User's Manual

NetX Duo™

Hypertext Transfer Protocol (NetX Duo HTTP)

User Guide

Renesas Synergy™ Platform

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

Renesas Electronics
www.renesas.com

Feb 25, 2019

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit: www.renesas.com/contact/.

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Renesas Synergy Specific Information

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

Multipart Support

Page 12: Multipart support for the HTTP client has not been tested for SSP v1.5.0.

Multi-Thread Support

Page 12: Multi-thread support for the HTTP client has not been tested for SSP v1.5.0.

Product Distribution

Page 13: The distribution of HTTP included with the Renesas Synergy SSP installation does not include the file **demo_netxduo_http.c**. Please ignore references to this file.

HTTP Installation

Page 13: If you are using Renesas Synergy SSP and the e^2 studio ISDE, HTTP will already be installed. You can ignore the Installation and Use of HTTP section.

Hypertext Transfer Protocol (NetX Duo HTTP)

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

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

Trademarks

Warranty Limitations

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

Part Number: 000-1054
Revision 5.11SP1

Contents

Chapter 1 Introduction to HTTP 5
HTTP Requirements..5
HTTP Constraints..5
HTTP URL (Resource Names)..6
HTTP Client Requests..6
HTTP Server Responses 7
HTTP Communication..7
HTTP Authentication 8
HTTP Authentication Callback..9
HTTP Invalid Username/Password Callback..10
HTTP Insert GMT Date Header Callback..11
HTTP Cache Info Get Callback 11
HTTP Multipart Support..12
HTTP Multi-Thread Support 12
HTTP RFCs..12
Chapter 2 Installation and Use of HTTP 13
Product Distribution..13
HTTP Installation..13
Using HTTP..13
Small Example System 14
Configuration Options..22
Chapter 3 Description of HTTP Services 26
nx_http_client_create 29
nx_http_client_delete 31
nx_http_client_get_start 32
nxd_http_client_get_start 33
nx_http_client_get_packet..37
nx_http_client_put_start 39
nxd_http_client_put_start 41
nx_http_client_put_packet..43
nx_http_client_set_connect_port..45
nx_http_server_cache_info_callback_set..46
nx_http_server_callback_data_send..48
nx_http_server_callback_generate_response_header..50
nx_http_server_callback_packet_send 52
nx_http_server_callback_response_send 54
nx_http_server_content_get..56
nx_http_server_content_get_extended 58
nx_http_server_content_length_get 60
nx_http_server_content_length_get_extended 61
nx_http_server_create..62

nx_http_server_delete 64
nx_http_server_get_entity_content 65
nx_http_server_get_entity_header 67
nx_http_server_gmt_callback_set 69
nx_http_server_invalid_userpassword_notify_set 70
nx_http_server_mime_maps_additional_set 72
nx_http_server_packet_content_find 74
nx_http_server_packet_get 76
nx_http_server_param_get 77
nx_http_server_query_get 79
nx_http_server_start 81
nx_http_server_stop 82
nx_http_server_type_get 83

Chapter 1

Introduction to HTTP

The Hypertext Transfer Protocol (HTTP) is a protocol designed for transferring content on the Web. HTTP is a simple protocol that utilizes reliable Transmission Control Protocol (TCP) services to perform its content transfer function. Because of this, HTTP is a highly reliable content transfer protocol. HTTP is one of the most used application protocols. All operations on the Web utilize the HTTP protocol. NetX Duo HTTP accommodates both IPv4 and IPv6 networks. IPv6 does not directly change the HTTP protocol, although some changes in the original NetX HTP API are necessary to accommodate IPv6 and will be described in this document.

HTTP Requirements

In order to function properly, the NetX Duo HTTP package requires that a NetX Duo (version 5.2 or later) is installed. In addition, an IP instance must already be created and TCP must be enabled on that same IP instance. An IPv6 host application must set its link local and global IPv6 address using the IPv6 API and/or DHCPv6. The demo file in section “Small Example System” in **Chapter 2** will demonstrate how this is done.

The HTTP Client portion of the NetX Duo HTTP package has no further requirements.

The HTTP Server portion of the NetX Duo HTTP package has several additional requirements. First, it requires complete access to TCP *well-known port 80* for handling all Client HTTP requests. The HTTP Server is also designed for use with the FileX embedded file system. If FileX is not available, the user may port the portions of FileX used to their own environment. This is discussed in later sections of this guide.

HTTP Constraints

The NetX Duo HTTP protocol implements the HTTP 1.0 standard. However, there are following constraints:

1. Persistent connections are not supported
2. Request pipelining is not supported
3. The HTTP Server supports both basic and MD5 digest authentication, but not MD5-sess. At present, the HTTP Client supports only basic authentication.
4. No content compression is supported.
5. TRACE, OPTIONS, and CONNECT requests are not supported.
6. The packet pool associated with the HTTP Server or Client must be large enough to hold the complete HTTP header.
7. HTTP Client services are for content transfer only—there are no display utilities provided in this package.

HTTP URL (Resource Names)

The HTTP protocol is designed to transfer content on Web. The requested content is specified by the Universal Resource Locator (URL). This is the primary component of every HTTP request. URLs always start with a “/” character and typically correspond to files on the HTTP Server. Common HTTP file extensions are shown below:

Extension	**Meaning**
.htm (or .html)	Hypertext Markup Language (HTML)
.txt	Plain ASCII text
.gif	Binary GIF image
.xbm	Binary Xbitmap image

HTTP Client Requests

The HTTP has a simple mechanism for requesting Web content. There is basically a set of standard HTTP commands that are issued by the Client after a connection has been successfully established on the TCP *well-known port 80*. The following shows some of the basic HTTP commands:

HTTP Command	**Meaning**
GET resource HTTP/1.0	*Get the specified resource*
POST resource HTTP/1.0	*Get the specified resource and pass attached input to the HTTP Server*

HEAD resource HTTP/1.0	*Treated like a GET but not content is returned by the HTTP Server*
PUT resource HTTP/1.0	*Place resource on HTTP Server*
DELETE resource HTTP/1.0	*Delete resource on the Server*

These ASCII commands are generated internally by Web browsers and the NetX HTTP Client services to perform HTTP operations with an HTTP Server.

Note that the HTTP Client application default to the connect port of 80. However, it can change the connect port to the HTTP Server at runtime using the *nx_http_client_set_connect_port* service. See Chapter 4 for more details of this service. This is to accommodate web servers that occasionally use alternate ports for Client connections.

HTTP Server Responses

The HTTP Server utilizes the same *well-known TCP port 80* to send Client command responses. Once the HTTP Server processes the Client command, it returns an ASCII response string that includes a 3-digit numeric status code. The numeric response is used by the HTTP Client software to determine whether the operation succeeded or failed. Following is a list of various HTTP Server responses to Client commands:

Numeric Field	**Meaning**
200	*Request was successful*
400	*Request was not formed properly*
401	*Unauthorized request, client needs to send authentication*
404	*Specified resource in request was not found*
500	*Internal HTTP Server error*
501	*Request not implemented by HTTP Server*
502	*Service is not available*

For example, a successful Client request to PUT the file “test.htm” is responded with the message “HTTP/1.0 200 OK.”

HTTP Communication

As mentioned previously, the HTTP Server utilizes the *well-known TCP port 80* to field Client requests. HTTP Clients may use any available TCP port. The general sequence of HTTP events is as follows:

HTTP GET Request:

1. Client issues TCP connect to Server port 80.
2. Client sends "**GET resource HTTP/1.0**" request (along with other header information).
3. Server builds an "**HTTP/1.0 200 OK**" message with additional information followed immediately by the resource content (if any).
4. Server performs a disconnection.
5. Client performs a disconnection.

HTTP PUT Request:

1. Client issues TCP connect to Server port 80.

2. Client sends "**PUT resource HTTP/1.0**" request, along with other header information, and followed by the resource content.
3. Server builds an "**HTTP/1.0 200 OK**" message with additional information followed immediately by the resource content.
4. Server performs a disconnection.
5. Client performs a disconnection.

Note: as mentioned previously, the HTTP Client can change the default connect port from 80 to another port using the *nx_http_client_set_connect_port* for web servers that use alternate ports to connect to clients.

HTTP Authentication

HTTP authentication is optional and isn't required for all Web requests. There are two flavors of authentication, namely *basic* and *digest*. Basic authentication is equivalent to the *name* and *password* authentication found in many protocols. In HTTP basic authentication, the name and passwords are concatenated and encoded in the base64 format. The main disadvantage of basic authentication is the name and password are transmitted openly in the request. This makes it somewhat easy for the name and password to be stolen. Digest authentication addresses this problem by never transmitting the name and password in the request. Instead, an algorithm is used to derive a 128-bit key or digest from the name, password, and other information. The NetX HTTP Server supports the standard MD5 digest algorithm.

When is authentication required? Basically, the HTTP Server decides if a requested resource requires authentication. If authentication is required and the Client request did not include the proper authentication, a “HTTP/1.0 401 Unauthorized” response with the type of authentication required is sent to the Client. The Client is then expected to form a new request with the proper authentication.

HTTP Authentication Callback

As mentioned before, HTTP authentication is optional and isn’t required on all Web transfers. In addition, authentication is typically resource dependent. Access of some resources on the Server require authentication, while others do not. The NetX HTTP Server package allows the application to specify (via the *nx_http_server_create* call) an authentication callback routine that is called at the beginning of handling each HTTP Client request.

The callback routine provides the NetX HTTP Server with the username, password, and realm strings associated with the resource and return the type of authentication necessary. If no authentication is necessary for the resource, the authentication callback should return the value of **NX_HTTP_DONT_AUTHENTICATE**. Otherwise, if basic authentication is required for the specified resource, the routine should return **NX_HTTP_BASIC_AUTHENTICATE**. And finally, if MD5 digest authentication is required, the callback routine should return **NX_HTTP_DIGEST_AUTHENTICATE**. If no authentication is required for any resource provided by the HTTP Server, the callback is not needed and a NULL pointer can be provided to the HTTP Server create call.

The format of the application authenticate callback routine is very simple and is defined below:

```
UINT nx_http_server_authentication_check(NX_HTTP_SERVER *server_ptr,
                                          UINT request_type, CHAR *resource,
                                          CHAR **name, CHAR **password,
                                          CHAR **realm);
```

The input parameters are defined as follows:

Parameter	**Meaning**

request_type Specifies the HTTP Client request, valid requests are defined as:

NX_HTTP_SERVER_GET_REQUEST
NX_HTTP_SERVER_POST_REQUEST
NX_HTTP_SERVER_HEAD_REQUEST
NX_HTTP_SERVER_PUT_REQUEST
NX_HTTP_SERVER_DELETE_REQUEST

resource Specific resource requested.

name Destination for the pointer to the required username.

password Destination for the pointer to the required password.

realm Destination for the pointer to the realm for this authentication.

The return value of the authentication routine specifies if authentication is required. `name, password, and realm` pointers are not used if **NX_HTTP_DONT_AUTHENTICATE** is returned by the authentication callback routine. Otherwise the HTTP server developer must ensure that **NX_HTTP_MAX_USERNAME** and **NX_HTTP_MAX_PASSWORD** defined in *nxd_http_server.h* are large enough for the username and password specified in the authentication callback. These are both defaulted to size 20 chars.

