Contents

Azure RTOS NetX 概述

2021/4/29 •

Azure RTOS NetX 是工业级 TCP/IP IPv4 嵌入式网络堆栈，专门用于深度嵌入式实时应用程序和 IoT 应用程序。 Azure RTOS NetX 是 Microsoft 最初的 IPv4 网络堆栈，本质上是 Azure RTOS 的子集。NetX 为嵌入式应用程序提供 IPv4、TCP 和 UDP 等核心网络协议以及一整套其他更高级别的附加协议。Azure RTOS NetX 占用内存少、执行速度快、易于使用，因而成为最苛刻的嵌入式 IoT 应用程序的理想选择。

API 协议

TELNET

- 最少占用 0.5 KB 和 0.3 KB RAM
- 客户端和服务器支持
- 直观的 Telnet API：nx_telnet_ *

自动 IP

- 自动 IPv4 地址分配
- 最少占用 1.2 KB、300 字节的 RAM
- 直观的 AutoIP API：nx_autoip_ *

HTTP - 超文本传输协议 (HTTP)

- 最少占用 2.8 KB 到 4.8 KB 闪存，0.4 KB 到 1.0 KB RAM
- 客户端和服务器支持
- 直观的 HTTP API：nx_http_ *

SMTP - 简单邮件传输协议 (SMTP)

- 最少占用 4.1 KB 和 0.6 KB RAM
- 客户端支持
- 直观的 SMTP API：nx_smtp_ *

DHCP - 动态主机配置协议 (DHCP)

- 最少占用 3.6 KB 到 4.6 KB 闪存，2.7 KB RAM
- 客户端和服务器支持
- IPv4 支持
- 直观的 DHCP API：nx_dhcp_ *

P0P3 - 邮局协议版本 3 (POP3)

- 最少占用 8.1 KB 和 1.4 KB RAM
- 客户端支持
- 直观的 P0P3 API：nx_pop3_ *

SNMP - 简单网络管理协议 (SNMP)

- 最少占用 10.9 KB 和 2.6 KB RAM
- 适用于 VI、V2 和 V3 的代理支持
- 直观的 SNMP API：nx_snmp_ *

FTP、TFTP - 文件传输协议 (FTP)，日常文件传输协议 (TFTP)

- FTP 最少占用 1.8 KB 到 7.2 KB 闪存，0.6 KB 到 2.1 KB RAM

- TFTP 最少占用 1.7 KB 到 2.4 KB 闪存，0.3 KB 到 1.8 KB RAM
- 客户端和服务器支持
- 直观的 FTP 和 TFTP API：nx_ftp_ 或 nx_tftp_

PPP - 点对点协议 (PPP)

- 最少占用 7.1 KB 和 3.8 KB RAM
- 直观的 PPP API：nx_ppp_ *

SNTP - 简单网络时间协议 (SNTP)

- 最少占用 4 KB 和 0.5 KB RAM
- 客户端支持
- 直观的 SNTP API：nx_sntp_ *

Azure RTOS NetX API

- API 直观且一致
- 名词-动词命名约定
- 快速零复制 API 实现
- 所有 API 函数均具有前导 nx_*，可轻松识别为 Azure RTOS NetX
- API 阻塞函数具有可选线程超时
- 用于移植旧套接字代码的可选 BSD 层

IGMP - Internet 组管理协议 (IGMP)

- 最少占用 2.5 KB 闪存
- IPv4 多播组支持
- 已验证 IXIA IxANVL
- 可选 IGMP 统计信息
- 通过 Azure RTOS TraceX 进行系统级跟踪
- 直观的 IGMP API：nx_igmp_ *

UDP - 用户数据报协议 (UDP)

- 最少占用 2.5 KB 闪存，每套接字 124 个套接字字节的 RAM
- 快速、近线速的 TCP 数据包处理：
- 100 Mbps 以太网接收速度达 95 Mbps、MCU@100MHz、MCU 利用率为 14%
- 100 Mbps 以太网传输速度达 94 Mbps、MCU@100MHz、MCU 利用率为 10%
- UDP Fast Path™ 技术
- UDP 数量无限制
- 已验证 IXIA IxANVL
- 套接字接收时的可选挂起
- 所有挂起的可选超时
- 可选 UDP 统计信息
- 通过 Azure RTOS TraceX 进行系统级跟踪
- 直观的 UDP API：nx_udp_ *

TCP - 传输控制协议 (TCP)

- 最少占用 10.5 KB 到 12.5 KB 闪存，每套接字 280 字节的 RAM
- 快速、接近线速的 TCP 数据包处理：
- 100 Mbps 以太网接收速度达 93 Mbps、MCU@100MHz、MCU 利用率为 20%
- 100 Mbps 以太网传输速度达 94 Mbps、MCU@100MHz、MCU 利用率为 27%
- 可靠连接
- TCP 套接字的数量无限制

- 已验证 IXIA IxANVL
- 套接字接收/发送时的可选挂起
- 所有挂起的可选超时
- 可选 TCP 统计信息
- 通过 Azure RTOS TraceX 进行系统级跟踪
- 直观的 TCP API：nx_tcp_ *

ICMP - Internet 控制消息协议 (ICMP)

- 最少占用 2.5 KB 闪存
- IPv4 支持
- 已验证 IXIA IxANVL
- Ping 请求和 ping 响应
- 对 ping 请求的可选线程挂起
- 所有挂起的可选超时
- 可选 ICMP 统计信息
- 通过 Azure RTOS TraceX 进行系统级跟踪
- 直观的 ICMP API：nx_icmp_ *

IPv4 - Internet 协议 (IP)

- 最少占用 3.5 KB 到 8.5 KB 闪存，2 KB 到 3 KB RAM
- Piconet™ 体系结构
- 快速、近线速的性能
- 多接口支持
- 多宿主支持
- 静态路由支持
- IP 分段/重组支持
- IPv4 支持
- 已验证 IXIA IxANVL
- 阶段 II 就绪徽标认证
- 可选 IP 统计信息
- 明确定义的直观物理层驱动程序接口
- 通过 Azure RTOS TraceX 进行系统级跟踪
- 直观的 IP 层 API：nx_ip_ *、nxd_ip_ *
- 通过 TUV 和 UL 预认证，符合 IEC 61508 SIL 4、IEC 62304 C 类、ISO 26262 ASIL D 和 EN 50128 SW-SIL4

ARP/RARP - 地址解析协议 (ARP)、反向地址解析协议 (RARP)

- 最少占用 1.7 KB 闪存，RAM 大小
- 动态解析 32位 IPv4 和 48 位 MAC 地址
- 已验证 IXIA IxANVL
- 灵活的、用户定义的 ARP 缓存
- 免费 ARP 支持
- 由应用程序确定的可选 ARP/RARP 统计信息
- 通过 Azure RTOS TraceX 进行系统级跟踪
- 直观的 ARP/RARP API：nx_arp_ *、nx_rarp_ *

以太网、WiFi、蓝牙低功耗、15.4 等

内存占用少

对于基本的 IP 和 UDP 支持，Azure RTOS NetX 占用空间极小，只有 9 KB 到 15 KB。TCP 功能需要额外的 10 KB 到 13 KB 的指令区域内存。Azure RTOS NetX RAM 使用量通常介于 2.6 KB 到 3.6 KB 之间，外加由应用程序定义的数据包池内存。与 Azure RTOS ThreadX 一样，Azure RTOS NetX 的大小会根据应用程序使用的服务自动缩放。这几乎无需复杂的配置和生成参数，使开发人员能够更轻松地工作。

执行速度快

Azure RTOS NetX 提供零复制数据包发送/接收实现，与 Azure RTOS ThreadX 高度集成，以实现最快的性能。例如，Azure RTOS NetX 通常可以在 80 MHz（或以下）处理器上实现近线速的数据传输，而只使用一小部分处理器周期。

简单易用

Azure RTOS NetX 易于使用。Azure RTOS NetX API 既直观又功能强大。API 名称由实词组成，而非由其他网络产品中常见的“字母汤”或高度缩写的名称组成。所有 Azure RTOS NetX API 均具有前导 nx_，并遵循名词-动词命名约定。此外，整个 API 具有功能一致性。例如，所有挂起的 API 函数均具有可选超时，其功能完全一致。对于旧版应用程序，Azure RTOS NetX 提供额外的 BSD 套接字兼容层。此层可帮助开发人员轻松迁移大型网络应用程序。

互操作性验证

Azure RTOS NetX 符合 RFC 标准，可提供与大多数供应商设备的完全互操作性。Azure RTOS NetX 还将行业标准 IxANVL（自动网络验证库）用于 Azure RTOS NetX 核心 TCP/IP 协议实现。

高级技术

Azure RTOS NetX 是高级技术，其中包括：

- Piconet™ 体系结构
- 自动缩放
- DP Fast-Path Technology™
- 灵活的数据包管理
- 零复制 API 和实现
- 多宿主支持
- 所有挂起的可选超时
- 静态路由支持
- Azure RTOS TraceX 系统分析支持

最快面市时间

Azure RTOS NetX 易于安装、学习、使用、调试、验证、认证和维护。因此，Azure RTOS NetX 是最受嵌入式 IoT 设备（包括 Broadcom、Gainspan 等众多 SoC）欢迎的 TCP/IP 堆栈之一。我们的面市时间一致，其优势基于：

- 优质文档 – 请查看我们的《Azure RTOS NetX 用户指南》，亲自了解一下！
- 提供完整的源代码
- 易于使用的 API
- 全面且高级的功能集

只需一份简单的许可证

使用和测试源代码无需任何费用，部署到预许可设备中时，亦无需生产许可证费用，所有其他设备仅需要一份简单的年度许可证。

最优质的完整源代码

多年来，Azure RTOS NetX 源代码在质量和易于理解方面树立了标杆。此外，Azure RTOS NetX 约定每个文件具有一个功能，正因为此，你可以轻松导览至源代码。

支持最常用的体系结构

Azure RTOS NetX 开箱即用，在最流行的 32/64 位微处理器上运行，已经过充分测试且完全受支持，其中包括以下高级体系结构：

Analog Devices：SHARC、Blackfin、CM4xx

Andes Core：RISC-V

Ambiqmicro：Apollo MCU

ARM：ARM7、ARM9、ARM11、Cortex-M0/M3/M4/M7/A15/A5/A7/A8/A9/A5x 64-bi/A7x 64-bit/R4/R5, TrustZone ARMv8-M

Cadence：Xtensa、Diamond

CEVA：PSoC、PSoC 4、PSoC 5、PSoC 6、FM0+、FM3、MF4、WICED WiFi

Cypress：RISC-V

EnSilica：eSi-RISC

Infineon：XMC1000、XMC4000、TriCore

Intel；Intel FPGA：x36/Pentium、XScale、NIOS II、Cyclone、Arria 10

Microchip：AVR32、ARM7、ARM9、Cortex-M3/M4/M7、SAM3/4/7/9/A/C/D/E/G/L/SV、PIC24/PIC32

Microsemi：RISC-V

NXP：LPC、 ARM7、ARM9、PowerPC、68 K、i.MX、ColdFire、Kinetis Cortex-M3/M4

Renesas：SH、HS、V850、RX、RZ、Synergy

Silicon Labs：EFM32

Synopsys：ARC 600、700、ARC EM、ARC HS

ST：STM32、ARM7、ARM9、Cortex-M3/M4/M7

TI：C5xxx、C6xxx、Stellaris、Sitara、Tiva-C

Wave Computing：MIPS32 4K、24 K、34 K、1004 K、MIPS64 5K、microAptiv、interAptiv、proAptiv、M-Class

Xilinx：MicroBlaze、PowerPC 405、ZYNQ、ZYNQ UltraSCALE

列出的所有时间和大小数字均为估计值，可能与你的开发平台上的值有所不同。

关于 Azure RTOS NetX 用户指南

2021/4/29 •

本指南包含有关 Microsoft 高性能网络堆栈 Azure RTOS NetX 的综合信息。

本指南面向熟悉基本网络概念、Azure RTOS ThreadX 和 C 编程语言的嵌入式实时软件开发人员。

组织

第 1 章 - 介绍 Azure RTOS NetX

第 2 章 - 介绍在 ThreadX 应用程序中安装和使用 Azure RTOS NetX 的基本步骤。

第 3 章 - 提供 Azure RTOS NetX 系统的功能概述以及有关 TCP/IP 网络标准的基本信息。

第 4 章 - 详细介绍如何将应用程序的接口应用到 Azure RTOS NetX。

第 5 章 - 描述 Azure RTOS NetX 的网络驱动程序。

附录 A - Azure RTOS NetX 服务

附录 B - Azure RTOS NetX 常量

附录 C - Azure RTOS NetX 数据类型

附录 D - 与 BSD 兼容的套接字 API

附录 E - ASCII 图表

Azure RTOS NetX 数据类型

除了自定义的 Azure RTOS NetX 控制结构数据类型之外，Azure RTOS NetX 服务调用接口中还使用几种特殊数据类型。这些特殊数据类型会直接与底层 C 编译器的数据类型相映射。这样做是为了确保不同 C 编译器之间的可移植性。具体实现继承自 ThreadX，可在 ThreadX 分发中包含的 tx_port.h 文件中找到。

下面列出了 Azure RTOS NetX 服务调用数据类型及其关联的含义：

tttt	tt
UINT	基本的无符号整数。此类型必须支持 32 位无符号数据；但是，它将与最方便的无符号数据类型相映射。
ULONG	无符号 long 类型。此类型必须支持 32 位无符号数据。
VOID	几乎始终等效于编译器的 void 类型。
CHAR	通常为标准的 8 位字符类型。

Azure RTOS NetX 源中还使用其他数据类型。这些数据类型位于 tx_port.h 或 nx_port.h 文件中。

客户支持中心

请按照此处介绍的步骤，在 Azure 门户中提交支持票证，以进行提问或获取帮助。请在电子邮件中提供以下信息，以便我们可以更高效地解决你的支持请求：

1. 详细描述该问题，包括发生频率以及能否可靠地重现该问题。
2. 发生问题前对应用程序和/或 Azure RTOS NetX 所做的任何更改的详细说明。
3. 在分发的 tx_port.h_ 和 _nx_port.h 文件中找到的 _tx_version_id 和 *nx_version_id 字符串的内容*。这两个字符串将为我们提供有关运行时间环境的重要信息。
4. RAM 中以下 ULONG 变量的内容：

   _tx_build_options

   _nx_system_build_options1

   _nx_system_build_options2

   _nx_system_build_options3

   _nx_system_build_options4

   _nx_system_build_options5

   这两个变量将为我们提供有关 Azure RTOS ThreadX 库和 Azure RTOS NetX 库的生成方式的信息。
5. 在检测到问题后立即捕获的跟踪缓冲区。可以通过使用 TX_ENABLE_EVENT_TRACE 生成 Azure RTOS ThreadX 库和 Azure RTOS NetX 库以及使用跟踪缓冲区信息调用 tx_trace_enable 来实现。有关详细信息，请参阅 Azure RTOS TraceX 用户指南。

第 1 章 - Azure RTOS NetX 简介

2021/4/29 •

Azure RTOS NetX 是 TCP/IP 标准的一种高性能实时实现，专用于基于 ThreadX 的嵌入式应用程序。本章提供了 NetX 的简介，并说明了其应用和优势。

NetX 的独特功能

与其他 TCP/IP 实现不同，NetX 的设计用途非常广泛，可轻松地从基于小型微控制器的应用程序扩展到使用功能强大的 RISC 和 DSP 处理器的应用程序。这与公共域或最初适用于工作站环境但随后被挤压为嵌入式设计的商业实现形成了鲜明对比。

Piconet™ 体系结构

NetX 的卓越可扩展性和性能的基础是 Piconet，这是一种专为嵌入式系统设计的软件体系结构。Piconet 体系结构通过将 NetX 服务实现为 C 库来最大程度地提高可扩展性。通过这种方式，只有应用程序实际使用的那些服务才会被引入最终运行时映像。因此，NetX 的实际大小完全由应用程序决定。对于大多数应用程序，NetX 的映像大小要求介于 5 KB 到 30 KB 之间。

仅当 NetX 绝对必需时，通过对内部组件函数调用分层可实现卓越的网络性能。此外，很多 NetX 处理都是以内联方式直接进行的，因而与过去的嵌入式设计中经常使用的工作站网络软件相比，其性能优势更为突出。

零复制实现

NetX 提供了 TCP/IP 的基于数据包的零复制实现。零复制意味着永远不会在 NetX 内复制应用程序的数据包缓冲区中的数据。这极大地提高了性能，并为应用程序释放了宝贵的处理器周期，这在嵌入式应用程序中极为重要。

UDP Fast Path™ 技术

利用 UDP Fast Path 技术，NetX 可以提供尽可能快的 UDP 处理。在发送端，UDP 处理（包括可选的 UDP 校验和）完全包含在 nx_udp_socket_send 服务中。在将数据包准备就绪可通过内部 NetX IP 发送例程之前，不会进行其他函数调用。此例程也是平面的（也就是说，其函数调用嵌套最少），因此可将数据包快速调度到应用程序的网络驱动程序中。接收 UDP 数据包时，NetX 数据包接收处理会将数据包直接置于适当的 UDP 套接字的接收队列上，或将数据包交给第一个挂起等待 UDP 套接字接收队列接收数据包的线程。无需进行其他 ThreadX 上下文切换。

ANSI C 源代码

NetX 完全以 ANSI C 写入，可立即移植到几乎任何具有 ANSI C 编译器和 ThreadX 支持的处理器体系结构。

不是黑盒

NetX 的大多数分发都包含完整的 C 源代码。这消除了许多商业网络堆栈所出现的“黑盒”问题。通过使用 NetX，应用程序开发人员可以准确地了解网络堆栈正在进行的操作，没有秘密！

如果有源代码，还允许进行特定于应用程序的修改。尽管不建议这样做，但如果需要，具备修改网络堆栈的能力当然是有益的。

对于习惯使用内部或公共域网络堆栈的开发人员而言，这些功能尤为贴心。他们期望拥有源代码和修改源代码的能力。NetX 是面向此类开发人员的终极网络软件。

与 BSD 兼容的套接字 API

对于旧版应用程序，NetX 提供了与 BSD 兼容的套接字接口，用于调用下面的高性能 NetX API。这有助于将现有的网络应用程序代码迁移到 NetX。

NetX 支持的 RFC

NetX 对描述基本网络协议的 RFC 的支持包括但不限于以下网络协议。NetX 遵循内存占用少且执行效率高的实时操作系统的限制中的所有常规建议和基本要求。

RFC	说明
RFC 1112	IP 多播的主机扩展 (IGMPv1)
RFC 1122	Internet 主机的要求 - 通信层
RFC 2236	Internet 组管理协议，版本 2
RFC 768	用户数据报协议 (UDP)
RFC 791	Internet 协议 (IP)
RFC 792	Internet 控制消息协议 (ICMP)
RFC 793	传输控制协议 (TCP)
RFC 826	以太网地址解析协议 (ARP)
RFC 903	反向地址解析协议 (RARP)

嵌入式网络应用程序

嵌入式网络应用程序是需要网络访问的应用程序，并在产品内隐藏的微处理器（如移动电话、通信设备、汽车引擎、激光打印机和医疗设备等）上执行这些应用程序。此类应用程序几乎总是具有一定的内存和性能限制。嵌入式网络应用程序的另一个不同之处在于，其软件和硬件具有专门的用途。

实时网络软件

大致说来，必须在确切的一段时间内执行其处理的网络软件被称为实时网络软件，当对网络应用程序施加时间约束时，它们会被分类为实时应用程序。由于嵌入式网络应用程序与外部世界的固有交互，因此嵌入式网络应用程序几乎总是实时的。

NetX 优势

将 NetX 用于嵌入式应用程序的主要优势在于高速 Internet 连接和非常小的内存需求。NetX 还与高性能、多任务 ThreadX 实时操作系统完全集成。

更高的响应能力

高性能 NetX 协议堆栈使嵌入式网络应用程序的响应速度比以往更快。对于具有大量网络流量或对单个数据包的严格处理要求的嵌入式应用程序而言，这一点尤其重要。

软件维护

使用 NetX，开发人员可以轻松地对其嵌入式应用程序的网络方面进行分区。这种分区让整个开发过程变得简单，并极大地加强了未来的软件维护。

更高的吞吐量

NetX 提供了最高性能的网络，这可以通过最少的数据包处理开销来实现。这还可以增加吞吐量。

处理器隔离

NetX 在应用程序、基础处理器和网络硬件之间提供可靠的独立于处理器的接口。这使开发人员能够专注于应用

程序的网络方面，而不是花费额外时间处理直接影响网络的硬件问题。

易用性

NetX 在设计时考虑到了应用程序开发人员。NetX 体系结构和服务调用接口易于理解。因此，NetX 开发人员可以快速使用其高级功能。

缩短上市时间

NetX 强大的功能加速了软件开发过程。NetX 提取了大多数的处理器和网络硬件问题，因而从大部分特定于应用程序网络的区域中消除了这些顾虑。这一点再加上易用性和高级功能集，缩短了上市时间。

保护软件投资

NetX 专门以 ANSI C 写入，与 ThreadX 实时操作系统完全集成。这意味着 NetX 应用程序可立即移植到所有 ThreadX 支持的处理器。更好的是，一个全新的处理器体系结构可以在几周内得到 ThreadX 的支持。因此，使用 NetX 可以确保应用程序的迁移路径并保护原始开发投资。

第 2 章 - 安装和使用 Azure RTOS NetX

2021/4/30 •

本章旨在介绍与安装、设置和使用高性能网络堆栈 Azure RTOS NetX 相关的各种问题。

主机注意事项

嵌入式开发通常在 Windows 或 Linux (Unix) 主机上进行。在主机上编译、链接应用程序并生成可执行文件之后，应用程序将下载到目标硬件，以便执行。

通常，目标下载是在开发工具的调试器内完成的。在下载后，调试器负责提供目标执行控件（“执行”、“暂停”、“断点”等）以及对内存和处理器寄存器的访问。

大多数开发工具调试器通过 JTAG (IEEE 1149.1) 和后台调试模式 (BDM) 等芯片调试 (OCD) 连接与目标硬件进行通信。调试器还通过线路内仿真 (ICE) 连接与目标硬件进行通信。OCD 和 ICE 连接提供可靠的解决方案，使对目标常驻软件的入侵最少。

对于主机上使用的资源，NetX 的源代码以 ASCII 格式提供，并需要大约 1 MB 的主机计算机硬盘空间。

目标注意事项

NetX 要求目标有 5 到 45 KB 只读内存 (ROM)。NetX 线程堆栈和其他全局数据结构要求目标另外有 1 到 5 KB 的随机存取内存 (RAM)。

此外，NetX 要求使用两个 ThreadX 计时器对象和一个 ThreadX mutex 对象。这些设施用于处理 NetX 协议堆栈中的定期处理需求和线程保护。

产品分发

可以从我们的公共源代码存储库获取 Azure RTOS NetX，网址为：https://github.com/azure-rtos/netx/。

下面列出了存储库中的几个重要文件：

- ***nx_api.h***：C 头文件，包含所有系统等式、数据结构和服务原型。
- ***nx_port.h***：C 头文件，包含所有开发工具及目标特定的数据定义和结构。
- ***demo_netx.c***：C 文件，包含小型演示应用程序。
- *nx.a（或 nx.lib）_：随标准包一起分发的 NetX C 库的二进制版本。|

安装 NetX

可以通过将 GitHub 存储库克隆到本地计算机来安装 NetX。下面是用于在电脑上创建 NetX 存储库的克隆的典型语法：

```
git clone https://github.com/azure-rtos/netx
```

或者，也可以使用 GitHub 主页上的“下载”按钮来下载存储库的副本。

还可以在联机存储库的首页上找到有关生成 NetX 库的说明。

IMPORTANT

应用程序软件需要访问 NetX 库文件(通常为 nx.a 或 nx.lib)和 C 包含文件 nx_api.h 和 nx_port.h。为实现此目的，可以设置开发工具的相应路径，或者将这些文件复制到应用程序开发区域。

使用 NetX

若要使用 NetX，应用程序代码必须在编译期间包括 nx_api.h，并且必须与 NetX 库 nx.a(或 nx.lib)链接。

下面是生成 NetX 应用程序所需的四个步骤：

1. 在所有使用 NetX 服务或数据结构的应用程序文件中包括 nx_api.h 文件。
2. 通过从 tx_application_define 函数或应用程序线程调用 nx_system_initialize 来初始化 NetX 系统。
3. 创建 IP 实例，在调用 nx_system_initialize 后启用地址解析协议 (ARP)(如有必要)以及任何套接字。
4. 编译应用程序源并与 NetX 运行时库 nx.a(或 nx.lib)链接。生成的映像可下载到目标并执行！

疑难解答

每个 NetX 端口都提供一个或多个在实际网络上或通过模拟网络驱动程序执行的示例。先运行示例系统始终是一个不错的主意。

如果示例系统无法正常运行，请执行以下操作来缩小问题范围：

1. 确定示例的运行量。
2. 增大任何新应用程序线程中的堆栈大小。
3. 使用配置选项部分中列出的合适调试选项重新编译 NetX 库。
4. 检查 NX_IP 结构以查看正在发送还是接收数据包。
5. 检查默认数据包池，查看是否存在可用的数据包。
6. 请确保网络驱动程序正在提供 ARP 和 IP 数据包并且其标头在 4 字节边界上(适用于需要 IP 连接的应用程序)。
7. 暂时跳过最近所做的任何更改，以查看问题是否消失或发生更改。此类信息对我们的支持工程师非常有用。

按照客户支持中心主题中概述的步骤，发送从故障排除步骤中收集的信息。

配置选项

使用 NetX 构建 NetX 库和应用程序时，有几个配置选项。除非另有说明，否则可在应用程序源、命令行或 nx_user.h 包含文件中配置选项。

IMPORTANT

只有当应用程序和 NetX 库在构建时定义了 NX_INCLUDE_USER_DEFINE_FILE，才会应用在 nx_user.h 中定义的选项。

以下部分列出了 NetX 中可用的配置选项。

系统配置选项

"	"
X_DEBUG	定义后，此选项用于启用 RAM 以太网网络驱动程序提供的可选打印调试信息。

“	“
NX_DISABLE_ERROR_CHECKING	定义后，此选项用于删除基本 NetX 错误检查 API 并提高性能。不受禁用错误检查影响的 API 返回代码将在 API 定义中以粗体字样列出。此定义通常在应用程序调试之后使用，使用它可提高性能并减少代码大小。
NX_DRIVER_DEFERRED_PROCESSING	定义后，此选项用于启用延迟网络驱动程序数据包处理。它允许网络驱动程序将数据包放置在 IP 实例上，并从 NetX 内部 IP 帮助程序线程中调用真实处理例程。
NX_ENABLE_EXTENDED_NOTIFY_SUPPORT	允许堆栈中有更多回调挂钩。BSD 包装层使用这些回调函数。默认不定义此选项。
NX_ENABLE_SOURCE_ADDRESS_CHECK	定义后，此选项允许检查传入数据包的源地址。默认已禁用此选项。
NX_LITTLE_ENDIAN	定义后，此选项用于对 little endian 环境执行必要的字节交换，以确保协议标头使用正确的 big endian 格式。请注意，默认值通常在 nx_port.h 中设置。
NX_MAX_PHYSICAL_INTERFACES	指定设备上物理网络接口的总数。默认值为 1，在 nx_api.h 中定义。设备必须至少有一个物理接口。请注意，此数量不包括环回接口。
NX_PHYSICAL_HEADER	指定帧的物理标头的大小（以字节为单位）。默认值为 16（基于与 32 位边界对齐的典型 14 字节以太网帧），在 nx_api.h 中定义。应用程序可以在包含 nx_api.h（例如 nx_user.h）之前定义该值，以替代默认值。

ARP 配置选项

“	“
NX_ARP_DEFEND_BY_REPLY	定义后，允许 NetX 通过发送 ARP 响应保护其 IP 地址。
NX_ARP_DEFEND_INTERVAL	定义 ARP 模块发出下一个防御数据包以响应指示地址冲突的传入 ARP 消息的时间间隔（以秒为单位）。
NX_ARP_DISABLE_AUTO_ARP_ENTRY	已重命名为 NX_DISABLE_ARP_AUTO_ENTRY。尽管此设计仍受支持，但建议使用新设计以使用 NX_DISABLE_ARP_AUTO_ENTRY。
NX_ARP_EXPIRATION_RATE	指定 ARP 条目保持有效的秒数。默认值为 0 时，将禁用 ARP 条目的过期或老化，在 nx_api.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。
NX_ARP_MAC_CHANGE_NOTIFICATION_ENABLE	已重命名为 NX_ENABLE_ARP_MAC_CHANGE_NOTIFICATION。尽管此设计仍受支持，但建议使用新设计以使用 NX_ENABLE_ARP_MAC_CHANGE_NOTIFICATION。
NX_ARP_MAX_QUEUE_DEPTH	指定在等待 ARP 响应时可以排队的数据包的最大数量。默认值为 4，在 nx_api.h 中定义。

NX_ARP_MAXIMUM_RETRIES	指定在没有 ARP 响应的情况下进行 ARP 重试的最大次数。默认值为 18，在 nx_api.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。
NX_ARP_UPDATE_RATE	指定 ARP 重试之间的秒数。默认值为 10（表示 10 秒），在 nx_api.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。
NX_DISABLE_ARP_AUTO_ENTRY	定义后，此选项用于禁止在 ARP 缓存中输入 ARP 请求信息。
NX_DISABLE_ARP_INFO	定义后，此选项用于禁用 ARP 信息收集。
NX_ENABLE_ARP_MAC_CHANGE_NOTIFICATION	定义后，允许 ARP 调用回调通知函数来检测 MAC 地址是否已更新。

ICMP 配置选项

NX_DISABLE_ICMP_INFO	定义后，此选项用于禁用 ICMP 信息收集。
NX_DISABLE_ICMP_RX_CHECKSUM	禁用接收到的 ICMP 数据包上的 ICMP 校验和计算。当网络接口驱动程序能够验证 ICMP 校验和，且应用程序不使用 IP 拆分功能时，此选项很有用。默认不定义此选项。
NX_DISABLE_ICMP_TX_CHECKSUM	禁用传输的 ICMP 数据包上的 ICMP 校验和计算。当网络接口驱动程序能够计算 ICMP 校验和，且应用程序不使用 IP 拆分功能时，此选项很有用。默认不定义此选项。

IGMP 配置选项

NX_DISABLE_IGMP_INFO	定义后，此选项用于禁用 IGMP 信息收集。
NX_DISABLE_IGMPV2	定义后，此选项用于禁用 IGMPv2 支持，NetX 仅支持 IGMPv1。默认情况下，此选项未设置，且在 nx_api.h 中定义。
NX_MAX_MULTICAST_GROUPS	指定可以联接的多播组的最大数量。默认值为 7，在 nx_api.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。

IP 配置选项

NX_DISABLE_FRAGMENTATION	定义后，此选项用于禁用 IP 拆分和重组逻辑。
NX_DISABLE_IP_INFO	定义后，此选项用于禁用 IP 信息收集。
NX_DISABLE_IP_RX_CHECKSUM	定义后，此选项用于禁用接收到的 IP 数据包上的校验和逻辑。当网络设备能够验证 IP 头校验和，且应用程序不使用 IP 拆分功能时，此选项很有用。

NX_DISABLE_IP_TX_CHECKSUM	定义后，此选项用于禁用发送的 IP 数据包上的校验和逻辑。当基础网络设备能够生成 IP 标头校验和，且应用程序不使用 IP 拆分功能时，此选项很有用。
NX_DISABLE_LOOPBACK_INTERFACE	定义后，此选项用于禁用 NetX 对环回接口的支持。
NX_DISABLE_RX_SIZE_CHECKING	定义后，此选项用于禁用对接收到的数据包的大小检查。
NX_ENABLE_IP_STATIC_ROUTING	定义后，此选项用于启用 IP 静态路由，可在其中向目标地址分配特定的下一跃点地址。默认情况下，IP 静态路由处于禁用状态。
NX_IP_PERIODIC_RATE	定义后，此选项用于指定 ThreadX 计时器时钟周期的数量（以秒为单位）。默认值派生自 ThreadX 符号 TX_TIMER_TICKS_PER_SECOND，默认情况下，此值设置为 100（10ms 计时器）。应用程序应谨慎修改此值，因为其他 NetX 模块将从 NX_IP_PERIODIC_RATE 派生计时信息。
NX_IP_ROUTING_TABLE_SIZE	定义后，此选项用于设置 IP 静态路由表中的最大条目数，这是传出接口以及
给定目标地址的下一个跃点地址的列表。默认值为 8，在 nx_api.h. 中定义。此符号仅在定义 NX_ENABLE_IP_STATIC_ROUTING 时使用。	

数据包配置选项

NX_DISABLE_PACKET_INFO	定义后，此选项用于禁用数据包池信息收集。
NX_PACKET_HEADER_PAD	定义后，此选项允许在 NX_PACKET 控制块的末尾处进行填充。要填充的 ULONG 字词的数量由 NX_PACKET_HEADER_PAD_SIZE 定义 。
NX_PACKET_HEADER_PAD_SIZE	设置要填充到 NX_PACKET 结构中的 ULONG 字词的数量，允许数据包有效负载区从所需的
对齐位置开始。当缓冲区描述符点直接传入 NX_PACKET 有效负载区，且网络接口接收逻辑或缓存操作逻辑要求缓冲区起始地址满足特定的对齐要求时，此功能很有用。仅当定义了 NX_PACKET_HEADER_PAD 时，此值才有效。	

RARP 配置选项

NX_DISABLE_RARP_INFO	定义后，此选项用于禁用 RARP 信息收集。

TCP 配置选项

“	“
NX_DISABLE_RESET_DISCONNECT	定义后，当提供的超时值指定为 NX_NO_WAIT 时，将禁用断开连接期间的重置处理。
NX_DISABLE_TCP_INFO	定义后，此选项用于禁用 TCP 信息收集。
NX_DISABLE_TCP_RX_CHECKSUM	定义后，此选项用于禁用接收到的 TCP 数据包上的校验和逻辑。仅当链路层具有可靠的校验和或 CRC 处理，或者接口驱动程序能够验证硬件中的 TCP 校验和时，此选项才很有用。
NX_DISABLE_TCP_TX_CHECKSUM	定义后，此选项用于禁用用于发送 TCP 数据包的校验和逻辑。仅当接收网络节点接收到禁用了 TCP 校验和逻辑，或者基础网络驱动程序能够生成 TCP 校验和时，此选项才很有用。
NX_ENABLE_TCP_KEEPALIVE	定义后，此选项用于启用可选的 TCP KeepAlive 计时器。默认设置未启用。
NX_ENABLE_TCP_MSS_CHECKING	定义后，可以在接受 TCP 连接之前验证对等 MSS 的最小值。若要使用此功能，必须定义符号 NX_ENABLE_TCP_MSS_MINIMUM。默认情况下不启用此选项。
NX_ENABLE_TCP_WINDOW_SCALING	启用 TCP 应用程序的窗口缩放选项。定义后，在 TCP 连接阶段会协商窗口缩放选项，且应用程序可以指定大于 64 K 的窗口大小。未启用默认设置（未定义）。
NX_MAX_LISTEN_REQUESTS	指定服务器侦听请求的最大数量。默认值为 10，在 nx_api.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。
NX_TCP_ACK_EVERY_N_PACKETS	指定发送 ACK 之前要接收的 TCP 数据包的数量。注意，如果启用了 NX_TCP_IMMEDIATE_ACK，但未启用 NX_TCP_ACK_EVERY_N_PACKETS，则此值将自动设置为 1，以实现向后兼容性 。
NX_TCP_ACK_TIMER_RATE	指定如何划分系统时钟周期 (NX_IP_PERIODIC_RATE) 的数量，以计算 TCP 延迟 ACK 处理的计时器速率。默认值为 5（表示 200ms），在 nx_tcp.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。
NX_TCP_ENABLE_KEEPALIVE	已重命名为 NX_ENABLE_TCP_KEEPALIVE。尽管此设计仍受支持，但建议使用新设计以使用 NX_ENABLE_TCP_KEEPALIVE。
NX_TCP_ENABLE_WINDOW_SCALING	已重命名为 NX_ENABLE_TCP_WINDOW_SCALING。尽管此设计仍受支持，但建议使用新设计以使用 NX_ENABLE_TCP_WINDOW_SCALING。
NX_TCP_FAST_TIMER_RATE	指定如何划分 NetX 的内部时钟周期 (NX_IP_PERIODIC_RATE) 的数量，以计算快速 TCP 计时器速率。快速 TCP 计时器用于驱动各种 TCP 计时器，包括延迟 ACK 计时器。默认值为 10，表示 100ms（假设 ThreadX 计时器在 10ms 时运行）。此值在 nx_tcp.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。

NX_TCP_IMMEDIATE_ACK	定义后，此选项用于启用可选的 TCP 即时 ACK 响应处理。定义此符号等效于将 NX_TCP_ACK_EVERY_N_PACKETS 定义为 1。
NX_TCP_KEEPALIVE_INITIAL	指定在 KeepAlive 计时器激活之前处于非活动状态的秒数。默认值为 7200（表示 2 小时），在 nx_tcp.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。
NX_TCP_KEEPALIVE_RETRIES	指定在连接被视为中断之前允许 KeepAlive 重试的次数。默认值为 10（表示 10 次重试），在 nx_tcp.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。
NX_TCP_KEEPALIVE_RETRY	指定在连接的另一方没有响应时，KeepAlive 计时器重试之间间隔的秒数。默认值为 75（表示重试之间间隔 75 秒），在 nx_tcp.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。
NX_TCP_MAX_OUT_OF_ORDER_PACKETS	用于定义可在 TCP 套接字接收队列中保留的无序 TCP 数据包最大数量的符号。此符号可用于限制在 TCP 接收套接字中排队的数据包数量，防止数据包池耗尽。默认情况下未定义此符号，因此在 TCP 套接字中排队的无序数据包数没有限制。
NX_TCP_MAXIMUM_RETRIES	指定在连接被视为中断之前允许的数据传输重试次数。默认值为 10（表示 10 次重试），在 nx_tcp.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。
NX_TCP_MAXIMUM_TX_QUEUE	指定 TCP 发送请求被挂起或拒绝之前 TCP 传输队列的最大深度。默认值为 20，这表示在任意给定时间，传输队列中最多可以有 20 个数据包。请注意，数据包将保留在传输队列中，直到从连接的另一端接收到涵盖部分或全部数据包数据的 ACK。此常量在 nx_tcp.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。
X_TCP_MSS_CHECKING_ENABLED	已重命名为 NX_ENABLE_TCP_MSS_CHECKING。尽管此设计仍受支持，但建议使用新设计以使用 NX_ENABLE_TCP_MSS_CHECKING。
NX_TCP_MSS_MINIMUM	定义 NetX TCP 模块接受的最小 MSS 值的符号。此功能由 NX_ENABLE_TCP_MSS_CHECK 启用
NX_TCP_RETRY_SHIFT	指定重试之间的重新传输超时期限如何变化。如果此值为 0，则初始重新传输超时与后续重新传输超时相同。如果此值为 1，则每个连续重新传输时长为前一个的 2 倍。如果此值为 2，则每个后续的重新传输超时时长为前一个的 4 倍。默认值为 0，在 nx_tcp.h 中定义。应用程序可以在包含 nx_api.h 之前定义该值，以替代默认值。
NX_TCP_TRANSMIT_TIMER_RATE	指定如何划分系统时钟周期 (NX_IP_PERIODIC_RATE) 的数量，以计算 TCP 传输重试处理的计时器速率。默认值为 1，表示 1 秒，在 nx_tcp.h 中定义 ********** *nx_api.h* ******, ******。

UDP 配置选项

NX_DISABLE_UDP_INFO	定义后，此选项用于禁用 UDP 信息收集。

NX_DISABLE_UDP_RX_CHECKSUM	定义后，此选项用于禁用传入 UDP 数据包上的 UDP 校验和计算。当网络接口驱动程序能够验证硬件中的 UDP 标头校验和，且应用程序不启用 IP 拆分逻辑时，此选项很有用。
NX_DISABLE_UDP_TX_CHECKSUM	定义后，此选项用于禁用传出 UDP 数据包上的 UDP 校验和计算。当网络接口驱动程序能够计算 UDP 标头校验和，并在传输数据之前将值插入 IP 标头，且应用程序不启用 IP 拆分逻辑时，此选项很有用。

NetX 版本 ID

当前版本的 NetX 在运行时可供用户和应用程序软件使用。程序员可以从 nx_port.h 文件获取 NetX 版本。此外，该文件还包含相应端口的版本历史记录。应用程序软件可以通过在 nx_port.h 中检查全局字符串 _nx_version_id 来获取 NetX 版本 。

应用程序软件还可以从以下常量中获取版本信息，这些常量在 nx_api.h 中定义。

这些常量通过名称和产品的主要版本和次要版本来确定产品的当前版本。

```
#define EL_PRODUCT_NETX

#define NETX_MAJOR_VERSION

#define NETX_MINOR_VERSION
```

第 4 章 - Azure RTOS NetX 服务说明

2021/4/29 •

本章按字母顺序介绍所有 Azure RTOS NetX 服务。服务名称设计为将所有相似的服务组合在一起。例如，所有的 ARP 服务都可以在本章开头找到。

> **NOTE**
> 请注意，无法充分利用高性能 NetX API 的旧式应用程序代码可以使用与 BSD 兼容的套接字 API。有关与 BSD 兼容的套接字 API 的详细信息，请参阅“附录 D”。

在每一项说明的“返回值”部分中，以粗体显示的值不受用于禁用 API 错误检查的 NX_DISABLE_ERROR_CHECKING 选项影响，而不以粗体显示的值则会被完全禁用。“允许调用自”部分指示可从中调用每项 NetX 服务的来源。

nx_arp_dynamic_entries_invalidate

使 ARP 缓存中的所有动态条目失效

原型

```
UINT nx_arp_dynamic_entries_invalidate(NX_IP *ip_ptr);
```

说明

此服务可使 ARP 缓存中当前包含的所有动态 ARP 条目失效。

参数

- ip_ptr 先前创建的 IP 实例的指针。

返回值

- NX_SUCCESS (0x00) 使 ARP 缓存失效成功。
- NX_NOT_ENABLED (0x14) ARP 未启用。
- NX_PTR_ERROR (0x07) IP 地址无效。
- NX_CALLER_ERROR (0x11) 调用方不是线程。

允许调用自

线程数

可以抢占

否

示例

```
/* Invalidate all dynamic entries in the ARP cache. */
status = nx_arp_dynamic_entries_invalidate(&ip_0);
/* If status is NX_SUCCESS the dynamic ARP entries were successfully invalidated. */
```

另请参阅

- nx_arp_dynamic_entry_set、nx_arp_enable、nx_arp_gratuitous_send,

- nx_arp_hardware_address_find、nx_arp_info_get,
- nx_arp_ip_address_find、nx_arp_static_entries_delete,
- nx_arp_static_entry_create、nx_arp_static_entry_delete

nx_arp_dynamic_entry_set

设置动态 ARP 条目

原型

```
UINT nx_arp_dynamic_entry_set(
    NX_IP *ip_ptr,
    ULONG ip_address,
    ULONG physical_msw,
    ULONG physical_lsw);
```

说明

此服务可从 ARP 缓存分配动态条目，并将指定的 IP 设置为物理地址映射。如果指定了零物理地址，则会将实际的 ARP 请求发送到网络，以便解析该物理地址。另请注意，如果已激活 ARP 老化，或者 ARP 缓存已用尽并且这是最近最少使用的 ARP 条目，则会删除此条目。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **ip_address** 要映射的 IP 地址。
- **physical_msw** 物理地址的高 16 位 (47-32)。
- **physical_lsw** 物理地址的低 32 位 (31-0)。

返回值

- **NX_SUCCESS** (0x00) 成功设置 ARP 动态条目。
- **NX_NO_MORE_ENTRIES** (0x17) 在 ARP 缓存中没有更多的 ARP 条目可供使用。
- **NX_IP_ADDRESS_ERROR** (0x21) IP 地址无效。
- **NX_PTR_ERROR** (0x07) IP 实例指针无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Setup a dynamic ARP entry on the previously created IP Instance 0. */
status = nx_arp_dynamic_entry_set(&ip_0, IP_ADDRESS(1,2,3,4),
    0x1022, 0x1234);
/* If status is NX_SUCCESS, there is now a dynamic mapping between
    the IP address of 1.2.3.4 and the physical hardware address of
    10:22:00:00:12:34. */
```

另请参阅

- nx_arp_dynamic_entries_invalidate、nx_arp_enable,
- nx_arp_gratuitous_send、nx_arp_hardware_address_find,
- nx_arp_info_get、nx_arp_ip_address_find、nx_arp_static_entries_delete,

- nx_arp_static_entry_create、nx_arp_static_entry_delete

nx_arp_enable

启用地址解析协议 (ARP)。

原型

```
UINT nx_arp_enable(
    NX_IP *ip_ptr,
    VOID *arp_cache_memory,
    ULONG arp_cache_size);
```

说明

此服务可为特定的 IP 实例初始化 NetX 的 ARP 组件。ARP 初始化包括设置 ARP 缓存，以及设置发送和接收 ARP 消息所需的各种 ARP 处理例程。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **arp_cache_memory** 指向要放置 ARP 缓存的内存区域的指针。
- **arp_cache_size** 每个 ARP 条目为 52 个字节，因此 ARP 条目总数为大小除以 52。

返回值

- **NX_SUCCESS** (0x00) 成功启用 ARP。
- **NX_PTR_ERROR** (0x07) IP 或缓存内存指针无效。
- **NX_SIZE_ERROR** (0x09) 用户提供的 ARP 缓存内存太小。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_ALREADY_ENABLED** (0x15) 已启用此组件。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Enable ARP and supply 1024 bytes of ARP cache memory for
    previously created IP Instance ip_0. */
status = nx_arp_enable(&ip_0, (void *) pointer, 1024);
/* If status is NX_SUCCESS, ARP was successfully enabled for this IP instance.*/
```

另请参阅

- nx_arp_dynamic_entries_invalidate、nx_arp_dynamic_entry_set,
- nx_arp_gratuitous_send、nx_arp_hardware_address_find,
- nx_arp_info_get、nx_arp_ip_address_find、nx_arp_static_entries_delete,
- nx_arp_static_entry_create、nx_arp_static_entry_delete

nx_arp_gratuitous_send

发送免费 ARP 请求

原型

```
UINT nx_arp_gratuitous_send(
    NX_IP *ip_ptr,
    VOID (*response_handler) (NX_IP *ip_ptr, NX_PACKET *packet_ptr));
```

说明

只要接口 IP 地址有效，此服务就会通过所有物理接口传输免费 ARP 请求。如果随后收到 ARP 响应，则会调用提供的响应处理程序处理对该免费 ARP 的响应。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **response_handler** 指向响应处理函数的指针。如果提供了 NX_NULL，则会忽略响应。

