetX NAT service.

NetX NAT Installation

NetX NAT is shipped on a single CD-ROM compatible disk. The package includes one NetX NAT API source and one header file, a demonstration application file, several NetX files, and a PDF file for this document, as follows:

nx_nat.c C Source file for NetX NAT API
nx_nat.h C Header file for NetX NAT API
demo_netx_nat.c Example host NAT application C source file

nx_nat.pdf PDF description of NetX NAT API for host applications (this document)

Copy the NAT source code files to the same directory where NetX and ThreadX are installed. For example, if NetX and ThreadX are installed in the directory "*threadx**mcf5485**green*" then *nx_nat.c*, *nx_nat.h* and the modified NetX files should be copied into this directory. Copy the modified NetX files over the existing NetX files. Copy the Ethernet controller driver files into this directory as well.

- The demo file included on the CD can be used as a template to create a NAT based application. However, it must be modified for the IP addresses, memory and packet pool creation, and driver files specific to the host environment.

- The modified NetX files do not need further modification to be used with a NAT enabled NetX library. Modifications made to these files are described in the **Changes to NetX for NAT Awareness** section below. Read this section before building the NAT enabled NetX library. It is anticipated that these changes will be incorporated into future releases of NetX source code.

Changes to NetX for NAT Awareness

- To enable NAT awareness uncomment #define NX_NAT_ENABLE and rebuild the NetX library with NX_INCLUDE_USER_DEFINE_FILE also defined if it is not already so.

Enabling NAT in the Application Code

Enabling NAT in the application code is straightforward.

- The project NetX library *nx.a* must be rebuilt with these modified NetX files as described above.

- The application code must include *nx_nat.h* after *tx_api.h* and *nx_api.h.* The latter two header files are necessary in order to use ThreadX and NetX.

- The application project code is compiled and linked with the modified NetX library to create the host NAT application executable.

The application code can dynamically disable NAT by calling the *_nx_ip_forwarding_disable* service. However, it is recommended to use the NetX NAT service *_nx_nat_server_suspend* to suspend NAT operation.

Small Example Demo NAT Setup

An example of how an application sets up NetX NAT is shown in the *tx_application_define* function in Figure 4 below. Unlike most NetX demo files distributed on the installation CD, this demo runs on an actual processor board with two Ethernet controllers, instead of a Windows PC using the virtual network driver *_nx_ram_network_driver*(). The NAT device is connected to the local domain through a local switch on its local interface, and to the external network through second switch on its external interface.

The private network is defined as 192.1.1.xx and has two local host nodes. The global network is defined as 192.2.2.xx and defines its gateway for out of network packets as 192.2.2.1. The two NetX IP instances for the private and global domains are created in line 109 and line 130 respectively and invoke the MCF5485 driver. A packet pool which NAT will use for allocating packets to send its own ICMP error messages is created in line 98.

The NAT translation table is created in line 149 and is preloaded with an entry for a UDP and an ICMP server among its local hosts in lines 165-170. The ThreadX byte pool NAT will need for dynamic creation of table entries and packet queues is created on line 174 along with a mutex lock on the byte pool. Finally NAT itself is created on line 182.

Optional NAT packet filter callback functions are assigned on line 206 (see **Appendix A** for example packet filter callback functions).