HTTP Invalid Username/Password Callback

The optional invalid username/password callback in NetX HTTP Server is invoked if HTTP server receives an invalid username and password combination in a Client request. If the HTTP server application registers a callback with HTTP server it will be invoked if either basic or digest authentication fails *in nx_http_server_get_process*, in *nx_http_server_put_process,* or *in nx_http_server_delete_process.*

To register a callback with the HTTP server, the following service is defined in NetX Duo HTTP Server.

```
UINT nx_http_server_invalid_userpassword_notify_set(
            NX_HTTP_SERVER *http_server_ptr,
            UINT *invalid_username_password_callback)
                        (CHAR *resource,
                         NXD_ADDRESS *client_nxd_address,
                         UINT request_type))
```

The request types are defined as follows:

NX_HTTP_SERVER_GET_REQUEST
NX_HTTP_SERVER_POST_REQUEST
NX_HTTP_SERVER_HEAD_REQUEST
NX_HTTP_SERVER_PUT_REQUEST
NX_HTTP_SERVER_DELETE_REQUEST

HTTP Insert GMT Date Header Callback

There is an optional callback in NetX Duo HTTP Server to insert a date header in its response messages. This callback is invoked when the HTTP Server is responding to a put or get request

To register a GMT date callback with the HTTP server, the following service is defined in the NetX Duo HTTP Server.

```
UINT  _nx_http_server_gmt_callback_set(
                      NX_HTTP_SERVER *server_ptr,
                      VOID (*gmt_get)(NX_HTTP_SERVER_DATE *date)
```

The NX_HTTP_SERVER_DATE data type is defined as follows:

```
typedef struct NX_HTTP_SERVER_DATE_STRUCT
{
    USHORT          nx_http_server_year;           /* Year                 */
    UCHAR           nx_http_server_month;          /* Month                */
    UCHAR           nx_http_server_day;            /* Day                  */
    UCHAR           nx_http_server_hour;           /* Hour                 */
    UCHAR           nx_http_server_minute;         /* Minute               */
    UCHAR           nx_http_server_second;         /* Second               */
    UCHAR           nx_http_server_weekday;        /* Weekday              */
} NX_HTTP_SERVER_DATE;
```

HTTP Cache Info Get Callback

The HTTP Server has a callback to request the max age and date from the HTTP application for a specific resource. This information is used to determine if the HTTP server sends the entire page in response to a Client Get request. If the “if modified since” in the Client request is not found or does not match the “last modified” date returned by the get cache callback, the entire page is sent.

To register the callback with the HTTP server the following service is defined:

```
UINT  _nx_http_server_cache_info_callback_set(
                              NX_HTTP_SERVER *server_ptr,
                              UINT (*cache_info_get)
                                    (CHAR *, UINT *, NX_HTTP_SERVER_DATE *))
```

HTTP Multipart Support

Multipurpose Internet Mail Extensions (MIME) was originally intended for the SMTP protocol, but its use has spread to HTTP. MIME allows messages to contain mixed message types (e.g. image/jpg and text/plain) within the same message. NetX Duo HTTP Server has added services to determine content type in HTTP messages containing MIME from the Client. To enable HTTP multipart support and use these services, the configuration option **NX_HTTP_MULTIPART_ENABLE** must be defined.

```
UINT  nx_http_server_get_entity_header(NX_HTTP_SERVER *server_ptr,
                                        NX_PACKET **packet_pptr,
                                        UCHAR *entity_header_buffer,
                                        ULONG buffer_size);

UINT  nx_http_server_get_entity_content(NX_HTTP_SERVER *server_ptr,
                                         NX_PACKET **packet_pptr,
                                         ULONG *available_offset,
                                         ULONG *available_length)
```

For more details on the use of these services, see their description in Chapter 3 “Description of HTTP Services”.

HTTP Multi-Thread Support

The NetX HTTP Client services can be called from multiple threads simultaneously. However, read or write requests for a particular HTTP Client instance should be done in sequence from the same thread.

HTTP RFCs

NetX HTTP is compliant with RFC1945 “Hypertext Transfer Protocol/1.0, RFC 2581 “TCP Congestion Control”, RFC 1122 “Requirements for Internet Hosts”, and related RFCs.

Chapter 2

Installation and Use of HTTP

This chapter contains a description of various issues related to installation, setup, and usage of the NetX HTTP component.

Product Distribution

HTTP for NetX is shipped on a single CD-ROM compatible disk. The package includes three source files, two include files, and a file that contains this document, as follows:

nxd_http_client.h	Header file for HTTP Client for NetX Duo
nxd_http_server.h	Header file for HTTP Server for NetX Duo
nxd_http_client.c	C Source file for HTTP Client for NetX Duo
nxd_http_server.c	C Source file for HTTP Server for NetX Duo
nx_md5.c	MD5 digest algorithms
filex_stub.h	Stub file if FileX is not present
nxd_http.pdf	Description of HTTP for NetX Duo
demo_netxduo_http.c	NetX Duo HTTP demonstration

HTTP Installation

In order to use HTTP for NetX Duo, the entire distribution mentioned previously should be copied to the same directory where NetX Duo is installed. For example, if NetX Duo is installed in the directory "*\threadx\arm7\green*" then the *nxd_http_client.h* and *nxd_http_client.*c for NetX Duo HTTP Client applications*,* and *nxd_http_server.h* and *nxd_http_server.c* for NetX Duo HTTP Server applications*. nx_md5.c* should be copied into this directory. For the demo 'ram driver' application NetX Duo HTTP Client and Server files should be copied into the same directory.

Using HTTP

Using HTTP for NetX Duo is easy. Basically, the application code must include *nxd_http_client.h* and/or *nxd_http_server.h* after it includes *tx_api.h, fx_api.h,* and *nx_api.h*, in order to use ThreadX, FileX, and NetX Duo, respectively. Once the HTTP header files are included, the application code is then able to make the HTTP function calls specified later in this guide. The application must also include *nxd_http_client.c*, *nxd_http_server.c*, and *md5.c* in the build process. These files 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 HTTP.

Note that if NX_HTTP_DIGEST_ENABLE is not specified in the build process, the *md5.c* file does not need to be added to the application. Similarly, if no HTTP Client capabilities are required, the *nxd_http_client.c* file may be omitted.

Note also that since HTTP utilizes NetX Duo TCP services, TCP must be enabled with the *nx_tcp_enable* call prior to using HTTP.

Small Example System

An example of how easy it is to use NetX Duo HTTP is described in Figure 1.1 that appears below. This example works with the ‘duo’ services available in NetX Duo HTTP placement of #define USE_DUO on line 23. Otherwise it uses the legacy NetX HTTP equivalent (limited to IPv4 only). Developers are encouraged to migrate existing applications to using the NetX Duo HTTP services.

To specify IPv6 communication, the application defines IPTYPE to IPv6 in line 24.

In this example, the HTTP include files *nxd_http_client.h* and *nxd_http_server.h* are brought in at line 8 and 9. Next, the helper HTTP Server thread, packet pool and IP instance are created in lines 89 – 112. The HTTP Server IP instance must be TCP enabled, as seen in line 137. The HTTP Server is then itself is created in at line 159.

Next the HTTP Client is created. First the client thread is created in line 172 followed by packet pool and IP instance, similar to the HTTP Server, in lines 186 – 200. Again the HTTP Client IP instance must be TCP enabled (line 217).

The HTTP Server thread runs and its first task is validate its IP address with NetX Duo which it does in lines 423 - 450. Now the HTTP Server is ready to take requests.

The HTTP Client thread‘s first task is create and format the FileX media (lines 236 and 260. After the media is initialized, the HTTP Client is created in line 271. This must be done before the HTTP server can service HTTP requests. It must then validate its IP address with NetX Duo which it does in lines 282 – 316. The HTTP Client then creates and sends the file client_test.html to the HTTP Server, waits briefly, then attempts to read the file back from the HTTP Server.

Note that the HTTP Client API uses a different service if IPv6 is not enabled (*nx_http_client_put_start* in line 343 and *nx_http_client_get_start* in line 399). This enables NetX Duo to support existing NetX HTTP Client applications.

