Microsoft

Azure RTOS NetX Duo DNS User Guide

Published: February 2020

For the latest information, please see
azure.com/rtos

Part Number: 000-1051

Revision 6.0

Contents

Chapter 1

Introduction to the NetX Duo DNS Client

The DNS provides a distributed database that contains mapping between domain names and physical IP addresses. The database is referred to as *distributed* because there is no single entity on the Internet that contains the complete mapping. An entity that maintains a portion of the mapping is called a DNS Server. The Internet is composed of numerous DNS Servers, each of which contains a subset of the database. DNS Servers also respond to DNS Client requests for domain name mapping information, only if the server has the requested mapping.

The DNS Client protocol for NetX Duo provides the application with services to request mapping information from one or more DNS Servers.

DNS Client Setup

In order to function properly, the DNS Client package requires that a NetX Duo IP instance has already been created.

After creating the DNS Client, the application must add one or more DNS servers to the server list maintained by the DNS Client. To add DNS servers, the application uses the *nxd_dns_server_add* service. The NetX Duo DNS Client service *nx_dns_server_add* can also be used to add servers. However it only accepts IPv4 addresses and it is recommended that developers use the *nxd_dns_server_add* service instead.

If the NX_DNS_IP_GATEWAY_SERVER option is enabled, and the IP instance gateway address is non zero, the IP instance gateway is automatically added as the primary DNS server. If DNS server information is not statically known, it may also be derived through the Dynamic Host Configuration Protocol (DHCP) for NetX Duo. Please refer to the NetX Duo DHCP User Guide for more information.

The DNS Client requires a packet pool for transmitting DNS messages. By default, the DNS Client creates this packet pool when the *nx_dns_create* service is called. The configuration options NX_DNS_PACKET_PAYLOAD_UNALIGNED and NX_DNS_PACKET_POOL_SIZE allow the application to determine the packet payload and packet pool size (e.g. number of packets) of this packet pool respectively. These options are described in section “Configuration Options” in Chapter Two.

An alternative to the DNS Client creating its own packet pool is for the application to create the packet pool and set it as the DNS Client's packet pool using the *nx_dns_packet_pool_set* service. To do so, the NX_DNS_CLIENT_USER_CREATE_PACKET_POOL option must be defined. This option also requires a previously created packet pool using *nx_packet_pool_create* as the packet pool pointer input to *nx_dns_packet_pool_set* . When the DNS Client instance is deleted, the application is responsible for deleting the DNS Client packet pool if NX_DNS_CLIENT_USER_CREATE_PACKET_POOL is enabled if it is no longer needed.

Note: For applications choosing to provide its own packet pool using the NX_DNS_CLIENT_USER_CREATE_PACKET_POOL option, the packet size needs to be able to hold the DNS maximum massage size (512 bytes) plus rooms for UDP header, IPv4 or IPv6 header, and the MAC header.

DNS Messages

The DNS has a very simple mechanism for obtaining mapping between host names and IP addresses. To obtain a mapping, the DNS Client prepares a DNS query message containing the name or the IP address that needs to be resolved. The message is then sent to the first DNS server in the server list. If the server has such a mapping, it replies to the DNS Client using a DNS response message that contains the requested mapping information. If the server does not respond, the DNS Client queries the next server on its list until all its DNS servers have been queried. If no response from all its DNS servers is received, the DNS Client has retry logic to retransmit the DNS message. On resending a DNS query, the retransmission timeout is doubled. This process continues until the maximum transmission timeout (defined as NX_DNS_MAX_RETRANS_TIMEOUT in *nxd_dns.h*) is reached or until a successful response is received from that server is obtained.

NetX Duo DNS Client can perform both IPv6 address lookups (type AAAA) and IPv4 address lookups (type A) by specifying the version of the IP address in the *nxd_dns_host_by_name_get* call. The DNS Client can perform reverse lookups of IP addresses (type PTR queries) to obtain web host names using *nxd_dns_host_by_address_get*. The NetX Duo DNS Client still supports the *nx_dns_host_by_name_get* and *nx_dns_host_by_address_get* which are the equivalent services but which are limited to IPv4 network communication. However, developers are encouraged to port existing DNS Client applications to the *nxd_dns_host_by_name_get* and *nxd_dns_host_by_address_get* services.

DNS messaging utilizes the UDP protocol to send requests and field responses. A DNS Server listens on port number 53 for queries from clients. Therefore UDP services must be enabled in NetX Duo using the ***nx_udp_enable*** service on a previously created IP instance (***nx_ip_create***).

At this point, the DNS Client is ready to accept requests from the application and send out DNS queries.

Extended DNS Resource Record Types

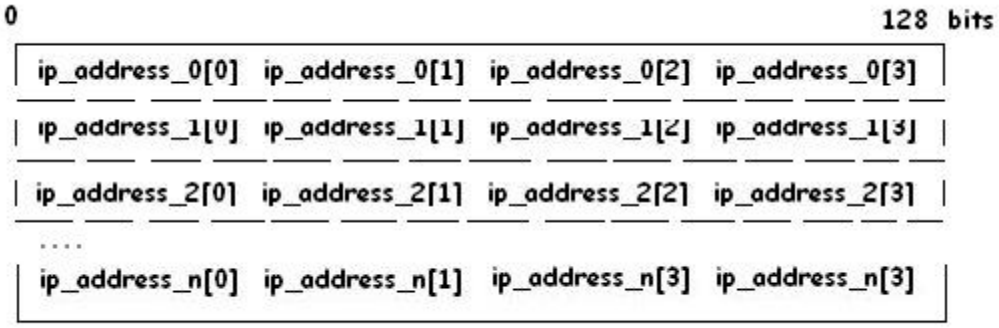
If NX_DNS_ENABLE_EXTENDED_RR_TYPES is enabled, NetX Duo DNS Client also supports the following record type queries:

CNAME	contains the canonical name for an alias
TXT	contains a text string
NS	contains an authoritative name server
SOA	contains the start of a zone of authority
MX	used for mail exchange
SRV	contains information on the service offered by the domain

With the exception of CNAME and TXT record types, the application must supply a 4-byte aligned buffer to receive the DNS data record.

In NetX Duo DNS Client, record data is stored in such a way to make most efficient use of buffer space.

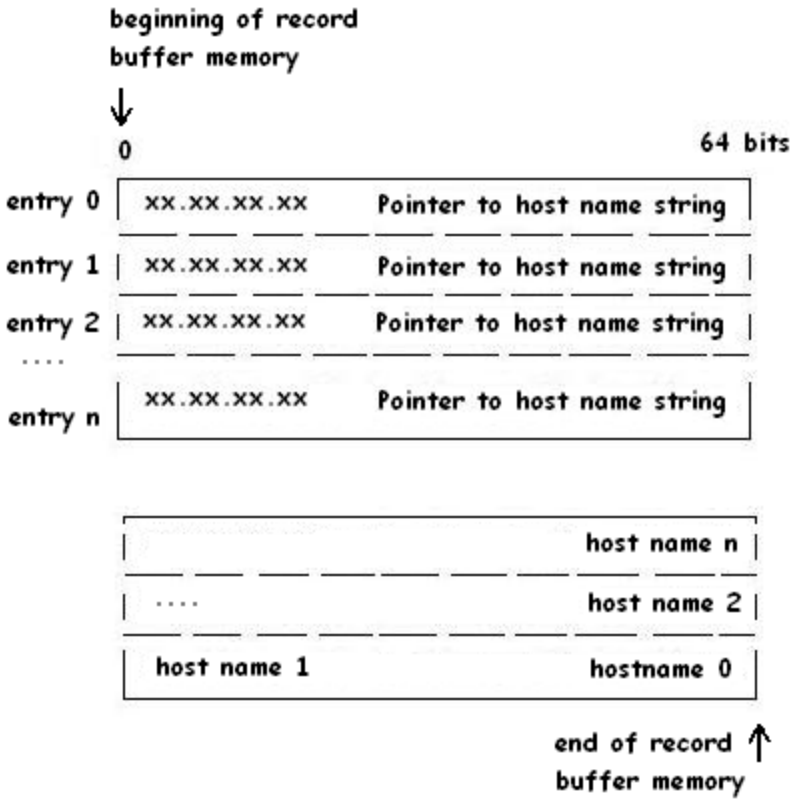
An example of a record buffer of fixed length (type AAAA record) is shown below:



For those queries whose record types have variable data length, such as NS records whose host names are of variable length, NetX Duo DNS Client saves the data as follows. The buffer supplied in the DNS Client query is

organized into an area of fixed length data and an area of unstructured memory. The top of the memory buffer is organized into 4-byte aligned record entries. Each record entry contains the IP address and a pointer to the variable length data for that IP address. The variable length data for each IP address are stored in the unstructured area memory starting at the end of the memory buffer. The variable length data for each successive record entry is saved in the next area memory adjacent to the previous record entries variable data. Hence, the variable data 'grows' towards the structured area of memory containing the record entries until there is insufficient memory to store another record entry and variable data.

This is shown in the figure below:



The example of the DNS domain name (NS) data storage is shown above.

NetX Duo DNS Client queries using the record storage format return the number of records saved to the record buffer. This information enables the application to extract NS records from the record buffer.

An example of a DNS Client query that stores variable length DNS data using this record storage format is shown below:

```
UINT  _nx_dns_domain_name_server_get(NX_DNS *dns_ptr,
                    UCHAR *host_name, VOID *record_buffer,
                    UINT buffer_size, UINT *record_count,
                    ULONG wait_option)
```

More details are available in Chapter 3, “Description of DNS Client Services”.

DNS Cache

If NX_DNS_CACHE_ENABLE is enabled, NetX Duo DNS Client supports the DNS Cache feature. After creating the DNS Client, the application can call the API *nx_dns_cache_initialize()* to set the special DNS Cache. If enable DNS Cache feature, DNS Client will find the available answer from DNS Cache before starts to send DNS query, if find the available answer, directly return the answer to application, otherwise DNS Client sends out query message to DNS server and waits for the reply. When DNS Client gets the response message and there is free cache available, DNS Client returns the answer to the application and also adds the answer as resource record into DNS cache.

Each answer a data structure *NX_DNS_RR* (Resource Record) in the cache. Strings (resource record name and data) in Records are variable length, therefore are not stored in the NX_DNS_RR structure. The Record contains pointers to the actual memory location where the strings are stored. The string table and the Records share the cache. Records are stored from the beginning of the cache, and grow towards the end of the cache. The string table starts from the end of the cache and grows towards the beginning of the cache. Each string in the string table has a length field and a counter field. When a string is added to the string table, if the same string is already present in the table, the counter value is incremented and no memory is allocated for the string. The cache is considered full if no more resource records or new strings can be added to the cache.

DNS Client Limitations

The DNS Client supports one DNS request at a time. Threads attempting to make another DNS request are temporarily blocked until the previous DNS request is complete.

The NetX Duo DNS Client does not use data from authoritative answers to forward additional DNS queries to other DNS servers.

DNS RFCs

NetX Duo DNS is compliant with the following RFCs:

RFC1034 DOMAIN NAMES - CONCEPTS AND FACILITIES
RFC1035 DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION
RFC1480 The US Domain
RFC 2782 A DNS RR for specifying the location of services (DNS SRV)
RFC 3596 DNS Extensions to Support IP Version 6

Chapter 2

Installation and Use of NetX Duo DNS Client

This chapter contains a description of various issues related to installation, setup, and usage of the NetX Duo DNS Client.

Product Distribution

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

nxd_dns.h	Header file for NetX Duo DNS Client
nxd_dns.c	C Source file for NetX Duo DNS Client
nxd_dns.pdf	PDF description of NetX Duo DNS Client

DNS Client Installation

To use NetX Duo DNS Client, copy the source code files *nxd_dns.c* and *nxd_dns.h* to the same directory where NetX Duo is installed. For example, if NetX Duo is installed in the directory "*\threadx\arm7\green*" then the *nxd_dns.h* and *nxd_dns.c* files should be copied into this directory.

Using the DNS Client

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

Note that since DNS utilizes NetX Duo UDP services, UDP must be enabled with the *nx_udp_enable* call prior to using DNS.

Small Example System for NetX Duo DNS Client

NetX Duo DNS Client is compatible with existing NetX DNS applications. The list of legacy services and their NetX Duo equivalent is shown below:

NetX DNS API service (IPv4 only)	**NetX Duo DNS API service (IPv4 and IPv6 supported)**
nx_dns_host_by_name_get	nxd_dns_host_by_name_get
nx_dns_host_by_address_get	nxd_dns_host_by_address_get
nx_dns_server_get	nxd_dns_server_get
nx_dns_server_add	nxd_dns_server_add
nx_dns_server_remove	nxd_dns_server_remove

See the description of NetX Duo DNS Client API services in Chapter 3 for more details.

In the example DNS application program provided in this section, *nxd_dns.h* is included at line 6. NX_DNS_CLIENT_USER_CREATE_PACKET_POOL, which allows the DNS Client application to create the packet pool for the DNS Client, is declared on lines 24-26. This packet pool is used for allocating packets for sending DNS messages. If NX_DNS_CLIENT_USER_CREATE_PACKET_POOL is defined, a packet pool is created in lines 78-98. If this option is not enabled, the DNS Client creates its own packet pool as per the packet payload and pool size set by configuration parameters in *nxd_dns.h* and described elsewhere in this chapter.

Another packet pool is created in lines 100-112 for the Client IP instance which is used for internal NetX Duo operations. Next the IP instance is created using the *nx_ip_create* call in line 115. It is possible for the IP task and the DNS Client to share the same packet pool, but since the DNS Client typically sends out larger messages than the control packets sent by the IP task, using separate packet pools makes more efficient use of memory.

ARP and UDP (which is used by IPv4 networks) are enabled in lines 129 and 141 respectively.

Note this demo uses the ‘ram’ driver declared on line 44 and used in the *nx_ip_create* call. This ram driver is distributed with the NetX Duo source code. To actually run the DNS Client the application must supply an actual physical network driver to transmit and receive packets from the DNS server.

The Client thread entry function *thread_client_entry* is defined below the *tx_application_define* function. It initially relinquishes control to the system to allow the IP task thread to be initialized by the network driver.

It then creates the DNS Client in line 257, initializes the DNS cache on lines 267- 278, and sets the packet pool previously created to the DNS Client instance on lines 281-295. It then adds DNS server on lines 297-341.

The remainder of the example program uses the DNS Client services to make DNS queries. Host IP address lookups are performed on lines 193 and 207. The difference between these two services, *nxd_dns_host_by_name_get* and *nx_dns_host_by_name_get*, is that the former saves the address data in an NXD_ADDRESS data type, while the latter saves the data in a ULONG data type. Further the latter is limited to IPv4 networks, while the former can be used with IPv6 or IPv4 networks. This is only possible if the IP instance is enabled for IPv6. See the NetX Duo User Guide for more details on enabling the IP instance for IPv6 networking.

Another service for host IP address lookups is shown on line 464, *nx_dns_ipv4_address_by_name_get*. This service differs from *nx_dns_host_by_name_get* in that it returns all (or as many will fit in the supplied buffer) of the IPv4 addresses discovered for the domain name, not just the first address received in the DNS Server reply.

Similarly, the *nxd_dns_ipv6_address_by_name_get* service, called on line 380, returns all the IPv6 addresses discovered by the DNS Client, not just the first one.

