



Point-to-Point Protocol (PPP)

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

©2002-2017 by Express Logic, Inc.

All rights reserved. This document and the associated NetX software are the sole property of Express Logic, Inc. Each contains proprietary information of Express Logic, Inc. Reproduction or duplication by any means of any portion of this document without the prior written consent of Express Logic, Inc. is expressly forbidden. 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

NetX, Piconet, and UDP Fast Path are trademarks of Express Logic, Inc. ThreadX is a registered trademark of Express Logic, Inc.

All other product and company names are trademarks or registered trademarks of their respective holders.

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: 000-1053

Revision 5.10

Contents

<i>Chapter 1 Introduction to PPP</i>	4
PPP Serial Communication	4
PPP Packet	5
Link Control Protocol (LCP).....	5
Password Authentication Protocol (PAP).....	6
Challenge-Handshake Authentication Protocol (CHAP)	6
Internet Protocol Control Protocol (IPCP)	6
Data Transfer	7
PPP RFCs.....	7
<i>Chapter 2 Installation and Use of PPP</i>	8
Product Distribution	8
PPP Installation.....	8
Using PPP	8
Using Modems	8
Small Example System.....	11
Configuration Options.....	13
<i>Chapter 3 Description of PPP Services</i>	17
nx_ppp_byte_receive	19
nx_ppp_chap_challenge.....	20
nx_ppp_chap_enable	21
nx_ppp_create	24
nx_ppp_delete.....	26
nx_ppp_dns_address_get	27
nx_ppp_interface_index_get	28
nx_ppp_ip_address_assign.....	29
nx_ppp_link_down_notify	30
nx_ppp_link_up_notify.....	31
nx_ppp_nak_authentication_notify	32
nx_ppp_pap_enable.....	33
nx_ppp_ping_request.....	35
nx_ppp_raw_string_send	37
nx_ppp_restart	38
nx_ppp_start	39
nx_ppp_status_get.....	40
nx_ppp_stop.....	42

Chapter 1

Introduction to PPP

Typically, NetX applications connect to the actual physical network through Ethernet. This provides network access that is both fast and efficient. However, there are situations where the application does not have Ethernet access. In such cases, the application may still connect to the network through a serial interface connected directly to another network member. The most common software protocol used to manage such a connection is the Point-to-Point Protocol (PPP).

Although serial communication is relatively straightforward, the PPP is somewhat complex. The PPP is actually comprised of multiple protocols, such as the Link Control Protocol (LCP), Internet Protocol Control Protocol (IPCP), Password Authentication Protocol (PAP), and the Challenge-Handshake Authentication Protocol (CHAP). The LCP is the main protocol for PPP. This is where the basic components of the link are dynamically negotiated in a peer-to-peer fashion. Once the basic characteristics of the link have been successfully negotiated, the PAP and/or CHAP are used to ensure a connected peer is valid. If both peers are valid, the IPCP is then utilized to negotiate the IP addresses used by the peers. Once IPCP completes, PPP is then able to send and receive IP packets.

NetX views the PPP primarily as a device driver. The *nx_ppp_driver* function is supplied to the NetX IP create function, *nx_ip_create*. Otherwise, NetX does not have any direct knowledge of PPP.

PPP Serial Communication

The NetX PPP package requires the application to provide a serial communication driver. The driver must support 8-bit characters and may also employ software flow control. It is the application's responsibility to initialize the driver, which should be done prior to creating the PPP instance.

In order to send PPP packets, a serial driver output byte routine must be provided to PPP (specified in the *nx_ppp_create* function). This serial driver byte output routine will be called repetitively in order to transmit the entire PPP packet. It is the serial driver's responsibility to buffer the output. On the receive side, the application's serial driver must call the PPP *nx_ppp_byte_receive* function whenever a new byte arrives. This is

typically done from within the context of an Interrupt Service Routine (ISR). The *nx_ppp_byte_receive* function places the incoming byte into a circular buffer and alerts the PPP receive thread of its presence.

PPP Packet

PPP utilizes AHDLC framing (a subset of HDLC) for encapsulating all PPP protocol control and user data. An AHDLC frame looks like the following:

Flag	Addr	Ctrl	Information	CRC	Flag
7E	FF	03	[0-1502 bytes]	2-byte	7E

Each and every PPP frame has this overall appearance. The first two bytes of the information field contain the PPP protocol type. Valid values are defined as follows:

C021	LCP
8021	IPCP
C023	PAP
C223	CHAP
0021	IP Data Packet

If the 0x0021 protocol type is present, the IP packet follows immediately. Otherwise, if one of the other protocols is present, the following bytes correspond to that particular protocol.

In order to ensure unique 0x7E beginning/end-of frame markers and to support software flow control, AHDLC uses escape sequences to represent various byte values. The 0x7D value specifies that the character following is encoded, which is basically the original character exclusive ORed with 0x20. For example, the 0x03 value for the Ctrl field in the header is represented by the two byte sequence: 7D 23. By default, values less than 0x20 are converted into an escape sequence, as well as 0x7E and 0x7D values found in the Information field. Note that escape sequences also apply to the CRC field.

Link Control Protocol (LCP)

The LCP is the primary PPP protocol and is the first protocol to run. LCP is responsible for negotiating various PPP parameters, including the Maximum Receive Unit (MRU) and the Authentication Protocol (PAP, CHAP, or none) to use. Once both sides of LCP agree on PPP parameters, the authentication protocols—if any—then start running.

Password Authentication Protocol (PAP)

The PAP is a relatively straightforward protocol that relies on a name and password being supplied by one side of the connection (as negotiated during LCP). The other side then verifies this information. If correct, an acceptance message is returned to the sender and PPP can then proceed to the IPCP state machine. Otherwise, if either the name or password is incorrect, the connection is rejected.