Note that the HTTP Client API calls are made with relatively short timeouts. It may be necessary to extend those timeouts if an HTTP client is communicating with a busy server or remote server on a slower processor.

```
1     /* This is a small demo of the NetX Duo HTTP Client Server API running on a
2        high-performance NetX Duo TCP/IP stack.  This demo is applicable for
3        either IPv4 or IPv6 enabled applications.  */
4
5     #include   "tx_api.h"
6     #include   "fx_api.h"
7     #include   "nx_api.h"
8     #include   "nxd_http_client.h"
9     #include   "nxd_http_server.h"
10
11    #define     DEMO_STACK_SIZE         2048
12
13    /* Set up FileX and file memory resources. */
14    CHAR            *ram_disk_memory;
15    FX_MEDIA        ram_disk;
16    unsigned char   media_memory[512];
17
18    /* Define device drivers.  */
19    extern void _fx_ram_driver(FX_MEDIA *media_ptr);
20    VOID        _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr);
22
23    #define USE_DUO          /* Use the duo service (not legacy netx) */
24    #define IPTYPE   6       /* Send packets over IPv6 */
25
26    /* Set up the HTTP client. */
27    TX_THREAD       client_thread;
28    NX_PACKET_POOL  client_pool;
29    NX_HTTP_CLIENT  my_client;
30    NX_IP           client_ip;
31    #define         CLIENT_PACKET_SIZE  (NX_HTTP_SERVER_MIN_PACKET_SIZE * 2)
32    void            thread_client_entry(ULONG thread_input);
33
34    #define HTTP_SERVER_ADDRESS  IP_ADDRESS(1,2,3,4)
35    #define HTTP_CLIENT_ADDRESS  IP_ADDRESS(1,2,3,5)
36
37    /* Set up the HTTP server */
38
39    NX_HTTP_SERVER  my_server;
40    NX_PACKET_POOL  server_pool;
41    TX_THREAD       server_thread;
42    NX_IP           server_ip;
43    #define         SERVER_PACKET_SIZE  (NX_HTTP_SERVER_MIN_PACKET_SIZE * 2)
44
45    void            thread_server_entry(ULONG thread_input);
46    #ifdef FEATURE_NX_IPV6
47    NXD_ADDRESS     server_ip_address;
48    #endif
49
50
```

```
/* Define the application's authentication check.  This is called by
   the HTTP server whenever a new request is received.  */
UINT  authentication_check(NX_HTTP_SERVER *server_ptr, UINT request_type,
            CHAR *resource, CHAR **name, CHAR **password, CHAR **realm)
{

    /* Just use a simple name, password, and realm for all
       requests and resources.  */
    *name =     "name";
    *password = "password";
    *realm =    "NetX Duo HTTP demo";

    /* Request basic authentication.  */
    return(NX_HTTP_BASIC_AUTHENTICATE);
}

/* Define main entry point.  */

int main()
{

    /* Enter the ThreadX kernel.  */
    tx_kernel_enter();
}


/* Define what the initial system looks like.  */
void    tx_application_define(void *first_unused_memory)
{

CHAR    *pointer;
UINT    status;


    /* Setup the working pointer.  */
    pointer =  (CHAR *) first_unused_memory;

    /* Create a helper thread for the server. */
    tx_thread_create(&server_thread, "HTTP Server thread", thread_server_entry, 0,
                     pointer, DEMO_STACK_SIZE,
                     1, 1, TX_NO_TIME_SLICE, TX_AUTO_START);

    pointer =  pointer + DEMO_STACK_SIZE;

    /* Initialize the NetX system.  */
    nx_system_initialize();

    /* Create the server packet pool.  */
    status =  nx_packet_pool_create(&server_pool, "HTTP Server Packet Pool",
          SERVER_PACKET_SIZE, pointer, SERVER_PACKET_SIZE*4);


    pointer = pointer + SERVER_PACKET_SIZE * 4;

    /* Check for pool creation error.  */
    if (status)
    {

        return;
    }

    /* Create an IP instance.  */
    status = nx_ip_create(&server_ip, "HTTP Server IP", HTTP_SERVER_ADDRESS,
                          0xFFFFFF00UL, &server_pool, _nx_ram_network_driver,
                          pointer, 4096, 1);

    pointer =  pointer + 4096;

    /* Check for IP create errors.  */
    if (status)
    {
        printf("nx_ip_create failed. Status 0x%x\n", status);
        return;
    }

    /* Enable ARP and supply ARP cache memory for the server IP instance.  */
    status = nx_arp_enable(&server_ip, (void *) pointer, 1024);

    /* Check for ARP enable errors.  */
    if (status)
    {
        return;
    }

    pointer = pointer + 1024;

     /* Enable TCP traffic.  */
    status = nx_tcp_enable(&server_ip);

    if (status)
    {
        return;
    }

#if (IP_TYPE==6)

    /* Set up HTTPv6 server, but we have to wait till its address has been
       validated before we can start the thread_server_entry thread. */

    /* Set up the server's IPv6 address here. */
    server_ip_address.nxd_ip_address.v6[3] = 0x105;
    server_ip_address.nxd_ip_address.v6[2] = 0x0;
    server_ip_address.nxd_ip_address.v6[1] = 0x0000f101;
    server_ip_address.nxd_ip_address.v6[0] = 0x20010db8;
    server_ip_address.nxd_ip_version = NX_IP_VERSION_V6;

#endif

    /* Create the NetX HTTP Server.  */
    status = nx_http_server_create(&my_server, "My HTTP Server", &server_ip,
                 &ram_disk, pointer, 2048, &server_pool, authentication_check,
                 NX_NULL);

    if (status)
    {
        return;
    }

    pointer =  pointer + 2048;

    /* Save the memory pointer for the RAM disk.  */
    ram_disk_memory =  pointer;

    /* Create the HTTP client thread. */
    status = tx_thread_create(&client_thread, "HTTP Client", thread_client_entry, 0,
                     pointer, DEMO_STACK_SIZE,
                     2, 2, TX_NO_TIME_SLICE, TX_AUTO_START);

    pointer =  pointer + DEMO_STACK_SIZE;

    /* Check for thread create error.  */
    if (status)
    {

        return;
    }

    /* Create the Client packet pool.  */
    status =  nx_packet_pool_create(&client_pool, "HTTP Client Packet Pool",
           SERVER_PACKET_SIZE, pointer, SERVER_PACKET_SIZE*4);


    pointer = pointer + SERVER_PACKET_SIZE * 4;

    /* Check for pool creation error.  */
    if (status)
    {

        return;
    }


    /* Create an IP instance.  */
    status = nx_ip_create(&client_ip, "HTTP Client IP", HTTP_CLIENT_ADDRESS,
                          0xFFFFFF00UL, &client_pool, _nx_ram_network_driver,
                          pointer, 2048, 1);

    pointer =  pointer + 2048;

    /* Check for IP create errors.  */
    if (status)
    {
        return;
    }

    nx_arp_enable(&client_ip, (void *) pointer, 1024);

    pointer =  pointer + 2048;

     /* Enable TCP traffic.  */
    nx_tcp_enable(&client_ip);

    return;
}


VOID thread_client_entry(ULONG thread_input)
{

UINT            status;
NX_PACKET       *my_packet;
#ifdef FEATURE_NX_IPV6
NXD_ADDRESS     client_ip_address;
UINT            address_index;
#endif


    /* Format the RAM disk - the memory for the RAM disk was setup in
      tx_application_define above.  This must be set up before the client(s) start
      sending requests. */
    status = fx_media_format(&ram_disk,
                            _fx_ram_driver,         // Driver entry
                            ram_disk_memory,        // RAM disk memory pointer
                            media_memory,           // Media buffer pointer
                            sizeof(media_memory),   // Media buffer size
                            "MY_RAM_DISK",          // Volume Name
                            1,                      // Number of FATs
                            32,                     // Directory Entries
                            0,                      // Hidden sectors
                            256,                    // Total sectors
                            128,                    // Sector size
                            1,                      // Sectors per cluster
                            1,                      // Heads
                            1);                     // Sectors per track

    /* Check the media format status.  */
    if (status != FX_SUCCESS)
    {

        /* Error, bail out.  */
        return ;
    }

    /* Open the RAM disk.  */
    status =  fx_media_open(&ram_disk, "RAM DISK", _fx_ram_driver, ram_disk_memory,
                         media_memory, sizeof(media_memory));

    /* Check the media open status.  */
    if (status != FX_SUCCESS)
    {

        /* Error, bail out.  */
        return ;
    }

    /* Create an HTTP client instance.  */
    status = nx_http_client_create(&my_client, "HTTP Client", &client_ip,
                                &client_pool, 600);

    /* Check status.  */
    if (status != NX_SUCCESS)
    {
        return;
    }

    /* Attempt to upload a file to the HTTP server. */


#if (IPTYPE== 6)

    /* Relinquish control so the HTTP server can get set up...*/
    tx_thread_relinquish();

    /* Set up the client's IPv6 address here. */
    client_ip_address.nxd_ip_address.v6[3] = 0x101;
    client_ip_address.nxd_ip_address.v6[2] = 0x0;
    client_ip_address.nxd_ip_address.v6[1] = 0x0000f101;
    client_ip_address.nxd_ip_address.v6[0] = 0x20010db1;
    client_ip_address.nxd_ip_version = NX_IP_VERSION_V6;

    /* Here's where we make the HTTP Client IPv6 enabled. */

    nxd_ipv6_enable(&client_ip);
    nxd_icmp_enable(&client_ip);

    /* Wait till the IP task thread has set the device MAC address. */
    tx_thread_sleep(100);

    /* Now update NetX Duo the Client's link local and global IPv6 address. */
    nxd_ipv6_address_set(&server_ip, 0, NX_NULL, 10, &address_index)
    nxd_ipv6_ address_set(&server_ip, 0, &client_ip_address, 64, &address_index);

    /* Then make sure NetX Duo has had time to validate the addresses. */

    tx_thread_sleep(400);

    /* Now upload an HTML file to the HTTPv6 server. */
    status =  nxd_http_client_put_start(&my_client, &server_ip_address,
                 "/client_test.htm", “name", "password", 103, 500);


    /* Check status.  */
    if (status != NX_SUCCESS)
    {

        return;
    }


#else

    /* Relinquish control so the HTTP server can get set up...*/
    tx_thread_relinquish();

    do
    {

        /* Attempt to upload to the HTTP IPv4 server. */
        status =  nx_http_client_put_start(&my_client, HTTP_SERVER_ADDRESS,
                 "/client_test.htm", "name", "password", 103, 500);


        /* Check status.  */
        if (status != NX_SUCCESS)
        {
            tx_thread_sleep(100);
        }

    } while (status != NX_SUCCESS);


#endif  /* (IPTYPE== 6) */


    /* Allocate a packet.  */
    status =  nx_packet_allocate(&client_pool, &my_packet, NX_TCP_PACKET,
                                    NX_WAIT_FOREVER);

    /* Check status.  */
    if (status != NX_SUCCESS)
    {
        return;
    }

    /* Build a simple 103-byte HTML page.  */
    nx_packet_data_append(my_packet, "<HTML>\r\n", 8,
                        &client_pool, NX_WAIT_FOREVER);
    nx_packet_data_append(my_packet,
                 "<HEAD><TITLE>NetX HTTP Test</TITLE></HEAD>\r\n", 44,
                        &client_pool, NX_WAIT_FOREVER);
    nx_packet_data_append(my_packet, "<BODY>\r\n", 8,
                        &client_pool, NX_WAIT_FOREVER);
    nx_packet_data_append(my_packet, "<H1>Another NetX Test Page!</H1>\r\n", 25,
                        &client_pool, NX_WAIT_FOREVER);
    nx_packet_data_append(my_packet, "</BODY>\r\n", 9,
                        &client_pool, NX_WAIT_FOREVER);
    nx_packet_data_append(my_packet, "</HTML>\r\n", 9,
                        &client_pool, NX_WAIT_FOREVER);

    /* Complete the PUT by writing the total length.  */
    status =  nx_http_client_put_packet(&my_client, my_packet, 50);

    /* Check status.  */
    if (status != NX_SUCCESS)
    {
        return;
    }

    /* Now GET the test file  */

#ifdef USE_DUO

    status =  nxd_http_client_get_start(&my_client, &server_ip_address,
                   "/client_test.htm", NX_NULL, 0, "name", "password", 50);
#else

    status =  nx_http_client_get_start(&my_client, HTTP_SERVER_ADDRESS,
                   "/client_test.htm",  NX_NULL, 0, "name", "password", 50);

#endif

    /* Check status.  */
    if (status != NX_SUCCESS)
    {
        return;
    }

    status = nx_http_client_delete(&my_client);

    return;
}

/* Define the helper HTTP server thread.  */
void    thread_server_entry(ULONG thread_input)
{

UINT            status;
#if (IPTYPE == 6)
UINT            address_index
NXD_ADDRESS     ip_address

    /* Allow time for the IP task to initialize the driver. */
    tx_thread_sleep(100);

  ip_address.nxd_ip_version = NX_IP_VERSION_V6;
  ip_address.nxd_ip_address.v6[0] = 0x20010000;
  ip_address.nxd_ip_address.v6[1] = 0;
  ip_address.nxd_ip_address.v6[2] = 0;
  ip_address.nxd_ip_address.v6[3] = 4;

    /* Here's where we make the HTTP server IPv6 enabled. */
    nxd_ipv6_enable(&server_ip);
    nxd_icmp_enable(&server_ip);

    /* Wait till the IP task thread has set the device MAC address. */
    while (server_ip.nx_ip_arp_physical_address_msw == 0 ||
           server_ip.nx_ip_arp_physical_address_lsw == 0)
    {
        tx_thread_sleep(30);
    }

    nxd_ipv6_address_set(&server_ip, 0, NX_NULL, 10, &address_index)
    nxd_ipv6_ address_set(&server_ip, 0, &ip_address, 64, &address_index);

    /* Wait for NetX Duo to validate server address. */
    tx_thread_sleep(400);

#endif  /* (IPTYPE == 6) */

    /* OK to start the HTTPv6 Server.   */
    status = nx_http_server_start(&my_server);

    if (status != NX_SUCCESS)
    {
        return;
    }

    /* HTTP server ready to take requests! */

    /* Let the IP threads execute.    */
    tx_thread_relinquish();

    return;
}
```

Figure 1.1 Example of HTTP use with NetX Duo

Configuration Options

There are several configuration options for building HTTP for NetX Duo. Following is a list of all options, where each is described in detail. The default values are listed, but can be redefined prior to inclusion of *nxd_http_client.h and nxd_http_server.h*:

Define	Meaning
NX_DISABLE_ERROR_CHECKING	Defined, this option removes the basic HTTP error checking. It is typically used after the application has been debugged.
NX_HTTP_SERVER_PRIORITY	The priority of the HTTP Server thread. By default, this value is defined as 16 to specify priority 16.
NX_HTTP_NO_FILEX	Defined, this option provides a stub for FileX dependencies. The HTTP Client will function without any change if this option is defined. The HTTP Server will need to either be modified or the user will have to create a handful of FileX services in order to function properly.
NX_HTTP_TYPE_OF_SERVICE	Type of service required for the HTTP TCP requests. By default, this value is defined as NX_IP_NORMAL to indicate normal IP packet service.
NX_HTTP_SERVER_THREAD_TIME_SLICE	The number of timer ticks the Server thread is allowed to run before yielding to threads of the same priority. The default value is 2.
NX_HTTP_FRAGMENT_OPTION	Fragment enable for HTTP TCP requests. By default, this value is

NX_DONT_FRAGMENT to disable HTTP TCP fragmenting.