Reverse lookups (host name from IP address) are performed on lines 606 (*nx_dns_host_by_address_get*) and again on line 564 and 588 (*nxd_dns_host_by_address_get*). *nx_dns_host_by_address_get* will only work on IPv4 networks, while *nxd_dns_host_by_address_get* will work on either IPv4 or IPv6 networks (e.g. the IP instance is enabled for IPv6 as well as IPv4 networks).

Two more services for DNS lookups, CNAME and TXT, are demonstrated on lines 627 and 649 respectively, to discover CNAME and domain name data for the input domain name. NetX Duo DNS Client as similar services for other record types, e.g. MX, NS, SRV and SOA. See Chapter 3 for detailed descriptions of all record type lookups available in NetX Duo DNS Client.

When the DNS Client is deleted on line 846, using the *nx_dns_delete* service, the packet pool for the DNS Client is not deleted unless the DNS Client created its own packet pool. Otherwise, it is up to the application to delete the packet pool if it has no further use for it.

```
1 /* This is a small demo of DNS Client for the high-performance NetX TCP/IP stack.
*/
 2
 3 #include   "tx_api.h"
 4 #include   "nx_api.h"
 5 #include   "nx_udp.h"
 6 #include   "nxd_dns.h"
 7
```

```
 8 #ifdef FEATURE_NX_IPV6
 9 #include   "nx_ipv6.h"
10 #endif
11
12 #define     DEMO_STACK_SIZE         4096
13
14 #define     NX_PACKET_PAYLOAD       1536
15 #define     NX_PACKET_POOL_SIZE     30 * NX_PACKET_PAYLOAD
16 #define     LOCAL_CACHE_SIZE        2048
17
18 /* Define the ThreadX and NetX object control blocks...  */
19
20 NX_DNS                  client_dns;
21 TX_THREAD               client_thread;
22 NX_IP                   client_ip;
23 NX_PACKET_POOL          main_pool;
24 #ifdef NX_DNS_CLIENT_USER_CREATE_PACKET_POOL
25 NX_PACKET_POOL          client_pool;
26 #endif
27 UCHAR                   local_cache[LOCAL_CACHE_SIZE];
28
29 UINT                    error_counter = 0;
30
31 #ifdef FEATURE_NX_IPV6
32 /* If IPv6 is enabled in NetX Duo, allow DNS Client to try using IPv6 */
33 //#define USE_IPV6
34 #endif
35
36 #define CLIENT_ADDRESS      IP_ADDRESS(192,168,0,11)
37 #define DNS_SERVER_ADDRESS  IP_ADDRESS(192,168,0,1)
38
39 /* Define thread prototypes.  */
40
41 void    thread_client_entry(ULONG thread_input);
42
43 /***** Substitute your ethernet driver entry function here *********/
44 extern  VOID _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr);
45
46
47 /* Define main entry point.  */
48
49 int main()
50 {
51
52     /* Enter the ThreadX kernel.  */
53     tx_kernel_enter();
54 }
55
56
57 /* Define what the initial system looks like.  */
58
59 void    tx_application_define(void *first_unused_memory)
60 {
61
62 CHAR    *pointer;
63 UINT    status;
64
65
66     /* Setup the working pointer.  */
67     pointer =  (CHAR *) first_unused_memory;
68
69     /* Create the main thread.  */
70     tx_thread_create(&client_thread, "Client thread", thread_client_entry, 0,
71            pointer, DEMO_STACK_SIZE, 4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);
72
73     pointer =  pointer + DEMO_STACK_SIZE;
74
75     /* Initialize the NetX system.  */
76     nx_system_initialize();
77
78 #ifdef NX_DNS_CLIENT_USER_CREATE_PACKET_POOL
79
80     /* Create the packet pool for the DNS Client to send packets.
81
82         If the DNS Client is configured for letting the host application create
83         the DNS packet pool, (see NX_DNS_CLIENT_USER_CREATE_PACKET_POOL option),
see
84        nx_dns_create() for guidelines on packet payload size and pool size.
85        packet traffic for NetX processes.
86     */
```

```
 87     status =  nx_packet_pool_create(&client_pool, "DNS Client Packet Pool",
NX_DNS_PACKET_PAYLOAD, pointer, NX_DNS_PACKET_POOL_SIZE);
 88
 89     pointer = pointer + NX_DNS_PACKET_POOL_SIZE;
 90
 91     /* Check for pool creation error.  */
 92     if (status)
 93     {
 94
 95         error_counter++;
 96         return;
 97      }
 98 #endif
 99
100     /* Create the packet pool which the IP task will use to send packets. Also
available to the host
101        application to send packet. */
102     status =  nx_packet_pool_create(&main_pool, "Main Packet Pool",
NX_PACKET_PAYLOAD, pointer, NX_PACKET_POOL_SIZE);
103
104     pointer = pointer + NX_PACKET_POOL_SIZE;
105
106     /* Check for pool creation error.  */
107     if (status)
108     {
109
110         error_counter++;
111         return;
112      }
113
114     /* Create an IP instance for the DNS Client.  */
115     status = nx_ip_create(&client_ip, "DNS Client IP Instance", CLIENT_ADDRESS,
0xFFFFFF00UL,
116                           &main_pool, _nx_ram_network_driver, pointer, 2048, 1);
117
118     pointer =  pointer + 2048;
119
120     /* Check for IP create errors.  */
121     if (status)
122     {
123
124         error_counter++;
125         return;
126      }
127
128     /* Enable ARP and supply ARP cache memory for the DNS Client IP.  */
129     status =  nx_arp_enable(&client_ip, (void *) pointer, 1024);
130     pointer = pointer + 1024;
131
132     /* Check for ARP enable errors.  */
133     if (status)
134     {
135
136         error_counter++;
137         return;
138      }
139
140     /* Enable UDP traffic because DNS is a UDP based protocol.  */
141     status =  nx_udp_enable(&client_ip);
142
143     /* Check for UDP enable errors.  */
144     if (status)
145     {
146
147         error_counter++;
148         return;
149      }
150 }
151
152 #define BUFFER_SIZE     200
153 #define RECORD_COUNT    10
154
155 /* Define the Client thread.  */
156
157 void    thread_client_entry(ULONG thread_input)
158 {
159
160 UCHAR           record_buffer[200];
161 UINT            record_count;
162 UINT            status;
163 ULONG           host_ip_address;
```

```
#ifdef FEATURE_NX_IPV6
NXD_ADDRESS     host_ipduo_address;
NXD_ADDRESS     test_ipduo_server_address;
#ifdef USE_IPV6
NXD_ADDRESS     client_ipv6_address;
NXD_ADDRESS     dns_ipv6_server_address;
UINT            iface_index, address_index;
#endif
#endif
UINT            i;
ULONG           *ipv4_address_ptr[RECORD_COUNT];
NX_DNS_IPV6_ADDRESS
                *ipv6_address_ptr[RECORD_COUNT];
#ifdef NX_DNS_ENABLE_EXTENDED_RR_TYPES
NX_DNS_NS_ENTRY
                *nx_dns_ns_entry_ptr[RECORD_COUNT];
NX_DNS_MX_ENTRY
                *nx_dns_mx_entry_ptr[RECORD_COUNT];
NX_DNS_SRV_ENTRY
                *nx_dns_srv_entry_ptr[RECORD_COUNT];
NX_DNS_SOA_ENTRY
                *nx_dns_soa_entry_ptr;
ULONG           host_address;
USHORT          host_port;
#endif

    /* Give NetX IP task a chance to get initialized . */
    tx_thread_sleep(100);

#ifdef FEATURE_NX_IPV6
#ifdef USE_IPV6

    /* Make the DNS Client IPv6 enabled. */
    status = nxd_ipv6_enable(&client_ip);

    /* Check for enable errors.  */
    if (status)
    {

        error_counter++;
        return;
     }
    status = nxd_icmp_enable(&client_ip);

    /* Check for enable errors.  */
    if (status)
    {

        error_counter++;
        return;
    }

    client_ipv6_address.nxd_ip_address.v6[3] = 0x101;
    client_ipv6_address.nxd_ip_address.v6[2] = 0x0;
    client_ipv6_address.nxd_ip_address.v6[1] = 0x0000f101;
    client_ipv6_address.nxd_ip_address.v6[0] = 0x20010db8;
    client_ipv6_address.nxd_ip_version = NX_IP_VERSION_V6;


     /* Set the link local address with the host MAC address. */
    iface_index = 0;

    /* This assumes we are using the primary network interface (index 0). */
    status = nxd_ipv6_address_set(&client_ip, iface_index, NX_NULL, 10,
&address_index);

    /* Check for link local address set error.  */
    if (status)
    {

        error_counter++;
        return;
     }

    /* Set the host global IP address. We are assuming a 64
       bit prefix here but this can be any value (< 128). */
    status = nxd_ipv6_address_set(&client_ip, iface_index, &client_ipv6_address,
64, &address_index);

    /* Check for global address set error.  */
    if (status)
```

```
243     {
244
245         error_counter++;
246         return;
247      }
248
249     /* Wait while NetX Duo validates the link local and global address. */
250     tx_thread_sleep(500);
251 #endif
252 #endif
253
254     /* Create a DNS instance for the Client.  Note this function will create
255        the DNS Client packet pool for creating DNS message packets intended
256        for querying its DNS server. */
257     status =  nx_dns_create(&client_dns, &client_ip, (UCHAR *)"DNS Client");
258
259     /* Check for DNS create error.  */
260     if (status)
261     {
262
263         error_counter++;
264         return;
265      }
266
267 #ifdef NX_DNS_CACHE_ENABLE
268     /* Initialize the cache.  */
269     status = nx_dns_cache_initialize(&client_dns, local_cache, LOCAL_CACHE_SIZE);
270
271     /* Check for DNS cache error.  */
272     if (status)
273     {
274
275         error_counter++;
276         return;
277      }
278 #endif
279
280     /* Is the DNS client configured for the host application to create the pecket
pool? */
281 #ifdef NX_DNS_CLIENT_USER_CREATE_PACKET_POOL
282
283     /* Yes, use the packet pool created above which has appropriate payload size
284        for DNS messages. */
285      status = nx_dns_packet_pool_set(&client_dns, &client_pool);
286
287      /* Check for set DNS packet pool error.  */
288      if (status)
289      {
290
291          error_counter++;
292          return;
293       }
294
295 #endif /* NX_DNS_CLIENT_USER_CREATE_PACKET_POOL */
296
297 #ifdef FEATURE_NX_IPV6
298 #ifdef USE_IPV6
299
300     /* Add an IPv6 DNS server to the DNS client.  */
301     dns_ipv6_server_address.nxd_ip_address.v6[3] = 0x106;
302     dns_ipv6_server_address.nxd_ip_address.v6[2] = 0x0;
303     dns_ipv6_server_address.nxd_ip_address.v6[1] = 0x0000f101;
304     dns_ipv6_server_address.nxd_ip_address.v6[0] = 0x20010db8;
305     dns_ipv6_server_address.nxd_ip_version = NX_IP_VERSION_V6;
306
307     status = nxd_dns_server_add(&client_dns, &dns_ipv6_server_address);
308
309     /* Check for DNS add server error.  */
310     if (status)
311     {
312
313         error_counter++;
314         return;
315      }
316 #else
317
318     /* Add an IPv4 server address to the Client list. */
319     status = nx_dns_server_add(&client_dns, DNS_SERVER_ADDRESS);
320
321     /* Check for DNS add server error.  */
322     if (status)
```

```
323     {
324
325         error_counter++;
326         return;
327      }
328 #endif
329 #else
330
331     /* Add an IPv4 server address to the Client list. */
332     status = nx_dns_server_add(&client_dns, DNS_SERVER_ADDRESS);
333
334     /* Check for DNS add server error.  */
335     if (status)
336     {
337
338         error_counter++;
339         return;
340      }
341 #endif
342
343
344
/*****************************************************************************/
345 /*                                   Type AAAA
*/
346 /*     Send AAAA type DNS Query to its DNS server and get the IPv6 address.
*/
347
/*****************************************************************************/
348 #ifdef FEATURE_NX_IPV6
349
350     /* Send a DNS Client name query. Indicate the Client expects an IPv6 address
(containing an AAAA record). The DNS
351        Client will send AAAA type query to its DNS server. */
352     status = nxd_dns_host_by_name_get(&client_dns, (UCHAR *)"www.my_example.com",
&host_ipduo_address, 400, NX_IP_VERSION_V6);
353
354     /* Check for DNS query error.  */
355     if (status != NX_SUCCESS)
356     {
357         error_counter++;
358     }
359
360     else
361     {
362
363         printf("-----------------------------------------------------\n");
364         printf("Test AAAA: \n");
365
366         printf("IP address: %x:%x:%x:%x:%x:%x:%x:%x\n",
367             (UINT)host_ipduo_address.nxd_ip_address.v6[0]  >>16 & 0xFFFF,
368             (UINT)host_ipduo_address.nxd_ip_address.v6[0]  & 0xFFFF,
369             (UINT)host_ipduo_address.nxd_ip_address.v6[1]  >>16 & 0xFFFF,
370             (UINT)host_ipduo_address.nxd_ip_address.v6[1]  & 0xFFFF,
371             (UINT)host_ipduo_address.nxd_ip_address.v6[2]  >>16 & 0xFFFF,
372             (UINT)host_ipduo_address.nxd_ip_address.v6[2]  & 0xFFFF,
373             (UINT)host_ipduo_address.nxd_ip_address.v6[3]  >>16 & 0xFFFF,
374             (UINT)host_ipduo_address.nxd_ip_address.v6[3]  & 0xFFFF);
375     }
376
377 #endif
378
379     /* Look up IPv6 addresses(AAAA TYPE) to record multiple IPv6 addresses in
record_buffer and return the IPv6 address count.  */
380     status = nxd_dns_ipv6_address_by_name_get(&client_dns, (UCHAR
*)"www.my_example.com", &record_buffer[0], BUFFER_SIZE, &record_count, 400);
381
382     /* Check for DNS add server error.  */
383     if (status != NX_SUCCESS)
384     {
385         error_counter++;
386     }
387
388     else
389     {
390
391         printf("-----------------------------------------------------\n");
392         printf("Test AAAA: ");
393         printf("record_count = %d \n", record_count);
394     }
395
```

```
396     /* Get the IPv6 addresses of host.  */
397     for(i =0; i< record_count; i++)
398     {
399         ipv6_address_ptr[i] = (NX_DNS_IPV6_ADDRESS *)(record_buffer + i *
sizeof(NX_DNS_IPV6_ADDRESS));
400
401         printf("record %d: IP address: %x:%x:%x:%x:%x:%x:%x:%x\n", i,
402                 (UINT)(ipv6_address_ptr[i] -> ipv6_address[0]  >>16 & 0xFFFF),
403                 (UINT)(ipv6_address_ptr[i] -> ipv6_address[0]  & 0xFFFF),
404                 (UINT)(ipv6_address_ptr[i] -> ipv6_address[1]  >>16 & 0xFFFF),
405                 (UINT)(ipv6_address_ptr[i] -> ipv6_address[1]  & 0xFFFF),
406                 (UINT)(ipv6_address_ptr[i] -> ipv6_address[2]  >>16 & 0xFFFF),
407                 (UINT)(ipv6_address_ptr[i] -> ipv6_address[2]  & 0xFFFF),
408                 (UINT)(ipv6_address_ptr[i] -> ipv6_address[3]  >>16 & 0xFFFF),
409                 (UINT)(ipv6_address_ptr[i] -> ipv6_address[3]  & 0xFFFF));
410     }
411
412
413
/*****************************************************************************/
414 /*                                    Type A
*/
415 /*       Send A type DNS Query to its DNS server and get the IPv4 address.
*/
416
/*****************************************************************************/
417 #ifdef FEATURE_NX_IPV6
418     /* Send a DNS Client name query. Indicate the Client expects an IPv4 address
(containing an A record). If the DNS client
419        is using an IPv6 DNS server it will send this query over IPv6; otherwise
it will be sent over IPv4. */
420     status = nxd_dns_host_by_name_get(&client_dns, (UCHAR *)"www.my_example.com",
&host_ipduo_address, 400, NX_IP_VERSION_V4);
421
422     /* Check for DNS query error.  */
423     if (status != NX_SUCCESS)
424     {
425         error_counter++;
426     }
427     else
428     {
429
430         printf("-------------------------------------------------------\n");
431         printf("Test A: \n");
432         printf("IP address: %lu.%lu.%lu.%lu\n",
433               host_ipduo_address.nxd_ip_address.v4 >> 24,
434               host_ipduo_address.nxd_ip_address.v4 >> 16 & 0xFF,
435               host_ipduo_address.nxd_ip_address.v4 >> 8 & 0xFF,
436               host_ipduo_address.nxd_ip_address.v4 & 0xFF);
437     }
438
439 #endif
440
441     /* Look up an IPv4 address over IPv4. */
442     status = nx_dns_host_by_name_get(&client_dns, (UCHAR *)"www.my_example.com",
&host_ip_address, 400);
443
444     /* Check for DNS query error.  */
445     if (status != NX_SUCCESS)
446     {
447         error_counter++;
448     }
449
450     else
451     {
452
453         printf("-------------------------------------------------------\n");
454         printf("Test A: \n");
455         printf("IP address: %lu.%lu.%lu.%lu\n",
456         host_ip_address >> 24,
457         host_ip_address >> 16 & 0xFF,
458         host_ip_address >> 8 & 0xFF,
459         host_ip_address & 0xFF);
460     }
461
462
463     /* Look up IPv4 addresses to record multiple IPv4 addresses in record_buffer
and return the IPv4 address count.  */
464     status = nx_dns_ipv4_address_by_name_get(&client_dns, (UCHAR
*)"www.my_example.com", &record_buffer[0], BUFFER_SIZE, &record_count, 400);