返回值

- **NX_SUCCESS** (0x00) 成功发送免费 ARP。
- **NX_NO_PACKET** (0x01) 没有可用的数据包。
- **NX_NOT_ENABLED** (0x14) ARP 未启用。
- **NX_IP_ADDRESS_ERROR** (0x21) 当前 IP 地址无效。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 调用方不是线程。

允许调用自

线程数

可以抢占

否

示例

```
/* Send gratuitous ARP without any response handler. */
status = nx_arp_gratuitous_send(&ip_0, NX_NULL);

/* If status is NX_SUCCESS the gratuitous ARP was successfully sent. */
```

另请参阅

- nx_arp_dynamic_entries_invalidate、nx_arp_dynamic_entry_set,
- nx_arp_enable、nx_arp_hardware_address_find、nx_arp_info_get,
- nx_arp_ip_address_find、nx_arp_static_entries_delete,
- nx_arp_static_entry_create、nx_arp_static_entry_delete

nx_arp_hardware_address_find

根据给定的 IP 地址查找物理硬件地址

原型

```
UINT nx_arp_hardware_address_find(
    NX_IP *ip_ptr,
    ULONG ip_address,
    ULONG *physical_msw,
    ULONG *physical_lsw);
```

说明

此服务尝试在 ARP 缓存中查找与提供的 IP 地址相关联的物理硬件地址。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **ip_address** 要搜索的 IP 地址。
- **physical_msw** 指向变量的指针，该变量用于返回物理地址的高 16 位 (47-32)。
- **physical_lsw** 指向变量的指针，该变量用于返回物理地址的低 32 位 (31-0)。

返回值

- **NX_SUCCESS** (0x00) 成功找到 ARP 硬件地址。
- **NX_ENTRY_NOT_FOUND** (0x16) 在 ARP 缓存中找不到映射。
- **NX_IP_ADDRESS_ERROR** (0x21) IP 地址无效。
- **NX_PTR_ERROR** (0x07) IP 指针或内存指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
/* Search for the hardware address associated with the IP address of
    1.2.3.4 in the ARP cache of the previously created IP
    Instance 0. */
status = nx_arp_hardware_address_find(
    &ip_0,
    IP_ADDRESS(1,2,3,4),
    &physical_msw,
    &physical_lsw);

/* If status is NX_SUCCESS, the variables physical_msw and
    physical_lsw contain the hardware address.*/
```

另请参阅

- nx_arp_dynamic_entries_invalidate、nx_arp_dynamic_entry_set,
- nx_arp_enable、nx_arp_gratuitous_send、nx_arp_info_get,
- nx_arp_ip_address_find、nx_arp_static_entries_delete,
- nx_arp_static_entry_create、nx_arp_static_entry_delete

nx_arp_info_get

检索 ARP 活动的相关信息

原型

```
UINT nx_arp_info_get(
    NX_IP *ip_ptr,
    ULONG *arp_requests_sent,
    ULONG *arp_requests_received,
    ULONG *arp_responses_sent,
    ULONG *arp_responses_received,
    ULONG *arp_dynamic_entries,
    ULONG *arp_static_entries,
    ULONG *arp_aged_entries,
    ULONG *arp_invalid_messages);
```

说明

此服务可检索相关联的 IP 实例的 ARP 活动相关信息。

如果目标指针为 NX_NULL，则不会将该特定信息返回给调用方。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **arp_requests_sent** 此指针指向从此 IP 实例发送的全部 ARP 请求的目标。
- **arp_requests_received** 此指针指向从网络收到的全部 ARP 请求的目标。
- **arp_responses_sent** 此指针指向从此 IP 实例发送的全部 ARP 响应的目标。
- **arp_responses_received** 此指针指向从网络收到的全部 ARP 响应的目标。
- **arp_dynamic_entries** 此指针指向当前数量的动态 ARP 条目的目标。
- **arp_static_entries** 此指针指向当前数量的静态 ARP 条目的目标。
- **arp_aged_entries** 此指针指向已老化且已失效的 ARP 条目总数的目标。
- **arp_invalid_messages** 此指针指向已收到但无效 ARP 消息总数的目标。

返回值

- **NX_SUCCESS** (0x00) 成功检索 ARP 信息。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
/* Pickup ARP information for ip_0. */
status = nx_arp_info_get(
    &ip_0,
    &arp_requests_sent,
    &arp_requests_received,
    &arp_responses_sent,
    &arp_responses_received,
    &arp_dynamic_entries,
    &arp_static_entries,
    &arp_aged_entries,
    &arp_invalid_messages);
/* If status is NX_SUCCESS, the ARP information has been stored in
    the supplied variables. */
```

另请参阅

- nx_arp_dynamic_entries_invalidate、nx_arp_dynamic_entry_set,
- nx_arp_enable、nx_arp_gratuitous_send,
- nx_arp_hardware_address_find、nx_arp_ip_address_find,
- nx_arp_static_entries_delete、nx_arp_static_entry_create,
- nx_arp_static_entry_delete

nx_arp_ip_address_find

根据给定的物理地址查找 IP 地址

原型

```
UINT nx_arp_ip_address_find(
    NX_IP *ip_ptr,
    ULONG *ip_address,
    ULONG physical_msw,
    ULONG physical_lsw);
```

说明

此服务尝试在 ARP 缓存中查找与提供的物理地址相关联的 IP 地址。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **ip_address** 指向返回的 IP 地址的指针（如果找到已映射的 IP 地址）。
- **physical_msw** 要搜索的物理地址的高 16 位 (47-32)。
- **physical_lsw** 要搜索的物理地址的低 32 位 (31-0)。

返回值

- **NX_SUCCESS** (0x00) 查找 ARP IP 地址成功
- **NX_ENTRY_NOT_FOUND** (0x16) 在 ARP 缓存中找不到映射。
- **NX_PTR_ERROR** (0x07) IP 指针或内存指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。
- **NX_INVALID_PARAMETERS** (0x4D) physical_msw 和 physical_lsw 均为 0。

允许调用自

线程数

可以抢占

否

示例

```
/* Search for the IP address associated with the hardware address of
    0x0:0x01234 in the ARP cache of the previously created IP
    Instance ip_0. */
status = nx_arp_ip_address_find(&ip_0, &ip_address, 0x0, 0x1234);

/* If status is NX_SUCCESS, the variables ip_address contains the
    associated IP address.*/
```

另请参阅

- nx_arp_dynamic_entries_invalidate、nx_arp_dynamic_entry_set,
- nx_arp_enable、nx_arp_gratuitous_send,
- nx_arp_hardware_address_find、nx_arp_info_get,
- nx_arp_static_entries_delete、nx_arp_static_entry_create,
- nx_arp_static_entry_delete

nx_arp_static_entries_delete

删除所有静态 ARP 条目

原型

```
UINT nx_arp_static_entries_delete(NX_IP *ip_ptr);
```

说明

此服务可删除 ARP 缓存中的所有静态条目。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 静态条目已删除。
- **NX_PTR_ERROR** (0x07) ip_ptr 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Delete all the static ARP entries for IP Instance 0, assuming
    "ip_0" is the NX_IP structure for IP Instance 0. */

status = nx_arp_static_entries_delete(&ip_0);

/* If status is NX_SUCCESS all static ARP entries in the ARP cache
have been deleted. */
```

另请参阅

- nx_arp_dynamic_entries_invalidate、nx_arp_dynamic_entry_set,
- nx_arp_enable、nx_arp_gratuitous_send,
- nx_arp_hardware_address_find、nx_arp_info_get,
- nx_arp_ip_address_find、nx_arp_static_entry_create,
- nx_arp_static_entry_delete

nx_arp_static_entry_create

在 ARP 缓存中创建静态 IP 到硬件的映射

原型

```
UINT nx_arp_static_entry_create(
    NX_IP *ip_ptr,
    ULONG ip_address,
    ULONG physical_msw,
    ULONG physical_lsw);
```

说明

此服务针对指定的 IP 实例，在 ARP 缓存中创建静态 IP 到物理地址映射。静态 ARP 条目不受 ARP 定期更新的影响。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **ip_address** 要映射的 IP 地址。
- **physical_msw** 要映射的物理地址的高 16 位 (47-32)。
- **physical_lsw** 要映射的物理地址的低 32 位 (31-0)。

返回值

- **NX_SUCCESS** (0x00) 成功创建 ARP 静态条目。
- **NX_NO_MORE_ENTRIES** (0x17) 在 ARP 缓存中没有更多的 ARP 条目可供使用。
- **NX_IP_ADDRESS_ERROR** (0x21) IP 地址无效。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。
- **NX_INVALID_PARAMETERS** (0x4D) physical_msw 和 physical_lsw 均为 0。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Create a static ARP entry on the previously created IP
    Instance 0. */

status = nx_arp_static_entry_create(&ip_0, IP_ADDRESS(1,2,3,4),
    0x0, 0x1234);

/* If status is NX_SUCCESS, there is now a static mapping between
    the IP address of 1.2.3.4 and the physical hardware address of
    00:00:00:00:12:34. */
```

另请参阅

- nx_arp_dynamic_entries_invalidate、nx_arp_dynamic_entry_set,
- nx_arp_enable、nx_arp_gratuitous_send,
- nx_arp_hardware_address_find、nx_arp_info_get,
- nx_arp_ip_address_find、nx_arp_static_entries_delete,
- nx_arp_static_entry_delete

nx_arp_static_entry_delete

从 ARP 缓存中删除静态 IP 到硬件的映射

原型

```
UINT nx_arp_static_entry_delete(
    NX_IP *ip_ptr,
    ULONG ip_address,
    ULONG physical_msw,
    ULONG physical_lsw);
```

说明

此服务针对指定的 IP 实例，在 ARP 缓存中查找并删除先前创建的静态 IP 到物理地址映射。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **ip_address** 以静态方式映射的 IP 地址。
- **physical_msw** 以静态方式映射的物理地址的高 16 位 (47 - 32)。
- **physical_lsw** 以静态方式映射的物理地址的低 32 位 (31 - 0)。

返回值

- **NX_SUCCESS** (0x00) 成功删除 ARP 静态条目。
- **NX_ENTRY_NOT_FOUND** (0x16) 在 ARP 缓存中找不到静态 ARP 条目。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。
- **NX_IP_ADDRESS_ERROR** (0x21) IP 地址无效。
- **NX_INVALID_PARAMETERS** (0x4D) physical_msw 和 physical_lsw 均为 0。

允许调用自

线程数

可以抢占

否

示例

```
/* Delete a static ARP entry on the previously created IP
    instance ip_0. */
status = nx_arp_static_entry_delete(&ip_0, IP_ADDRESS(1,2,3,4),
    0x0, 0x1234);
/* If status is NX_SUCCESS, the previously created static ARP entry
    was successfully deleted. */
```

另请参阅

- nx_arp_dynamic_entries_invalidate、nx_arp_dynamic_entry_set,
- nx_arp_enable、nx_arp_gratuitous_send,
- nx_arp_hardware_address_find、nx_arp_info_get,
- nx_arp_ip_address_find、nx_arp_static_entries_delete,
- nx_arp_static_entry_create

nx_icmp_enable

启用 Internet 控制消息协议 (ICMP)

原型

```
UINT nx_icmp_enable(NX_IP *ip_ptr);
```

说明

此服务针对指定的 IP 实例启用 ICMP 组件。ICMP 组件负责处理 Internet 错误消息以及 ping 请求和回复。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功启用 ICMP。
- **NX_ALREADY_ENABLED** (0x15) 已启用 ICMP。

- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Enable ICMP on the previously created IP Instance ip_0. */
status = nx_icmp_enable(&ip_0);

/* If status is NX_SUCCESS, ICMP is enabled. */
```

另请参阅

- nx_icmp_info_get、nx_icmp_ping

nx_icmp_info_get

检索 ICMP 活动的相关信息

原型

```
UINT nx_icmp_info_get(
    NX_IP *ip_ptr,
    ULONG *pings_sent,
    ULONG *ping_timeouts,
    ULONG *ping_threads_suspended,
    ULONG *ping_responses_received,
    ULONG *icmp_checksum_errors,
    ULONG *icmp_unhandled_messages);
```

说明

此服务针对指定的 IP 实例检索 ICMP 活动相关信息。

如果目标指针为 NX_NULL，则不会将该特定信息返回给调用方。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **pings_sent** 此指针指向已发送的 ping 总数的目标。
- **ping_timeouts** 此指针指向 ping 超时总数的目标。
- **ping_threads_suspended** 此指针指向在 ping 请求上挂起的线程总数的目标。
- **ping_responses_received** 此指针指向已收到的 ping 响应总数的目标。
- **icmp_checksum_errors** 此指针指向 ICMP 校验和错误总数的目标。
- **icmp_unhandled_messages** 此指针指向未处理的 ICMP 消息总数的目标。

返回值

- **NX_SUCCESS** (0x00) 成功检索 ICMP 信息。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Retrieve ICMP information from previously created IP
    instance ip_0. */
status = nx_icmp_info_get(
    &ip_0,
    &pings_sent,
    &ping_timeouts,
    &ping_threads_suspended,
    &ping_responses_received,
    &icmp_checksum_errors,
    &icmp_unhandled_messages);

/* If status is NX_SUCCESS, ICMP information was retrieved. */
```

另请参阅

- nx_icmp_enable、nx_icmp_ping

nx_icmp_ping

向指定的 IP 地址发送 ping 请求

原型

```
UINT nx_icmp_ping(
    NX_IP *ip_ptr,
    ULONG ip_address,
    CHAR *data, ULONG data_size,
    NX_PACKET **response_ptr,
    ULONG wait_option);
```

说明

此服务向指定的 IP 地址发送 ping 请求，并按指定的时间长度等待 ping 响应消息。如果未收到响应，则会返回错误。否则，将在 response_ptr 所指向的变量中返回完整的响应消息。

如果返回了 NX_SUCCESS，则应用程序负责在不再需要所收到的数据包后将其释放。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **ip_address** 要 ping 的 IP 地址，以主机字节顺序表示。data：指向 ping 消息数据区域的指针。
- **data_size** ping 数据中的字节数
- **response_ptr** 此指针指向将在其中返回 ping 响应消息的数据包指针。
- **wait_option** 定义等待 ping 响应的时长。等待选项的定义如下：
  - NX_NO_WAIT (0x00000000)
  - NX_WAIT_FOREVER (0xFFFFFFFF)
  - 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) ping 成功。响应消息指针已放入 response_ptr 所指向的变量中。

- **NX_NO_PACKET** (0x01) 无法分配 ping 请求数据包。
- **NX_OVERFLOW** (0x03) 指定的数据区域超过此 IP 实例的默认数据包大小。
- **NX_NO_RESPONSE** (0x29) 请求的 IP 未响应。
- **NX_WAIT_ABORTED** (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。
- **NX_IP_ADDRESS_ERROR** (0x21) IP 地址无效。
- **NX_PTR_ERROR** (0x07) IP 指针或响应指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
/* Issue a ping to IP address 1.2.3.5 from the previously created IP
    Instance ip_0. */
status = nx_icmp_ping(&ip_0, IP_ADDRESS(1,2,3,5), "abcd", 4,
    &response_ptr, 10);

/* If status is NX_SUCCESS, a ping response was received from IP
    address 1.2.3.5 and the response packet is contained in the
    packet pointed to by response_ptr. It should have the same "abcd"
    four bytes of data. */
```

另请参阅

- nx_icmp_enable、nx_icmp_info_get

nx_igmp_enable

启用 Internet 组管理协议 (IGMP)

原型

```
UINT nx_igmp_enable(NX_IP *ip_ptr);
```

说明

此服务可在指定的 IP 实例上启用 IGMP 组件。IGMP 组件负责对 IP 多播组管理操作提供支持。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功启用 IGMP。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_ALREADY_ENABLED** (0x15) 已启用此组件。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Enable IGMP on the previously created IP Instance ip_0. */
status = nx_igmp_enable(&ip_0);

/* If status is NX_SUCCESS, IGMP is enabled. */
```

另请参阅

- nx_igmp_info_get、nx_igmp_loopback_disable,
- nx_igmp_loopback_enable、nx_igmp_multicast_interface_join,
- nx_igmp_multicast_join、nx_igmp_multicast_leave

nx_igmp_info_get

检索 IGMP 活动的相关信息

原型

```
UINT nx_igmp_info_get(
    NX_IP *ip_ptr,
    ULONG *igmp_reports_sent,
    ULONG *igmp_queries_received,
    ULONG *igmp_checksum_errors,
    ULONG *current_groups_joined);
```

说明

此服务检索指定的 IP 实例的 IGMP 活动相关信息。

如果目标指针为 NX_NULL，则不会将该特定信息返回给调用方。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **igmp_reports_sent** 此指针指向已发送的 ICMP 报告总数的目标。
- **igmp_queries_received** 此指针指向多播路由器收到的查询总数的目标。
- **igmp_checksum_errors** 此指针指向接收数据包中 IGMP 校验和错误总数的目标。
- **current_groups_joined** 此指针指向通过此 IP 实例加入的当前数量的组的目标。

返回值

- **NX_SUCCESS** (0x00) 成功检索 IGMP 信息。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Retrieve IGMP information
    from previously created IP Instance ip_0. */
status = nx_igmp_info_get(
    &ip_0,
    &igmp_reports_sent,
    &igmp_queries_received,
    &igmp_checksum_errors,
    &current_groups_joined);
/* If status is NX_SUCCESS, IGMP information was retrieved. */
```

另请参阅

- nx_igmp_enable、nx_igmp_loopback_disable,
- nx_igmp_loopback_enable、nx_igmp_multicast_interface_join,
- nx_igmp_multicast_join、nx_igmp_multicast_leave

nx_igmp_loopback_disable

禁用 IGMP 环回

原型

```
UINT nx_igmp_loopback_disable(NX_IP *ip_ptr);
```

说明

此服务对加入的所有后续多播组禁用 IGMP 环回。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功禁用 IGMP 环回。
- **NX_NOT_ENABLED** (0x14) IGMP 未启用。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 调用方不是线程或初始化。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Disable IGMP loopback for all subsequent multicast groups joined. */
status = nx_igmp_loopback_disable(&ip_0);

/* If status is NX_SUCCESS IGMP loopback is disabled. */
```

另请参阅

- nx_igmp_enable、nx_igmp_info_get、nx_igmp_loopback_enable,
- nx_igmp_multicast_interface_join、nx_igmp_multicast_join,
- nx_igmp_multicast_leave

nx_igmp_loopback_enable

启用 IGMP 环回

原型

```
UINT nx_igmp_loopback_enable(NX_IP *ip_ptr);
```

说明

此服务对加入的所有后续多播组启用 IGMP 环回。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功禁用 IGMP 环回。
- **NX_NOT_ENABLED** (0x14) IGMP 未启用。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 调用方不是线程或初始化。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Enable IGMP loopback for all subsequent multicast groups joined. */
status = nx_igmp_loopback_enable(&ip_0);

/* If status is NX_SUCCESS IGMP loopback is enabled. */
```

另请参阅

- nx_igmp_enable、nx_igmp_info_get、nx_igmp_loopback_disable,
- nx_igmp_multicast_interface_join、nx_igmp_multicast_join,
- nx_igmp_multicast_leave

nx_igmp_multicast_interface_join

通过接口将 IP 实例加入指定的多播组

原型

```
UINT nx_igmp_multicast_interface_join(
    NX_IP *ip_ptr,
    ULONG group_address,
    UINT interface_index);
```

说明

此服务可通过指定的网络接口将 IP 实例加入指定的多播组。系统会维护内部计数器，以跟踪加入同一个组的次数。加入多播组之后，IGMP 组件会允许通过指定的网络接口接收具有该组地址的 IP 数据包，并且还会向路由器报告该 IP 是该多播组的成员。IGMP 成员身份加入、报告和退出消息也会通过指定的网络接口发送。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **group_address** 要加入的 D 类 IP 多播组地址，以主机字节顺序表示。
- **interface_index** 附加到 NetX 实例的接口的索引。

返回值

- **NX_SUCCESS** (0x00) 成功加入多播组。
- **NX_NO_MORE_ENTRIES** (0x17) 无法再加入更多的多播组，已超过最大数目。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_INVALID_INTERFACE** (0x4C) 设备索引指向无效的网络接口。
- **NX_IP_ADDRESS_ERROR** (0x21) 提供的多播组地址不是有效的 D 类地址。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) IP 多播支持未启用。

允许调用自

线程数

可以抢占

否

示例

```
/* Previously created IP Instance joins the multicast group
    244.0.0.200, via the interface at index 1 in the IP interface list. */
#define INTERFACE_INDEX 1
status = nx_igmp_multicast_interface_join
    (&ip, IP_ADDRESS(244,0,0,200),
    INTERFACE_INDEX);

/* If status is NX_SUCCESS, the IP instance has successfully joined
    the multicast group. */
```

另请参阅

- nx_igmp_enable、nx_igmp_info_get、nx_igmp_loopback_disable,
- nx_igmp_loopback_enable、nx_igmp_multicast_join,
- nx_igmp_multicast_leave

nx_igmp_multicast_join

将 IP 实例加入指定的多播组

原型

```
UINT nx_igmp_multicast_join(
    NX_IP *ip_ptr,
    ULONG group_address);
```

说明

此服务可将 IP 实例加入指定的多播组。系统会维护内部计数器，以跟踪加入同一个组的次数。如果这是网络上的第一个指示主机打算加入该组的加入请求，则会命令驱动程序发送 IGMP 报告。加入之后，IGMP 组件会允许接收具有该组地址的 IP 数据包，并向路由器报告该 IP 是该多播组的成员。

NOTE

若要在非主要设备上加入多播组，请使用 nx_igmp_multicast_interface_join 服务。

- **参数**
- **ip_ptr** 先前创建的 IP 实例的指针。
- **group_address** 要加入的 D 类 IP 多播组地址。

返回值

- **NX_SUCCESS** (0x00) 成功加入多播组。
- **NX_NO_MORE_ENTRIES** (0x17) 无法再加入更多的多播组，已超过最大数目。
- **NX_INVALID_INTERFACE** (0x4C) 设备索引指向无效的网络接口。
- **NX_IP_ADDRESS_ERROR** (0x21) IP 组地址无效。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
/* Previously created IP Instance ip_0 joins the multicast group
    224.0.0.200. */
status = nx_igmp_multicast_join(&ip_0, IP_ADDRESS(224,0,0,200));

/* If status is NX_SUCCESS, this IP instance has successfully
    joined the multicast group 224.0.0.200. */
```

另请参阅

- nx_igmp_enable、nx_igmp_info_get、nx_igmp_loopback_disable,
- nx_igmp_loopback_enable、nx_igmp_multicast_interface_join,
- nx_igmp_multicast_leave

nx_igmp_multicast_leave

使 IP 实例退出指定的多播组

原型

```
UINT nx_igmp_multicast_leave(
    NX_IP *ip_ptr,
    ULONG group_address);
```

说明

如果退出请求数与加入请求数相匹配，该服务会使 IP 实例退出指定的多播组。否则，只是将内部加入计数减小。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

- group_address 要退出的多播组。

返回值

- NX_SUCCESS (0x00) 成功加入多播组。
- NX_ENTRY_NOT_FOUND (0x16) 找不到以前的加入请求。
- NX_INVALID_INTERFACE (0x4C) 设备索引指向无效的网络接口。
- NX_IP_ADDRESS_ERROR (0x21) IP 组地址无效。
- NX_PTR_ERROR (0x07) IP 指针无效。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_NOT_ENABLED (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
/* Cause IP instance to leave the multicast group 224.0.0.200. */
status = nx_igmp_multicast_leave(&ip_0, IP_ADDRESS(224,0,0,200);

/* If status is NX_SUCCESS, this IP instance has successfully left
    the multicast group 224.0.0.200. */
```

另请参阅

- nx_igmp_enable、nx_igmp_info_get、nx_igmp_loopback_disable,
- nx_igmp_loopback_enable、nx_igmp_multicast_interface_join,
- nx_igmp_multicast_join

nx_ip_address_change_notifiy

在 IP 地址更改时通知应用程序

原型

```
UINT nx_ip_address_change_notify(
    NX_IP *ip_ptr,
    VOID(*change_notify)(NX_IP *,VOID *),
    VOID *additional_info);
```

说明

此服务注册一个应用程序通知函数，每当 IP 地址更改时，都会调用该函数。

参数

- ip_ptr 先前创建的 IP 实例的指针。
- change_notify 指向 IP 更改通知函数的指针。如果此参数为 NX_NULL，则会禁用 IP 地址更改通知。
- additional_info 指向可选附加信息的指针，此信息也会在 IP 地址更改时提供给通知函数。

返回值

- NX_SUCCESS (0x00) 通知成功更改 IP 地址。
- NX_PTR_ERROR (0x07) IP 指针无效。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Register the function "my_ip_changed" to be called whenever
the IP address is changed. */
status = nx_ip_address_change_notify(&ip_0, my_ip_changed, NX_NULL);

/* If status is NX_SUCCESS, the "my_ip_changed" function will be
    called whenever the IP address changes. */
```

另请参阅

- nx_ip_address_get、nx_ip_address_set、nx_ip_create、nx_ip_delete,
- nx_ip_driver_direct_command、nx_ip_driver_interface_direct_command,
- nx_ip_forwarding_disable、nx_ip_forwarding_enable,
- nx_ip_fragment_disable、nx_ip_fragment_enable、nx_ip_info_get,
- nx_ip_status_check、nx_system_initialize

nx_ip_address_get

检索 IP 地址和网络掩码

原型

```
UINT nx_ip_address_get(
    NX_IP *ip_ptr,
    ULONG *ip_address,
    ULONG *network_mask);
```

说明

此服务检索主要网络接口的 IP 地址及其子网掩码。

*若要获取辅助设备的信息，请使用以下服务

- nx_ip_interface_address_get。*

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **ip_address** 指向 IP 地址目标的指针。
- **network_mask** 指向网络掩码目标的指针。

返回值

- **NX_SUCCESS** (0x00) 成功获取 IP 地址。
- **NX_PTR_ERROR** (0x07) IP 或返回变量指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Get the IP address and network mask from the previously created
    IP Instance ip_0. */
status = nx_ip_address_get(&ip_0,
    &ip_address, &network_mask);

/* If status is NX_SUCCESS, the variables ip_address and
    network_mask contain the IP and network mask respectively. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_set、nx_ip_create,
- nx_ip_delete、nx_ip_driver_direct_command,
- nx_ip_driver_interface_direct_command、nx_ip_forwarding_disable,
- nx_ip_forwarding_enable、nx_ip_fragment_disable,
- nx_ip_fragment_enable、nx_ip_info_get、nx_ip_status_check,
- nx_system_initialize

nx_ip_address_set

设置 IP 地址和网络掩码

原型

```
UINT nx_ip_address_set(
    NX_IP *ip_ptr,
    ULONG ip_address,
    ULONG network_mask);
```

说明

此服务可设置主要网络接口的 IP 地址和网络掩码。

若要设置辅助设备的 IP 地址和网络掩码，请使用 nx_ip_interface_address_set 服务。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **ip_address** 新 IP 地址。
- **network_mask** 新网络掩码。

返回值

- **NX_SUCCESS** (0x00) 成功设置 IP 地址。
- **NX_IP_ADDRESS_ERROR** (0x21) IP 地址无效。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Set the IP address and network mask to 1.2.3.4 and 0xFFFFFF00
    for the previously created IP Instance ip_0. */
status = nx_ip_address_set(&ip_0, IP_ADDRESS(1,2,3,4), 0xFFFFFF00UL);

/* If status is NX_SUCCESS, the IP instance now has an IP address
    of 1.2.3.4 and a network mask of 0xFFFFFF00. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_get、nx_ip_create,
- nx_ip_delete、nx_ip_driver_direct_command,
- nx_ip_driver_interface_direct_command、nx_ip_forwarding_disable,
- nx_ip_forwarding_enable、nx_ip_fragment_disable,
- nx_ip_fragment_enable、nx_ip_info_get、nx_ip_status_check,
- nx_system_initialize

nx_ip_create

创建 IP 实例

原型

```
UINT nx_ip_create(
    NX_IP *ip_ptr,
    CHAR *name,
    ULONG ip_address,
    ULONG network_mask,
    NX_PACKET_POOL *default_pool,
    VOID (*ip_network_driver)(NX_IP_DRIVER *),
    VOID *memory_ptr,
    ULONG memory_size,
    UINT priority);
```

说明

此服务使用用户提供的 IP 地址和网络驱动程序创建 IP 实例。此外，应用程序必须为 IP 实例提供先前创建的数据包池，以用于内部数据包分配。请注意，直到此 IP 的线程执行之后，才会调用提供的应用程序网络驱动程序。

参数

- **ip_ptr** 指向用于创建新 IP 实例的控制块的指针。
- **name** 此新 IP 实例的名称。
- **ip_address** 此新 IP 实例的 IP 地址。
- **network_mask** 描述 IP 地址网络部分的掩码，用于实现子网和超网。
- **default_pool** 指向先前创建的 NetX 数据包池的控制块的指针。
- **ip_network_driver** 用户提供的网络驱动程序，用于发送和接收 IP 数据包。
- **memory_ptr** 指向 IP 帮助程序线程堆栈区域的内存区域的指针。
- **memory_size** IP 帮助程序线程堆栈的内存区域中的字节数。
- **priority** IP 帮助程序线程的优先级。

返回值

- **NX_SUCCESS** (0x00) 成功创建 IP 实例。
- **NX_NOT_IMPLEMENTED** (0x4A) 未正确配置 NetX 库。
- **NX_PTR_ERROR** (0x07) IP、网络驱动程序函数指针、数据包池或内存指针无效。
- **NX_SIZE_ERROR** (0x09) 提供的堆栈大小太小。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

- **NX_IP_ADDRESS_ERROR** (0x21) 提供的 IP 地址无效。
- **NX_OPTION_ERROR** (0x21) 提供的 IP 线程优先级无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Create an IP instance with an IP address of 1.2.3.4 and a network
    mask of 0xFFFFFF00UL. The "ethernet_driver" specifies the entry
    point of the application specific network driver and the
    "stack_memory_ptr" specifies the start of a 1024 byte memory
    area that is used for this IP instance's helper thread.  */
status = nx_ip_create(
    &ip_0,
    "NetX IP Instance ip_0",
    IP_ADDRESS(1, 2, 3, 4),
    0xFFFFFF00UL, &pool_0,
    ethernet_driver,
    stack_memory_ptr,
    1024,
    1);

/* If status is NX_SUCCESS, the IP instance has been created. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_get、nx_ip_address_set,
- nx_ip_delete、nx_ip_driver_direct_command,
- nx_ip_driver_interface_direct_command、nx_ip_forwarding_disable,
- nx_ip_forwarding_enable、nx_ip_fragment_disable,
- nx_ip_fragment_enable、nx_ip_info_get、nx_ip_status_check,
- nx_system_initialize

nx_ip_delete

删除先前创建的 IP 实例

原型

```
UINT nx_ip_delete(NX_IP *ip_ptr);
```

说明

此服务会删除先前创建的 IP 实例，并释放该 IP 实例拥有的所有系统资源。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功删除 IP。
- **NX_SOCKETS_BOUND** (0x28) 此 IP 实例仍有与其绑定的 UDP 或 TCP 套接字。必须先取消绑定并删除所有套接字，然后才能删除该 IP 实例。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

线程数

可以抢占

是

示例

```
/* Delete a previously created IP instance. */
status = nx_ip_delete(&ip_0);

/* If status is NX_SUCCESS, the IP instance has been deleted. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_get、nx_ip_address_set,
- nx_ip_create、nx_ip_driver_direct_command,
- nx_ip_driver_interface_direct_command、nx_ip_forwarding_disable,
- nx_ip_forwarding_enable、nx_ip_fragment_disable,
- nx_ip_fragment_enable、nx_ip_info_get、nx_ip_status_check,
- nx_system_initialize

nx_ip_driver_direct_command

向网络驱动程序发送命令

原型

```
UINT nx_ip_driver_direct_command(
    NX_IP *ip_ptr,
    UINT command,
    ULONG *return_value_ptr);
```

说明

此服务提供在 nx_ip_create 调用期间指定的应用程序主网络接口驱动程序的直接接口。

可以使用应用程序特定的命令，但前提是其数值大于或等于 NX_LINK_USER_COMMAND。

若要针对辅助设备发出命令，请使用 nx_ip_driver_interface_direct_command 服务。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **command** 数字命令代码。标准命令的定义如下：
- NX_LINK_GET_STATUS (10)
- NX_LINK_GET_SPEED (11)
- NX_LINK_GET_DUPLEX_TYPE (12)
- NX_LINK_GET_ERROR_COUNT (13)
- NX_LINK_GET_RX_COUNT (14)
- NX_LINK_GET_TX_COUNT (15)
- NX_LINK_GET_ALLOC_ERRORS (16)

- NX_LINK_USER_COMMAND (50)
- **return_value_ptr** 指向调用方中的返回变量的指针。

返回值

- **NX_SUCCESS** (0x00) 网络驱动程序直接命令成功。
- **NX_UNHANDLED_COMMAND** (0x44) 网络驱动程序命令未处理或未实现。
- **NX_PTR_ERROR** (0x07) IP 或返回值指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_INVALID_INTERFACE** (0x4C) 接口索引无效。

允许调用自

线程数

- **可以抢占**

否

示例

```
/* Make a direct call to the application-specific network driver
    for the previously created IP instance. For this example, the
    network driver is interrogated for the link status. */
status = nx_ip_driver_direct_command(
    &ip_0, NX_LINK_GET_STATUS,
    &link_status);

/* If status is NX_SUCCESS, the link_status variable contains a
    NX_TRUE or NX_FALSE value representing the status of the
    physical link. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_get、nx_ip_address_set,
- nx_ip_create、nx_ip_delete、nx_ip_driver_interface_direct_command,
- nx_ip_forwarding_disable、nx_ip_forwarding_enable,
- nx_ip_fragment_disable、nx_ip_fragment_enable、nx_ip_info_get,
- nx_ip_status_check、nx_system_initialize

nx_ip_driver_interface_direct_command

向网络驱动程序发送命令

原型

```
UINT nx_ip_driver_interface_direct_command(
    NX_IP *ip_ptr,
    UINT command,
    UINT interface_index,
    ULONG *return_value_ptr);
```

说明

此服务向 IP 实例中应用程序的网络设备驱动程序发出直接命令。可以使用应用程序特定的命令，但前提是其数值大于或等于 NX_LINK_USER_COMMAND。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

- command 数字命令代码。标准命令的定义如下：
- NX_LINK_GET_STATUS (10)
- NX_LINK_GET_SPEED (11)
- NX_LINK_GET_DUPLEX_TYPE (12)
- NX_LINK_GET_ERROR_COUNT (13)
- NX_LINK_GET_RX_COUNT (14)
- NX_LINK_GET_TX_COUNT (15)
- NX_LINK_GET_ALLOC_ERRORS (16)
- NX_LINK_USER_COMMAND (50)
- interface_index 应将该命令发送到的网络接口的索引。
- return_value_ptr 指向调用方中的返回变量的指针。

返回值

- NX_SUCCESS (0x00) 网络驱动程序直接命令成功。
- NX_UNHANDLED_COMMAND (0x44) 网络驱动程序命令未处理或未实现。
- NX_INVALID_INTERFACE (0x4C) 接口索引无效
- NX_PTR_ERROR (0x07) IP 或返回值指针无效。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Make a direct call to the application-specific network driver
    for the previously created IP instance. For this example, the
    network driver is interrogated for the link status. */

/* Set the interface index to the primary device. */
UINT interface_index = 0;
    status = nx_ip_driver_interface_direct_command(&ip_0,
    NX_LINK_GET_STATUS,
    interface_index,
    &link_status);
/* If status is NX_SUCCESS, the link_status variable contains a
    NX_TRUE or NX_FALSE value representing the status of the
    physical link. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_get、nx_ip_address_set,
- nx_ip_create、nx_ip_delete、nx_ip_driver_direct_command,
- nx_ip_forwarding_disable、nx_ip_forwarding_enable,
- nx_ip_fragment_disable、nx_ip_fragment_enable、nx_ip_info_get,
- nx_ip_status_check、nx_system_initialize

nx_ip_forwarding_disable

禁用 IP 数据包转发

原型

```
UINT nx_ip_forwarding_disable(NX_IP *ip_ptr);
```

说明

此服务用于在 NetX IP 组件内禁止转发 IP 数据包。创建 IP 任务时，会自动禁用此服务。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功禁用 IP 转发。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程、计时器

可以抢占

否

示例

```
/* Disable IP forwarding on this IP instance. */
status = nx_ip_forwarding_disable(&ip_0);

/* If status is NX_SUCCESS, IP forwarding has been disabled on the
    previously created IP instance. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_get、nx_ip_address_set,
- nx_ip_create、nx_ip_delete、nx_ip_driver_direct_command,
- nx_ip_driver_interface_direct_command、nx_ip_forwarding_enable,
- nx_ip_fragment_disable、nx_ip_fragment_enable、nx_ip_info_get,
- nx_ip_status_check、nx_system_initialize

nx_ip_forwarding_enable

启用 IP 数据包转发

原型

```
UINT nx_ip_forwarding_enable(NX_IP *ip_ptr);
```

说明

此服务用于在 NetX IP 组件内允许转发 IP 数据包。创建 IP 任务时，会自动禁用此服务。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功启用 IP 转发。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程、计时器

可以抢占

否

示例

```
/* Enable IP forwarding on this IP instance. */
status = nx_ip_forwarding_enable(&ip_0);

/* If status is NX_SUCCESS, IP forwarding has been enabled on the
    previously created IP instance. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_get、nx_ip_address_set,
- nx_ip_create、nx_ip_delete、nx_ip_driver_direct_command,
- nx_ip_driver_interface_direct_command、nx_ip_forwarding_disable,
- nx_ip_fragment_disable、nx_ip_fragment_enable、nx_ip_info_get,
- nx_ip_status_check、nx_system_initialize

nx_ip_fragment_disable

禁用 IP 数据包分段

原型

```
UINT nx_ip_fragment_disable(NX_IP *ip_ptr);
```

说明

此服务可禁用 IP 数据包分段和重组功能。对于正在等待重组的数据包，此服务会将其释放。创建 IP 任务时，会自动禁用此服务。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功禁用 IP 分段。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 该 IP 实例上未启用 IP 分段。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Disable IP fragmenting on this IP instance. */
status = nx_ip_fragment_disable(&ip_0);

/* If status is NX_SUCCESS, disables IP fragmenting on the
    previously created IP instance. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_get、nx_ip_address_set,
- nx_ip_create、nx_ip_delete、nx_ip_driver_direct_command,
- nx_ip_driver_interface_direct_command、nx_ip_forwarding_disable,
- nx_ip_forwarding_enable、nx_ip_fragment_enable、nx_ip_info_get,
- nx_ip_status_check、nx_system_initialize

nx_ip_fragment_enable

启用 IP 数据包分段

原型

```
UINT nx_ip_fragment_enable(NX_IP *ip_ptr);
```

说明

此服务可启用 IP 数据包分段和重组功能。创建 IP 任务时，会自动禁用此服务。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功启用 IP 分段。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) IP 分段功能未编译到 NetX 中。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Enable IP fragmenting on this IP instance. */
status = nx_ip_fragment_enable(&ip_0);

/* If status is NX_SUCCESS, IP fragmenting has been enabled on the
    previously created IP instance. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_get、nx_ip_address_set,
- nx_ip_create、nx_ip_delete、nx_ip_driver_direct_command,
- nx_ip_driver_interface_direct_command、nx_ip_forwarding_disable,
- nx_ip_forwarding_enable、nx_ip_fragment_disable、nx_ip_info_get,

- nx_ip_status_check、nx_system_initialize

nx_ip_gateway_address_set

设置网关 IP 地址

原型

```
UINT nx_ip_gateway_address_set(
    NX_IP *ip_ptr,
    ULONG ip_address);
```

说明

此服务用于设置 IP 网关 IP 地址。所有网络出站流量都会路由到该网关进行传输。该网关必须可通过其中一个网络接口直接访问。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **ip_address** 网关的 IP 地址。

返回值

- **NX_SUCCESS** (0x00) 成功设置网关 IP 地址。
- **NX_PTR_ERROR** (0x07) IP 实例指针无效。
- **NX_IP_ADDRESS_ERROR** (0x21) IP 地址无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Setup the Gateway address for previously created IP
    Instance ip_0. */
status = nx_ip_gateway_address_set(&ip_0, IP_ADDRESS(1,2,3,99));

/* If status is NX_SUCCESS, all out-of-network send requests are
    routed to 1.2.3.99. */
```

另请参阅

- nx_ip_info_get、nx_ip_static_route_add、nx_ip_static_route_delete

nx_ip_info_get

检索 IP 活动的相关信息

原型

```
UINT nx_ip_info_get(
    NX_IP *ip_ptr,
    ULONG *ip_total_packets_sent,
    ULONG *ip_total_bytes_sent,
    ULONG *ip_total_packets_received,
    ULONG *ip_total_bytes_received,
    ULONG *ip_invalid_packets,
    ULONG *ip_receive_packets_dropped,
    ULONG *ip_receive_checksum_errors,
    ULONG *ip_send_packets_dropped,
    ULONG *ip_total_fragments_sent,
    ULONG *ip_total_fragments_received);
```

说明

此服务检索指定的 IP 实例的 IP 活动相关信息。

如果目标指针为 NX_NULL，则不会将该特定信息返回给调用方。

参数

- ip_ptr 先前创建的 IP 实例的指针。
- ip_total_packets_sent 此指针指向已发送的 IP 数据包总数的目标。
- ip_total_bytes_sent 此指针指向已发送的总字节数的目标。
- ip_total_packets_received 此指针指向 IP 接收数据包总数的目标。
- ip_total_bytes_received 此指针指向已接收的 IP 总字节数的目标。
- ip_invalid_packets 此指针指向无效 IP 数据包总数的目标。
- ip_receive_packets_dropped 此指针指向已删除的接收数据包总数的目标。
- ip_receive_checksum_errors 此指针指向接收数据包中校验和错误总数的目标。
- ip_send_packets_dropped 此指针指向已删除的发送数据包总数的目标。
- ip_total_fragments_sent 此指针指向已发送的片段总数的目标。
- ip_total_fragments_received 此指针指向已接收的片段总数的目标。

返回值

- NX_SUCCESS (0x00) 成功检索 IP 信息。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_PTR_ERROR (0x07) IP 指针无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Retrieve IP information from previously created IP
Instance 0. */
status = nx_ip_info_get(&ip_0,
    &ip_total_packets_sent,
    &ip_total_bytes_sent,
    &ip_total_packets_received,
    &ip_total_bytes_received,
    &ip_invalid_packets,
    &ip_receive_packets_dropped,
    &ip_receive_checksum_errors,
    &ip_send_packets_dropped,
    &ip_total_fragments_sent,
    &ip_total_fragments_received);

/* If status is NX_SUCCESS, IP information was retrieved. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_get、nx_ip_address_set,
- nx_ip_create、nx_ip_delete、nx_ip_driver_direct_command,
- nx_ip_driver_interface_direct_command、nx_ip_forwarding_disable,
- nx_ip_forwarding_enable、nx_ip_fragment_disable,
- nx_ip_fragment_enable、nx_ip_status_check、nx_system_initialize

nx_ip_interface_address_get

检索接口 IP 地址

原型

```
UINT nx_ip_interface_address_get (
    NX_IP *ip_ptr,
    UINT interface_index,
    ULONG *ip_address,
    ULONG *network_mask);
```

说明

此服务可检索指定网络接口的 IP 地址。

如果指定的设备不是主要设备，则必须预先附加到该 IP 实例。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **interface_index** 接口索引，与附加到 IP 实例的网络接口的索引值相同。
- **ip_address** 指向设备接口 IP 地址的目标的指针。
- **network_mask** 指向设备接口网络掩码的目标的指针。

返回值

- **NX_SUCCESS** (0x00) 成功获取 IP 地址。
- **NX_INVALID_INTERFACE** (0x4C) 指定的网络接口无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_PTR_ERROR** (0x07) IP 指针无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
#define INTERFACE_INDEX 1
/* Get device IP address and network mask for the specified
    interface index 1 in IP instance list of interfaces). */
status = nx_ip_interface_address_get(ip_ptr,INTERFACE_INDEX,
    &ip_address, &network_mask);

/* If status is NX_SUCCESS the interface address was successfully
    retrieved. */
```

另请参阅

- nx_ip_interface_address_set、nx_ip_interface_attach,
- nx_ip_interface_info_get、nx_ip_interface_status_check,
- nx_ip_link_status_change_notify_set

nx_ip_interface_address_set

设置接口 IP 地址和网络掩码

原型

```
UINT nx_ip_interface_address_set(
    NX_IP *ip_ptr,
    UINT interface_index,
    ULONG ip_address,
    ULONG network_mask);
```

说明

此服务可为指定的 IP 接口设置 IP 地址和网络掩码。

指定的接口必须预先附加到该 IP 实例。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **interface_index** 附加到 NetX 实例的接口的索引。
- **ip_address** 新的网络接口 IP 地址。
- **network_mask** 新的接口网络掩码。

返回值

- **NX_SUCCESS** (0x00) 成功设置 IP 地址。
- **NX_INVALID_INTERFACE** (0x4C) 指定的网络接口无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_PTR_ERROR** (0x07) 指针无效。
- **NX_IP_ADDRESS_ERROR** (0x21) IP 地址无效

允许调用自

初始化、线程

可以抢占

否

示例

```
#define INTERFACE_INDEX 1
/* Set device IP address and network mask for the specified
    interface index 1 in IP instance list of interfaces). */
status = nx_ip_interface_address_set(ip_ptr, INTERFACE_INDEX,
    ip_address, network_mask);

/* If status is NX_SUCCESS the interface IP address and mask was
successfully set. */
```

另请参阅

- nx_ip_interface_address_get、nx_ip_interface_attach,
- nx_ip_interface_info_get、nx_ip_interface_status_check,
- nx_ip_link_status_change_notify_set

nx_ip_interface_attach

将网络接口附加到 IP 实例

原型

```
UINT nx_ip_interface_attach(
    NX_IP *ip_ptr,
    CHAR *interface_name,
    ULONG ip_address,
    ULONG network_mask,
    VOID(*ip_link_driver)
    (struct NX_IP_DRIVER_STRUCT *));
```

说明

此服务用于向 IP 接口添加物理网络接口。请注意，IP 实例是使用主接口创建的，因此每个额外的接口都是主接口的辅助接口。附加到 IP 实例的网络接口总数（包括主接口）不得超过 NX_MAX_PHYSICAL_INTERFACES。

如果 IP 线程尚未运行，则辅助接口会在初始化所有物理接口的 IP 线程启动过程中进行初始化。

如果 IP 线程未在运行，则辅助接口会在 nx_ip_interface_attach 服务中进行初始化。

ip_ptr 必须指向有效的 NetX IP 结构。

- 必须根据 IP 实例的网络接口数配置 NX_MAX_PHYSICAL_INTERFACES。默认值为 1。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **interface_name** 指向接口名称字符串的指针。
- **ip_address** 设备 IP 地址，以主机字节顺序表示。
- **network_mask** 设备网络掩码，以主机字节顺序表示。
- **ip_link_driver** 接口的以太网驱动程序。