NX_HTTP_SERVER_WINDOW_SIZE Server socket window size. By default, this value is 2048 bytes.

NX_HTTP_TIME_TO_LIVE Specifies the number of routers this packet can pass before it is discarded. The default value is set to 0x80.

NX_HTTP_SERVER_TIMEOUT Specifies the number of ThreadX ticks that internal services will suspend for. The default value is set to 10 seconds (10 * NX_IP_PERIODIC_RATE).

.

NX_HTTP_SERVER_TIMEOUT_ACCEPT

Specifies the number of ThreadX ticks that internal services will suspend for in internal *nx_tcp_server_socket_accept* calls. The default value is set to (10 * NX_IP_PERIODIC_RATE).

NX_HTTP_SERVER_TIMEOUT_DISCONNECT

Specifies the number of ThreadX ticks that internal services will suspend for in internal *nx_tcp_socket_disconnect* calls. The default value is set to 10 seconds (10 * NX_IP_PERIODIC_RATE).

.

NX_HTTP_SERVER_TIMEOUT_RECEIVE

Specifies the number of ThreadX ticks that internal services will suspend for in internal *nx_tcp_socket_receive* calls. The default value is set to 10 seconds (10 * NX_IP_PERIODIC_RATE).

.

NX_HTTP_SERVER_TIMEOUT_SEND

Specifies the number of ThreadX ticks that internal services will suspend for in internal *nx_tcp_socket_send* calls. The default value is set to 10 seconds (10 * NX_IP_PERIODIC_RATE).

.

NX_HTTP_MAX_HEADER_FIELD Specifies the maximum size of the HTTP header field. The default value is 256.

NX_HTTP_MULTIPART_ENABLE If defined, enables HTTP Server to support multipart HTTP requests.

NX_HTTP_SERVER_MAX_PENDING Specifies the number of connections that can be queued for the HTTP Server. The default value is set to 5.

NX_HTTP_MAX_RESOURCE Specifies the number of bytes allowed in a client supplied *resource name*. The default value is set to 40.

NX_HTTP_MAX_NAME Specifies the number of bytes allowed in a client supplied *username*. The default value is set to 20.

NX_HTTP_MAX_PASSWORD Specifies the number of bytes allowed in a client supplied *password*. The default value is set to 20.

NX_HTTP_SERVER_MIN_PACKET_SIZE
Specifies the minimum size of the packets in the pool specified at Server creation. The minimum size is needed to ensure the complete HTTP header can be contained in one packet. The default value is set to 600.

NX_HTTP_CLIENT_MIN_PACKET_SIZE

Specifies the minimum size of the packets in the pool specified at Client creation. The minimum size is needed to ensure the complete HTTP header can be contained in one packet. The default value is set to 300*.*

NX_HTTP_SERVER_RETRY_SECONDS

Set the Server socket retransmission timeout in seconds. The default value is set to 2.

NX_HTTP_ SERVER_ RETRY_MAX

This sets the maximum number of retransmissions on Server socket. The default value is set to 10*.*

NX_HTTP_ SERVER_ RETRY_SHIFT

This value is used to set the next retransmission timeout. The current timeout is multiplied by the number of retransmissions thus far, shifted by the value of the socket timeout shift. The default value is set to 1 for doubling the timeout*.*

NX_HTTP_ SERVER_RETRY_TRANSMIT_QUEUE_DEPTH

This specifies the maximum number of packets that can be enqueued on the Server socket retransmission queue. If the number of packets enqueued reaches this number, no more packets can be sent until one or more enqueued packets are released. The default value is set to 20*.*

Chapter 3

Description of HTTP Services

This chapter contains a description of all NetX Duo HTTP services (listed below) in alphabetical order except for the ‘NetX’ (IPv4 only) equivalent of the same service are paired together).

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.

HTTP Client services:

nx_http_client_create
Create an HTTP Client Instance

nx_http_client_delete
Delete an HTTP Client instance

nx_http_client_get_start
Start an HTTP GET request (IPv4 only)

nxd_http_client_get_start
Start an HTTP GET request (IPv4 or IPv6)

nx_http_client_get_packet
Get next resource data packet

nx_http_client_put_start
Start an HTTP PUT request (IPv4 only)

nxd_http_client_put_start
Start an HTTP PUT request (IPv4 or IPv6)

nx_http_client_put_packet
Send next resource data packet

nx_http_client_set_connect_port
Change the port to connect to the HTTP Server

HTTP server services:

nx_http_server_cache_info_callback_set
Set callback to retrieve age and last modified date of specified URL

nx_http_server_callback_data_send
Send HTTP data from callback function

nx_http_server_callback_generate_response_header
Create response header in callback functions

nx_http_server_callback_packet_send
Send an HTTP packet from an HTTP callback

nx_http_server_callback_response_send
Send response from callback function

nx_http_server_content_get
Get content from the request

nx_http_server_content_get_extended
Get content from the request; supports empty (zero Content Length) requests

nx_http_server_content_length_get
Get length of content in the request

nx_http_server_content_length_get_extended
Get length of content in the request; supports empty (zero Content Length) requests

nx_http_server_create
Create an HTTP Server instance

nx_http_server_delete
Delete an HTTP Server instance

nx_http_server_get_entity_content
Return size and location of entity content in URL

nx_http_server_get_entity_header
Extract URL entity header into specified buffer

nx_http_server_gmt_callback_set
Set callback to retrieve GMT date and time

nx_http_server_invalid_userpassword_notify_set
Set callback for when invalid username and password is received in a Client request
nx_http_server_mime_maps_additional_set
Define additional mime maps for HTML

nx_http_server_packet_content_find
Extract content length in HTTP header and set pointer to start of content data

nx_http_server_packet_get
Receive client packet directly

nx_http_server_param_get
Get parameter from the request

nx_http_server_query_get
Get query from the request

nx_http_server_start
Start the HTTP Server

nx_http_server_stop
Stop the HTTP Server

nx_http_server _type_get
Extract HTTP type e.g. text/plain from header

nx_http_client_create

Create an HTTP Client Instance

Prototype

```
UINT nx_http_client_create(NX_HTTP_CLIENT *client_ptr,
            CHAR *client_name, NX_IP *ip_ptr, NX_PACKET_POOL *pool_ptr,
            ULONG window_size);
```

Description

This service creates an HTTP Client instance on the specified IP instance.

Input Parameters

client_ptr	Pointer to HTTP Client control block.
client_name	Name of HTTP Client instance.
ip_ptr	Pointer to IP instance.
pool_ptr	Pointer to default packet pool. Note that the packets in this pool must have a payload large enough to handle the complete response header. This is defined by NX_HTTP_CLIENT_MIN_PACKET_SIZE in *nx_http.h*.
window_size	Size of the Client's TCP socket receive window.

Return Values

NX_SUCCESS	(0x00)	Successful HTTP Client create
NX_PTR_ERROR	(0x07)	Invalid HTTP, ip_ptr, or packet pool pointer
NX_HTTP_POOL_ERROR	(0xE9)	Invalid payload size in packet pool

Allowed From

Initialization, Threads

Example

```
/* Create the HTTP Client instance “my_client” on “ip_0”.  */
status =  nx_http_client_create(&my_client, “my client”, &ip_0, &pool_0, 100);


/* If status is NX_SUCCESS an HTTP Client instance was successfully
   created.  */
```

nx_http_client_delete

Delete an HTTP Client Instance

Prototype

```
UINT nx_http_client_delete(NX_HTTP_CLIENT *client_ptr);
```

Description

This service deletes a previously created HTTP Client instance.

Input Parameters

client_ptr	Pointer to HTTP Client control block.

Return Values

NX_SUCCESS	(0x00)	Successful HTTP Client delete
NX_PTR_ERROR	(0x07)	Invalid HTTP pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Delete the HTTP Client instance “my_client.”  */
status =  nx_http_client_delete(&my_client);


/* If status is NX_SUCCESS an HTTP Client instance was successfully
   deleted.  */
```

nx_http_client_get_start

Start an HTTP GET request over IPv4

Prototype

```
UINT nx_http_client_get_start(NX_HTTP_CLIENT *client_ptr,
            ULONG ip_address, CHAR *resource, CHAR *input_ptr,
            UINT input_size, CHAR *username, CHAR *password,
            ULONG wait_option);
```

Description

This service attempts to GET the resource specified by “resource” pointer on the previously created HTTP Client instance. If this routine returns NX_SUCCESS, the application can then make multiple calls to *nx_http_client_get_packet* to retrieve packets of data corresponding to the requested resource content.

Note that the resource string can refer to a local file e.g. “/index.htm” or it can refer to another URL e.g. http://abc.website.com/index.htm if the HTTP Server indicates it supports referring PUT requests.

Input Parameters

client_ptr Pointer to HTTP Client control block.

ip_address IP address of the HTTP Server.

resource Pointer to URL string for requested resource.

input_ptr Pointer to additional data for the GET request. This is optional. If valid, the specified input is placed in the content area of the message and a POST is used instead of a GET operation.

input_size Number of bytes in optional additional input pointed to by `input_ptr`.

username Pointer to optional user name for authentication.

password Pointer to optional password for authentication.

wait_option Defines how long the service will wait for the HTTP Client get start request. The wait options are defined as follows:

time out value	(0x00000001 through 0xFFFFFFFE)
TX_WAIT_FOREVER	(0xFFFFFFFF)

Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until the HTTP Server responds to the request.

Selecting a numeric value (0x1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the HTTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successfully sent HTTP Client GET start message
NX_HTTP_ERROR	(0xE0)	Internal HTTP Client error
NX_HTTP_NOT_READY	(0xEA)	HTTP Client not ready
NX_HTTP_FAILED	(0xE2)	HTTP Client error communicating with the HTTP Server.
NX_HTTP_AUTHENTICATION_ERROR	(0xEB)	Invalid name and/or password.
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Start the GET operation on the HTTP Client “my_client.”  */
status =  nx_http_client_get_start(&my_client, IP_ADDRESS(1,2,3,5), “/TEST.HTM”,
                                   NX_NULL, 0, “myname”, “mypassword”, 1000);

/* If status is NX_SUCCESS, the GET request for TEST.HTM is started and is so
   far successful. The client must now call nx_http_client_get_packet multiple
   times to retrieve the content associated with TEST.HTM. */


#define POST_MESSAGE   “Add this data to the message content”

/* Start the POST operation on the HTTP Client “my_client.”  */
status =  nx_http_client_get_start(&my_client, IP_ADDRESS(1,2,3,5), “/TEST.HTM”,
                                    POST_MESSAGE, strlen(POST_MESSAGE) + 1,
                                    “myname”, “mypassword”, 1000);


/* If status is NX_SUCCESS, the POST_MESSAGE is added to the message in the POST request
   for TEST.HTM and successfully sent. */
```

nxd_http_client_get_start

Send an HTTP GET request (IPv4 or IPv6)

Prototype

```
UINT nxd_http_client_get_start(NX_HTTP_CLIENT *client_ptr,
                    NXD_ADDRESS *server_ip, CHAR *resource,
                    CHAR *input_ptr, UINT input_size, CHAR *username,
                    CHAR *password, ULONG wait_option);
```

Description

This service attempts to create and send a GET request with the resource specified by “resource” pointer on the previously created HTTP Client instance. If this routine returns NX_SUCCESS, the application can then make multiple calls to *nx_http_client_get_packet* to retrieve packets of data corresponding to the requested resource content.