465
```

```
466     /* Check for DNS query error.  */
467     if (status != NX_SUCCESS)
468     {
469         error_counter++;
470     }
471
472     else
473     {
474
475         printf("-----------------------------------------------------\n");
476         printf("Test A: ");
477         printf("record_count = %d \n", record_count);
478     }
479
480     /* Get the IPv4 addresses of host.  */
481     for(i =0; i< record_count; i++)
482     {
483         ipv4_address_ptr[i] = (ULONG *)(record_buffer + i * sizeof(ULONG));
484         printf("record %d: IP address: %lu.%lu.%lu.%lu\n", i,
485                 *ipv4_address_ptr[i] >> 24,
486                 *ipv4_address_ptr[i] >> 16 & 0xFF,
487                 *ipv4_address_ptr[i] >> 8 & 0xFF,
488                 *ipv4_address_ptr[i] & 0xFF);
489     }
490
491
492
/*****************************************************************************/
493 /*                                  Type A + CNAME response
*/
494 /*       Send A type DNS Query to its DNS server and get the IPv4 address.
*/
495
/*****************************************************************************/
496     /* Look up an IPv4 address over IPv4. */
497     status = nx_dns_host_by_name_get(&client_dns, (UCHAR *)"www.my_example.com",
&host_ip_address, 400);
498
499     /* Check for DNS query error.  */
500     if (status != NX_SUCCESS)
501     {
502         error_counter++;
503     }
504
505     else
506     {
507
508         printf("-----------------------------------------------------\n");
509         printf("Test A + CNAME response: \n");
510         printf("IP address: %lu.%lu.%lu.%lu\n",
511         host_ip_address >> 24,
512         host_ip_address >> 16 & 0xFF,
513         host_ip_address >> 8 & 0xFF,
514         host_ip_address & 0xFF);
515     }
516
517
518     /* Look up IPv4 addresses to record multiple IPv4 addresses in record_buffer
and return the IPv4 address count.  */
519     status = nx_dns_ipv4_address_by_name_get(&client_dns, (UCHAR
*)"www.my_example.com", &record_buffer[0], BUFFER_SIZE, &record_count, 400);
520
521     /* Check for DNS query error.  */
522     if (status != NX_SUCCESS)
523     {
524         error_counter++;
525     }
526
527     else
528     {
529
530         printf("-----------------------------------------------------\n");
531         printf("Test Test A + CNAME response: ");
532         printf("record_count = %d \n", record_count);
533     }
534
535     /* Get the IPv4 addresses of host.  */
536     for(i =0; i< record_count; i++)
537     {
538         ipv4_address_ptr[i] = (ULONG *)(record_buffer + i * sizeof(ULONG));
539         printf("record %d: IP address: %lu.%lu.%lu.%lu\n", i,
```

```
540                 *ipv4_address_ptr[i] >> 24,
541                 *ipv4_address_ptr[i] >> 16 & 0xFF,
542                 *ipv4_address_ptr[i] >> 8 & 0xFF,
543                 *ipv4_address_ptr[i] & 0xFF);
544     }
545
546
547
/**************************************************************************************/
548 /*                                  Type PTR
*/
549 /*       Send PTR type DNS Query to its DNS server and get the host name.
*/
550
/**************************************************************************************/
551
552 #ifdef FEATURE_NX_IPV6
553
554     /* Look up a host name from an IPv6 address (reverse lookup). */
555
556     /* Create an IPv6 address for a reverse lookup. */
557     test_ipduo_server_address.nxd_ip_version = NX_IP_VERSION_V6;
558     test_ipduo_server_address.nxd_ip_address.v6[0] = 0x24046800;
559     test_ipduo_server_address.nxd_ip_address.v6[1] = 0x40050c00;
560     test_ipduo_server_address.nxd_ip_address.v6[2] = 0x00000000;
561     test_ipduo_server_address.nxd_ip_address.v6[3] = 0x00000065;
562
563     /* This will be sent over IPv6 to the DNS server who should return a PTR
record if it can find the information. */
564     status = nxd_dns_host_by_address_get(&client_dns, &test_ipduo_server_address,
&record_buffer[0], BUFFER_SIZE, 450);
565
566     /* Check for DNS query error.  */
567     if (status != NX_SUCCESS)
568     {
569         error_counter++;
570     }
571
572     else
573     {
574
575         printf("------------------------------------------------------\n");
576         printf("Test PTR: %s\n", record_buffer);
577     }
578 #endif
579
580 #ifdef FEATURE_NX_IPV6
581
582     /* Create an IPv4 address for the reverse lookup. If the DNS client is IPv6
enabled, it will send this over
583        IPv6 to the DNS server; otherwise it will send it over IPv4. In either
case the respective server will
584        return a PTR record if it has the information. */
585     test_ipduo_server_address.nxd_ip_version = NX_IP_VERSION_V4;
586     test_ipduo_server_address.nxd_ip_address.v4 = IP_ADDRESS(74, 125, 71, 106);
587
588     status = nxd_dns_host_by_address_get(&client_dns, &test_ipduo_server_address,
&record_buffer[0], BUFFER_SIZE, 450);
589
590      /* Check for DNS query error.  */
591      if (status != NX_SUCCESS)
592      {
593          error_counter++;
594      }
595
596     else
597     {
598
599         printf("------------------------------------------------------\n");
600         printf("Test PTR: %s\n", record_buffer);
601     }
602 #endif
603
604      /* Look up host name over IPv4. */
605      host_ip_address = IP_ADDRESS(74, 125, 71, 106);
606      status = nx_dns_host_by_address_get(&client_dns, host_ip_address,
&record_buffer[0], BUFFER_SIZE, 450);
607
608      /* Check for DNS query error.  */
609      if (status != NX_SUCCESS)
610      {
```

```
611          error_counter++;
612      }
613
614     else
615     {
616         printf("-------------------------------------------------\n");
617         printf("Test PTR: %s\n", record_buffer);
618     }
619
620 #ifdef NX_DNS_ENABLE_EXTENDED_RR_TYPES
621
/*****************************************************************************/
622 /*                                     Type CNAME
*/
623 /*   Send CNAME type DNS Query to its DNS server and get the canonical name .
*/
624
/*****************************************************************************/
625
626      /* Send CNAME type to record the canonical name of host in record_buffer.
*/
627      status = nx_dns_cname_get(&client_dns, (UCHAR *)"www.my_example.com",
&record_buffer[0], BUFFER_SIZE, 400);
628
629      /* Check for DNS query error.  */
630      if (status != NX_SUCCESS)
631      {
632          error_counter++;
633      }
634
635     else
636     {
637
638         printf("-------------------------------------------------\n");
639         printf("Test CNAME: %s\n", record_buffer);
640     }
641
642
643
/*****************************************************************************/
644 /*                                     Type TXT
*/
645 /*      Send TXT type DNS Query to its DNS server and get descriptive text.
*/
646
/*****************************************************************************/
647
648      /* Send TXT type to record the descriptive test of host in record_buffer.
*/
649      status = nx_dns_host_text_get(&client_dns, (UCHAR *)"www.my_example.com",
&record_buffer[0], BUFFER_SIZE, 400);
650
651      /* Check for DNS query error.  */
652      if (status != NX_SUCCESS)
653      {
654          error_counter++;
655      }
656
657     else
658     {
659
660         printf("-------------------------------------------------\n");
661         printf("Test TXT: %s\n", record_buffer);
662     }
663
664
665
/*****************************************************************************/
666 /*                                     Type NS
*/
667 /*   Send NS type DNS Query to its DNS server and get the domain name server.
*/
668
/*****************************************************************************/
669
670      /* Send NS type to record multiple name servers in record_buffer and return
the name server count.
671         If the DNS response includes the IPv4 addresses of name server, record it
similarly in record_buffer.  */
672      status = nx_dns_domain_name_server_get(&client_dns, (UCHAR
*)"www.my_example.com", &record_buffer[0], BUFFER_SIZE, &record_count, 400);
```

```
673
674      /* Check for DNS query error.  */
675      if (status != NX_SUCCESS)
676      {
677          error_counter++;
678      }
679
680     else
681     {
682
683         printf("-----------------------------------------------------\n");
684         printf("Test NS: ");
685         printf("record_count = %d \n", record_count);
686     }
687
688     /* Get the name server.  */
689     for(i =0; i< record_count; i++)
690     {
691         nx_dns_ns_entry_ptr[i] = (NX_DNS_NS_ENTRY *)(record_buffer + i *
sizeof(NX_DNS_NS_ENTRY));
692
693         printf("record %d: IP address: %d.%d.%d.%d\n", i,
694                 nx_dns_ns_entry_ptr[i] -> nx_dns_ns_ipv4_address  >> 24,
695                 nx_dns_ns_entry_ptr[i] -> nx_dns_ns_ipv4_address >> 16 & 0xFF,
696                 nx_dns_ns_entry_ptr[i] -> nx_dns_ns_ipv4_address >> 8 & 0xFF,
697                 nx_dns_ns_entry_ptr[i] -> nx_dns_ns_ipv4_address & 0xFF);
698         if(nx_dns_ns_entry_ptr[i] -> nx_dns_ns_hostname_ptr)
699             printf("hostname = %s\n", nx_dns_ns_entry_ptr[i] ->
nx_dns_ns_hostname_ptr);
700         else
701             printf("hostname is not set\n");
702     }
703
704
/*****************************************************************************/
705 /*                                    Type MX
*/
706 /*   Send MX type DNS Query to its DNS server and get the domain mail exchange.
*/
707
/*****************************************************************************/
708
709      /* Send MX DNS query type to record multiple mail exchanges in record_buffer
and return the mail exchange count.
710         If the DNS response includes the IPv4 addresses of mail exchange, record
it similarly in record_buffer.  */
711      status = nx_dns_domain_mail_exchange_get(&client_dns, (UCHAR
*)"www.my_example.com", &record_buffer[0], BUFFER_SIZE, &record_count, 400);
712
713      /* Check for DNS query error.  */
714      if (status != NX_SUCCESS)
715      {
716          error_counter++;
717      }
718
719     else
720     {
721
722         printf("-----------------------------------------------------\n");
723         printf("Test MX: ");
724         printf("record_count = %d \n", record_count);
725     }
726
727     /* Get the mail exchange.  */
728     for(i =0; i< record_count; i++)
729     {
730         nx_dns_mx_entry_ptr[i] = (NX_DNS_MX_ENTRY *)(record_buffer + i *
sizeof(NX_DNS_MX_ENTRY));
731
732         printf("record %d: IP address: %d.%d.%d.%d\n", i,
733                 nx_dns_mx_entry_ptr[i] -> nx_dns_mx_ipv4_address >> 24,
734                 nx_dns_mx_entry_ptr[i] -> nx_dns_mx_ipv4_address >> 16 & 0xFF,
735                 nx_dns_mx_entry_ptr[i] -> nx_dns_mx_ipv4_address >> 8 & 0xFF,
736                 nx_dns_mx_entry_ptr[i] -> nx_dns_mx_ipv4_address & 0xFF);
737         printf("preference = %d \n ", nx_dns_mx_entry_ptr[i] ->
nx_dns_mx_preference);
738         if(nx_dns_mx_entry_ptr[i] -> nx_dns_mx_hostname_ptr)
739             printf("hostname = %s\n", nx_dns_mx_entry_ptr[i] ->
nx_dns_mx_hostname_ptr);
740         else
741             printf("hostname is not set\n");
```

```
742     }
743
744
/*****************************************************************************/
745 /*                                     Type SRV
*/
746 /*  Send SRV type DNS Query to its DNS server and get the location of services.
*/
747
/*****************************************************************************/
748
749      /* Send SRV DNS query type to record the location of services in
record_buffer and return count.
750         If the DNS response includes the IPv4 addresses of service name, record
it similarly in record_buffer.  */
751      status = nx_dns_domain_service_get(&client_dns, (UCHAR
*)"www.my_example.com", &record_buffer[0], BUFFER_SIZE, &record_count, 400);
752
753      /* Check for DNS query error.  */
754      if (status != NX_SUCCESS)
755      {
756          error_counter++;
757      }
758
759     else
760     {
761
762         printf("-----------------------------------------------------\n");
763         printf("Test SRV: ");
764         printf("record_count = %d \n", record_count);
765     }
766
767     /* Get the location of services.  */
768     for(i =0; i< record_count; i++)
769     {
770         nx_dns_srv_entry_ptr[i] = (NX_DNS_SRV_ENTRY *)(record_buffer + i *
sizeof(NX_DNS_SRV_ENTRY));
771
772         printf("record %d: IP address: %d.%d.%d.%d\n", i,
773                 nx_dns_srv_entry_ptr[i] -> nx_dns_srv_ipv4_address >> 24,
774                 nx_dns_srv_entry_ptr[i] -> nx_dns_srv_ipv4_address >> 16 & 0xFF,
775                 nx_dns_srv_entry_ptr[i] -> nx_dns_srv_ipv4_address >> 8 & 0xFF,
776                 nx_dns_srv_entry_ptr[i] -> nx_dns_srv_ipv4_address & 0xFF);
777         printf("port number = %d\n", nx_dns_srv_entry_ptr[i] ->
nx_dns_srv_port_number );
778         printf("priority = %d\n", nx_dns_srv_entry_ptr[i] ->
nx_dns_srv_priority );
779         printf("weight = %d\n", nx_dns_srv_entry_ptr[i] -> nx_dns_srv_weight );
780         if(nx_dns_srv_entry_ptr[i] -> nx_dns_srv_hostname_ptr)
781             printf("hostname = %s\n", nx_dns_srv_entry_ptr[i] ->
nx_dns_srv_hostname_ptr);
782         else
783             printf("hostname is not set\n");
784     }
785
786     /* Get the service info, NetX old API.*/
787     status = nx_dns_info_by_name_get(&client_dns, (UCHAR *)"www.my_example.com",
&host_address, &host_port, 200);
788
789     /* Check for DNS add server error.  */
790      if (status != NX_SUCCESS)
791      {
792          error_counter++;
793      }
794
795     else
796     {
797
798         printf("-----------------------------------------------------\n");
799         printf("Test SRV: ");
800         printf("IP address: %d.%d.%d.%d\n",
801                 host_address >> 24,
802                 host_address >> 16 & 0xFF,
803                 host_address >> 8 & 0xFF,
804                 host_address & 0xFF);
805         printf("port number = %d\n", host_port);
806     }
807
808
/*****************************************************************************/
```

```
809 /*                                      Type SOA
*/
810 /* Send SOA type DNS Query to its DNS server and get zone of start of
authority.*/
811
/**************************************************************************************/
812
813      /* Send SOA DNS query type to record the zone of start of authority in
record_buffer.  */
814      status = nx_dns_authority_zone_start_get(&client_dns, (UCHAR
*)"www.my_example.com", &record_buffer[0], BUFFER_SIZE, 400);
815
816      /* Check for DNS query error.  */
817      if (status != NX_SUCCESS)
818      {
819          error_counter++;
820      }
821
822      /* Get the loc*/
823      nx_dns_soa_entry_ptr = (NX_DNS_SOA_ENTRY *) record_buffer;
824      printf("------------------------------------------------------\n");
825      printf("Test SOA: \n");
826      printf("serial = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_serial );
827      printf("refresh = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_refresh );
828      printf("retry = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_retry );
829      printf("expire = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_expire );
830      printf("minmum = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_minmum );
831      if(nx_dns_soa_entry_ptr -> nx_dns_soa_host_mname_ptr)
832          printf("host mname = %s\n", nx_dns_soa_entry_ptr ->
nx_dns_soa_host_mname_ptr);
833      else
834          printf("host mame is not set\n");
835      if(nx_dns_soa_entry_ptr -> nx_dns_soa_host_rname_ptr)
836          printf("host rname = %s\n", nx_dns_soa_entry_ptr ->
nx_dns_soa_host_rname_ptr);
837      else
838          printf("host rname is not set\n");
839
840
841 #endif
842
843     /* Shutting down...*/
844
845     /* Terminate the DNS Client thread. */
846     status = nx_dns_delete(&client_dns);
847
848     return;
849 }
850
851
```