NetX NAT shares its own global IP address among its local hosts in line 210 using the NAPT configuration of network translation. Note that when NAT was created in line 182, a callback function was assigned for its *nx_nat_get_available_port* service. This function determines the global port number to assign to local host packets. See **Appendix A** for the example application defined *get_available_port* callback function.

```
1    /*
2       demo_netx_nat.c
3
4       This is a small demo of NAT (Network Address Translation) on the high-performance
5       NetX TCP/IP stack.  This demo relies on ThreadX, NetX and NAT APIs to perform network
6       address translation for IP packets traveling between private and external networks.
7    */
8
9
10
11   #define GLOBAL_IP_INSTANCE
12   #define INSIDE_IP_INSTANCE
13
14
15   #include  "nx_api.h"
16   #include  "nx_ip.h"
17   #include  "nx_nat.h"
18   #include  "nx_tcp.h"
19   #include  "nx_udp.h"
20   #include  "nx_mcf5485_fec.h"
21
22   /* Set up the NAT components. */
23
24   /* Create a NAT instance, packet pool and translation table. */
25
26   NX_NAT_DEVICE                        demo_nat_server;
27   NX_NAT_TRANSLATION_TABLE             demo_nat_table;
28   NX_PACKET_POOL                       nat_packet_pool;
29
30
31   /* IP instances for NAT routing packets to hosts on internal and external networks. */
32
33   NX_IP                                demo_nat_ip_private;
34   NX_IP                                demo_nat_ip_global;
35
36
37   /* Set up dynamic memory resources for NAT. */
38
39   TX_BYTE_POOL                         demo_nat_bytepool;
40   TX_MUTEX                             demo_nat_bytepool_mutex;
41
42
43   /* Define a global IP address shared among all local hosts for external bound packets.
*/
44   #define reserved_global_ip_address  NX_NAT_GLOBAL_IPADR
45
46   /* Set up a reserved IP address for a source IP address for outbound packets.*/
47   NX_NAT_RESERVED_IP_ITEM  reserved_global_ip_address_item;
48
```

```
49   /* Define NAT IP addresses and local host private IP addresses. */
50   #define NX_NAT_PRIVATE_IPADR           (IP_ADDRESS(192, 1, 1, 99))
51   #define NX_NAT_LOCAL_INSIDE_HOST1      (IP_ADDRESS(192, 1, 1, 2))
52   #define NX_NAT_LOCAL_INSIDE_HOST2      (IP_ADDRESS(192, 1, 1, 3))
53   #define NX_NAT_GLOBAL_IPADR            (IP_ADDRESS(192, 2, 2, 89))
54   #define NX_NAT_GLOBAL_GATEWAY_IP       (IP_ADDRESS(192, 2, 2, 1))
55
56   /* Create NAT structures for preloading NAT tables with static
57      entries for local server hosts. */
58   #define NX_NAT_ICMP_SERVER_IP       NX_NAT_GLOBAL_IPADR
59   #define NX_NAT_UDP_SERVER_IP        NX_NAT_GLOBAL_IPADR
60   NX_NAT_TRANSLATION_ENTRY            server_inbound_entry_udp;
61   NX_NAT_TRANSLATION_ENTRY            server_inbound_entry_icmp;
62
63   /* Network driver/configuration for mcf 5485. */
64   void nx_etherDriver_mcf5485(struct NX_IP_DRIVER_STRUCT *driver_req);
65
66   /* Set up NAT thread entry point. */
67   void    demo_nat_session_thread_entry(ULONG info);
68
69   /* Declare user defined callback services. */
70
71   UINT get_available_port(NX_NAT_TRANSLATION_TABLE *nat_table_ptr, UINT protocol, ULONG
       global_inside_ip_address,  UINT source_port, UINT *assigned_port);
72   VOID nat_tcp_filter(NX_PACKET *packet_ptr, UINT incoming_outgoing_direction, UINT
       *passed_filter);
73   VOID nat_udp_filter(NX_PACKET *packet_ptr, UINT incoming_outgoing_direction, UINT
       *passed_filter);
74   VOID nat_icmp_filter(NX_PACKET *packet_ptr, UINT incoming_outgoing_direction, UINT
       *passed_filter);
75   VOID set_table_entry_timeout(NX_PACKET *packet_ptr, UINT packet_direction, UINT
       *response_timeout);
76
77
78   /* Define main entry point.  */
79   int main()
80   {
81       /* Enter the ThreadX kernel.  */
82       tx_kernel_enter();
83   }
84
85
86   /* Define what the initial system looks like.  */
87   void    tx_application_define(void *first_unused_memory)
88   {
89
90   UINT      status;
91   UCHAR    *free_memory_pointer;
92
93
94       /* Setup the pointer to unallocated memory.  */
95       free_memory_pointer =  (UCHAR *) first_unused_memory;
96
97       /* Create NAT packet pool. */
98       status =  nx_packet_pool_create(&nat_packet_pool, "NAT Packet Pool",
99                                       NX_NAT_PACKET_SIZE, free_memory_pointer,
100                                      NX_NAT_PACKET_POOL_SIZE);
101
102      /* Update pointer to unallocated (free) memory. */
103      free_memory_pointer = free_memory_pointer + NX_NAT_PACKET_POOL_SIZE;
104
105      /* Initialize the NetX system. */
106      nx_system_initialize();
107
108      /* Create IP instances for stub domain (private network) */
109      status = nx_ip_create(&demo_nat_ip_private, "NAT Private IP Instance",
                  NX_NAT_PRIVATE_IPADR, NX_NAT_PRIVATE_NETMASK,
                  &nat_packet_pool, nx_etherDriver_mcf5485, free_memory_pointer,
                  NX_NAT_IP_THREAD_STACK_SIZE, NX_NAT_IP_THREAD_PRIORITY);
111
112
113      /* Update pointer to unallocated (free) memory. */
114      free_memory_pointer =  free_memory_pointer + NX_NAT_IP_THREAD_STACK_SIZE;
115
116      /* Enable ARP and supply ARP cache memory for private network IP isntance. */
117      status =  nx_arp_enable(&demo_nat_ip_private, (void **) free_memory_pointer,
118                              NX_NAT_ARP_CACHE_SIZE);
119
120      /* Update pointer to unallocated (free) memory. */
121      free_memory_pointer = free_memory_pointer + NX_NAT_ARP_CACHE_SIZE;
122
```

```
123     /* Enable network protocols on the private network. */
124     nx_tcp_enable(&demo_nat_ip_private);
125     nx_udp_enable(&demo_nat_ip_private);
126     nx_icmp_enable(&demo_nat_ip_private);
127     nx_ip_fragment_enable(&demo_nat_ip_private);
128
129     /* Create the IP instance for the external network. */
130     status = nx_ip_create(&demo_nat_ip_global, "NAT Global IP Instance",
                  NX_NAT_GLOBAL_IPADR, NX_NAT_GLOBAL_NETMASK,
                  &nat_packet_pool, nx_etherDriver_mcf5485, free_memory_pointer,
                  NX_NAT_IP_THREAD_STACK_SIZE, NX_NAT_IP_THREAD_PRIORITY);
132
133
134     /* Update pointer to unallocated (free) memory. */
135     free_memory_pointer =  free_memory_pointer + NX_NAT_IP_THREAD_STACK_SIZE;
136
137     /* Enable network protocols on the external network. */
138     nx_tcp_enable(&demo_nat_ip_global);
139     nx_udp_enable(&demo_nat_ip_global);
140     nx_icmp_enable(&demo_nat_ip_global);
141     nx_ip_fragment_enable(&demo_nat_ip_global);
142     nx_arp_enable(&demo_nat_ip_global, (void **) free_memory_pointer,
                    NX_NAT_ARP_CACHE_SIZE);
143
144     /* Update pointer to unallocated (free) memory. */
145     free_memory_pointer = free_memory_pointer + NX_NAT_ARP_CACHE_SIZE;
146
147     /* Create the NetX NAT translation table with dynamic translation and IP address
148        overloading enabled. */
149     status =  nx_nat_table_create(&demo_nat_server, &demo_nat_table, 100,
                                     NX_TRUE, NX_TRUE);
150
151     /* Check for error. */
152     if (status != NX_SUCCESS)
153     {
154
155         /* Log the event. */
156         NX_NAT_EVENT_LOG(SEVERE, ("Error creating NAT translation table instance. "
157                                   "Status 0x%x.\n\r", status));
158
159         /* Abort. */
160         return;
161     }
162
163     /* Preload the NAT translation table with an entry for a local UDP server on port
164        123 and is a permanent entry. */
165     status = nx_nat_table_entry_preload(&server_inbound_entry_udp, &demo_nat_table,
                                        NX_NAT_UDP_SERVER_IP, NX_NAT_LOCAL_INSIDE_HOST1,
                                        123, NX_IP_UDP, NX_TRUE);
166
167
168     /* Preload the NAT translation table with an entry for a local ICMP server as a
169        permanent entry. */
170     status = nx_nat_table_entry_preload(&server_inbound_entry_icmp, &demo_nat_table,
                           NX_NAT_ICMP_SERVER_IP, NX_NAT_LOCAL_INSIDE_HOST2, 0,
                           NX_IP_ICMP, NX_TRUE);
171
172
173     /* Setup memory resource for dynamic NAT translation table entries. */
174     status = tx_byte_pool_create(&demo_nat_bytepool, "NAT Bytepool",
                                    (VOID **)free_memory_pointer, NX_NAT_BYTE_POOL_SIZE);
175
176     free_memory_pointer += NX_NAT_BYTE_POOL_SIZE;
177
178     status = tx_mutex_create(&demo_nat_bytepool_mutex, "NAT Bytepool Mutex",
                                TX_NO_INHERIT);
179
180     /* Create a NetX NAT Server with a static translation table. This will
181        also create the NAT server thread. */
182     status =  nx_nat_create(&demo_nat_server, &demo_nat_table, &demo_nat_ip_private,
183                         &demo_nat_ip_global, NX_NAT_GLOBAL_GATEWAY_IP, NX_TRUE, NX_TRUE,
184                         free_memory_pointer, &demo_nat_bytepool, &demo_nat_bytepool_mutex,
185                         NX_NAT_BYTE_POOL_MUTEX_WAIT, NULL,
186                         get_available_port, set_table_entry_timeout);
187
188     /* Set the external network gateway for out of network packets to be forwarded
189        beyond the external network.*/
190     demo_nat_ip_global.nx_ip_gateway_address = NX_NAT_GLOBAL_GATEWAY_IP;
191
192     /* Check for error. */
     if (status != NX_SUCCESS)
```

```
193      {
194
195          /* Log the event. */
196          NX_NAT_EVENT_LOG(SEVERE, ("Error creating NAT instance. Status 0x%x.\n\r",
                                       status));
197
198          /* Abort. */
199          return;
200      }
201
202      /* Update pointer to unallocated (free) memory. */
203      free_memory_pointer += NX_NAT_THREAD_STACK_SIZE;
204
205      /* Apply protocol specific NAT filter rules for inbound and outbound packets. */
206      nx_nat_set_filters(&demo_nat_server, nat_tcp_filter, nat_udp_filter,
                           nat_icmp_filter);
207
208      /* Create a reserved global IP address that NAT will assign as the global source IP
209         address for all outbound packets. */
210      nx_nat_add_reserved_ip_list(&demo_nat_server, reserved_global_ip_address,
                                    &reserved_global_ip_address_item);
211
212      /* Start the NAT server. */
213      NX_NAT_EVENT_LOG(ALL, ("Starting NAT service...\r\n"));
214
215      /* Start NAT up. No need to reset the packet counts, already initialized to zero. */
216      status = nx_nat_server_resume(&demo_nat_server, NX_FALSE);
217
218      return;
219  }
```

Figure 4. Setting up NetX NAT

Chapter 3 NetX NAT Configuration Options

Configurable options for the NetX NAT API can be found in *nx_nat.*hwith the exception of the first one, **NX_DISABLE_ERROR_CHECKING** which is found in *nx_nat.c*. The following list includes all options and their function described in detail:

Define	Meaning
NX_DISABLE_ERROR_CHECKING	This option if defined removes the basic NAT error checking. It is typically used after the application has been debugged. The default NetX NAT status is defined (enabled).
NX_NAT_ DEBUG	This option sets the level of NAT event logging, from logging ALL messages, to only logging SEVERE errors. To disable logging, set level to NONE. The default NetX NAT level is MODERATE.