Note that the resource string can refer to a local file e.g. “/index.htm” or it can refer to another URL e.g. http://abc.website.com/index.htm if the HTTP Server indicates it supports referring GET requests.

Input Parameters

client_ptr Pointer to HTTP Client control block.

Server_ip IP address of the HTTP Server.

resource Pointer to URL string for requested resource.

input_ptr Pointer to additional data for the GET request. This is optional. If valid, the specified input is placed in the content area of the message and a POST is used instead of a GET operation.

input_size Number of bytes in optional additional input pointed to by `input_ptr`.

username Pointer to optional user name for authentication.

password Pointer to optional password for authentication.

wait_option Defines how long the service will wait internally to process the HTTP Client get start. 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 the HTTP Server responds to the request.

Selecting a numeric value (0x1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the HTTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successfully sent GET request
NX_HTTP_PASSWORD_TOO_LONG	(0xF0)	Password exceeds buffer size
NX_HTTP_NOT_READY	(0xEA)	HTTP Client not ready
NX_HTTP_FAILED	(0xE2)	Invalid packet parameters.
NX_HTTP_AUTHENTICATION_ERROR	(0xEB)	Invalid name or password
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
NXD_ADDRESS server_ip_address;

/* for an IPv4 address, define as follows: */
server_ip_address.nxd_ip_version = NX_IP_VERSION_V4;
server_ip_address.nxd_ip_address.v4 = IP_ADDRESS(1,2,3,4);

/* for an IPv6 address, define as follows: */
server_ip_address.nxd_ip_version = NX_IP_VERSION_V6;
server_ip_address.nxd_ip_address.v6[0] = 0x20010db8;
server_ip_address.nxd_ip_address.v6[1] = 0x0;
server_ip_address.nxd_ip_address.v6[2] = 0xf101;
server_ip_address.nxd_ip_address.v6[3] = 0x106;


/* Start the GET operation on the HTTP Client “my_client.”  */
status =  nxd_http_client_get_start(&my_client, server_ip_address, “/TEST.HTM”,
                                     NX_NULL, 0, “myname”, “mypassword”, 1000);

/* If status is NX_SUCCESS, the GET request for TEST.HTM is started and is so
   far successful. The client must now call nx_http_client_get_packet multiple
   times to retrieve the content associated with TEST.HTM. */
```

nx_http_client_get_packet

Get next resource data packet

Prototype

```
UINT nx_http_client_get_packet(NX_HTTP_CLIENT *client_ptr,
                               NX_PACKET **packet_ptr, ULONG
                               wait_option);
```

Description

This service retrieves the next packet of content of the resource requested by the previous *nx_http_client_get_start* call. Successive calls to this routine should be made until the return status of NX_HTTP_GET_DONE is received.

Input Parameters

client_ptr Pointer to HTTP Client control block.

packet_ptr Destination for packet pointer containing partial resource content.

wait_option Defines how long the service will wait for the HTTP Client get packet. 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 the HTTP Server responds to the request.

Selecting a numeric value (0x1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the HTTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successful HTTP Client get packet.

NX_HTTP_GET_DONE	(0xEC)	HTTP Client get packet is done
NX_HTTP_NOT_READY	(0xEA)	HTTP Client not in get mode.
NX_HTTP_BAD_PACKET_LENGTH	(0xED)	Invalid packet length
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Get the next packet of resource content on the HTTP Client “my_client.”
   Note that the nx_http_client_get_start routine must have been called
   previously. */
status =  nx_http_client_get_packet(&my_client, &next_packet, 1000);


/* If status is NX_SUCCESS, the next packet of content is pointed to
   by “next_packet”. */
```

nx_http_client_put_start

Start an HTTP PUT request over IPv4

Prototype

```
UINT nx_http_client_put_start(NX_HTTP_CLIENT *client_ptr,
                               ULONG ip_address, CHAR *resource,
                               CHAR *username, CHAR *password,
                               ULONG total_bytes, ULONG wait_option);
```

Description

This service attempts to send a PUT request with the specified resource to the HTTP Server at the supplied IP address. If this routine is successful, the application code should make successive calls to the *nx_http_client_put_packet* routine to actually send the resource contents to the HTTP Server.

Note that the resource string can refer to a local file e.g. “/index.htm” or it can refer to another URL e.g. http://abc.website.com/index.htm if the HTTP Server indicates it supports referring PUT requests.

Input Parameters

client_ptr Pointer to HTTP Client control block.

ip_address IP address of the HTTP Server.

resource Pointer to URL string for resource to send to Server.

username Pointer to optional user name for authentication.

password Pointer to optional password for authentication.

total_bytes Total bytes of resource being sent. Note that the combined length of all packets sent via subsequent calls to *nx_http_client_put_packet* must equal this value.

wait_option Defines how long the service will wait for the HTTP Client PUT start. 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 the HTTP Server responds to the request.

Selecting a numeric value (0x1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the HTTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successfully sent PUT request
NX_HTTP_USERNAME_TOO_LONG	(0xF1)	Username too large for buffer
NX_HTTP_NOT_READY	(0xEA)	HTTP Client not ready
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_SIZE_ERROR	(0x09)	Invalid total size of resource
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Start an HTTP PUT to place the 20-byte resource “/TEST.HTM” on the HTTP Server
   at IP address 1.2.3.5.  */
status =  nx_http_client_put_start(&my_client, IP_ADDRESS(1, 2, 3, 5),
               “/TEST.HTM”, “myname”, “mypassword”, 20, NX_WAIT_FOREVER);

/* If status is NX_SUCCESS, the PUT operation for TEST.HTM has successfully been
   started. */
```

nxd_http_client_put_start

Start an HTTP PUT request (IPv4 or IPv6)

Prototype

```
UINT nxd_http_client_put_start(NX_HTTP_CLIENT *client_ptr,
                               NXD_ADDRESS *server_ip, CHAR *resource,
                               CHAR *username, CHAR *password,
                               ULONG total_bytes, ULONG wait_option);
```

Description

This service attempts to PUT (send) the specified resource on the HTTP Server at the supplied IP address over IPv6. If this routine is successful, the application code should make successive calls to the *nx_http_client_put_packet* routine to actually send the resource contents to the HTTP Server.

Note that the resource string can refer to a local file e.g. "/index.htm" or it can refer to another URL e.g. http://abc.website.com/index.htm if the HTTP Server indicates it supports referring PUT requests.

Input Parameters

client_ptr	Pointer to HTTP Client control block.
server_ip	IP address of the HTTP Server.
resource	Pointer to URL string for resource to send to Server.
username	Pointer to optional user name for authentication.
password	Pointer to optional password for authentication.
total_bytes	Total bytes of resource being sent. Note that the combined length of all packets sent via subsequent calls to *nx_http_client_put_packet* must equal this value.
wait_option	Defines how long the service will wait for the HTTP Client PUT start. 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 the HTTP Server responds to the request.

Selecting a numeric value (0x1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the HTTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successfully sent HTTP Client PUT request
NX_HTTP_ERROR	(0xE0)	HTTP Client internal error
NX_HTTP_NOT_READY	(0xEA)	HTTP Client not ready
NX_HTTP_FAILED	(0xE2)	HTTP Client error communicating with the HTTP Server
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_SIZE_ERROR	(0x09)	Invalid total size of resource
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
NXD_ADDRESS server_ip_address;

/* for an IPv4 address, define as follows: */
server_ip_address.nxd_ip_version = NX_IP_VERSION_V4;
server_ip_address.nxd_ip_address.v4 = IP_ADDRESS(1,2,3,4);

/* for an IPv6 address, define as follows: */
server_ip_address.nxd_ip_version = NX_IP_VERSION_V6;
server_ip_address.nxd_ip_address.v6[0] = 0x20010db8;
server_ip_address.nxd_ip_address.v6[1] = 0x0;
server_ip_address.nxd_ip_address.v6[2] = 0xf101;
server_ip_address.nxd_ip_address.v6[3] = 0x106;

/* Start an HTTP PUT to place the 20-byte resource Client_test.HTM" on the HTTPv6
   Server.  */
status =  nxd_http_client_put_start(&my_client, &server_ip_address,
                      "/client_test.htm", "name", "password", 103, 50);

/* If status is NX_SUCCESS, the PUT operation for Client_test.HTM has successfully
   been started. */
```

nx_http_client_put_packet

Send next resource data packet

Prototype

```
UINT nx_http_client_put_packet(NX_HTTP_CLIENT *client_ptr,
                               NX_PACKET *packet_ptr,
                               ULONG wait_option);
```

Description

This service attempts to send the next packet of resource content to the HTTP Server. Note that this routine should be called repetitively until the combined length of the packets sent equals the “total_bytes” specified in the previous *nx_http_client_put_start* call.

Input Parameters

client_ptr Pointer to HTTP Client control block.

packet_ptr Pointer to next content of the resource to being sent to the HTTP Server.

wait_option Defines how long the service will wait internally to process the HTTP Client PUT packet. 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 the HTTP Server responds to the request.

Selecting a numeric value (0x1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the HTTP Server response.

Return Values

NX_SUCCESS	(0x00)	Successfully sent HTTP Client packet.

NX_HTTP_NOT_READY	(0xEA)	HTTP Client not ready
NX_HTTP_REQUEST_UNSUCCESSFUL_CODE	(0xEE)	Received Server error code
NX_HTTP_BAD_PACKET_LENGTH	(0xED)	Invalid packet length
NX_HTTP_AUTHENTICATION_ERROR	(0xEB)	Invalid name and/or Password
NX_HTTP_INCOMPLETE_PUT_ERROR	(0xEF)	Server responds before PUT Is complete
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_INVALID_PACKET	(0x12)	Packet too small for TCP header
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Send a 20-byte packet representing the content of the resource
   “/TEST.HTM” to the HTTP Server.  */
status =  nx_http_client_put_packet(NX_HTTP_CLIENT *client_ptr, NX_PACKET
*packet_ptr, ULONG wait_option);

/* If status is NX_SUCCESS, the 20-byte resource contents of TEST.HTM has
   successfully been sent. */
```

nx_http_client_set_connect_port

Set the connection port to the Server

Prototype

```
UINT nx_http_client_set_connect_port(NX_HTTP_CLIENT *client_ptr,
                                      UINT port);
```

Description

This service changes the connect port when connecting to the HTTP Server to the specified port at runtime. Otherwise the connect port defaults to 80. This must be called before *nx_http_client_get_start*() and *nx_http_client_put_start*() e.g. when the HTTP Client connects with the Server.

Input Parameters

client_ptr	Pointer to HTTP Client control block.
port	Port for connecting to the Server.

Return Values

NX_SUCCESS	(0x00)	Successfully change port
NX_INVALID_PORT	(0x46)	Port exceeds the maximum (0xFFFF) or is zero.
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Threads, Initialization

Example

```
NX_HTTP_CLIENT *client_ptr;

/* Change the connect port to 114.  */
status =  nx_http_client_set_connect_port(client_ptr, 114);

/* If status is NX_SUCCESS, the connect port is successfully changed. */
```

nx_http_server_cache_info_callback_set

Set the callback to retrieve URL max age and date

Prototype