Configuration Options

There are several configuration options for building DNS for NetX. These options can be redefined in *nxd_dns.h.* The following list describes each in detail:

Define	Meaning
NX_DNS_TYPE_OF_SERVICE	Type of service required for the DNS UDP requests. By default, this value is defined as NX_IP_NORMAL for normal IP packet service.
NX_DNS_TIME_TO_LIVE	Specifies the maximum number of routers a packet can pass before it is discarded. The default value is 0x80.
NX_DNS_FRAGMENT_OPTION	Sets the socket property to allow or disallow fragmentation of outgoing packets. The default value is NX_DONT_FRAGMENT.
NX_DNS_QUEUE_DEPTH	Sets the maximum number of packets to store on the socket receive queue. The default value is 5.
NX_DNS_MAX_SERVERS	Specifies the maximum number of DNS Servers in the Client server list. The default value is 5.
NX_DNS_MESSAGE_MAX	The maximum DNS message size for sending DNS queries. The default value is 512, which is also the maximum size specified in RFC 1035 Section 2.3.4.
NX_DNS_PACKET_PAYLOAD_UNALIGNED	If not defined, the size of the Client packet payload which includes the Ethernet, IP (or IPv6), and UDP headers plus the maximum DNS message size specified by

NX_DNS_MESSAGE_MAX. Regardless if defined, the packet payload is the 4-byte aligned and stored in NX_DNS_PACKET_PAYLOAD.

NX_DNS_PACKET_POOL_SIZE Size of the Client packet pool for sending DNS queries if NX_DNS_CLIENT_USER_CREATE_PACKET_POOL is not defined. The default value is large enough for 16 packets of payload size defined by NX_DNS_PACKET_PAYLOAD, and is 4-byte aligned.

NX_DNS_MAX_RETRIES The maximum number of times the DNS Client will query the current DNS server before trying another server or aborting the DNS query. The default value is 3.

NX_ DNS_MAX_RETRANS_TIMEOUT The maximum retransmission timeout on a DNS query to a specific DNS server. The default value is 64 seconds (64 * NX_IP_PERIODIC_RATE).

NX_DNS_IP_GATEWAY_AND_DNS_SERVER
If defined and the Client IPv4 gateway address is non zero, the DNS Client sets the IPv4 gateway as the Client's primary DNS server. The default value is disabled.

NX_ DNS_PACKET_ALLOCATE_TIMEOUT
This sets the timeout option for allocating a packet from the DNS client packet pool. The default value is 1 second (1*NX_IP_PERIODIC_RATE).

NX_DNS_CLIENT_USER_CREATE_PACKET_POOL
This enables the DNS Client to let the application create and set the DNS Client packet pool. By default this option is disabled, and the DNS Client creates its own packet pool in *nx_dns_create*.

NX_DNS_CLIENT_CLEAR_QUEUE This enables the DNS Client

to clear old DNS messages off the receive queue before sending a new query. Removing these packets from previous DNS queries prevents the DNS Client socket queue from overflowing and dropping valid packets.

NX_DNS_ENABLE_EXTENDED_RR_TYPES

This enables the DNS Client to query on additional DNS record types in (e.g. CNAME, NS, MX, SOA, SRV and TXT).

NX_DNS_CACHE_ENABLE

This enables the DNS Client to store the answer records into DNS cache.

Chapter 3

Description of DNS Client Services

This chapter contains a description of all NetX DNS services (listed below) in alphabetic 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_dns_authority_zone_start_get
Look up the start of a zone of authority associated with the specified host name

nx_dns_cache_initialize
Initialize a DNS Cache.

nx_dns_cache_notify_clear
Clear the cache full notify function.

nx_dns_cache_notify_set
Set the cache full notify function.

nx_dns_cname_get
Look up the canonical domain name for the input domain name alias

nx_dns_create
Create a DNS Client instance

nx_dns_delete
Delete a DNS Client instance

nx_dns_domain_name_server_get
Look up the authoritative name servers for the input domain zone

nx_dns_domain_mail_exchange_get
Look up the mail exchange associated the specified host name.

nx_dns_domain_service_get

Look up the service(s) associated with the specified host name

nx_dns_get_serverlist_size
Return the size of the DNS Client server list

nx_dns_info_by_name_get
Return IP address, port querying on input host name

nx_dns_ipv4_address_by_name_get
Look up the IPv4 address from the specified host name

nxd_dns_ipv6_address_by_name_get
Look up the IPv6 address from the specified host name

nx_dns_host_by_address_get
Wrapper function for nxd_dns_host_by_address_get *to look up a host name from a specified IP address (supports only IPv4 addresses)*

nxd_dns_host_by_address_get
Look up an IP address from the input host name (supports both IPv4 and IPv6 addresses)

nx_dns_host_by_name_get
Wrapper function for nxd_dns_host_by_address_get *to look up a host name from the specified address (supports only IPv4 addresses)*

nxd_dns_host_by_name_get
Look up an IP address from the input host name (supports both IPv4 and IPv6 addresses)

nx_dns_host_text_get
Look up the text data for the input domain name

nx_dns_packet_pool_set
Set the DNS Client packet pool

nx_dns_server_add
Wrapper function for nxd_dns_server_add *to add a DNS Server at the specified address to the Client list (supports only IPv4)*

nxd_dns_server_add
Add a DNS Server of the specified IP address

to the Client server list (supports both IPv4 or IPv6 addresses)

nx_dns_server_get
Return the DNS Server in the Client list (supports only IPv4 addresses)

nxd_dns_server_get
Return the DNS Server in the Client list (supports both IPv4 and IPv6 addresses)

nx_dns_server_remove
Wrapper function for nxd_dns_server_remove *to remove a DNS Server from the Client list*

nxd_dns_server_remove
Remove a DNS Server of the specified IP address from the Client list (supports both IPv4 and IPv6 addresses)

nx_dns_server_remove_all
Remove all DNS Servers from the Client list

nx_dns_authority_zone_start_get

Look up the start of the zone of authority for the input host

Prototype

```
UINT nx_dns_authority_zone_start_get (NX_DNS *dns_ptr, UCHAR *host_name,
                                      VOID *record_buffer,
                                      UINT buffer_size,
                                      UINT *record_count,
                                      ULONG wait_option);
```

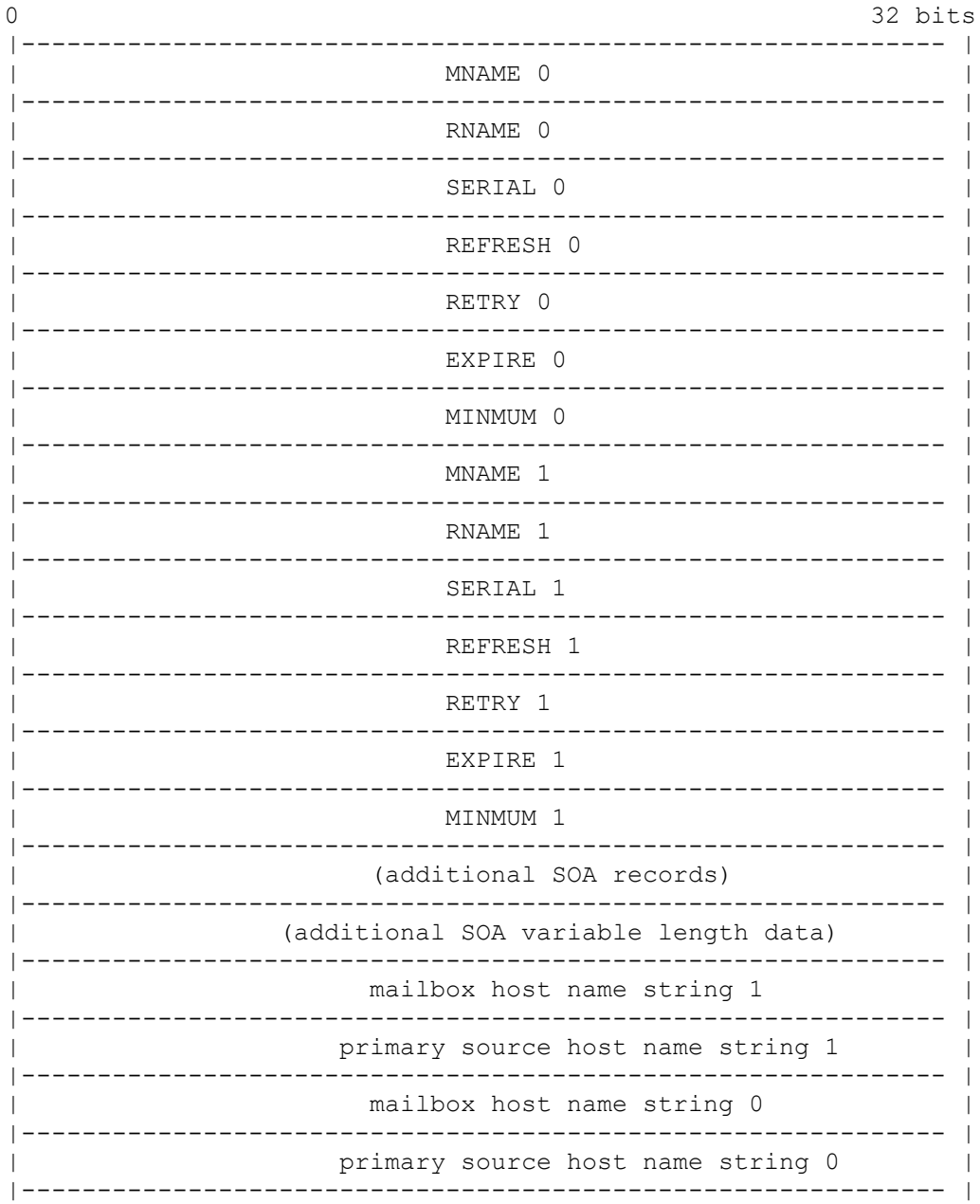
Description

If NX_DNS_ENABLE_EXTENDED_RR_TYPES is defined, this service sends a query of type SOA with the specified domain name to obtain the start of the zone of authority for the input domain name. The DNS Client copies the SOA record(s) returned in the DNS Server response into the *record_buffer* memory location. Note that *record_buffer* must be 4-byte aligned to receive the data.

In NetX Duo DNS Client, the SOA record type, NX_DNS_SOA_ENTRY, is saved as seven 4 byte parameters, totaling 28 bytes:

nx_dns_soa_host_mname_ptr	Pointer to primary source of data for this zone
nx_dns_soa_host_rname_ptr	Pointer to mailbox responsible for this zone
nx_dns_soa_serial	Zone version number
nx_dns_soa_refresh	Refresh interval
nx_dns_soa_retry	Interval between SOA query retries
nx_dns_soa_expire	Time duration when SOA expires
nx_dns_soa_minmum	Minimum TTL field in SOA hostname DNS reply messages

The storage of a two SOA records is shown below. The SOA records containing fixed length data are entered starting at the top of the buffer. The pointers MNAME and RNAME point to the variable length data (host names) which are stored at the bottom of the buffer. Additional SOA records are entered after the first record (“additional SOA records…”) and their variable length data is stored above the last entry’s variable length data (“additional SOA variable length data”):



If the input *record_buffer* cannot hold all the SOA data in the server reply, the the *record_buffer* holds as many records as will fit and returns the number of records in the buffer.

With the number of SOA records returned in **record_count,* the application can parse the data from *record_buffer* and extract the start of zone authority host name strings.