NX_NAT_GLOBAL_NETMASK

This option sets the network mask for NAT's global network interface. The default NetX NAT global netmask is 0xFFFFFF00.

NX_NAT_PRIVATE_NETMASK

This option sets the network mask for NAT's private network interface. The default NetX NAT private netmask is 0xFFFFFF00.

NX_NAT_THREAD_PRIORITY

This option sets the NAT thread priority. The default NetX NAT value is 2.

NX_NAT_PREEMPTION_THRESHOLD

This option sets the NAT preemption level. The default NetX NAT value is set to the same level as NX_NAT_NAT_THREAD_PRIORITY (thread preemption not enabled).

NX_NAT_THREAD_STACK_SIZE

This option sets the background NAT server thread stack size. The default thread size is 2048 bytes.

NX_NAT_THREAD_TIME_SLICE

This option sets the time slice the scheduler allows for thread execution. The default NetX NAT time slice is TX_NO_TIME_SLICE.

NX_NAT_BYTE_POOL_SIZE

This option sets the NetX NAT byte pool size. The NetX NAT byte pool default size is 4096 bytes.

NX_NAT_BYTE_POOL_ NAME

This option sets the name of the byte pool. The NetX NAT default is "NAT Bytepool."

NX_NAT_BYTE_POOL_MUTEX_NAME

This option sets the name of the byte pool mutex. The default NetX NAT byte pool mutex name is “NAT Bytepool Mutex.”

NX_NAT_BYTE_POOL_MUTEX_WAIT

This option sets the timeout to obtain a lock on NAT’s byte pool mutex. The default timeout is 5 seconds.

NX_NAT_TIMER_TIMEOUT_INTERVAL

This option sets the interval at which the timer that updates the time remaining on NAT translation table entries and packet fragment queues is run. The default value is 10 seconds.

NX_NAT_TABLE_ENTRY_RESPONSE_TIMEOUT

This option sets the expiration timeout without receiving a matching packet for a NAT translation table entry. The default value is 60 seconds.

NX_NAT_TABLE_MUTEX_WAIT

This option sets the time to wait for obtaining the NAT translation table mutex. The default value is 20 seconds.

NX_NAT_PACKET_SIZE

This option sets the NetX NAT packet size. This should be set to below the MTU size of the underlying Ethernet hardware. The default NetX NAT packet size in bytes is 1500.

NX_NAT_PACKET_POOL_SIZE

This option sets the NetX NAT packet pool size for NAT to allocate packets for sending its own ICMP error messages. The

default NetX NAT packet pool size in bytes is (10 * NX_NAT_PACKET_SIZE).

NX_NAT_PACKET_ALLOCATE_TIMEOUT

This option sets the timeout for allocating a packet from the NAT packet pool. The default NetX timeout in seconds is NX_NO_WAIT.

NX_NAT_IP_THREAD_PRIORITY

This option sets the thread priority of NAT's IP thread tasks. The default value is 2.

NX_NAT_IP_THREAD_STACK_SIZE

This option sets the thread stack size for NAT's IP helper threads. The default thread stack size is 2048 bytes.

NX_NAT_ARP_CACHE_SIZE

This option sets the ARP cache size for NAT's IP tasks. Each ARP entry is 52 bytes, so the number of ARP entries is the memory size divided by 52. The default NetX NAT ARP cache memory size is 1040 (20 entries).

NX_NAT_TRANSLATION_TABLE_MAX_ENTRIES

This option sets maximum allowed number of entries in the NAT translation table. The default value is 1000.

NX_NAT_ALLOW_INBOUND_BROADCAST_PACKETS

This option enables NAT to forward broadcasts packets from the external network onto the internal network. The default setting is NX_FALSE (disabled).

NX_NAT_ALLOW_OUTBOUND_BROADCAST_PACKETS

This option enables NAT to forward broadcasts packets from the internal network onto the external network. The default setting is NX_FALSE (disabled).

NX_NAT_REFRESH_TIMER_ON_INBOUND_PACKETS

This option enables NAT to refresh the time out for a translation table entry receiving an packet that matches that entry. The default value is NX_TRUE (enabled).

NX_NAT_ENABLE_FRAGMENT_RECEIVE

This option enables NAT to accept fragmented datagrams. The default value is NX_TRUE (enabled). NetX NAT accepts fragmented packets received out of order.

NX_NAT_ENABLE_FRAGMENTATION

This option enables NAT to fragment packets when their payload exceeds the maximum transmit capacity of the adjacent link (if the packet DON'T_FRAGMENT bit is not set of course). The default value is NX_TRUE (enabled).

NX_NAT_PACKET_QUEUE_TIMEOUT

This option sets the timeout NetX NAT imposes on a packet queue storing fragments belonging to the same datagram to receive all the fragments. The default value in seconds is 120.

NX_NAT_DISABLE_TRANSLATION_TABLE_INFO

This option prevents NAT from displaying the translation table when the source code calls *_nx_nat_utility_translation_table*. This service call is intended for debugging and troubleshooting

purposes. The default value is enabled.

NX_NAT_DISABLE_WHOLE_IP_CHECKSUM

This option allows NAT to update the IP header checksum for changed IP address data instead of computing the entire checksum. This applies to both inbound and outbound packets. The default value is enabled.

NX_NAT_DISABLE_WHOLE_TCP_RX_CHECKSUM

This option allows NAT to update the TCP header checksum for inbound TCP packets with just the changed TCP header data instead of computing the entire checksum. The default value is enabled.

NX_NAT_DISABLE_WHOLE_TCP_TX_CHECKSUM

This option allows NAT to update the TCP header checksum for outbound TCP packets with just the changed TCP header data instead of computing the entire checksum. The default value is enabled.

NX_NAT_DISABLE_WHOLE_UDP_RX_CHECKSUM

This option allows NAT to update the UDP header checksum for inbound UDP packets with just the changed UDP header data instead of computing the entire checksum. The default value is enabled.

NX_NAT_DISABLE_WHOLE_UDP_TX_CHECKSUM

This option allows NAT to update the UDP header checksum for outbound UDP packets with just the changed UDP header data instead of computing the entire checksum. The default value is enabled.

NX_NAT_DISABLE_WHOLE_ICMP_RX_CHECKSUM

This option allows NAT to update the ICMP header checksum for inbound ICMP packets with just the changed ICMP header data instead of computing the entire checksum. The default value is enabled.

NX_NAT_DISABLE_WHOLE_ICMP_TX_CHECKSUM

This option allows NAT to update the ICMP header checksum for outbound ICMP packets with just the changed ICMP header data instead of computing the entire checksum. The default value is enabled.

NX_NAT_FTP_CONTROL_PORT

This option defines the port number for connecting to an FTP server for sending commands ("control channel"). The default control port is 21.

NX_NAT_FTP_DATA_PORT

This option defines the port number for receiving data from an FTP server ("data channel"). The default data port is 20.

NX_NAT_TCP_PORT_BIND_TIMEOUT

This option defines the timeout for binding a port number to a TCP socket. The default value in seconds is 100.

NX_NAT_ENABLE_ZERO_UDP_CHECKSUM

This option enables NAT to forward UDP packets with the UDP header checksum set to zero. The default value is NX_TRUE (enabled).

NX_NAT_MAX_UDP_RX_QUEUE

This option sets upper limit on the number of UDP packets stored in the UDP socket receive queue

waiting for IP address resolution. The default value is 5 packets.

NX_NAT_UDP_PORT_BIND_TIMEOUT

This option defines the timeout for binding a port number to a UDP socket. The default value in seconds is 100.

NX_NAT_IMCP_MESSAGE_SIZE

This option sets maximum allowed message size for an ICMP message to be embedded in an ICMP error message initiated by NAT. The default size is 200 bytes.