```
UINT nx_http_server_cache_info_callback_set(NX_HTTP_SERVER *server_ptr,
                         UINT (*cache_info_get)(CHAR *resource,
                                                UINT *max_age,
                                                NX_HTTP_SERVER_DATE
                                                     *date));
```

Description

This service sets the callback service invoked to obtain the maximum age and last modified date of the specified resource.

Input Parameters

server_ptr	Pointer to HTTP Server control block.
cache_info_get	Pointer to the callback
max_age	Pointer to maximum age of a resource
data	Pointer to last modified date returned.

Return Values

NX_SUCCESS	(0x00)	Successfully set the callback
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Initialization

Example

```
NX_HTTP_SERVER my_server;

UINT cache_info_get(CHAR *resource, UINT *max_age,
                           NX_HTTP_SERVER_DATE *last_modified);

/* After my_server is created with nx_http_server_create and before the HTTP
   server is set by nx_http_server_start(), set the cache info callback: */

status = nx_http_server_cache_info_callback_set(&my_server, cache_info_get);


/* If status is NX_SUCCESS, the callback was successfully sent. */
```

nx_http_server_callback_data_send

Send data from callback function

Prototype

```
UINT nx_http_server_callback_data_send(NX_HTTP_SERVER *server_ptr,
                                       VOID *data_ptr,
                                       ULONG data_length);
```

Description

This service sends the data in the supplied packet from the application's callback routine. This is typically used to send dynamic data associated with GET/POST requests. Note that if this function is used, the callback routine is responsible for sending the entire response in the proper format. In addition, the callback routine must return the status of NX_HTTP_CALLBACK_COMPLETED.

Input Parameters

server_ptr	Pointer to HTTP Server control block.
data_ptr	Pointer to the data to send.
data_length	Number of bytes to send.

Return Values

NX_SUCCESS	(0x00)	Successfully sent Server data
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Threads

Example

```
UINT  my_request_notify(NX_HTTP_SERVER *server_ptr, UINT request_type,
                        CHAR *resource, NX_PACKET *packet_ptr)
{

    /* Look for the test resource!  */
    if ((request_type == NX_HTTP_SERVER_GET_REQUEST) &&
        (strcmp(resource, "/test.htm") == 0))
    {

        /* Found it, override the GET processing by sending the resource
           contents directly.  */
        nx_http_server_callback_data_send(server_ptr,
                                          "HTTP/1.0 200 \r\nContent-Length:
                                          103\r\nContent-Type: text/html\r\n\r\n",
                                          63);

        nx_http_server_callback_data_send(server_ptr, "<HTML>\r\n<HEAD><TITLE>NetX
                                          HTTP Test </TITLE></HEAD>\r\n
                                          <BODY>\r\n<H1>NetX Test Page
                                          </H1>\r\n</BODY>\r\n</HTML>\r\n", 103);

        /* Return completion status.  */
        return(NX_HTTP_CALLBACK_COMPLETED);
    }

    return(NX_SUCCESS);
}
```

nx_http_server_callback_generate_response_header

Create a response header in a callback function

Prototype

```
UINT nx_http_server_callback_generate_response_header(
                        NX_HTTP_SERVER *server_ptr,
                        NX_PACKET **packet_pptr,
                        CHAR *status_code, UINT content_length,
                        CHAR *content_type, CHAR* additional_header);
```

Description

This service calls the internal function *_nx_http_server_generate_response_header* when the HTTP server responds to Client get, put and delete requests. It is intended for use in HTTP server callback functions when the HTTP server application is designing its response to the Client.

Input Parameters

server_ptr	Pointer to HTTP Server control block.
packet_pptr	Pointer a packet pointer allocated for message
status_code	Indicate status of resource. Examples: **NX_HTTP_STATUS_OK** **NX_HTTP_STATUS_MODIFIED** **NX_HTTP_STATUS_INTERNAL_ERROR**
content_length	Size of content in bytes
content_type	Type of HTTP e.g. “text/plain”
additional_header	Pointer to additional header text

Return Values

NX_SUCCESS	(0x00)	Successfully created header
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Threads

Example

```
  CHAR demotestbuffer[] = “<html>\r\n\r\n<head>\r\n\r\n<title>Main   \
           Window</title>\r\n</head>\r\n\r\n<body>Test message\r\n   \
           </body>\r\n</html>\r\n";

/* my_request_notify is the application request notify callback registered with
  the HTTP server in nx_http_server_create, creates a response to the received
  Client request. */

  UINT  my_request_notify(NX_HTTP_SERVER *server_ptr, UINT request_type,
                           CHAR *resource, NX_PACKET *recv_packet_ptr)
  {

    NX_PACKET   *sresp_packet_ptr;
    ULONG       string_length;
    CHAR        temp_string[30];
    ULONG       length = 0;


       length = strlen(&demotestbuffer[0]);

    /* Derive the client request type from the client request.  */
       string_length =  (ULONG) nx_http_server_type_get(server_ptr, server_ptr ->
                                  nx_http_server_request_resource, temp_string);

   /* Null terminate the string. */
       temp_string[temp] = 0;

   /* Now build a response header with server status is OK and no additional header
      info.  */
       status = nx_http_server_callback_generate_response_header(http_server_ptr,
                                     &resp_packet_ptr, NX_HTTP_STATUS_OK,
                                     length, temp_string, NX_NULL);

    /* If status is NX_SUCCESS, the header was successfully appended. */

    /* Now add data to the packet.  */
       status = nx_packet_data_append(resp_packet_ptr, &demotestbuffer[0],
                     strlen(&demotestbuffer[0]),  server_ptr ->
                     nx_http_server_packet_pool_ptr, NX_WAIT_FOREVER);
       if (status != NX_SUCCESS)
       {
           nx_packet_release(resp_packet_ptr);
           return status;
       }



    /* Now send the packet! */
        status = nx_tcp_socket_send(&(server_ptr -> nx_http_server_socket),
                                   resp_packet_ptr, NX_HTTP_SERVER_TIMEOUT_SEND);

       if (status != NX_SUCCESS)
       {
          nx_packet_release(resp_packet_ptr);

          return status;
       }

    /* Let HTTP server know the response has been sent. */
      return NX_HTTP_CALLBACK_COMPLETED;

 }
```

nx_http_server_callback_packet_send

Send an HTTP packet from callback function

Prototype

```
UINT nx_http_server_callback_packet_send(NX_HTTP_SERVER *server_ptr,
                                         NX_PACKET *packet_ptr);
```

Description

This service sends a complete HTTP server response from an HTTP callback. HTTP server will send the packet with the NX_HTTP_SERVER _TIMEOUT_SEND. The HTTP header and data must be appended to the packet. If the return status indicates an error, the HTTP application must release the packet.

The callback should return NX_HTTP_CALLBACK_COMPLETED.

See *nx_http_server_callback_generate_response_header* for a more detailed example.

Input Parameters

server_ptr	Pointer to HTTP Server control block
packet_ptr	Pointer to the packet to send

Return Values

NX_SUCCESS	(0x00)	Successfully sent Server packet
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Threads

Example

```
/* The packet is appended with HTTP header and data and is ready to send to the
   Client directly. */

   status = nx_http_server_callback_response_send(server_ptr, packet_ptr);

   if (status != NX_SUCCESS)
   {

       nx_packet_release(packet_ptr);
}
 return(NX_HTTP_CALLBACK_COMPLETED);
```

nx_http_server_callback_response_send

Send response from callback function

Prototype

```
UINT nx_http_server_callback_response_send(NX_HTTP_SERVER *server_ptr,
                                            CHAR *header,
                                            CHAR *information,
                                            CHAR additional_info);
```

Description

This service sends the supplied response information from the application’s callback routine. This is typically used to send custom responses associated with GET/POST requests. Note that if this function is used, the callback routine must return the status of NX_HTTP_CALLBACK_COMPLETED.

Input Parameters

server_ptr	Pointer to HTTP Server control block.
header	Pointer to the response header string.
information	Pointer to the information string.
additional_info	Pointer to the additional information string.

Return Values

NX_SUCCESS	(0x00)	Successfully sent Server response

Allowed From

Threads

Example

```
UINT  my_request_notify(NX_HTTP_SERVER *server_ptr, UINT request_type,
                      CHAR *resource, NX_PACKET *packet_ptr)
{

    /* Look for the test resource!  */
    if ((request_type == NX_HTTP_SERVER_GET_REQUEST) &&
       (strcmp(resource, "/test.htm") == 0))
    {

        /* In this example, we will complete the GET processing with
           a resource not found response.  */
         nx_http_server_callback_response_send(server_ptr,
                                             "HTTP/1.0 404 ",
                                             "NetX HTTP Server unable to find
                                              file: ", resource);

        /* Return completion status.  */
        return(NX_HTTP_CALLBACK_COMPLETED);
    }

    return(NX_SUCCESS);
}
```

nx_http_server_content_get

Get content from the request

Prototype

```
UINT nx_http_server_content_get(NX_HTTP_SERVER *server_ptr,
                                NX_PACKET *packet_ptr,
                                ULONG byte_offset,
                                CHAR *destination_ptr,
                                UINT destination_size,
                                UINT *actual_size);
```

Description

This service attempts to retrieve the specified amount of content from the POST or PUT HTTP Client request. It should be called from the application’s request notify callback specified during HTTP Server creation (*nx_http_server_create*).

Input Parameters

server_ptr Pointer to HTTP Server control block.

packet_ptr Pointer to the HTTP Client request packet. Note that this packet must not be released by the request notify callback.

byte_offset Number of bytes to offset into the content area.

destination_ptr Pointer to the destination area for the content.

destination_size Maximum number of bytes available in the destination area.

actual_size Pointer to the destination variable that will be set to the actual size of the content copied.

Return Values

NX_SUCCESS	(0x00)	Successful HTTP Server content get
NX_HTTP_ERROR	(0xE0)	HTTP Server internal error
NX_HTTP_DATA_END	(0xE7)	End of request content
NX_HTTP_TIMEOUT	(0xE1)	HTTP Server timeout in getting next packet of content

NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Assuming we are in the application’s request notify callback
   routine, retrieve up to 100 bytes of content starting at offset
   0.  */
status =  nx_http_server_content_get(&my_server, packet_ptr,
                                     0, my_buffer, 100, &actual_size);

/* If status is NX_SUCCESS, “my_buffer” contains “actual_size” bytes of
   request content. */
```

nx_http_server_content_get_extended

Get content from the request/supports zero length Content Length

Prototype

```
UINT nx_http_server_content_get_extended(NX_HTTP_SERVER *server_ptr,
                                         NX_PACKET *packet_ptr,
                                         ULONG byte_offset,
                                         CHAR *destination_ptr,
                                         UINT destination_size,
                                         UINT *actual_size);
```

Description

This service is almost identical to *nx_http_server_content_get*; it attempts to retrieve the specified amount of content from the POST or PUT HTTP Client request. However it handles requests with Content Length of zero value (‘empty request’) as a valid request. It should be called from the application’s request notify callback specified during HTTP Server creation (*nx_http_server_create*).

Input Parameters

server_ptr	Pointer to HTTP Server control block.
packet_ptr	Pointer to the HTTP Client request packet. Note that this packet must not be released by the request notify callback.
byte_offset	Number of bytes to offset into the content area.
destination_ptr	Pointer to the destination area for the content.
destination_size	Maximum number of bytes available in the destination area.
actual_size	Pointer to the destination variable that will be set to the actual size of the content copied.