Note that both sides of the interface can request PAP, but PAP is typically used in only one direction.

Challenge-Handshake Authentication Protocol (CHAP)

The CHAP is a more complex authentication protocol than PAP. The CHAP authenticator supplies its peer with a name and a value. The peer then uses the supplied name to find a shared “secret” between the two entities. A computation is then done over the ID, value, and the “secret.” The result of this computation is returned in the response. If correct, PPP can then proceed to the IPCP state machine. Otherwise, if the result is incorrect, the connection is rejected.

Another interesting aspect of CHAP is that it can occur at random intervals after a connection has been established. This is used to prevent a connection from being hijacked – after it has been authenticated. If a challenge fails at one of these random times, the connection is immediately terminated.

Note that both sides of the interface can request CHAP, but CHAP is typically used in only one direction.

Internet Protocol Control Protocol (IPCP)

The IPCP is the last protocol to execute before the PPP communication is available for NetX IP data transfer. The main purpose of this protocol is for one peer to inform the other of its IP address. Once the IP address is setup, NetX IP data transfer is enabled.

Data Transfer

As mentioned previously, NetX IP data packets reside in PPP frames with a protocol ID of 0x0021. All received data packets are placed in one or more NX_PACKET structures and transferred to the NetX receive processing. On transmission, the NetX packet contents are placed in an AHDLC frame and transmitted.

PPP RFCs

Net PPP is compliant with RFC1332, RFC1334, RFC1661, RFC1994, and related RFCs.

Chapter 2

Installation and Use of PPP

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

Product Distribution

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

<code>nx_ppp.h</code>	Header file for PPP for NetX
<code>nx_ppp.c</code>	C Source file for PPP for NetX
<code>nx_ppp.pdf</code>	PDF description of PPP for NetX
<code>demo_netx_ppp.c</code>	NetX PPP demonstration

PPP Installation

In order to use PPP for NetX, the entire distribution mentioned previously should be copied to the same directory where NetX is installed. For example, if NetX is installed in the directory "*\threadx\arm7\green*" then the *nx_ppp.h* and *nx_ppp.c* files should be copied into this directory.

Using PPP

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

Using Modems

If a modem is required for connection to the internet, some special considerations are required in order to use the NetX PPP product. Basically, using a modem introduces additional initialization logic and logic

for loss of communication. In addition, most of the additional modem logic is done outside the context of NetX PPP. The basic flow of using the NetX PPP with a modem goes something like this:

1. Initialize Modem
2. Dial Internet Service Provider (ISP)
3. Wait for Connection
4. Wait for UserID Prompt
5. Start NetX PPP
- [PPP in operation]
6. Loss of Communication
7. Stop NetX PPP (or restart via nx_ppp_restart)

Initialize Modem

Using the application's low-level serial output routine, the modem is initialized via a series of ASCII character commands (see modem's documentation for more details).

Dial Internet Service Provider

Using the application's low-level serial output routine, the modem is instructed to dial the ISP. For example, the following is typical of an ASCII string used to dial an ISP at the number 123-4567:

"ATDT123456\r"

Wait for Connection

At this point, the application waits to receive indication from the modem that a connection has been established. This is accomplished by looking for characters from the application's low-level serial input routine. Typically, modems return an ASCII string "CONNECT" when a connection has been established.

Wait for User ID Prompt

Once the connection has been established, the application must now wait for an initial login request from the ISP. This typically takes the form of an ASCII string like "Login?"

Start NetX PPP

At this point, the NetX PPP can be started. This is accomplished by calling the *nx_ppp_create* service followed by the *nx_ip_create* service. Additional services to enable PAP and to setup the PPP IP addresses might also be required. Please review the following sections of this guide for more information.

Loss of Communication

Once PPP is started, any non-PPP information is passed to the "invalid packet handling" routine the application specified to the *nx_ppp_create* service. Typically, modems send an ASCII string such as "NO CARRIER" when communication is lost with the ISP. When the application receives a non-PPP packet with such information, it should proceed to either stop the NetX PPP instance or to restart the PPP state machine via the *nx_ppp_restart* API.

Stop NetX PPP

Stopping the NetX PPP is fairly straightforward. Basically, all created sockets must be unbound and deleted. Next, delete the IP instance via the *nx_ip_delete* service. Once the IP instance is deleted, the *nx_ppp_delete* service should be called to finish the process of stopping PPP. At this point, the application is now able to attempt to reestablish communication with the ISP.

Small Example System

An example that illustrates how easy it is to use NetX PPP is described in Figure 1.1 that appears below. In this example, the PPP include file *nx_ppp.h* is brought in at line 3. Next, PPP is created in *tx_application_define* at line 56. The PPP control block “*my_ppp*” was defined as a global variable at line 9 previously. Note that PPP should be created prior to creating the IP instance. After successful creation of PPP and IP, the thread “*my_thread*” waits for the PPP link to come alive at line 98. At line 104, both PPP and NetX are fully operational.