返回值

- **NX_SUCCESS** (0x00) 条目已添加到静态路由表中。
- **NX_NO_MORE_ENTRIES** (0x17) 已超过最大接口数
- NX_MAX_PHYSICAL_INTERFACES。
- **NX_DUPLICATED_ENTRY** (0x52) 提供的 IP 地址已在该 IP 实例上使用。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_PTR_ERROR** (0x07) 指针输入无效。
- **NX_IP_ADDRESS_ERROR** (0x21) IP 地址输入无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Attach secondary device for device IP address 192.168.1.68 with
    the specified Ethernet driver. */
status = nx_ip_interface_attach(ip_ptr, "secondary_port",
    IP_ADDRESS(192,168,1,68),
    0xFFFFFF00UL,
    nx_etherDriver);
/* If status is NX_SUCCESS the interface was successfully added to
    the IP instance interface table. */
```

另请参阅

- nx_ip_interface_address_get、nx_ip_interface_address_set,
- nx_ip_interface_info_get、nx_ip_interface_status_check,
- nx_ip_link_status_change_notify_set

nx_ip_interface_info_get

检索网络接口参数

原型

```
UINT nx_ip_interface_info_get(
    NX_IP *ip_ptr,
    UINT interface_index,
    CHAR **interface_name,
    ULONG *ip_address,
    ULONG *network_mask,
    ULONG *mtu_size,
    ULONG *physical_address_msw,
    ULONG *physical_address_lsw);
```

说明

此服务用于检索指定网络接口的网络参数相关信息。所有数据都按主机字节顺序进行检索。

ip_ptr 必须指向有效的 NetX IP 结构。如果指定的接口不是主接口，它必须预先附加到该 IP 实例。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **interface_index** 指定网络接口的索引。
- **interface_name** 指向缓冲区的指针，该缓冲区中包含网络接口名称。
- **ip_address** 指向接口 IP 地址目标的指针。
- **network_mask** 指向网络掩码目标的指针。
- **mtu_size** 指向此接口的最大传输单元目标的指针。
- **physical_address_msw** 指向设备 MAC 地址高 16 位的目标的指针。
- **physical_address_lsw** 指向设备 MAC 地址低 32 位的目标的指针。

返回值

- **NX_SUCCESS** (0x00) 已获取接口信息。

- **NX_PTR_ERROR** (0x07) 指针输入无效。
- **NX_INVALID_INTERFACE** (0x4C) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 未从系统初始化或线程上下文调用该服务。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Retrieve interface parameters for the specified interface (index
    1 in IP instance list of interfaces). */
#define INTERFACE_INDEX 1

status = nx_ip_interface_info_get(ip_ptr, INTERFACE_INDEX,
    &name_ptr, &ip_address,
    &network_mask,
    &mtu_size,
    &physical_address_msw,
    &physical_address_lsw);

/* If status is NX_SUCCESS the interface information is
    successfully retrieved. */
```

另请参阅

- nx_ip_interface_address_get、nx_ip_interface_address_set,
- nx_ip_interface_attach、nx_ip_interface_status_check,
- nx_ip_link_status_change_notify_set

nx_ip_interface_status_check

检查 IP 实例的状态

原型

```
UINT nx_ip_interface_status_check(
    NX_IP *ip_ptr,
    UINT interface_index
    ULONG needed_status,
    ULONG *actual_status,
    ULONG wait_option);
```

说明

此服务用于检查并选择性等待先前创建的 IP 实例的网络接口的指定状态。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **interface_index** 接口索引号
- **needed_status** 所请求的 IP 状态，以位图形式定义，如下所示：
  - NX_IP_INITIALIZE_DONE (0x0001)
  - NX_IP_ADDRESS_RESOLVED (0x0002)
  - NX_IP_LINK_ENABLED (0x0004)
  - NX_IP_ARP_ENABLED (0x0008)
  - NX_IP_UDP_ENABLED (0x0010)

- NX_IP_TCP_ENABLED (0x0020)
- NX_IP_IGMP_ENABLED (0x0040)
- NX_IP_RARP_COMPLETE (0x0080)
- NX_IP_INTERFACE_LINK_ENABLED (0x0100)

- **actual_status** 指向实际位集目标的指针。
- **wait_option** 定义所请求的状态位不可用时该服务的行为方式。等待选项的定义如下：
  - NX_NO_WAIT (0x00000000)
  - NX_WAIT_FOREVER (0xFFFFFFFF)
  - 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) 成功检查 IP 状态。
- **NX_NOT_SUCCESSFUL** (0x43) 未在指定的超时时间内满足状态请求。
- **NX_PTR_ERROR** (0x07) IP 指针无效或已变为无效，或者实际状态指针无效。
- **NX_OPTION_ERROR** (0x0a) 必需的状态选项无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_INVALID_INTERFACE** (0x4C) 接口索引超出范围，或接口无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Wait 10 ticks for the link up status on the previously created IP
    instance. */
status = nx_ip_interface_status_check(&ip_0, 1, NX_IP_LINK_ENABLED,
    &actual_status, 10);

/* If status is NX_SUCCESS, the secondary link for the specified IP
    instance is up. */
```

另请参阅

- nx_ip_interface_address_get、nx_ip_interface_address_set,
- nx_ip_interface_attach、nx_ip_interface_info_get,
- nx_ip_link_status_change_notify_set

nx_ip_link_status_change_notify_set

设置链路状态更改通知回调函数

原型

```
UINT nx_ip_link_status_change_notify_set(
    NX_IP *ip_ptr,
    VOID (*link_status_change_notify)(NX_IP *ip_ptr, UINT interface_index, UINT link_up));
```

说明

此服务用于配置链路状态更改通知回调函数。当主要或辅助接口状态更改（例如 IP 地址更改）时，会调用用户提供的 link_status_change_notify 例程。如果 link_status_change_notify 为 NULL，则会禁用链路状态更改通知回调功能 。

参数

- **ip_ptr** IP 控制块指针
- **link_status_change_notify** 用户提供的回调函数，物理接口更改时会调用该函数。

返回值

- **NX_SUCCESS** (0x00) 设置成功
- **NX_PTR_ERROR** (0x07) IP 控制块指针或新的物理地址指针无效
- **NX_CALLER_ERROR** (0x11) 未从系统初始化或线程上下文调用该服务。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Configure a callback function to be used when the physical
    interface status is changed. */
status = nx_ip_link_status_change_notify_set(&ip_0, my_change_cb);

/* If status == NX_SUCCESS, the link status change notify function
    is set. */
```

另请参阅

- nx_ip_interface_address_get、nx_ip_interface_address_set,
- nx_ip_interface_attach、nx_ip_interface_info_get,
- nx_ip_interface_status_check

nx_ip_raw_packet_disable

禁用原始数据包发送/接收

原型

```
UINT nx_ip_raw_packet_disable(NX_IP *ip_ptr);
```

说明

此服务用于禁止此 IP 实例传输和接收原始 IP 数据包。如果先前已启用原始数据包服务，并且接收队列中存在原始数据包，则此服务会释放所有已收到的原始数据包。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功禁用 IP 原始数据包。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Disable raw packet sending/receiving for this IP instance. */
status = nx_ip_raw_packet_disable(&ip_0);

/* If status is NX_SUCCESS, raw IP packet sending/receiving has
    been disabled for the previously created IP instance. */
```

另请参阅

- nx_ip_raw_packet_enable、nx_ip_raw_packet_receive,
- nx_ip_raw_packet_send、nx_ip_raw_packet_interface_send

nx_ip_raw_packet_enable

启用原始数据包处理

原型

```
UINT nx_ip_raw_packet_enable(NX_IP *ip_ptr);
```

说明

此服务用于允许此 IP 实例传输和接收原始 IP 数据包。传入的 TCP、UDP、ICMP 和 IGMP 数据包仍由 NetX 处理。上层协议类型未知的数据包由原始数据包接收例程处理。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功启用 IP 原始数据包。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Enable raw packet sending/receiving for this IP instance. */
status = nx_ip_raw_packet_enable(&ip_0);

/* If status is NX_SUCCESS, raw IP packet sending/receiving has
    been enabled for the previously created IP instance. */
```

另请参阅

- nx_ip_raw_packet_disable、nx_ip_raw_packet_receive,
- nx_ip_raw_packet_send、nx_ip_raw_packet_interface_send

nx_ip_raw_packet_interface_send

通过指定网络接口发送原始 IP 数据包

原型

```
UINT nx_ip_raw_packet_interface_send(
    NX_IP *ip_ptr,
    NX_PACKET *packet_ptr,
    ULONG destination_ip,
    UINT address_index,
    ULONG type_of_service);
```

说明

此服务将指定的本地 IP 地址用作源地址，通过关联的网络接口将原始 IP 数据包发送到目标 IP 地址。请注意，此例程会立即返回，因此不知道实际是否已发送该 IP 数据包。网络驱动程序负责在传输完成后释放该数据包。此服务与其他服务的不同之处在于，无法知道实际是否已发送该数据包。该数据包可能会在 Internet 上丢失。

请注意，必须启用原始 IP 处理。

此服务与 nx_ip_raw_packet_send 类似，只不过此服务允许应用程序从指定的物理接口发送原始 IP 数据包。

参数

- ip_ptr 指向先前创建的 IP 任务的指针。
- packet_ptr 指向要传输的数据包的指针。
- destination_ip 用于发送数据包的 IP 地址。
- address_index 用于发送数据包的接口的地址索引。
- type_of_service 用于数据包的服务类型。

返回值

- NX_SUCCESS (0x00) 数据包传输成功。
- NX_IP_ADDRESS_ERROR (0x21) 没有合适的传出接口可用。
- NX_NOT_ENABLED (0x14) Raw IP 数据包处理未启用。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_PTR_ERROR (0x07) 指针输入无效。
- NX_OPTION_ERROR (0x0A) 指定了无效的服务类型。
- NX_OVERFLOW (0x03) 数据包前置指针无效。
- NX_UNDERFLOW (0x02) 数据包前置指针无效。
- NX_INVALID_INTERFACE (0x4C) 指定了无效的接口索引。

允许调用自

线程数

可以抢占

否

示例

```
#define ADDRESS_IDNEX 1
/* Send packet out on interface 1 with normal type of service. */
status = nx_ip_raw_packet_interface_send(ip_ptr, packet_ptr,
    destination_ip,
    ADDRESS_INDEX,
    NX_IP_NORMAL);
/* If status is NX_SUCCESS the packet was successfully transmitted. */
```

另请参阅

- nx_ip_raw_packet_disable、nx_ip_raw_packet_enable,
- nx_ip_raw_packet_receive、nx_ip_raw_packet_send

nx_ip_raw_packet_receive

接收原始 IP 数据包

原型

```
UINT nx_ip_raw_packet_receive(
    NX_IP *ip_ptr,
    NX_PACKET **packet_ptr,
    ULONG wait_option);
```

说明

此服务用于从指定的 IP 实例接收原始 IP 数据包。如果原始数据包接收队列中已有 IP 数据包，则会将第一个（最旧）数据包返回给调用方。如果没有可用的数据包，则调用方可能会挂起，如等待选项所指定。

如果返回了 NX_SUCCESS，则应用程序负责在不再需要收到的数据包时将其释放。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **packet_ptr** 此指针指向放置收到的原始 IP 数据包的指针。
- **wait_option** 定义没有可用的原始 IP 数据包时该服务的行为方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) 成功接收 IP 原始数据包。
- **NX_NO_PACKET** (0x01) 没有可用的数据包。
- **NX_WAIT_ABORTED** (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。
- **NX_PTR_ERROR** (0x07) IP 指针或返回数据包指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效

允许调用自

线程数

可以抢占

否

示例

```
/* Receive a raw IP packet for this IP instance, wait for a maximum
    of 4 timer ticks. */
status = nx_ip_raw_packet_receive(&ip_0, &packet_ptr, 4);

/* If status is NX_SUCCESS, the raw IP packet pointer is in the
    variable packet_ptr. */
```

另请参阅

- nx_ip_raw_packet_disable、nx_ip_raw_packet_enable,
- nx_ip_raw_packet_send、nx_ip_raw_packet_interface_send

nx_ip_raw_packet_send

发送原始 IP 数据包

原型

```
UINT nx_ip_raw_packet_send(
    NX_IP *ip_ptr,
    NX_PACKET *packet_ptr,
    ULONG destination_ip,
    ULONG type_of_service);
```

说明

此服务用于将原始 IP 数据包发送到目标 IP 地址。请注意，此例程会立即返回，因此不知道实际是否已发送该 IP 数据包。网络驱动程序负责在传输完成后释放该数据包。

对于多宿主系统，NetX 会使用目标 IP 地址来查找适当的网络接口，并将该接口的 IP 地址用作源地址。如果目标 IP 地址为广播或多播，则使用第一个有效接口。在本例中，应用程序使用 nx_ip_raw_packet_interface_send。

除非返回了错误，否则应用程序不应在此调用后释放该数据包。这样做会导致不可预知的结果，因为网络驱动程序会在传输后释放该数据包。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **packet_ptr** 指向要发送的原始 IP 数据包的指针。
- **destination_ip** 目标 IP 地址，这可以是特定的主机 IP 地址、网络广播、内部环回或多播地址。
- **type_of_service** 定义用于传输的服务类型，合法值如下所示：
- NX_IP_NORMAL (0x00000000)
- NX_IP_MIN_DELAY (0x00100000)
- NX_IP_MAX_DATA (0x00080000)
- NX_IP_MAX_RELIABLE (0x00040000)
- NX_IP_MIN_COST (0x00020000)

返回值

- **NX_SUCCESS** (0x00) 成功发起 IP 原始数据包发送。
- **NX_IP_ADDRESS_ERROR** (0x21) IP 地址无效。
- **NX_NOT_ENABLED** (0x14) 原始 IP 功能未启用。
- **NX_OPTION_ERROR** (0x0A) 服务类型无效。
- **NX_UNDERFLOW** (0x02) 空间不足，无法在数据包上前置 IP 标头。
- **NX_OVERFLOW** (0x03) 数据包追加指针无效。
- **NX_PTR_ERROR** (0x07) IP 指针或数据包指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Send a raw IP packet to IP address 1.2.3.5. */
status = nx_ip_raw_packet_send(&ip_0, packet_ptr,
    IP_ADDRESS(1,2,3,5),
    NX_IP_NORMAL);

/* If status is NX_SUCCESS, the raw IP packet pointed to by
    packet_ptr has been sent. */
```

另请参阅

- nx_ip_raw_packet_disable、nx_ip_raw_packet_enable,
- nx_ip_raw_packet_receive、nx_ip_raw_packet_send,
- nx_ip_raw_packet_interface_send

nx_ip_static_route_add

将静态路由添加到路由表

原型

```
UINT nx_ip_static_route_add(
    NX_IP *ip_ptr,
    ULONG network_address,
    ULONG net_mask,
    ULONG next_hop);
```

说明

此服务用于在静态路由表中添加一个条目。请注意，next_hop 地址必须可从其中一个本地网络设备直接访问。

请注意，ip_ptr 必须指向有效的 NetX IP 结构，而且必须在定义了 NX_ENABLE_IP_STATIC_ROUTING 的情况下生成 NetX 库才能使用此服务。默认情况下，NetX 是在未定义 NX_ENABLE_IP_STATIC_ROUTING 的情况下生成的。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **network_address** 目标网络地址，以主机字节顺序表示
- **net_mask** 目标网络掩码，以主机字节顺序表示
- **next_hop** 目标网络的下一跃点地址，以主机字节顺序表示

返回值

- **NX_SUCCESS** (0x00) 条目已添加到静态路由表中。
- **NX_OVERFLOW** (0x03) 静态路由表已满。
- **NX_NOT_SUPPORTED** (0x4B) 此功能尚未编译在内。
- **NX_IP_ADDRESS_ERROR** (0x21) 无法通过本地接口直接访问下一个跃点。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_PTR_ERROR** (0x07) ip_ptr 指针无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Specify the next hop for the 192.168.10.0 through the gateway
    192.168.1.1. */
status = nx_ip_static_route_add(ip_ptr, IP_ADDRESS(192,168,10,0),
    0xFFFFFF00UL,
    IP_ADDRESS(192,168,1,1));

/* If status is NX_SUCCESS the route was successfully added to the
    static routing table. */
```

另请参阅

- nx_ip_gateway_address_set、nx_ip_info_get、nx_ip_static_route_delete

nx_ip_static_route_delete

从路由表中删除静态路由

原型

```
UINT nx_ip_static_route_delete(
    NX_IP *ip_ptr,
    ULONG network_address,
    ULONG net_mask);
```

说明

此服务用于从静态路由表中删除条目。

请注意，ip_ptr 必须指向有效的 NetX IP 结构，而且必须在定义了 NX_ENABLE_IP_STATIC_ROUTING 的情况下生成 NetX 库才能使用此服务。默认情况下，NetX 是在未定义 NX_ENABLE_IP_STATIC_ROUTING 的情况下生成的。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **network_address** 目标网络地址，以主机字节顺序表示。
- **net_mask** 目标网络掩码，以主机字节顺序表示。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Remove the static route for 192.168.10.0 from the routing table.*/
status = nx_ip_static_route_delete(ip_ptr,
    IP_ADDRESS(192,168,10,0), 0xFFFFFF00UL,);

/* If status is NX_SUCCESS the route was successfully removed from
    the static routing table. */
```

另请参阅

- nx_ip_gateway_address_set、nx_ip_info_get、nx_ip_static_route_add

nx_ip_status_check

检查 IP 实例的状态

原型

```
UINT nx_ip_status_check(
    NX_IP *ip_ptr,
    ULONG needed_status,
    ULONG *actual_status,
    ULONG wait_option);
```

说明

此服务用于检查并选择性等待先前创建的 IP 实例的主网络接口的指定状态。若要获取辅助接口的状态，应用程序应使用 nx_ip_interface_status_check 服务。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **needed_status** 所请求的 IP 状态，以位图形式定义，如下所示：
- NX_IP_INITIALIZE_DONE (0x0001)
- NX_IP_ADDRESS_RESOLVED (0x0002)
- NX_IP_LINK_ENABLED (0x0004)
- NX_IP_ARP_ENABLED (0x0008)
- NX_IP_UDP_ENABLED (0x0010)
- NX_IP_TCP_ENABLED (0x0020)
- NX_IP_IGMP_ENABLED (0x0040)
- NX_IP_RARP_COMPLETE (0x0080)
- NX_IP_INTERFACE_LINK_ENABLED (0x0100)
- **actual_status** 指向实际位集目标的指针。
- **wait_option** 定义所请求的状态位不可用时该服务的行为方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) 成功检查 IP 状态。
- **NX_NOT_SUCCESSFUL** (0x43) 未在指定的超时时间内满足状态请求。
- **NX_PTR_ERROR** (0x07) IP 指针无效或已变为无效，或者实际状态指针无效。
- **NX_OPTION_ERROR** (0x0a) 必需的状态选项无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Wait 10 ticks for the link up status on the previously created IP
    instance. */
status = nx_ip_status_check(&ip_0, NX_IP_LINK_ENABLED,
    &actual_status, 10);

/* If status is NX_SUCCESS, the link for the specified IP instance
    is up. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_get、nx_ip_address_set,
- nx_ip_create、nx_ip_delete、nx_ip_driver_direct_command,
- nx_ip_driver_interface_direct_command、nx_ip_forwarding_disable,
- nx_ip_forwarding_enable、nx_ip_fragment_disable,
- nx_ip_fragment_enable、nx_ip_info_get、nx_system_initialize

nx_packet_allocate

从指定的池分配数据包

原型

```
UINT nx_packet_allocate(
    NX_PACKET_POOL *pool_ptr,
    NX_PACKET **packet_ptr,
    ULONG packet_type,
    ULONG wait_option);
```

说明

此服务用于从指定的池分配数据包，并根据指定的数据包类型调整该数据包中的前置指针。没有可用的数据包时，此服务会根据提供的等待选项挂起。

参数

- **pool_ptr** 指向先前创建的数据包池的指针。
- **packet_ptr** 此指针指向分配的数据包指针的指针。
- **packet_type** 定义所请求的数据包类型。有关支持的数据包类型的列表，请参阅第 3 章第 49 页的"数据包池"。
- **wait_option** 定义数据包池中没有可用数据包时的等待时间（以时钟周期为单位）。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) 成功分配数据包。
- **NX_NO_PACKET** (0x01) 没有可用的数据包。
- **NX_WAIT_ABORTED** (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。
- **NX_INVALID_PARAMETERS** (0x4D) 数据包大小无法支持协议。
- **NX_OPTION_ERROR** (0x0A) 数据包类型无效。
- **NX_PTR_ERROR** (0x07) 池指针或数据包返回指针无效。
- **NX_CALLER_ERROR** (0x11) 来自非线程的无效等待选项。

允许调用自

初始化、线程、计时器和 ISR（应用程序网络驱动程序）。在 ISR 或计时器上下文中使用时，等待选项必须是 NX_NO_WAIT。

可以抢占

否

示例

```
/* Allocate a new UDP packet from the previously created packet pool
    and suspend for a maximum of 5 timer ticks if the pool is
    empty. */
status = nx_packet_allocate(&pool_0, &packet_ptr, NX_UDP_PACKET, 5);

/* If status is NX_SUCCESS, the newly allocated packet pointer is
    found in the variable packet_ptr. */
```

另请参阅

- nx_packet_copy、nx_packet_data_append,
- nx_packet_data_extract_offset、nx_packet_data_retrieve,
- nx_packet_length_get、nx_packet_pool_create、nx_packet_pool_delete,
- nx_packet_pool_info_get、nx_packet_release,
- nx_packet_transmit_release

nx_packet_copy

复制数据包

原型

```
UINT nx_packet_copy(
    NX_PACKET *packet_ptr,
    NX_PACKET **new_packet_ptr,
    NX_PACKET_POOL *pool_ptr,
    ULONG wait_option);
```

说明

此服务用于将提供的数据包中的信息复制到一个或多个从提供的数据包池分配的新数据包。如果成功，会在 new_packet_ptr 所指向的目标中返回指向新数据包的指针。

参数

- **packet_ptr** 指向源数据包的指针。
- **new_packet_ptr** 目标的指针，在该目标中返回指向数据包新副本的指针。
- **pool_ptr** 先前创建的数据包池的指针，该池用于为副本分配一个或多个数据包。
- **wait_option** 定义没有可用的数据包时该服务的等待方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) 成功复制数据包。
- **NX_NO_PACKET** (0x01) 没有可供复制的数据包。
- **NX_INVALID_PACKET** (0x12) 源数据包为空，或者复制失败。
- **NX_WAIT_ABORTED** (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。
- **NX_INVALID_PARAMETERS** (0x4D) 数据包大小无法支持协议。
- **NX_PTR_ERROR** (0x07) 池指针、数据包指针或目标指针无效。
- **NX_UNDERFLOW** (0x02) 数据包前置指针无效。
- **NX_OVERFLOW** (0x03) 数据包追加指针无效。
- **NX_CALLER_ERROR** (0x11) 在初始化期间或在 ISR 中指定了等待选项。

允许调用自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
NX_PACKET *new_copy_ptr;

/* Copy packet pointed to by "old_packet_ptr" using packets from
    previously created packet pool_0. */
status = nx_packet_copy(old_packet, &new_copy_ptr, &pool_0, 20);

/* If status is NX_SUCCESS, new_copy_ptr points to the packet copy. */
```

另请参阅

- nx_packet_allocate、nx_packet_data_append,
- nx_packet_data_extract_offset、nx_packet_data_retrieve,
- nx_packet_length_get、nx_packet_pool_create、nx_packet_pool_delete,
- nx_packet_pool_info_get、nx_packet_release,
- nx_packet_transmit_release

nx_packet_data_append

将数据追加到数据包末尾

原型

```
UINT nx_packet_data_append(
    NX_PACKET *packet_ptr,
    VOID *data_start,
    ULONG data_size,
    NX_PACKET_POOL *pool_ptr,
    ULONG wait_option);
```

说明

此服务用于将数据追加到指定的数据包末尾。提供的数据区域会复制到该数据包中。如果可用内存不足，并且已启用链式数据包功能，则会分配一个或多个数据包来满足该请求。如果未启用链式数据包功能，则返回 NX_SIZE_ERROR。

参数

- **packet_ptr** 数据包指针。
- **data_start** 此指针指向要追加到数据包的用户数据区域的开头。
- **data_size** 用户数据区域的大小。
- **pool_ptr** 数据包池的指针，在当前数据包中空间不足时，会从该池分配另一个数据包。
- **wait_option** 定义没有可用的数据包时该服务的行为方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) 成功追加数据包。
- **NX_NO_PACKET** (0x01) 没有可用的数据包。
- **NX_WAIT_ABORTED** (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。

- **NX_INVALID_PARAMETERS** (0x4D) 数据包大小无法支持协议。
- **NX_UNDERFLOW** (0x02) 前置指针小于有效负载开始位置。
- **NX_OVERFLOW** (0x03) 追加指针大于有效负载结束位置。
- **NX_PTR_ERROR** (0x07) 池、数据包或数据指针无效。
- **NX_SIZE_ERROR** (0x09) 数据大小无效。
- **NX_CALLER_ERROR** (0x11) 来自非线程的无效等待选项。

允许调用自

初始化、线程、计时器和 ISR（应用程序网络驱动程序）

可以抢占

否

示例

```
/* Append "abcd" to the specified packet. */
status = nx_packet_data_append(packet_ptr, "abcd", 4, &pool_0, 5);

/* If status is NX_SUCCESS, the additional four bytes "abcd" have
    been appended to the packet. */
```

另请参阅

- nx_packet_allocate、nx_packet_copy、nx_packet_data_extract_offset,
- nx_packet_data_retrieve、nx_packet_length_get、nx_packet_pool_create,
- nx_packet_pool_delete、nx_packet_pool_info_get、nx_packet_release,
- nx_packet_transmit_release

nx_packet_data_extract_offset

通过偏移量从数据包中提取数据

原型

```
UINT nx_packet_data_extract_offset(
    NX_PACKET *packet_ptr,
    ULONG offset,
    VOID *buffer_start,
    ULONG buffer_length,
    ULONG *bytes_copied);
```

说明

此服务用于将 NetX 数据包（或数据包链）中的数据复制到指定的缓冲区，该数据从相对于数据包前置指针的指定偏移量开头，并具有指定的大小（以字节为单位）。实际复制的字节数会在 bytes_copied 中返回。此服务不会从该数据包中删除数据，也不会调整前置指针或其他内部状态信息。

参数

- **packet_ptr** 指向要提取的数据包的指针
- **offset** 相对于当前前置指针的偏移量。
- **buffer_start** 指向保存缓冲区开头的指针
- **buffer_length** 要复制的字节数
- **bytes_copied** 实际复制的字节数

返回值

- **NX_SUCCESS** (0x00) 复制数据包成功

- **NX_PACKET_OFFSET_ERROR** (0x53) 提供了无效的偏移量值
- **NX_PTR_ERROR** (0x07) 数据包指针或缓冲区指针无效

允许调用自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
/* Extract 10 bytes from the start of the received packet buffer
    into the specified memory area. */
status = nx_packet_data_extract_offset(my_packet, 0, &data[0], 10,
    &bytes_copied);

/* If status is NX_SUCCESS, 10 bytes were successfully copied into
    the data buffer. */
```

另请参阅

- nx_packet_allocate、nx_packet_copy、nx_packet_data_append,
- nx_packet_data_retrieve、nx_packet_length_get、nx_packet_pool_create,
- nx_packet_pool_delete、nx_packet_pool_info_get、nx_packet_release,
- nx_packet_transmit_release

nx_packet_data_retrieve

从数据包中检索数据

原型

```
UINT nx_packet_data_retrieve(
    NX_PACKET *packet_ptr,
    VOID *buffer_start,
    ULONG *bytes_copied);
```

说明

此服务用于将提供的数据包中的数据复制到提供的缓冲区。复制的实际字节数会在 bytes_copied 所指向的目标中返回。

请注意，此服务不会更改该数据包的内部状态。所检索的数据仍在该数据包中提供。

目标缓冲区的大小必须足以容纳该数据包的内容。否则内存会损坏，导致不可预知的结果。

参数

- **packet_ptr** 指向源数据包的指针。
- **buffer_start** 指向缓冲区开头的指针。
- **bytes_copied** 此指针指向复制的字节数的目标。

返回值

- **NX_SUCCESS** (0x00) 成功检索数据包数据。
- **NX_INVALID_PACKET** (0x12) 数据包无效。
- **NX_PTR_ERROR** (0x07) 数据包指针、缓冲区开头指针或复制字节数指针无效。

允许调用自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
UCHAR buffer[512];
ULONG bytes_copied;

/* Retrieve data from packet pointed to by "packet_ptr". */
status = nx_packet_data_retrieve(packet_ptr, buffer, &bytes_copied);

/* If status is NX_SUCCESS, buffer contains the contents of the
    packet, the size of which is contained in "bytes_copied." */
```

另请参阅

- nx_packet_allocate、nx_packet_copy、nx_packet_data_append,
- nx_packet_data_extract_offset、nx_packet_length_get,
- nx_packet_pool_create、nx_packet_pool_delete,
- nx_packet_pool_info_get、nx_packet_release,
- nx_packet_transmit_release

nx_packet_length_get

获取数据包数据的长度

原型

```
UINT nx_packet_length_get(
    NX_PACKET *packet_ptr,
    ULONG *length);
```

说明

此服务用于获取指定的数据包中的数据长度。

参数

- **packet_ptr** 指向数据包的指针。
- **length** 数据包长度的目标。

允许调用自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
/* Get the length of the data in "my_packet." */
status = nx_packet_length_get(my_packet, &my_length);

/* If status is NX_SUCCESS, data length is in "my_length". */
```

另请参阅

- nx_packet_allocate、nx_packet_copy、nx_packet_data_append,

- nx_packet_data_extract_offset、nx_packet_data_retrieve,
- nx_packet_pool_create、nx_packet_pool_delete,
- nx_packet_pool_info_get、nx_packet_release,
- nx_packet_transmit_release

nx_packet_pool_create

在指定的内存区域创建数据包池

原型

```
UINT nx_packet_pool_create(
    NX_PACKET_POOL *pool_ptr,
    CHAR *name,
    ULONG payload_size,
    VOID *memory_ptr,
    ULONG memory_size);
```

说明

此服务用于在用户提供的内存区域中创建指定的数据包大小的数据包池。

参数

- **pool_ptr** 指向数据包池控制块的指针。
- **name** 指向数据包池的应用程序名称的指针。
- **payload_size** 池中每个数据包的字节数。此值必须至少为 40 个字节，还必须能被 4 整除。
- **memory_ptr** 指向要放置数据包池的内存区域的指针。此指针应该在 ULONG 边界上对齐。
- **memory_size** 池内存区域的大小。

返回值

- **NX_SUCCESS** (0x00) 成功创建数据包池。
- **NX_PTR_ERROR** (0x07) 池指针或内存指针无效。
- **NX_SIZE_ERROR** (0x09) 块大小或内存大小无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Create a packet pool of 32000 bytes starting at physical
    address 0x10000000. */
status = nx_packet_pool_create(&pool_0, "Default Pool", 128,
    (void *) 0x10000000, 32000);

/* If status is NX_SUCCESS, the packet pool has been successfully
    created. */
```

另请参阅

- nx_packet_allocate、nx_packet_copy、nx_packet_data_append,
- nx_packet_data_extract_offset、nx_packet_data_retrieve,
- nx_packet_length_get、nx_packet_pool_delete、nx_packet_pool_info_get,

- nx_packet_release、nx_packet_transmit_release

nx_packet_pool_delete

删除先前创建的数据包池

原型

```
UINT nx_packet_pool_delete(NX_PACKET_POOL *pool_ptr);
```

说明

此服务用于删除先前创建的数据包池。NetX 会检查该数据包池中的数据包上当前是否有任何挂起的线程，并清除挂起。

参数

- **pool_ptr** 数据包池控制块指针。

返回值

- **NX_SUCCESS** (0x00) 成功删除数据包池。
- **NX_PTR_ERROR** (0x07) 池指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

线程数

可以抢占

是

示例

```
/* Delete a previously created packet pool. */
status = nx_packet_pool_delete(&pool_0);

/* If status is NX_SUCCESS, the packet pool has been successfully
    deleted. */
```

另请参阅

- nx_packet_allocate、nx_packet_copy、nx_packet_data_append,
- nx_packet_data_extract_offset、nx_packet_data_retrieve,
- nx_packet_length_get、nx_packet_pool_create,
- nx_packet_pool_info_get、nx_packet_release,
- nx_packet_transmit_release

nx_packet_pool_info_get

检索数据包池的相关信息

原型

```
UINT nx_packet_pool_info_get(
    NX_PACKET_POOL *pool_ptr,
    ULONG *total_packets,
    ULONG *free_packets,
    ULONG *empty_pool_requests,
    ULONG *empty_pool_suspensions,
    ULONG *invalid_packet_releases);
```

说明

此服务用于检索指定的据包池的相关信息。

如果目标指针为 NX_NULL，则不会将该特定信息返回给调用方。

参数

- **pool_ptr** 指向先前创建的数据包池的指针。
- **total_packets** 此指针指向池中数据包总数的目标。
- **free_packets** 此指针指向当前可用数据包总数的目标。
- **empty_pool_requests** 此指针指向该池为空时分配请求总数的目标。
- **empty_pool_suspensions** 此指针指向空池挂起总数的目标。
- **invalid_packet_releases** 此指针指向无效数据包释放总数的目标。

返回值

- **NX_SUCCESS** (0x00) 成功检索数据包池信息。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程和计时器

可以抢占

否

示例

```
/* Retrieve packet pool information. */
status = nx_packet_pool_info_get(&pool_0,
    &total_packets,
    &free_packets,
    &empty_pool_requests,
    &empty_pool_suspensions,
    &invalid_packet_releases);

/* If status is NX_SUCCESS, packet pool information was
    retrieved. */
```

另请参阅

- nx_packet_allocate、nx_packet_copy、nx_packet_data_append,
- nx_packet_data_extract_offset、nx_packet_data_retrieve,
- nx_packet_length_get、nx_packet_pool_create、nx_packet_pool_delete
- nx_packet_release、nx_packet_transmit_release

nx_packet_release

释放先前分配的数据包

原型

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

说明

此服务用于释放数据包，包括链接到指定数据包的任何其他数据包。如有其他线程在分配数据包时遭到止，则该线程会获得该数据包并继续执行。

应用程序必须防止多次释放同一数据包，否则会导致不可预知的结果。

参数

- **packet_ptr** 数据包指针。

返回值

- **NX_SUCCESS** (0x00) 成功释放数据包。
- **NX_PTR_ERROR** (0x07) 数据包指针无效。
- **NX_UNDERFLOW** (0x02) 前置指针小于有效负载开始位置。
- **NX_OVERFLOW** (0x03) 追加指针大于有效负载结束位置。

允许调用自

初始化、线程、计时器和 ISR（应用程序网络驱动程序）

可以抢占

是

示例

```
/* Release a previously allocated packet. */
status = nx_packet_release(packet_ptr);

/* If status is NX_SUCCESS, the packet has been returned to the
    packet pool it was allocated from. */
```

另请参阅

- nx_packet_allocate、nx_packet_copy、nx_packet_data_append,
- nx_packet_data_extract_offset、nx_packet_data_retrieve,
- nx_packet_length_get、nx_packet_pool_create、nx_packet_pool_delete,
- nx_packet_pool_info_get、nx_packet_transmit_release

nx_packet_transmit_release

释放已传输的数据包

原型

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

说明

对于非 TCP 数据包，此服务可释放已传输的数据包，包括链接到指定数据包的任何其他数据包。如有其他线程在分配数据包时遭到止，则该线程会获得该数据包并继续执行。对于传输的 TCP 数据包，该数据包会标记为正在传输，直到该数据包获得认可后才会释放。通常，在传输数据包之后，会从应用程序的网络驱动程序中调用此服务。

在调用此服务之前，网络驱动程序应删除物理介质标头，并调整数据包的长度。

参数

- **packet_ptr** 数据包指针。

返回值

- **NX_SUCCESS** (0x00) 成功释放传输数据包。
- **NX_PTR_ERROR** (0x07) 数据包指针无效。
- **NX_UNDERFLOW** (0x02) 前置指针小于有效负载开始位置。
- **NX_OVERFLOW** (0x03) 追加指针大于有效负载结束位置。

允许调用自

初始化、线程、计时器、应用程序网络驱动程序（包括 ISR）

可以抢占

是

示例

```
/* Release a previously allocated packet that was just transmitted
    from the application network driver. */
status = nx_packet_transmit_release(packet_ptr);

/* If status is NX_SUCCESS, the transmitted packet has been
    returned to the packet pool it was allocated from. */
```

另请参阅

- nx_packet_allocate、nx_packet_copy、nx_packet_data_append,
- nx_packet_data_extract_offset、nx_packet_data_retrieve,
- nx_packet_length_get、nx_packet_pool_create、nx_packet_pool_delete,
- nx_packet_pool_info_get、nx_packet_release

nx_rarp_disable

禁用反向地址解析协议 (RARP)

原型

```
UINT nx_rarp_disable(NX_IP *ip_ptr);
```

说明

此服务用于对特定的 IP 实例禁用 NetX 的 RARP 组件。对于多宿主系统，此服务会在所有接口上禁用 RARP。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功禁用 RARP。
- **NX_NOT_ENABLED** (0x14) RARP 未启用。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Disable RARP on the previously created IP instance. */
status = nx_rarp_disable(&ip_0);

/* If status is NX_SUCCESS, RARP is disabled. */
```

另请参阅

- nx_rarp_enable、nx_rarp_info_get

nx_rarp_enable

启用反向地址解析协议 (RARP)

原型

```
UINT nx_rarp_enable(NX_IP *ip_ptr);
```

说明

此服务用于对特定的 IP 实例启用 NetX 的 RARP 组件。RARP 组件会通过所有已附加的网络接口搜索零 IP 地址。零 IP 地址表示该接口还没有 IP 地址分配。RARP 会尝试通过在该接口上启用 RARP 进程来解析 IP 地址。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功启用 RARP。
- **NX_IP_ADDRESS_ERROR** (0x21) IP 地址已有效。
- **NX_ALREADY_ENABLED** (0x15) 已启用 RARP。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程、计时器

可以抢占

否

示例

```
/* Enable RARP on the previously created IP instance. */
status = nx_rarp_enable(&ip_0);

/* If status is NX_SUCCESS, RARP is enabled and is attempting to
    resolve this IP instance’s address by querying the network. */
```

另请参阅

- nx_rarp_disable、nx_rarp_info_get

nx_rarp_info_get

检索 RARP 活动的相关信息

原型

```
UINT nx_rarp_info_get(
    NX_IP *ip_ptr,
    ULONG *rarp_requests_sent,
    ULONG *rarp_responses_received,
    ULONG *rarp_invalid_messages);
```

说明

此服务会检索指定的 IP 实例的 RARP 活动相关信息。

如果目标指针为 NX_NULL，则不会将该特定信息返回给调用方。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **rarp_requests_sent** 此指针指向已发送的 RARP 请求总数的目标。
- **rarp_responses_received** 此指针指向已收到的 RARP 响应总数的目标。
- **rarp_invalid_messages** 此指针指向无效消息总数的目标。

返回值

- **NX_SUCCESS** (0x00) 成功检索 RARP 信息。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Retrieve RARP information from previously created IP
    Instance 0. */
status = nx_rarp_info_get(&ip_0,
    &rarp_requests_sent,
    &rarp_responses_received,
    &rarp_invalid_messages);

/* If status is NX_SUCCESS, RARP information was retrieved. */
```

另请参阅

- nx_rarp_disable、nx_rarp_enable

nx_system_initialize

初始化 NetX 系统

原型

```
VOID nx_system_initialize(VOID);
```

说明

此服务用于初始化基本的 NetX 系统资源以备使用。应用程序应在初始化期间和进行任何其他 NetX 调用之前调

用此服务。

参数

无

返回值

无

允许调用自

初始化、线程、计时器、ISR

可以抢占

否

示例

```
/* Initialize NetX for operation. */
nx_system_initialize();

/* At this point, NetX is ready for IP creation and all subsequent
    network operations. */
```

另请参阅

- nx_ip_address_change_notify、nx_ip_address_get、nx_ip_address_set,
- nx_ip_create、nx_ip_delete、nx_ip_driver_direct_command,
- nx_ip_driver_interface_direct_command、nx_ip_forwarding_disable,
- nx_ip_forwarding_enable、nx_ip_fragment_disable,
- nx_ip_fragment_enable、x_ip_info_get、x_ip_status_check

nx_tcp_client_socket_bind

将客户端 TCP 套接字与 TCP 端口绑定

原型

```
UINT nx_tcp_client_socket_bind(
    NX_TCP_SOCKET *socket_ptr,
    UINT port, ULONG wait_option);
```

说明

此服务用于将先前创建的 TCP 客户端套接字与指定的 TCP 端口绑定。有效的 TCP 套接字介于 0 到 0xFFFF 之间。如果指定的 TCP 端口不可用，此服务会根据提供的等待选项挂起。

参数

- **socket_ptr** 此指针指向先前创建的 TCP 套接字实例。
- **port** 要绑定的端口号（1 到 0xFFFF）。如果端口号为 NX_ANY_PORT (0x0000)，IP 实例会搜索下一个可用端口并将其用于绑定。
- **wait_option** 定义该端口已与另一套接字绑定时此服务的行为方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) 成功绑定套接字。

- **NX_ALREADY_BOUND** (0x22) 套接字已与另一 TCP 端口绑定。
- **NX_PORT_UNAVAILABLE** (0x23) 端口已与其他套接字绑定。
- **NX_NO_FREE_PORTS** (0x45) 没有可用的端口。
- **NX_WAIT_ABORTED** (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。
- **NX_INVALID_PORT** (0x46) 端口无效。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
/* Bind a previously created client socket to port 12 and wait for 7
    timer ticks for the bind to complete. */
status = nx_tcp_client_socket_bind(&client_socket, 12, 7);

/* If status is NX_SUCCESS, the previously created client_socket is
    bound to port 12 on the associated IP instance. */
```

另请参阅

- nx_tcp_client_socket_connect、x_tcp_client_socket_port_get,
- nx_tcp_client_socket_unbind、nx_tcp_enable、nx_tcp_free_port_find,
- nx_tcp_info_get、nx_tcp_server_socket_accept,
- nx_tcp_server_socket_listen、nx_tcp_server_socket_relisten,
- nx_tcp_server_socket_unaccept、nx_tcp_server_socket_unlisten,
- nx_tcp_socket_bytes_available、nx_tcp_socket_create,
- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_client_socket_connect

连接客户端 TCP 套接字

原型

```
UINT nx_tcp_client_socket_connect(
    NX_TCP_SOCKET *socket_ptr,
    ULONG server_ip,
    UINT server_port,
    ULONG wait_option);
```

说明

此服务用于将先前创建并绑定的 TCP 客户端套接字连接到指定的服务器端口。有效的 TCP 服务器端口介于 0 到 0xFFFF 之间。如果连接无法立即完成，该服务会根据提供的等待选项挂起。

参数

- socket_ptr 此指针指向先前创建的 TCP 套接字实例。
- server_ip 服务器的 IP 地址。
- server_port 要连接到的服务器端口号（1 到 0xFFFF）。
- wait_option 定义建立该连接时此服务的行为方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- NX_SUCCESS (0x00) 成功连接套接字。
- NX_NOT_BOUND (0x24) 未绑定套接字。
- NX_NOT_CLOSED (0x35) 套接字未处于关闭状态。
- NX_IN_PROGRESS (0x37) 未指定等待，正在尝试连接。
- NX_INVALID_INTERFACE (0x4C) 提供了无效的接口。
- NX_WAIT_ABORTED (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。
- NX_IP_ADDRESS_ERROR (0x21) 服务器 IP 地址无效。
- NX_INVALID_PORT (0x46) 端口无效。
- NX_PTR_ERROR (0x07) 套接字指针无效。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_NOT_ENABLED (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
/* Initiate a TCP connection from a previously created and bound
    client socket. The connection requested in this example is to
    port 12 on the server with the IP address of 1.2.3.5. This
    service will wait 300 timer ticks for the connection to take
    place before giving up. */
status = nx_tcp_client_socket_connect(&client_socket,
    IP_ADDRESS(1,2,3,5), 12, 300);

/* If status is NX_SUCCESS, the previously created and bound
    client_socket is connected to port 12 on IP 1.2.3.5. */
```

另请参阅

- nx_tcp_client_socket_bind、x_tcp_client_socket_port_get,
- nx_tcp_client_socket_unbind、nx_tcp_enable、nx_tcp_free_port_find,
- nx_tcp_info_get、nx_tcp_server_socket_accept,
- nx_tcp_server_socket_listen、nx_tcp_server_socket_relisten,
- nx_tcp_server_socket_unaccept、nx_tcp_server_socket_unlisten,
- nx_tcp_socket_bytes_available、nx_tcp_socket_create,
- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、_tcp_socket_receive
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_client_socket_port_get

获取与客户端 TCP 套接字绑定的端口号

原型

```
UINT nx_tcp_client_socket_port_get(
    NX_TCP_SOCKET *socket_ptr,
    UINT *port_ptr);
```

说明

此服务会检索与套接字关联的端口号；如果在绑定套接字时指定了 NX_ANY_PORT，它对查找 NetX 分配的端口非常有用。

参数

- **socket_ptr** 此指针指向先前创建的 TCP 套接字实例。
- **port_ptr** 此指针指向返回端口号的目标。有效端口号介于 1 到 0xFFFF 之间。

返回值

- **NX_SUCCESS** (0x00) 成功绑定套接字。
- **NX_NOT_BOUND** (0x24) 此套接字未与任何端口绑定。
- **NX_PTR_ERROR** (0x07) 套接字指针或端口返回指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
/* Get the port number of previously created and bound client
    socket. */
status = nx_tcp_client_socket_port_get(&client_socket, &port);

/* If status is NX_SUCCESS, the port variable contains the port this
    socket is bound to. */
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_unbind、nx_tcp_enable、nx_tcp_free_port_find,
- nx_tcp_info_get、nx_tcp_server_socket_accept,
- nx_tcp_server_socket_listen、nx_tcp_server_socket_relisten,
- nx_tcp_server_socket_unaccept、nx_tcp_server_socket_unlisten,
- nx_tcp_socket_bytes_available、nx_tcp_socket_create,
- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_client_socket_unbind

取消 TCP 客户端套接字与 TCP 端口之间的绑定

原型

```
UINT nx_tcp_client_socket_unbind(NX_TCP_SOCKET *socket_ptr);
```

说明

此服务用于解除 TCP 客户端套接字与 TCP 端口之间的绑定。如果有其他线程在等待将其他套接字绑定到同一端口号，则第一个挂起的线程会与该端口绑定。

参数

- **socket_ptr** 此指针指向先前创建的 TCP 套接字实例。

返回值

- **NX_SUCCESS** (0x00) 成功取消绑定套接字。
- **NX_NOT_BOUND** (0x24) 套接字未与任何端口绑定。
- **NX_NOT_CLOSED** (0x35) 套接字尚未断开连接。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