Return Values

NX_SUCCESS	(0x00)	Successful HTTP content get
NX_HTTP_ERROR	(0xE0)	HTTP Server internal error
NX_HTTP_DATA_END	(0xE7)	End of request content
NX_HTTP_TIMEOUT	(0xE1)	HTTP Server timeout in getting next packet

NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Assuming we are in the application’s request notify callback
   routine, retrieve up to 100 bytes of content starting at offset
   0.  */
status =  nx_http_server_content_get_extended(&my_server, packet_ptr,
                                              0, my_buffer, 100, &actual_size);

/* If status is NX_SUCCESS, “my_buffer” contains “actual_size” bytes of
   request content. */
```

nx_http_server_content_length_get

Get length of content in the request

Prototype

```
UINT nx_http_server_content_length_get(NX_PACKET *packet_ptr);
```

Description

This service attempts to retrieve the HTTP content length in the supplied packet. If there is no HTTP content, this routine returns a value of zero. It should be called from the application’s request notify callback specified during HTTP Server creation (*nx_http_server_create*).

Input Parameters

packet_ptr Pointer to the HTTP Client request packet. Note that this packet must not be released by the request notify callback.

Return Values

content length
On error, a value of zero is returned

Allowed From

Threads

Example

```
/* Assuming we are in the application’s request notify callback
   routine, get the content length of the HTTP Client request.  */
length =  nx_http_server_content_length_get(packet_ptr);

/* The “length” variable now contains the length of the HTTP Client
   request content area. */
```

nx_http_server_content_length_get_extended

Get length of content in the request/supports Content Length of zero value

Prototype

```
UINT nx_http_server_content_length_get_extended(NX_PACKET *packet_ptr,
                                                UINT *content_length);
```

Description

This service is similar to *nx_http_server_content_length_get*; attempts to retrieve the HTTP content length in the supplied packet. However, the return value indicates successful completion status, and the actual length value is returned in the input pointer `content_length`. If there is no HTTP content/Content Length = 0, this routine still returns a successful completion status and the `content_length` input pointer points to a valid length (zero). It should be called from the application's request notify callback specified during HTTP Server creation (*nx_http_server_create*).

Input Parameters

packet_ptr	Pointer to the HTTP Client request packet. Note that this packet must not be released by the request notify callback.
content_length	Pointer to value retrieved from Content Length field

Return Values

NX_SUCCESS	(0x00)	Successful Server content get
NX_HTTP_INCOMPLETE_PUT_ERROR	(0xEF)	Improper HTTP header format
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Threads

Example

```
/* Assuming we are in the application's request notify callback
   routine, get the content length of the HTTP Client request.  */
ULONG content_length;

status =  nx_http_server_content_length_get_extended(packet_ptr, &content_length);

/* If the "status" variable indicates successful completion, the"length" variable
   contains the length of the HTTP Client request content area. */
```

nx_http_server_create

Create an HTTP Server instance

Prototype

```
UINT nx_http_server_create(NX_HTTP_SERVER *http_server_ptr,
      CHAR *http_server_name, NX_IP *ip_ptr, FX_MEDIA *media_ptr,
      VOID *stack_ptr, ULONG stack_size, NX_PACKET_POOL *pool_ptr,
      UINT (*authentication_check)(NX_HTTP_SERVER *server_ptr,
            UINT request_type, CHAR *resource, CHAR **name,
            CHAR **password, CHAR **realm),
      UINT (*request_notify)(NX_HTTP_SERVER *server_ptr,
            UINT request_type, CHAR *resource, NX_PACKET *packet_ptr));
```

Description

This service creates an HTTP Server instance, which runs in the context of its own ThreadX thread. The optional *authentication_check* and *request_notify* application callback routines give the application software control over the basic operations of the HTTP Server.

Input Parameters

http_server_ptr Pointer to HTTP Server control block.

http_server_name Pointer to HTTP Server's name.

ip_ptr Pointer to previously created IP instance.

media_ptr Pointer to previously created FileX media instance.

stack_ptr Pointer to HTTP Server thread stack area.

stack_size Pointer to HTTP Server thread stack size.

authentication_check Function pointer to application's authentication checking routine. If specified, this routine is called for each HTTP Client request. If this parameter is NULL, no authentication will be performed.

request_notify Function pointer to application's request notify routine. If specified, this routine is called prior to the HTTP server processing of the request. This allows the resource name to be redirected or fields within a resource to be updated prior to completing the HTTP Client request.

Return Values

NX_SUCCESS	(0x00)	Successful HTTP Server create.
NX_PTR_ERROR	(0x07)	Invalid HTTP Server, IP, media, stack, or packet pool pointer.
NX_HTTP_POOL_ERROR	(0xE9)	Packet payload of pool is not large enough to contain complete HTTP request.

Allowed From

Initialization, Threads

Example

```
/* Create an HTTP Server instance called “my_server.”  */
status =  nx_http_server_create(&my_server, “my server”, &ip_0, &ram_disk,
               stack_ptr, stack_size, &pool_0,
               my_authentication_check, my_request_notify);

/* If status equals NX_SUCCESS, the HTTP Server creation was successful. */
```

nx_http_server_delete

Delete an HTTP Server instance

Prototype

```
UINT nx_http_server_delete(NX_HTTP_SERVER *http_server_ptr);
```

Description

This service deletes a previously created HTTP Server instance.

Input Parameters

http_server_ptr Pointer to HTTP Server control block.

Return Values

NX_SUCCESS	(0x00)	Successful HTTP Server delete
NX_PTR_ERROR	(0x07)	Invalid HTTP Server pointer
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Delete the HTTP Server instance called “my_server.”  */
status =  nx_http_server_delete(&my_server);

/* If status equals NX_SUCCESS, the HTTP Server delete was successful. */
```

nx_http_server_get_entity_content

Retrieve the location and length of entity data

Prototype

```
UINT nx_http_server_get_entity_content(NX_HTTP_SERVER *server_ptr,
                                       NX_PACKET **packet_pptr,
                                       ULONG *available_offset,
                                       ULONG *available_length);
```

Description

This service determines the location of the start of data within the current multipart entity in the received Client messages, and the length of data not including the boundary string. Internally HTTP server updates its own offsets so that this function can be called again on the same Client datagram for messages with multiple entities. The packet pointer is updated to the next packet where the Client message is a multi-packet datagram.

Note that NX_HTTP_MULTIPART_ENABLE must be enabled to use this service.

See *nx_http_server_get_entity_header* for more details.

Input Parameters

server_ptr	Pointer to HTTP Server
packet_pptr	Pointer to location of packet pointer. Note that the application should not release this packet.
available_offset	Pointer to offset of entity data from the packet prepend pointer
available_length	Pointer to length of entity data

Return Values

NX_SUCCESS	(0x00)	Successfully retrieved size and location of entity content
NX_HTTP_BOUNDARY_ALREADY_FOUND		

	(0xF4)	Content for the HTTP server internal multipart markers is already found
NX_HTTP_ERROR	(0xE0)	Internal HTTP error
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Threads

Example

```
NX_HTTP_SERVER my_server;

UINT        offset, length;
NX_PACKET  *packet_ptr;

/* Inside the request notify callback, the HTTP server application first obtains
   the entity header to determine details about the multipart data.  If
   successful, it then calls this service to get the location of entity data:  */

status =  nx_http_server_get_entity_content(&my_server, &packet_ptr, *offset,
                                            &length);

/* If status equals NX_SUCCESS, offset and location determine the location of the
   entity data.  */
```

nx_http_server_get_entity_header

Retrieve the contents of entity header

Prototype

```
UINT nx_http_server_get_entity_header(NX_HTTP_SERVER *server_ptr,
                                       NX_PACKET **packet_pptr,
                                       UCHAR *entity_header_buffer,
                                       ULONG buffer_size);
```

Description

This service retrieves the entity header into the specified buffer. Internally HTTP Server updates its own pointers to locate the next multipart entity in a Client datagram with multiple entity headers. The packet pointer is updated to the next packet where the Client message is a multi-packet datagram.

Note that NX_HTTP_MULTIPART_ENABLE must be enabled to use this service.

Input Parameters

server_ptr	Pointer to HTTP Server
packet_pptr	Pointer to location of packet pointer. Note that the application should not release this packet.
entity_header_buffer	Pointer to location to store entity header
buffer_size	Size of input buffer

Return Values

NX_SUCCESS	(0x00)	Successfully retrieved entity heade
NX_HTTP_NOT_FOUND	**(0xE6)**	Entity header field not found
NX_HTTP_TIMEOUT	**(0xE1)**	Time expired to receive next packet for multipacket client message
NX_HTTP_ERROR	(0xE0)	Internal HTTP error
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* my_request_notify is the application request notify callback registered with
   the HTTP server in nx_http_server_create, creates a response to the received
   Client request. */

   UINT  my_request_notify(NX_HTTP_SERVER *server_ptr, UINT request_type,
                           CHAR *resource, NX_PACKET *packet_ptr)
   {

    NX_PACKET   *sresp_packet_ptr;
    UINT        offset, length;
    NX_PACKET   *response_pkt;
    UCHAR       buffer[1440];

    /* Process multipart data. */
    if(request_type == NX_HTTP_SERVER_POST_REQUEST)
    {

       /* Get the content header. */
       while(nx_http_server_get_entity_header(server_ptr, &packet_ptr, buffer,
                                              sizeof(buffer)) == NX_SUCCESS)
       {

          /* Header obtained successfully. Get the content data location. */
          while(nx_http_server_get_entity_content(server_ptr, &packet_ptr, &offset,
                                                  &length) == NX_SUCCESS)
          {
               /* Write content data to buffer. */
               nx_packet_data_extract_offset(packet_ptr, offset, buffer, length,
                                             &length);
               buffer[length] = 0;
          }

        }

        /* Generate HTTP header. */
        status = nx_http_server_callback_generate_response_header(server_ptr,
                             &response_pkt, NX_HTTP_STATUS_OK, 800, "text/html",
                             "Server: NetXDuo HTTP 5.3\r\n");

        if(status == NX_SUCCESS)
        {
            if(nx_http_server_callback_packet_send(server_ptr, response_pkt) !=
                                                   NX_SUCCESS)
             {
                    nx_packet_release(response_pkt);
             }
        }


    }
    else
    {
           /* Indicate we have not processed the response to client yet.*/
           return(NX_SUCCESS);
    }

    /* Indicate the response to client is transmitted. */
    return(NX_HTTP_CALLBACK_COMPLETED);
```

nx_http_server_gmt_callback_set

Set the callback to obtain GMT date and time

Prototype

```
UINT nx_http_server_gmt_callback_set(NX_HTTP_SERVER *server_ptr,
                          VOID (*gmt_get)(NX_HTTP_SERVER_DATE *date);
```

Description

This service sets the callback to obtain GMT date and time with a previously created HTTP server. This service is invoked with the HTTP server is creating a header in HTTP server responses to the Client.

Input Parameters

server_ptr	Pointer to HTTP Server
gmt_get	Pointer to GMT callback
date	Pointer to the date retrieved

Return Values

NX_SUCCESS	(0x00)	Successfully set the callback
NX_PTR_ERROR	(0x07)	Invalid packet or parameter pointer.

Allowed From

Threads

Example

```
NX_HTTP_SERVER my_server;

VOID get_gmt(NX_HTTP_SERVER_DATE *now);

/* After the HTTP server is created by calling nx_http_server_create, and before
   starting HTTP services when nx_http_server_start is called, set the GMT
   retrieve callback:  */

status =  nx_http_server_gmt_callback_set(&my_server, gmt_get);

/* If status equals NX_SUCCESS, the gmt_get will be called to set the HTTP server
   response header date.  */
```

nx_http_server_invalid_userpassword_notify_set

Set the callback to to handle invalid user/password

Prototype