NX_NAT_START_ICMP_QUERY_ID

This option sets the starting value for finding an unused query ID to assign an outbound ICMP query packet. The default value is 1000 (most PCs for example use 200, so this will not conflict with this class of hosts).

NX_NAT_END_ICMP_QUERY_ID

This option sets the upperlimit of query IDs to assign an outbound ICMP query packet. The default value is 100 above the starting value (1100).

Chapter 4 Description of NetX NAT Services

This chapter contains a description of all NetX NAT API services (listed below) in alphabetical order.

In the “Return Values” section in the following API descriptions, values in **BOLD** are not affected by the **NX_DISABLE_ERROR_CHECKING** define that is used to disable API error checking, while non-bold values are completely disabled.

nx_nat_add_reserved_ip_list
Add an IP address to NAT’s list of global IP addresses

nx_nat_create
Create a NAT Instance

nx_nat_delete
Delete a NAT instance

nx_nat_server_resume
Resume the NAT server thread and NAT operation

nx_nat_server_suspend
Suspend the NAT server and NAT operation

nx_nat_set_filters
Set function pointers to user defined protocol specific filters to apply to inbound and outbound packets forwarded by NAT

nx_nat_table_create
Create the NAT translation table

nx_nat_table_delete
Delete the NAT translation table

nx_nat_table_entry_create
Create a translation table entry

nx_nat_table_entry_delete
Delete a translation table entry

nx_nat_table_entry_preload

Create a static translation table entry

nx_nat_table_find_entry
Find an entry in the translation table based on packet criteria such as source and destination IP address, port and protocol

nx_nat_utility_display_arp_table
Display entries in the native NetX ARP table for debugging and troubleshooting purposes

nx_nat_utility_display_bytepool_reserves
Display total and available bytes in the NAT byte pool for debugging and troubleshooting purposes

nx_nat_utility_display_packetpool_reserves
Display total and available packets used by the NAT IP tasks for debugging and troubleshooting purposes

nx_nat_utility_display_translation_table
Display each entry in the NAT translation table

nx_nat_utility_get_destination_port
Parse the destination port from a packet subprotocol header. For ICMP packets, the query ID is parsed.

nx_nat_utility_get_source_port
Parse the source port from a packet subprotocol header. For ICMP packets, the query ID is parsed.

nx_ nat_add_reserved_ip_list

Add global (reserved) IP address to NAT list

Prototype

```
UINT  nx_nat_add_reserved_ip_list(
                          NX_NAT_DEVICE *nat_ptr,
                          ULONG reserved_ip_address,
                          NX_NAT_RESERVED_IP_ITEM *nat_ip_item_ptr)
```

Description

This service adds a reserved IP address to the list of IP addresses owned by NAT for assigning to local host packet source IP addresses for outbound packets.

Input Parameters

nat_ptr	Pointer to NAT server
reserved_ip_address	IP address to add to NAT list
nat_ip_item_ptr	Pointer to reserved address item

Return Values

NX_SUCCESS	(0x00)	Client successfully connected to Server
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter
NX_NAT_PARAM_ERROR	(0xD02)	Invalid non pointer input

Allowed From

Application code

Example

```
/* Add an IP address to NAT's list for source IP address translation. */
status = _nx_nat_add_reserved_ip_list(nat_ptr, global_ip_address, nat_ip_item);

/* If item was successfully added, status = NX_SUCCESS. */
```

See Also

nx_nat_table_create, nx_nat_table_entry_create, nx_nat_table_entry_delete, nx_nat_table_entry_preload

nx_nat_create

Create a NAT instance

Prototype

```
UINT    nx_nat_create(NX_NAT_DEVICE *nat_ptr,
            NX_NAT_TRANSLATION_TABLE *nat_table_ptr,
            NX_IP *nat_private_ip_ptr, NX_IP *nat_global_ip_ptr,
            ULONG nat_global_gateway_address,
            UINT icmp_query_respond_enabled,
            UINT icmp_errmsg_receive_enabled,
            VOID *stack_ptr, TX_BYTE_POOL *bytepool_ptr,
            TX_MUTEX *bytepool_mutex_ptr, UINT bytepool_mutex_timeout,
            UINT *nx_nat_get_global_IP_address)
                  (NX_NAT_TRANSLATION_TABLE *table_ptr, UINT protocol,
                  ULONG *global_ip_address,
                  ULONG private_inside_ip_address,
                  ULONG destination_ip_address),
            UINT (*nx_nat_get_available_port)
                  (NX_NAT_TRANSLATION_TABLE *nat_table_ptr,
                  UINT protocol, ULONG global_inside_ip_address,
                  UINT private_inside_port, UINT *assigned_port),
            VOID (*nx_nat_set_entry_timeout)
                  (NX_PACKET *packet_ptr, UINT packet_direction,
                  UINT *response_timeout))
```

Description

This service creates an instance of the NAT server.

Input Parameters

nat_ptr	Pointer to NAT instance to create
nat_table_ptr	Pointer to NAT translation table
nat_private_ip_ptr	Pointer to private IP network instance
nat_global_ip_ptr	Pointer to global IP network instance
nat_global_gateway_address	Global gateway IP address
icmp_query_respond_enabled	Enable NAT to accept ICMP query packets
icmp_errmsg_receive_enabled	Enable NAT to accept ICMP error message packets
stack_ptr	Pointer in free memory for NAT server thread
bytepool_ptr	Pointer to NAT byte pool
bytepool_mutex_ptr	Pointer to NAT byte pool mutex
bytepool_mutex_timeout	Timeout for obtaining byte pool mutex
nx_nat_get_global_IP_address	Pointer to callback function service for obtaining global IP address
nx_nat_get_available_port	Pointer to callback function service for obtaining global port number
nx_nat_set_entry_timeout	Pointer to callback function service for setting translation table entry timeout

Return Values

NX_SUCCESS	(0x00)	Client successfully created
NX_NAT_OVERLAPPING_SUBNET_ERROR	(0xD08)	Private and public networks overlap
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter
NX_NAT_PARAM_ERROR	(0xD02)	Invalid non pointer input

Allowed From

Application code

Example

```
/* Create a NAT instance which will accept ICMP query and error message packets. */
status =  nx_nat_create(&demo_nat_server, &demo_nat_table, &demo_nat_ip_private,
            &demo_nat_ip_global, NX_NAT_GLOBAL_GATEWAY_IP, NX_TRUE, NX_TRUE,
            free_memory_pointer, &demo_nat_bytepool, &demo_nat_bytepool_mutex,
            NX_NAT_BYTE_POOL_MUTEX_WAIT, get_available_address, get_available_port,
            set_table_entry_timeout);

/* If the NAT instance was successfully created, status = NX_SUCCESS. */
```

See Also

nx_nat_delete, nx_nat_table_create, nx_nat_server_resume, nx_nat_server_suspend, nx_nat_server_thread_entry

nx_nat_delete

Delete a NAT instance

Prototype

```
UINT    nx_nat_delete(NX_NAT_DEVICE *nat_ptr)
```

Description

This service deletes a previously created NAT instance.

Input Parameters

nat_ptr	Pointer to NAT instance to delete

Return Values

NX_SUCCESS	(0x00)	NAT successfully deleted
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Application code

Example

```
/* Delete the NAT instance. */
status =  nx_nat_delete (&demo_nat_server);

/* If the NAT instance was successfully deleted, status = NX_SUCCESS. */
```

See Also

nx_nat_create, nx_nat_table_create, nx_nat_server_resume, nx_nat_server_suspend, nx_nat_server_thread_entry

nx_nat_server_resume

Resume the NAT server operation

Prototype

```
UINT    nx_nat_server_resume( NX_NAT_DEVICE *nat_ptr,
                              UINT reset_packet_counts)
```

Description

This service resumes the background NAT server thread and reactivates NAT operation, e.g. NetX forwarding IP packets to NAT. If the *reset_packet_counts* argument is set, it will also reset the count of packet and bytes processed by NAT to zero.