The one item not shown in this example is the application’s serial byte receive ISR. It will need to call *nx_ppp_byte_receive* with “*my_ppp*” and the byte received as input parameters.

```

0001 #include "tx_api.h"
0002 #include "nx_api.h"
0003 #include "nx_ppp.h"
0004
0005 #define DEMO_STACK_SIZE 4096
0006 TX_THREAD my_thread;
0007 NX_PACKET_POOL my_pool;
0008 NX_IP my_ip;
0009 NX_PPP my_ppp;
0010
0011 /* Define function prototypes. */
0012
0013 void my_thread_entry(ULONG thread_input);
0014 void my_serial_driver_byte_output(UCHAR byte);
0015 void my_invalid_packet_handler(NX_PACKET *packet_ptr);
0016
0017 /* Define main entry point. */
0018 intmain()
0019 {
0020
0021     /* Enter the ThreadX kernel. */
0022     tx_kernel_enter();
0023 }
0024
0025 /* Define what the initial system looks like. */
0026
0027 void tx_application_define(void *first_unused_memory)
0028 {
0029     CHAR *pointer;
0030     UINT status;
0031
0032     /* Setup the working pointer. */
0033     pointer = (CHAR *) first_unused_memory;
0034
0035     /* Create "my_thread". */
0036     tx_thread_create(&my_thread, "my thread", my_thread_entry, 0,
0037                     pointer, DEMO_STACK_SIZE,
0038                     2, 2, TX_NO_TIME_SLICE, TX_AUTO_START);
0039     pointer = pointer + DEMO_STACK_SIZE;
0040
0041     /* Initialize the NetX system. */
0042     nx_system_initialize();
0043
0044     /* Create a packet pool. */
0045     status = nx_packet_pool_create(&my_pool, "NetX Main Packet Pool",
0046                                   1024, pointer, 64000);
0047     pointer = pointer + 64000;
0048
0049     /* Check for pool creation error. */
0050     if (status)
0051         error_counter++;
0052 }

```

```

0055
0055 /* Create a PPP instance. */
0056 status = nx_ppp_create(&my_ppp, "My PPP", &my_ip, pointer, 1024, 2,
0057 &my_pool, my_invalid_packet_handler, my_serial_driver_byte_output);
0058 pointer = pointer + 1024;
0059 /* Check for PPP creation pool. */
0060 if (status)
0061     error_counter++;
0062
0063 /* Create an IP instance with the PPP driver. */
0064 status = nx_ip_create(&my_ip, "My NetX IP Instance",
0065 IP_ADDRESS(216,2,3,1), 0xFFFFFFFF00, &my_pool,
0066 nx_ppp_driver, pointer, DEMO_STACK_SIZE, 1);
0067 pointer = pointer + DEMO_STACK_SIZE;
0068
0069 /* Check for IP create errors. */
0070 if (status)
0071     error_counter++;
0072
0073 /* Enable ICMP for my IP Instance. */
0074 status = nx_icmp_enable(&my_ip);
0075
0076 /* Check for ICMP enable errors. */
0077 if (status)
0078     error_counter++;
0079
0080 /* Enable UDP. */
0081 status = nx_udp_enable(&my_ip);
0082 if (status)
0083     error_counter++;
0084 }
0085
0086
0087 /* Define my thread. */
0088
0089 void my_thread_entry(ULONG thread_input)
0090 {
0091
0092     UINT status;
0093     ULONG ip_status;
0094     NX_PACKET *my_packet;
0095
0096
0097     /* Wait for the PPP link in my_ip to become enabled. */
0098     status = nx_ip_status_check(&my_ip, NX_IP_LINK_ENABLED, &ip_status, 3000);
0099
0100     /* Check for IP status error. */
0101     if (status)
0102         return;
0103
0104     /* Link is fully up and operational. All NetX activities
0105        are now available. */
0106
0107 }

```

Figure 1.1 Example of PPP use with NetX

Configuration Options

There are several configuration options for building PPP for NetX. The following list describes each in detail:

Define	Meaning
NX_DISABLE_ERROR_CHECKING	Defined, this option removes the basic PPP error checking. It is typically used after the application has been debugged.
NX_PPP_BASE_TIMEOUT	This defines the period rate (in timer ticks) that the PPP thread task is woken to check for PPP events. The default value is $1 \times \text{NX_IP_PERIODIC_RATE}$ (100 ticks).
NX_PPP_DISABLE_INFO	If defined, internal PPP information gathering is disabled.
NX_PPP_DEBUG_LOG_ENABLE	If defined, internal PPP debug log is enabled.
NX_PPP_DEBUG_LOG_PRINT_ENABLE	If defined, internal PPP debug log <i>printf</i> to <i>stdio</i> is enabled. This is only valid if the debug log is also enabled.
NX_PPP_DEBUG_LOG_SIZE	Size of debug log (number of entries in the debug log). On reaching the last entry, the debug capture wraps to the first entry and overwrites any data previously captured. The default value is 50.
NX_PPP_DEBUG_FRAME_SIZE	Maximum amount of data captured from a received packet payload and saved to debug output. The default value is 50.

NX_PPP_DISABLE_CHAP	If defined, internal PPP CHAP logic is removed, including the MD5 digest logic.
NX_PPP_DISABLE_PAP	If defined, internal PPP PAP logic is removed.
NX_PPP_DNS_OPTION_DISABLE	If defined, DNS Option is disabled in the IPCP response. By default this option is not defined (DNS option is set).
NX_PPP_HASHED_VALUE_SIZE	Specifies the size of “hashed value” strings used in CHAP authentication. The default value is set to 16 bytes, but can be redefined prior to inclusion of <i>nx_ppp.h</i> .
NX_PPP_MAX_LCP_PROTOCOL_RETRIES	This defines the max number of retries if the PPP times out before sending another LCP configure request message. When this number is reached the PPP handshake is aborted and the link status is down. The default value is 20.
NX_PPP_MAX_PAP_PROTOCOL_RETRIES	This defines the max number of retries if the PPP times out before sending another PAP authentication request message. When this number is reached the PPP handshake is aborted and the link status is down. The default value is 20.
NX_PPP_MAX_CHAP_PROTOCOL_RETRIES	This defines the max number of retries if the PPP times out before sending another CHAP challenge message. When this number is reached the PPP handshake is

aborted and the link status is down. The default value is 20.