是

示例

```
/* Unbind a previously created and bound client TCP socket. */
status = nx_tcp_client_socket_unbind(&client_socket);

/* If status is NX_SUCCESS, the client socket is no longer
    bound. */
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、x_tcp_enable、x_tcp_free_port_find,
- nx_tcp_info_get、nx_tcp_server_socket_accept,
- nx_tcp_server_socket_listen、nx_tcp_server_socket_relisten,
- nx_tcp_server_socket_unaccept、nx_tcp_server_socket_unlisten,
- nx_tcp_socket_bytes_available、nx_tcp_socket_create,
- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_enable

启用 NetX 的 TCP 组件

原型

```
UINT nx_tcp_enable(NX_IP *ip_ptr);
```

说明

此服务用于启用 NetX 的传输控制协议 (TCP) 组件。启用后，应用程序可建立 TCP 连接。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功启用 TCP。
- **NX_ALREADY_ENABLED** (0x15) 已启用 TCP。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

初始化、线程、计时器

可以抢占

否

示例

```
/* Enable TCP on a previously created IP instance ip_0. */
status = nx_tcp_enable(&ip_0);

/* If status is NX_SUCCESS, TCP is enabled on the IP instance. */
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_free_port_find、nx_tcp_info_get、nx_tcp_server_socket_accept,
- nx_tcp_server_socket_listen、nx_tcp_server_socket_relisten,
- nx_tcp_server_socket_unaccept、nx_tcp_server_socket_unlisten,
- nx_tcp_socket_bytes_available、nx_tcp_socket_create,
- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_free_port_find

查找下一个可用的 TCP 端口

原型

```
UINT nx_tcp_free_port_find(
    NX_IP *ip_ptr,
    UINT port, UINT *free_port_ptr);
```

说明

此服务尝试从应用程序提供的端口开始查找可用未绑定的 TCP 端口。如果搜索恰好达到最大端口值 0xFFFF，搜索逻辑会从头开始。如果搜索成功，则会在 free_port_ptr 所指向的变量中返回可用端口。

可从其他线程调用此服务，此时会返回同一端口。若要防止出现这种争用情况，应用程序可能希望将此服务和实际的客户端套接字绑定置于互斥体的保护之下。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **port** 充当搜索起点的端口号（1 到 0xFFFF）。
- **free_port_ptr** 指向目标可用端口返回值的指针。

返回值

- **NX_SUCCESS** (0x00) 成功找到可用端口。
- **NX_NO_FREE_PORTS** (0x45) 找不到可用端口。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。
- **NX_INVALID_PORT** (0x46) 指定的端口号无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Locate a free TCP port, starting at port 12, on a previously
    created IP instance. */
status = nx_tcp_free_port_find(&ip_0, 12, &free_port);

/* If status is NX_SUCCESS, "free_port" contains the next free port
    on the IP instance. */
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_info_get、nx_tcp_server_socket_accept,
- nx_tcp_server_socket_listen、nx_tcp_server_socket_relisten,
- nx_tcp_server_socket_unaccept、nx_tcp_server_socket_unlisten,
- nx_tcp_socket_bytes_available、nx_tcp_socket_create,
- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_info_get

检索 TCP 活动的相关信息

原型

```
UINT nx_tcp_info_get(
    NX_IP *ip_ptr,
    ULONG *tcp_packets_sent,
    ULONG *tcp_bytes_sent,
    ULONG *tcp_packets_received,
    ULONG *tcp_bytes_received,
    ULONG *tcp_invalid_packets,
    ULONG *tcp_receive_packets_dropped,
    ULONG *tcp_checksum_errors,
    ULONG *tcp_connections,
    ULONG *tcp_disconnections,
    ULONG *tcp_connections_dropped,
    ULONG *tcp_retransmit_packets);
```

说明

此服务会检索指定的 IP 实例的 TCP 活动相关信息。

如果目标指针为 NX_NULL，则不会将该特定信息返回给调用方。

参数

- ip_ptr 先前创建的 IP 实例的指针。
- tcp_packets_sent 此指针指向已发送的 TCP 数据包总数的目标。
- tcp_bytes_sent 此指针指向已发送的 TCP 总字节数的目标。
- tcp_packets_received 此指针指向已接收的 TCP 数据包总数的目标。
- tcp_bytes_received 此指针指向已接收的 TCP 总字节数的目标。
- tcp_invalid_packets 此指针指向无效 TCP 数据包总数的目标。
- tcp_receive_packets_dropped 此指针指向已丢弃的 TCP 接收数据包总数的目标。
- tcp_checksum_errors 此指针指向有校验和错误的 TCP 数据包总数的目标。
- tcp_connections 此指针指向 TCP 连接总数的目标。
- tcp_disconnections 此指针指向 TCP 断开连接总数的目标。
- tcp_connections_dropped 此指针指向已丢弃的 TCP 连接总数的目标。
- tcp_retransmit_packets 此指针指向重新传输的 TCP 数据包总数的目标。

返回值

- NX_SUCCESS (0x00) 成功检索 TCP 信息。
- NX_PTR_ERROR (0x07) IP 指针无效。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_NOT_ENABLED (0x14) 尚未启用此组件。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Retrieve TCP information from previously created IP Instance ip_0. */
status = nx_tcp_info_get(&ip_0,
    &tcp_packets_sent,
    &tcp_bytes_sent,
    &tcp_packets_received,
    &tcp_bytes_received,
    &tcp_invalid_packets,
    &tcp_receive_packets_dropped,
    &tcp_checksum_errors,
    &tcp_connections,
    &tcp_disconnections
    &tcp_connections_dropped,
    &tcp_retransmit_packets);

/* If status is NX_SUCCESS, TCP information was retrieved. */
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_server_socket_accept,
- nx_tcp_server_socket_listen、nx_tcp_server_socket_relisten,
- nx_tcp_server_socket_unaccept、nx_tcp_server_socket_unlisten,
- nx_tcp_socket_bytes_available、nx_tcp_socket_create,
- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_server_socket_accept

接受 TCP 连接

原型

```
UINT nx_tcp_server_socket_accept(
    NX_TCP_SOCKET *socket_ptr,
    ULONG wait_option);
```

说明

此服务用于接受(或准备接受)针对先前设置进行侦听的端口的 TCP 客户端套接字连接请求。在应用程序调用侦听或重新侦听服务之后，或者在客户端连接实际存在时调用侦听回调例程之后，可以立即调用此服务。如果无法立即建立连接，此服务会根据提供的等待选项挂起。

不再需要该连接之后，应用程序必须调用 nx_tcp_server_socket_unaccept，以删除服务器套接字与服务器端口的绑定。

应用程序回调例程是从 IP 的帮助器线程中调用的。

参数

- **socket_ptr** 指向 TCP 服务器套接字控制块的指针。
- **wait_option** 定义建立该连接时此服务的行为方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值(0x00000001 到 0xFFFFFFFE)

返回值

- **NX_SUCCESS** (0x00) 成功接受 TCP 服务器套接字(被动连接)。
- **NX_NOT_LISTEN_STATE** (0x36) 提供的服务器套接字未处于侦听状态。
- **NX_IN_PROGRESS** (0x37) 未指定等待，正在尝试连接。
- **NX_WAIT_ABORTED** (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。
- **NX_PTR_ERROR** (0x07) 套接字指针错误。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

初始化、线程

可以抢占

否

示例

```
NX_PACKET_POOL my_pool;
NX_IP my_ip;
NX_TCP_SOCKET server_socket;
void port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{
    /* Simply set the semaphore to wake up the server thread. */
    tx_semaphore_put(&port_12_semaphore);
}

void port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This
    example doesn't use this callback. */
}

void port_12_server_thread_entry(ULONG id)
{
    NX_PACKET *my_packet;
    UINT status, i;
    /* Assuming that:
        "port_12_semaphore" has already been created with an
        initial count of 0 "my_ip" has already been created and the
        link is enabled "my_pool" packet pool has already been
        created */

    /* Create the server socket. */
    nx_tcp_socket_create(&my_ip, &server_socket,
        "Port 12 Server Socket",
        NX_IP_NORMAL, NX_FRAGMENT_OKAY,
        NX_IP_TIME_TO_LIVE, 100,
        NX_NULL, port_12_disconnect_request);

    /* Setup server listening on port 12. */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
        port_12_connect_request);

    /* Loop to process 5 server connections, sending
        "Hello_and_Goodbye" to each client and then disconnecting.*/
    for (i = 0; i < 5; i++)
    {
        /* Get the semaphore that indicates a client connection
            request is present. */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

        /* Wait for 200 ticks for the client socket connection to
            complete.*/
```

```
        status = nx_tcp_server_socket_accept(&server_socket, 200);

        /* Check for a successful connection. */
        if (status == NX_SUCCESS)
        {
            /* Allocate a packet for the "Hello_and_Goodbye"
                message */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                NX_WAIT_FOREVER);

            /* Place "Hello_and_Goodbye" in the packet. */
            nx_packet_data_append(my_packet, "Hello_and_Goodbye",
                sizeof("Hello_and_Goodbye"),
                &my_pool, NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client. */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error. */
            if (status)
            {
                /* Error, release the packet. */
                nx_packet_release(my_packet);
            }

            /* Now disconnect the server socket from the client. */
            nx_tcp_socket_disconnect(&server_socket, 200);
        }

        /* Unaccept the server socket. Note that unaccept is called
            even if disconnect or accept fails. */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket
            again. */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }

    /* We are now done so unlisten on server port 12. */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket. */
    nx_tcp_socket_delete(&server_socket);
}
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_info_get,
- nx_tcp_server_socket_listen、nx_tcp_server_socket_relisten,
- nx_tcp_server_socket_unaccept、nx_tcp_server_socket_unlisten,
- nx_tcp_socket_bytes_available、nx_tcp_socket_create,
- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_server_socket_listen

允许侦听 TCP 端口上的客户端连接

原型

```
UINT nx_tcp_server_socket_listen(
    NX_IP *ip_ptr,
    UINT port,
    NX_TCP_SOCKET *socket_ptr,
    UINT listen_queue_size,
    VOID (*listen_callback)(NX_TCP_SOCKET *socket_ptr, UINT port));
```

说明

通过此服务可侦听指定的 TCP 端口上的客户端连接请求。接收到客户端连接请求时，提供的服务器套接字会绑定到指定的端口，并调用所提供的侦听回调函数。

侦听回调例程的处理完全由应用程序决定。这可能包含唤醒应用程序线程来便于随后执行接受操作的逻辑。如果应用程序已有线程在接受此套接字的处理时挂起，则可能不需要侦听回调例程。

如果应用程序希望在同一端口上处理其他客户端连接，则必须使用可用的套接字（处于关闭状态的套接字）调用 nx_tcp_server_socket_relisten，以建立下一个连接。在调用重新侦听服务之前，其他客户端连接会进行排队。超过最大队列深度时，会丢弃最早的连接请求，而将新连接请求排入队列。最大队列深度由此服务指定。

应用程序回调例程是从内部 IP 帮助器线程中调用的。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **port** 要侦听的端口号（1 到 0xFFFF）。
- **socket_ptr** 此指针指向用于连接的套接字。
- **listen_queue_size** 可排入队列的客户端连接请求数。
- **listen_callback** 接收到连接时要调用的应用程序函数。如果指定了 NULL，则会禁用侦听回调功能。

返回值

- **NX_SUCCESS** (0x00) 成功启用 TCP 端口侦听。
- **NX_MAX_LISTEN** (0x33) 没有更多的侦听请求结构可供使用。nx_api.h 中的常量 NX_MAX_LISTEN_REQUESTS 定义了活动侦听请求数上限。
- **NX_NOT_CLOSED** (0x35) 提供的服务器套接字未处于关闭状态。
- **NX_ALREADY_BOUND** (0x22) 提供的服务器套接字已与某个端口绑定。
- **NX_DUPLICATE_LISTEN** (0x34) 此端口已有活动的侦听请求。
- **NX_INVALID_PORT** (0x46) 指定的端口无效。
- **NX_PTR_ERROR** (0x07) IP 或套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
NX_PACKET_POOL my_pool;
NX_IP my_ip;
NX_TCP_SOCKET server_socket;

void port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{
    /* Simply set the semaphore to wake up the server thread.*/
    tx_semaphore_put(&port_12_semaphore);
}
```

```
}

void port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket.
        This example doesn't use this callback. */
}

void port_12_server_thread_entry(ULONG id)
{
    NX_PACKET *my_packet;
    UINT status, i;

    /* Assuming that:
        "port_12_semaphore" has already been created with an
        initial count of 0 "my_ip" has already been created
        and the link is enabled "my_pool" packet pool has already
        been created.
    */

    /* Create the server socket. */
    nx_tcp_socket_create(&my_ip, &server_socket, "Port 12 Server
        Socket",
        NX_IP_NORMAL, NX_FRAGMENT_OKAY,
        NX_IP_TIME_TO_LIVE, 100,
        NX_NULL, port_12_disconnect_request);

    /* Setup server listening on port 12. */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
        port_12_connect_request);

    /* Loop to process 5 server connections, sending
        "Hello_and_Goodbye" to
        each client and then disconnecting. */
    for (i = 0; i < 5; i++)
    {
        /* Get the semaphore that indicates a client connection
            request is present. */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

    /* Wait for 200 ticks for the client socket connection
        to complete. */
    status = nx_tcp_server_socket_accept(&server_socket, 200);

    /* Check for a successful connection. */
    if (status == NX_SUCCESS)
    {
        /* Allocate a packet for the "Hello_and_Goodbye"
            message. */
        nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
            NX_WAIT_FOREVER);

        /* Place "Hello_and_Goodbye" in the packet. */
        nx_packet_data_append(my_packet, "Hello_and_Goodbye",
            sizeof("Hello_and_Goodbye"),
            &my_pool,
            NX_WAIT_FOREVER);

        /* Send "Hello_and_Goodbye" to client. */
        nx_tcp_socket_send(&server_socket, my_packet, 200);

        /* Check for an error. */
        if (status)
        {
            /* Error, release the packet. */
            nx_packet_release(my_packet);
        }
        /* Now disconnect the server socket from the client. */
        nx_tcp_socket_disconnect(&server_socket, 200);
    }
```

```
    }

    /* Unaccept the server socket. Note that unaccept is called
        even if disconnect or accept fails. */
    nx_tcp_server_socket_unaccept(&server_socket);

    /* Setup server socket for listening with this socket
    again. */
    nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
}

/* We are now done so unlisten on server port 12. */
nx_tcp_server_socket_unlisten(&my_ip, 12);

/* Delete the server socket. */
nx_tcp_socket_delete(&server_socket);
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_info_get,
- nx_tcp_server_socket_accept、nx_tcp_server_socket_relisten,
- nx_tcp_server_socket_unaccept、nx_tcp_server_socket_unlisten,
- nx_tcp_socket_bytes_available、nx_tcp_socket_create,
- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_server_socket_relisten

重新侦听 TCP 端口上的客户端连接

原型

```
UINT nx_tcp_server_socket_relisten(
    NX_IP *ip_ptr,
    UINT port,
    NX_TCP_SOCKET *socket_ptr);
```

说明

在先前已设置进行侦听的端口上接收到连接之后，会调用此服务。此服务的主要目的是，提供新的服务器套接字用于下一个客户端连接。如果已有连接请求在排队，调用此服务期间会立即处理该连接。

存在这个新服务器套接字的连接时，也会调用原始侦听请求所指定的回调例程。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **port** 要重新侦听的端口号(1 到 0xFFFF)。
- **socket_ptr** 要用于下一客户端连接的套接字。

返回值

- **NX_SUCCESS** (0x00) 成功重新侦听 TCP 端口。
- **NX_NOT_CLOSED** (0x35) 提供的服务器套接字未处于关闭状态。
- **NX_ALREADY_BOUND** (0x22) 提供的服务器套接字已与某个端口绑定。
- **NX_INVALID_RELISTEN** (0x47) 此端口已有一个有效的套接字指针，或者指定的端口没有活动的侦听请求。

- **NX_CONNECTION_PENDING** (0x48) 与 NX_SUCCESS 相同，区别是存在已排队的连接请求，并且已在此调用期间处理该请求。
- **NX_INVALID_PORT** (0x46) 指定的端口无效。
- **NX_PTR_ERROR** (0x07) IP 或侦听回调指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
NX_PACKET_POOL my_pool;
NX_IP my_ip;
NX_TCP_SOCKET server_socket;

void port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{
    /* Simply set the semaphore to wake up the server thread.*/
    tx_semaphore_put(&port_12_semaphore);
    }

void port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This
    example doesn't use this callback. */
}

void port_12_server_thread_entry(ULONG id)
{
    NX_PACKET *my_packet;
    UINT status, i;

    /* Assuming that:
    "port_12_semaphore" has already been created with an initial
        count of 0.
        "my_ip" has already been created and the link is enabled.
        "my_pool" packet pool has already been created. */

    /* Create the server socket. */
    nx_tcp_socket_create(&my_ip, &server_socket,
        "Port 12 Server Socket",
        NX_IP_NORMAL, NX_FRAGMENT_OKAY,
        NX_IP_TIME_TO_LIVE, 100,
        NX_NULL,
        port_12_disconnect_request);

    /* Setup server listening on port 12. */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
        port_12_connect_request);
    /* Loop to process 5 server connections, sending
        "Hello_and_Goodbye" to each client then disconnecting. */
    for (i = 0; i < 5; i++)
    {
        /* Get the semaphore that indicates a client connection
        request is present. */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

        /* Wait for 200 ticks for the client socket connection to
        complete. */
        status = nx_tcp_server_socket_accept(&server_socket, 200);
```

```
        /* Check for a successful connection. */
        if (status == NX_SUCCESS)
        {
            /* Allocate a packet for the "Hello_and_Goodbye"
            message. */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                NX_WAIT_FOREVER);

            /* Place "Hello_and_Goodbye" in the packet. */
            nx_packet_data_append(my_packet, "Hello_and_Goodbye",
                sizeof("Hello_and_Goodbye"),
                &my_pool, NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client. */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error. */
            if (status)
            {
                /* Error, release the packet. */
                nx_packet_release(my_packet);
            }

            /* Now disconnect the server socket from the client. */
            nx_tcp_socket_disconnect(&server_socket, 200);
        }

        /* Unaccept the server socket. Note that unaccept is
        called even if disconnect or accept fails. */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket
        again. */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }
    /* We are now done so unlisten on server port 12. */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket. */
    nx_tcp_socket_delete(&server_socket);
}
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_info_get,
- nx_tcp_server_socket_accept、nx_tcp_server_socket_listen,
- nx_tcp_server_socket_unaccept、nx_tcp_server_socket_unlisten,
- nx_tcp_socket_bytes_available、nx_tcp_socket_create,
- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_server_socket_unaccept

删除套接字与侦听端口的关联

原型

```
UINT nx_tcp_server_socket_unaccept(NX_TCP_SOCKET *socket_ptr);
```

说明

此服务用于删除此服务器套接字与指定的服务器端口之间的关联。断开连接后或在调用接受失败后，应用程序必须调用此服务。

参数

- **socket_ptr** 此指针指向先前设置的服务器套接字实例。

返回值

- **NX_SUCCESS** (0x00) 成功取消接受服务器套接字。
- **NX_NOT_LISTEN_STATE** (0x36) 服务器套接字状态不正确，可能未断开连接。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
NX_PACKET_POOL my_pool;
NX_IP my_ip;
NX_TCP_SOCKET server_socket;

void port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{
    /* Simply set the semaphore to wake up the server thread. */
    tx_semaphore_put(&port_12_semaphore);
}

void port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This example
    doesn't use this callback. */
}

void port_12_server_thread_entry(ULONG id)
{
    NX_PACKET *my_packet;
    UINT status, i;

    /* Assuming that:
        "port_12_semaphore" has already been created with an initial count
        of 0 "my_ip" has already been created and the link is enabled
        "my_pool" packet pool has already been created
        */

    /* Create the server socket. */
    nx_tcp_socket_create(&my_ip, &server_socket,
        "Port 12 Server Socket",NX_IP_NORMAL, NX_FRAGMENT_OKAY,
        NX_IP_TIME_TO_LIVE, 100,NX_NULL,
        port_12_disconnect_request);

    /* Setup server listening on port 12. */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
        port_12_connect_request);
```

```
    /* Loop to process 5 server connections, sending "Hello_and_Goodbye"
        to each client and then disconnecting. */
    for (i = 0; i < 5; i++)
    {
        /* Get the semaphore that indicates a client connection request
            is present. */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

        /* Wait for 200 ticks for the client socket connection to
            complete.*/
        status = nx_tcp_server_socket_accept(&server_socket, 200);

        /* Check for a successful connection. */
        if (status == NX_SUCCESS)
        {
            /* Allocate a packet for the "Hello_and_Goodbye" message. */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                NX_WAIT_FOREVER);

            /* Place "Hello_and_Goodbye" in the packet. */
            nx_packet_data_append(my_packet,
                "Hello_and_Goodbye",sizeof("Hello_and_Goodbye"),
                &my_pool, NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client. */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error. */
            if (status)
            {
                /* Error, release the packet. */
                nx_packet_release(my_packet);
            }

            /* Now disconnect the server socket from the client. */
            nx_tcp_socket_disconnect(&server_socket, 200);
        }

        /* Unaccept the server socket. Note that unaccept is called even
            if disconnect or accept fails. */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket again. */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }
    /* We are now done so unlisten on server port 12. */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket. */
    nx_tcp_socket_delete(&server_socket);
}
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind、nx_tcp_enable,
- nx_tcp_free_port_find、nx_tcp_info_get、nx_tcp_server_socket_accept,
- nx_tcp_server_socket_listen、nx_tcp_server_socket_relisten,
- nx_tcp_server_socket_unlisten、nx_tcp_socket_bytes_available,
- nx_tcp_socket_create、nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_server_socket_unlisten

禁止侦听 TCP 端口上的客户端连接

原型

```
UINT nx_tcp_server_socket_unlisten(NX_IP *ip_ptr, UINT port);
```

说明

此服务禁止侦听指定的 TCP 端口上的客户端连接请求。

参数

- ip_ptr 先前创建的 IP 实例的指针。
- port 要禁止侦听的端口号(0 到 0xFFFF)。

返回值

- NX_SUCCESS (0x00) 成功禁用 TCP 侦听。
- NX_ENTRY_NOT_FOUND (0x16) 指定的端口未启用侦听。
- NX_INVALID_PORT (0x46) 指定的端口无效。
- NX_PTR_ERROR (0x07) IP 指针无效。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_NOT_ENABLED (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
NX_PACKET_POOL my_pool;
NX_IP my_ip;
NX_TCP_SOCKET server_socket;

void port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{
    /* Simply set the semaphore to wake up the server thread. */
    tx_semaphore_put(&port_12_semaphore);
}

void port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This example
    doesn't use this callback.*/
}

void port_12_server_thread_entry(ULONG id)
{
    NX_PACKET *my_packet;
    UINT status, i;

    /* Assuming that:
        "port_12_semaphore" has already been created with an initial count
        of 0 "my_ip" has already been created and the link is enabled
        "my_pool" packet pool has already been created
    */

    /* Create the server socket. */
    nx_tcp_socket_create(&my_ip, &server_socket, "Port 12 Server Socket",
```

```
        NX_IP_NORMAL, NX_FRAGMENT_OKAY,
        NX_IP_TIME_TO_LIVE, 100,
        NX_NULL, port_12_disconnect_request);

    /* Setup server listening on port 12. */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
        port_12_connect_request);

    /* Loop to process 5 server connections, sending "Hello_and_Goodbye" to
        each client and then disconnecting. */
    for (i = 0; i < 5; i++)
    {
        /* Get the semaphore that indicates a client connection request is
            present. */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

        /* Wait for 200 ticks for the client socket connection to complete.*/
        status = nx_tcp_server_socket_accept(&server_socket, 200);

        /* Check for a successful connection. */
        if (status == NX_SUCCESS)
        {
            /* Allocate a packet for the "Hello_and_Goodbye" message. */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                NX_WAIT_FOREVER);

            /* Place "Hello_and_Goodbye" in the packet. */
            nx_packet_data_append(my_packet, "Hello_and_Goodbye",
                sizeof("Hello_and_Goodbye"), &my_pool,
                NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client. */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error. */
            if (status)
            {
                /* Error, release the packet. */
                nx_packet_release(my_packet);
            }
            /* Now disconnect the server socket from the client. */
            nx_tcp_socket_disconnect(&server_socket, 200);
        }
        /* Unaccept the server socket. Note that unaccept is called even if
        disconnect or accept fails. */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket again. */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }
    /* We are now done so unlisten on server port 12. */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket. */
    nx_tcp_socket_delete(&server_socket);
}
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_info_get,
- nx_tcp_server_socket_accept、nx_tcp_server_socket_listen,
- nx_tcp_server_socket_relisten、nx_tcp_server_socket_unaccept,
- nx_tcp_socket_bytes_available、nx_tcp_socket_create,

- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_socket_bytes_available

检索可供检索的字节数

原型

```
UINT nx_tcp_socket_bytes_available(
    NX_TCP_SOCKET *socket_ptr,
    ULONG *bytes_available);
```

说明

此服务用于获取指定的 TCP 套接字可供检索的字节数。请注意，必须已连接该 TCP 套接字。

参数

- **socket_ptr** 此指针指向先前创建并连接的 TCP 套接字。
- **bytes_available** 此指针指向可用字节数的目标。

返回值

- **NX_SUCCESS** (0x00) 成功执行服务。可供读取的字节数会返回给调用方。
- **NX_NOT_CONNECTED** (0x38) 套接字未处于已连接状态。
- **NX_PTR_ERROR** (0x07) 指针无效。
- **NX_NOT_ENABLED** (0x14) 未启用 TCP。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Get the bytes available for retrieval on the specified socket. */
status = nx_tcp_socket_bytes_available(&my_socket,
    &bytes_available);

/* Is status = NX_SUCCESS, the available bytes is returned in
    bytes_available. */
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_info_get,
- nx_tcp_server_socket_accept、nx_tcp_server_socket_listen,
- nx_tcp_server_socket_relisten、nx_tcp_server_socket_unaccept,
- nx_tcp_server_socket_unlisten、nx_tcp_socket_create,
- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,

- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_socket_create

创建 TCP 客户端或服务器套接字

原型

```
UINT nx_tcp_socket_create(
    NX_IP *ip_ptr,
    NX_TCP_SOCKET *socket_ptr,
    CHAR *name, ULONG type_of_service,
    ULONG fragment,
    UINT time_to_live,
    ULONG window_size,
    VOID (*urgent_data_callback)(NX_TCP_SOCKET *socket_ptr),
    VOID (*disconnect_callback)(NX_TCP_SOCKET *socket_ptr));
```

说明

此服务用于为指定的 IP 实例创建 TCP 客户端或服务器套接字。

应用程序回调例程是从与此 IP 实例关联的线程中调用的。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **socket_ptr** 指向新的 TCP 套接字控制块的指针。
- **name** 此 TCP 套接字的应用程序名称。
- **type_of_service** 定义用于传输的服务类型，合法值如下所示：
- NX_IP_NORMAL (0x00000000)
- NX_IP_MIN_DELAY (0x00100000)
- NX_IP_MAX_DATA (0x00080000)
- NX_IP_MAX_RELIABLE (0x00040000)
- NX_IP_MIN_COST (0x00020000)
- **fragment** 指定是否允许进行 IP 分段。如果指定 NX_FRAGMENT_OKAY (0x0)，则允许进行 IP 分段。如果指定 NX_DONT_FRAGMENT (0x4000)，则禁止进行 IP 分段。
- **time_to_live** 指定一个 8 位的值，用于定义此数据包在被丢弃之前可通过的路由器数目。默认值由 NX_IP_TIME_TO_LIVE 指定。
- **window_size** 定义此套接字的接收队列中允许的最大字节数
- **urgent_data_callback** 在接收流中每次检测到紧急数据时都会调用的应用程序函数。如果此值为 NX_NULL，则会忽略紧急数据。
- **disconnect_callback** 在连接另一端的套接字每次发出断开连接时都会调用的应用程序函数。如果此值为 NX_NULL，则会禁用断开连接回调函数。

返回值

- **NX_SUCCESS** (0x00) 成功创建 TCP 客户端套接字。
- **NX_OPTION_ERROR** (0x0A) 服务类型、分段、窗口大小或生存时间选项无效。

- **NX_PTR_ERROR** (0x07) IP 或套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

初始化和线程

可以抢占

否

示例

```
/* Create a TCP client socket on the previously created IP instance,
    with normal delivery, IP fragmentation enabled, 0x80 time to
    live, a 200-byte receive window, no urgent callback routine, and
    the "client_disconnect" routine to handle disconnection initiated
    from the other end of the connection. */
status = nx_tcp_socket_create(&ip_0, &client_socket,
    "Client Socket",
    NX_IP_NORMAL, NX_FRAGMENT_OKAY,
    0x80, 200, NX_NULL
    client_disconnect);

/* If status is NX_SUCCESS, the client socket is created and ready
    to be bound. */
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_info_get,
- nx_tcp_server_socket_accept、nx_tcp_server_socket_listen,
- nx_tcp_server_socket_relisten、nx_tcp_server_socket_unaccept,
- nx_tcp_server_socket_unlisten、nx_tcp_socket_bytes_available,
- nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_socket_delete

删除 TCP 套接字

原型

```
UINT nx_tcp_socket_delete(NX_TCP_SOCKET *socket_ptr);
```

说明

此服务用于删除先前创建的 TCP 套接字。如果该套接字仍处于绑定或连接状态，此服务会返回错误代码。

参数

- **socket_ptr** 先前创建的 TCP 套接字

返回值

- **NX_SUCCESS** (0x00) 成功删除套接字。

- **NX_NOT_CREATED** (0x27) 尚未创建套接字。
- **NX_STILL_BOUND** (0x42) 套接字仍处于绑定状态。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
/* Delete a previously created TCP client socket. */
status = nx_tcp_socket_delete(&client_socket);

/* If status is NX_SUCCESS, the client socket is deleted. */
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_info_get,
- nx_tcp_server_socket_accept、nx_tcp_server_socket_listen,
- nx_tcp_server_socket_relisten、nx_tcp_server_socket_unaccept,
- nx_tcp_server_socket_unlisten、nx_tcp_socket_bytes_available,
- nx_tcp_socket_create、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send,
- nx_tcp_socket_state_wait

nx_tcp_socket_disconnect

断开客户端和服务器套接字连接

原型

```
UINT nx_tcp_socket_disconnect(
    NX_TCP_SOCKET *socket_ptr,
    ULONG wait_option);
```

说明

此服务用于断开已建立的客户端或服务器套接字连接。在服务器套接字断开连接后应该有一个取消接受请求，而断开连接的客户端套接字会处于准备好接受其他连接请求的状态。如果断开连接过程无法立即完成，该服务会根据提供的等待选项挂起。

参数

- **socket_ptr** 此指针指向先前连接的客户端或服务器套接字实例。
- **wait_option** 定义正在断开连接时此服务的行为方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) 成功断开套接字连接。
- **NX_NOT_CONNECTED** (0x38) 指定的套接字未连接。
- **NX_IN_PROGRESS** (0x37) 正在断开连接，未指定等待。
- **NX_WAIT_ABORTED** (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

是

示例

```
/* Disconnect from a previously established connection and wait a
    maximum of 400 timer ticks. */
status = nx_tcp_socket_disconnect(&client_socket, 400);

/* If status is NX_SUCCESS, the previously connected socket (either
    as a result of the client socket connect or the server accept) is
    disconnected. */
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_info_get,
- nx_tcp_server_socket_accept、nx_tcp_server_socket_listen,
- nx_tcp_server_socket_relisten、nx_tcp_server_socket_unaccept,
- nx_tcp_server_socket_unlisten、nx_tcp_socket_bytes_available,
- nx_tcp_socket_create、nx_tcp_socket_delete、nx_tcp_socket_info_get,
- nx_tcp_socket_receive、nx_tcp_socket_receive_queue_max_set,
- nx_tcp_socket_send、nx_tcp_socket_state_wait

nx_tcp_socket_disconnect_complete_notify

安装 TCP 断开连接完成通知回调函数

原型

```
UINT nx_tcp_socket_disconnect_complete_notify(
    NX_TCP_SOCKET *socket_ptr,
    VOID (*tcp_disconnect_complete_notify)
    (NX_TCP_SOCKET *socket_ptr));
```

说明

此服务用于注册一个回调函数，套接字断开连接操作完成后会调用该函数。如果 NetX 是在定义了 NX_ENABLE_EXTENDED_NOTIFY_SUPPORT 选项的情况下生成的，

- 则 TCP 套接字断开连接完成回调函数可用。

参数

- socket_ptr 此指针指向先前连接的客户端或服务器套接字实例。
- tcp_disconnect_complete_notify 要安装的回调函数。

返回值

- NX_SUCCESS (0x00) 已成功注册回调函数。
- NX_NOT_SUPPORTED (0x4B) 扩展通知功能未生成到 NetX 库中
- NX_PTR_ERROR (0x07) 套接字指针无效。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_NOT_ENABLED (0x14) TCP 功能未启用。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Install the disconnect complete notify callback function. */
status = nx_tcp_socket_disconnect_complete_notify(&client_socket,
    callback);
```

另请参阅

- nx_tcp_enable、nx_tcp_socket_create、nx_tcp_socket_establish_notify,
- nx_tcp_socket_mss_get、nx_tcp_socket_mss_peer_get,
- nx_tcp_socket_mss_set、nx_tcp_socket_peer_info_get,
- nx_tcp_socket_receive_notify、nx_tcp_socket_timed_wait_callback,
- nx_tcp_socket_transmit_configure,
- nx_tcp_socket_window_update_notify_set

nx_tcp_socket_establish_notify

设置 TCP 建立通知回调函数

原型

```
UINT nx_tcp_socket_establish_notify(
    NX_TCP_SOCKET *socket_ptr,
    VOID (*tcp_establish_notify)(NX_TCP_SOCKET *socket_ptr));
```

说明

此服务用于注册一个回调函数，TCP 套接字建立连接后会调用该函数。如果 NetX 是在定义了 NX_ENABLE_EXTENDED_NOTIFY_SUPPORT 选项的情况下生成的，则 TCP 套接字建立回调函数可用。

参数

- socket_ptr 此指针指向先前连接的客户端或服务器套接字实例。
- tcp_establish_notify 建立 TCP 连接后调用的回调函数。

返回值

- NX_SUCCESS (0x00) 成功设置通知函数。
- NX_NOT_SUPPORTED (0x4B) 扩展通知功能未生成到 NetX 库中
- NX_PTR_ERROR (0x07) 套接字指针无效。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。

- **NX_NOT_ENABLED** (0x14) 应用程序尚未启用 TCP。

允许调用自

线程数

可以抢占

否

示例

```
/* Set the function pointer "callback" as the notify function NetX
will call when the connection is in the established state. */
status = nx_tcp_socket_establish_notify(&client_socket, callback);
```

另请参阅

- nx_tcp_enable、nx_tcp_socket_create,
- nx_tcp_socket_disconnect_complete_notify、nx_tcp_socket_mss_get,
- nx_tcp_socket_mss_peer_get、nx_tcp_socket_mss_set,
- nx_tcp_socket_peer_info_get、nx_tcp_socket_receive_notify,
- nx_tcp_socket_timed_wait_callback、nx_tcp_socket_transmit_configure,
- nx_tcp_socket_window_update_notify_set

nx_tcp_socket_info_get

检索 TCP 套接字活动的相关信息

原型

```
UINT nx_tcp_socket_info_get(
    NX_TCP_SOCKET *socket_ptr,
    ULONG *tcp_packets_sent,
    ULONG *tcp_bytes_sent,
    ULONG *tcp_packets_received,
    ULONG *tcp_bytes_received,
    ULONG *tcp_retransmit_packets,
    ULONG *tcp_packets_queued,
    ULONG *tcp_checksum_errors,
    ULONG *tcp_socket_state,
    ULONG *tcp_transmit_queue_depth,
    ULONG *tcp_transmit_window,
    ULONG *tcp_receive_window);
```

说明

此服务会检索指定的 TCP 套接字实例的 TCP 套接字活动相关信息。

如果目标指针为 NX_NULL，则不会将该特定信息返回给调用方。

参数

- **socket_ptr** 此指针指向先前创建的 TCP 套接字实例。
- **tcp_packets_sent** 此指针指向在套接字上发送的 TCP 数据包总数的目标。
- **tcp_bytes_sent** 此指针指向在套接字上发送的 TCP 总字节数的目标。
- **tcp_packets_received** 此指针指向在套接字上接收的 TCP 数据包总数的目标。
- **tcp_bytes_received** 此指针指向在套接字上接收的 TCP 总字节数的目标。
- **tcp_retransmit_packets** 此指针指向重新传输的 TCP 数据包总数的目标。
- **tcp_packets_queued** 此指针指向在套接字上排队的 TCP 数据包总数的目标。

- **tcp_checksum_errors** 此指针指向套接字上有校验和错误的 TCP 数据包总数的目标。
- **tcp_socket_state** 此指针指向套接字当前状态的目标。
- **tcp_transmit_queue_depth** 此指针指向仍在排队等待 ACK 的传输数据包总数的目标。
- **tcp_transmit_window** 此指针指向当前传输窗口大小的目标。
- **tcp_receive_window** 此指针指向当前接收窗口大小的目标。

返回值

- **NX_SUCCESS** (0x00) 成功检索 TCP 套接字信息。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

返回值

```
/* Retrieve TCP socket information from previously created socket_0.*/
status = nx_tcp_socket_info_get(&socket_0,
    &tcp_packets_sent,
    &tcp_bytes_sent,
    &tcp_packets_received,
    &tcp_bytes_received,
    &tcp_retransmit_packets,
    &tcp_packets_queued,
    &tcp_checksum_errors,
    &tcp_socket_state,
    &tcp_transmit_queue_depth,
    &tcp_transmit_window,
    &tcp_receive_window);

/* If status is NX_SUCCESS, TCP socket information was retrieved. */
```

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Retrieve TCP socket information from previously created socket_0.*/
status = nx_tcp_socket_info_get(&socket_0,
    &tcp_packets_sent,
    &tcp_bytes_sent,
    &tcp_packets_received,
    &tcp_bytes_received,
    &tcp_retransmit_packets,
    &tcp_packets_queued,
    &tcp_checksum_errors,
    &tcp_socket_state,
    &tcp_transmit_queue_depth,
    &tcp_transmit_window,
    &tcp_receive_window);

/* If status is NX_SUCCESS, TCP socket information was retrieved. */
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_info_get,

- nx_tcp_server_socket_accept、nx_tcp_server_socket_listen,
- nx_tcp_server_socket_relisten、nx_tcp_server_socket_unaccept,
- nx_tcp_server_socket_unlisten、nx_tcp_socket_bytes_available,
- nx_tcp_socket_create、nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_receive、nx_tcp_socket_receive_queue_max_set,
- nx_tcp_socket_send、nx_tcp_socket_state_wait

nx_tcp_socket_mss_get

获取套接字的 MSS

原型

```
UINT nx_tcp_socket_mss_get(
    NX_TCP_SOCKET *socket_ptr,
    ULONG *mss);
```

说明

此服务会检索指定的套接字的本地最大段大小 (MSS)。

参数

- **socket_ptr** 此指针指向先前创建的套接字。
- **mss** 用于返回 MSS 的目标。

返回值

- **NX_SUCCESS** (0x00) 成功获取 MSS。
- **NX_PTR_ERROR** (0x07) 套接字指针或 MSS 目标指针无效。
- **NX_NOT_ENABLED** (0x14) 未启用 TCP。
- **NX_CALLER_ERROR** (0x11) 调用方不是线程或初始化。

允许调用自

初始化和线程

可以抢占

否

示例

```
/* Get the MSS for the socket "my_socket". */
status = nx_tcp_socket_mss_get(&my_socket, &mss_value);

/* If status is NX_SUCCESS, the "mss_value" variable contains the
    socket's current MSS value. */
```

另请参阅

- nx_tcp_enable、nx_tcp_socket_create,
- nx_tcp_socket_disconnect_complete_notify,
- nx_tcp_socket_establish_notify、nx_tcp_socket_mss_peer_get,
- nx_tcp_socket_mss_set、nx_tcp_socket_peer_info_get,
- nx_tcp_socket_receive_notify、nx_tcp_socket_timed_wait_callback,
- nx_tcp_socket_transmit_configure,
- nx_tcp_socket_window_update_notify_set

nx_tcp_socket_mss_peer_get

获取对等 TCP 套接字的 MSS

原型

```
UINT nx_tcp_socket_mss_peer_get(
    NX_TCP_SOCKET *socket_ptr,
    ULONG *mss);
```

说明

此服务会检索对等套接字播发的最大段大小 (MSS)。

参数

- **socket_ptr** 此指针指向先前创建并连接的套接字。
- **mss** 用于返回 MSS 的目标。

返回值

- **NX_SUCCESS** (0x00) 成功获取对等 MSS。
- **NX_PTR_ERROR** (0x07) 套接字指针或 MSS 目标指针无效。
- **NX_NOT_ENABLED** (0x14) 未启用 TCP。
- **NX_CALLER_ERROR** (0x11) 调用方不是线程或初始化。

允许调用自

线程数

可以抢占

否

示例

```
/* Get the MSS of the connected peer to the socket "my_socket". */
status = nx_tcp_socket_mss_peer_get(&my_socket, &mss_value);

/* If status is NX_SUCCESS, the "mss_value" variable contains the
    socket peer's advertised MSS value. */
```

另请参阅

- nx_tcp_enable、nx_tcp_socket_create,
- nx_tcp_socket_disconnect_complete_notify,
- nx_tcp_socket_establish_notify、nx_tcp_socket_mss_get,
- nx_tcp_socket_mss_set、nx_tcp_socket_peer_info_get,
- nx_tcp_socket_receive_notify、nx_tcp_socket_timed_wait_callback,
- nx_tcp_socket_transmit_configure,
- nx_tcp_socket_window_update_notify_set

nx_tcp_socket_mss_set

设置套接字的 MSS

原型

```
UINT nx_tcp_socket_mss_set(
    NX_TCP_SOCKET *socket_ptr,
    ULONG mss);
```

说明

此服务用于设置指定的套接字的最大段大小 (MSS)。请注意，MSS 值必须小于网络接口 IP MTU，从而为 IP 和 TCP 标头留出空间。

应该在 TCP 套接字开始连接过程之前使用此服务。如果在建立 TCP 连接后使用此服务，新值不会对该连接产生任何影响。

参数

- **socket_ptr** 此指针指向先前创建的套接字。
- **mss** 要设置的 MSS 值。

返回值

- **NX_SUCCESS** (0x00) 成功设置 MSS。
- **NX_SIZE_ERROR** (0x09) 指定的 MSS 值太大。
- **NX_NOT_CONNECTED** (0x38) 尚未建立 TCP 连接
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_NOT_ENABLED** (0x14) 未启用 TCP。
- **NX_CALLER_ERROR** (0x11) 调用方不是线程或初始化。

允许调用自

初始化和线程

可以抢占

否

示例

```
/* Set the MSS of the socket "my_socket" to 1000 bytes. */
status = nx_tcp_socket_mss_set(&my_socket, 1000);

/* If status is NX_SUCCESS, the MSS of "my_socket" is 1000 bytes. */
```

另请参阅

- nx_tcp_enable、nx_tcp_socket_create,
- nx_tcp_socket_disconnect_complete_notify,
- nx_tcp_socket_establish_notify、nx_tcp_socket_mss_get,
- nx_tcp_socket_mss_peer_get、nx_tcp_socket_peer_info_get,
- nx_tcp_socket_receive_notify、nx_tcp_socket_timed_wait_callback,
- nx_tcp_socket_transmit_configure,
- nx_tcp_socket_window_update_notify_set

nx_tcp_socket_peer_info_get

检索对等 TCP 套接字的相关信息

原型

```
UINT nx_tcp_socket_peer_info_get(
    NX_TCP_SOCKET *socket_ptr,
    ULONG *peer_ip_address,
    ULONG *peer_port);
```

说明

此服务用于通过 IP 网络检索已连接的 TCP 套接字的对等 IP 地址和端口信息。

参数

- **socket_ptr** 指向先前创建的 TCP 套接字的指针。
- **peer_ip_address** 此指针指向对等 IP 地址的目标，以主机字节顺序表示。
- **peer_port** 此指针指向对等端口号的目标，以主机字节顺序表示。

返回值

- **NX_SUCCESS** (0x00) 成功执行服务。对等 IP 地址和端口号会返回给调用方。
- **NX_NOT_CONNECTED** (0x38) 套接字未处于已连接状态。
- **NX_PTR_ERROR** (0x07) 指针无效。
- **NX_NOT_ENABLED** (0x14) 未启用 TCP。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Obtain peer IP address and port on the specified TCP socket. */
status = nx_tcp_socket_peer_info_get(&my_socket, &peer_ip_address,
    &peer_port);

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

另请参阅

- nx_tcp_enable、nx_tcp_socket_create,
- nx_tcp_socket_disconnect_complete_notify,
- nx_tcp_socket_establish_notify、nx_tcp_socket_mss_get,
- nx_tcp_socket_mss_peer_get、nx_tcp_socket_mss_set,
- nx_tcp_socket_receive_notify、nx_tcp_socket_timed_wait_callback,
- nx_tcp_socket_transmit_configure,
- nx_tcp_socket_window_update_notify_set

nx_tcp_socket_receive

从 TCP 套接字接收数据

原型

```
UINT nx_tcp_socket_receive(
    NX_TCP_SOCKET *socket_ptr,
    NX_PACKET **packet_ptr,
    ULONG wait_option);
```

说明

此服务用于从指定的套接字接收 TCP 数据。如果指定的套接字上没有数据在排队，调用方会根据提供的等待选项挂起。

如果返回了 NX_SUCCESS，则应用程序负责在不再需要所收到的数据包时将其释放。

参数

- **socket_ptr** 此指针指向先前创建的 TCP 套接字实例。
- **packet_ptr** 此指针指向 TCP 数据包指针。
- **wait_option** 定义此套接字上当前没有数据排队时此服务的行为方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) 成功接收套接字数据。
- **NX_NOT_BOUND** (0x24) 套接字尚未绑定。
- **NX_NO_PACKET** (0x01) 未收到任何数据。
- **NX_WAIT_ABORTED** (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。
- **NX_NOT_CONNECTED** (0x38) 该套接字不再处于已连接状态。
- **NX_PTR_ERROR** (0x07) 套接字指针或返回数据包指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

/* 从先前创建并连接的 TCP 客户端套接字接收数据包。如果无数据包可用，请等待 200 个计时器时钟周期，然后放弃。*/ status = nx_tcp_socket_receive(&client_socket, &packet_ptr, 200);