Input Parameters

nat_ptr	Pointer to NAT instance
reset_packet_counts	Indicate if NAT should reset its count of packet and bytes processed.

Return Values

NX_SUCCESS	(0x00)	NAT successfully resumed
NX_NAT_NO_TRANSLATION_TABLE	(0xD0C)	NAT has no translation table
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Threads

Example

```
/* Resume the NAT server operation and background thread, and reset count of packets and
bytes processed to zero. */
status =  nx_nat_server_resume (nat_ptr, NX_TRUE);

/* If the NAT operation successfully resumed, status = NX_SUCCESS. */
```

See Also

nx_nat_create, nx_nat_delete, nx_nat_table_create, nx_nat_server_suspend, nx_nat_server_thread_entry

nx_nat_server_suspend

Suspend the NAT Server operation

Prototype

```
UINT    nx_nat_server_suspend( NX_NAT_DEVICE *nat_ptr)
```

Description

This service suspends the background NAT server thread and deactivates the IP forwarding packets to NAT.

Input Parameters

nat_ptr	Pointer to NAT instance

Return Values

NX_SUCCESS	(0x00)	NAT successfully suspended
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Threads

Example

```
/* Suspend the NAT server operation and background thread. */
status =  nx_nat_server_suspend (nat_ptr);

/* If the NAT operation successfully suspended, status = NX_SUCCESS. */
```

nx_nat_set_filters

Set filters for the NAT server to apply to packet traffic

Prototype

```
UINT    nx_nat_set_filters(NX NX_NAT_DEVICE *nat_ptr,
            VOID (*nat_tcp_filter)
                  (NX_PACKET *packet_ptr,
                  UINT incoming_outgoing_direction,
                  UINT *passed_filter),
            VOID (*nat_udp_filter)
                  (NX_PACKET *packet_ptr,
                  UINT incoming_outgoing_direction,
                  UINT *passed_filter),
            VOID (*nat_icmp_filter)
                  (NX_PACKET *packet_ptr,
                  UINT incoming_outgoing_direction,
                  UINT *passed_filter))
```

Description

This service sets the function pointers to the user defined protocol specific packet filters for NAT to apply to packet traffic across the private and global domains.

Input Parameters

nat_ptr	Pointer to the NAT instance
nat_tcp_filter	Pointer to TCP packet filter
nat_udp_filter	Pointer to UDP packet filter
nat_icmp_filter	Pointer to ICMP packet filter

Return Values

NX_SUCCESS	(0x00)	Packet filters successfully set
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Application Code

Example

```
/* Set the following packet filters to the NAT instance. */
status = nx_nat_set_filters((&demo_nat_server, nat_tcp_filter, nat_udp_filter,
                           nat_icmp_filter);

/* If the filters are successfully set, status = NX_SUCCESS. */
```

See Also

nx_nat_process_outbound_TCP_packet, nx_nat_process_inbound_TCP_packet,

nx_nat_process_outbound_UDP_packet, nx_nat_process_inbound_UDP_packet,
nx_nat_process_outbound_ICMP_packet, nx_nat_process_inbound_ICMP_packet

nx_nat_table_create

Create the NAT translation table

Prototype

```
UINT    nx_nat_table_create(NX_NAT_DEVICE *nat_ptr,
                NX_NAT_TRANSLATION_TABLE *nat_table_ptr,
                UINT table_mutex_timeout,
                UINT dynamic_translation_ok,
                UINT overload_ip_address_enabled)
```

Description

This service creates the NAT translation table with mutex protection.

Input Parameters

nat_ptr	Pointer to the NAT server
nat_table_ptr	Pointer to the translation table
table_mutex_timeout	Timeout to obtain exclusive lock on table
dynamic_translation_ok	Enable NAT to add table entries once NAT operation has started
overload_ip_address_enabled	Enable NAT to use IP address overloading

Return Values

NX_SUCCESS	(0x00)	NAT table successfully created
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter
NX_NAT_PARAM_ERROR	(0xD02)	Invalid non pointer input

Allowed From

Application code

Example

```
/* Create the translation table for the NAT server with both dynamic translation and IP
overloading enabled. */

status = nx_nat_table_Create(nat_ptr, nat_table_ptr, NX_TRUE, NX_TRUE);

/* If the table was successfully created, status = NX_SUCCESS. */
```

See Also

nx_nat_create, nx_nat_delete, nx_nat_table_delete, nx_nat_table_entry_create, nx_nat_table_entry_preload, nx_nat_table_find_entry, nx_nat_table_entry_delete

nx_nat_table_delete

Delete the NAT translation table

Prototype

```
UINT    nx_nat_table_delete(NX_NAT_TRANSLATION_TABLE *nat_table_ptr)
```

Description

This service deletes the NAT translation table. However it can only delete each entry if not in use by a current NAT operation in progress. At the end of the list of table entries, if it has not been able to delete all table entries, it logs a warning that not all entries could be deleted. This situation is still handled as a successful outcome of table deletion.

Input Parameters

nat_table_ptr	Pointer to NAT translation table

Return Values

NX_SUCCESS	(0x00)	Table successfully deleted
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Threads

Example

```
/* Delete the NAT translation table. */
status = nx_nat_table_delete(nat_table_ptr);

/* If the table was successfully deleted, status = NX_SUCCESS. */
```

See Also

nx_nat_create, nx_nat_delete, nx_nat_table_create, nx_nat_table_entry_create, nx_nat_table_entry_preload, nx_nat_table_find_entry, nx_nat_table_entry_delete

nx_nat_table_entry_create

Create an entry for the NAT translation table

Prototype

```
UINT    nx_nat_table_entry_create(
                NX_NAT_TRANSLATION_TABLE *nat_table_ptr,
                UINT protocol, ULONG private_inside_ip_address,
                ULONG global_inside_ip_address,
                UINT  private_inside_port, UINT global_inside_port,
                ULONG external_ip_address, UINT external_port,
                UINT response_timeout, ULONG fragment_id,
                NX_TCP_SOCKET *tcp_socket_ptr,
                NX_UDP_SOCKET *udp_socket_ptr,
                UINT inbound_packet_initiated,
                NX_NAT_TRANSLATION_ENTRY **match_entry_ptr)
```

Description

This service creates an entry for the NAT translation table containing a relevant data such as local host private and global source IP address, and private and global source port (or query ID) depending on packet protocol. The caller must also indicate if the entry is for a host acting as a server (accepts inbound packets). The service creates the entry and returns a pointer to the new entry in the table.

Input Parameters

nat_table_ptr	Pointer to NAT translation table
protocol	Packet subprotocol (e.g TCP)
private_inside_ip_address	Local host private IP address
global_inside_ip_address	Local host global IP address
private_inside_port	Local host private port
external_ip_address	IP address of external host local host is sending packets to
external_port	Port number of external host local host is sending packets to
response_timeout	Table entry timeout value
fragment_id	IP header ID for fragmented packets
tcp_socket_ptr	Pointer to TCP socket for binding local host packet port number
udp_socket_ptr	Pointer to UDP socket for binding the local host packet port number
inbound_packet_initiated	Indicates if entry is for a local host server
match_entry_ptr	Pointer to table entry just created

Return Values

NX_SUCCESS	(0x00)	Entry successfully created
NX_NAT_INVALID_TABLE_ENTRY	(0xD0E)	Invalid entry or missing information
NX_NAT_TRANSLATION_TABLE_FULL	(0xD0C)	Translation table is full
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter
NX_NAT_PARAM_ERROR	(0xD02)	Invalid non pointer input

Allowed From

Threads

Example

```
/* Create an entry for an outbound TCP packet. */

status = nx_nat_table_entry_create(nat_table_ptr, nat_ptr -> nat_table_ptr, NX_IP_TCP,
                                                  private_inside_ip_address,
                                                  global_inside_ip_address,
                                                  private_inside_port,
                                                  global_inside_port,
                                                  external_ip_address,
                                                  external_port, response_timeout,
                                                  fragment_id, 0x0, 0x0,
                                                  NX_FALSE, &entry_ptr);

/* If the entry was successfully created, status = NX_SUCCESS. */
```

See Also

nx_nat_table_create, nx_nat_table_entry_delete, nx_nat_table_entry_preload, nx_nat_table_find_entry, nx_nat_table_entry_preload

nx_nat_table_entry_delete

Delete an entry from the NAT translation table

Prototype

```
UINT    nx_nat_table_entry_delete(NX_NAT_TRANSLATION_ENTRY *entry_ptr,
                                  NX_NAT_TRANSLATION_TABLE *nat_table_ptr)
```

Description

This service deletes the specified entry from the translation table and releases memory (for dynamically created entries) back to the NAT byte pool.