NX_PPP_MAX_IPCP_PROTOCOL_RETRIES

This defines the max number of retries if the PPP times out before sending another IPCP configure request message. When this number is reached the PPP handshake is aborted and the link status is down. The default value is 20.

NX_PPP_MRU

Specifies the Maximum Receive Unit (MRU) for PPP. By default, this value is 1,500 bytes (the minimum value). This define can be set by the application prior to inclusion of *nx_ppp.h*.

NX_PPP_MINIMUM_MRU

Specifies the minimum MRU received in an LCP configure request message. By default, this value is 1,500 bytes (the minimum value). This define can be set by the application prior to inclusion of *nx_ppp.h*.

NX_PPP_NAME_SIZE

Specifies the size of “name” strings used in authentication. The default value is set to 32bytes, but can be redefined prior to inclusion of *nx_ppp.h*.

NX_PPP_PASSWORD_SIZE

Specifies the size of “password” strings used in authentication. The default value is set to 32bytes, but can be redefined prior to inclusion of *nx_ppp.h*.

NX_PPP_PROTOCOL_TIMEOUT

This defines the wait option (in seconds) for the PPP task to receive a response to a PPP protocol request message. The default value is 4 seconds.

NX_PPP_RECEIVE_TIMEOUTS	This defines the number of times the PPP thread task times out waiting to receive the next character in a PPP message stream. Thereafter, PPP releases the packet and begins waiting to receive the next PPP message. The default value is 4.
NX_PPP_SERIAL_BUFFER_SIZE	Specifies the size of the receive character serial buffer. By default, this value is 3,000 bytes. This define can be set by the application prior to inclusion of <i>nx_ppp.h</i> .
NX_PPP_TIMEOUT	This defines the wait option (in timer ticks) for allocating packets to transmit data as well as buffer PPP serial data into packets to send to the IP layer. The default value is 4*NX_IP_PERIODIC_RATE (400 ticks).
NX_PPP_THREAD_TIME_SLICE	Time-slice option for PPP threads. By default, this value is TX_NO_TIME_SLICE. This define can be set by the application prior to inclusion of <i>nx_ppp.h</i> .
NX_PPP_VALUE_SIZE	Specifies the size of “value” strings used in CHAP authentication. The default value is set to 32bytes, but can be redefined prior to inclusion of <i>nx_ppp.h</i> .

Chapter 3

Description of PPP Services

This chapter contains a description of all NetX PPP services (listed below) in alphabetic order.

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

`nx_ppp_byte_receive`
Receive a byte from serial ISR

`nx_ppp_chap_challenge`
Generate a CHAP challenge

`nx_ppp_chap_enable`
Enable CHAP authentication

`nx_ppp_create`
Create a PPP instance

`nx_ppp_delete`
Delete a PPP instance

`nx_ppp_dns_address_get`
Get DNS IP address

`nx_ppp_interface_index_get`
Get IP interface index

`nx_ppp_ip_address_assign`
Assign IP addresses for IPCP

`nx_ppp_link_down_notify`
Notify application on link down

`nx_ppp_link_up_notify`
Notify application on link up

`nx_ppp_nak_authentication_notify`
Notify application if authentication NAK is received

`nx_ppp_pap_enable`
Enable PAP authentication

`nx_ppp_ping_request`
Send an LCP echo request

`nx_ppp_raw_string_send`
Send non PPP string

`nx_ppp_restart`
Restart PPP processing

`nx_ppp_start`
Startt PPP service

`nx_ppp_status_get`
Get current PPP status

`nx_ppp_stop`
Stop PPP service

nx_ppp_byte_receive

Receive a byte from serial ISR

Prototype

```
UINT nx_ppp_byte_receive(NX_PPP *ppp_ptr, UCHAR byte);
```

Description

This service is typically called from the application's serial driver Interrupt Service Routine (ISR) to transfer a received byte to PPP. When called, this routine places the received byte into a circular byte buffer and notifies the appropriate PPP thread for processing.

Input Parameters

ppp_ptr Pointer to PPP control block.

byte Byte received from serial device

Return Values

NX_SUCCESS	(0x00)	Successful PPP byte receive.
NX_PPP_BUFFER_FULL	(0xB1)	PPP serial buffer is already full.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.

Allowed From

Threads, ISRs

Example

```
/* Notify "my_ppp" of a received byte. */
status = nx_ppp_byte_receive(&my_ppp, new_byte);

/* If status is NX_SUCCESS the received byte was successfully
   buffered. */
```

nx_ppp_chap_challenge

Generate a CHAP challenge

Prototype

```
UINT nx_ppp_chap_challenge(NX_PPP *ppp_ptr);
```

Description

This service initiates a CHAP challenge after the PPP connection is already up and running. This gives the application the ability to verify the authenticity of the connection on a periodic basis. If the challenge is unsuccessful, the PPP link is closed.

Input Parameters

ppp_ptr Pointer to PPP control block.

