NETX™

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

Point-to-Point Protocol (PPP)

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

Part Number: 000-1053
Revision 5.0

Contents

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

NetX 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:

nx_ppp.h	Header file for PPP for NetX
nx_ppp.c	C Source file for PPP for NetX
nx_ppp.pdf	PDF description of PPP for NetX
demo_netx_ppp.c	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

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 UserID 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 stop the NetX PPP.

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 of 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 int main()
0019 {
0020
0021     /* Enter the ThreadX kernel. */
0022     tx_kernel_enter();
0023 }
0024
0025
0026 /* Define what the initial system looks like. */
0027
0028 void    tx_application_define(void *first_unused_memory)
0029 {
0030
0031 CHAR    *pointer;
0032 UINT    status;
0033
0034
0035     /* Setup the working pointer. */
0036     pointer =  (CHAR *) first_unused_memory;
0037
0038     /* Create “my_thread”. */
0039      tx_thread_create(&my_thread, "my thread", my_thread_entry, 0,
0040                  pointer, DEMO_STACK_SIZE,
0041                  2, 2, TX_NO_TIME_SLICE, TX_AUTO_START);
0042     pointer =  pointer + DEMO_STACK_SIZE;
0043
0044     /* Initialize the NetX system. */
0045     nx_system_initialize();
0046
0047     /* Create a packet pool. */
0048     status =  nx_packet_pool_create(&my_pool, "NetX Main Packet Pool",
0049                                         1024, pointer, 64000);
0050     pointer = pointer + 64000;
0051
0052     /* Check for pool creation error. */
0053     if (status)
0054         error_counter++;
```

```

    /* Create a PPP instance. */
    status = nx_ppp_create(&my_ppp, “My PPP”, &my_ip, pointer, 1024, 2,
          &my_pool, my_invalid_packet_handler, my_serial_driver_byte_output);
    pointer =  pointer + 1024;
    /* Check for PPP creation pool. */
    if (status)
        error_counter++;

    /* Create an IP instance with the PPP driver. */
    status = nx_ip_create(&my_ip,"My NetX IP Instance",
            IP_ADDRESS(216,2,3,1), 0xFFFFFF00, &my_pool,
            nx_ppp_driver, pointer, DEMO_STACK_SIZE, 1);
    pointer =  pointer + DEMO_STACK_SIZE;

    /* Check for IP create errors. */
    if (status)
        error_counter++;

    /* Enable ICMP for my IP Instance. */
    status =  nx_icmp_enable(&my_ip);

    /* Check for ICMP enable errors. */
    if (status)
        error_counter++;

    /* Enable UDP. */
    status =  nx_udp_enable(&my_ip);
    if (status)
        error_counter++;
}


/* Define my thread. */

void    my_thread_entry(ULONG thread_input)
{

UINT        status;
ULONG       ip_status;
NX_PACKET   *my_packet;


    /* Wait for the PPP link in my_ip to become enabled. */
    status =  nx_ip_status_check(&my_ip,NX_IP_LINK_ENABLED,&ip_status,3000);

    /* Check for IP status error. */
    if (status)
       return;

    /* Link is fully up and operational. All NetX activities
       are now available. */

}
```

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

NX_PPP_VALUE_SIZE	Specifies the size of “value” strings used in authentication. The default value is set to 32 bytes, but can be redefined prior to inclusion of *nx_ppp.h.*
NX_PPP_HASHED_VALUE_SIZE	Specifies the size of “hashed value” strings used in authentication. The default value is set to 16 bytes, but can be redefined prior to inclusion of *nx_ppp.h.*
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 printf to stdio is enabled. This is only valid if the debug log is also enabled.
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.

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_ip_address_assign
Assign IP addresses for IPCP

nx_ppp_pap_enable
Enable PAP authentication

nx_ppp_driver
NetX PPP driver function

nx_ppp_raw_string_send
Send non PPP string

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	(0x16)	Invalid PPP pointer.

Allowed From

Application ISRs, threads

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. */
```

See Also

nx_ppp_chap_challenge, nx_ppp_chap_enable, nx_ppp_create, nx_ppp_delete, nx_ppp_ip_address_assign, nx_ppp_pap_enable, nx_ppp_driver, nx_ppp_raw_string_send

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	(0x16)	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. */
```

See Also

nx_ppp_byte_receive, nx_ppp_chap_enable, nx_ppp_create, nx_ppp_delete, nx_ppp_ip_address_assign, nx_ppp_pap_enable, nx_ppp_driver, nx_ppp_raw_string_send

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.

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 *rand_value*, *id*, and *secret* 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 *system*, *name*, and *secret* 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 *system*, *name*, and *secret* 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	(0x16)	Invalid PPP pointer or callback function pointer. Note that if *get_challenge_values* is specified, then the *get_verification_values* 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, my_get_challenge_values,
                                     my_get_responder_values,
                                     my_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_string[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);
}
```