Input Parameters

entry_ptr	Pointer to entry to delete
nat_table_ptr	Pointer to the NAT translation table

Return Values

NX_SUCCESS	(0x00)	Entry successfully deleted
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Threads

Example

```
/* Delete the specified entry from the translation table. */
status = nx_nat_table_entry_delete(entry_ptr, nat_table_ptr);

/* If the entry was successfully deleted, status = NX_SUCCESS. */
```

See Also

nx_nat_create, nx_nat_delete, nx_nat_table_create, nx_nat_table_delete, nx_nat_table_entry_create, nx_nat_table_find_entry, nx_nat_table_entry_preload

nx_nat_table_entry_preload

Add a static entry to the NAT table

Prototype

```
UINT    nx_nat_table_entry_preload(
                    NX_NAT_TRANSLATION_ENTRY *entry_ptr,
                    NX_NAT_TRANSLATION_TABLE *nat_table_ptr,
                    ULONG global_inside_ip_address,
                    ULONG private_inside_ip_address,
                    UINT  global_inside_port, UINT protocol,
                    UINT inbound_packet_initiated)
```

Description

This function creates a static entry (permanent entry, has no expiration timeout) and adds it to the NAT translation table. This can only be called when NAT is not in operation. It is intended to be used when the host application is setting up its NAT operation, and there is no table locking protection in place. These entries are usually created for local host servers, although NAT does allow for local hosts to be statically assigned global IP addresses.

Input Parameters

entry_ptr	Pointer to static entry to add
nat_table_ptr	Pointer to NAT translation table
global_inside_ip_address	Global IP address to assign
private_inside_ip_address	Private IP address of local host
global_inside_port	Global port number to assign
protocol	Protocol for when the global IP address:port applies
inbound_packet_initiated	Enable NAT to accept inbound packets without a previously sent packet from a local host

Return Values

NX_SUCCESS	(0x00)	Entry successfully added to table
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Application code

Example

```
/* Add a static entry to the translation table for a UDP server to accept inbound packets
on port 123 (SNTP server port). */
```

```
status = nx_nat_table_entry_preload(entry_ptr, nat_table_ptr, NX_NAT_UDP_SERVER_IP,
                                    NX_NAT_LOCAL_INSIDE_HOST1, 123, NX_IP_UDP, NX_TRUE);

/* If the entry was successfully created and added to the table, status = NX_SUCCESS. */
```

See Also

nx_nat_create, nx_nat_delete, nx_nat_table_create, nx_nat_table_delete, nx_nat_table_entry_create, nx_nat_table_find_entry, nx_nat_table_entry_delete

nx_nat_table_find_entry

Find an entry in the NAT translation table

Prototype

```
UINT    nx_nat_table_find_entry(NX_NAT_DEVICE *nat_ptr,
                    NX_NAT_TRANSLATION_ENTRY *entry_to_match,
                    NX_NAT_TRANSLATION_ENTRY **match_entry_ptr,
                    UINT skip_inbound_init_entries)
```

Description

This service finds a matching entry in the NAT translation table based on the criteria in the *entry_to_match* argument. If the *skip_inbound_init_entries* argument is set, the search will exclude entries in the table for local host servers (type = static). If the search does not turn up a matching entry, the *match_entry_ptr* argument is returned as NULL with the return value NX_SUCCESS to indicate the search completed without errors**.**

Input Parameters

nat_ptr	Pointer to NAT instance
entry_to_match	Pointer to entry containing search criteria
match_entry_ptr	Pointer to location in table of matching entry
skip_inbound_init_entries	Indicate if search excludes local servers

Return Values

NX_SUCCESS	(0x00)	Table search was successful
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Threads

Example

```
/* Search the table for the entry matching the criteria specified in the
    entry_search_criteria_ptr and skipping entries for local host
    servers. */

status = nx_nat_table_find_entry(nat_ptr, &entry_search_criteria_ptr,
                                 &matching_entry_ptr, NX_TRUE);

/* If the search completed with no errors, status = NX_SUCCESS. */
```

See Also

nx_nat_create, nx_nat_delete, nx_nat_table_create, nx_nat_table_delete, nx_nat_table_entry_create, nx_nat_table_entry_preload, nx_nat_table_entry_delete

nx_nat_utility_display_arp_table

Display a native NetX ARP table for the specified IP instance

Prototype

```
UINT    nx_nat_utility_display_arp_table(NX_IP *ip_ptr)
```

Description

This service displays the ARP table entries of the specified IP instance for troubleshooting and debugging purposes.

Input Parameters

ip_ptr	Pointer to IP instance for theNAT network ARP table to display

Return Values

NX_SUCCESS	(0x00)	ARP table was successfully displayed
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Threads

Example

```
/* Display ARP entries for NAT's global IP interface. */
status = nx_nat_utility_display_arp_table(nat_ptr -> nat_global_ip_ptr);

/* If the table was successfully displayed, status = NX_SUCCESS. */
```

See Also

nx_nat_utility_display_bytepool_reserves,
nx_nat_utility_display_packetpool_reserves,
nx_nat_utility_display_translation_table

nx_nat_utility_display_bytepool_reserves

Display NAT byte pool reserves

Prototype

```
UINT    nx_pop3_mail_create(NX_NAT_DEVICE *nat_ptr)
```

Description

This service displays the total number of bytes in the NAT byte pool, the current available remaining bytes for memory allocation, and the number of fragments in the byte pool.

Input Parameters

nat_ptr	Pointer to NAT instance

Return Values

NX_SUCCESS	**(**0x00)	Byte pool data successfully displayed
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Threads

Example

```
/* Display the NAT byte pool data. */
status = nx_nat_utility_display_bytepool_reserves(nat_ptr);

/* If the data was successfully displayed, status = NX_SUCCESS. */
```

See Also

nx_nat_utility_display_arp_table, nx_nat_utility_display_packetpool_reserves, nx_nat_utility_display_translation_table

nx_nat_utility_display_packetpool_reserves

Display packet pool reserves for in and outbound packets

Prototype

```
UINT    nx_nat_utility_private_packetpool_reserves(NX_NAT_DEVICE *nat_ptr)
```

Description

This service displays the total number of packets, and available packets in both the NAT private IP packet pool and global IP packet pools.

Input Parameters

nat_ptr	Pointer to NAT instance

Return Values

NX_SUCCESS	**(**0x00)	Byte pool data successfully displayed
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Threads

Example

```
/* Display the NAT packet pool data for private and global IP interfaces. */
status = nx_nat_utility_display_packetpool_reserves(nat_ptr);

/* If the NAT private and global packet pool data was successfully displayed, status =
NX_SUCCESS. */
```

See Also

nx_nat_utility_display_arp_table, nx_nat_utility_display_bytepool_reserves, nx_nat_utility_display_translation_table

nx_nat_utility_display_translation_table

Display entries in the NAT translation table

Prototype

```
UINT    nx_nat-utility_display_translation_table(NX_NAT_DEVICE *nat_ptr)
```

Description

This service displays the entries in the NAT translation table, including both static and dynamically created entries, and for the latter the time remaining before their timeout period expires.