/* 如果状态为 NX_SUCCESS，则"packet_ptr"指向接收到的数据包。*/

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_info_get,
- nx_tcp_server_socket_accept、nx_tcp_server_socket_listen,
- nx_tcp_server_socket_relisten、nx_tcp_server_socket_unaccept,
- nx_tcp_server_socket_unlisten、nx_tcp_socket_bytes_available,
- nx_tcp_socket_create、nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive_queue_max_set,
- nx_tcp_socket_send、nx_tcp_socket_state_wait

nx_tcp_socket_receive_notify

在收到数据包时通知应用程序

原型

```
UINT nx_tcp_socket_receive_notify(
    NX_TCP_SOCKET *socket_ptr,
    VOID (*tcp_receive_notify) (NX_TCP_SOCKET *socket_ptr));
```

说明

此服务使用应用程序指定的回调函数来配置接收通知函数指针。每当在套接字上收到一个或多个数据包时，都调用该回调函数。如果提供了 NX_NULL 指针，则会禁用通知功能。

参数

- **socket_ptr** 此指针指向 TCP 套接字。
- **tcp_receive_notify** 在套接字上收到一个或多个数据包时调用的应用程序回调函数指针。

返回值

- **NX_SUCCESS** (0x00) 通知成功接收套接字。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) TCP 功能未启用。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Setup a receive packet callback function for the "client_socket"
socket. */
status = nx_tcp_socket_receive_notify(&client_socket,
    my_receive_notify);

/* If status is NX_SUCCESS, NetX will call the function named
    "my_receive_notify" whenever one or more packets are received for
    "client_socket". */
```

另请参阅

- nx_tcp_enable、nx_tcp_socket_create,
- nx_tcp_socket_disconnect_complete_notify,
- nx_tcp_socket_establish_notify、nx_tcp_socket_mss_get,
- nx_tcp_socket_mss_peer_get、nx_tcp_socket_mss_set,
- nx_tcp_socket_peer_info_get、nx_tcp_socket_timed_wait_callback,
- nx_tcp_socket_transmit_configure,
- nx_tcp_socket_window_update_notify_set

nx_tcp_socket_send

通过 TCP 套接字发送数据

原型

```
UINT nx_tcp_socket_send(
    NX_TCP_SOCKET *socket_ptr,
    NX_PACKET *packet_ptr,
    ULONG wait_option);
```

说明

此服务用于通过先前连接的 TCP 套接字发送 TCP 数据。如果接收方上次播发的窗口大小低于此请求，此服务可选择根据指定的等待选项挂起。此服务可保证不会将大于 MSS 的数据包数据发送到 IP 层。

除非返回了错误，否则应用程序不应在此调用后释放该数据包。这样做会导致不可预知的结果，因为网络驱动程序还会在传输后尝试释放该数据包。

参数

- **socket_ptr** 此指针指向先前连接的 TCP 套接字实例。
- **packet_ptr** TCP 数据包指针。
- **wait_option** 定义当请求大于接收方窗口大小时此服务的行为方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) 成功发送套接字。
- **NX_NOT_BOUND** (0x24) 套接字未与任何端口绑定。
- **NX_NO_INTERFACE_ADDRESS** (0x50) 找不到合适的传出接口。
- **NX_NOT_CONNECTED** (0x38) 套接字不再处于已连接状态。
- **NX_WINDOW_OVERFLOW** (0x39) 请求大于接收方播发的窗口大小（以字节为单位）。
- **NX_WAIT_ABORTED** (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。
- **NX_INVALID_PACKET** (0x12) 未分配数据包。
- **NX_TX_QUEUE_DEPTH** (0x49) 已达到最大传输队列深度。
- **NX_OVERFLOW** (0x03) 数据包追加指针无效。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。
- **NX_UNDERFLOW** (0x02) 数据包前置指针无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Send a packet out on the previously created and connected TCP
    socket. If the receive window on the other side of the connection
    is less than the packet size, wait 200 timer ticks before giving up. */
status = nx_tcp_socket_send(&client_socket, packet_ptr, 200);

/* If status is NX_SUCCESS, the packet has been sent! */
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,

- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_info_get,
- nx_tcp_server_socket_accept、nx_tcp_server_socket_listen,
- nx_tcp_server_socket_relisten、nx_tcp_server_socket_unaccept,
- nx_tcp_server_socket_unlisten、nx_tcp_socket_bytes_available,
- nx_tcp_socket_create、nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_state_wait

nx_tcp_socket_state_wait

等待 TCP 套接字进入特定状态

原型

```
UINT nx_tcp_socket_state_wait(
    NX_TCP_SOCKET *socket_ptr,
    UINT desired_state,
    ULONG wait_option);
```

说明

此服务用于等待套接字进入所需状态。如果该套接字未处于所需状态，此服务会根据提供的等待选项挂起。

参数

- **socket_ptr** 此指针指向先前连接的 TCP 套接字实例。
- **desired_state** 所需的 TCP 状态。有效的 TCP 套接字状态定义如下：
- NX_TCP_CLOSED (0x01)
- NX_TCP_LISTEN_STATE (0x02)
- NX_TCP_SYN_SENT (0x03)
- NX_TCP_SYN_RECEIVED (0x04)
- NX_TCP_ESTABLISHED (0x05)
- NX_TCP_CLOSE_WAIT (0x06)
- NX_TCP_FIN_WAIT_1 (0x07)
- NX_TCP_FIN_WAIT_2 (0x08)
- NX_TCP_CLOSING (0x09)
- NX_TCP_TIMED_WAIT (0x0A)
- NX_TCP_LAST_ACK (0x0B)
- **wait_option** 定义所请求的状态不存在时此服务的行为方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) 成功等到所需状态。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_NOT_SUCCESSFUL** (0x43) 指定的等待时间内不存在所需状态。
- **NX_WAIT_ABORTED** (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。
- **NX_OPTION_ERROR** (0x0A) 所需的套接字状态无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Wait 300 timer ticks for the previously created socket to enter
    the established state in the TCP state machine. */
status = nx_tcp_socket_state_wait(&client_socket,
    NX_TCP_ESTABLISHED, 300);

/* If status is NX_SUCCESS, the socket is now in the established
    state! */
```

另请参阅

- nx_tcp_client_socket_bind、nx_tcp_client_socket_connect,
- nx_tcp_client_socket_port_get、nx_tcp_client_socket_unbind,
- nx_tcp_enable、nx_tcp_free_port_find、nx_tcp_info_get,
- nx_tcp_server_socket_accept、nx_tcp_server_socket_listen,
- nx_tcp_server_socket_relisten、nx_tcp_server_socket_unaccept,
- nx_tcp_server_socket_unlisten、nx_tcp_socket_bytes_available,
- nx_tcp_socket_create、nx_tcp_socket_delete、nx_tcp_socket_disconnect,
- nx_tcp_socket_info_get、nx_tcp_socket_receive,
- nx_tcp_socket_receive_queue_max_set、nx_tcp_socket_send

nx_tcp_socket_timed_wait_callback

安装用于定时等待状态的回调

原型

```
UINT nx_tcp_socket_timed_wait_callback(
    NX_TCP_SOCKET *socket_ptr,
    VOID (*tcp_timed_wait_callback) (NX_TCP_SOCKET *socket_ptr));
```

说明

此服务用于注册一个回调函数，当 TCP 套接字处于定时等待状态时，会调用该函数。若要使用此服务，必须在定义了 NX_ENABLE_EXTENDED_NOTIFY 选项的情况下生成 NetX 库。

参数

- **socket_ptr** 此指针指向先前连接的客户端或服务器套接字实例。
- **tcp_timed_wait_callback** 定时等待回调函数

返回值

- **NX_SUCCESS** (0x00) 成功注册回调函数套接字
- **NX_NOT_SUPPORTED** (0x4B) NetX 库是在未启用扩展通知功能的情况下生成的。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) TCP 功能未启用。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Install the timed wait callback function */
nx_tcp_socket_timed_wait_callback(&client_socket, callback);
```

另请参阅

- nx_tcp_enable、nx_tcp_socket_create,
- nx_tcp_socket_disconnect_complete_notify,
- nx_tcp_socket_establish_notify、nx_tcp_socket_mss_get,
- nx_tcp_socket_mss_peer_get、nx_tcp_socket_mss_set,
- nx_tcp_socket_peer_info_get、nx_tcp_socket_receive_notify,
- nx_tcp_socket_transmit_configure,
- nx_tcp_socket_window_update_notify_set

nx_tcp_socket_transmit_configure

配置套接字的传输参数

原型

```
UINT nx_tcp_socket_transmit_configure(
    NX_TCP_SOCKET *socket_ptr,
    ULONG max_queue_depth,
    ULONG timeout,
    ULONG max_retries,
    ULONG timeout_shift);
```

说明

此服务用于配置指定的 TCP 套接字的各种传输参数。

参数

- **socket_ptr** 此指针指向 TCP 套接字。
- **max_queue_depth** 允许排队等待传输的数据包最大数量。
- **timeout** 在重新发送数据包之前等待 ACK 的 ThreadX 计时器时钟周期数。
- **max_retries** 允许的最大重试次数。
- **timeout_shift** 用于更改每次后续重试超时时间的值。值为 0 表示连续重试之间的超时时间相同。值为 1 表示重试之间的超时时间加倍。

返回值

- **NX_SUCCESS** (0x00) 成功配置传输套接字。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_OPTION_ERROR** (0x0a) 队列深度选项无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) TCP 功能未启用。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Configure the "client_socket" for a maximum transmit queue depth
    of 12, 100 tick timeouts, a maximum of 20 retries, and a timeout
    double on each successive retry. */
status = nx_tcp_socket_transmit_configure(&client_socket,
    12,100,20, 1);

/* If status is NX_SUCCESS, the socket's transmit retry has been
    configured. */
```

另请参阅

- nx_tcp_enable、nx_tcp_socket_create,
- nx_tcp_socket_disconnect_complete_notify,
- nx_tcp_socket_establish_notify、nx_tcp_socket_mss_get,
- nx_tcp_socket_mss_peer_get、nx_tcp_socket_mss_set,
- nx_tcp_socket_peer_info_get、nx_tcp_socket_receive_notify,
- nx_tcp_socket_timed_wait_callback,
- nx_tcp_socket_window_update_notify_set

nx_tcp_socket_window_update_notify_set

在窗口大小更新时通知应用程序

原型

```
UINT nx_tcp_socket_window_update_notify_set(
    NX_TCP_SOCKET
    *socket_ptr,
    VOID (*tcp_window_update_notify)
    (NX_TCP_SOCKET *socket_ptr));
```

说明

此服务用于安装套接字窗口更新回调例程。每当指定的套接字收到数据包指出远程主机窗口大小增加时，都会自动调用该例程。

参数

- **socket_ptr** 指向先前创建的 TCP 套接字的指针。
- **tcp_window_update_notify** 要在窗口大小更改时调用的回调例程。值为 NULL 表示禁用窗口更改更新。

返回值

- **NX_SUCCESS** (0x00) 套接字上已安装回调例程。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_PTR_ERROR** (0x07) 指针无效。
- **NX_NOT_ENABLED** (0x14) TCP 功能未启用。

允许调用自

初始化、线程

可以抢占

否

示例

```
/* Set the function pointer to the windows update callback after creating the
socket. */
status = nx_tcp_socket_window_update_notify_set(&data_socket,
    my_windows_update_callback);

/* Define the window callback function in the host application. */
void my_windows_update_callback(NX_TCP_SCOCKET *data_socket)
{
    /* Process update on increase TCP transmit socket window size. */
    return;
}
```

另请参阅

- nx_tcp_enable、nx_tcp_socket_create,
- nx_tcp_socket_disconnect_complete_notify,
- nx_tcp_socket_establish_notify、nx_tcp_socket_mss_get,
- nx_tcp_socket_mss_peer_get、nx_tcp_socket_mss_set,
- nx_tcp_socket_peer_info_get、nx_tcp_socket_receive_notify,
- nx_tcp_socket_timed_wait_callback、nx_tcp_socket_transmit_configure

nx_udp_enable

启用 NetX 的 UDP 组件

原型

```
UINT nx_udp_enable(NX_IP *ip_ptr);
```

说明

此服务用于启用 NetX 的用户数据报协议 (UDP) 组件。启用后，应用程序可发送和接收 UDP 数据报。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。

返回值

- **NX_SUCCESS** (0x00) 成功启用 UDP。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_ALREADY_ENABLED** (0x15) 已启用此组件。

允许调用自

初始化、线程、计时器

可以抢占

否

示例

```
/* Enable UDP on the previously created IP instance. */
status = nx_udp_enable(&ip_0);

/* If status is NX_SUCCESS, UDP is now enabled on the specified IP
    instance. */
```

另请参阅

- nx_udp_free_port_find、nx_udp_info_get、nx_udp_packet_info_extract,
- nx_udp_socket_bind、nx_udp_socket_bytes_available,
- nx_udp_socket_checksum_disable、nx_udp_socket_checksum_enable,
- nx_udp_socket_create、nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive,
- nx_udp_socket_receive_notify、nx_udp_socket_send,
- nx_udp_socket_interface_send、nx_udp_socket_unbind,
- nx_udp_source_extract

nx_udp_free_port_find

查找下一个可用的 UDP 端口

原型

```
UINT nx_udp_free_port_find(
    NX_IP *ip_ptr,
    UINT port,
    UINT *free_port_ptr);
```

说明

此服务从应用程序提供的端口号开始查找可用未绑定的 UDP 端口。搜索达到最大端口值 0xFFFF 时，搜索逻辑会从头开始。如果搜索成功，则会在 free_port_ptr 所指向的变量中返回可用端口。

可从其他线程调用此服务，这可返回同一端口。若要防止出现这种争用情况，应用程序可能希望将此服务和实际的套接字绑定置于互斥体的保护之下。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **port** 充当搜索起点的端口号（1 到 0xFFFF）。
- **free_port_ptr** 此指针指向目标可用端口返回变量。

返回值

- **NX_SUCCESS** (0x00) 成功找到可用端口。
- **NX_NO_FREE_PORTS** (0x45) 找不到可用端口。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。
- **NX_INVALID_PORT** (0x46) 指定的端口号无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Locate a free UDP port, starting at port 12, on a previously
    created IP instance. */
status = nx_udp_free_port_find(&ip_0, 12, &free_port);

/* If status is NX_SUCCESS pointer, "free_port" identifies the next
    free UDP port on the IP instance. */
```

另请参阅

- nx_udp_enable、nx_udp_info_get、nx_udp_packet_info_extract,
- nx_udp_socket_bind、nx_udp_socket_bytes_available,
- nx_udp_socket_checksum_disable、nx_udp_socket_checksum_enable,
- nx_udp_socket_create、nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive,
- nx_udp_socket_receive_notify、nx_udp_socket_send,
- nx_udp_socket_interface_send、nx_udp_socket_unbind,
- nx_udp_source_extract

nx_udp_info_get

检索 UDP 活动的相关信息

原型

```
UINT nx_udp_info_get(
    NX_IP *ip_ptr,
    ULONG *udp_packets_sent,
    ULONG *udp_bytes_sent,
    ULONG *udp_packets_received,
    ULONG *udp_bytes_received,
    ULONG *udp_invalid_packets,
    ULONG *udp_receive_packets_dropped,
    ULONG *udp_checksum_errors);
```

说明

此服务会检索指定的 IP 实例的 UDP 活动相关信息。

如果目标指针为 NX_NULL，则不会将该特定信息返回给调用方。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **udp_packets_sent** 此指针指向已发送的 UDP 数据包总数的目标。
- **udp_bytes_sent** 此指针指向已发送的 UDP 总字节数的目标。
- **udp_packets_received** 此指针指向已接收的 UDP 数据包总数的目标。
- **udp_bytes_received** 此指针指向已接收的 UDP 总字节数的目标。
- **udp_invalid_packets** 此指针指向无效 UDP 数据包总数的目标。
- **udp_receive_packets_dropped** 此指针指向已丢弃的 UDP 接收数据包总数的目标。
- **udp_checksum_errors** 此指针指向有校验和错误的 UDP 数据包总数的目标。

返回值

- **NX_SUCCESS** (0x00) 成功检索 UDP 信息。
- **NX_PTR_ERROR** (0x07) IP 指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

初始化、线程和计时器

可以抢占

否

示例

```
/* Retrieve UDP information from previously created IP Instance ip_0. */
status = nx_udp_info_get(&ip_0, &udp_packets_sent,
    &udp_bytes_sent,
    &udp_packets_received,
    &udp_bytes_received,
    &udp_invalid_packets,
    &udp_receive_packets_dropped,
    &udp_checksum_errors);

/* If status is NX_SUCCESS, UDP information was retrieved. */
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_packet_info_extract,
- nx_udp_socket_bind、nx_udp_socket_bytes_available,
- nx_udp_socket_checksum_disable、nx_udp_socket_checksum_enable,
- nx_udp_socket_create、nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive,
- nx_udp_socket_receive_notify、nx_udp_socket_send,
- nx_udp_socket_interface_send、nx_udp_socket_unbind,
- nx_udp_source_extract

nx_udp_packet_info_extract

从 UDP 数据包提取网络参数

原型

```
UINT nx_udp_packet_info_extract(
    NX_PACKET *packet_ptr,
    ULONG *ip_address,
    UINT *protocol,
    UINT *port,
    UINT *interface_index);
```

说明

此服务用于从传入接口上收到的数据包提取网络参数，例如 IP 地址、对等端口号和协议类型（此服务始终返回 UDP 类型）。

参数

- **packet_ptr** 指向数据包的指针。
- **ip_address** 指向发送方 IP 地址的指针。
- **protocol** 指向协议 (UDP) 的指针。
- **port** 指向发送方端口号的指针。
- **interface_index** 指向接收接口索引的指针。

返回值

- **NX_SUCCESS** (0x00) 已成功提取数据包接口数据。

- **NX_INVALID_PACKET** (0x12) 数据包不含 IP 帧。
- **NX_PTR_ERROR** (0x07) 指针输入无效
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Extract network data from UDP packet interface.*/
status = nx_udp_packet_info_extract( packet_ptr, &ip_address,
    &protocol, &port,
    &interface_index);

/* If status is NX_SUCCESS packet data was successfully retrieved. */
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_socket_bind、nx_udp_socket_bytes_available,
- nx_udp_socket_checksum_disable、nx_udp_socket_checksum_enable,
- nx_udp_socket_create、nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive,
- nx_udp_socket_receive_notify、nx_udp_socket_send,
- nx_udp_socket_interface_send、nx_udp_socket_unbind,
- nx_udp_source_extract

nx_udp_socket_bind

将 UDP 套接字与 UDP 端口绑定

原型

```
UINT nx_udp_socket_bind(
    NX_UDP_SOCKET *socket_ptr,
    UINT port,
    ULONG wait_option);
```

说明

此服务用于将先前创建的 UDP 套接字与指定的 UDP 端口绑定。有效 UDP 套接字介于 0 到 0xFFFF 之间。如果请求的端口号已绑定到其他套接字，此服务会按指定的时间长度，等待该套接字与该端口号取消绑定。

参数

- **socket_ptr** 此指针指向先前创建的 UDP 套接字实例。
- **port** 要绑定到的端口号（1 到 0xFFFF）。如果端口号为 NX_ANY_PORT (0x0000)，IP 实例会搜索下一个可用端口并将其用于绑定。
- **wait_option** 定义该端口已与另一套接字绑定时此服务的行为方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

返回值

- **NX_SUCCESS** (0x00) 成功绑定套接字。
- **NX_ALREADY_BOUND** (0x22) 此套接字已与另一端口绑定。
- **NX_PORT_UNAVAILABLE** (0x23) 端口已与其他套接字绑定。
- **NX_NO_FREE_PORTS** (0x45) 没有可用的端口。
- **NX_WAIT_ABORTED** (0x1A) 已通过调用 tx_thread_wait_abort 中止所请求的挂起。
- **NX_INVALID_PORT** (0x46) 指定的端口无效。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
/* Bind the previously created UDP socket to port 12 on the
    previously created IP instance. If the port is already bound,
    wait for 300 timer ticks before giving up. */
status = nx_udp_socket_bind(&udp_socket, 12, 300);

/* If status is NX_SUCCESS, the UDP socket is now bound to
    port 12.*/
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bytes_available,
- nx_udp_socket_checksum_disable、nx_udp_socket_checksum_enable,
- nx_udp_socket_create、nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive,
- nx_udp_socket_receive_notify、nx_udp_socket_send,
- nx_udp_socket_interface_send、nx_udp_socket_unbind,
- nx_udp_source_extract

nx_udp_socket_bytes_available

检索可供检索的字节数

原型

```
UINT nx_udp_socket_bytes_available(
    NX_UDP_SOCKET *socket_ptr,
    ULONG *bytes_available);
```

说明

此服务会检索指定的 UDP 套接字中可供接收的字节数。

参数

- **socket_ptr** 此指针指向先前创建的 UDP 套接字。
- **bytes_available** 此指针指向可用字节数的目标。

返回值

- **NX_SUCCESS** (0x00) 成功检索可用字节数。
- **NX_NOT_SUCCESSFUL** (0x43) 套接字未与任何端口绑定。
- **NX_PTR_ERROR** (0x07) 指针无效。
- **NX_NOT_ENABLED** (0x14) 未启用 UDP 功能。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。

允许调用自

线程数

可以抢占

否

示例

```
/* Get the bytes available for retrieval from the UDP socket. */
status = nx_udp_socket_bytes_available(&my_socket, &bytes_available);

/* If status == NX_SUCCESS, the number of bytes was successfully
    retrieved.*/
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_checksum_disable、nx_udp_socket_checksum_enable,
- nx_udp_socket_create、nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive,
- nx_udp_socket_receive_notify、nx_udp_socket_send,
- nx_udp_socket_interface_send、nx_udp_socket_unbind,
- nx_udp_source_extract

nx_udp_socket_checksum_disable

对 UDP 套接字禁用校验和

原型

```
UINT nx_udp_socket_checksum_disable(NX_UDP_SOCKET *socket_ptr);
```

说明

此服务禁用在指定的 UDP 套接字上发送和接收数据包的校验和逻辑。禁用校验和逻辑后，对于所有通过此套接字发送的数据包，系统都会将零值加载到 UDP 标头的校验和字段中。通过将 UDP 标头中的校验和值设置为零值，可告知接收方未计算该数据包的校验和。

另请注意，如果接收和发送 UDP 数据包时分别定义了 NX_DISABLE_UDP_RX_CHECKSUM 和 NX_DISABLE_UDP_TX_CHECKSUM，则这不起作用。

参数

- **socket_ptr** 此指针指向先前创建的 UDP 套接字实例。

返回值

- **NX_SUCCESS** (0x00) 成功禁用套接字校验和。
- **NX_NOT_BOUND** (0x24) 未绑定套接字。

- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

初始化、线程、计时器

可以抢占

否

示例

```
/* Disable the UDP checksum logic for packets sent on this socket. */
status = nx_udp_socket_checksum_disable(&udp_socket);

/* If status is NX_SUCCESS, outgoing packets will not have a checksum
    calculated. */
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_bytes_available、nx_udp_socket_checksum_disable,
- nx_udp_socket_create、nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive,
- nx_udp_socket_receive_notify、nx_udp_socket_send,
- nx_udp_socket_interface_send、nx_udp_socket_unbind,
- nx_udp_source_extract

nx_udp_socket_checksum_enable

对 UDP 套接字启用校验和

原型

```
UINT nx_udp_socket_checksum_enable(NX_UDP_SOCKET *socket_ptr);
```

说明

此服务启用在指定的 UDP 套接字上发送和接收数据包的校验和逻辑。校验和涵盖整个 UDP 数据区域以及一个伪 IP 标头。

另请注意，如果接收和发送 UDP 数据包时分别定义了 NX_DISABLE_UDP_RX_CHECKSUM 和 NX_DISABLE_UDP_TX_CHECKSUM，则这不起作用 。

参数

- **socket_ptr** 此指针指向先前创建的 UDP 套接字实例。

返回值

- **NX_SUCCESS** (0x00) 成功启用套接字校验和。
- **NX_NOT_BOUND** (0x24) 未绑定套接字。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

初始化、线程、计时器

可以抢占

否

示例

```
/* Enable the UDP checksum logic for packets sent on this socket. */
status = nx_udp_socket_checksum_enable(&udp_socket);

/* If status is NX_SUCCESS, outgoing packets will have a checksum
    calculated. */
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_bytes_available、nx_udp_socket_checksum_disable,
- nx_udp_socket_create、nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive,
- nx_udp_socket_receive_notify、nx_udp_socket_send,
- nx_udp_socket_interface_send、nx_udp_socket_unbind,
- nx_udp_source_extract

nx_udp_socket_create

创建 UDP 套接字。

原型

```
UINT nx_udp_socket_create(
    NX_IP *ip_ptr,
    NX_UDP_SOCKET *socket_ptr, CHAR *name,
    ULONG type_of_service, ULONG fragment,
    UINT time_to_live, ULONG queue_maximum);
```

说明

此服务用于为指定的 IP 实例创建 UDP 套接字。

参数

- **ip_ptr** 先前创建的 IP 实例的指针。
- **socket_ptr** 指向新的 UDP 套接字控制块的指针。
- **name** 此 UDP 套接字的应用程序名称。
- **type_of_service** 定义用于传输的服务类型，合法值如下所示：
  - NX_IP_NORMAL (0x00000000)
  - NX_IP_MIN_DELAY (0x00100000)
  - NX_IP_MAX_DATA (0x00080000)
  - NX_IP_MAX_RELIABLE (0x00040000)
  - NX_IP_MIN_COST (0x00020000)
- **fragment** 指定是否允许进行 IP 分段。如果指定 NX_FRAGMENT_OKAY (0x0)，则允许进行 IP 分段。如果指定 NX_DONT_FRAGMENT (0x4000)，则禁止进行 IP 分段。
- **time_to_live** 指定一个 8 位的值，用于定义此数据包在被丢弃之前可通过的路由器数目。默认值由

NX_IP_TIME_TO_LIVE 指定。

- **queue_maximum** 定义可为该套接字排队的 UDP 数据报最大数目。达到队列限制后，接收到每个新的数据包时，都会释放最早的 UDP 数据包。

返回值

- **NX_SUCCESS** (0x00) 成功创建 UDP 套接字。
- **NX_OPTION_ERROR** (0x0A) 服务类型、分段或生存时间选项无效。
- **NX_PTR_ERROR** (0x07) IP 或套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

初始化和线程

可以抢占

否

示例

```
/* Create a UDP socket with a maximum receive queue of 30 packets.*/
status = nx_udp_socket_create(&ip_0, &udp_socket, "Sample UDP Socket",
    NX_IP_NORMAL, NX_FRAGMENT_OKAY, 0x80, 30);

/* If status is NX_SUCCESS, the new UDP socket has been created and
    is ready for binding. */
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_bytes_available、nx_udp_socket_checksum_disable,
- nx_udp_socket_checksum_enable、nx_udp_socket_delete,
- nx_udp_socket_info_get、nx_udp_socket_port_get,
- nx_udp_socket_receive、nx_udp_socket_receive_notify,
- nx_udp_socket_send、nx_udp_socket_interface_send,
- nx_udp_socket_unbind、nx_udp_source_extract

nx_udp_socket_delete

删除 UDP 套接字

原型

```
UINT nx_udp_socket_delete(NX_UDP_SOCKET *socket_ptr);
```

说明

此服务用于删除先前创建的 UDP 套接字。如果该套接字已与某一端口绑定，则必须先取消绑定。

参数

- **socket_ptr** 此指针指向先前创建的 UDP 套接字实例。

返回值

- **NX_SUCCESS** (0x00) 成功删除套接字。
- **NX_STILL_BOUND** (0x42) 套接字仍处于绑定状态。

- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
/* Delete a previously created UDP socket. */
status = nx_udp_socket_delete(&udp_socket);

/* If status is NX_SUCCESS, the previously created UDP socket has
    been deleted. */
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_bytes_available、nx_udp_socket_checksum_disable,
- nx_udp_socket_checksum_enable、nx_udp_socket_create,
- nx_udp_socket_info_get、nx_udp_socket_port_get,
- nx_udp_socket_receive、nx_udp_socket_receive_notify,
- nx_udp_socket_send、nx_udp_socket_interface_send,
- nx_udp_socket_unbind、nx_udp_source_extract

nx_udp_socket_info_get

检索 UDP 套接字活动的相关信息

原型

```
UINT nx_udp_socket_info_get(
    NX_UDP_SOCKET *socket_ptr,
    ULONG *udp_packets_sent,
    ULONG *udp_bytes_sent,
    ULONG *udp_packets_received,
    ULONG *udp_bytes_received,
    ULONG *udp_packets_queued,
    ULONG *udp_receive_packets_dropped,
    ULONG *udp_checksum_errors);
```

说明

此服务会检索指定的 UDP 套接字实例的 UDP 套接字活动相关信息。

如果目标指针为 NX_NULL，则不会将该特定信息返回给调用方。

参数

- **socket_ptr** 此指针指向先前创建的 UDP 套接字实例。
- **udp_packets_sent** 此指针指向在套接字上发送的 UDP 数据包总数的目标。
- **udp_bytes_sent** 此指针指向在套接字上发送的 UDP 总字节数的目标。
- **udp_packets_received** 此指针指向在套接字上接收的 UDP 数据包总数的目标。
- **udp_bytes_received** 此指针指向在套接字上接收的 UDP 总字节数的目标。

- **udp_packets_queued** 此指针指向在套接字上排队的 UDP 数据包总数的目标。
- **udp_receive_packets_dropped** 此指针指向该套接字上因超过队列大小而被丢弃的 UDP 接收数据包总数的目标。
- **udp_checksum_errors** 此指针指向套接字上有校验和错误的 UDP 数据包总数的目标。

返回值

- **NX_SUCCESS** (0x00) 成功检索 UDP 套接字信息。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

初始化、线程和计时器

可以抢占

否

示例

```
/* Retrieve UDP socket information from socket 0.*/
status = nx_udp_socket_info_get(&socket_0,
    &udp_packets_sent,
    &udp_bytes_sent,
    &udp_packets_received,
    &udp_bytes_received,
    &udp_queued_packets,
    &udp_receive_packets_dropped,
    &udp_checksum_errors);

/* If status is NX_SUCCESS, UDP socket information was retrieved.*/
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_bytes_available、nx_udp_socket_checksum_disable,
- nx_udp_socket_checksum_enable、nx_udp_socket_create,
- nx_udp_socket_delete、nx_udp_socket_port_get,
- nx_udp_socket_receive、nx_udp_socket_receive_notify,
- nx_udp_socket_send、nx_udp_socket_interface_send,
- nx_udp_socket_unbind、nx_udp_source_extract

nx_udp_socket_port_get

选取与 UDP 套接字绑定的端口号

原型

```
UINT nx_udp_socket_port_get(NX_UDP_SOCKET *socket_ptr, UINT *port_ptr);
```

说明

此服务会检索与套接字关联的端口号；如果在绑定套接字时指定了 NX_ANY_PORT，它对查找 NetX 分配的端口非常有用。

参数

- **socket_ptr** 此指针指向先前创建的 UDP 套接字实例。
- **port_ptr** 此指针指向返回端口号的目标。有效的端口号为 1 - 0xFFFF。

返回值

- **NX_SUCCESS** (0x00) 成功绑定套接字。
- **NX_NOT_BOUND** (0x24) 此套接字未与任何端口绑定。
- **NX_PTR_ERROR** (0x07) 套接字指针或端口返回指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

否

示例

```
/* Get the port number of created and bound UDP socket. */
status = nx_udp_socket_port_get(&udp_socket, &port);

/* If status is NX_SUCCESS, the port variable contains the port this
    socket is bound to. */
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_bytes_available、nx_udp_socket_checksum_disable,
- nx_udp_socket_checksum_enable、nx_udp_socket_create,
- nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_receive、nx_udp_socket_receive_notify,
- nx_udp_socket_send、nx_udp_socket_interface_send,
- nx_udp_socket_unbind、nx_udp_source_extract

nx_udp_socket_receive

从 UDP 套接字接收数据报

原型

```
UINT nx_udp_socket_receive(
    NX_UDP_SOCKET *socket_ptr,
    NX_PACKET **packet_ptr,
    ULONG wait_option);
```

说明

此服务用于从指定的套接字接收 UDP 数据报。如果指定的套接字上数据报在排队，调用方会根据提供的等待选项挂起。

如果返回了 NX_SUCCESS，则应用程序负责在不再需要所收到的数据包时将其释放。

参数

- **socket_ptr** 此指针指向先前创建的 UDP 套接字实例。
- **packet_ptr** 此指针指向 UDP 数据报数据包指针。

- **wait_option** 定义此套接字上当前数据报在排队时此服务的行为方式。等待选项的定义如下：
- NX_NO_WAIT (0x00000000)
- NX_WAIT_FOREVER (0xFFFFFFFF)
- 以时钟周期为单位的超时值（0x00000001 到 0xFFFFFFFE）

允许调用自

线程数

可以抢占

否

示例

```
/* Receive a packet from a previously created and bound UDP socket.
    If no packets are currently available, wait for 500 timer ticks
    before giving up. */
status = nx_udp_socket_receive(&udp_socket, &packet_ptr, 500);

/* If status is NX_SUCCESS, the received UDP packet is pointed to by
    packet_ptr. */
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_bytes_available、nx_udp_socket_checksum_disable,
- nx_udp_socket_checksum_enable、nx_udp_socket_create,
- nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive_notify,
- nx_udp_socket_send、nx_udp_socket_interface_send,
- nx_udp_socket_unbind、nx_udp_source_extract

nx_udp_socket_receive_notify

在收到每个数据包时通知应用程序

原型

```
UINT nx_udp_socket_receive_notify(
    NX_UDP_SOCKET *socket_ptr,
    VOID (*udp_receive_notify)
    (NX_UDP_SOCKET *socket_ptr));
```

说明

此服务用于将接收通知函数指针设置为应用程序所指定的回调函数。然后，每当在该套接字上收到数据包时，就会调用该回调函数。如果提供了 NX_NULL 指针，则会禁用接收通知功能。

参数

- **socket_ptr** 此指针指向 UDP 套接字。
- **udp_receive_notify** 在套接字上收到数据包时调用的应用程序回调函数指针。

允许调用自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
/* Setup a receive packet callback function for the "udp_socket"
socket. */
status = nx_udp_socket_receive_notify(&udp_socket,
    my_receive_notify);

/* If status is NX_SUCCESS, NetX will call the function named
    "my_receive_notify" whenever a packet is received for
    "udp_socket". */
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_bytes_available、nx_udp_socket_checksum_disable,
- nx_udp_socket_checksum_enable、nx_udp_socket_create,
- nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive、nx_udp_socket_send,
- nx_udp_socket_interface_send、nx_udp_socket_unbind,
- nx_udp_source_extract

nx_udp_socket_send

发送 UDP 数据报

原型

```
UINT nx_udp_socket_send(
    NX_UDP_SOCKET *socket_ptr,
    NX_PACKET *packet_ptr,
    ULONG ip_address,
    UINT port);
```

说明

此服务通过先前创建并绑定的 UDP 套接字发送 UDP 数据报。NetX 会根据目标 IP 地址查找合适的本地 IP 地址作为源地址。若要指定特定的接口和源 IP 地址，应用程序应使用 nx_udp_socket_interface_send 服务。

请注意，无论是否已成功发送 UDP 数据报，此服务都会立即返回。

该套接字必须已与某个本地端口绑定。

参数

- **socket_ptr** 此指针指向先前创建的 UDP 套接字实例
- **packet_ptr** UDP 数据报数据包指针
- **ip_address** 目标 IP 地址
- **port** 有效的目标端口号介于 1 与 0xFFFF 之间，以主机字节顺序表示

返回值

- **NX_SUCCESS** (0x00) 成功发送 UDP 套接字
- **NX_NOT_BOUND** (0x24) 套接字未与任何端口绑定
- **NX_NO_INTERFACE_ADDRESS** (0x50) 找不到合适的传出接口。
- **NX_IP_ADDRESS_ERROR** (0x21) 服务器 IP 地址无效

- **NX_UNDERFLOW** (0x02) 没有足够的空间容纳数据包中的 UDP 标头
- **NX_OVERFLOW** (0x03) 数据包追加指针无效
- **NX_PTR_ERROR** (0x07) 套接字指针无效
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效
- **NX_NOT_ENABLED** (0x14) UDP 尚未启用
- **NX_INVALID_PORT** (0x46) 端口号不在有效范围内

允许调用自

线程数

可以抢占

否

示例

```
ULONG server_address;

/* Set the UDP Client IP address. */
server_address = IP_ADDRESS(1,2,3,5);

/* Send a packet to the UDP server at server_address on port 12. */
status = nx_udp_socket_send(&client_socket, packet_ptr,
    server_address, 12);

/* If status == NX_SUCCESS, the application successfully transmitted
    the packet out the UDP socket to its peer. */
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_bytes_available、nx_udp_socket_checksum_disable,
- nx_udp_socket_checksum_enable、nx_udp_socket_create,
- nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive,
- nx_udp_socket_receive_notify、nx_udp_socket_interface_send,
- nx_udp_socket_unbind、nx_udp_source_extract

nx_udp_socket_interface_send

通过 UDP 套接字发送数据报。

原型

```
UINT nx_udp_socket_interface_send(
    NX_UDP_SOCKET *socket_ptr,
    NX_PACKET *packet_ptr,
    ULONG ip_address,
    UINT port,
    UINT address_index);
```

说明

此服务以指定的 IP 地址作为源地址，通过网络接口使用先前创建并绑定的 UDP 套接字发送 UDP 数据报。请注意，无论是否已成功发送 UDP 数据报，此服务都会立即返回。

参数

- socket_ptr 要在其上传出数据包的套接字。
- packet_ptr 指向要传输的数据包的指针。
- ip_address 用于发送数据包的 IP 地址。
- port 目标端口。
- address_index 与用于发送数据包的接口相关联的地址索引。

返回值

- NX_SUCCESS (0x00) 已成功发送数据包。
- NX_NOT_BOUND (0x24) 套接字未与任何端口绑定。
- NX_IP_ADDRESS_ERROR (0x21) IP 地址无效。
- NX_NOT_ENABLED (0x14) 未启用 UDP 处理。
- NX_PTR_ERROR (0x07) 指针无效。
- NX_OVERFLOW (0x03) 数据包追加指针无效。
- NX_UNDERFLOW (0x02) 数据包前置指针无效。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_INVALID_INTERFACE (0x4C) 址索引无效。
- NX_INVALID_PORT (0x46) 端口号超出最大端口号。

允许调用自

线程数

可以抢占

否

示例

```
#define ADDRESS_INDEX 1

/* Send packet out on port 80 to the specified destination IP on the
interface at index 1 in the IP task interface list. */
status = nx_udp_packet_interface_send(socket_ptr, packet_ptr,
    destination_ip, 80,
    ADDRESS_INDEX);

/* If status is NX_SUCCESS packet was successfully transmitted. */
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_bytes_available、nx_udp_socket_checksum_disable,
- nx_udp_socket_checksum_enable、nx_udp_socket_create,
- nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive,
- nx_udp_socket_receive_notify、nx_udp_socket_send,
- nx_udp_socket_unbind

nx_udp_socket_unbind

取消 UDP 套接字与 UDP 端口的绑定。

原型

```
UINT nx_udp_socket_unbind(NX_UDP_SOCKET *socket_ptr);
```

说明

此服务用于解除 UDP 套接字与 UDP 端口之间的绑定。在取消绑定操作过程中，会释放存储在接收队列中的所有已接收的数据包。

如果有其他线程在等待将其他套接字绑定到该取消绑定的端口，则第一个挂起的线程会与新取消绑定的端口绑定。

参数

- **socket_ptr** 此指针指向先前创建的 UDP 套接字实例。

返回值

- **NX_SUCCESS** (0x00) 成功取消绑定套接字。
- **NX_NOT_BOUND** (0x24) 套接字未与任何端口绑定。
- **NX_PTR_ERROR** (0x07) 套接字指针无效。
- **NX_CALLER_ERROR** (0x11) 此服务的调用方无效。
- **NX_NOT_ENABLED** (0x14) 尚未启用此组件。

允许调用自

线程数

可以抢占

是

示例

```
/* Unbind the previously bound UDP socket. */
status = nx_udp_socket_unbind(&udp_socket);

/* If status is NX_SUCCESS, the previously bound socket is now
    unbound. */
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_bytes_available、nx_udp_socket_checksum_disable,
- nx_udp_socket_checksum_enable、nx_udp_socket_create,
- nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive,
- nx_udp_socket_receive_notify、nx_udp_socket_send,
- nx_udp_socket_interface_send、nx_udp_source_extract

nx_udp_source_extract

从 UDP 数据报提取 IP 和发送端口

原型

```
UINT nx_udp_source_extract(
    NX_PACKET *packet_ptr,
    ULONG *ip_address, UINT *port);
```

说明

此服务用于从提供的 UDP 数据报的 IP 和 UDP 标头中提取发送方的 IP 和端口号。

参数

- **packet_ptr** UDP 数据报数据包指针。
- **ip_address** 指向返回 IP 地址变量的有效指针。
- **port** 指向返回端口变量的有效指针。

返回值

- **NX_SUCCESS** (0x00) 成功提取源 IP/端口。
- **NX_INVALID_PACKET** (0x12) 提供的数据包无效。
- **NX_PTR_ERROR** (0x07) 数据包或者 IP 或端口目标无效。

允许调用自

初始化、线程、计时器、ISR

可以抢占

否

示例

```
/* Extract the IP and port information from the sender of the UPD
    packet. */
status = nx_udp_source_extract(packet_ptr, &sender_ip_address,
    &sender_port);

/* If status is NX_SUCCESS, the sender IP and port information has
    been stored in sender_ip_address and sender_port respectively.*/
```

另请参阅

- nx_udp_enable、nx_udp_free_port_find、nx_udp_info_get,
- nx_udp_packet_info_extract、nx_udp_socket_bind,
- nx_udp_socket_bytes_available、nx_udp_socket_checksum_disable,
- nx_udp_socket_checksum_enable、nx_udp_socket_create,
- nx_udp_socket_delete、nx_udp_socket_info_get,
- nx_udp_socket_port_get、nx_udp_socket_receive,
- nx_udp_socket_receive_notify、nx_udp_socket_send,
- nx_udp_socket_interface_send、nx_udp_socket_unbind

第 5 章 - NetX 网络驱动程序

2021/4/29 •

本章介绍适用于 NetX 的网络驱动程序。介绍的信息旨在帮助开发人员编写特定于应用程序的适用于 Azure RTOS NetX 的网络驱动程序。

驱动程序简介

NX_IP 结构包含用于管理单个 IP 实例的所有内容。这包括常规 TCP/IP 协议信息，以及特定于应用程序的物理网络驱动程序的入口例程。驱动程序的入口例程在 nx_ip_create 服务期间定义。可以通过 nx_ip_interface_attach 服务向 IP 实例添加其他设备。

NetX 与应用程序的网络驱动程序之间的通信是通过 NX_IP_DRIVER 请求结构来完成的。此结构通常是在调用方的堆栈上本地定义的，因此在驱动程序和调用函数返回后发布。此结构的定义如下所示。

```
typedef struct NX_IP_DRIVER_STRUCT
{
    UINT nx_ip_driver_command;
    UINT nx_ip_driver_status;
    ULONG nx_ip_driver_physical_address_msw;
    ULONG nx_ip_driver_physical_address_lsw;
    NX_PACKET *nx_ip_driver_packet;
    ULONG *nx_ip_driver_return_ptr;
    NX_IP *nx_ip_driver_ptr;
    NX_INTERFACE *nx_ip_driver_interface;
} NX_IP_DRIVER;
```

驱动程序入口

NetX 调用网络驱动程序入口函数以进行驱动程序初始化和发送数据包，以及进行各种控制和状态操作，包括初始化和启用网络设备。通过设置 *NX_IP_DRIVER* 请求结构中的 *nx_ip_driver_command* 字段，NetX 向网络驱动程序发出命令。驱动程序入口函数的格式如下：

```
VOID my_driver_entry(NX_IP_DRIVER *request);
```

驱动程序请求

NetX 使用特定命令创建驱动程序请求，并调用驱动程序入口函数以执行命令。由于每个网络驱动程序都有单个入口函数，因此 NetX 通过驱动程序请求数据结构发出所有请求。驱动程序请求数据结构 (NX_IP_DRIVER) 的 nx_ip_driver_command 成员定义请求。状态信息将报告回 nx_ip_driver_status 成员的调用方。如果此字段为 NX_SUCCESS，则驱动程序请求已成功完成。

NetX 序列化对驱动程序的所有访问。因此，驱动程序无需处理多个异步调用入口函数的线程。请注意，设备驱动程序函数在锁定了 IP 互斥的情况下执行。因此，设备驱动程序内部函数不应自行阻止。

通常，设备驱动程序还会处理中断。因此，所有驱动程序函数都需要是中断安全的。

驱动程序初始化

尽管实际驱动程序初始化处理特定于应用程序，但它通常由数据结构和物理硬件初始化组成。NetX 进行驱动程序初始化所需的信息是 IP 最大传输单元 (MTU)，它是 IP 层有效负载可用的字节数，包括 IP 标头以及物理接口是否需要逻辑到物理映射。驱动程序需求

通过在 *NX_INTERFACE* 结构的 *nx_interface_ip_mtu_size* 中设置值来配置接口 MTU。

设备驱动程序还需要在 nx_ip_interface_address_mapping_needed 中

(在 NX_INTERFACE 中)设置值来告知 NetX 是否需要接口地址映射。如果需要地址映射，驱动程序将负责配置具有有效 MAC 地址的接口，并向 NetX 提供 MAC 地址。

当网络驱动程序接收来自 NetX 的 NX_LINK INITIALIZE 请求时，它会收到一个指向 IP 控制块的指针，作为上面所示的 NX_IP_DRIVER 请求控制块的一部分。

应用程序调用 nx_ip_create 后，IP 帮助程序线程会将命令设置为 NX_LINK_INITIALIZE 的驱动程序请求发送给驱动程序，以初始化其物理网络接口。以下 NX_IP_DRIVER 成员用于初始化请求。

NX_IP_DRIVER 结	结
nx_ip_driver_command	NX_LINK_INITIALIZE
nx_ip_driver_ptr	指向 IP 实例的指针。此值应由驱动程序保存，以便驱动程序函数可以找到要操作的 IP 实例。
nx_ip_driver_interface	指向 IP 实例中的网络接口结构的指针。此信息应由驱动程序保存。接收数据包时，驱动程序应在堆栈上方发送数据包时使用接口结构信息。
nx_ip_driver_status	完成状态。如果驱动程序无法初始化 IP 实例的指定接口，则会返回非零错误状态。

> **IMPORTANT**
> 大多数驱动程序命令都是从为 IP 实例创建的 IP 帮助程序线程中调用的。因此，驱动程序例程应避免执行阻止操作，或者 IP 帮助程序线程可能会停止，从而导致依赖于 IP 线程的应用程序出现无限延迟。

启用链接

接下来，通过将驱动程序命令设置为驱动程序请求中的 NX_LINK_ENABLE 并将请求发送到网络驱动程序，IP 帮助程序线程会启用物理网络。IP 帮助程序线程完成初始化请求后不久将发生这种情况。启用链接可能与在接口实例中设置 nx_interface_link_up 字段一样简单。但它可能还涉及物理硬件的操作。以下 NX_IP_DRIVER 成员用于启用链接请求。