Input Parameters

dns_ptr	Pointer to DNS Client.
host_name	Pointer to host name to obtain SOA data for
record_buffer	Pointer to location to extract SOA data into
buffer_size	Size of buffer to hold SOA data
record_count	Pointer to the number of SOA records retrieved
wait_option	Wait option to receive DNS Server response

Return Values

NX_SUCCESS	(0x00)	Successfully obtained SOA data
NX_DNS_NO_SERVER	(0xA1)	Client server list is empty
NX_DNS_QUERY_FAILED	(0xA3)	No valid DNS response received
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer input

Allowed From

Threads

Example

```
UCHAR  record_buffer[50];
UINT   record_count;
NX_DNS_SOA_ENTRY *nx_dns_soa_entry_ptr;

/* Request the start of authority zone(s) for the specified host.  */
status =  nx_dns_authority_zone_start_get(&client_dns, (UCHAR *)"www.my_example.com",
                                          record _buffer, sizeof(record_buffer),
                                          &record_count, 500);

/* Check for DNS query error.  */
if (status != NX_SUCCESS)
{
         error_counter++;
}
else
{
    /* If status is NX_SUCCESS a DNS query was successfully completed and SOA data is
    returned in soa_buffer.  */

    /* Set a local pointer to the SOA buffer. */
    nx_dns_soa_entry_ptr = (NX_DNS_SOA_ENTRY *) record_buffer;

    printf("-----------------------------------------------------\n");
    printf("Test SOA: \n");
    printf("serial = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_serial );
    printf("refresh = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_refresh );
    printf("retry = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_retry );
    printf("expire = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_expire );
```

```
    printf("minmum = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_minmum );

    if(nx_dns_soa_entry_ptr -> nx_dns_soa_host_mname_ptr)
    {
        printf("host mname = %s\n",
               nx_dns_soa_entry_ptr -> nx_dns_soa_host_mname_ptr);
    }
    else
    {
        printf("host mame is not set\n");
    }

    if(nx_dns_soa_entry_ptr -> nx_dns_soa_host_rname_ptr)
    {
        printf("host rname = %s\n",
               nx_dns_soa_entry_ptr -> nx_dns_soa_host_rname_ptr);
    }
    else
    {

        printf("host rname is not set\n");
    }
}
```

```
[Output]
-----------------------------------------------------
Test SOA:
serial = 2012111212
refresh = 7200
retry = 1800
expire = 1209600
minmum = 300
host mname = ns1.www.my_example.com
host rname = dns-admin.www.my_example.com
```

nx_dns_cache_initialize

Initialize the DNS Cache

Prototype

```
UINT nx_dns_cache_initialize(NX_DNS *dns_ptr,
                             VOID *cache_ptr, UINT cache_size);
```

Description

This service creates and initializes a DNS Cache.

Input Parameters

dns_ptr	Pointer to DNS control block.
cache_ptr	Pointer to DNS Cache.
cache_size	Size of DNS Cache, in bytes.

Return Values

NX_SUCCESS	(0x00)	DNS Cache successfully initialized
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer input
NX_DNS_CACHE_ERROR	(0xB7)	Invalid Cache pointer.
NX_PTR_ERROR	(0x07)	Invalid DNS pointer.
NX_DNS_ERROR	(0xA0)	Cache is not 4-byte aligned.

Allowed From

Threads

Example

```
/* Initialize the DNS Cache.  */
status =  nx_dns_cache_initialize(&my_dns, &dns_cache, 2048);

/* If status is NX_SUCCESS DNS Cache was successfully initialized.  */
```

nx_dns_cache_notify_clear

Clear the DNS Cache full notify function

Prototype

```
UINT nx_dns_cache_notify_clear(NX_DNS *dns_ptr);
```

Description

This service clears the cache full notify function.

Input Parameters

dns_ptr	Pointer to DNS control block.

Return Values

NX_SUCCESS	(0x00)	DNS cache notify successfully set
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non-pointer input
NX_PTR_ERROR	(0x07)	Invalid DNS pointer.

Allowed From

Threads

Example

```
/* Clear the DNS Cache full notify function.  */
status =  nx_dns_cache_notify_clear(&my_dns);

/* If status is NX_SUCCESS DNS Cache full notify function was successfully cleared. */
```

nx_dns_cache_notify_set

Set the DNS Cache full notify function

Prototype

```
UINT nx_dns_cache_notify_set(NX_DNS *dns_ptr,
                             VOID (*cache_full_notify_cb)(NX_DNS *dns_ptr));
```

Description

This service sets the cache full notify function.

Input Parameters

dns_ptr	Pointer to DNS control block.
cache_full_notify_cb	The callback function to be invoked when cache become full.

Return Values

NX_SUCCESS	(0x00)	DNS cache notify successfully set
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non-pointer input
NX_PTR_ERROR	(0x07)	Invalid DNS pointer.

Allowed From

Threads

Example

```
/* Set the DNS Cache full notify function.  */
status =  nx_dns_cache_notify_set(&my_dns, cache_full_notify_cb);

/* If status is NX_SUCCESS DNS Cache full notify function was successfully set.  */
```

nx_dns_cname_get

Look up the canonical name for the input hostname

Prototype

```
UINT nx_dns_cname_get(NX_DNS *dns_ptr, UCHAR *host_name,
                      UCHAR *record_buffer, UINT buffer_size,
                      ULONG wait_option);
```

Description

If NX_DNS_ENABLE_EXTENDED_RR_TYPES is defined in *nxd_dns.h*, this service sends a query of type CNAME with the specified domain name to obtain the canonical domain name. The DNS Client copies the CNAME string returned in the DNS Server response into the *record_buffer* memory location.

Input Parameters

dns_ptr	Pointer to DNS Client.
host_name	Pointer to host name to obtain CNAME data for
record_buffer	Pointer to location to extract CNAME data into
buffer_size	Size of buffer to hold CNAME data
wait_option	Wait option to receive DNS Server response

Return Values

NX_SUCCESS	(0x00)	Successfully obtained CNAME data
NX_DNS_NO_SERVER	(0xA1)	Client server list is empty
NX_DNS_QUERY_FAILED	(0xA3)	No valid DNS response received
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non-pointer input

Allowed From

Threads

Example

```
CHAR            record _buffer[50];

/* Request the canonical name for the specified host.  */
status =  nx_dns_cname_get(&client_dns, (UCHAR *)"www.my_example.com ",
                           record_buffer, sizeof(record_buffer), 500);

/* Check for DNS query error.  */
if (status != NX_SUCCESS)
{
    error_counter++;
```

```
}
else
{
    /* If status is NX_SUCCESS a DNS query was successfully completed and the
       canonical host name is returned in record_buffer.  */

    printf("-----------------------------------------------------\n");
    printf("Test CNAME: %s\n", record_buffer);
}
```

```
[Output]
-----------------------------------------------------
Test CNAME: my_example.com
```

nx_dns_create

Create a DNS Client instance

Prototype

```
UINT nx_dns_create(NX_DNS *dns_ptr, NX_IP *ip_ptr, CHAR *domain_name);
```

Description

This service creates a DNS Client instance for the previously created IP instance.

Important Note: The application must ensure that the packet payload of the packet pool used by the DNS Client is large enough for the maximum 512 byte DNS message, plus UDP, IP and Ethernet headers. If the DNS Client creates its own packet pool, this is defined by NX_DNS_PACKET_PAYLOAD and NX_DNS_PACKET_POOL_SIZE.

If the DNS Client application prefers to supply a previously created packet pool, the payload for IPv4 DNS Client should be 512 bytes for the maximum DNS plus 20 bytes for the IP header, 8 bytes for the UDP header and 14 bytes for the Ethernet header. For IPv6 the only difference is the IP header is 40 bytes, therefore the packet needs to accommodate the IPv6 header of 40 bytes.

Input Parameters

dns_ptr	Pointer to DNS Client.
ip_ptr	Pointer to previously created IP instance.
domain_name	Pointer to domain name for DNS instance.

Return Values

NX_SUCCESS	(0x00)	Successful DNS create
NX_DNS_ERROR	(0xA0)	DNS create error
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Create a DNS Client instance.  */
status =  nx_dns_create(&my_dns, &my_ip, "My DNS");

/* If status is NX_SUCCESS a DNS Client instance was successfully
   created.  */
```

nx_dns_delete

Delete a DNS Client instance

Prototype

```
UINT nx_dns_delete(NX_DNS *dns_ptr);
```

Description

This service deletes a previously created DNS Client instance and frees up its resources. Note that if NX_DNS_CLIENT_USER_CREATE_PACKET_POOL is defined and the DNS Client was assigned a user defined packet pool, it is up to the application to delete the DNS Client packet pool if it no longer needs it.

Input Parameters

dns_ptr	Pointer to previously created DNS Client instance.

Return Values

NX_SUCCESS	(0x00)	Successful DNS Client delete.
NX_PTR_ERROR	(0x07)	Invalid IP or DNS Client pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Delete a DNS Client  instance.  */
status =  nx_dns_delete(&my_dns);

/* If status is NX_SUCCESS the DNS Client  instance was successfully
   deleted.  */
```

nx_dns_domain_name_server_get

Look up the authoritative name servers for the input domain zone

Prototype

```
UINT nx_dns_domain_name_server_get(NX_DNS *dns_ptr, UCHAR *host_name,
                                   VOID *record_buffer, UINT buffer_size,
                                   UINT *record_count, ULONG wait_option);
```

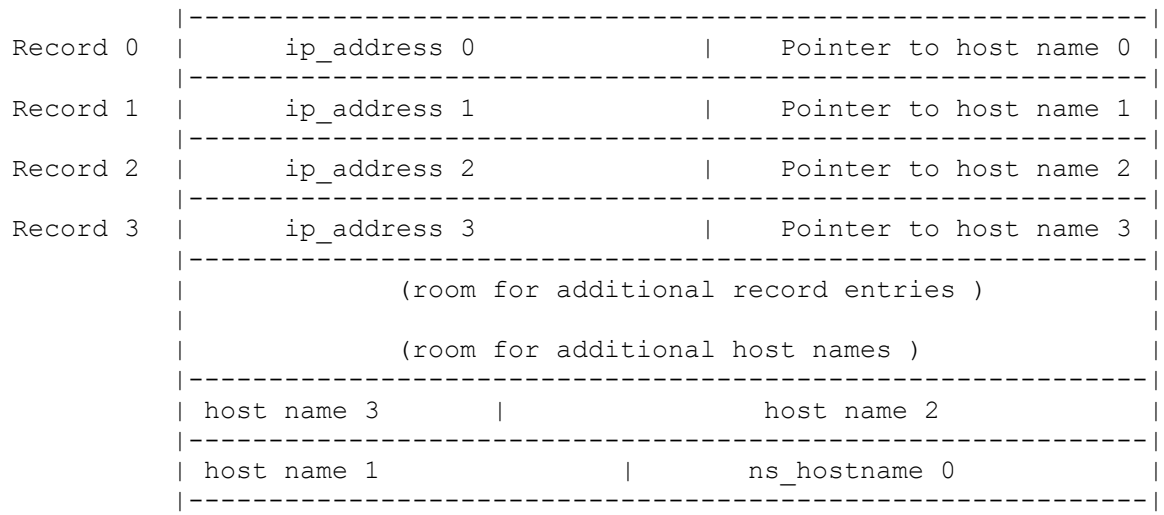
Description

If NX_DNS_ENABLE_EXTENDED_RR_TYPES is defined, this service sends a query of type NS with the specified domain name to obtain the name servers for the input domain name. The DNS Client copies the NS record(s) returned in the DNS Server response into the *record_buffer* memory location. Note that *record_buffer* must be 4-byte aligned to receive the data.

In NetX Duo DNS Client the NS data type, NX_DNS_NS_ENTRY, is saved as two 4-byte parameters:

nx_dns_ns_ipv4_address	Name server's IPv4 address
nx_dns_ns_hostname_ptr	Pointer to the name server's hostname

The buffer shown below contains four NX_DNS_NS_ENTRY records. The pointer to host name string in each entry points to the corresponding host name string in the bottom half of the buffer:



If the input *record_buffer* cannot hold all the NS data in the server reply, the the *record_buffer* holds as many records as will fit and returns the number of records in the buffer.

With the number of NS records returned in **record_count,* the application can parse the IP address and host name of each record in the *record_buffer*.

Input Parameters

dns_ptr	Pointer to DNS Client.
host_name	Pointer to host name to obtain NS data for
record_buffer	Pointer to location to extract NS data into
buffer_size	Size of buffer to hold NS data
record_count	Pointer to the number of NS records retrieved
wait_option	Wait option to receive DNS Server response

Return Values

NX_SUCCESS	(0x00)	Successfully obtained NS data
NX_DNS_NO_SERVER	(0xA1)	Client server list is empty
NX_DNS_QUERY_FAILED	(0xA3)	No valid DNS response received
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non-pointer input
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
#define RECORD_COUNT    10

ULONG  record_buffer[50];
UINT   record_count;
NX_DNS_NS_ENTRY  *nx_dns_ns_entry_ptr[RECORD_COUNT];

/* Request the name server(s) for the specified host.  */
status =  nx_dns_domain_name_server_get(&client_dns, (UCHAR *)" www.my_example.com ",
                                        record_buffer, sizeof(record_buffer),
                                        &record_count, 500);

/* Check for DNS query error.  */
if (status != NX_SUCCESS)
{
    error_counter++;
}

else
{
    /* If status is NX_SUCCESS a DNS query was successfully completed and NS data is
       returned in record_buffer.  */

    printf("-----------------------------------------------------\n");
    printf("Test NS: ");
    printf("record_count = %d \n", record_count);
```

```
    /* Get the name server.  */
    for(i =0; i< record_count; i++)
    {
        nx_dns_ns_entry_ptr[i] = (NX_DNS_NS_ENTRY *)
                                 (record_buffer + i * sizeof(NX_DNS_NS_ENTRY));

        printf("record %d: IP address: %d.%d.%d.%d\n", i,
                      nx_dns_ns_entry_ptr[i] -> nx_dns_ns_ipv4_address  >> 24,
                      nx_dns_ns_entry_ptr[i] -> nx_dns_ns_ipv4_address >> 16 & 0xFF,
                      nx_dns_ns_entry_ptr[i] -> nx_dns_ns_ipv4_address >> 8 & 0xFF,
                     nx_dns_ns_entry_ptr[i] -> nx_dns_ns_ipv4_address & 0xFF);
        if(nx_dns_ns_entry_ptr[i] -> nx_dns_ns_hostname_ptr)
        {
            printf("hostname = %s\n",
                    nx_dns_ns_entry_ptr[i] -> nx_dns_ns_hostname_ptr);
         }
        else
            printf("hostname is not set\n");
    }
}

[Output]
-------------------------------------------------------
Test NS: record_count = 4
record 0: IP address: 192.2.2.10
hostname = ns2.www.my_example.com
record 1: IP address: 192.2.2.11
hostname = ns1.www.my_example.com
record 2: IP address: 192.2.2.12
hostname = ns3.www.my_example.com
record 3: IP address: 192.2.2.13
hostname = ns4.www.my_example.com
```

nx_dns_domain_mail_exchange_get

Look up the mail exchange(s) for the input host name

Prototype