Return Values

NX_SUCCESS	(0x00)	Successful PPP challenge initiated.
NX_PPP_FAILURE	(0xB0)	Invalid PPP challenge, CHAP was enabled only for response.
NX_NOT_IMPLEMENTED	(0x80)	CHAP logic was disabled via NX_PPP_DISABLE_CHAP.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Initiate a PPP challenge for instance "my_ppp". */
status = nx_ppp_chap_challenge(&my_ppp);

/* If status is NX_SUCCESS a CHAP challenge "my_ppp" was successfully
initiated. */
```

nx_ppp_chap_enable

Enable CHAP authentication

Prototype

```
UINT nx_ppp_chap_enable(NX_PPP *ppp_ptr,
    UINT (*get_challenge_values)(CHAR *rand_value, CHAR *id, CHAR *name),
    UINT (*get_responder_values)(CHAR *system, CHAR *name, CHAR *secret),
    UINT (*get_verification_values)(CHAR *system, CHAR
                                    *name, CHAR *secret));
```

Description

This service enables the Challenge-Handshake Authentication Protocol (CHAP) for the specified PPP instance.

If the “***get_challenge_values***” and “***get_verification_values***” function pointers are specified, CHAP is required by this PPP instance. Otherwise, CHAP only responds to the peer’s challenge requests.

There are several data items referenced below in the required callback functions. The data items *secret*, *name*, and *system* are expected to be NULL-terminated strings with a maximum size of NX_PPP_NAME_SIZE-1. The data item *rand_value* is expected to be a NULL-terminated string with a maximum size of NX_PPP_VALUE_SIZE-1. The data item *id* is a simple unsigned character type.

Note that this function must be called after *nx_ppp_create* but before *nx_ip_create* or *nx_ip_interface_attach*.

Input Parameters

ppp_ptr	Pointer to PPP control block.
get_challenge_values	Pointer to application function to retrieve values used for the challenge. Note that the <i>rand_value</i> , <i>id</i> , and <i>secret</i> values must be copied into the supplied destinations.
get_responder_values	Pointer to application function that retrieves values used to respond to a challenge. Note that the <i>system</i> , <i>name</i> , and <i>secret</i> values must be copied into the supplied destinations.
get_verification_values	Pointer to application function that retrieves values used to verify the challenge response. Note that the <i>system</i> , <i>name</i> , and <i>secret</i> values must be copied into the supplied destinations.

Return Values

NX_SUCCESS	(0x00)	Successful PPP CHAP enable
NX_NOT_IMPLEMENTED	(0x80)	CHAP logic was disabled via NX_PPP_DISABLE_CHAP.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer or callback function pointer. Note that if <i>get_challenge_values</i> is specified, then the <i>get_verification_values</i> function must also be supplied.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Example

```

CHAR    name_string[] = "username";
CHAR    rand_value_string[] = "123456";
CHAR    system_string[] = "system";
CHAR    secret_string[] = "secret";

/* Enable CHAP in both directions (CHAP challenger and CHAP responder) for
"my_ppp". */
status = nx_ppp_chap_enable(&my_ppp, get_challenge_values,
                             get_responder_values,
                             get_verification_values);

/* If status is NX_SUCCESS, "my_ppp" has CHAP enabled. */
...

/* Define the CHAP enable routines. */
UINT get_challenge_values(CHAR *rand_value, CHAR *id, CHAR *name)
{
    UINT    i;

    for (i = 0; i < (NX_PPP_NAME_SIZE-1); i++)
    {
        name[i] = name_string[i];
    }
    name[i] = 0;

    *id = '1'; /* One byte */
    for (i = 0; i < (NX_PPP_VALUE_SIZE-1); i++)
    {
        rand_value[i] = rand_value_string[i];
    }
    rand_value[i] = 0;

    return(NX_SUCCESS);
}

UINT get_responder_values(CHAR *system, CHAR *name, CHAR *secret)
{
    UINT    i;

```

```

    for (i = 0; i < (NX_PPP_NAME_SIZE-1); i++)
    {
        name[i] = name_string[i];
    }
    name[i] = 0;

    for (i = 0; i < (NX_PPP_NAME_SIZE-1); i++)
    {
        system[i] = system_string[i];
    }
    system[i] = 0;

    for (i = 0; i < (NX_PPP_NAME_SIZE-1); i++)
    {
        secret[i] = secret_string[i];
    }
    secret[i] = 0;

    return(NX_SUCCESS);
}

UINT get_verification_values(CHAR *system, CHAR *name, CHAR *secret)
{
    UINT i;

    for (i = 0; i < (NX_PPP_NAME_SIZE-1); i++)
    {
        name[i] = name_string[i];
    }
    name[i] = 0;

    for (i = 0; i < (NX_PPP_NAME_SIZE-1); i++)
    {
        system[i] = system_string[i];
    }
    system[i] = 0;

    for (i = 0; i < (NX_PPP_NAME_SIZE-1); i++)
    {
        secret[i] = secret_string[i];
    }
    secret[i] = 0;

    return(NX_SUCCESS);
}

```

nx_ppp_create

Create a PPP instance

Prototype

```
UINT nx_ppp_create(NX_PPP *ppp_ptr, CHAR *name, NX_IP *ip_ptr,
                  VOID *stack_memory_ptr, ULONG stack_size,
                  UINT thread_priority, NX_PACKET_POOL *pool_ptr,
                  void (*ppp_invalid_packet_handler)(NX_PACKET *packet_ptr)),
                  void (*ppp_byte_send)(UCHAR byte));
```

Description

This service creates a PPP instance for the specified NetX IP instance.

Note that it is generally a good idea to create the NetX IP thread at a higher priority than the PPP thread priority. Please refer to the *nx_ip_create* service for more information on specifying the IP thread priority.

Input Parameters

ppp_ptr	Pointer to PPP control block.
name	Name of this PPP instance.
ip_ptr	Pointer to control block for not-yet-created IP instance.
stack_memory_ptr	Pointer to start of PPP thread's stack area.
stack_size	Size in bytes in the thread's stack.
pool_ptr	Pointer to default packet pool.
thread_priority	Priority of internal PPP threads (1-31).
ppp_invalid_packet_handler	Function pointer to application's handler for all non-PPP packets. The NetX PPP typically calls this routine during initialization. This is where the application can respond to modem commands or in the case of Windows XP, the NetX PPP application can initiate PPP by responding with "CLIENT SERVER" to the initial "CLIENT" sent by Windows XP.
ppp_byte_send	Function pointer to application's serial byte output routine.

Return Values

NX_SUCCESS	(0x00)	Successful PPP create.
NX_PTR_ERROR	(0x07)	Invalid PPP, IP, or byte output function pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Example