NX_IP_DRIVER 结	结
nx_ip_driver_command	NX_LINK_ENABLE
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法启用指定的接口，则会返回非零错误状态。

禁用链接

此请求在 *nx_ip_delete_ 服务删除 IP 实例的过程中由 NetX 发出。或者，为了暂时禁用链接以节能，应用程序可能会发出此命令。此服务禁用 IP 实例上的物理网络接口。禁用链接的处理可能与清除接口实例中的 nx_interface_link_up* 标志一样简单。但它可能还涉及物理硬件的操作。通常，它是 *启用链接操作的反向操作。禁用链接后，应用程序请求 _启用链接* 操作以启用接口。以下 NX_IP_DRIVER 成员用于禁用链接请求。

NX_IP_DRIVER 结构	结构
nx_ip_driver_command	NX_LINK_DISABLE
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法禁用 IP 实例中的指定接口，则会返回非零错误状态。

取消初始化链接

此请求在 nx_ip_delete 服务删除 IP 实例的过程中由 NetX 发出。此请求取消初始化接口，并释放在初始化阶段创建的任何资源。通常，它是初始化链接操作的反向操作。取消初始化接口后，将无法使用设备，直到再次初始化该接口。

以下 NX_IP_DRIVER 成员用于禁用链接请求。

NX_IP_DRIVER 结构	结构
nx_ip_driver_command	NX_LINK_UNINITIALZE
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法取消初始化 IP 实例的指定接口，则会返回非零错误状态。

数据包发送

此请求是在进行内部 IP 发送处理期间发出的，所有 NetX 协议都使用该过程传输数据包（ARP、RARP 除外）。接收数据包发送命令后，nx_packet_prepend_ptr 会指向要发送的数据包的开头，即 IP 标头的开头。nx_packet_length 指示正在传输的数据的总大小（以字节为单位）。如果 nx_packet_next 是有效的，则传出 IP 数据报将存储在多个数据包中，需要驱动程序才能跟踪链式数据包并传输整个帧。请注意，每个链式数据包中的有效数据区域存储在 nx_packet_prepend_ptr 和 nx_packet_append_ptr 之间。

驱动程序负责构造物理标头。如果需要物理地址到 IP 地址的映射（如以太网），则 IP 层已解析 MAC 地址。目标 MAC 地址是从存储在 nx_ip_driver_physical_address_msw 和 nx_ip_driver_physical_address_lsw 中的 IP 实例传递的。

添加物理标头后，数据包发送处理将调用驱动程序的输出函数以传输数据包。

以下 NX_IP_DRIVER 成员用于数据包发送请求。

NX_IP_DRIVER 结构	结构
nx_ip_driver_command	NX_LINK_PACKET_SEND
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_packet	指向要发送的数据包的指针
nx_ip_driver_interface	指向接口实例的指针。

NX_IP_DRIVER 结构	说明
nx_ip_driver_physical_address_msw	物理地址的最高有效 32 位（仅当需要物理映射时）
nx_ip_driver_physical_address_lsw	物理地址的最低有效 32 位（仅当需要物理映射时）
nx_ip_driver_status	完成状态。如果驱动程序无法发送数据包，则会返回非零错误状态。

数据包广播

此请求与发送数据包请求几乎完全相同。唯一的区别是，目标物理地址字段设置为以太网广播 MAC 地址。以下 NX_IP_DRIVER 成员用于数据包广播请求。

NX_IP_DRIVER 结构	说明
nx_ip_driver_command	NX_LINK_PACKET_BROADCAST
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_packet	指向要结束的数据包的指针
nx_ip_driver_physical_address_ms w	0x0000FFFF（广播）
nx_ip_driver_physical_address_lsw	0xFFFFFFFF（广播）
nx_ip_driver_interface	指向接口实例的指针。
nx_ip_driver_status	完成状态。如果驱动程序无法发送数据包，则会返回非零错误状态。

ARP 发送

此请求还类似于 IP 数据包发送请求。唯一的区别是，以太网标头指定 ARP 数据包（而不是 IP 数据包），并且目标物理地址字段设置为 MAC 广播地址。以下 NX_IP_DRIVER 成员用于 ARP 发送请求。

NX_IP_DRIVER 结构	说明
nx_ip_driver_command	NX_LINK_ARP_SEND
nx_ip_driver_ptr	指向 IP 实例的指针。
nx_ip_driver_packet	指向要发送的数据包的指针。
nx_ip_driver_physical_address_msw	0x0000FFFF（广播）
nx_ip_driver_physical_address_lsw	0xFFFFFFFF（广播）
nx_ip_driver_interface	指向接口实例的指针。
nx_ip_driver_status	完成状态。如果驱动程序无法发送 ARP 数据包，则会返回非零错误状态。

如果不需要物理映射，则不需要实现此请求。

ARP 响应发送

此请求与 ARP 发送数据包请求几乎完全相同。唯一的区别是，目标物理地址字段是从 IP 实例传递的。以下 NX_IP_DRIVER 成员用于 ARP 响应发送请求。

NX_IP_DRIVER 结构	结构
nx_ip_driver_command	NX_LINK_ARP_RESPONSE_SEND
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_packet	指向要发送的数据包的指针
nx_ip_driver_physical_address_msw	物理地址的最高有效 32 位
nx_ip_driver_physical_address_lsw	物理地址的最低有效 32 位
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法发送 ARP 数据包，则会返回非零错误状态。

> **NOTE**
> 如果不需要物理映射，则不需要实现此请求。

RARP 发送

此请求与 ARP 发送数据包请求几乎完全相同。唯一的区别是，不需要数据包标头的类型和物理地址字段，因为物理目标始终是广播地址。

以下 NX_IP_DRIVER 成员用于 RARP 发送请求。

NX_IP_DRIVER 结构	结构
nx_ip_driver_command	NX_LINK_RARP_SEND
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_packet	指向要发送的数据包的指针
nx_ip_driver_physical_address_msw	0x0000FFFF（广播）
nx_ip_driver_physical_address_lsw	0xFFFFFFFF（广播）
NX_IP_DRIVER 成员	含义
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法发送 RARP 数据包，则会返回非零错误状态。

> **NOTE**
> 需要 RARP 服务的应用程序必须实现此命令。

多播组联接

此请求是通过 nx_igmp_multicast_interface 联接服务发出的。网络驱动程序使用提供的多播组地址，并设置物理媒体以接受来自该多播组地址的传入数据包。请注意，对于不支持多播筛选器的驱动程序，驱动程序接收逻辑可能必须处于混杂模式。在这种情况下，驱动程序可能需要根据目标 MAC 地址筛选传入帧，从而减少传递到 IP 实例的流量。以下 NX_IP_DRIVER 成员用于多播组联接请求。

NX_IP_DRIVER 结构	值
nx_ip_driver_command	NX_LINK_MULTICAST_JOIN
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_physical_address_msw	物理多播地址的最高有效 32 位
nx_ip_driver_physical_address_lsw	物理多播地址的最低有效 32 位
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法联接多播组，则会返回非零错误状态。

多播组离开

此请求是通过显式调用 nx_igmp_multicast_leave 服务来调用的。驱动程序从多播列表中删除提供的以太网多播地址。主机离开多播组后，此 IP 实例将不再收到网络上带有此以太网多播地址的数据包。以下 NX_IP_DRIVER 成员用于多播组离开请求。

NX_IP_DRIVER 结构	值
nx_ip_driver_command	NX_LINK_MULTICAST_LEAVE
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_physical_address_msw	物理多播地址的最高有效 32 位
nx_ip_driver_physical_address_lsw	物理多播地址的最低有效 32 位
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法离开多播组，则会返回非零错误状态。

附加接口

此请求从 NetX 调用到设备驱动程序，允许驱动程序将驱动程序实例与 IP 中的相应 IP 实例和物理接口实例相关联。以下 NX_IP_DRIVER 成员用于附加接口请求。

NX_IP_DRIVER 结构	值
nx_ip_driver_command	X_LINK_INTERFACE_ATTACH

NX_IP_DRIVER 成员	说明
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_interface	指向接口实例的指针。
nx_ip_driver_status	完成状态。如果驱动程序无法分离 IP 实例的指定接口，则会返回非零错误状态。

分离接口

此请求由 NetX 调用到设备驱动程序，允许驱动程序将驱动程序实例与 IP 中的相应 IP 实例和物理接口实例解除关联。以下 NX_IP_DRIVER 成员用于附加接口请求。

NX_IP_DRIVER 成员	说明
nx_ip_driver_command	NX_LINK_INTERFACE_DETACH
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_interface	指向接口实例的指针。
nx_ip_driver_status	完成状态。如果驱动程序无法附加 IP 实例的指定接口，则会返回非零错误状态。

获取链接状态

应用程序可以使用 NetX 服务 nx_ip_interface_status_check 服务为主机上的任何接口查询网络接口链接状态。有关这些服务的更多详细信息，请参阅第 107 页上的第 4 章“NetX 服务的说明”。

链接状态包含在 nx_ip_driver_interface 指针所指向的 NX_INTERFACE 结构的 nx_interface_link_up 字段中。以下 NX_IP_DRIVER 成员用于链接状态请求。

NX_IP_DRIVER 成员	说明
nx_ip_driver_command	NX_LINK_GET_STATUS
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_return_ptr	指向要放置状态的目标的指针。
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法获取特定状态，则会返回非零错误状态。

> **NOTE**
>
> nx_ip_status_check 仍可用于检查主要接口的状态。但是，建议应用程序开发人员使用特定于接口的服务 nx_ip_interface_status_check。

获取链接速度

此请求是从 nx_ip_driver_direct_command 服务内发出的。驱动程序将链接的线速度存储在提供的目标中。以下

NX_IP_DRIVER 成员用于链接线速度请求。

NX_IP_DRIVER 结构	说明
nx_ip_driver_command	NX_LINK_GET_SPEED
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_return_ptr	指向要放置线速度的目标的指针
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法获取速度信息，则会返回非零错误状态。

> **NOTE**
> 此请求不是由 NetX 在内部使用的，因此其实现是可选的。

获取双工类型

此请求是从 nx_ip_driver_direct_command 服务内发出的。驱动程序将链接的双工类型存储在提供的目标中。以下 NX_IP_DRIVER 成员用于双工类型请求。

NX_IP_DRIVER 结构	说明
nx_ip_driver_command	NX_LINK_GET_DUPLEX_TYPE
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_return_ptr	指向要放置双工类型的目标的指针
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法获取双工信息，则会返回非零错误状态。

> **NOTE**
> 此请求不是由 NetX 在内部使用的，因此其实现是可选的。

获取错误计数

此请求是从 nx_ip_driver_direct_command 服务内发出的。驱动程序将链接的错误计数存储在提供的目标中。若要支持此功能，驱动程序需要跟踪操作错误。以下 NX_IP_DRIVER 成员用于链接错误计数请求。

NX_IP_DRIVER 结构	说明
nx_ip_driver_command	NX_LINK_GET_ERROR_COUNT
nx_ip_driver_ptr	指向 IP 实例的指针

NX_IP_DRIVER 结构	说明
nx_ip_driver_return_ptr	指向要放置错误计数的目标的指针
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法获取错误计数，则会返回非零错误状态。

> **NOTE**
> 此请求不是由 NetX 在内部使用的，因此其实现是可选的。

获取接收数据包计数

此请求是从 nx_ip_driver_direct_command 服务内发出的。驱动程序将链接的接收数据包计数存储在提供的目标中。若要支持此功能，驱动程序需要跟踪接收到的数据包数。以下 NX_IP_DRIVER 成员用于链接接收数据包计数请求。

NX_IP_DRIVER 结构	说明
nx_ip_driver_command	NX_LINK_GET_RX_COUNT
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_return_ptr	指向放置接收数据包计数的目标的指针
nx_ip_driver_interface	指向物理网络接口的指针
nx_ip_driver_status	完成状态。如果驱动程序无法获取接收计数，则会返回非零错误状态。

> **NOTE**
> 此请求不是由 NetX 在内部使用的，因此其实现是可选的。

获取传输数据包计数

此请求是从 nx_ip_driver_direct_command 服务内发出的。驱动程序将链接的传输数据包计数存储在提供的目标中。若要支持此功能，驱动程序需要跟踪每个接口上传输的每个数据包。以下 NX_IP_DRIVER 成员用于链接传输数据包计数请求。

NX_IP_DRIVER 结构	说明
nx_ip_driver_command	NX_LINK_GET_TX_COUNT
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_return_ptr	指向放置传输数据包计数的目标的指针
nx_ip_driver_interface	指向接口实例的指针

NX_IP_DRIVER 结构	描述
nx_ip_driver_status	完成状态。如果驱动程序无法获取传输计数，则会返回非零错误状态。

> **NOTE**
> 此请求不是由 NetX 在内部使用的，因此其实现是可选的。

获取分配错误

此请求是从 nx_ip_driver_direct_command 服务内发出的。驱动程序将链接的数据包池分配错误计数存储在提供的目标中。以下 NX_IP_DRIVER 成员用于链接分配错误计数请求。

NX_IP_DRIVER 结构	描述
nx_ip_driver_command	NX_LINK_GET_ALLOC_ERRORS
nx_ip_driver_ptr	指向 IP 实例的指针

NX_IP_DRIVER 结构	描述
nx_ip_driver_return_ptr	指向要放置分配错误计数的目标的指针
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法获取分配错误，则会返回非零错误状态。

此请求不是由 NetX 在内部使用的，因此其实现是可选的。

驱动程序延迟处理

此请求是从 IP 帮助程序线程发出的，以响应从传输或接收 ISR 调用 _nx_ip_driver_deferred_processing 例程的驱动程序。这允许驱动程序 ISR 将数据包接收和传输处理延迟到 IP 帮助程序线程，从而减少要在 ISR 中处理的数量。nx_ip_driver_interface 所指向的 NX_INTERFACE 结构中的 nx_interface_additional_link_info 字段可由驱动程序用来存储有关来自 IP 帮助程序线程上下文的延迟处理事件的信息。以下 NX_IP_DRIVER 成员用于延迟处理事件。

NX_IP_DRIVER

结构	描述
nx_ip_driver_command	NX_LINK_DEFERRED_PROCESSING
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_interface	指向接口实例的指针

用户命令

此请求是从 nx_ip_driver_direct_command 服务内发出的。驱动程序处理特定于应用程序的用户命令。以下 NX_IP_DRIVER 成员用于用户命令请求。

NX_IP_DRIVER 结构	描述
nx_ip_driver_command	NX_LINK_USER_COMMAND
nx_ip_driver_ptr	指向 IP 实例的指针
nx_ip_driver_return_ptr	用户定义
nx_ip_driver_interface	指向接口实例的指针
nx_ip_driver_status	完成状态。如果驱动程序无法执行用户命令，则会返回非零错误状态。

此请求不是由 NetX 在内部使用的，因此其实现是可选的。

未实现的命令

网络驱动程序未实现的命令必须将返回状态字段设置为 NX_UNHANDLED_COMMAND。

驱动程序输出

前面提到的所有数据包传输请求都需要驱动程序中实现的输出函数。特定传输逻辑特定于硬件，但它通常包含检查硬件容量以便立即发送数据包。如果可能，数据包有效负载（以及数据包链中的其他有效负载）会加载到一个或多个硬件传输缓冲区中，并启动传输操作。如果数据包不适合可用的传输缓冲区，则数据包应排队，并在传输缓冲区可用时进行传输。

建议的传输队列是单向链接的列表，同时包含头和尾指针。新数据包将添加到队列的末尾，并将最旧的数据包置于前端。nx_packet_queue_next 字段用作队列中数据包的下一个链接。驱动程序定义传输队列的头和尾指针。

由于此队列是从驱动程序的线程和中断部分访问的，因此，必须围绕队列操作放置中断保护。

大多数物理硬件实现都会在数据包传输完成时生成中断。当驱动程序收到此类中断时，它通常会释放与正在传输的数据包关联的资源。如果传输逻辑直接从 NX_PACKET 缓冲区读取数据，则驱动程序应使用 nx_packet_transmit_release 服务将与传输完成中断关联的数据包释放回可用的数据包池。接下来，驱动程序会检查传输队列中是否有其他等待发送的数据包。适合硬件传输缓冲区的多个排队传输数据包将被取消排队并加载到缓冲区中。然后会启动另一个发送操作。

一旦将 NX_PACKET 中的数据移动到发送器 FIFO 中（或者如果驱动程序支持 zerocopy 操作，则已传输 NX_PACKET 中的数据），则在调用 nx_packet_transmit_release 之前，驱动程序必须将 nx_packet_prepend_ptr 移动到 IP 标头的开头。请记住相应地调整 nx_packet_length 字段。如果 IP 帧由多个数据包组成，则只需释放数据包链的头。

驱动程序输入

接收收到的数据包中断后，网络驱动程序将从物理硬件接收缓冲区检索数据包，并生成有效的 NetX 数据包。生成有效的 NetX 数据包包括设置适当的长度字段，以及在传入数据包的大小大于单个数据包有效负载时将多个数据包链接在一起。正确生成后，会将 prepend_ptr 移动到物理层标头之后，并将接收数据包调度到 NetX。

NetX 假定 IP 和 ARP 标头在 ULONG 边界上对齐。因此，NetX 驱动程序必须确保此对齐方式。在以太网环境中，这是通过从数据包开头的两个字节开始使用以太网标头来完成的。将 nx_packet_prepend_ptr 移动到以太网标头之外时，基础 IP 或 ARP 标头采用 4 字节对齐。

NetX 中提供了几个接收数据包函数。如果接收到的数据包是 ARP 数据包，则调用 nx_arp_packet_deferred_receive *。如果接收到的数据包是 RARP 数据包，则调用 _nx_rarp_packet_deferred_receive。有几个选项可用于处理传入 IP 数据包。为了以最快的速度处理 IP 数据包，将调用 _nx_ip_packet_receive。此方法的开销最少，但在驱动程序的接收中断服务处理程序 (ISR) 中需要更多处

理。对于最小 ISR 处理，将调用 __nx_ip_packet_deferred_receive*。

正确生成新的接收数据包后，将设置物理硬件的接收缓冲区以接收更多数据。这可能需要分配 NetX 数据包并在硬件接收缓冲区中放置有效负载地址，或者只需更改硬件接收缓冲区中的设置。若要最大限度地减少溢出可能性，请务必在收到数据包后，尽快获取硬件的可用接收缓冲区。

在驱动程序初始化过程中设置初始接收缓冲区。

延迟接收数据包处理

驱动程序可能会将接收数据包处理延迟到 NetX IP 帮助程序线程。对于某些应用程序，这可能是最大限度地减少 ISR 处理和丢弃的数据包所必需的。若要使用延迟数据包处理，必须先在定义了 *NX_DRIVER_DEFERRED_PROCESSING_ 的情况下编译 NetX 库。这会将延迟数据包逻辑添加到 NetX IP 帮助程序线程。接下来，在接收数据包时，驱动程序必须调用 __nx_ip_packet_deferred_receive():*

```
_nx_ip_packet_deferred_receive(ip_ptr, packet_ptr);
```

延迟接收函数将 packet_ptr 表示的接收数据包放置在 FIFO（链接列表）中，并通知 IP 帮助程序线程。执行后，IP 帮助程序重复调用延迟处理函数来处理每个延迟数据包。延迟处理程序处理通常包括删除数据包的物理层标头（通常为以太网），并将其调度到以下 NetX 接收函数之一：

_nx_ip_packet_deferred_receive
_nx_arp_packet_deferred_receive
_nx_rarp_packet_deferred_receive

示例 RAM 以太网网络驱动程序

NetX 演示系统附带了一个基于 RAM 的小型网络驱动程序，该驱动程序在文件 nx_ram_network_driver.c 中定义。此驱动程序假设 IP 实例都位于同一网络上，只需在创建虚拟硬件地址（MAC 地址）后将其分配给每个设备实例。此文件提供了 NetX 物理网络驱动程序的基本结构的典型示例。用户可以使用此示例中提供的驱动程序框架开发自己的网络驱动程序。

网络驱动程序的入口函数为 *_nx_ram_network_driver_，该函数传递到 IP 实例创建调用。其他网络接口的入口函数可以传递到 nx_ip_interface_attach 服务。在 IP 实例开始运行后，将调用驱动程序入口函数以初始化和启用设备（请参阅 _NX_LINK_INITIALIZE* 和 NX_LINK_ENABLE）。发出 NX_LINK_ENABLE 命令后，设备应准备好传输和接收数据包。

IP 实例通过以下命令之一传输网络数据包：

NX_LINK_PACKET_SEND	IP ,
NX_LINK_ARP_SEND	正在传输 ARP 请求或 ARP 响应数据包，
NX_LINK_ARP_RARP_SEND	正在传输反向 ARP 请求或响应数据包，

处理这些命令时，网络驱动程序需要预置适当的以太网框架标头，然后将其发送到基础硬件进行传输。在传输过程中，网络驱动程序具有数据包缓冲区的独占所有权。因此，在传输数据后（或在将数据复制到驱动程序内部传输缓冲区后），网络驱动程序将负责通过先将超过以太网标头的预置指针移动到 IP 标头（并相应地调整数据包长度），然后通过调用 nx_packet_transmit_release 服务释放数据包，从而释放数据包缓冲区。在数据传输后不释放数据包将导致数据包泄露。

网络设备驱动程序还负责处理传入数据包。在 RAM 驱动程序示例中，接收到的数据包由函数 _nx_ram_network_driver_receive 处理。

设备收到以太网帧后，驱动程序就会负责将数据存储在

NX_PACKET 结构中。请注意，NetX 假设 IP 标头以 4 字节对齐地址开头。由于以太网标头的长度为 14 个字节，

因此驱动程序需要在 2 字节对齐地址上存储以太网标头的开头，以保证 IP 标头以 4 字节对齐地址开头。

附录 A - Azure RTOS NetX 服务

2021/4/29 •

地址解析协议 (ARP)

```
UINT nx_arp_dynamic_entries_invalidate(NX_IP *ip_ptr);
UINT nx_arp_dynamic_entry_set(
    NX_IP *ip_ptr,
    ULONG ip_address,
    ULONG physical_msw,
    ULONG physical_lsw);
UINT nx_arp_enable(
    NX_IP *ip_ptr,
    VOID *arp_cache_memory,
    ULONG arp_cache_size);
UINT nx_arp_gratuitous_send(
    NX_IP *ip_ptr,
    VOID (*response_handler)(NX_IP *ip_ptr,NX_PACKET *packet_ptr));
UINT nx_arp_hardware_address_find(
    NX_IP *ip_ptr,
    ULONG ip_address, ULONG*physical_msw,
    ULONG *physical_lsw);
UINT nx_arp_info_get(
    NX_IP *ip_ptr,
    ULONG *arp_requests_sent,
    ULONG*arp_requests_received,
    ULONG *arp_responses_sent,
    ULONG*arp_responses_received,
    ULONG *arp_dynamic_entries,
    ULONG *arp_static_entries,
    ULONG *arp_aged_entries,
    ULONG *arp_invalid_messages);
UINT nx_arp_ip_address_find(
    NX_IP *ip_ptr,
    ULONG *ip_address,
    ULONG physical_msw,
    ULONG physical_lsw);
UINT nx_arp_static_entries_delete(NX_IP *ip_ptr);
UINT nx_arp_static_entry_create(
    NX_IP *ip_ptr,
    ULONG ip_address,
    ULONG physical_msw,
    ULONG physical_lsw);
UINT nx_arp_static_entry_delete(
    NX_IP *ip_ptr,
    ULONG ip_address,
    ULONG physical_msw,
    ULONG physical_lsw);
```

Internet 控制消息协议 (ICMP)

```
UINT nx_icmp_enable(NX_IP *ip_ptr);
UINT nx_icmp_info_get(
    NX_IP *ip_ptr,
    ULONG *pings_sent,
    ULONG *ping_timeouts,
    ULONG *ping_threads_suspended,
    ULONG *ping_responses_received,
    ULONG *icmp_checksum_errors,
     ULONG *icmp_unhandled_messages);
UINT nx_icmp_ping(
    NX_IP *ip_ptr,
    ULONG ip_address,
    CHAR *data,
    ULONG data_size,
    NX_PACKET **response_ptr, ULONG wait_option);
```

Internet 组管理协议 (IGMP)

```
UINT nx_igmp_enable(NX_IP *ip_ptr);
UINT nx_igmp_info_get(
    NX_IP *ip_ptr,
    ULONG *igmp_reports_sent,
    ULONG *igmp_queries_received,
    ULONG *igmp_checksum_errors,
    ULONG *current_groups_joined);
UINT nx_igmp_loopback_disable(NX_IP *ip_ptr);
UINT nx_igmp_loopback_enable(NX_IP *ip_ptr);
UINT nx_igmp_multicast_interface_join(
    NX_IP *ip_ptr,
    ULONG group_address,
    UINT interface_index);
UINT nx_igmp_multicast_join(
    NX_IP *ip_ptr,
    ULONG group_address);
UINT nx_igmp_multicast_leave(
    NX_IP *ip_ptr,
    ULONG group_address);
```

Internet 协议 (IP)

```
UINT nx_ip_address_change_notify(
    NX_IP *ip_ptr,
    VOID (*change_notify)(NX_IP *, VOID *),
    VOID *additional_info);
UINT nx_ip_address_get(
    NX_IP *ip_ptr,
    ULONG *ip_address,
    ULONG *network_mask);
UINT nx_ip_address_set(
    NX_IP *ip_ptr,
    ULONG ip_address,
    ULONG network_mask);
UINT nx_ip_create(
    NX_IP *ip_ptr,
    CHAR *name,
    ULONG ip_address,
    ULONG network_mask,
    NX_PACKET_POOL *default_pool,
    VOID (*ip_network_driver)(NX_IP_DRIVER *),
    VOID *memory_ptr,
    ULONG memory_size,
    UINT priority);
```

```
UINT nx_ip_delete(NX_IP *ip_ptr);
UINT nx_ip_driver_direct_command(
    NX_IP *ip_ptr,
    UINT command,
    ULONG *return_value_ptr);
UINT nx_ip_driver_interface_direct_command(
    NX_IP *ip_ptr,
    UINT command,
    UINT interface_index,
    ULONG
    *return_value_ptr);
UINT nx_ip_forwarding_disable(NX_IP *ip_ptr);
UINT nx_ip_forwarding_enable(NX_IP *ip_ptr);
UINT nx_ip_fragment_disable(NX_IP *ip_ptr);
UINT nx_ip_fragment_enable(NX_IP *ip_ptr);
UINT nx_ip_gateway_address_set(NX_IP *ip_ptr,
    ULONG ip_address);
UINT nx_ip_info_get(NX_IP *ip_ptr,
    ULONG *ip_total_packets_sent,
    ULONG *ip_total_bytes_sent,
    ULONG *ip_total_packets_received,
    ULONG *ip_total_bytes_received,
    ULONG *ip_invalid_packets,
    ULONG *ip_receive_packets_dropped,
    ULONG *ip_receive_checksum_errors,
    ULONG *ip_send_packets_dropped,
    ULONG *ip_total_fragments_sent,
    ULONG *ip_total_fragments_received);
UINT nx_ip_interface_address_get(
    NX_IP *ip_ptr,
    ULONG interface_index,
    ULONG *ip_address,
    ULONG *network_mask);
UINT nx_ip_interface_address_set(
    NX_IP *ip_ptr,
    ULONG interface_index,
    ULONG ip_address, ULONG
    network_mask);
UINT nx_ip_interface_attach(
    NX_IP *ip_ptr,
    CHAR* interface_name,
    ULONG ip_address,
    ULONG network_mask,
    VOID (*ip_link_driver)(struct NX_IP_DRIVER_STRUCT *));
UINT nx_ip_interface_info_get(
    NX_IP *ip_ptr,
    UINT interface_index,
    CHAR **interface_name,
    ULONG *ip_address,
    ULONG *network_mask,
    ULONG *mtu_size,
    ULONG *physical_address_msw,
    ULONG *physical_address_lsw);
UINT nx_ip_interface_status_check(
    NX_IP *ip_ptr,
    UINT interface_index,
    ULONG needed_status,
    ULONG *actual_status,
    ULONG wait_option);
UINT nx_ip_link_status_change_notify_set(
    NX_IP *ip_ptr,
    VOID (*link_status_change_notify)(NX_IP *ip_ptr, UINT interface_index, UINT link_up));
UINT nx_ip_raw_packet_disable(NX_IP *ip_ptr);
UINT nx_ip_raw_packet_enable(NX_IP *ip_ptr);
UINT nx_ip_raw_packet_interface_send(
    NX_IP *ip_ptr,
    NX_PACKET *packet_ptr,
    ULONG destination_ip,
    UINT interface_index,
```

```
    ULONG type_of_service);
UINT nx_ip_raw_packet_receive(
    NX_IP *ip_ptr,
    NX_PACKET **packet_ptr,
    ULONG wait_option);
UINT nx_ip_raw_packet_send(
    NX_IP *ip_ptr,
    NX_PACKET *packet_ptr,
    ULONG destination_ip,
    ULONG type_of_service);
UINT nx_ip_static_route_add(
    NX_IP *ip_ptr,
    ULONG network_address,
    ULONG net_mask,
    ULONG next_hop);
UINT nx_ip_static_route_delete(
    NX_IP *ip_ptr,
    ULONG network_address,
    ULONG net_mask);
UINT nx_ip_status_check(
    NX_IP *ip_ptr,
    ULONG needed_status,
    ULONG *actual_status,
    ULONG wait_option);
```

数据包管理

```
UINT nx_packet_allocate(
    NX_PACKET_POOL *pool_ptr,
    NX_PACKET **packet_ptr,
    ULONG packet_type,
    ULONG wait_option);
UINT nx_packet_copy(
    NX_PACKET *packet_ptr,
    NX_PACKET **new_packet_ptr,
    NX_PACKET_POOL *pool_ptr,
    ULONG wait_option);
UINT nx_packet_data_append(
    NX_PACKET *packet_ptr,
    VOID *data_start,
    ULONG data_size,
    NX_PACKET_POOL *pool_ptr,
    ULONG wait_option);
UINT nx_packet_data_extract_offset(
    NX_PACKET *packet_ptr,
    ULONG offset,
    VOID *buffer_start,
    ULONG buffer_length,
    ULONG *bytes_copied);
UINT nx_packet_data_retrieve(
    NX_PACKET *packet_ptr,
    VOID *buffer_start,
    ULONG *bytes_copied);
UINT nx_packet_length_get(
    NX_PACKET *packet_ptr,
    ULONG *length);
UINT nx_packet_pool_create(
    NX_PACKET_POOL *pool_ptr,
    CHAR *name,
    ULONG block_size,
    VOID *memory_ptr,
    ULONG memory_size);
UINT nx_packet_pool_delete(NX_PACKET_POOL *pool_ptr);
UINT nx_packet_pool_info_get(
    NX_PACKET_POOL *pool_ptr,
    ULONG *total_packets,
    ULONG *free_packets,
    ULONG *empty_pool_requests,
    ULONG *empty_pool_suspensions,
    ULONG *invalid_packet_releases);
UINT nx_packet_release(NX_PACKET *packet_ptr);
UINT nx_packet_transmit_release(NX_PACKET *packet_ptr);
```

反向地址解析协议 (RARP)

```
UINT nx_rarp_disable(NX_IP *ip_ptr);
UINT nx_rarp_enable(NX_IP *ip_ptr);
UINT nx_rarp_info_get(
    NX_IP *ip_ptr,
    ULONG *rarp_requests_sent,
    ULONG *rarp_responses_received,
    ULONG *rarp_invalid_messages);
```

系统管理

```
VOID nx_system_initialize(VOID);
```

传输**控制**协议 (TCP)

```
UINT nx_tcp_client_socket_bind(
    NX_TCP_SOCKET *socket_ptr,
    UINT port, ULONG wait_option);
UINT nx_tcp_client_socket_connect(NX_TCP_SOCKET *socket_ptr,
    ULONG server_ip, UINT server_port,
    ULONG wait_option);
UINT nx_tcp_client_socket_port_get(
    NX_TCP_SOCKET *socket_ptr,
    UINT *port_ptr);
UINT nx_tcp_client_socket_unbind(NX_TCP_SOCKET *socket_ptr);
UINT nx_tcp_enable(NX_IP *ip_ptr);
UINT nx_tcp_free_port_find(
    NX_IP *ip_ptr,
    UINT port,
    UINT *free_port_ptr);
UINT nx_tcp_info_get(
    NX_IP *ip_ptr,
    ULONG *tcp_packets_sent,
    ULONG *tcp_bytes_sent,
    ULONG *tcp_packets_received,
    ULONG *tcp_bytes_received,
    ULONG *tcp_invalid_packets,
    ULONG *tcp_receive_packets_dropped,
    ULONG *tcp_checksum_errors,
    ULONG *tcp_connections,
    ULONG *tcp_disconnections,
    ULONG *tcp_connections_dropped,
    ULONG*tcp_retransmit_packets);
UINT nx_tcp_server_socket_accept(
    NX_TCP_SOCKET *socket_ptr,
    ULONG wait_option);
UINT nx_tcp_server_socket_listen(
    NX_IP *ip_ptr,
    UINT port, NX_TCP_SOCKET *socket_ptr,
    UINT listen_queue_size,
    VOID (*tcp_listen_callback)(NX_TCP_SOCKET *socket_ptr, UINT port));
UINT nx_tcp_server_socket_relisten(
    NX_IP *ip_ptr,
    UINT port,
    NX_TCP_SOCKET *socket_ptr);
UINT nx_tcp_server_socket_unaccept(NX_TCP_SOCKET *socket_ptr);
UINT nx_tcp_server_socket_unlisten(NX_IP *ip_ptr, UINT port);
UINT nx_tcp_socket_bytes_available(
    NX_TCP_SOCKET *socket_ptr,
    ULONG *bytes_available);
UINT nx_tcp_socket_create(
    NX_IP *ip_ptr,
    NX_TCP_SOCKET *socket_ptr,
    CHAR *name,
    ULONG type_of_service,
    ULONG fragment,
    UINT time_to_live,
    ULONG window_size,
    VOID (*tcp_urgent_data_callback)(NX_TCP_SOCKET*socket_ptr),
    VOID (*tcp_disconnect_callback)(NX_TCP_SOCKET *socket_ptr));
UINT nx_tcp_socket_delete(NX_TCP_SOCKET *socket_ptr);
UINT nx_tcp_socket_disconnect(
    NX_TCP_SOCKET *socket_ptr,
    ULONG wait_option);
UINT nx_tcp_socket_establish_notify(
    NX_TCP_SOCKET *socket_ptr,
    VOID *tcp_establish_notify)(NX_TCP_SOCKET *socket_ptr));
UINT nx_tcp_socket_disconnect_complete_notify(
    NX_TCP_SOCKET *socket_ptr,
    VOID (*tcp_disconnect_complete_notify)(NX_TCP_SOCKET *socket_ptr));
```

```
UINT nx_tcp_socket_timed_wait_callback(
    NX_TCP_SOCKET *socket_ptr,
    VOID (*tcp_timed_wait_callback)(NX_TCP_SOCKET *socket_ptr));
UINT nx_tcp_socket_info_get(
    NX_TCP_SOCKET *socket_ptr,
    ULONG *tcp_packets_sent,
    ULONG *tcp_bytes_sent,
    ULONG *tcp_packets_received,
    ULONG *tcp_bytes_received,
    ULONG *tcp_retransmit_packets,
    ULONG *tcp_packets_queued,
    ULONG *tcp_checksum_errors,
    ULONG *tcp_socket_state,
    ULONG *tcp_transmit_queue_depth,
    ULONG *tcp_transmit_window,
    ULONG *tcp_receive_window);
UINT nx_tcp_socket_mss_get(
    NX_TCP_SOCKET *socket_ptr,
    ULONG *mss);
UINT nx_tcp_socket_mss_peer_get(
    NX_TCP_SOCKET *socket_ptr,
    ULONG *peer_mss);
UINT nx_tcp_socket_mss_set(
    NX_TCP_SOCKET *socket_ptr,
    ULONG mss);
UINT nx_tcp_socket_peer_info_get(
    NX_TCP_SOCKET *socket_ptr,
    ULONG *peer_ip_address,
    ULONG *peer_port);
UINT nx_tcp_socket_receive(
    NX_TCP_SOCKET *socket_ptr,
N    X_PACKET **packet_ptr,
    ULONG wait_option);
UINT nx_tcp_socket_receive_notify(
    NX_TCP_SOCKET *socket_ptr,
    VOID (*tcp_receive_notify)(NX_TCP_SOCKET *socket_ptr));
UINT nx_tcp_socket_send(
    NX_TCP_SOCKET *socket_ptr,
    NX_PACKET *packet_ptr,
    ULONG wait_option);
UINT nx_tcp_socket_state_wait(
    NX_TCP_SOCKET *socket_ptr,
    UINT desired_state,
    ULONG wait_option);
UINT nx_tcp_socket_transmit_configure(
    NX_TCP_SOCKET *socket_ptr,
    ULONG max_queue_depth,
    ULONG timeout,
    ULONG max_retries,
    ULONG timeout_shift);
UINT nx_tcp_socket_window_update_notify_set(
    NX_TCP_SOCKET *socket_ptr,
    VOID (*tcp_window_update_notify) (NX_TCP_SOCKET *socket_ptr));
```

用户数据报协议 (UDP)

```
UINT nx_udp_enable(NX_IP *ip_ptr);
UINT nx_udp_free_port_find(
    NX_IP *ip_ptr, UINT port,
    UINT *free_port_ptr);
UINT nx_udp_info_get(
    NX_IP *ip_ptr,
    ULONG *udp_packets_sent,
    ULONG *udp_bytes_sent,
    ULONG *udp_packets_received,
    ULONG *udp_bytes_received,
    ULONG *udp_invalid_packets
```

```
    ULONG *udp_invalid_packets,
    ULONG *udp_receive_packets_dropped,
    ULONG *udp_checksum_errors);
UINT nx_udp_packet_info_extract(
    NX_PACKET *packet_ptr,
    ULONG *ip_address,
    UINT *protocol, UINT *port,
    UINT *interface_index);
UINT nx_udp_socket_bind(
    NX_UDP_SOCKET *socket_ptr,
UINT port, ULONG wait_option);
UINT nx_udp_socket_bytes_available(
    NX_UDP_SOCKET *socket_ptr,
    ULONG *bytes_available);
UINT nx_udp_socket_checksum_disable(NX_UDP_SOCKET *socket_ptr);
UINT nx_udp_socket_checksum_enable(NX_UDP_SOCKET *socket_ptr);
UINT nx_udp_socket_create(
    NX_IP *ip_ptr,
    NX_UDP_SOCKET *socket_ptr,
    CHAR *name,
    ULONG type_of_service,
    ULONG fragment,
    UINT time_to_live,
    ULONG queue_maximum);
UINT nx_udp_socket_delete(NX_UDP_SOCKET *socket_ptr);
UINT nx_udp_socket_info_get(
    NX_UDP_SOCKET *socket_ptr,
    ULONG *udp_packets_sent,
    ULONG *udp_bytes_sent,
    ULONG *udp_packets_received,
    ULONG *udp_bytes_received,
    ULONG *udp_packets_queued,
    ULONG *udp_receive_packets_dropped,
    ULONG *udp_checksum_errors);
UINT nx_udp_socket_interface_send(
    NX_UDP_SOCKET *socket_ptr,
    NX_PACKET *packet_ptr,
    ULONG ip_address,
    UINT port,
    UINT address_index);
UINT nx_udp_socket_port_get(
    NX_UDP_SOCKET *socket_ptr,
    UINT *port_ptr);
UINT nx_udp_socket_receive(
    NX_UDP_SOCKET *socket_ptr,
    NX_PACKET **packet_ptr,
    ULONG wait_option);
UINT nx_udp_socket_receive_notify(
    NX_UDP_SOCKET *socket_ptr,
    VOID (*udp_receive_notify)(NX_UDP_SOCKET *socket_ptr));
UINT nx_udp_socket_send(
    NX_UDP_SOCKET *socket_ptr,
    NX_PACKET *packet_ptr,
    ULONG ip_address,
    UINT port);
UINT nx_udp_socket_unbind(NX_UDP_SOCKET *socket_ptr);
UINT nx_udp_source_extract(
    NX_PACKET *packet_ptr,
    ULONG *ip_address,
    UINT *port);
```

附录 B - Azure RTOS NetX 常量

2021/4/29 •

字母顺序列表

tttttt	"t"
NX_ALL_HOSTS_ADDRESS	0xFE000001
NX_ALL_ROUTERS_ADDRESS	0xFE000002
NX_ALREADY_BOUND	0x22
NX_ALREADY_ENABLED	0x15
NX_ALREADY_RELEASED	0x31
NX_ALREADY_SUSPENDED	0x40
NX_ANY_PORT	0
NX_ARP_EXPIRATION_RATE	0
NX_ARP_HARDWARE_SIZE	0x06
NX_ARP_HARDWARE_TYPE	0x0001
NX_ARP_MAX_QUEUE_DEPTH	4
NX_ARP_MAXIMUM_RETRIES	18
NX_ARP_MESSAGE_SIZE	28
NX_ARP_OPTION_REQUEST	0x0001
NX_ARP_OPTION_RESPONSE	0x0002
NX_ARP_PROTOCOL_SIZE	0x04
NX_ARP_PROTOCOL_TYPE	0x0800
NX_ARP_TIMER_ERROR	0x18
NX_ARP_UPDATE_RATE	10
NX_ARP_TABLE_SIZE	0x2F
NX_ARP_TABLE_MASK	0x1F

ɛɛɛɛɛ	“ɛ”
NX_CALLER_ERROR	0x11
NX_CARRY_BIT	0x10000
NX_CONNECTION_PENDING	0x48
NX_DELETE_ERROR	0x10
NX_DELETED	0x05
NX_DISCONNECT_FAILED	0x41
NX_DONT_FRAGMENT	0x00004000
NX_DRIVER_TX_DONE	0xDDDDDDDD
NX_DUPLICATE_LISTEN	0x34
NX_ENTRY_NOT_FOUND	0x16
NX_FALSE	0
NX_FOREVER	1
NX_FRAG_OFFSET_MASK	0x00001FFF
NX_FRAGMENT_OKAY	0x00000000
NX_ICMP_ADDRESS_MASK_REP_TYPE	18
NX_ICMP_ADDRESS_MASK_REQ_TYPE	17
NX_ICMP_DEST_UNREACHABLE_TYPE	3
NX_ICMP_ECHO_REPLY_TYPE	0
NX_ICMP_ECHO_REQUEST_TYPE	8
NX_ICMP_FRAMENT_NEEDED_CODE	4
NX_ICMP_HOST_PROHIBIT_CODE	10
NX_ICMP_HOST_SERVICE_CODE	12
NX_ICMP_HOST_UNKNOWN_CODE	7
NX_ICMP_HOST_UNREACH_CODE	1
NX_ICMP_NETWORK_PROHIBIT_CODE	9

ɛɛɛɛɛ	“ɛ”
NX_ICMP_NETWORK_SERVICE_CODE	11
NX_ICMP_NETWORK_UNKNOWN_CODE	6
NX_ICMP_NETWORK_UNREACH_CODE	0
NX_ICMP_PACKET	36
NX_ICMP_PARAMETER_PROB_TYPE	12
NX_ICMP_PORT_UNREACH_CODE	3
NX_ICMP_PROTOCOL_UNREACH_CODE	2
NX_ICMP_REDIRECT_TYPE	5
NX_ICMP_SOURCE_ISOLATED_CODE	8
NX_ICMP_SOURCE_QUENCH_TYPE	4
NX_ICMP_SOURCE_ROUTE_CODE	5
NX_ICMP_TIME_EXCEEDED_TYPE	11
NX_ICMP_TIMESTAMP_REP_TYPE	14
NX_ICMP_TIMESTAMP_REQ_TYPE	13
NX_IGMP_HEADER_SIZE	8
NX_IGMP_HOST_RESPONSE_TYPE	0x02000000
NX_IGMP_HOST_V2_JOIN_TYPE	0x16000000
NX_IGMP_HOST_V2_LEAVE_TYPE	0x17000000
NX_IGMP_HOST_VERSION_1	1
NX_IGMP_HOST_VERSION_2	2
NX_IGMP_MAX_RESP_TIME_MASK	0x00FF0000
NX_IGMP_MAX_UPDATE_TIME	10
NX_IGMP_PACKET	36
NX_IGMP_ROUTER_QUERY_TYPE	0x01000000
NX_IGMP_TTL	1

ɛɛɛɛɛ	“ɛ”
NX_IGMP_TYPE_MASK	0x0F000000
NX_IGMP_VERSION	0x10000000
NX_IGMPV2_TYPE_MASK	0xFF000000
NX_IN_PROGRESS	0x37
NX_INIT_PACKET_ID	1
NX_NOT_IMPLEMENTED	0x4A
NX_NOT_SUPPORTED	0x4B
NX_INVALID_INTERFACE	0x4C
NX_INVALID_PACKET	0x12
NX_INVALID_PORT	0x46
NX_INVALID_RELISTEN	0x47
NX_INVALID_SOCKET	0x13
NX_IP_ADDRESS_ERROR	0x21
NX_IP_ADDRESS_RESOLVED	0x0002
NX_IP_ALIGN_FRAGS	8
NX_IP_ALL_EVENTS	0xFFFFFFFF
NX_IP_ARP_ENABLED	0x0008
NX_IP_ARP_REC_EVENT	0x00000010
NX_IP_CLASS_A_HOSTID	0x00FFFFFF
NX_IP_CLASS_A_MASK	0x80000000
NX_IP_CLASS_A_NETID	0x7F000000
NX_IP_CLASS_A_TYPE	0x00000000
NX_IP_CLASS_B_HOSTID	0x0000FFFF
NX_IP_CLASS_B_MASK	0xC0000000
NX_IP_CLASS_B_NETID	0x3FFF0000

ɛɛɛɛɛ	“ɛ”
NX_IP_CLASS_B_TYPE	0x80000000
NX_IP_CLASS_C_HOSTID	0x000000FF
NX_IP_CLASS_C_MASK	0xE0000000
NX_IP_CLASS_C_NETID	0x1FFFFF00
NX_IP_CLASS_C_TYPE	0xC0000000
NX_IP_CLASS_D_GROUP	0x0FFFFFFF
NX_IP_CLASS_D_HOSTID	0x00000000
NX_IP_CLASS_D_MASK	0xF0000000
NX_IP_CLASS_D_TYPE	0xE0000000
NX_IP_DEBUG_LOG_SIZE	100
NX_IP_DONT_FRAGMENT	0x00004000
NX_IP_DRIVER_DEFERRED_EVENT	0x00000800
NX_IP_DRIVER_PACKET_EVENT	0x00000200
NX_IP_FRAGMENT_MASK	0x00003FFF
NX_IP_ICMP	0x00010000
NX_IP_ICMP_EVENT	0x00000004
NX_IP_ID	0x49502020
NX_IP_IGMP	0x00020000
NX_IP_IGMP_ENABLE_EVENT	0x00000400
NX_IP_IGMP_ENABLED	0x0040
NX_IP_IGMP_EVENT	0x00000040
NX_IP_INITIALIZE_DONE	0x0001
NX_IP_INTERNAL_ERROR	0x20
NX_IP_LENGTH_MASK	0x0F000000
NX_IP_LIMITIED_BROADCAST	0xFFFFFFFF