See Also

nx_ppp_byte_receive, nx_ppp_chap_challenge, nx_ppp_create, nx_ppp_delete, nx_ppp_ip_address_assign, nx_ppp_pap_enable, nx_ppp_driver, nx_ppp_raw_string_send

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. *This function must be called prior to creating the 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 "CLIENTSERVER" 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_PPP_FAILURE	(0xB0)	Internal PPP in creation.
NX_PTR_ERROR	(0x16)	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. */
```

See Also

nx_ppp_byte_receive, nx_ppp_chap_challenge, nx_ppp_chap_enable, nx_ppp_delete, nx_ppp_ip_address_assign, nx_ppp_pap_enable, nx_ppp_driver, nx_ppp_raw_string_send

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. Note that the corresponding IP instance must have already been deleted.

Note that the IP instance must already be deleted before this function is called.

Input Parameters

ppp_ptr	Pointer to PPP control block.

Return Values

NX_SUCCESS	(0x00)	Successful PPP deletion.
NX_PPP_FAILURE	(0xB0)	Invalid PPP deletion or IP instance was not deleted prior to this call.
NX_PTR_ERROR	(0x16)	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. */
```

See Also

nx_ppp_byte_receive, nx_ppp_chap_challenge, nx_ppp_chap_enable, nx_ppp_create, nx_ppp_ip_address_assign, nx_ppp_pap_enable, nx_ppp_driver, nx_ppp_raw_string_send

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). It should be used when IP address assignment is needed by either peer. If this PPP instance has the valid IP addresses for itself and the other peer, they should be specified in this call. Otherwise, if this peer relies on the other to define its IP address, this routine should be called with zero values for both IP addresses.

Note that the only instance where this routine should not be called is if both entities have their IP addresses hard-coded.

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	(0x16)	Invalid PPP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.

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. */
```

See Also

nx_ppp_byte_receive, nx_ppp_chap_challenge, nx_ppp_chap_enable, nx_ppp_create, nx_ppp_delete, nx_ppp_pap_enable, nx_ppp_driver, nx_ppp_raw_string_send

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.

Input Parameters

ppp_ptr	Pointer to PPP control block.
generate_login	Pointer to application function that produces a *name* and *password* for authentication by the peer. Note that the *name* and *password* values must be copied into the supplied destinations.
verify_login	Pointer to application function that verifies the *name* and *password* supplied by the peer. This routine must compare the supplied *name* and *password*. If this routine returns NX_SUCCESS, the name and password are correct and PPP can proceed to the next step. Otherwise, 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	(0x16)	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);
}
```

See Also

nx_ppp_byte_receive, nx_ppp_chap_challenge, nx_ppp_chap_enable, nx_ppp_create, nx_ppp_delete, nx_ppp_ip_address_assign, nx_ppp_driver, nx_ppp_raw_string_send

nx_ppp_driver

NetX PPP driver function

Prototype

```
void  nx_ppp_driver(NX_IP_DRIVER *driver_req_ptr);
```

Description

This service is the NetX PPP driver function. It is not directly called by the application. Instead, it is supplied to the NetX IP create function. The remainder of initialization and link establishment is done through the context of NetX driver calls.

Input Parameters

driver_req_ptr	NetX driver request structure

Return Values

None

Allowed From

Threads

Example

```
/* Create an IP instance with the PPP driver specified. */
status =  nx_ip_create(&my_ip,"My NetX IP Instance",
                    IP_ADDRESS(216,2,3,1), 0xFFFFFF00, &my_pool,
                    nx_ppp_driver, pointer, DEMO_STACK_SIZE, 1);

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

See Also

nx_ppp_byte_receive, nx_ppp_chap_challenge, nx_ppp_chap_enable, nx_ppp_create, nx_ppp_delete, nx_ppp_ip_address_assign, nx_ppp_pap_enable, nx_ppp_raw_string_send

nx_ppp_raw_string_send

Send a raw ASCII string

Prototype

```
UINT  nx_ppp_raw_sting_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 an 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	(0x16)	Invalid PPP pointer or application function 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. */
```

See Also

nx_ppp_byte_receive, nx_ppp_chap_challenge, nx_ppp_chap_enable, nx_ppp_create, nx_ppp_delete, nx_ppp_ip_address_assign, nx_ppp_pap_enable, nx_ppp_driver