```
UINT nx_http_server_invalid_userpassword_notify_set(
            NX_HTTP_SERVER *http_server_ptr,
            UINT (*invalid_username_password_callback)
                        (CHAR *resource,
                        NXD_ADDRESS *client_address,
                        UINT request_type));
```

Description

This service sets the callback invoked when an invalid username and password is received in a Client get, put or delete request, either by digest or basic authentication. The HTTP server must be previously created.

Input Parameters

server_ptr Pointer to HTTP Server

invalid_username_password_callback
Pointer to invalid user/pass callback

resource Pointer to the resource specified by the client

client_address Pointer to client address. Can be IPv4 or IPv6

request_type Indicates client request type. May be:

NX_HTTP_SERVER_GET_REQUEST
NX_HTTP_SERVER_POST_REQUEST
NX_HTTP_SERVER_HEAD_REQUEST
NX_HTTP_SERVER_PUT_REQUEST
NX_HTTP_SERVER_DELETE_REQUEST

Return Values

NX_SUCCESS	(0x00)	Successfully set the callback
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Threads

Example

```
NX_HTTP_SERVER my_server;
VOID invalid_username_password_callback (NX_ CHAR *resource,
                                        NXD_ADDRESS *client_address,
                                        UINT   request_type);

/* After the HTTP server is created by calling nx_http_server_create, and before
   starting HTTP services when nx_http_server_start is called, set the invalid
   username password callback:  */

status =  nx_http_server_gmt_callback_set(&my_server,
                                         invalid_username_password_callback);

/* If status equals NX_SUCCESS, the invalid_username_password_callback function
   will be called when the HTTP server receives an invalid username/password.  */
```

nx_http_server_mime_maps_additional_set

Set additional MIME maps for HTML

Prototype

```
UINT nx_http_server_mime_maps_additional_set(
                        NX_HTTP_SERVER *server_ptr,
                        NX_HTTP_SERVER_MIME_MAP *mime_maps,
                        UINT mime_maps_num);
```

Description

This service allows the HTTP application developer to add additional MIME types from the default MIME types supplied by NetX Duo HTTP Server (see *nx_http_server_get_type* for list of defined types).

When a client request is received, e.g. a GET request, HTTP server parses the requested file type from the HTTP header using preferentially the additional MIME map set and if no match if found, it looks for a match in the default MIME map of the HTTP server. If no match is found, the MIME type defaults to “text/plain”.

If the request notify function is registered with the HTTP server, the request notify callback can call *nx_http_server_type_get* to parse the file type.

Input Parameters

server_ptr	Pointer to HTTP Server instance
mime_maps	Pointer to a MIME map array
mime_map_num	Number of MIME maps in array

Return Values

NX_SUCCESS	(0x00)	Successful HTTP Server MIME map set
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Initialization, Threads

Example

```
/* my_server is an NX_HTTP_SERVER previously created.  */

NX_HTTP_SERVER_MIME_MAP my_mime_maps [2];

static NX_HTTP_SERVER_MIME_MAP my_mime_maps[] =
{
    {"abc",      "yourtype/abc"},
    {"xyz",      "mytype/xyz"},
};

status =  nx_http_server_mime_maps_additional_set(&my_server,
                                                  &my_mime_maps[0], 2);

/* If status equals NX_SUCCESS, two additional MIME types are added to the HTTP
  server MIME map set." */
```

nx_http_server_packet_content_find

Extract content length and set pointer to start of data

Prototype

```
UINT nx_http_server_packet_content_find(NX_HTTP_SERVER *server_ptr,
                                         NX_PACKET **packet_ptr,
                                         UINT *content_length);
```

Description

This service extracts the content length from the HTTP header. It also updates the supplied packet as follows: the packet prepend pointer (start of location of packet buffer to write to) is set to the HTTP content (data) just passed the HTTP header.

If the beginning of content is not found in the current packet, the function waits for the next packet to be received using the NX_HTTP_SERVER_TIMEOUT_RECEIVE wait option.

Note this should not be called before calling *nx_http_server_get_entity_header* because it modifies the prepend pointer past the entity header.

Input Parameters

server_ptr	Pointer to HTTP server instance
packet_ptr	Pointer to packet pointer for returning the packet with updated prepend pointer
content_length	Pointer to extracted `content_length`

Return Values

NX_SUCCESS	(0x00)	HTTP content length found and packet successfully updated
NX_HTTP_TIMEOUT	(0xE1)	Time expired waiting on next packet
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Threads

Example

```
/* The HTTP server pointed to by server_ptr is previously created and started.
The server has received a Client request packet, recv_packet_ptr, and the packet
content find service is called from the request notify callback function
registered with the HTTP server.  */

UINT content_length;

status =  nx_http_server_packet_content_find(server_ptr, recv_packet_ptr,
                                              &content_length);

/* If status equals NX_SUCCESS, the content length specifies the content length
   and the packet pointer prepend pointer is set to the HTTP content (data). */
```

nx_http_server_packet_get

Receive the next HTTP packet

Prototype

```
UINT nx_http_server_packet_get(NX_HTTP_SERVER *server_ptr,
                               NX_PACKET **packet_ptr);
```

Description

This service returns the next packet received on the HTTP server socket. The wait option to receive a packet is NX_HTTP_SERVER_TIMEOUT_RECEIVE.

Input Parameters

server_ptr	Pointer to HTTP server instance
packet_ptr	Pointer to received packet

Return Values

NX_SUCCESS	(0x00)	Successfully received next packet
NX_HTTP_TIMEOUT	(0xE1)	Time expired waiting on next packet
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Threads

Example

```
/* The HTTP server pointed to by server_ptr is previously created and started. */

UINT content_length;
NX_PACKET *recv_packet_ptr;

status =  nx_http_server_packet_get(server_ptr, &recv_packet_ptr);

/* If status equals NX_SUCCESS, a Client packet is obtained. */
```

nx_http_server_param_get

Get parameter from the request

Prototype

```
UINT nx_http_server_param_get(NX_PACKET *packet_ptr,
                              UINT param_number, CHAR *param_ptr,
                              UINT max_param_size);
```

Description

This service attempts to retrieve the specified HTTP URL parameter in the supplied request packet. If the requested HTTP parameter is not present, this routine returns a status of NX_HTTP_NOT_FOUND. This routine should be called from the application's request notify callback specified during HTTP Server creation (*nx_http_server_create*).

Input Parameters

packet_ptr	Pointer to HTTP Client request packet. Note that the application should not release this packet.
param_number	Logical number of the parameter starting at zero, from left to right in the parameter list.
param_ptr	Destination area to copy the parameter.
max_param_size	Maximum size of the parameter destination area.

Return Values

NX_SUCCESS	(0x00)	Successful HTTP Server parameter get
NX_HTTP_NOT_FOUND	(0xE6)	Specified parameter not found
NX_HTTP_IMPROPERLY_TERMINATED_PARAM	(0xF3)	Request parameter not properly terminated
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Assuming we are in the application’s request notify callback
   routine, get the first parameter of the HTTP Client request.  */

status =  nx_http_server_param_get(request_packet_ptr, 0, param_destination,
               30);

/* If status equals NX_SUCCESS, the NULL-terminated first parameter can be found
   in “param_destination.” */
```

nx_http_server_query_get

Get query from the request

Prototype

```
UINT nx_http_server_query_get(NX_PACKET *packet_ptr, UINT query_number,
                  CHAR *query_ptr, UINT max_query_size);
```

Description

This service attempts to retrieve the specified HTTP URL query in the supplied request packet. If the requested HTTP query is not present, this routine returns a status of NX_HTTP_NOT_FOUND. This routine should be called from the application’s request notify callback specified during HTTP Server creation (*nx_http_server_create*).

Input Parameters

packet_ptr	Pointer to HTTP Client request packet. Note that the application should not release this packet.
query_number	Logical number of the parameter starting at zero, from left to right in the query list.
query_ptr	Destination area to copy the query.
max_query_size	Maximum size of the query destination area.

Return Values

NX_SUCCESS	(0x00)	Successful HTTP Server query get
NX_HTTP_FAILED	(0xE2)	Query size too small.
NX_HTTP_NOT_FOUND	(0xE6)	Specified query not found
NX_HTTP_NO_QUERY_PARSED	(0xF2)	No query in Client request
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Assuming we are in the application’s request notify callback
   routine, get the first query of the HTTP Client request.  */
status =  nx_http_server_query_get(request_packet_ptr, 0, query_destination,
               30);

/* If status equals NX_SUCCESS, the NULL-terminated first query can be found
   in “query_destination.” */
```

nx_http_server_start

Start the HTTP Server

Prototype

```
UINT nx_http_server_start(NX_HTTP_SERVER *http_server_ptr);
```

Description

This service starts the previously create HTTP Server instance.

Input Parameters

http_server_ptr	Pointer to HTTP Server instance.

Return Values

NX_SUCCESS	(0x00)	Successful Server start
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Initialization, Threads

Example

```
/* Start the HTTP Server instance “my_server.”  */
status =  nx_http_server_start(&my_server);

/* If status equals NX_SUCCESS, the HTTP Server has been started. */
```

nx_http_server_stop

Stop the HTTP Server

Prototype

```
UINT nx_http_server_stop(NX_HTTP_SERVER *http_server_ptr);
```

Description

This service stops the previously create HTTP Server instance. This routine should be called prior to deleting an HTTP Server instance.

Input Parameters

http_server_ptr	Pointer to HTTP Server instance.

Return Values

NX_SUCCESS	(0x00)	Successful Server stop
NX_PTR_ERROR	(0x07)	Invalid pointer input
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Example

```
/* Stop the HTTP Server instance “my_server.”  */
status =  nx_http_server_stop(&my_server);

/* If status equals NX_SUCCESS, the HTTP Server has been stopped. */
```

nx_http_server_type_get

Extract file type from Client HTTP request

Prototype

```
UINT nx_http_server_type_get(NX_HTTP_SERVER *http_server_ptr,
                             CHAR *name, CHAR *http_type_string);
```

Description

This service extracts the HTTP request type in the buffer *http_type_string* and its length in the return valud from the input buffer *name*, usually the URL. If no MIME map is found, it defaults to the “text/plain” type. Otherwise it compares the extracted type against the HTTP Server default MIME maps for a match. The default MIME maps in NetX Duo HTTP Server are:

html	text/html
htm	text/html
txt	text/plain
gif	image/gif
jpg	image/jpeg
ico	image/x-icon

If supplied, it will also search a user defined set of additional MIME maps. See *nx_http_server_mime_maps_addtional_set* for more details on user defined maps.

Input Parameters

http_server_ptr Pointer to HTTP Server instance

name Pointer to buffer to search

http_type_string (Pointer to extracted HTML type)

Return Values

Length of string in bytes Non zero value is success
Zero indicates error

Allowed From

Application

Example

```
/* my_server is a previously created HTTP server, which starts accepting client
   requests when nx_http_server_start is called*/

CHAR temp_string[20];
UINT string_length;

/* Extract the HTTP type. */
    string_length =  nx_http_server_type_get(&my_server_ptr,
               my_server.nx_http_server_request_resource, temp_string);

/* If string_length is non zero, the HTTP string is extracted. */
```

For a more detailed example, see the description for *nx_http_server_callback_generate_response_header.*

NetX Duo Hypertext Transfer Protocol (NetX Duo HTTP)
User Guide

Publication Date: Rev.5.14 Feb 25, 2019

Published by: Renesas Electronics Corporation

NetX Duo Hypertext Transfer Protocol (NetX Duo HTTP) User Guide

Renesas Electronics Corporation