ɛɛɛɛɛ	“ɛ”
NX_IP_LINK_ENABLED	0x0004
NX_IP_LOOPBACK_FIRST	0x7F000000
NX_IP_LOOPBACK_LAST	0x7FFFFFFF
NX_IP_MAX_DATA	0x00080000
NX_IP_MAX_RELIABLE	0x00040000
NX_IP_MIN_COST	0x00020000
NX_IP_MIN_DELAY	0x00100000
NX_IP_MORE_FRAGMENT	0x00002000
NX_IP_MULTICAST_LOWER	0x5E000000
NX_IP_MULTICAST_MASK	0x007FFFFF
NX_IP_MULTICAST_UPPER	0x00000100
NX_IP_NORMAL	0x00000000
NX_IP_NORMAL_LENGTH	5
NX_IP_OFFSET_MASK	0x00001FFF
NX_IP_PACKET	36
NX_IP_PACKET_SIZE_MASK	0x0000FFFF
NX_IP_PERIODIC_EVENT	0x00000001
NX_IP_PERIODIC_RATE	100
NX_IP_PROTOCOL_MASK	0x00FF0000
NX_IP_RARP_COMPLETE	0x0080
NX_IP_RARP_REC_EVENT	0x00000020
NX_IP_RECEIVE_EVENT	0x00000008
NX_IP_TCP	0x00060000
NX_IP_TCP_CLEANUP_DEFERRED	0x00001000
NX_IP_TCP_ENABLED	0x0020

ɛɛɛɛɛ	“ɛ”
NX_IP_TCP_EVENT	0x00000080
NX_IP_TCP_FAST_EVENT	0x00000100
NX_IP_TIME_TO_LIVE	0x00000080
NX_IP_TIME_TO_LIVE_MASK	0xFF000000
NX_IP_TIME_TO_LIVE_SHIFT	24
NX_IP_TOS_MASK	0x00FF0000
NX_IP_UDP	0x00110000
NX_IP_UDP_ENABLED	0x0010
NX_IP_UNFRAG_EVENT	0x00000002
NX_IP_VERSION	0x450000000x80
NX_LINK_ARP_RESPONSE_SEND	6
NX_LINK_ARP_SEND	5
NX_LINK_DEFERRED_PROCESSING	18
NX_LINK_DISABLE	3
NX_LINK_ENABLE	2
NX_LINK_GET_ALLOC_ERRORS	16
NX_LINK_GET_DUPLEX_TYPE	12
NX_LINK_GET_ERROR_COUNT	13
NX_LINK_GET_RX_COUNT	14
NX_LINK_GET_SPEED	11
NX_LINK_GET_STATUS	10
NX_LINK_GET_TX_COUNT	15
NX_LINK_INITIALIZE	1
NX_LINK_INTERFACE_ATTACH	19
NX_LINK_MULTICAST_JOIN	8

ɛɛɛɛɛ	“ɛ”
NX_LINK_MULTICAST_LEAVE	9
NX_LINK_PACKET_BROADCAST	4
NX_LINK_PACKET_SEND	0
NX_LINK_RARP_SEND	7
NX_LINK_UNINITIALIZE	17
NX_LINK_USER_COMMAND	50
NX_LOWER_16_MASK	0x0000FFFF
NX_MAX_LISTEN	0x33
NX_MAX_LISTEN_REQUESTS	10
NX_MAX_MULTICAST_GROUPS	7
NX_MAX_PORT	0xFFFF
NX_MORE_FRAGMENTS	0x00002000
NX_NO_FREE_PORTS	0x45
NX_NO_MAPPING	0x04
NX_NO_MORE_ENTRIES	0x17
NX_NO_PACKET	0x01
NX_NO_RESPONSE	0x29
NX_NO_WAIT	0
NX_NOT_BOUND	0x24
NX_NOT_CLOSED	0x35
NX_NOT_CONNECTED	0x38
NX_NOT_CREATED	0x27
NX_NOT_ENABLED	0x14
NX_NOT_IMPLEMENTED	0x4A
NX_NOT_LISTEN_STATE	0x36

ɛɛɛɛɛ	“ɛ”
NX_NOT_SUCCESSFUL	0x43
NX_NULL	0
NX_OPTION_ERROR	0x0a
NX_OVERFLOW	0x03
NX_PACKET_ALLOCATED	0xAAAAAAAA
NX_PACKET_DEBUG_LOG_SIZE	100
NX_PACKET_ENQUEUED	0xEEEEEEEE
NX_PACKET_FREE	0xFFFFFFFF
NX_PACKET_POOL_ID	0x5041434B
NX_PACKET_READY	0xBBBBBBBB
NX_PHYSICAL_HEADER	16
NX_PHYSICAL_TRAILER	4
NX_POOL_DELETED	0x30
NX_POOL_ERROR	0x06
NX_PORT_UNAVAILABLE	0x23
NX_PTR_ERROR	0x07
NX_RARP_HARDWARE_SIZE	0x06
NX_RARP_HARDWARE_TYPE	0x0001
NX_RARP_MESSAGE_SIZE	28
NX_RARP_OPTION_REQUEST	0x0003
NX_RARP_OPTION_RESPONSE	0x0004
NX_RARP_PROTOCOL_SIZE	0x04
NX_RARP_PROTOCOL_TYPE	0x0800
NX_RECEIVE_PACKET	0
NX_RESERVED_CODE0	0x19

ɛɛɛɛɛ	“ɛ”
NX_RESERVED_CODE1	0x25
NX_RESERVED_CODE2	0x32
NX_ROUTE_TABLE_MASK	0x1F
NX_ROUTE_TABLE_SIZE	32
NX_SEARCH_PORT_START	49152
NX_SHIFT_BY_16	16
NX_SIZE_ERROR	0x09
NX_SOCKET_UNBOUND	0x26
NX_SOCKETS_BOUND	0x28
NX_STILL_BOUND	0x42
NX_SUCCESS	0x00
NX_TCP_ACK_BIT	0x00100000
NX_TCP_ACK_TIMER_RATE	5
NX_TCP_CLIENT	1
NX_TCP_CLOSE_WAIT	6
NX_TCP_CLOSED	1
NX_TCP_CLOSING	9
NX_TCP_CONTROL_MASK	0x00170000
NX_TCP_EOL_KIND	0x00
NX_TCP_ESTABLISHED	5
NX_TCP_FAST_TIMER_RATE	10
NX_TCP_FIN_BIT	0x00010000
NX_TCP_FIN_WAIT_1	7
NX_TCP_FIN_WAIT_2	8
NX_TCP_HEADER_MASK	0xF000000

ɛɛɛɛɛ	“ɛ”
NX_TCP_HEADER_SHIFT	28
NX_TCP_HEADER_SIZE	0x5000000
NX_TCP_ID	0x5443502
NX_TCP_KEEPALIVE_INITIAL	7200
NX_TCP_KEEPALIVE_RETRIES	10
NX_TCP_KEEPALIVE_RETRY	75
NX_TCP_LAST_ACK	11
NX_TCP_LISTEN_STATE	2
NX_TCP_MAXIMUM_RETRIES	10
NX_TCP_MAXIMUM_TX_QUEUE	20
NX_TCP_MSS_KIND	0x02
NX_TCP_MSS_OPTION	0x02040000
NX_TCP_MSS_SIZE	1460
NX_TCP_NOP_KIND	0x01
NX_TCP_OPTION_END	0x01010100
NX_TCP_PACKET	56
NX_TCP_PORT_TABLE_MASK	0x1F
NX_TCP_PORT_TABLE_SIZE	32
NX_TCP_PSH_BIT	0x00080000
NX_TCP_RETRY_SHIFT	0
NX_TCP_RST_BIT	0x00040000
NX_TCP_SERVER	2
NX_TCP_SYN_BIT	0x00020000
NX_TCP_SYN_HEADER	0x70000000
NX_TCP_SYN_RECEIVED	4

ɛɛɛɛɛ	"ɛ"
NX_TCP_SYN_SENT	3
NX_TCP_TIMED_WAIT	10
NX_TCP_TRANSMIT_TIMER_RATE	1
NX_TCP_URG_BIT	0x00200000
NX_TRUE	1
NX_TX_QUEUE_DEPTH	0x49
NX_UDP_ID	0x55445020
NX_UDP_PACKET	44
NX_UDP_PORT_TABLE_MASK	0x1F
NX_UDP_PORT_TABLE_SIZE	32
NX_UNDERFLOW	0x02
NX_UNHANDLED_COMMAND	0x44
NX_WAIT_ABORTED	0x1A
NX_WAIT_ERROR	0x08
NX_WAIT_FOREVER	0xFFFFFFFF
NX_WINDOW_OVERFLOW	0x39

Listings by Value

ɛɛɛɛɛ	"ɛ"
NX_ANY_PORT	0
NX_ARP_EXPIRATION_RATE	0
NX_FALSE	0
NX_ICMP_ECHO_REPLY_TYPE	0
NX_ICMP_NETWORK_UNREACH_CODE	0
NX_LINK_PACKET_SEND	0
NX_NO_WAIT	0

ɛɛɛɛɛ	“ɛ”
NX_NULL	0
NX_RECEIVE_PACKET	0
NX_TCP_RETRY_SHIFT	0
NX_SUCCESS	0x00
NX_TCP_EOL_KIND	0x00
NX_FRAGMENT_OKAY	0x00000000
NX_IP_CLASS_A_TYPE	0x00000000
NX_IP_CLASS_D_HOSTID	0x00000000
NX_IP_NORMAL	0x00000000
NX_FOREVER	1
NX_ICMP_HOST_UNREACH_CODE	1
NX_IGMP_HOST_VERSION_1	1
NX_IGMP_TTL	1
NX_INIT_PACKET_ID	1
NX_LINK_INITIALIZE	1
NX_TCP_CLIENT	1
NX_TCP_CLOSED	1
NX_TCP_TRANSMIT_TIMER_RATE	1
NX_TRUE	1
NX_IP_PERIODIC_EVENT	0x00000001
NX_ARP_HARDWARE_TYPE	0x0001
NX_ARP_OPTION_REQUEST	0x0001
NX_IP_INITIALIZE_DONE	0x0001
NX_RARP_HARDWARE_TYPE	0x0001
NX_NO_PACKET	0x01

ɛɛɛɛɛ	“ɛ”
NX_TCP_NOP_KIND	0x01
NX_ICMP_PROTOCOL_UNREACH_CODE	2
NX_IGMP_HOST_VERSION_2	2
NX_LINK_ENABLE	2
NX_TCP_LISTEN_STATE	2
NX_TCP_SERVER	2
NX_IP_UNFRAG_EVENT	0x00000002
NX_ARP_OPTION_RESPONSE	0x0002
NX_IP_ADDRESS_RESOLVED	0x0002
NX_TCP_MSS_KIND	0x02
NX_UNDERFLOW	0x02
NX_ICMP_DEST_UNREACHABLE_TYPE	3
NX_ICMP_PORT_UNREACH_CODE	3
NX_LINK_DISABLE	3
NX_TCP_SYN_SENT	3
NX_RARP_OPTION_REQUEST	0x0003
NX_OVERFLOW	0x03
NX_ARP_MAX_QUEUE_DEPTH	4
NX_ICMP_FRAMENT_NEEDED_CODE	4
NX_ICMP_SOURCE_QUENCH_TYPE	4
NX_LINK_PACKET_BROADCAST	4
NX_PHYSICAL_TRAILER	4
NX_TCP_SYN_RECEIVED	4
NX_IP_ICMP_EVENT	0x00000004
NX_IP_LINK_ENABLED	0x0004

ɛɛɛɛɛ	“ɛ”
NX_RARP_OPTION_RESPONSE	0x0004
NX_ARP_PROTOCOL_SIZE	0x04
NX_NO_MAPPING	0x04
NX_RARP_PROTOCOL_SIZE	0x04
NX_ICMP_REDIRECT_TYPE	5
NX_ICMP_SOURCE_ROUTE_CODE	5
NX_IP_NORMAL_LENGTH	5
NX_LINK_ARP_SEND	5
NX_TCP_ACK_TIMER_RATE	5
NX_TCP_ESTABLISHED	5
NX_DELETED	0x05
NX_ICMP_NETWORK_UNKNOWN_CODE	6
NX_LINK_ARP_RESPONSE_SEND	6
NX_TCP_CLOSE_WAIT	6
NX_ARP_HARDWARE_SIZE	0x06
NX_POOL_ERROR	0x06
NX_RARP_HARDWARE_SIZE	0x06
NX_ICMP_HOST_UNKNOWN_CODE	7
NX_LINK_RARP_SEND	7
NX_MAX_MULTICAST_GROUPS	7
NX_TCP_FIN_WAIT_1	7
NX_PTR_ERROR	0x07
NX_ICMP_ECHO_REQUEST_TYPE	8
NX_ICMP_SOURCE_ISOLATED_CODE	8
NX_IP_ALIGN_FRAGS	8

ɛɛɛɛɛ	“ɛ”
NX_LINK_MULTICAST_JOIN	8
NX_TCP_FIN_WAIT_2	8
NX_IGMP_HEADER_SIZE	8
NX_IP_RECEIVE_EVENT	0x00000008
NX_IP_ARP_ENABLED	0x0008
NX_WAIT_ERROR	0x08
NX_ICMP_NETWORK_PROHIBIT_CODE	9
NX_LINK_MULTICAST_LEAVE	9
NX_TCP_CLOSING	9
NX_SIZE_ERROR	0x09
NX_ARP_UPDATE_RATE	10
NX_ICMP_HOST_PROHIBIT_CODE	10
NX_IGMP_MAX_UPDATE_TIME	10
NX_LINK_GET_STATUS	10
NX_MAX_LISTEN_REQUESTS	10
NX_TCP_FAST_TIMER_RATE	10
NX_TCP_KEEPALIVE_RETRIES	10
NX_TCP_MAXIMUM_RETRIES	10
NX_TCP_TIMED_WAIT	10
NX_OPTION_ERROR	0x0A
NX_ICMP_NETWORK_SERVICE_CODE	11
NX_ICMP_TIME_EXCEEDED_TYPE	11
NX_LINK_GET_SPEED	11
NX_TCP_LAST_ACK	11
NX_ICMP_HOST_SERVICE_CODE	12

ɛɛɛɛɛ	“ɛ”
NX_ICMP_PARAMETER_PROB_TYPE	12
NX_LINK_GET_DUPLEX_TYPE	12
NX_ICMP_TIMESTAMP_REQ_TYPE	13
NX_LINK_GET_ERROR_COUNT	13
NX_ICMP_TIMESTAMP_REP_TYPE	14
NX_LINK_GET_RX_COUNT	14
NX_LINK_GET_TX_COUNT	15
NX_LINK_GET_ALLOC_ERRORS	16
NX_PHYSICAL_HEADER	16
NX_SHIFT_BY_16	16
NX_IP_ARP_REC_EVENT	0x00000010
NX_IP_UDP_ENABLED	0x0010
NX_DELETE_ERROR	0x10
NX_ICMP_ADDRESS_MASK_REQ_TYPE	17
NX_LINK_UNINITIALIZE	17
NX_CALLER_ERROR	0x11
NX_ARP_MAXIMUM_RETRIES	18
NX_ICMP_ADDRESS_MASK_REP_TYPE	18
NX_LINK_DEFERRED_PROCESSING	18
NX_INVALID_PACKET	0x12
NX_INVALID_SOCKET	0x13
NX_LINK_INTERFACE_ATTACH	19
NX_TCP_MAXIMUM_TX_QUEUE	20
NX_NOT_ENABLED	0x14
NX_ALREADY_ENABLED	0x15

ɛɛɛɛɛ	“ɛ”
NX_ENTRY_NOT_FOUND	0x16
NX_NO_MORE_ENTRIES	0x17
NX_IP_TIME_TO_LIVE_SHIFT	24
NX_ARP_TIMER_ERROR	0x18
NX_RESERVED_CODE0	0x19
NX_WAIT_ABORTED	0x1A
NX_ARP_MESSAGE_SIZE	28
NX_RARP_MESSAGE_SIZE	28
NX_TCP_HEADER_SHIFT	28
NX_ROUTE_TABLE_MASK	0x1F
NX_TCP_PORT_TABLE_MASK	0x1F
NX_UDP_PORT_TABLE_MASK	0x1F
NX_ROUTE_TABLE_SIZE	32
NX_TCP_PORT_TABLE_SIZE	32
NX_UDP_PORT_TABLE_SIZE	32
NX_IP_RARP_REC_EVENT	0x00000020
NX_IP_TCP_ENABLED	0x0020
NX_IP_INTERNAL_ERROR	0x20
NX_IP_ADDRESS_ERROR	0x21
NX_ALREADY_BOUND	0x22
NX_PORT_UNAVAILABLE	0x23
NX_ICMP_PACKET	36
NX_IGMP_PACKET	36
NX_IP_PACKET	36
NX_IPV4_ICMP_PACKET	36

ττττττ	“τ”
NX_IPV4_IGMP_PACKET	36
NX_NOT_BOUND	0x24
NX_RESERVED_CODE1	0x25
NX_SOCKET_UNBOUND	0x26
NX_NOT_CREATED	0x27
NX_SOCKETS_BOUND	0x28
NX_NO_RESPONSE	0x29
NX_IPV4_UDP_PACKET	44
NX_UDP_PACKET	44
NX_POOL_DELETED	0x30
NX_ALREADY_RELEASED	0x31
NX_LINK_USER_COMMAND	50
NX_RESERVED_CODE2	0x32
NX_MAX_LISTEN	0x33
NX_DUPLICATE_LISTEN	0x34
NX_NOT_CLOSED	0x35
NX_NOT_LISTEN_STATE	0x36
NX_IN_PROGRESS	0x37
NX_NOT_CONNECTED	0x38
NX_IPV4_TCP_PACKET	56
NX_TCP_PACKET	56
NX_WINDOW_OVERFLOW	0x39
NX_IP_IGMP_EVENT	0x00000040
NX_IP_IGMP_ENABLED	0x0040
NX_ALREADY_SUSPENDED	0x40

ɛɛɛɛɛ	“ɛ”
NX_DISCONNECT_FAILED	0x41
NX_STILL_BOUND	0x42
NX_NOT_SUCCESSFUL	0x43
NX_UNHANDLED_COMMAND	0x44
NX_NO_FREE_PORTS	0x45
NX_INVALID_PORT	0x46
NX_INVALID_RELISTEN	0x47
NX_CONNECTION_PENDING	0x48
NX_TX_QUEUE_DEPTH	0x49
NX_NOT_IMPLEMENTED	0x4A
NX_NOT_SUPPORTED	0x4B
NX_TCP_KEEPALIVE_RETRY	75
NX_INVALID_INTERFACE	0x4C
NX_ARP_DEBUG_LOG_SIZE	100
NX_ICMP_DEBUG_LOG_SIZE	100
NX_IGMP_DEBUG_LOG_SIZE	100
NX_IP_DEBUG_LOG_SIZE	100
NX_IP_PERIODIC_RATE	100
NX_PACKET_DEBUG_LOG_SIZE	100
NX_RARP_DEBUG_LOG_SIZE	100
NX_TCP_DEBUG_LOG_SIZE	100
NX_UDP_DEBUG_LOG_SIZE	100
NX_IP_TCP_EVENT	0x00000080
NX_IP_TIME_TO_LIVE	0x00000080
NX_IP_RARP_COMPLETE	0x0080

ɛɛɛɛɛ	“ɛ”
NX_NOT_IMPLEMENTED	0x4A
NX_IP_CLASS_C_HOSTID	0x000000FF
NX_IP_MULTICAST_UPPER	0x00000100
NX_IP_TCP_FAST_EVENT	0x00000100
NX_IP_DRIVER_PACKET_EVENT	0x00000200
NX_IP_IGMP_ENABLE_EVENT	0x00000400
NX_IP_DRIVER_DEFERRED_EVENT	0x00000800
NX_ARP_PROTOCOL_TYPE	0x0800
NX_RARP_PROTOCOL_TYPE	0x0800
NX_IP_TCP_CLEANUP_DEFERRED	0x00001000
NX_TCP_KEEPALIVE_INITIAL	7200
NX_FRAG_OFFSET_MASK	0x00001FFF
NX_IP_OFFSET_MASK	0x00001FFF
NX_IP_MORE_FRAGMENT	0x00002000
NX_MORE_FRAGMENTS	0x00002000
NX_IP_FRAGMENT_MASK	0x00003FFF
NX_TCP_MSS_SIZE	16384
NX_DONT_FRAGMENT	0x00004000
NX_IP_DONT_FRAGMENT	0x00004000
NX_SEARCH_PORT_START	49152
NX_IP_CLASS_B_HOSTID	0x0000FFFF
NX_IP_PACKET_SIZE_MASK	0x0000FFFF
NX_LOWER_16_MASK	0x0000FFFF
NX_MAX_PORT	0xFFFF
NX_IP_ICMP	0x00010000

ɛɛɛɛɛ	“ɛ”
NX_TCP_FIN_BIT	0x00010000
NX_CARRY_BIT	0x10000
NX_IP_IGMP	0x00020000
NX_IP_MIN_COST	0x00020000
NX_TCP_SYN_BIT	0x00020000
NX_IP_MAX_RELIABLE	0x00040000
NX_TCP_RST_BIT	0x00040000
NX_IP_TCP	0x00060000
NX_IP_MAX_DATA	0x00080000
NX_TCP_PSH_BIT	0x00080000
NX_IP_MIN_DELAY	0x00100000
NX_TCP_ACK_BIT	0x00100000
NX_IP_UDP	0x00110000
NX_TCP_CONTROL_MASK	0x00170000
NX_TCP_URG_BIT6	0x00200000
NX_IP_MULTICAST_MASK	0x007FFFFF
NX_IP_PROTOCOL_MASK	0x00FF0000
NX_IP_TOS_MASK	0x00FF0000
NX_IGMP_ROUTER_QUERY_TYPE	0x01000000
NX_TCP_OPTION_END	0x01010402
NX_IGMP_HOST_RESPONSE_TYPE	0x02000000
NX_TCP_MSS_OPTION	0x02040000
NX_IGMP_TYPE_MASK	0x0F000000
NX_IP_LENGTH_MASK	0x0F000000
NX_IGMP_MAX_RESP_TIME_MASK	0x00FF0000

ɛɛɛɛɛ	“ɛ”
NX_IP_CLASS_A_HOSTID	0x00FFFFFF
NX_IP_CLASS_D_GROUP	0x0FFFFFFF
NX_IGMP_VERSION	0x10000000
NX_IGMP_HOST_V2_JOIN_TYPE	0x16000000
NX_IGMP_HOST_V2_LEAVE_TYPE	0x17000000
NX_IP_CLASS_C_NETID	0x1FFFFF00
NX_IP_CLASS_B_NETID	0x3FFF0000
NX_IP_VERSION	0x45000000
NX_IP_ID	0x49502020
NX_TCP_HEADER_SIZE	0x50000000
NX_PACKET_POOL_ID	0x5041434B
NX_TCP_ID	0x54435020
NX_UDP_ID	0x55445020
NX_IP_MULTICAST_LOWER	0x5E000000
NX_IP_CLASS_A_NETID	0x7F000000
NX_TCP_SYN_HEADER	0x70000000
NX_IP_LOOPBACK_FIRST	0x7F000000
NX_IP_LOOPBACK_LAST	0x7FFFFFFF
NX_IP_CLASS_A_MASK	0x80000000
NX_IP_CLASS_B_TYPE	0x80000000
NX_PACKET_ALLOCATED	0xAAAAAAAA
NX_PACKET_READY	0xBBBBBBBB
NX_IP_CLASS_B_MASK	0xC0000000
NX_IP_CLASS_C_TYPE	0xC0000000
NX_DRIVER_TX_DONE	0xDDDDDDDD

ɛɛɛɛɛ	“ɛ”
NX_IP_CLASS_C_MASK	0xE0000000
NX_IP_CLASS_D_TYPE	0xE0000000
NX_PACKET_ENQUEUED	0xEEEEEEEE
NX_IGMP_VERSION_MASK	0xF0000000
NX_IP_CLASS_D_MASK	0xF0000000
NX_TCP_HEADER_MASK	0xF0000000
NX_ALL_HOSTS_ADDRESS	0xFE000001
NX_IGMPV2_TYPE_MASK	0xFF000000
NX_IP_TIME_TO_LIVE_MASK	0xFF000000
NX_IP_ALL_EVENTS	0xFFFFFFFF
NX_IP_LIMITIED_BROADCAST	0xFFFFFFFF
NX_PACKET_FREE	0xFFFFFFFF
NX_WAIT_FOREVER	0xFFFFFFFF

附录 C - Azure RTOS NetX 数据类型

2021/4/29 •

NX_ARP

```
typedef struct NX_ARP_STRUCT
{
    UINT nx_arp_route_static;
    UINT nx_arp_entry_next_update;
    UINT nx_arp_retries;
    struct NX_ARP_STRUCT *nx_arp_pool_next,
    *nx_arp_pool_previous;
    struct NX_ARP_STRUCT *nx_arp_active_next,
    *nx_arp_active_previous,
    **nx_arp_active_list_head;
    ULONG nx_arp_ip_address;
    ULONG nx_arp_physical_address_msw;
    ULONG nx_arp_physical_address_lsw;
    struct NX_INTERFACE_STRUCT *nx_arp_ip_interface;
    struct NX_PACKET_STRUCT *nx_arp_packets_waiting;
}
```

NX_INTERFACE

```
typedef struct NX_INTERFACE_STRUCT
{
    CHAR *nx_interface_name;
    UCHAR nx_interface_valid;
    UCHAR nx_interface_address_mapping_needed;
    UCHAR nx_interface_link_up;
    UCHAR nx_interface_link_status_change;
    struct NX_IP_STRUCT *nx_interface_ip_instance;
    ULONG nx_interface_physical_address_msw;
    ULONG nx_interface_physical_address_lsw;
    ULONG nx_interface_ip_address;
    ULONG nx_interface_ip_network_mask;
    ULONG nx_interface_ip_network;
    ULONG nx_interface_ip_mtu_size;
    VOID *nx_interface_additional_link_info;
    VOID (*nx_interface_link_driver_entry)
        (struct NX_IP_DRIVER_STRUCT *);
    ULONG nx_interface_arp_defend_timeout;
}
```

NX_IP

```
typedef struct NX_IP_STRUCT
{
    ULONG nx_ip_id;
    CHAR *nx_ip_name;
    #define nx_ip_address nx_ip_interface[0].nx_interface_ip_address
    #define nx_ip_driver_mtu nx_ip_interface[0].nx_interface_ip_mtu_size
    #define nx_ip_driver_mapping_needed nx_ip_interface[0].nx_interface_address_mapping_needed
    #define nx_ip_network_mask nx_ip_interface[0].nx_interface_ip_network_mask
    #define nx_ip_network nx_ip_interface[0].nx_interface_ip_network
    #define nx_ip_arp_physical_address_msw nx_ip_interface[0].nx_interface_physical_address_msw
```

```
#define nx_ip_arp_physical_address_lsw nx_ip_interface[0].nx_interface_physical_address_lsw
#define nx_ip_driver_link_up nx_ip_interface[0].nx_interface_link_up
#define nx_ip_link_driver_entry nx_ip_interface[0].nx_interface_link_driver_entry
#define nx_ip_additional_link_info nx_ip_interface[0].nx_interface_additional_link_info

ULONG nx_ip_gateway_address;
struct NX_INTERFACE_STRUCT *nx_ip_gateway_interface;
ULONG nx_ip_total_packet_send_requests;
ULONG nx_ip_total_packets_sent;
ULONG nx_ip_total_bytes_sent;
ULONG nx_ip_total_packets_received;
ULONG nx_ip_total_packets_delivered;
ULONG nx_ip_total_bytes_received;
ULONG nx_ip_packets_forwarded;
ULONG nx_ip_packets_reassembled;
ULONG nx_ip_reassembly_failures;
ULONG nx_ip_invalid_packets;
ULONG nx_ip_invalid_transmit_packets;
ULONG nx_ip_invalid_receive_address;
ULONG nx_ip_unknown_protocols_received;
ULONG nx_ip_transmit_resource_errors;
ULONG nx_ip_transmit_no_route_errors;
ULONG nx_ip_receive_packets_dropped;
ULONG nx_ip_receive_checksum_errors;
ULONG nx_ip_send_packets_dropped;
ULONG nx_ip_total_fragment_requests;
ULONG nx_ip_successful_fragment_requests;
ULONG nx_ip_fragment_failures;
ULONG nx_ip_total_fragments_sent;
ULONG nx_ip_total_fragments_received;
ULONG nx_ip_arp_requests_sent;
ULONG nx_ip_arp_requests_received;
ULONG nx_ip_arp_responses_sent;
ULONG nx_ip_arp_responses_received;
ULONG nx_ip_arp_aged_entries;
ULONG nx_ip_arp_invalid_messages;
ULONG nx_ip_arp_static_entries;
ULONG nx_ip_udp_packets_sent;
ULONG nx_ip_udp_bytes_sent;
ULONG nx_ip_udp_packets_received;
ULONG nx_ip_udp_bytes_received;
ULONG nx_ip_udp_invalid_packets;
ULONG nx_ip_udp_no_port_for_delivery;
ULONG nx_ip_udp_receive_packets_dropped;
ULONG nx_ip_udp_checksum_errors;
ULONG nx_ip_tcp_packets_sent;
ULONG nx_ip_tcp_bytes_sent;
ULONG nx_ip_tcp_packets_received;
ULONG nx_ip_tcp_bytes_received;
ULONG nx_ip_tcp_invalid_packets;
ULONG nx_ip_tcp_receive_packets_dropped;
ULONG nx_ip_tcp_checksum_errors;
ULONG nx_ip_tcp_connections;
ULONG nx_ip_tcp_passive_connections;
ULONG nx_ip_tcp_active_connections;
ULONG nx_ip_tcp_disconnections;
ULONG nx_ip_tcp_connections_dropped;
ULONG nx_ip_tcp_retransmit_packets;
ULONG nx_ip_tcp_resets_received;
ULONG nx_ip_tcp_resets_sent;
ULONG nx_ip_icmp_total_messages_received;
ULONG nx_ip_icmp_checksum_errors;
ULONG nx_ip_icmp_invalid_packets;
ULONG nx_ip_icmp_unhandled_messages;
ULONG nx_ip_pings_sent;
ULONG nx_ip_ping_timeouts;
ULONG nx_ip_ping_threads_suspended;
ULONG nx_ip_ping_responses_received;
ULONG nx_ip_pings_received;
```

```
    ULONG nx_ip_pings_responded_to;
    ULONG nx_ip_igmp_invalid_packets;
    ULONG nx_ip_igmp_reports_sent;
    ULONG nx_ip_igmp_queries_received;
    ULONG nx_ip_igmp_checksum_errors;
    ULONG nx_ip_igmp_groups_joined;

    #ifndef NX_DISABLE_IGMPV2
        ULONG nx_ip_igmp_router_version;
    #endif

    ULONG nx_ip_rarp_requests_sent;
    ULONG nx_ip_rarp_responses_received;
    ULONG nx_ip_rarp_invalid_messages;
    VOID (*nx_ip_forward_packet_process) (struct NX_IP_STRUCT *, NX_PACKET *);
    ULONG nx_ip_packet_id;
    struct NX_PACKET_POOL_STRUCT *nx_ip_default_packet_pool;
    TX_MUTEX nx_ip_protection;
    UINT nx_ip_initialize_done;
    NX_PACKET *nx_ip_driver_deferred_packet_head, *nx_ip_driver_deferred_packet_tail;
    VOID (*nx_ip_driver_deferred_packet_handler)(struct NX_IP_STRUCT *, NX_PACKET *);
    NX_PACKET *nx_ip_deferred_received_packet_head, *nx_ip_deferred_received_packet_tail;
    UINT (*nx_ip_raw_ip_processing)(struct NX_IP_STRUCT *, ULONG, NX_PACKET *);

    #ifdef NX_ENABLE_IP_RAW_PACKET_FILTER
    UINT (*nx_ip_raw_packet_filter)(struct NX_IP_STRUCT *, ULONG, NX_PACKET *);
    #endif /* NX_ENABLE_IP_RAW_PACKET_FILTER */

    NX_PACKET *nx_ip_raw_received_packet_head, *nx_ip_raw_received_packet_tail;
    ULONG nx_ip_raw_received_packet_count;
    ULONG nx_ip_raw_received_packet_max;
    TX_THREAD *nx_ip_raw_packet_suspension_list;
    ULONG nx_ip_raw_packet_suspended_count;
    TX_THREAD nx_ip_thread;
    TX_EVENT_FLAGS_GROUP nx_ip_events;
    TX_TIMER nx_ip_periodic_timer;
    VOID (*nx_ip_fragment_processing)(struct NX_IP_DRIVER_STRUCT *);
    VOID (*nx_ip_fragment_assembly)(struct NX_IP_STRUCT *);
    VOID (*nx_ip_fragment_timeout_check) (struct NX_IP_STRUCT *);
    NX_PACKET *nx_ip_timeout_fragment;
    NX_PACKET *nx_ip_received_fragment_head, *nx_ip_received_fragment_tail;
    NX_PACKET *nx_ip_fragment_assembly_head, *nx_ip_fragment_assembly_tail;
    VOID (*nx_ip_address_change_notify)(struct NX_IP_STRUCT *, VOID *);
    VOID *nx_ip_address_change_notify_additional_info;
    ULONG nx_ip_igmp_join_list[NX_MAX_MULTICAST_GROUPS];
    NX_INTERFACE *nx_ip_igmp_interfacejoin_list[NX_MAX_MULTICAST_GROUPS];
    ULONG nx_ip_igmp_join_count[NX_MAX_MULTICAST_GROUPS];
    ULONG nx_ip_igmp_update_time[NX_MAX_MULTICAST_GROUPS];
    UINT nx_ip_igmp_global_loopback_enable;
    ULONG nx_ip_igmp_group_loopback_enable[NX_MAX_MULTICAST_GROUPS]
        void (*nx_ip_igmp_packet_receive)(struct NX_IP_STRUCT *,
        struct NX_PACKET_STRUCT *);
    void (*nx_ip_igmp_periodic_processing) (struct NX_IP_STRUCT *);
    void (*nx_ip_igmp_queue_process)(struct NX_IP_STRUCT *);
    NX_PACKET *nx_ip_igmp_queue_head;
    ULONG nx_ip_icmp_sequence;
    void (*nx_ip_icmp_packet_receive)(struct NX_IP_STRUCT *,
        struct NX_PACKET_STRUCT *);
    void (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *);
    NX_PACKET *nx_ip_icmp_queue_head;
    TX_THREAD *nx_ip_icmp_ping_suspension_list;
    ULONG nx_ip_icmp_ping_suspended_count;
    struct NX_UDP_SOCKET_STRUCT *nx_ip_udp_port_table[NX_UDP_PORT_TABLE_SIZE];
    struct NX_UDP_SOCKET_STRUCT *nx_ip_udp_created_sockets_ptr;
    ULONG nx_ip_udp_created_sockets_count;
    void (*nx_ip_udp_packet_receive)(struct NX_IP_STRUCT *,
        struct NX_PACKET_STRUCT *);
    UINT nx_ip_udp_port_search_start;
    struct NX_TCP_SOCKET_STRUCT *nx_ip_tcp_port_table[NX_TCP_PORT_TABLE_SIZE];
```

```
    struct NX_TCP_SOCKET_STRUCT *nx_ip_tcp_created_sockets_ptr;
    ULONG nx_ip_tcp_created_sockets_count;
    void (*nx_ip_tcp_packet_receive)(struct NX_IP_STRUCT *,
        struct NX_PACKET_STRUCT *);
    void (*nx_ip_tcp_periodic_processing)
        (struct NX_IP_STRUCT *);
    void (*nx_ip_tcp_fast_periodic_processing)(struct NX_IP_STRUCT *);
    void (*nx_ip_tcp_queue_process)(struct NX_IP_STRUCT *);
    NX_PACKET *nx_ip_tcp_queue_head, *nx_ip_tcp_queue_tail;
    ULONG nx_ip_tcp_received_packet_count;
    struct NX_TCP_LISTEN_STRUCT nx_ip_tcp_server_listen_reqs[NX_MAX_LISTEN_REQUESTS];
    struct NX_TCP_LISTEN_STRUCT *nx_ip_tcp_available_listen_requests;
    struct NX_TCP_LISTEN_STRUCT *nx_ip_tcp_active_listen_requests;
    UINT nx_ip_tcp_port_search_start;
    UINT nx_ip_fast_periodic_timer_created;
    TX_TIMER nx_ip_fast_periodic_timer;
    struct NX_ARP_STRUCT *nx_ip_arp_table[NX_ARP_TABLE_SIZE];
    struct NX_ARP_STRUCT *nx_ip_arp_static_list;
    struct NX_ARP_STRUCT *nx_ip_arp_dynamic_list;
    ULONG nx_ip_arp_dynamic_active_count;
    NX_PACKET *nx_ip_arp_deferred_received_packet_head,
        *nx_ip_arp_deferred_received_packet_tail;
    UINT (*nx_ip_arp_allocate)(struct NX_IP_STRUCT *, struct NX_ARP_STRUCT **, UINT);
    void (*nx_ip_arp_periodic_update)(struct NX_IP_STRUCT *);
    void (*nx_ip_arp_queue_process)(struct NX_IP_STRUCT *);
    void (*nx_ip_arp_packet_send)(struct NX_IP_STRUCT *, ULONG destination_ip,
        NX_INTERFACE     *nx_interface);
    void (*nx_ip_arp_gratuitous_response_handler)(struct NX_IP_STRUCT *, NX_PACKET *);
    void (*nx_ip_arp_collision_notify_response_handler) (void *);
    void *nx_ip_arp_collision_notify_parameter;
    ULONG nx_ip_arp_collision_notify_ip_address;
    struct NX_ARP_STRUCT *nx_ip_arp_cache_memory;
    ULONG nx_ip_arp_total_entries;
    void (*nx_ip_rarp_periodic_update)(struct NX_IP_STRUCT *);
    void (*nx_ip_rarp_queue_process)(struct NX_IP_STRUCT *);
    NX_PACKET *nx_ip_rarp_deferred_received_packet_head,
        *nx_ip_rarp_deferred_received_packet_tail;
    struct NX_IP_STRUCT *nx_ip_created_next, *nx_ip_created_previous;
    void *nx_ip_reserved_ptr;
    void (*nx_tcp_deferred_cleanup_check) (struct NX_IP_STRUCT *);
    NX_INTERFACE nx_ip_interface[NX_MAX_IP_INTERFACES];

    #ifdef NX_ENABLE_IP_STATIC_ROUTING
        NX_IP_ROUTING_ENTRY nx_ip_routing_table[NX_IP_ROUTING_TABLE_SIZE];
        ULONG nx_ip_routing_table_entry_count;
    #endif /* NX_ENABLE_IP_STATIC_ROUTING */

    VOID (*nx_ip_link_status_change_callback)(struct NX_IP_STRUCT *, UINT, UINT);

    #ifdef NX_ENABLE_IP_PACKET_FILTER
        UINT (*nx_ip_packet_filter)(VOID *, UINT);
    #endif /* NX_ENABLE_IP_PACKET_FILTER */
}
```

NX_IP_DRIVER

```
typedef struct NX_IP_DRIVER_STRUCT
{
    UINT nx_ip_driver_command;
    UINT nx_ip_driver_status;
    ULONG nx_ip_driver_physical_address_msw;
    ULONG nx_ip_driver_physical_address_lsw;
    NX_PACKET *nx_ip_driver_packet;
    ULONG *nx_ip_driver_return_ptr;
    struct NX_IP_STRUCT *nx_ip_driver_ptr;
    NX_INTERFACE *nx_ip_driver_interface;
}
```

NX_IP_ROUTING_ENTRY

```
typedef struct NX_IP_ROUTING_ENTRY_STRUCT
{
    ULONG nx_ip_routing_dest_ip;
    ULONG nx_ip_routing_net_mask;
    412 NetX User Guide
    User Guide
    ULONG nx_ip_routing_next_hop_address;
    NX_INTERFACE *nx_ip_routing_entry_ip_interface;
}
```

NX_PACKET

```
typedef struct NX_PACKET_STRUCT
{
    struct NX_PACKET_POOL_STRUCT *nx_packet_pool_owner;
    struct NX_PACKET_STRUCT *nx_packet_queue_next;
    struct NX_PACKET_STRUCT *nx_packet_tcp_queue_next;
    struct NX_PACKET_STRUCT *nx_packet_next;
    struct NX_PACKET_STRUCT *nx_packet_last;
    struct NX_PACKET_STRUCT *nx_packet_fragment_next;
    ULONG nx_packet_length;
    struct NX_INTERFACE_STRUCT *nx_packet_ip_interface;
    ULONG nx_packet_next_hop_address;
    UCHAR *nx_packet_data_start;
    UCHAR *nx_packet_data_end;
    UCHAR *nx_packet_prepend_ptr;
    UCHAR *nx_packet_append_ptr;

    #ifdef NX_PACKET_HEADER_PAD
        ULONG nx_packet_pad[NX_PACKET_HEADER_PAD_SIZE];
    #endif
}
```

NX_PACKET_POOL

```
typedef struct NX_PACKET_POOL_STRUCT
{
    ULONG nx_packet_pool_id;
    CHAR *nx_packet_pool_name;
    ULONG nx_packet_pool_available;
    ULONG nx_packet_pool_total;
    ULONG nx_packet_pool_empty_requests;
    ULONG nx_packet_pool_empty_suspensions;
    ULONG nx_packet_pool_invalid_releases;
    struct NX_PACKET_STRUCT *nx_packet_pool_available_list;
    CHAR *nx_packet_pool_start;
    ULONG nx_packet_pool_size;
    ULONG nx_packet_pool_payload_size;
    TX_THREAD *nx_packet_pool_suspension_list;
    ULONG nx_packet_pool_suspended_count;
    struct NX_PACKET_POOL_STRUCT *nx_packet_pool_created_next,
    *nx_packet_pool_created_previous;
}
```

NX_TCP_LISTEN

```
typedef struct NX_TCP_LISTEN_STRUCT
{
    UINT nx_tcp_listen_port;
    VOID (*nx_tcp_listen_callback)(NX_TCP_SOCKET *socket_ptr,
    UINT port);
    NX_TCP_SOCKET *nx_tcp_listen_socket_ptr;
    ULONG nx_tcp_listen_queue_maximum;
    ULONG nx_tcp_listen_queue_current;
    NX_PACKET *nx_tcp_listen_queue_head, *nx_tcp_listen_queue_tail;
    struct NX_TCP_LISTEN_STRUCT *nx_tcp_listen_next, *nx_tcp_listen_previous;
}
```

NX_TCP_SOCKET;

```
typedef struct NX_TCP_SOCKET_STRUCT
{
    ULONG nx_tcp_socket_id;
    CHAR *nx_tcp_socket_name;
    UINT nx_tcp_socket_client_type;
    UINT nx_tcp_socket_port;
    ULONG nx_tcp_socket_mss;
    ULONG nx_tcp_socket_connect_ip;
    UINT nx_tcp_socket_connect_port;
    ULONG nx_tcp_socket_connect_mss;
    NetX Data Types 413
    struct NX_INTERFACE_STRUCT *nx_tcp_socket_connect_interface;
    ULONG nx_tcp_socket_next_hop_address;
    ULONG nx_tcp_socket_connect_mss2;
    ULONG nx_tcp_socket_tx_slow_start_threshold;
    UINT nx_tcp_socket_state;
    ULONG nx_tcp_socket_tx_sequence;
    ULONG nx_tcp_socket_rx_sequence;
    ULONG nx_tcp_socket_rx_sequence_acked;
    ULONG nx_tcp_socket_delayed_ack_timeout;
    ULONG nx_tcp_socket_fin_sequence;
    USHORT nx_tcp_socket_fin_received;
    ULONG nx_tcp_socket_tx_window_advertised;
    ULONG nx_tcp_socket_tx_window_congestion;
    ULONG nx_tcp_socket_tx_outstanding_bytes;
    ULONG nx_tcp_socket_tx_sequence_recover;
    ULONG nx_tcp_socket_previous_highest_ack;
```

```
    UCHAR nx_tcp_socket_fast_recovery;
    UCHAR nx_tcp_socket_reserved[3];
    ULONG nx_tcp_socket_ack_n_packet_counter;
    UINT nx_tcp_socket_duplicated_ack_received;
    ULONG nx_tcp_socket_rx_window_default;
    ULONG nx_tcp_socket_rx_window_current;
    ULONG nx_tcp_socket_rx_window_last_sent;
    ULONG nx_tcp_socket_packets_sent;
    ULONG nx_tcp_socket_bytes_sent;
    ULONG nx_tcp_socket_packets_received;
    ULONG nx_tcp_socket_bytes_received;
    ULONG nx_tcp_socket_retransmit_packets;
    ULONG nx_tcp_socket_checksum_errors;
    struct NX_IP_STRUCT *nx_tcp_socket_ip_ptr;
    ULONG nx_tcp_socket_type_of_service;
    UINT nx_tcp_socket_time_to_live;
    ULONG nx_tcp_socket_fragment_enable;
    ULONG nx_tcp_socket_receive_queue_count;
    NX_PACKET *nx_tcp_socket_receive_queue_head,
        *nx_tcp_socket_receive_queue_tail;
    ULONG nx_tcp_socket_transmit_queue_maximum;
    ULONG nx_tcp_socket_transmit_sent_count;
    NX_PACKET *nx_tcp_socket_transmit_sent_head,
        nx_tcp_socket_transmit_sent_tail;
    ULONG nx_tcp_socket_timeout;
    ULONG nx_tcp_socket_timeout_rate;
    ULONG nx_tcp_socket_timeout_retries;
    ULONG nx_tcp_socket_timeout_max_retries;
    ULONG nx_tcp_socket_timeout_shift;

    #ifdef NX_ENABLE_TCP_WINDOW_SCALING
        ULONG nx_tcp_socket_rx_window_maximum;
        ULONG nx_tcp_rcv_win_scale_value;
        ULONG nx_tcp_snd_win_scale_value;
    #endif /* NX_ENABLE_TCP_WINDOW_SCALING */

    ULONG nx_tcp_socket_keepalive_timeout;
    ULONG nx_tcp_socket_keepalive_retries;
    struct NX_TCP_SOCKET_STRUCT *nx_tcp_socket_bound_next,
        *nx_tcp_socket_bound_previous;
    TX_THREAD *nx_tcp_socket_bind_in_progress;
    TX_THREAD *nx_tcp_socket_receive_suspension_list;
    ULONG nx_tcp_socket_receive_suspended_count;
    TX_THREAD *nx_tcp_socket_transmit_suspension_list;
    ULONG nx_tcp_socket_transmit_suspended_count;
    TX_THREAD *nx_tcp_socket_connect_suspended_thread;
    TX_THREAD *nx_tcp_socket_disconnect_suspended_thread;
    TX_THREAD *nx_tcp_socket_bind_suspension_list;
    ULONG nx_tcp_socket_bind_suspended_count;
    struct NX_TCP_SOCKET_STRUCT *nx_tcp_socket_created_next,
        *nx_tcp_socket_created_previous;
    VOID (*nx_tcp_urgent_data_callback)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);

    #ifndef NX_DISABLE_EXTENDED_NOTIFY_SUPPORT
        UINT (*nx_tcp_socket_syn_received_notify)(struct NX_TCP_SOCKET_STRUCT *socket_ptr,
            NX_PACKET *packet_ptr);
        VOID (*nx_tcp_establish_notify)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
        VOID (*nx_tcp_disconnect_complete_notify)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
        VOID (*nx_tcp_timed_wait_callback)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    #endif
    VOID (*nx_tcp_disconnect_callback)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    VOID (*nx_tcp_receive_callback)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    VOID (*nx_tcp_socket_window_update_notify)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    void *nx_tcp_socket_reserved_ptr;
    ULONG nx_tcp_socket_transmit_queue_maximum_default;
    UINT nx_tcp_socket_keepalive_enabled;
}
```