```
/* Create "my_ppp" for IP instance "my_ip". */
status = nx_ppp_create(&my_ppp, "my PPP", &my_ip, stack_start, 1024, 2,
                      &my_pool, my_invalid_packet_handler, my_out_byte);

/* If status is NX_SUCCESS the PPP instance was successfully
   created. */
```

nx_ppp_delete

Delete a PPP instance

Prototype

```
UINT nx_ppp_delete(NX_PPP *ppp_ptr);
```

Description

This service deletes the previously created PPP instance.

Input Parameters

ppp_ptr	Pointer to PPP control block.
----------------	-------------------------------

Return Values

NX_SUCCESS	(0x00)	Successful PPP deletion.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Delete PPP instance "my_ppp". */  
status = nx_ppp_delete(&my_ppp);  
  
/* If status is NX_SUCCESS the "my_ppp" was successfully deleted. */
```

nx_ppp_dns_address_get

Get DNS IP address

Prototype

```
UINT nx_ppp_dns_address_get(NX_PPP *ppp_ptr, ULONG *dns_address_ptr);
```

Description

This service retrieves the DNS IP address supplied by the peer. If no IP address was supplied by the peer, an IP address of 0 is returned.

Input Parameters

ppp_ptr	Pointer to PPP control block.
dns_address_ptr	Destination for DNS IP address

Return Values

NX_SUCCESS	(0x00)	Successful PPP address get
NX_PPP_NOT_ESTABLISHED	(0xB5)	PPP has not completed negotiation with peer.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.

Allowed From

Initialization, threads, timers, ISRs

Example

```
ULONG my_dns_address;

/* Get DNS IP address supplied by peer. */
status = nx_ppp_dns_address_get(&my_ppp, &my_dns_address);

/* If status is NX_SUCCESS the "my_dns_address" contains the DNS IP address -
   if the peer supplied one. */
```

nx_ppp_interface_index_get

Get IP interface index

Prototype

```
UINT nx_ppp_interface_index_get(NX_PPP *ppp_ptr, UINT *index_ptr);
```

Description

This service retrieves the IP interface index associated with this PPP instance. This is only useful when the PPP instance is not the primary interface of an IP instance.

Input Parameters

ppp_ptr	Pointer to PPP control block.
index_ptr	Destination for interface index

Return Values

NX_SUCCESS	(0x00)	Successful PPP index get.
NX_IN_PROGRESS	(0x37)	PPP has not completed initialization.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.

Allowed From

Initialization, threads, timers, ISRs

Example

```
ULONG my_index;

/* Get the interface index for this PPP instance. */
status = nx_ppp_interface_index_get(&my_ppp, &my_index);

/* If status is NX_SUCCESS the "my_index" contains the IP interface index for
   this PPP instance. */
```

nx_ppp_ip_address_assign

Assign IP addresses for IPCP

Prototype

```
UINT nx_ppp_ip_address_assign(NX_PPP *ppp_ptr, ULONG local_ip_address,
                              ULONG peer_ip_address);
```

Description

This service sets up the local and peer IP addresses for use in the Internet Protocol Control Protocol (IPCP). The PPP application should invoke this service on a PPP instance with valid IP addresses for itself and the other peer. If no valid addresses are registered with a PPP instance, it must rely on the PPP peer to define its IP address.

Input Parameters

ppp_ptr	Pointer to PPP control block.
local_ip_address	Local IP address
peer_ip_address	Peer's IP address

Return Values

NX_SUCCESS	(0x00)	Successful PPP address assignment.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Example

```
/* Set IP addresses for "my_ppp". */
status = nx_ppp_ip_address_assign(&my_ppp, IP_ADDRESS(256,2,2,187),
                                  IP_ADDRESS(256,2,2,188));

/* If status is NX_SUCCESS the "my_ppp" has the IP addresses. */
```

nx_ppp_link_down_notify

Notify application on link down

Prototype

```
UINT nx_ppp_link_down_notify(NX_PPP *ppp_ptr,
                             VOID (*link_down_callback)(NX_PPP *ppp_ptr));
```

Description

This service registers the application's link down notification callback with the specified PPP instance. If non-NULL, the application's link down callback function is called whenever the link goes down.

Input Parameters

ppp_ptr	Pointer to PPP control block.
link_down_callback	Application's link down notification function pointer. If NULL, link down notification is disabled.

Return Values

NX_SUCCESS	(0x00)	Successful link down notification callback registration.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.

Allowed From

Initialization, threads, timers, ISRs

Example

```
/* Register "my_link_down_callback" to be called whenever the PPP
   link goes down. */
status = nx_ppp_link_down_notify(&my_ppp, my_link_down_callback);

/* If status is NX_SUCCESS the function "my_link_down_callback" has been
   registered with this PPP instance. */

....

VOID my_link_down_callback(NX_PPP *ppp_ptr)
{
    /* On link down, simply restart PPP. */
    nx_ppp_restart(ppp_ptr);
}
```

nx_ppp_link_up_notify

Notify application on link up

Prototype

```
UINT nx_ppp_link_up_notify(NX_PPP *ppp_ptr,
                          VOID (*link_up_callback)(NX_PPP *ppp_ptr));
```

Description

This service registers the application's link up notification callback with the specified PPP instance. If non-NULL, the application's link up callback function is called whenever the link comes up.