Input Parameters

Nat_ptr	Pointer to the NAT instance

Return Values

NX_SUCCESS	(0x00)	Translation table successfully displayed
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Threads

Example

```
/* Display the NAT translation table entries. */
status = nx_nat_utility_display_tranlsation_table(nat_ptr);

/* If the table was successfully displayed, status = NX_SUCCESS. */
```

See Also

nx_nat_utility_display_arp_table, nx_nat_utility_display_bytepool_reserves, nx_nat_utility_display_packetpool_reserves

nx_nat_utility_get_destination_port

Parse destination port from packet subprotocol header

Prototype

```
UINT    nx_nat_utility_destination_port(NX_PACKET *packet_ptr,
                                        UINT protocol, UINT *external_port)
```

Description

This service parses the destination port from the specified packet protocol header. For ICMP packets, it parses the ICMP query ID. If the packet is an ICMP error message packet, a zero port number is returned.

Input Parameters

packet_ptr	Pointer to packet to parse
protocol	Packet subprotocol (e.g. TCP)
external_port	Pointer to port number parsed

Return Values

NX_SUCCESS	(0x00)	Port number successfully parsed
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Threads, Application code

Example

```
/* Parse the destination port number of the specified TCP packet. */
status = nx_nat_utility_get_destination_port (packet_ptr, NX_IP_TCP, &external_port);

/* If the port was successfully parsed, status = NX_SUCCESS. */
```

See Also

nx_nat_utility_source_port

nx_nat_utility_get_source_port

Parse source port from packet subprotocol header

Prototype

```
UINT    nx_nat_utility_source_port(NX_PACKET *packet_ptr,
                                   UINT protocol, UINT *source_port)
```

Description

This service parses the source port from the specified packet protocol header. For ICMP packets, it parses the ICMP query ID. If the packet is an ICMP error message packet, a zero port number is returned.

Input Parameters

packet_ptr	Pointer to packet to parse
protocol	Packet subprotocol (e.g. TCP)
source_port	Pointer to port number parsed

Return Values

NX_SUCCESS	(0x00)	Port number successfully parsed
NX_PTR_ERROR	(0x16)	Invalid input pointer parameter

Allowed From

Threads, Application code

Example

```
/* Parse the source port number of the specified TCP packet. */
status = nx_nat_utility_get_source_port (packet_ptr, NX_IP_TCP, &external_port);

/* If the port was successfully parsed, status = NX_SUCCESS. */
```