```
UINT nx_dns_domain_mail_exchange_get(NX_DNS *dns_ptr, UCHAR *host_name,
                                     VOID *record_buffer,
                                     UINT buffer_size,
                                     UINT *record_count,
                                     ULONG wait_option);
```

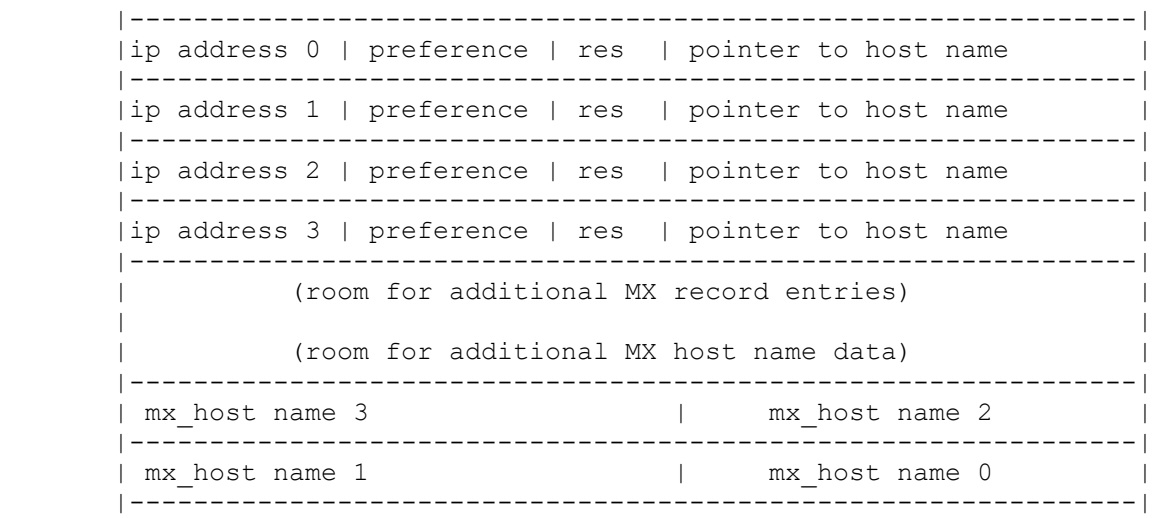
Description

If NX_DNS_ENABLE_EXTENDED_RR_TYPES is defined, this service sends a query of type MX with the specified domain name to obtain the mail exchange for the input domain name. The DNS Client copies the MX record(s) returned in the DNS Server response into the *record_buffer* memory location. Note that *record_buffer* must be 4-byte aligned to receive the data.

In NetX Duo DNS Client, the mail exchange record type, NX_DNS_MAIL_EXCHANGE_ENTRY, is saved as four parameters, totaling 12 bytes:

nx_dns_mx_ipv4_address	Mail exchange IPv4 address	4 bytes
nx_dns_mx_preference	Preference	2 bytes
nx_dns_mx_reserved0	Reserved	2 bytes
nx_dns_mx_hostname_ptr	Pointer to mail exchange server host name	4 bytes

A buffer containing four MX records is shown below. Each record contains the fixed length data from the list above. The pointer to the mail exchange server host name points to the corresponding host name at the bottom of the buffer.



If the input *record_buffer* cannot hold all the MX data in the server reply, the the *record_buffer* holds as many records as will fit and returns the number of records in the buffer.

With the number of MX records returned in **record_count,* the application can parse the MX parameters, including the mail host name of each record in the *record_buffer*.

Input Parameters

dns_ptr	Pointer to DNS Client.
host_name	Pointer to host name to obtain MX data for
record_buffer	Pointer to location to extract MX data into
buffer_size	Size of buffer to hold MX data
record_count	Pointer to the number of MX records retrieved
wait_option	Wait option to receive DNS Server response

Return Values

NX_SUCCESS	(0x00)	Successfully obtained MX data
NX_DNS_NO_SERVER	(0xA1)	Client server list is empty
NX_DNS_QUERY_FAILED	(0xA3)	No valid DNS response received
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non-pointer input
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
#define MAX_RECORD_COUNT 10

ULONG  record_buffer[50];
UINT   record_count;
NX_DNS_MX_ENTRY  *nx_dns_mx_entry_ptr[MAX_RECORD_COUNT];

/* Request the mail exchange data for the specified host.  */
status =  nx_dns_domain_mail_exchange_get(&client_dns, (UCHAR *)"  www.my_example.com
",
                                          record_buffer, sizeof(record_buffer),
                                          &record_count, 500);

/* Check for DNS query error.  */
if (status != NX_SUCCESS)
{
    error_counter++;
```

```
}

else
{
    /* If status is NX_SUCCESS a DNS query was successfully completed and MX data
       is returned in record_buffer.  */

    printf("-----------------------------------------------------\n");
    printf("Test MX: ");
    printf("record_count = %d \n", record_count);


    /* Get the mail exchange.  */
    for(i =0; i< record_count; i++)
    {
        nx_dns_mx_entry_ptr[i] = (NX_DNS_MX_ENTRY *)
               (record_buffer + i * sizeof(NX_DNS_MX_ENTRY));

        printf("record %d: IP address: %d.%d.%d.%d\n", i,
               nx_dns_mx_entry_ptr[i] -> nx_dns_mx_ipv4_address >> 24,
               nx_dns_mx_entry_ptr[i] -> nx_dns_mx_ipv4_address >> 16 & 0xFF,
               nx_dns_mx_entry_ptr[i] -> nx_dns_mx_ipv4_address >> 8 & 0xFF,
               nx_dns_mx_entry_ptr[i] -> nx_dns_mx_ipv4_address & 0xFF);

        printf("preference = %d \n ",
               nx_dns_mx_entry_ptr[i] -> nx_dns_mx_preference);

       if(nx_dns_mx_entry_ptr[i] -> nx_dns_mx_hostname_ptr)
               printf("hostname = %s\n",
                      nx_dns_mx_entry_ptr[i] -> nx_dns_mx_hostname_ptr);
       else
               printf("hostname is not set\n");
}


[Output]

-----------------------------------------------------
Test MX: record_count = 5
record 0: IP address: 192.2.2.10
preference = 40
hostname = alt3.aspmx.l.www.my_example.com
record 1: IP address: 192.2.2.11
preference = 50
hostname = alt4.aspmx.l.www.my_example.com
record 2: IP address: 192.2.2.12
preference = 10
hostname = aspmx.l.www.my_example.com
record 3: IP address: 192.2.2.13
preference = 20
hostname = alt1.aspmx.l.www.my_example.com
record 4: IP address: 192.2.2.14
preference = 30
hostname = alt2.aspmx.l.www.my_example.com
```

nx_dns_domain_service_get

Look up the service(s) provided by the input host name

Prototype

```
UINT nx_dns_domain_service_get (NX_DNS *dns_ptr, UCHAR *host_name,
                                VOID *record_buffer, UINT buffer_size,
                                UINT *record_count, ULONG wait_option);
```

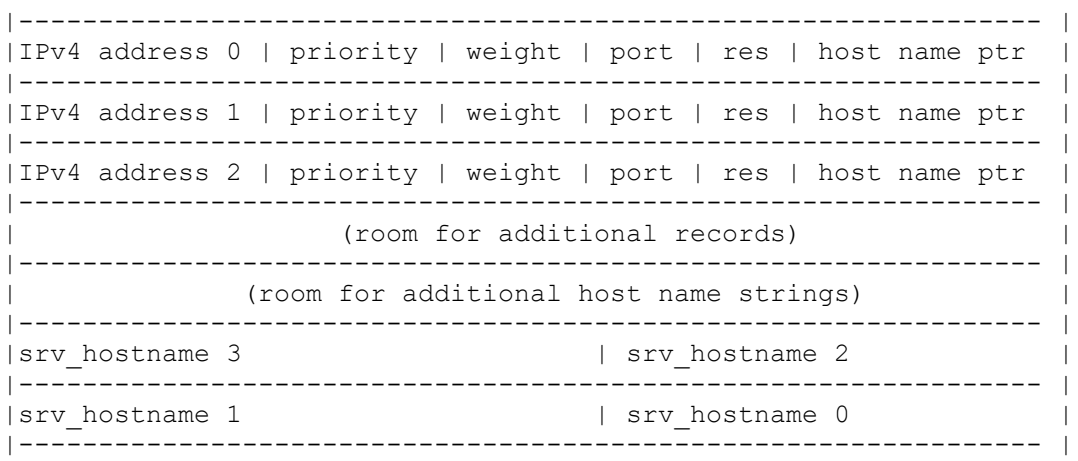
Description

If NX_DNS_ENABLE_EXTENDED_RR_TYPES is defined, this service sends a query of type SRV with the specified domain name to look up the service(s) and their port number associated with the specified domain. The DNS Client copies the SRV record(s) returned in the DNS Server response into the *record_buffer* memory location. Note that *record_buffer* must be 4-byte aligned to receive the data.

In NetX Duo DNS Client, the service record type, NX_DNS_SRV_ ENTRY, is saved as six parameters, totaling 16 bytes. This enables variable length SRV data to be stored in a memory efficient manner:

Server IPv4 address	nx_dns_srv_ipv4_address	4 bytes
Server priority	nx_dns_srv_priority	2 bytes
Server weight	nx_dns_srv_weight	2 bytes
Service port number	nx_dns_srv_port_number	2 bytes
Reserved for 4-byte alignment	nx_dns_srv_reserved0	2 bytes
Pointer to server host name	*nx_dns_srv_hostname_ptr	4 bytes

Four SRV records are stored in the supplied buffer. Each NX_DNS_SRV_ENTRY record contains a pointer, *nx_dns_srv_hostname_ptr*, that points to the corresponding host name string in the bottom of the record buffer:



If the input *record_buffer* cannot hold all the SRV data in the server reply, the the *record_buffer* holds as many records as will fit and returns the number of records in the buffer.

With the number of SRV records returned in **record_count,* the application can parse the SRV parameters, including the server host name of each record in the *record_buffer*.

Input Parameters

dns_ptr	Pointer to DNS Client.
host_name	Pointer to host name to obtain SRV data for
record_buffer	Pointer to location to extract SRV data into
buffer_size	Size of buffer to hold SRV data
record_count	Pointer to the number of SRV records retrieved
wait_option	Wait option to receive DNS Server response

Return Values

NX_SUCCESS	(0x00)	Successfully obtained SRV data
NX_DNS_NO_SERVER	(0xA1)	Client server list is empty
NX_DNS_QUERY_FAILED	(0xA3)	No valid DNS response received
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer parameter.
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
#define MAX_RECORD_COUNT  10

UCHAR  record_buffer[50];
UINT   record_count;
NX_DNS_SRV_ENTRY *nx_dns_srv_entry_ptr[MAX_RECORD_COUNT];

/* Request the service(s) provided by the specified host.  */
status =  nx_dns_domain_service_get(&client_dns, (UCHAR *)"www.my_example.com ",
                                    record_buffer, sizeof(record_buffer),
                                    &record_count, 500);

/* Check for DNS query error.  */
if (status != NX_SUCCESS)
{
    error_counter++;
}

else
```

```
{
    /* If status is NX_SUCCESS a DNS query was successfully completed and SRV data is
        returned in record_buffer.  */

    printf("------------------------------------------------------\n");
    printf("Test SRV: ");
    printf("record_count = %d \n", record_count);


    /* Get the location of services.  */
    for(i =0; i< record_count; i++)
    {
        nx_dns_srv_entry_ptr[i] = (NX_DNS_SRV_ENTRY *)
                                (record_buffer + i * sizeof(NX_DNS_SRV_ENTRY));

        printf("record %d: IP address: %d.%d.%d.%d\n", i,
               nx_dns_srv_entry_ptr[i] -> nx_dns_srv_ipv4_address >> 24,
                nx_dns_srv_entry_ptr[i] -> nx_dns_srv_ipv4_address >> 16 & 0xFF,
                nx_dns_srv_entry_ptr[i] -> nx_dns_srv_ipv4_address >> 8 & 0xFF,
                nx_dns_srv_entry_ptr[i] -> nx_dns_srv_ipv4_address & 0xFF);

       printf("port number = %d\n",
               nx_dns_srv_entry_ptr[i] -> nx_dns_srv_port_number );
       printf("priority = %d\n", nx_dns_srv_entry_ptr[i] -> nx_dns_srv_priority );
       printf("weight = %d\n", nx_dns_srv_entry_ptr[i] -> nx_dns_srv_weight );

       if(nx_dns_srv_entry_ptr[i] -> nx_dns_srv_hostname_ptr)
       {
            printf("hostname = %s\n",
                   nx_dns_srv_entry_ptr[i] -> nx_dns_srv_hostname_ptr);
       }
       else
            printf("hostname is not set\n");
    }
}

[Output]
------------------------------------------------------
Test SRV: record_count = 3
record 0: IP address: 192.2.2.10
port number = 5222
priority = 20
weight = 0
hostname = alt4.xmpp.l.www.my_example.com
record 1: IP address: 192.2.2.11
port number = 5222
priority = 5
weight = 0
hostname = xmpp.l.www.my_example.com
record 2: IP address: 192.2.2.12
port number = 5222
priority = 20
weight = 0
hostname = alt1.xmpp.l.www.my_example.com
```

nx_dns_get_serverlist_size

Return the size of the DNS Client's Server list

Prototype

```
UINT nx_dns_get_serverlist_size (NX_DNS *dns_ptr, UINT *size);
```

Description

This service returns the number of valid DNS Servers (both IPv4 and IPv6) in the Client list.

Input Parameters

dns_ptr	Pointer to DNS control block
size	Returns the number of servers in the list

Return Values

NX_SUCCESS	(0x00)	DNS Server list size successfully returned
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
UINT my_listsize;

/* Get the number of non null DNS Servers in the Client list.  */
status =  nx_dns_get_serverlist_size (&my_dns, 5, &my_listsize);

/* If status is NX_SUCCESS the size of the DNS Server list was successfully
   returned.  */
```

nx_dns_info_by_name_get

Return ip address and port of DNS server by host name

Prototype

```
UINT nx_dns_info_by_name_get(NX_DNS *dns_ptr, UCHAR *host_name,
                             ULONG *host_address_ptr,
                             USHORT *host_port_ptr, ULONG wait_option);
```

Description

This service returns the Server IP and port (service record) based on the input host name by DNS query. If a service record is not found, this routine returns a zero IP address in the input address pointer and a non-zero error status return to signal an error.

Input Parameters

dns_ptr	Pointer to DNS control block
host_name	Pointer to host name buffer
host_address_ptr	Pointer to address to return
host_port_ptr	Pointer to port to return
wait_option	Wait option for the DNS response

Return Values

NX_SUCCESS	(0x00)	DNS Server record successfully returned
NX_DNS_NO_SERVER	(0xA1)	No DNS Server registered with Client to send query on hostname
NX_DNS_QUERY_FAILED	(0xA3)	DNS query failed; no response from any DNS servers in Client list or no service record is available for the input hostname.
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer
NX_CALLER_ERROR	(0x11)	Invalid caller

Allowed From

Threads

Example

```
ULONG ip_address
USHORT port;

/* Attempt to resolve the IP address and ports for this host name.  */
status =  nx_dns_info_by_name_get(&my_dns, "www.abc1234.com", &ip_address, &port,
200);

/* If status is NX_SUCCESS the DNS query was successful and the IP address and
   report for the hostname are returned.  */
```

nx_dns_ipv4_address_by_name_get

Look up the IPv4 address for the input host name

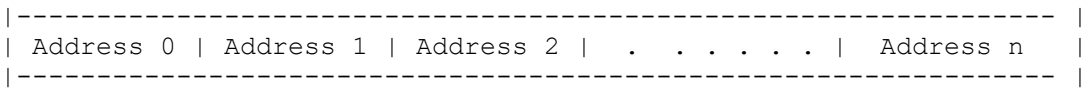
Prototype

```
UINT nx_ dns_ipv4_address_by_name_get (NX_DNS *dns_ptr,
                                       UCHAR *host_name_ptr, VOID *buffer,
                                       UINT buffer_size,
                                       UINT *record_count,
                                       ULONG wait_option);
```

Description

This service sends a query of Type A with the specified host name to obtain the IP addresses for the input host name. The DNS Client copies the IPv4 address from the A record(s) returned in the DNS Server response into the *record_buffer* memory location. Note that *record_buffer* must be 4-byte aligned to receive the data.

Multiple IPv4 addresses are stored in the 4-byte aligned buffer as shown below:



If the supplied buffer cannot hold all the IP address data, the remaining A records are not stored in *record_buffer*. This enables the application to retrieve one, some or all of the available IP address data in the server reply.