Input Parameters

ppp_ptr	Pointer to PPP control block.
link_up_callback	Application's link up notification function pointer. If NULL, link up notification is disabled.

Return Values

NX_SUCCESS	(0x00)	Successful link up notification callback registration.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.

Allowed From

Initialization, threads, timers, ISRs

Example

```
/* Register "my_link_up_callback" to be called whenever the PPP
   link comes up. */
status = nx_ppp_link_up_notify(&my_ppp, my_link_up_callback);

/* If status is NX_SUCCESS the function "my_link_up_callback" has been
   registered with this PPP instance. */

...

VOID my_link_up_callback(NX_PPP *ppp_ptr)
{
    /* On link up, the application my want to start sending/receiving
       UPD/TCP data. */
}
```

nx_ppp_nak_authentication_notify

Notify application if authentication NAK received

Prototype

```
UINT nx_ppp_nak_authentication_notify(NX_PPP *ppp_ptr,
                                       void (*nak_authentication_notify)(void));
```

Description

This service registers the application's authentication nak notification callback with the specified PPP instance. If non-NULL, this callback function is called whenever the PPP instance receives a NAK during authentication.

Input Parameters

ppp_ptr	Pointer to PPP control block.
nak_authentication_notify	Pointer to function called when the PPP instance receives an authentication NAK. If NULL, the notification is disabled.

Return Values

NX_SUCCESS	(0x00)	Successful notification callback registration.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.

Allowed From

Initialization, threads, timers, ISRs

Example

```
/* Register "my_nak_auth_callback" to be called whenever the PPP
   receives a NAK during authentication. */
status = nx_ppp_nak_authentication_notify(&my_ppp, my_nak_auth_callback);

/* If status is NX_SUCCESS the function "my_nak_auth_callback" has been
   registered with this PPP instance. */

VOID my_nak_auth_callback(NX_PPP *ppp_ptr)
{
    /* Handle the situation of receiving an authentication NAK */
}
```


nx_ppp_pap_enable

Enable PAP Authentication

Prototype

```
UINT nx_ppp_pap_enable(NX_PPP *ppp_ptr,
    UINT (*generate_login)(CHAR *name, CHAR *password),
    UINT (*verify_login)(CHAR *name, CHAR *password));
```

Description

This service enables the Password Authentication Protocol (PAP) for the specified PPP instance. If the “***verify_login***” function pointer is specified, PAP is required by this PPP instance. Otherwise, PAP only responds to the peer’s PAP requirements as specified during LCP negotiation.

There are several data items referenced below in the required callback functions. The data item *name* is expected to be NULL-terminated string with a maximum size of NX_PPP_NAME_SIZE-1. The data item *password* is also expected to be a NULL-terminated string with a maximum size of NX_PPP_PASSWORD_SIZE-1.

Note that this function must be called after *nx_ppp_create* but before *nx_ip_create* or *nx_ip_interface_attach*.

Input Parameters

ppp_ptr	Pointer to PPP control block.
generate_login	Pointer to application function that produces a <i>name</i> and <i>password</i> for authentication by the peer. Note that the <i>name</i> and <i>password</i> values must be copied into the supplied destinations.
verify_login	Pointer to application function that verifies the <i>name</i> and <i>password</i> supplied by the peer. This routine must compare the supplied <i>name</i> and <i>password</i> . If this routine returns NX_SUCCESS, the name and password are correct and PPP can proceed to the next step. Otherwise, this routine returns NX_PPP_ERROR and PPP simply waits for another name and password.

Return Values

NX_SUCCESS	(0x00)	Successful PPP PAP enable.
NX_NOT_IMPLEMENTED	(0x80)	PAP logic was disabled via NX_PPP_DISABLE_PAP.

NX_PTR_ERROR	(0x07)	Invalid PPP pointer or application function pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Example

```

CHAR    name_string[] = "username";
CHAR    password_string[] = "password";

/* Enable PAP for PPP instance "my_ppp". */
status = nx_ppp_pap_enable(&my_ppp, my_generate_login, my_verify_login);
/* If status is NX_SUCCESS the "my_ppp" now has PAP enabled. */
...

/* Define callback routines for PAP enable. */
UINT    generate_login(CHAR *name, CHAR *password)
{
    UINT    i;

    for (i = 0; i < (NX_PPP_NAME_SIZE-1); i++)
        name[i] = name_string[i];
    name[i] = 0;

    for (i = 0; i < (NX_PPP_PASSWORD_SIZE-1); i++)
        password[i] = password_string[i];
    password_string[i] = 0;

    return(NX_SUCCESS);
}

UINT    verify_login(CHAR *name, CHAR *password)
{
    /* Assume name and password are correct. Normally,
    a comparison would be made here! */
    printf("Name: %s, Password: %s\n", name, password);

    return(NX_SUCCESS);
}

```

nx_ppp_ping_request

Send an LCP ping request

Prototype

```
UINT nx_ppp_ping_request(NX_PPP *ppp_ptr, CHAR *data, UINT data_size,
                        ULONG wait_opion);
```

Description

This service sends an LCP ping request and sets a flag that the PPP device is waiting for an echo response. The service returns as soon as the request is sent. It does not wait for a response.

When a matching echo response is received, the PPP thread task will clear the flag. The PPP device must have completed the LCP part of the PPP negotiation.