See Also

nx_nat_utility_destination_port

Appendix A

Example with Netx NAT callback functions

```
221  /* Set the upper limit on assigned port numbers. */
222  #define NX_NAT_MAXIMUM_ASSIGNED_PORTID_NUMBER 65535
223
224  /* Set a starting value for available (not bound to TCP/UDP socket) port numbers. */
```

```
225  #define NX_NAT_ASSIGNED_SOURCE_PORTID_OFFSET  30000
226
227  /*
228     This is used by NAT to obtain an unused port number for a unique global IP
229     address:port for an outbound packet.  If a suitable port cannot be found, a port
230     value of zero is returned.
233  */

234  UINT get_available_port(NX_NAT_TRANSLATION_TABLE *nat_table_ptr, UINT protocol,
                         ULONG global_inside_ip_address,  UINT source_port, UINT
                         *assigned_port)
235  {
236
237
238  UINT                        status;
239  NX_NAT_TRANSLATION_ENTRY    *entry_ptr;
240
241
242      /* Initialized the port to zero, indicating available port not found. */
243      *assigned_port = 0;
244
245      /* Gain exlusive access to the translation table. */
246      status = tx_mutex_get(&nat_table_ptr -> table_mutex, nat_table_ptr ->
                              table_mutex_timeout);
247
248      /* Check for error. */
249      if (status != TX_SUCCESS)
250      {
251
252          /* Return error status. */
253          return status;
254      }
255
256      /* Reset the source port to by some offset from the local hosts. */
257      source_port += NX_NAT_ASSIGNED_SOURCE_PORTID_OFFSET;
258
259      /* Special case: translation table is empty. */
260      if (nat_table_ptr -> table_entries == 0)
261      {
262
263          *assigned_port = source_port;
264
265          /* Release the lock on the table. */
266          tx_mutex_put(&nat_table_ptr -> table_mutex);
267
268          /* Return successful completion. */
269          return NX_SUCCESS;
270      }
271
272      entry_ptr = nat_table_ptr -> start_table_entry_ptr;
273      while (entry_ptr)
274      {
275
276          /* Skip entries for designated servers. */
277          if (!entry_ptr -> inbound_packet_initiated)
278          {
279
280              /* Is there already a translation table entry for this packet? */
281              if (source_port == entry_ptr -> global_inside_port &&
282                   global_inside_ip_address == entry_ptr -> global_inside_ip_address)
283              {
284
285                  /* Yes so increase the port number. */
286                  source_port++;
287
288                  /* Is there an upper limit on port number?
289                     It is more likely that the table might reach
290                     its limit on the number of entries allowed first! */
291                  if (source_port > NX_NAT_MAXIMUM_ASSIGNED_PORTID_NUMBER)
292                  {
293
294                      /* Indicate the upper limit on assigned port numbers is reached.
*/
295                      tx_mutex_put(&nat_table_ptr -> table_mutex);
296
297                      /* Return the error status. */
298                      return NX_NAT_NO_FREE_PORT_AVAILABLE;
299                  }
300
301                  /* Restart the search at the first entry with the next port. */
302                  entry_ptr = nat_table_ptr -> start_table_entry_ptr;
```

```
303             }
304         }
305
306         /* Was this is the last entry to check in the table? */
307         if (entry_ptr == nat_table_ptr -> end_table_entry_ptr)
308         {
309
310             /* Yup. Then we have found an available port number. */
311             *assigned_port = source_port;
312             break;
313         }
314
315         /* Get the next entry and search on the same port number. */
316         entry_ptr = entry_ptr -> next_entry_ptr;
317     }
318
319     /* Release the lock on the translation table. */
320     tx_mutex_put(&nat_table_ptr -> table_mutex);
321
322     /* Return successful conclusion. */
323     return NX_SUCCESS;
324 }
325
326 /*
327    This is used by NAT to assign a global IP address for a host on a private network
328    wanting to send packets to a destination on the external network. Note that the
329    protocol argument is not used in this particular callback, but it is included for
330    the convenience of a protocol specific algorithm for finding an available port.
331    If this function is not able to find a suitable port, a port value of zero is
332    returned.
333 */
334 UINT get_available_address(NX_NAT_TRANSLATION_TABLE *nat_table_ptr, UINT protocol,
335                            ULONG *available_global_ip_address, ULONG
private_inside_ip_address,
336                            ULONG destination_ip_address)
337 {
338
339 UINT                            status;
340 UINT                            global_ip_is_in_use;
341 NX_NAT_DEVICE                   *nat_ptr;
342 NX_NAT_TRANSLATION_ENTRY        *entry_ptr;
343 NX_NAT_RESERVED_IP_ITEM         *reserved_ip_ptr;
344
345
346     /* Set local pointer for convenience. */
347     nat_ptr = nat_table_ptr -> nat_ptr;
348
349     /* Gain exlusive access to the translation table. */
350     status = tx_mutex_get(&nat_table_ptr -> table_mutex, nat_table_ptr ->
    table_mutex_timeout);
351
352     /* Check for error. */
353     if (status != TX_SUCCESS)
354     {
355
356         /* Return error status. */
357         return status;
358     }
359
360     /* Start with the first reserved IP address NAT has. */
361     reserved_ip_ptr = nat_ptr -> start_reserved_ip_item_ptr;
362
363     /* Go through all NAT's reserved IP addresses. */
364     while (reserved_ip_ptr)
365     {
366
367         /* Get the actual IP address from the reserved listing. */
368         *available_global_ip_address = reserved_ip_ptr -> nx_nat_ip_address;
369
370         /* Make sure it is not the same IP address as the destination address, which
can
371            happen if a local host sends a packet to another local host using
372            a global IP address. */
373         if (*available_global_ip_address == destination_ip_address)
374         {
375
376             /* Get the next reserved global address. */
377             reserved_ip_ptr = reserved_ip_ptr -> next_ptr;
378
379             /* Reset translation table pointer to the first entry. */
380             entry_ptr = nat_table_ptr -> start_table_entry_ptr;
```

```
381
382             continue;
383         }
384
385         /* We next need to check the translation table if this
386            reserved IP address is currently in use by another host connection. */
387
388         /* Special case is if the table is empty (skip the search!) */
389         if (nat_table_ptr -> table_entries == 0)
390         {
391             /* No need to search the table! */
392             break;
393         }
394
395         /* Start at the first entry in the translation table. */
396         entry_ptr = nat_table_ptr -> start_table_entry_ptr;
397
398         /* Assume unavailable unless a search of the translation table shows it
399            is not being used. */
400         global_ip_is_in_use = NX_FALSE;
401
402         /* Search thru the whole table until a match is found. */
403         while (entry_ptr)
404         {
405
406             /* Does it match this table entry IP address? */
407             if (*available_global_ip_address == entry_ptr -> global_inside_ip_address)
408             {
409
410                 global_ip_is_in_use = NX_TRUE;
411
412                  /* Skip on to the next global IP in the pool. */
413                  break;
414             }
415
416             /* Get the next entry in the table. */
417             entry_ptr = entry_ptr -> next_entry_ptr;
418         }
419
420         /* Did we find a matching IP address entry in the translation table? */
421         if (!global_ip_is_in_use)
422         {
423
424             /* No. Release the table and return with a NULL reserved IP address. */
425             tx_mutex_put(&nat_table_ptr -> table_mutex);
426
427             return NX_SUCCESS;
428         }
429
430         /* Get the next IP address item in the reserved list. */
431         reserved_ip_ptr = reserved_ip_ptr -> next_ptr;
432
433         /* Reset translation table pointer to the first entry. */
434         entry_ptr = nat_table_ptr -> start_table_entry_ptr;
435     }
436
437
438     /* If we got here, no reserved IP addresses are available. */
439
440     /* Log the event and bail out. */
441     NX_NAT_EVENT_LOG(MODERATE, ("No global IP addresses are currently available for
442                                  host 0x%x. \r\n", private_inside_ip_address));
443
444     /* Release exlusive access to the translation table. */
445     tx_mutex_put(&nat_table_ptr -> table_mutex);
446
447     /* Return successful status. */
448     return NX_SUCCESS;
449 }
450
451
452 /* This function determines a translation table entry timeout by protocol, source,
453    destination, etc. Application (e.g. HTTP, Telnet) and IP header SYN, RST and FIN
454    bit fields can also be used to determine an entry timeout.
455 */
456 VOID set_table_entry_timeout(NX_PACKET *packet_ptr, UINT packet_direction,
457                              UINT *response_timeout)
458 {
459
460 UINT            protocol;
461 NX_IP_HEADER    *ip_header_ptr;
```

```
462  NX_TCP_HEADER   *tcp_header_ptr;
463
464
465      /* Get a pointer to the IP header. */
466      ip_header_ptr = (NX_IP_HEADER *) packet_ptr -> nx_packet_prepend_ptr;
467
468      /* Pickup the pointer to the head of the TCP packet.  */
469      tcp_header_ptr =  (NX_TCP_HEADER *) (packet_ptr -> nx_packet_prepend_ptr +
                                              sizeof(NX_IP_HEADER));
470
471      /* Adjust header data for Endianness. */
472      NX_CHANGE_ULONG_ENDIAN(ip_header_ptr -> nx_ip_header_word_2);
473
474      /* Determine what protocol the current IP datagram is.  */
475      protocol =  ip_header_ptr -> nx_ip_header_word_2 & NX_IP_PROTOCOL_MASK;
476
477      /* Set the timeout in seconds per network protocol. */
478      switch (protocol)
479      {
480          case NX_IP_TCP:
481              *response_timeout = 30;
482
483              /* Shorten the timeout for this packet if the FIN or RST bit set. */
484              if ((tcp_header_ptr -> nx_tcp_header_word_3 & NX_TCP_FIN_BIT) ||
485                  (tcp_header_ptr -> nx_tcp_header_word_3 & NX_TCP_RST_BIT))
486              {
487
488                  *response_timeout = 20;
489              }
490
491              break;
492
493          case NX_IP_UDP:
494              *response_timeout = 30;
495              break;
496
497          case NX_IP_ICMP:
498              *response_timeout = 60;
499              break;
500
501          default:
502              /* Unknown or unsupported protocol. */
503
504              /* Log the event and bail out. */
505              NX_NAT_EVENT_LOG(MODERATE, ("Unsupported packet protocol (0x%x) for “
506                                          “setting packet timeout. \r\n", protocol));
507
508              *response_timeout = 0;
509      }
510
511      return;
512  }
513
514
515  /*
516     Callback for filtering inbound and/or outbound TCP packets.  This example
517     callback is merely demonstrates that filters are bi-directional.
518  */
519  VOID nat_tcp_filter(NX_PACKET *packet_ptr, UINT incoming_outgoing_direction, UINT
                            *passed_filter)
520  {
521
522      /* Apply network 'rules' for NAT forwarding TCP packets between
523         external and local hosts. Below the filter allows all
524         packets in either direction to be forwarde by NAT. */
525      if (incoming_outgoing_direction == NX_NAT_INBOUND_PACKET)
526      {
527          /* Do TCP filtering for inbound packets. */
528          *passed_filter = NX_TRUE;
529      }
530      else
531      {
532          /* Do TCP filtering for outbound packets. */
533          *passed_filter = NX_TRUE;
534      }
535
536      return;
537  }
538
539
540  /*
```

```
541     Callback for filtering inbound and/or outbound UDP packets.  This example
542     callback is merely demonstrates that filters are bi-directional.
543  */
544  VOID nat_udp_filter(NX_PACKET *packet_ptr, UINT incoming_outgoing_direction, UINT
                         UINT *passed_filter)
545  {
546
547      /* Apply network 'rules' for NAT forwarding UDP packets between
548         external and local hosts. Below the filter allows all
549         packets in either direction to be forwarde by NAT. */
550      if (incoming_outgoing_direction == NX_NAT_INBOUND_PACKET)
551      {
552          /* Do UDP filtering for inbound packets. */
553          *passed_filter = NX_TRUE;
554      }
555      else
556      {
557          /* Do UDP filtering for outbound packets. */
558          *passed_filter = NX_TRUE;
559      }
560
561      return;
562  }
563
564
565  /*
566     Callback for filtering inbound and/or outbound ICMP packets.  This example
567     callback is merely demonstrates that filters are bi-directional.
568  */
569  VOID nat_icmp_filter(NX_PACKET *packet_ptr, UINT incoming_outgoing_direction,
                         UINT *passed_filter)
570  {
571
572      /* Apply network 'rules' for NAT forwarding ICMP packets between
573         external and local hosts. Below the filter allows all
574         packets in either direction to be forwarde by NAT. */
575
576      if (incoming_outgoing_direction == NX_NAT_INBOUND_PACKET)
577      {
578          /* Do ICMP filtering for inbound packets. */
579          *passed_filter = NX_TRUE;
580      }
581      else
582      {
583          /* Do ICMP filtering for outbound packets. */
584          *passed_filter = NX_TRUE;
585      }
586
587      return;
588  }
589
```