With the number of A records returned in **record_count* the application can parse the IPv4 address data from the *record_buffer*.

Input Parameters

dns_ptr	Pointer to DNS Client.
host_name_ptr	Pointer to host name to obtain IPv4 address
buffer	Pointer to location to extract IPv4 data into
buffer_size	Size of buffer to hold IPv4 data
wait_option	Wait option to receive DNS Server response

Return Values

NX_SUCCESS	(0x00)	Successfully obtained IPv4 data
NX_DNS_NO_SERVER	(0xA1)	Client server list is empty
NX_DNS_QUERY_FAILED		

	(0xA3)	No valid DNS response received
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer parameter.

Allowed From

Threads

Example

```
#define MAX_RECORD_COUNT  20

ULONG           record_buffer[50];
UINT            record_count;
ULONG           *ipv4_address_ptr[MAX_RECORD_COUNT];

/* Request the IPv4 address for the specified host.  */
status =  nx_dns_ipv4_address_by_name_get(&client_dns,
                                          (UCHAR *)"www.my_example.com",
                                           record_buffer,
                                           sizeof(record_buffer),&record_count,
                                           500);

 /* Check for DNS query error.  */
if (status != NX_SUCCESS)
{
        error_counter++;
}
 else
{
        /* If status is NX_SUCCESS a DNS query was successfully completed the IPv4
           address(es) is returned in record_buffer.  */


        printf("-------------------------------------------------------\n");
        printf("Test A: ");
        printf("record_count = %d \n", record_count);


       /* Get the IPv4 addresses of host.  */
       for(i =0; i< record_count; i++)
       {
            ipv4_address_ptr[i] = (ULONG *)(record_buffer + i * sizeof(ULONG));
            printf("record %d: IP address: %d.%d.%d.%d\n", i,
                *ipv4_address_ptr[i] >> 24,
                *ipv4_address_ptr[i] >> 16 & 0xFF,
                *ipv4_address_ptr[i] >> 8 & 0xFF,
                *ipv4_address_ptr[i] & 0xFF);
        }

}

[Output]

-----------------------------------------------------
Test A: record_count = 5
record 0: IP address: 192.2.2.10
record 1: IP address: 192.2.2.11
record 2: IP address: 192.2.2.12
record 3: IP address: 192.2.2.13
record 4: IP address: 192.2.2.14
```

nxd_dns_ipv6_address_by_name_get

Look up the IPv6 address for the input host name

Prototype

```
UINT nxd_ dns_ipv6_address_by_name_get(NX_DNS *dns_ptr,
                                       UCHAR *host_name_ptr, VOID *buffer,
                                       UINT buffer_size,
                                       UINT *record_count,
                                       ULONG wait_option);
```

Description

This service sends a query of type AAAA with the specified domain name to obtain the IP addresses for the input domain name. The DNS Client copies the IPv6 address from the AAAA record(s) returned in the DNS Server response into the *record_buffer* memory location. Note that *record_buffer* must be 4-byte aligned to receive the data.

The format of IPv6 addresses stored in the 4-byte aligned buffer is shown below:

```
|-------------------------------------------------------------------------------|
| IPv6_address_0[0]  | IPv6_address_0[1]  | IPv6_address_0[2]  | IPv6_address_0[3] |
|-------------------------------------------------------------------------------|
| IPv6_address_1[0]  | IPv6_address_1[1]  | IPv6_address_1[2]  | IPv6_address_1[3] |
|-------------------------------------------------------------------------------|
| IPv6_address_2[0]  | IPv6_address_2[1]  | IPv6_address_2[2]  | IPv6_address_2[3] |
|-------------------------------------------------------------------------------|
| IPv6_address_n[0]  | IPv6_address_n[1]  | IPv6_address_n[2]  | IPv6_address_n[3] |
|-------------------------------------------------------------------------------|
```

If the input *record_buffer* cannot hold all the AAAA data in the server reply, the the *record_buffer* holds as many records as will fit and returns the number of records in the buffer.

With the number of AAAA records returned in **record_count,* the application can parse the IPv6 addresses from each record in the *record_buffer*.

Input Parameters

dns_ptr	Pointer to DNS Client.
host_name_ptr	Pointer to host name to obtain IPv6 address
buffer	Pointer to location to extract IPv6 data into
buffer_size	Size of buffer to hold IPv6 data
wait_option	Wait option to receive DNS Server response

Return Values

NX_SUCCESS	(0x00)	Successfully obtained IPv6 data

NX_DNS_NO_SERVER	(0xA1)	Client server list is empty
NX_DNS_QUERY_FAILED	(0xA3)	No valid DNS response received
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer parameter.

Allowed From

Threads

Example

```
#define         MAX_RECORD_COUNT  20

ULONG           record_buffer[50];
UINT            record_count;
NXD_ADDRESS    *ipv6_address_ptr[MAX_RECORD_COUNT];

/* Request the IPv4 address for the specified host.  */
status =  nxd_dns_ipv6_address_by_name_get(&client_dns,
                                           (UCHAR *)"www.my_example.com",
                                           record_buffer,
                                           sizeof(record_buffer),
                                           &record_count, 500);

 /* Check for DNS query error.  */
if (status != NX_SUCCESS)
{
        error_counter++;
}
 else
{
    /* If status is NX_SUCCESS a DNS query was successfully completed the IPv6
       address(es) is (are) returned in record_buffer.  */

    printf("-----------------------------------------------------\n");
    printf("Test AAAA: ");
    printf("record_count = %d \n", record_count);

    /* Get the IPv6 addresses of host.  */
    for(i =0; i< record_count; i++)
    {

        ipv6_address_ptr[i] =
            (NX_DNS_IPV6_ADDRESS *)(record_buffer + i * sizeof(NX_DNS_IPV6_ADDRESS));

        printf("record %d: IP address: %x:%x:%x:%x:%x:%x:%x:%x\n", i,
                ipv6_address_ptr[i] -> ipv6_address[0]  >>16 & 0xFFFF,
                ipv6_address_ptr[i] -> ipv6_address[0]  & 0xFFFF,
                ipv6_address_ptr[i] -> ipv6_address[1]  >>16 & 0xFFFF,
                ipv6_address_ptr[i] -> ipv6_address[1]  & 0xFFFF,
                ipv6_address_ptr[i] -> ipv6_address[2]  >>16 & 0xFFFF,
                ipv6_address_ptr[i] -> ipv6_address[2]  & 0xFFFF,
               ipv6_address_ptr[i] -> ipv6_address[3]  >>16 & 0xFFFF,
               ipv6_address_ptr[i] -> ipv6_address[3]  & 0xFFFF);
    }
}
```

```
[Output]
-----------------------------------------------------
Test AAAA: record_count = 1
record 0: IP address: 2001:0db8:0000:f101: 0000: 0000: 0000:01003
```

nx_dns_host_by_address_get

Look up a host name from an IP address

Prototype

```
UINT nx_dns_host_by_address_get(NX_DNS *dns_ptr, ULONG ip_address,
                                ULONG *host_name_ptr,
                                ULONG max_host_name_size,
                                ULONG wait_option);
```

Description

This service requests name resolution of the supplied IP address from one or more DNS Servers previously specified by the application. If successful, the NULL-terminated host name is returned in the string specified by *host_name_ptr*. This is a wrapper function for *nxd_dns_host_by_address_get* service and does not accept IPv6 addresses.

Input Parameters

dns_ptr	Pointer to previously created DNS instance.
ip_address	IP address to resolve into a name
host_name_ptr	Pointer to destination area for host name
max_host_name_size	Size of destination area for host name
wait_option	Defines how long the service will wait in timer ticks for a DNS server response after each DNS query and query retry. The wait options are defined as follows: timeout value (0x00000001-0xFFFFFFFE) TX_WAIT_FOREVER (0xFFFFFFFF) Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until a DNS server responds to the request. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the DNS resolution.

Return Values

NX_SUCCESS	(0x00)	Successful DNS resolution
NX_DNS_TIMEOUT	(0xA2)	Timed out on obtaining DNS mutex
NX_DNS_NO_SERVER	(0xA1)	No DNS Server address specified
NX_DNS_QUERY_FAILED	(0xA3)	Received no response to query
NX_DNS_BAD_ADDRESS_ERROR	(0xA4)	Null input address
NX_DNS_INVALID_ADDRESS_TYPE	(0xB2)	Index points to invalid address type (e.g. IPv6)
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer input
NX_DNS_IPV6_NOT_SUPPORTED	(0xB3)	Cannot process record with IPv6 disabled
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
#define BUFFER_SIZE   200

UCHAR  resolved_name[200];

/* Get the name associated with IP address 192.2.2.10.  */
status =  nx_dns_host_by_address_get(&my_dns, IP_ADDRESS(192.2.2.10),
                                     &resolved_name[0], BUFFER_SIZE, 450);

/* If status is NX_SUCCESS the name associated with the IP address
   can be found in the resolved_name variable.  */
```

nxd_dns_host_by_address_get

Look up a host name from the IP address

Prototype

```
UINT nxd_dns_host_by_address_get(NX_DNS *dns_ptr,
                                 NXD_ADDRESS ip_address,
                                 ULONG *host_name_ptr,
                                 ULONG max_host_name_size,
                                 ULONG wait_option);
```

Description

This service requests name resolution of the IPv6 or IPv4 address in the *ip_address* input argument from one or more DNS Servers previously specified by the application. If successful, the NULL-terminated host name is returned in the string specified by *host_name_ptr*.

Input Parameters

dns_ptr	Pointer to previously created DNS instance.
ip_address	IP address to resolve into a name
host_name_ptr	Pointer to destination area for host name
max_host_name_size	Size of destination area for host name
wait_option	Defines how long the service will wait in timer ticks for a DNS server response after each DNS query and query retry. The wait options are defined as follows: timeout value(0x00000001 through 0xFFFFFFFE) TX_WAIT_FOREVER (0xFFFFFFFF) Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until a DNS server responds to the request. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the DNS resolution.

Return Values

NX_SUCCESS	(0x00)	Successful DNS resolution
NX_DNS_TIMEOUT	(0xA2)	Timed out on obtaining DNS mutex
NX_DNS_NO_SERVER	(0xA1)	No DNS Server address specified
NX_DNS_QUERY_FAILED		

	(0xA3)	Received no response to query
NX_DNS_BAD_ADDRESS_ERROR		
	(0xA4)	Null input address
NX_DNS_IPV6_NOT_SUPPORTED		
	(0xB3)	Cannot process record with IPv6 disabled
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer input

Allowed From

Threads

Example

```
UCHAR   resolved_name[200];
NXD_ADDRESS host_address;

host_address.nxd_ip_version = NX_IP_VERISON_V6;
host_address.nxd_ip_address.v6[0] = 0x20010db8;
host_address.nxd_ip_address.v6[1] = 0x0;
host_address.nxd_ip_address.v6[2] = 0xf101;
host_address.nxd_ip-address.v6[3] = 0x108;

/* Get the name associated with theinput host_address.  */
status =  nxd_dns_host_by_address_get(&my_dns, &host_address,
                                      resolved_name, sizeof(resolved_name), 4000);

/* Check for DNS query error.  */
if (status != NX_SUCCESS)
{
     error_counter++;
}
else
{
     printf("-----------------------------------------------------\n");
     printf("Test PTR: %s\n", record_buffer);
}

/* If status is NX_SUCCESS the name associated with the IP address
   can be found in the resolved_name variable.  */
```

[Output]

```
 -----------------------------------------------------
 Test PTR: my_example.net
```

nx_dns_host_by_name_get

Look up an IP address from the host name

Prototype

```
UINT nx_dns_host_by_name_get(NX_DNS *dns_ptr, ULONG *host_name,
                             ULONG *host_address_ptr, ULONG wait_option);
```

Description

This service requests name resolution of the supplied name from one or more DNS Servers previously specified by the application. If successful, the associated IP address is returned in the destination pointed to by *host_address_ptr*. This is a wrapper function for the *nxd_dns_host_by_name_get* service, and is limited to IPv4 address input.

Input Parameters

dns_ptr	Pointer to previously created DNS instance.
host_name_ptr	Pointer to host name
host_address_ptr	Pointer to destination for IP address
wait_option	Defines how long the service will wait for the DNS resolution. The wait options are defined as follows: timeout value (0x00000001 through 0xFFFFFFFE) TX_WAIT_FOREVER (0xFFFFFFFF) Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until a DNS server responds to the request. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the DNS resolution.

Return Values

NX_SUCCESS	(0x00)	Successful DNS resolution.
NX_DNS_NO_SERVER	(0xA1)	No DNS Server address specified
NX_DNS_QUERY_FAILED	(0xA3)	Received no response to query
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer input
NX_PTR_ERROR	(0x07)	Invalid pointer input

NX_CALLER_ERROR (0x11) Invalid caller of this service

Allowed From

Threads

Example

```
ULONG ip_address;

/* Get the IP address for the name "www.my_example.com".  */
status =  nx_dns_host_by_name_get(&my_dns, "www.my_example.com", &ip_address, 4000);

/* Check for DNS query error.  */
if (status != NX_SUCCESS)
{
    error_counter++;
}

else
{

    /* If status is NX_SUCCESS the IP address for "www.my_example.com" can be found
       in the "ip_address" variable.  */

    printf("------------------------------------------------------\n");
    printf("Test A: \n");
    printf("IP address: %d.%d.%d.%d\n",
    host_ip_address >> 24,
    host_ip_address >> 16 & 0xFF,
    host_ip_address >> 8 & 0xFF,
    host_ip_address & 0xFF);
}

[Output]
 ------------------------------------------------------
Test A:
IP address: 192.2.2.10
```

nxd_dns_host_by_name_get

Lookup an IP address from the host name

Prototype

```
UINT nxd_dns_host_by_name_get(NX_DNS *dns_ptr, ULONG *host_name,
                              NXD_ADDRESS *host_address_ptr,
                              ULONG wait_option, UINT lookup_type);
```

Description

This service requests name resolution of the supplied IP address from one or more DNS Servers previously specified by the application. If successful, the associated IP address is returned in an NXD_ADDRESS pointed to by `host_address_ptr`. If the caller specifically sets the `lookup_type` input to NX_IP_VERSION_V6, this service will send out query for a host IPv6 address (AAAA record). If the caller specifically sets the `lookup_type` input to NX_IP_VERSION_V4, this service will send out query for a host IPv4 address (A record).

Input Parameters

dns_ptr	Pointer to previously created DNS Client instance.
host_name_ptr	Pointer to host name to find an IP address of
host_address_ptr	Pointer to destination for NXD_ADDRESS containing the IP address
lookup_type	Indicate type of lookup (A vs AAAA).
wait_option	Defines how long the service will wait in timer ticks for the DNS Server response for each query transmission and retransmission. The wait options are defined as follows: timeout value (0x00000001 through 0xFFFFFFFE) TX_WAIT_FOREVER (0xFFFFFFFF) Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until a DNS Server responds to the request. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the DNS resolution.

Return Values

NX_SUCCESS	(0x00)	Successful DNS resolution.
NX_DNS_NO_SERVER	(0xA1)	No DNS Server address specified
NX_DNS_QUERY_FAILED	(0xA3)	Received no response to query
NX_DNS_BAD_ADDRESS_ERROR	(0xA4)	Null input address
NX_DNS_IPV6_NOT_SUPPORTED	(0xB3)	Cannot process record with IPv6 disabled
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer input

Allowed From

Threads

Example