This service is useful for PPP set ups where polling the hardware for link status may not be readily possible.

Input Parameters

ppp_ptr	Pointer to PPP control block.
data	Pointer to data to send in echo request.
data_size	Size of data to send
wait_option	Time to wait to send the LCP echo message.

Return Values

NX_SUCCESS	(0x00)	Successful sent echo request.
NX_PPP_NOT_ESTABLISHED	(0xB5)	PPP connection not established.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer or application function pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Application threads

Example

```
CHAR    buffer[] = "username";
UINT    buffer_length = strlen("username ");
```

```

/* Send an LCP ping request". */
status = nx_ppp_ping_request(&my_ppp, &buffer[0], buffer_length, 200);

/* If status is NX_SUCCESS the LCP echo request was successfully sent. Now wait to
   receive a response. */

while(my_ppp.nx_ppp_lcp_echo_reply_id > 0)
{
    tx_thread_sleep(100);
}

/* Got a valid reply! */

```

nx_ppp_raw_string_send

Send a raw ASCII string

Prototype

```
UINT nx_ppp_raw_string_send(NX_PPP *ppp_ptr, CHAR *string_ptr);
```

Description

This service sends a non-PPP ASCII string directly out the PPP interface. It is typically used after PPP receives a non-PPP packet that contains modem control information.

Input Parameters

ppp_ptr	Pointer to PPP control block.
string_ptr	Pointer to string to send.

Return Values

NX_SUCCESS	(0x00)	Successful PPP raw string send.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer or string pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Send "CLIENTSERVER" to "CLIENT" sent by windows 98 before PPP is
initiated. */
status = nx_ppp_raw_string_send(&my_ppp, "CLIENTSERVER");
/* If status is NX_SUCCESS the raw string was successfully Sent via PPP. */
```

nx_ppp_restart

Restart PPP processing

Prototype

```
UINT nx_ppp_restart(NX_PPP *ppp_ptr);
```

Description

This service restarts the PPP processing. It is typically called when the link needs to be re-established either from a link down callback or by a non-PPP modem message indicating communication was lost.

Input Parameters

ppp_ptr	Pointer to PPP control block.
----------------	-------------------------------

Return Values

NX_SUCCESS	(0x00)	Successful PPP restart initiated.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Example

```
/* Restart the PPP instance "my_ppp". */
status = nx_ppp_restart(&my_ppp);

/* If status is NX_SUCCESS the PPP instance has been restarted. */
```

nx_ppp_start

Start PPP service

Prototype

```
UINT nx_ppp_start(NX_PPP *ppp_ptr);
```

Description

This service starts the PPP processing.

Input Parameters

ppp_ptr	Pointer to PPP control block.
----------------	-------------------------------

Return Values

NX_SUCCESS	(0x00)	Successful PPP start request.
NX_PPP_ALREADY_STARTED	(0xb9)	PPP already started.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers, ISRs

Example

```
UINT status;

/* Start the PPP instance "my_ppp". */
status = nx_ppp_start (&my_ppp);

/* If status is NX_SUCCESS the PPP instance has been started. */
```

nx_ppp_status_get

Get current PPP status

Prototype

```
UINT nx_ppp_status_get(NX_PPP *ppp_ptr, UINT *status_ptr);
```

Description

This service gets the current status of the specified PPP instance.

Input Parameters

ppp_ptr	Pointer to PPP control block.
status_ptr	Destination for the PPP status, the following are possible status values:
	NX_PPP_STATUS_ESTABLISHED
	NX_PPP_STATUS_LCP_IN_PROGRESS
	NX_PPP_STATUS_LCP_FAILED
	NX_PPP_STATUS_PAP_IN_PROGRESS
	NX_PPP_STATUS_PAP_FAILED
	NX_PPP_STATUS_CHAP_IN_PROGRESS
	NX_PPP_STATUS_CHAP_FAILED
	NX_PPP_STATUS_IPCP_IN_PROGRESS
	NX_PPP_STATUS_IPCP_FAILED
	Note that the status is only valid if the API returns NX_SUCCESS. In addition, if any of the *_FAILED status values are returned, PPP processing is effectively stopped until it is restarted again by the application.

Return Values

NX_SUCCESS	(0x00)	Successful PPP status request.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.

Allowed From

Initialization, threads, timers, ISRs

Example

```
UINT ppp_status;  
UINT status;  
  
/* Get the current status of PPP instance "my_ppp". */  
status = nx_ppp_status_get(&my_ppp, &ppp_status);  
  
/* If status is NX_SUCCESS the current internal PPP status is contained in  
   "ppp_status". */
```

nx_ppp_stop

Stop PPP service

Prototype

```
UINT nx_ppp_stop (NX_PPP *ppp_ptr);
```

Description

This service stops the PPP processing. If LCP connection is already established or in negotiation, PPP sends LCP terminate request to notify peer PPP instance to terminate the connection, and call link down notification callback function after receiving the LCP terminate ACK, or after the LCP retry counter exceeds the limit. In all other situations, the nx_ppp_stop service sets the state to stopped, and call the link down notification callback function.

Input Parameters

ppp_ptr Pointer to PPP control block.

Return Values

NX_SUCCESS	(0x00)	Successful PPP stop request.
NX_PPP_ALREADY_STOPPED	(0xb8)	PPP already stopped.
NX_PTR_ERROR	(0x07)	Invalid PPP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers, ISRs

Example

```
UINT status;

/* Stop the PPP instance "my_ppp". */
status = nx_ppp_stop (&my_ppp);

/* If status is NX_SUCCESS the PPP instance has been stopped. */
```