```
NXD_ADDRESS host_ipduo_address;

/* Create an AAAA query to obtain the IPv6 address for the host "www.my_example.com".
*/
status =  nxd_dns_host_by_name_get(&my_dns, "www.my_example.com", &
host_ipduo_address, 4000,
                                    NX_IP_VERSION_V6);

if (status != NX_SUCCESS)
{
        error_counter++;
}
else
{
   /* If status is NX_SUCCESS the IP address for "www.my_example.com" can be found in
      the "ip_address" variable.  */

    printf("-----------------------------------------------------\n");
    printf("Test AAAA: \n");

    printf("IP address: %x:%x:%x:%x:%x:%x:%x:%x\n",
           host_ipduo_address.nxd_ip_address.v6[0]  >>16 & 0xFFFF,
           host_ipduo_address.nxd_ip_address.v6[0]  & 0xFFFF,
           host_ipduo_address.nxd_ip_address.v6[1]  >>16 & 0xFFFF,
           host_ipduo_address.nxd_ip_address.v6[1]  & 0xFFFF,
           host_ipduo_address.nxd_ip_address.v6[2]  >>16 & 0xFFFF,
           host_ipduo_address.nxd_ip_address.v6[2]  & 0xFFFF,
           host_ipduo_address.nxd_ip_address.v6[3]  >>16 & 0xFFFF,
           host_ipduo_address.nxd_ip_address.v6[3]  & 0xFFFF);
}

[Output]

-----------------------------------------------------
Test AAAA:
IP address: 2607:f8b0:4007:800:0:0:0:1008
```

Another example of using this time service, this time using IPv4 addresses and A record types, is shown below:

```
/* Create a query to obtain the IPv4 address for the host "www.my_example.com".  */
status =  nxd_dns_host_by_name_get(&my_dns, "www.my_example.com", &ip_address, 4000,
                                   NX_IP_VERSION_V4);

/* Check for DNS query error.  */
if (status != NX_SUCCESS)
{
    error_counter++;
}
else
{
     /* If status is NX_SUCCESS the IP address for "www.my_example.com" can be found
        in the "ip_address" variable.  */

     printf("-----------------------------------------------------\n");
     printf("Test A: \n");
     printf("IP address: %d.%d.%d.%d\n",
            host_ipduo_address.nxd_ip_address.v4 >> 24,
            host_ipduo_address.nxd_ip_address.v4 >> 16 & 0xFF,
            host_ipduo_address.nxd_ip_address.v4 >> 8 & 0xFF,
            host_ipduo_address.nxd_ip_address.v4 & 0xFF);
 }

[Output]

-----------------------------------------------------
Test A:
IP address: 192.2.2.10
```

nx_dns_host_text_get

Look up the text string for the input domain name

Prototype

```
UINT nx_dns_host_text_get(NX_DNS *dns_ptr, UCHAR *host_name,
                          UCHAR *record_buffer,
                          UINT buffer_size, ULONG wait_option);
```

Description

This service sends a query of type TXT with the specified domain name and buffer to obtain the arbitrary string data.

The DNS Client copies the text string in the TXT record in the DNS Server response into the *record_buffer* memory location. Note that `record_buffer` does not need to be 4-byte aligned to receive the data.

Input Parameters

dns_ptr	Pointer to DNS Client.
host_name	Pointer to name of host to search on
record_buffer	Pointer to location to extract TXT data into
buffer_size	Size of buffer to hold TXT data
wait_option	Wait option to receive DNS Server response

Return Values

NX_SUCCESS	(0x00)	Successfully TXT string obtained
NX_DNS_NO_SERVER	(0xA1)	Client server list is empty
NX_DNS_QUERY_FAILED	(0xA3)	No valid DNS response received
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer input

Allowed From

Threads

Example

```
CHAR            record_buffer[50];

/* Request the text string for the specified host.  */
status =  nx_dns_host_text_get(&client_dns, (UCHAR *)"www.my_example.com",
                               record_buffer,
                               sizeof(record_buffer), 500);


/* Check for DNS query error.  */
if (status != NX_SUCCESS)
{
     error_counter++;
}
else
{
     /* If status is NX_SUCCESS a DNS query was successfully completed and the text
        string is returned in record_buffer.  */

     printf("-----------------------------------------------------\n");
     printf("Test TXT:\n %s\n", record_buffer);
}


[Output]


-----------------------------------------------------
Test TXT:
v=spf1 include:_www.my_example.com ip4:192.2.2.10/31 ip4:192.2.2.11/31 ~all
```

nx_dns_packet_pool_set

Set the DNS Client packet pool

Prototype

```
UINT nx_dns_packet_pool_set(NX_DNS *dns_ptr, NX_PACKET_POOL *pool_ptr);
```

Description

This service sets a previously created packet pool as the DNS Client packet pool. The DNS Client will use this packet pool to send DNS queries, so the packet payload should not be less than NX_DNS_PACKET_PAYLOAD which includes the Ethernet, IP and UDP headers and is defined in *nxd_dns.h.* Note that *w*hen the DNS Client is deleted, the packet pool is not deleted with it and it is the responsibility of the application to delete the packet pool when it no longer needs it.

Note: this service is only available if the configuration option NX_DNS_CLIENT_USER_CREATE_PACKET_POOL is defined in *nxd_dns.h*

Input Parameters

dns_ptr	Pointer to previously created DNS Client instance.
pool_ptr	Pointer to previously created packet pool

Return Values

NX_SUCCESS	(0x00)	Successful completion.
NX_NOT_ENABLED	(0x14)	Client not configured for this option
NX_PTR_ERROR	(0x07)	Invalid IP or DNS Client pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
NXD_DNS my_dns;
NX_PACKET_POOL client_pool;
NX_IP *ip_ptr;


/* Create the DNS Client.  */
status =  nx_dns_create(&my_dns, ip_ptr, "My DNS Client");

/* Create a  packet pool for the DNS Client.  */
status =  nx_packet_pool_create(&client_pool, "DNS Client Packet Pool",
                                NX_DNS_PACKET_PAYLOAD, free_mem_pointer,
                                NX_DNS_PACKET_POOL_SIZE);

/* Set the DNS Client  packet pool.  */
status =  nx_dns_packet_pool_set(&my_dns, &client_pool);

/* If status is NX_SUCCESS the DNS Client  packet pool was successfully set.  */
```

nx_dns_server_add

Add DNS Server IP Address

Prototype

```
UINT nx_dns_server_add(NX_DNS *dns_ptr, ULONG server_address);
```

Description

This service adds an IPv4 DNS Server to the server list.

Input Parameters

dns_ptr	Pointer to DNS control block.
server_address	IP address of DNS Server

Return Values

NX_SUCCESS	(0x00)	Server successfully added
NX_DNS_DUPLICATE_ENTRY		
NX_NO_MORE_ENTRIES	(0x17)	No more DNS Servers Allowed (list is full)
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer input
NX_DNS_IPV6_NOT_SUPPORTED	(0xB3)	Cannot process record with IPv6 disabled
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_DNS_BAD_ADDRESS_ERROR	(0xA4)	Null server address input

Allowed From

Threads

Example

```
/* Add a DNS Server at IP address 202.2.2.13.  */
status =  nx_dns_server_add(&my_dns, IP_ADDRESS(202,2,2,13));

/* If status is NX_SUCCESS a DNS Server was successfully added.  */
```

nxd_dns_server_add

Add DNS Server to the Client list

Prototype

```
UINT nxd_dns_server_add(NX_DNS *dns_ptr, NXD_ADDRESS *server_address);
```

Description

This service adds the IP address of a DNS server to the DNS Client server list. The `server_address` may be either an IPv4 or IPv6 address. If the Client wishes to be able to access the same server by either its IPv4 address or IPv6 address it should add both IP addresses as entries to the server list.

Input Parameters

dns_ptr	Pointer to DNS control block.
server_address	Pointer to the NXD_ADDRESS containing the server IP address of DNS Server.

Return Values

NX_SUCCESS	(0x00)	Server successfully added
NX_DNS_DUPLICATE_ENTRY		
NX_NO_MORE_ENTRIES	(0x17)	No more DNS Servers allowed (list is full)
NX_DNS_IPV6_NOT_SUPPORTED	(0xB3)	Cannot process record with IPv6 disabled
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer input
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_DNS_BAD_ADDRESS_ERROR	(0xA4)	Null server address input
NX_DNS_INVALID_ADDRESS_TYPE	(0xB2)	Index points to invalid address type (e.g. IPv6)

Allowed From

Threads

Example

```
NXD_ADDRESS server_address;

server_address.nxd_ip_version = NX_IP_VERISON_V6;
server_address.nxd_ip_address.v6[0] = 0x20010db8;
server_address.nxd_ip_address.v6[1] = 0x0;
server_address.nxd_ip_address.v6[2] = 0xf101;
server_address.nxd_ip-address.v6[3] = 0x108;


/* Add a DNS Server with the IP address pointed to by the server_address input.  */
status =  nxd_dns_server_add(&my_dns, &server_address);

/* If status is NX_SUCCESS a DNS Server was successfully added.  */
```

nx_dns_server_get

Return an IPv4 DNS Server from the Client list

Prototype

```
UINT nx_dns_server_get(NX_DNS *dns_ptr, UINT index,
                       ULONG *dns_server_address);
```

Description

This service returns the IPv4 DNS Server address from the server list at the specified index. Note that the index is zero based. If the input index exceeds the size of the DNS Client list, an IPv6 address is found at that index or a null address is found at the specified index, an error is returned. The *nx_dns_get_serverlist_size* service may be called first obtain the number of DNS servers in the Client list.

This service does only supports IPv4 addresses. It calls the *nxd_dns_server_get* service which supports both IPv4 and IPv6 addresses.

Input Parameters

dns_ptr	Pointer to DNS control block
index	Index into DNS Client's list of servers
dns_server_address	Pointer to IP address of DNS Server

Return Values

NX_SUCCESS	(0x00)	Successful server returned
NX_DNS_SERVER_NOT_FOUND	(0xA9)	Index points to empty slot
NX_DNS_BAD_ADDRESS_ERROR	(0xA4)	Index points to Null address
NX_DNS_INVALID_ADDRESS_TYPE	(0xB2)	Index points to invalid address type (e.g. IPv6)
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non-pointer input
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
ULONG my_server_address;

/* Get the DNS Server at index 5 (zero based) into the Client list.  */
status =  nx_dns_server_get(&my_dns, 5, &my_server_addres);

/* If status is NX_SUCCESS a DNS Server was successfully
   returned.  */
```

nxd_dns_server_get

Return a DNS Server from the Client list

Prototype

```
UINT nxd_dns_server_get(NX_DNS *dns_ptr, UINT index,
                        NXD_ADDRESS *dns_server_address);
```

Description

This service returns the DNS Server IP address from the server list at the specified index. Note that the index is zero based. If the input index exceeds the size of the DNS Client list, or a null address is found at the specified index, an error is returned. The *nx_dns_get_serverlist_size* service may be called first to obtain the number of DNS servers in the server list.

This service supports IPv4 and IPv6 addresses.

Input Parameters

dns_ptr	Pointer to DNS control block
index	Index into DNS Client's list of servers
dns_server_address	Pointer to IP address of DNS Server

Return Values

NX_SUCCESS	(0x00)	Successfully returned server IP address
NX_DNS_SERVER_NOT_FOUND	(0xA9)	Index points to empty slot
NX_DNS_BAD_ADDRESS_ERROR	(0xA4)	Index points to null server address
NX_DNS_INVALID_ADDRESS_TYPE	(0xB2)	Index points to invalid address type (e.g. IPv6)
NX_DNS_PARAM_ERROR	(0xA8)	Invalid non pointer input
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
NXD_ADDRESS my_server_address;

/* Get the DNS Server at index 5 (zero based) into the Client list.  */
status =  nxd_dns_server_get(&my_dns, 5, &my_server_addres);

/* If status is NX_SUCCESS a DNS Server was successfully
   returned.  */
```

nx_dns_server_remove

Remove an IPv4 DNS Server from the Client list

Prototype

```
UINT nx_dns_server_remove(NX_DNS *dns_ptr, ULONG server_address);
```

Description

This service removes an IPv4 DNS Server from the Client list. It is a wrapper function for *nxd_dns_server_remove*.

Input Parameters

dns_ptr	Pointer to DNS control block.
server_address	IP address of DNS Server.

Return Values

NX_SUCCESS	(0x00)	DNS Server successfully removed
NX_DNS_SERVER_NOT_FOUND	(0xA9)	Server not in Client list
NX_DNS_BAD_ADDRESS_ERROR	(0xA4)	Null server address input
NX_DNS_IPV6_NOT_SUPPORTED	(0xB3)	Cannot process record with IPv6 disabled
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Remove the DNS Server at IP address is 202.2.2.13.  */
status =  nx_dns_server_remove(&my_dns, IP_ADDRESS(202,2,2,13));

/* If status is NX_SUCCESS a DNS Server was successfully
   removed.  */
```

nxd_dns_server_remove

Remove a DNS Server from the Client list

Prototype

```
UINT nxd_dns_server_remove(NX_DNS *dns_ptr, NXD_ADDRESS *server_address);
```

Description

This service removes a DNS Server of the specified IP address from the Client list. The input IP address accepts both IPv4 and IPv6 addresses. After the server is removed, the remaining servers move down one index in the list to fill the vacated slot.

Input Parameters

dns_ptr	Pointer to DNS control block.
server_address	Pointer to DNS Server NXD_ADDRESS data containing server IP address.

Return Values

NX_SUCCESS	(0x00)	DNS Server successfully removed
NX_DNS_SERVER_NOT_FOUND	(0xA9)	Server not in Client list
NX_DNS_BAD_ADDRESS_ERROR	(0xA4)	Null server address input
NX_DNS_IPV6_NOT_SUPPORTED	(0xB3)	Cannot process record with IPv6 disabled
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service
NX_DNS_INVALID_ADDRESS_TYPE	(0xB2)	Index points to invalid address type (e.g. IPv6)

Allowed From

Threads

Example

```
NXD_ADDRESS server_address;

server_address.nxd_ip_version = NX_IP_VERISON_V6;
server_address.nxd_ip_address.v6[0] = 0x20010db8;
server_address.nxd_ip_address.v6[1] = 0x0;
server_address.nxd_ip_address.v6[2] = 0xf101;
server_address.nxd_ip-address.v6[3] = 0x108;


/* Remove the DNS Server at the specified IP address from the Client list.  */
status =  nxd_dns_server_remove(&my_dns,&server_ADDRESS);

/* If status is NX_SUCCESS a DNS Server was successfully removed.  */
```

nx_dns_server_remove_all

Remove all DNS Servers from the Client list

Prototype

```
UINT nx_dns_server_remove_all(NX_DNS *dns_ptr);
```

Description

This service removes all DNS Servers from the Client list.

Input Parameters

dns_ptr	Pointer to DNS control block.

Return Values

NX_SUCCESS	(0x00)	DNS Servers successfully removed
NX_DNS_ERROR	(0xA0)	Unable to obtain protection mutex
NX_PTR_ERROR	(0x07)	Invalid IP or DNS pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Remove all DNS Servers from the Client list.  */
status =  nx_dns_server_remove_all(&my_dns);

/* If status is NX_SUCCESS all DNS Servers were successfully removed.  */
```