NX_UDP_SOCKET

```
typedef struct NX_UDP_SOCKET_STRUCT
{
    ULONG nx_udp_socket_id;
    CHAR *nx_udp_socket_name;
    UINT nx_udp_socket_port;
    struct NX_IP_STRUCT *nx_udp_socket_ip_ptr;
    ULONG nx_udp_socket_packets_sent;
    ULONG nx_udp_socket_bytes_sent;
    ULONG nx_udp_socket_packets_received;
    ULONG nx_udp_socket_bytes_received;
    ULONG nx_udp_socket_invalid_packets;
    ULONG nx_udp_socket_packets_dropped;
    ULONG nx_udp_socket_checksum_errors;
    ULONG nx_udp_socket_type_of_service;
    UINT nx_udp_socket_time_to_live;
    ULONG nx_udp_socket_fragment_enable;
    UINT nx_udp_socket_disable_checksum;
    ULONG nx_udp_socket_receive_count;
    ULONG nx_udp_socket_queue_maximum;
    NX_PACKET *nx_udp_socket_receive_head,
        *nx_udp_socket_receive_tail;
    struct NX_UDP_SOCKET_STRUCT *nx_udp_socket_bound_next,
        *nx_udp_socket_bound_previous;
    TX_THREAD *nx_udp_socket_bind_in_progress;
    TX_THREAD *nx_udp_socket_receive_suspension_list;
    ULONG nx_udp_socket_receive_suspended_count;
    TX_THREAD *nx_udp_socket_bind_suspension_list;
    ULONG nx_udp_socket_bind_suspended_count;
    struct NX_UDP_SOCKET_STRUCT *nx_udp_socket_created_next,
        *nx_udp_socket_created_previous;
    VOID (*nx_udp_receive_callback)(struct NX_UDP_SOCKET_STRUCT
    *socket_ptr);
    void *nx_udp_socket_reserved_ptr;
    struct NX_INTERFACE_STRUCT *nx_udp_socket_ip_interface;
}
```

附录 D - 与 Azure RTOS NetX BSD 兼容的套接字 API

2021/4/29 •

与 BSD 兼容的套接字 API

与 BSD 兼容的套接字 API 通过利用下面的 Azure RTOS NetX 基元来支持 BSD 套接字 API 调用的一部分(存在一些限制)。支持 IPv4 协议和网络寻址。由于此 API 利用内部 NetX 基元并绕过不必要的 NetX 错误检查，这一个与 BSD 兼容的套接字 API 层的执行速度应比典型的 BSD 实现快或稍快。

可配置选项允许主机应用程序定义最大套件数、TCP 最大窗口大小和侦听队列的深度。

由于性能和体系结构约束，这一个与 BSD 兼容的套接字 API 不支持所有 BSD 套接字调用。此外，并非所有 BSD 选项都可用于 BSD 服务，具体如下：

- *select_ 函数只能与 _fd_set *readfds* 一起使用，不支持此调用中的其他参数(例如 writefds、exceptfds)。
- *send_、recv、sendto 和 *recvfrom * 函数不支持 int flags 参数。*与 BSD 兼容的套接字 API 仅支持有限的一组 BSD 套接字调用。

源代码设计简单，仅由两个文件组成，即 *nx_bsd.c__ 和 nx_bsd.h。安装需要将这两个文件添加到生成项目(不是 NetX 库)并创建将使用 BSD 套接字服务调用的主机应用程序。_nx_bsd.h* 文件也必须包含在应用程序源中。示例演示文件随分发一起提供，随时可用于 NetX。如需了解详细信息，请参阅与 BSD 兼容的套接字 API 417 帮助以及与 BSD 兼容的套接字 API 包随附的自述文件。

与 BSD 兼容的套接字 API 支持以下 BSD 套接字 API 调用：

```
*INT bsd_initialize (NX_IP *default_ip, NX_PACKET_POOL *default_pool,
    CHAR *bsd_memory_not_used);*

*INT getpeername( INT sockID, struct sockaddr *remoteAddress, INT *addressLength);*

*INT getsockname( INT sockID, struct sockaddr *localAddress, INT *addressLength);*

*INT recvfrom(INT sockID, CHAR *buffer, INT buffersize,
    INT flags,struct sockaddr *fromAddr, INT *fromAddrLen);*

*INT recv(INT sockID, VOID *rcvBuffer, INT bufferLength, INT flags);*

*INT sendto(INT sockID, CHAR *msg, INT msgLength, INT flags,
    struct sockaddr *destAddr, INT destAddrLen);*

*INT send(INT sockID, const CHAR *msg, INT msgLength, INT flags);*

 *INT accept(INT sockID, struct sockaddr *ClientAddress, INT *addressLength);*

*INT listen(INT sockID, INT backlog);*

*INT bind (INT sockID, struct sockaddr *localAddress, INT addressLength);*

*INT connect(INT sockID, struct sockaddr *remoteAddress, INT addressLength);*

*INT socket( INT protocolFamily, INT type, INT protocol);*

*INT soc_close ( INT sockID);*

*INT select(INT nfds, fd_set *readfds, fd_set *writefds,
    fd_set *exceptfds, struct timeval *timeout);*

*VOID FD_SET(INT fd, fd_set *fdset);*

*VOID FD_CLR(INT fd, fd_set *fdset);*

*INT FD_ISSET(INT fd, fd_set *fdset);*

*VOID FD_ZERO(fd_set *fdset);*
```

附录 E - Azure RTOS NetX ASCII 字符代码

2021/4/29 •

十六进制 ASCII 字符代码

least significant nibble \ most significant nibble	0_	1_	2_	3_	4_	5_	6_	7_
_0	NUL	DLE	SP	0	@	P	'	p
_1	SOH	DC1	!	1	A	Q	a	q
_2	STX	DC2	"	2	B	R	b	r
_3	ETX	DC3	#	3	C	S	c	s
_4	EOT	DC4	$	4	D	T	d	t
_5	ENQ	NAK	%	5	E	U	e	u
_6	ACK	SYN	&	6	F	V	f	v
_7	BEL	ETB	'	7	G	W	g	w
_8	BS	CAN	(	8	H	X	h	x
_9	HT	EM	)	9	I	Y	i	y
_A	LF	SUB	*	:	J	Z	j	z
_B	VT	ESC	+	;	K	[	K	}
_C	FF	FS	,	<	L	\	l	|
_D	CR	GS	-	=	M	]	m	}
_E	SO	RS	.	>	N	^	n	~
_F	SI	US	/	?	O	_	o	DEL

第 1 章 - Azure RTOS NetX AutoIP 简介

2021/4/29 •

Azure RTOS NetX AutoIP 协议是一种专用于在本地网络上动态配置 IPv4 地址的协议。AutoIP 是一种简单的协议，它利用 ARP 功能来执行其自动 IP 地址分配功能。AutoIP 在 169.254.1.0 到 169.254.254.255 的范围内分配地址。

AutoIP 的要求

为正常工作，NetX AutoIP 包要求已创建 NetX IP 实例。另外，必须在同一 IP 实例上启用 ARP。除此之外，NetX AutoIP 包没有其他要求。

AutoIP 约束

NetX AutoIP 协议实现 RFC3927 标准的要求，但是存在以下约束：

1. 如果使用了 NetX DHCP，则必须以高于 NetX IP 实例线程和 AutoIP 线程的优先级创建 DHCP 线程。
2. NetX AutoIP 未提供继续使用旧 IP 地址的机制。
3. 当 IP 地址更改时，应用程序负责撤销任何现有 TCP 连接，并在新的 IP 地址上重新建立这些连接。

AutoIP 协议实现

NetX AutoIP 协议先在 169.254.1.0 到 169.254.254.255 的 AutoIP IPv4 地址范围内选择随机地址。或者，应用程序可以通过向 nx_auto_ip_start 函数提供起始 IP 地址来强制使用该地址。这在以前的运行已成功使用 AutoIP 地址的情况下十分有用。

选择 AutoIP 地址后，NetX AutoIP 会针对选定的地址发送一系列 ARP 探测。ARP 探测中包含一条 ARP 请求消息，其发送方地址设置为 0.0.0.0，目标地址设置为期望的 AutoIP 地址。系统会发送一系列的 ARP 探测，实际数目由定义 NX_AUTO_IP_PROBE_NUM 确定。如果其他网络节点响应此探测或针对同一地址发送相同的探测，则会在 AutoIP IPv4 地址范围内随机选择新的 AutoIP 地址，并重复探测处理。

如果在无任何响应的情况下发送 NX_AUTO_IP_PROBE_NUM 探测，则 NetX AutoIP 会针对选定的地址发出一系列 ARP 公告。ARP 公告是由一则 ARP 请求消息组成，该 ARP 消息中的发送方地址和目标地址都设置为选定的 AutoIP 地址。系统会发送一系列 ARP 公告消息，其数目对应于定义 NX_AUTO_IP_ANNOUNCE_NUM。如果其他网络节点响应某公告消息或针对同一地址发送相同的公告，则会在 AutoIP IPv4 地址范围内随机选择新的 AutoIP 地址，并且探测处理重新开始。

当探测和公告在未检测到任何冲突的情况下完成时，就会认为选定的 AutoIP 地址有效，并将关联的 IP 实例设置为采用该地址。

AutoIP 地址更改

如前所述，NetX AutoIP 会在探测和公告处理成功后更改 IP 实例地址。监视这种情况并不十分重要。但是，将来可能会更改 AutoIP 地址。可能的原因包括未来的 AutoIP 地址冲突，以及 DHCP 地址解析。为正确处理这些潜在的情况，应用程序应使用以下 NetX API 接收任何及所有 IP 地址更改的相应警报：

```
nx_ip_address_change_notify(NX_IP *ip_ptr,
                            VOID (*ip_address_change_notify)(NX_IP *,VOID*),
                            VOID *additional_info);
```

所提供的 ip_address_change_notify 函数中的处理必须重新启动 NetX AutoIP 处理器，或者在 DHCP 随后解析该

IP 地址后禁用该处理器。有关示例处理，请参阅“小型示例系统”一节。

AutoIP RFC

NetX AutoIP 符合 RFC3927 和相关 RFC。

第 2 章 - 安装和使用 Azure RTOS NetX AutoIP

2021/4/29 •

本章介绍与安装、设置和使用 Azure RTOS NetX AutoIP 组件相关的各种问题。

产品分发

可通过 https://github.com/azure-rtos/netx 获取用于 NetX 的 AutoIP。软件包中有三个源文件、一个包含文件和一个包含本文档的 PDF 文件，如下所示：

- **nx_auto_ip.h**：NetX AutoIP 的头文件
- **nx_auto_ip.c**：NetX AutoIP 的 C 源文件
- **demo_netx_auto_ip.c**：NetX AutoIP 演示的 C 源文件
- **nx_auto_ip.pdf**：NetX AutoIP 的 PDF 说明

AutoIP 安装

若要使用 NetX AutoIP，应将之前提到的整个分发包复制到 NetX 的安装目录中。例如，如果 NetX 安装在“\threadx\arm7\green”目录中，则应将 nx_auto_ip.h、nx_auto_ip.c 和 demo_netx_auto_ip.c 文件复制到该目录中。

使用 AutoIP

使用 NetX AutoIP 很简单。大致来说，应用程序代码中必须先添加 tx_api.h 和 nx_api.h，然后添加 nx_auto_ip.h，才能使用 ThreadX 和 NetX。在添加 nx_auto_ip.h 之后，应用程序代码即可发出本指南后文所指定的 AutoIP 函数调用。应用程序还必须在生成过程中添加 nx_auto_ip.c。这些文件的编译方式必须与其他应用程序文件相同，并且其对象窗体必须与应用程序的文件链接起来。这就是使用 NetX AutoIP 所需的全部内容。

> **NOTE**
> 由于 AutoIP 使用 NetX ARP 服务，因此在使用 AutoIP 之前，必须通过 nx_arp_enable 调用来启用 ARP。

小型示例系统

下面的图 1.1 举例说明了 NetX AutoIP 是多么易于使用。在此示例中，第 002 行引入了 AutoIP 包含文件 nx_auto_ip.h。接下来，在第 090 行的“tx_application_define”中创建了 NetX AutoIP 实例。请注意，NetX AutoIP 控制块“auto_ip_0”以前在第 015 行定义为全局变量。创建成功后，将在第 098 行启动 NetX AutoIP。IP 地址更改回调函数处理过程从第 105 行开始，用于处理后续冲突或可能的 DHCP 地址解析。

> **NOTE**
> 下面的示例假定主机设备是单宿主设备。对于多宿主设备，主机应用程序可以使用 NetX AutoIP 服务 nx_auto_ip_interface_set 指定用于探测 IP 地址的辅助网络接口。要详细了解如何设置多宿主应用程序，请参阅《NetX 用户指南》。另请注意，主机应用程序应使用 NetX API nx_status_ip_interface_check 验证 AutoIP 是否已获得 IP 地址。

AutoIP 与 NetX 结合使用的示例

```
000 #include "tx_api.h"
001 #include "nx_api.h"
```

```
#include "nx_auto_ip.h"

#define         DEMO_STACK_SIZE         4096

/* Define the ThreadX and NetX object control blocks... */

TX_THREAD         thread_0;
NX_PACKET_POOL    pool_0;
NX_IP             ip_0;


/* Define the AUTO IP structures for the IP instance. */

NX_AUTO_IP         auto_ip_0;


/* Define the counters used in the demo application... */

ULONG             thread_0_counter;
ULONG             address_changes;
ULONG             error_counter;


/* Define thread prototypes. */

void     thread_0_entry(ULONG thread_input);
void     ip_address_changed(NX_IP *ip_ptr, VOID *auto_ip_address);
void     _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);


/* 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 the main thread. */
    tx_thread_create(&thread_0, "thread 0", thread_0_entry, 0,
                     pointer, DEMO_STACK_SIZE,
                     16, 16, 1, TX_AUTO_START);

    pointer = pointer + DEMO_STACK_SIZE;

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

    /* Create a packet pool. */
    status = nx_packet_pool_create(&pool_0, "NetX Main Packet Pool", 128,
                                   pointer, 4096);
                                   pointer = pointer + 4096;

    if (status)
        error_counter++;
```

```
    error_counter++;

/* Create an IP instance. */
status = nx_ip_create(&ip_0, "NetX IP Instance 0", IP_ADDRESS(0, 0, 0, 0),
                      0xFFFFFF00UL, &pool_0, _nx_ram_network_driver,
                      pointer, 4096, 1);
                      pointer = pointer + 4096;

if (status)
    error_counter++;

/* Enable ARP and supply ARP cache memory for IP Instance 0. */
status = nx_arp_enable(&ip_0, (void *) pointer, 1024);
pointer = pointer + 1024;

/* Check ARP enable status. */
if (status)
    error_counter++;

/* Create the AutoIP instance for IP Instance 0. */
status = nx_auto_ip_create(&auto_ip_0, "AutoIP 0", &ip_0, pointer, 4096, 1);
pointer = pointer + 4096;

/* Check AutoIP create status. */
if (status)
    error_counter++;

/* Start AutoIP instances. */
status = nx_auto_ip_start(&auto_ip_0, 0 /*IP_ADDRESS(169,254,254,255)*/);

/* Check AutoIP start status. */
if (status)
    error_counter++;

/* Register an IP address change function for IP Instance 0. */
status = nx_ip_address_change_notify(&ip_0, ip_address_changed,
                                     (void *) &auto_ip_0);

/* Check IP address change notify status. */
if (status)
    error_counter++;
}


/* Define the test thread. */

void thread_0_entry(ULONG thread_input)
{

UINT      status;
ULONG     actual_status;


     /* Wait for IP address to be resolved. */
    do
    {

        /* Call IP status check routine. */
        status = nx_ip_status_check(&ip_0, NX_IP_ADDRESS_RESOLVED,
                                    &actual_status, 10000);

    } while (status != NX_SUCCESS);

    /* Since the IP address is resolved at this point, the application
    can now fully utilize NetX! */

    while(1)
    {

```

```
139
140
141             /* Increment thread 0's counter. */
142             thread_0_counter++;
143
144             /* Sleep... */
145             tx_thread_sleep(10);
146         }
147     }
148
149
150     void ip_address_changed(NX_IP *ip_ptr, VOID *auto_ip_address)
151     {
152
153     ULONG         ip_address;
154     ULONG         network_mask;
155     NX_AUTO_IP    *auto_ip_ptr;
156
157
158     /* Setup pointer to auto IP instance. */
159     auto_ip_ptr = (NX_AUTO_IP *) auto_ip_address;
160
161     /* Pickup the current IP address. */
162     nx_ip_address_get(ip_ptr, &ip_address, &network_mask);
163
164     /* Determine if the IP address has changed back to zero. If so,
165     make sure the AutoIP instance is started. */
166     if (ip_address == 0)
167     {
168
169         /* Get the last AutoIP address for this node. */
170         nx_auto_ip_get_address(auto_ip_ptr, &ip_address);
171
172         /* Start this AutoIP instance. */
173         nx_auto_ip_start(auto_ip_ptr, ip_address);
174     }
175
176     /* Determine if IP address has transitioned to a non local IP address. */
177     else if ((ip_address & 0xFFFF0000UL) != IP_ADDRESS(169, 254, 0, 0))
178     {
179
180         /* Stop the AutoIP processing. */
181         nx_auto_ip_stop(auto_ip_ptr);
182     }
183
184     /* Increment a counter. */
185     address_changes++;
186 }
```

配置选项

有多个配置选项可用于生成 NetX AutoIP。下面是所有选项的列表，其中包含每个选项的详细说明：

- **NX_DISABLE_ERROR_CHECKING**：如果定义此选项，则会删除基本的 AutoIP 错误检查。通常在调试应用程序后使用此选项。
- **NX_AUTO_IP_PROBE_WAIT**：发送第一个探测前等待的秒数。默认情况下，此值定义为 1。
- **NX_AUTO_IP_PROBE_NUM**：要发送的 ARP 探测数。默认情况下，此值定义为 3。
- **NX_AUTO_IP_PROBE_MIN**：发送探测之间等待的最小秒数。默认情况下，此值定义为 1。
- **NX_AUTO_IP_PROBE_MAX**：发送探测之间等待的最大秒数。默认情况下，此值定义为 2。
- **NX_AUTO_IP_MAX_CONFLICTS**：增加处理延迟时间前的 AutoIP 冲突数。默认情况下，此值定义为 10。
- **NX_AUTO_IP_RATE_LIMIT_INTERVAL**：超过冲突总数后延长等待时间的秒数。默认情况下，此值定义为 60。
- **NX_AUTO_IP_ANNOUNCE_WAIT**：发送公告前等待的秒数。默认情况下，此值定义为 2。

- **NX_AUTO_IP_ANNOUNCE_NUM**：要发送的 ARP 公告数。默认情况下，此值定义为 2。
- **NX_AUTO_IP_ANNOUNCE_INTERVAL**：发送公告之间等待的秒数。默认情况下，此值定义为 2。
- **NX_AUTO_IP_DEFEND_INTERVAL**：在防御公告之间等待的秒数。默认情况下，此值定义为 10。

第 3 章 - Azure RTOS NetX AutoIP 服务的说明

2021/4/29 •

本章按字母顺序介绍如下所列的所有 Azure RTOS NetX AutoIP 服务。

在以下 API 说明的"返回值"部分，以 **粗体** 显示的值不受 NX_DISABLE_ERROR_CHECKING 定义（用于禁用 API 错误检查）影响，而对于非粗体值，则会完全禁用该检查。

- nx_auto_ip_create：创建 AutoIP 实例
- nx_auto_ip_delete：删除 AutoIP 实例
- nx_auto_ip_get_address：获取当前 AutoIP 地址
- nx_auto_ip_set_interface：设置需要 AutoIP 地址的 IP 接口
- nx_auto_ip_start：启动 AutoIP 处理
- nx_auto_ip_stop：停止 AutoIP 处理

nx_auto_ip_create

创建 AutoIP 实例

原型

```
UINT nx_auto_ip_create(NX_AUTO_IP *auto_ip_ptr, CHAR *name,
            NX_IP *ip_ptr, VOID *stack_ptr, ULONG stack_size,
            UINT priority);
```

说明

此服务在指定的 IP 实例上创建 AutoIP 实例。

- auto_ip_ptr：指向 AutoIP 控制块的指针。
- name：AutoIP 实例的名称。
- ip_ptr：指向 IP 实例的指针。
- stack_ptr：指向 AutoIP 线程堆栈区域的指针。
- stack_size：AutoIP 线程堆栈区域的大小。
- priority：AutoIP 线程的优先级。

> **NOTE**
> 如果使用 DHCP，则 DHCP 线程的优先级必须高于 IP 实例线程和 AutoIP 线程的优先级。

返回值

- NX_SUCCESS：(0x00) 创建 AutoIP 成功。
- NX_AUTO_IP_ERROR：(0xA00) 创建 AutoIP 出错。
- NX_PTR_ERROR：(0x16) AutoIP、ip_ptr 或堆栈指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

初始化和线程

示例

```
/* Create the AutoIP instance "auto_ip_0" on "ip_0". */
status = nx_auto_ip_create(&auto_ip_0, "AutoIP 0", &ip_0, pointer, 4096, 1);

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

另请参阅

nx_auto_ip_delete、nx_auto_ip_set_interface、nx_auto_ip_get_address、nx_auto_ip_start、nx_auto_ip_stop

nx_auto_ip_delete

删除 AutoIP 实例

原型

```
UINT nx_auto_ip_delete(NX_AUTO_IP *auto_ip_ptr);
```

说明

此服务删除所指定 IP 实例上先前创建的 AutoIP 实例。

输入参数

- auto_ip_ptr：指向 AutoIP 控制块的指针。

返回值

- NX_SUCCESS：(0x00) 删除 AutoIP 成功。
- NX_AUTO_IP_ERROR：(0xA00) 删除 AutoIP 出错。
- NX_PTR_ERROR (0x16)：AutoIP 指针无效。
- NX_CALLER_ERROR (0x11)：此服务的调用方无效。

允许来自

线程数

示例

```
/* Delete the AutoIP instance "auto_ip_0." */
status = nx_auto_ip_delete(&auto_ip_0);

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

另请参阅

nx_auto_ip_create、nx_auto_ip_set_interface、nx_auto_ip_get_address、nx_auto_ip_start、nx_auto_ip_stop

nx_auto_ip_get_address

获取当前 AutoIP 地址

原型

```
UINT nx_auto_ip_get_address(NX_AUTO_IP *auto_ip_ptr,
                            ULONG *local_ip_address);
```

说明

此服务检索当前设置的 AutoIP 地址，如果没有当前 AutoIP 地址，则会返回 IP 地址 0.0.0.0。

输入参数

- auto_ip_ptr：指向 AutoIP 控制块的指针。
- local_ip_address：用于返回 IP 地址的目标。

返回值

- NX_SUCCESS：(0x00) 获取 AutoIP 地址成功。
- NX_AUTO_IP_NO_LOCAL：(0xA01) 没有有效的 AutoIP 地址。
- NX_PTR_ERROR：(0x16) AutoIP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

初始化、计时器、线程和 ISR

示例

```
ULONG local_address;

/* Get the AutoIP address resolved by the instance "auto_ip_0." */
status = nx_auto_ip_get_address(&auto_ip_0, &local_address);

/* If status is NX_SUCCESS the local IP address is in "local_address." */
```

另请参阅

nx_auto_ip_create、nx_auto_ip_set_interface、nx_auto_ip_delete, nx_auto_ip_start、nx_auto_ip_stop

nx_auto_ip_set_interface

设置 AutoIP 的网络接口

原型

```
UINT nx_auto_ip_set_interface(NX_AUTO_IP *auto_ip_ptr,
                              UINT interface_index);
```

说明

此服务设置要由 AutoIP 探测网络 IP 地址的网络接口索引。默认值为零（主网络接口）。仅适用于多宿主设备。

输入参数

- auto_ip_ptr：指向 AutoIP 控制块的指针。
- interface_index：要探测 IP 地址的接口

返回值

- NX_SUCCESS：(0x00) 设置 AutoIP 接口成功
- NX_AUTO_IP_BAD_INTERFACE_INDEX：(0xA02) 网络接口无效
- NX_PTR_ERROR：(0x16) AutoIP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

初始化、计时器、线程和 ISR

示例

```
ULONG interface_index;

/* Set the network interface on which AutoIP probes for host address. */
status = nx_auto_ip_set_interface(&auto_ip_0, interface_index);

/* If status is NX_SUCCESS the network interface is valid and set in the AutoIP control block auto_ip_0. */
```

另请参阅

nx_auto_ip_create、nx_auto_ip_get_address、nx_auto_ip_delete、nx_auto_ip_start、nx_auto_ip_stop

nx_auto_ip_start

启动 AutoIP 处理

原型

```
UINT nx_auto_ip_start(NX_AUTO_IP *auto_ip_ptr,
                      ULONG starting_local_address);
```

说明

此服务在先前创建的 AutoIP 实例上启动 AutoIP 协议。

输入参数

- auto_ip_ptr：指向 AutoIP 控制块的指针。
- starting_local_address：可选的 AutoIP 启动地址。值为 IP_ADDRESS(0,0,0,0) 表示应派生随机的 AutoIP 地址。否则，如果指定了有效的 AutoIP 地址，则 NetX AutoIP 会尝试分配该地址。

返回值

- NX_SUCCESS：(0x00) 启动 AutoIP 成功。
- NX_AUTO_IP_ERROR：(0xA00) 启动 AutoIP 出错。
- NX_PTR_ERROR：(0x16) AutoIP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

初始化和线程

示例

```
/* Start the AutoIP instance "auto_ip_0." */
status = nx_auto_ip_start(&auto_ip_0, IP_ADDRESS(0,0,0,0));

/* If status is NX_SUCCESS an AutoIP instance was successfully started. */
```

另请参阅

nx_auto_ip_create、nx_auto_ip_set_interface、nx_auto_ip_delete、nx_auto_ip_get_address、nx_auto_ip_stop

nx_auto_ip_stop

停止 AutoIP 处理

原型

```
UINT nx_auto_ip_stop(NX_AUTO_IP *auto_ip_ptr);
```

说明

此服务在先前创建并启动的 AutoIP 实例上停止 AutoIP 协议。通常，此服务在 IP 地址通过 DHCP 或手动更改为非 AutoIP 地址时使用。

输入参数

- auto_ip_ptr：指向 AutoIP 控制块的指针。

返回值

- NX_SUCCESS：(0x00) 停止 AutoIP 成功。
- NX_AUTO_IP_ERROR：(0xA00) 停止 AutoIP 出错。
- NX_PTR_ERROR：(0x16) AutoIP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

线程数

示例

```
/* Stop the AutoIP instance "auto_ip_0." */
status = nx_auto_ip_stop(&auto_ip_0);

/* If status is NX_SUCCESS an AutoIP instance was successfully stopped. */
```

另请参阅

nx_auto_ip_create、nx_auto_ip_set_interface、nx_auto_ip_delete、nx_auto_ip_get_address、nx_auto_ip_start

第 1 章 - Azure RTOS NetX BSD 简介

2021/4/29 •

NetX BSD 套接字 API 兼容性包装器支持一些基本的 BSD 套接字 API 调用，存在一些限制，并在底层利用 NetX 基元。由于此包装器利用内部 NetX 基元并绕过基本的 NetX 错误检查，因此此 BSD 套接字 API 兼容性层的执行速度应比典型的 BSD 实现快或稍快。

BSD 套接字 API 兼容性包装器源代码

BSD 包装器源代码设计简单，仅由两个文件组成，即 nx_bsd.h 和 nx_bsd.c。nx_bsd.h 文件定义所有必需的 BSD 套接字 API 包装器常量和子例程原型，而 nx_bsd.c 包含实际的 BSD 套接字 API 兼容性源代码。这些 BSD 包装器源文件是所有 NetX 支持包所共有的。

包中包含：

- nx_bsd.c：包装器源代码
- nx_bsd.h：主头文件

示例演示程序：

- bsd_demo_tcp.c：具有单一 TCP 服务器和客户端的演示
- bsd_demo_udp：具有两个 UDP 客户端和一个 UDP 服务器的演示

第 2 章 - 安装和使用 Azure RTOS NetX BSD

2021/4/29 •

本章介绍与安装、设置和使用 Azure RTOS NetX BSD 组件相关的各种问题。

产品分发

可以从我们的公共源代码存储库获取 Azure RTOS NetX BSD，网址为：https://github.com/azure-rtos/netx/。此软件包包含两个源文件和一个包含本文档的 PDF 文件，如下所示：

- nx_bsd.h：NetX BSD 的头文件
- nx_bsd.c：NetX BSD 的 C 源文件
- nx_bsd.pdf：NetX BSD 用户指南

演示文件：

- bsd_demo_tcp.c
- bsd_demo_udp.c

NetX BSD 安装

若要使用 NetX BSD，应将之前提到的全部分发文件复制到 NetX 所安装的目录。例如，如果 NetX 安装在“\threadx\arm7\green”目录中，则应将 nx_bsd.h 和 nx_bsd.c 文件复制到此目录中。

生成 BSD 应用程序的 ThreadX 和 NetX 组件

ThreadX

ThreadX 库必须在线程本地存储中定义 bsd_errno。建议完成以下过程：

1. 在 tx_port.h 中，按如下所示设置其中一个 TX_THREAD_EXTENSION 宏：

```
#define TX_THREAD_EXTENSION_3     int bsd_errno
```

2. 重新生成 ThreadX 库。

> **NOTE**
> 如果 TX_THREAD_EXTENSION_3 已在使用中，用户可以随意使用其他任意一个 TX_THREAD_EXTENSION 宏。

NetX

在使用 NetX BSD 服务之前，必须在定义了 NX_ENABLE_EXTENDED_NOTIFY_SUPPORT 的情况下（例如，在 nx_user.h 中定义）生成 NetX 库。此参数默认未定义。

使用 NetX BSD

使用适用于 NetX 的 BSD 非常简单。总体上，应用程序代码必须先包含 tx_api.h 和 nx_api.h，然后包含 nx_bsd.h，才能分别使用 ThreadX 和 NetX。包含 nx_bsd.h 之后，应用程序代码即可使用本指南后文所指定的 BSD 服务。应用程序还必须在生成过程中包含 nx_bsd.c。此文件必须采用与其他应用程序文件相同的方式进行编译，并且其对象窗体必须与应用程序文件一起链接。这就是使用 NetX BSD 所需的一切。

若要利用 NetX BSD 服务，应用程序必须创建 IP 实例和数据包池，并调用 bsd_initialize 以初始化 BSD 服务。本指南后文的"小型示例"部分对此作了演示。原型如下所示：

```
INT bsd_initialize(NX_IP *default_ip, NX_PACKET_POOL *default_pool,
                   CHAR *free_memory_ptr, ULONG bsd_thread_stack_size,
                   UINT bsd_thread_priority);
```

最后三个参数用于创建线程以执行定期任务，例如检查是否有 TCP 事件以及定义线程堆栈空间。

> **NOTE**
> 与按网络字节顺序工作的 BSD 套接字不同，NetX 按主机处理器的主机字节顺序工作。出于源代码兼容性的原因，已定义宏 htons()、ntohs()、htonl() 和 ntohl()，但请勿修改传递的参数。

NetX BSD 限制

由于性能和体系结构问题，NetX BSD 仅支持部分 BSD 4.3 套接字功能：

send、recv、sendto 和 recvfrom 调用不支持 INT 标志。

支持 DNS 的 NetX BSD

如果定义了 NX_BSD_ENABLE_DNS，则 NetX BSD 可以发送 DNS 查询来获取主机名或主机 IP 信息。此功能要求先前已使用 nx_dns_create 服务创建 NetX DNS 客户端。必须通过使用 nx_dns_server_add 添加服务器定制，向 DNS 实例注册一个或多个已知的 DNS 服务器 IP 地址。

DNS 服务和内存分配由 getaddrinfo 和 getnameinfo 服务使用：

```
INT getaddrinfo(const CHAR *node, const CHAR *service,
              const struct addrinfo *hints, struct addrinfo **res)

INT getnameinfo(const struct sockaddr *sa, socklen_t salen,
              char *host, size_t hostlen, char *serv, size_t servlen, int flags)
```

当 BSD 应用程序使用主机名来调用 getaddrinfo 时，NetX BSD 将调用以下任何服务以获取 IP 地址：

- nx_dns_ipv4_address_by_name_get
- nx_dns_cname_get

对于 nx_dns_ipv4_address_by_name_get，NetX BSD 会使用 ipv4_addr_buffer 内存区域。这些缓冲区的大小由 (`NX_BSD_IPV4_ADDR_PER_HOST * 4`) 定义。

为从 getaddrinfo 返回地址信息，NetX BSD 使用 ThreadX 块内存表 nx_bsd_addrinfo_pool_memory，其内存区域是由另一组可配置选项 NX_BSD_IPV4_ADDR_MAX_NUM 定义。

有关上述配置选项的详细信息，请参阅"配置选项"。

此外，如果已定义 NX_DNS_ENABLE_EXTENDED_RR_TYPES，并且主机输入是规范名称，则 NetX BSD 将从先前创建的块池 _nx_bsd_cname_block_pool 动态分配内存。

> **NOTE**
> 调用 getaddrinfo 之后，BSD 应用程序负责使用 freeaddrinfo 服务将 res 参数所指向的内存释放回到块表中。

配置选项

nx_bsd.h 中的用户可配置选项允许应用程序根据其特定需求微调 NetX BSD 套接字。这些参数的列表如下所示：

- NX_BSD_TCP_WINDOW：用于 TCP 套接字创建调用。对于 100Mb 以太网，典型窗口大小是 65535。默认值为 65535。
- NX_BSD_SOCKFD_START：这是 BSD 套接字文件描述符起始值的逻辑索引。默认情况下，此选项为 32。
- NX_BSD_MAX_SOCKETS：指定 BSD 层中可用的最大插槽总数，并且必须是 32 的倍数。此值默认为 32。
- NX_BSD_SOCKET_QUEUE_MAX：指定存储在接收套接字队列上的最大 UDP 数据包数目。此值默认为 5。
- NX_BSD_MAX_LISTEN_BACKLOG：指定 BSD TCP 套接字侦听队列（积压工作）的大小。默认值为 5。
- NX_MICROSECOND_PER_CPU_TICK：指定每一个计时器中断的微秒数。
- NX_BSD_TIMEOUT：指定 BSD 在 NetX 内部调用上需要的超时值，以计时器时钟周期为单位。默认值为 20*NX_IP_PERIODIC_RATE。
- NX_BSD_TCP_SOCKET_DISCONNECT_TIMEOUT：指定 NetX 断开连接调用的超时值，以计时器时钟周期为单位。默认值为 1。
- NX_BSD_PRINT_ERRORS：如果设置此参数，则 BSD 函数的错误状态返回会返回发生错误的行号及错误类型，例如 NX_SOC_ERROR。这要求应用程序开发人员定义调试输出。默认设置是已禁用，并且在 nx_bsd.h 中未指定任何调试输出。
- NX_BSD_TIMER_RATE：BSD 定期计时器任务运行之前经过的时间间隔。默认值为 1 秒 (1 * NX_IP_PERIODIC_RATE)。
- NX_BSD_TIMEOUT_PROCESS_IN_TIMER：如果设置此选项，则允许在系统计时器上下文中执行 BSD 超时进程。默认行为是已禁用。第 2 章“安装和使用 NetX BSD”中更详细地介绍了此功能。
- NX_BSD_ENABLE_DNS：如果启用此参数，则 NetX BSD 会发送 DNS 查询，以查询主机名或主机 IP 地址。要求先创建并启动 DNS 客户端实例。默认情况下，此参数未启用。
- NX_BSD_IPV4_ADDR_MAX_NUM：getaddrinfo 所返回的最大 IPv4 地址数。此参数与 NX_BSD_IPV4_ADDR_MAX_NUM 共同定义用于动态分配内存的 NetX BSD 块池 nx_bsd_addrinfo_block_pool 的大小，以在 getaddrinfo 中进行信息存储寻址。默认值为 5。
- NX_BSD_IPV4_ADDR_PER_HOST：定义每个 DNS 查询所存储的最大 IPv4 地址数。默认值为 5。

BSD 套接字选项

利用 setsockopt 服务，可以在运行时按套接字启用（或禁用）下列 NetX BSD 套接字选项：

```
INT setsockopt(INT sockID, INT option_level, INT option_name, const
                void *option_value, INT option_length);
```

option_level 有两个不同的设置。

第一类运行时套接字选项是 SOL_SOCKET，对应于套接字级别选项。若要启用套接字级别选项，请调用 setsockopt，并将 option_level 设置为 SOL_SOCKET，将 option_name 设置为特定选项，例如 SO_BROADCAST。若要检索选项设置，请对 option_name 调用 getsockopt，option_level 依旧设置为 SOL_SOCKET。

运行时套接字级别选项的列表如下所示。

- SO_BROADCAST：如果设置此选项，则可以从 Netx 套接字发送和接收广播数据包。这是 NetX 的默认行为。所有套接字都具备此功能。
- SO_ERROR：用于通过 getsockopt 服务，获取所指定套接字的前一套接字操作的套接字状态。所有套接字都具备此功能。
- SO_KEEPALIVE：如果设置此选项，则会启用“TCP 保持活动状态”功能。这要求在 nx_user.h 中定义 NX_TCP_ENABLE_KEEPALIVE 的情况下生成 NetX 库。默认情况下，该功能已禁用。
- SO_RCVTIMEO：设置在 NetX BSD 套接字上接收数据包时的等待选项（以秒为单位）。默认值为 NX_WAIT_FOREVER (0xFFFFFFFF)，或者，如果已启用“非阻止性”，则为 NX_NO_WAIT (0x0)。

- SO_RCVBUF：设置 TCP 套接字的窗口大小。对于 BSD TCP 套接字，默认值 NX_BSD_TCP_WINDOW 设置为 64k。若要将大小设置为超过 65535，则需要在已定义 NX_TCP_ENABLE_WINDOW_SCALING 的情况下生成 NetX 库。
- SO_REUSEADDR：如果设置此选项，则可将多个套接字映射到同一个端口。典型的用法是用于 TCP 服务器套接字。这是 NetX 套接字的默认行为。

第二类运行时套接字选项是 IP 选项级别。若要启用 IP 级别选项，请调用 setsockopt，并将 option_level 设置为 IP_PROTO，将 option_name 设置为选项，例如 IP_MULTICAST_TTL。若要检索选项设置，请对 option_name 调用 getsockopt，option_level 依旧设置为 IP_PROTO。

运行时 IP 级别选项的列表如下所示。

- IP_MULTICAST_TTL：设置 UDP 套接字的生存时间。创建套接字时，默认值为 NX_IP_TIME_TO_LIVE (0x80)。通过使用此套接字选项来调用 setsockopt，可以重写该值。
- IP_ADD_MEMBERSHIP：如果设置此选项，则 BSD 套接字（仅适用于 UDP 套接字）可以加入指定的 IGMP 组。
- IP_DROP_MEMBERSHIP：如果设置此选项，则 BSD 套接字（仅适用于 UDP 套接字）可以离开指定的 IGMP 组。

小型示例系统

下面的图 1.0 显示使用 NetX BSD 的方式示例。在此示例中，第 7 行引入包含文件 nx_bsd.h。接下来，在第 20 行和第 21 行，将 IP 实例 bsd_ip 和数据包池 bsd_pool 创建为全局变量。请注意，此演示使用 ram（虚拟）网络驱动程序（第 41 行）。在此示例中，客户端与服务器共享单个 IP 实例上的同一个 IP 地址。

客户端线程和服务器线程分别在 tx_application_define 中的第 303 行和第 309 行创建，该函数用于设置应用程序，并在第 293-361 行定义。在第 327 行创建 IP 实例成功后，第 350 行对该 IP 实例启用 TCP 服务。在可以使用 BSD 服务之前，最后一项要求是在第 360 行调用 bsd_initialize，以设置 BSD 所需的所有数据结构以及 NetX 和 ThreadX 资源。

在服务器线程入口函数 thread_1_entry（在第 381-397 行定义），应用程序会等待驱动程序使用网络参数初始化 NetX。完成后，它会调用在第 146-253 行定义的 tcpServer，以处理有关设置 TCP 服务器套接字的详细信息。

tcpServer 在第 159 行通过调用 socket 服务来创建主套接字，并在第 176 行使用 bind 调用将该套接字与侦听套接字绑定。然后，在第 191 行，将其配置为侦听连接请求。请注意，主套接字不接受连接请求。它会以连续循环的形式运行，而在每次循环时都会调用 select 来检测连接请求。在第 218 行调用 accept 服务之后，系统会将连接请求分配给从 BSD 套接字数组中选择的辅助 BSD 套接字。

在客户端，第 366-377 行定义的客户端线程入口函数 thread_0_entry 也应等待驱动程序初始化 NetX。在这里，我们只需等待服务器端执行该操作。然后，调用第 54-142 行定义的 tcpClient，以处理有关设置 TCP 客户端套接字和请求 TCP 连接的详细信息。

该 TCP 客户端套接字在第 68 行创建。该套接字与指定的 IP 地址绑定，并在第 84 行调用 connect 以尝试连接至 TCP 服务器。您现已准备就绪，可以开始发送和接收数据包。

```
1 /*  This is a small demo of BSD Wrapper for the high-performance NetX TCP/IP stack.
2     This demo demonstrate TCP connection, disconnection, sending, and receiving using
3     ARP and a simulated Ethernet driver. */
4
5 #include     "tx_api.h"
6 #include     "nx_api.h"
7 #include     "nx_bsd.h"
8 #include     <string.h>
9 #include     <stdlib.h>
10
11 #define         DEMO_STACK_SIZE         (16*1024)
12
13
14 /* Define the ThreadX and NetX object control blocks... */
15
16 TX_THREAD       thread_0;
```

```
17 TX_THREAD       thread_1;
18
19
20 NX_PACKET_POOL  bsd_pool;
21 NX_IP           bsd_ip;
22
23
24 /* Define the counters used in the demo application... */
25
26 ULONG           error_counter;
27
28 /* Define fd_set for select call */
29 fd_set          master_list,read_ready,read_ready1;
30
31
32 /* Define thread prototypes. */
33
34 VOID            thread_0_entry(ULONG thread_input);
35 VOID            thread_1_entry(ULONG thread_input);
36
37 VOID            tcpClient(CHAR *msg0);
38 VOID            tcpServer(VOID);
39 INT             HandleClient(INT sock);
40
41 VOID            _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);
42
43
44 /* Define main entry point. */
45
46 int main()
47 {
48
49     /* Enter the ThreadX kernel. */
50     tx_kernel_enter();
51 }
52
53
54 VOID tcpClient(CHAR *msg0)
55 {
56
57 INT     status,status1,send_counter;
58 INT     sock_tcp_1,length,length1;
59 struct  sockaddr_in echoServAddr;       /* Echo server address */
60 struct  sockaddr_in localAddr;          /* Local address */
61 struct  sockaddr_in remoteAddr;         /* Remote address */
62
63 UINT    echoServPort; /* Echo Server Port */
64 CHAR    rcvBuffer1[32]
65
66
67     /* Create BSD TCP Socket */
68     sock_tcp_1 = **socket**( PF_INET, SOCK_STREAM, IPPROTO_TCP);
69     if (sock_tcp_1 == -1)
70     {
71         printf("\nError: BSD TCP Client socket create \n");
72         return;
73     }
74
75     printf("\nBSD TCP Client socket created %lu \n", sock_tcp_1);
76     /* Fill destination port and IP address */
77     echoServPort = 12;
78     memset(&echoServAddr, 0, sizeof(echoServAddr));
79     echoServAddr.sin_family = PF_INET;
80     echoServAddr.sin_addr.s_addr = htonl(0x01020304);
81     echoServAddr.sin_port = echoServPort;
82
83     /* Now connect this client the server */
84     status1 = connect(sock_tcp_1, (struct sockaddr *)&echoServAddr, sizeof(echoServAddr));
85     /* Check for error. */
```

```
86      if (status1 != OK)
87      {
88          printf("\nError: BSD TCP Client socket Connect, %d \n",sock_tcp_1);
89          status = soc_close(sock_tcp_1);
90          if (status != ERROR)
91              printf("\nConnect ERROR so BSD Client Socket Closed: %d\n",sock_tcp_1);
92          else
93              printf("\nError: BSD Client Socket close %d\n",sock_tcp_1);
94          return;
95      }
96      /* Get and print source and destination information */
97      printf("\nBSD TCP Client socket: %d connected \n", sock_tcp_1);
98
99      status = getsockname(sock_tcp_1, (struct sockaddr *)&localAddr, &length);
100     printf("Client port = %lu , Client = %lu,",
             localAddr.sin_port, localAddr.sin_addr.s_addr);
101     status = getpeername( sock_tcp_1, (struct sockaddr *) &remoteAddr, &length1);
102     printf("Remote port = %lu, Remote IP = %lu \n",
             remoteAddr.sin_port remoteAddr.sin_addr.s_addr);
103
104     send_counter = 1;
105
106     /* Now receive the echoed packet from the server */
107     while(1)
108     {
109         tx_thread_sleep(2);
110
111         printf("\nClient sock: %d Sending packet No: %d to
                 server\n",sock_tcp_1,send_counter);
112         status = send(sock_tcp_1,msg0, sizeof(msg0), 0);
113         if (status == ERROR)
114             printf("Error: BSD Client Socket send %d\n",sock_tcp_1);
115         else
116         {
117             printf("\nMessage sent: %s\n",msg0);
118             send_counter++;
119         }
120
121         status = recv(sock_tcp_1, (VOID *)rcvBuffer1, 31,0);
122         if (status == 0)
123             break;
124
125         rcvBuffer1[status] = 0;
126
127         if (status != ERROR)
128             printf("\nBSD Client Socket: %d received %lu bytes: %s ",
                         sock_tcp_1,(ULONG)status,rcvBuffer1);
129         else
130             printf("\nError: BSD Client Socket receive %d \n",sock_tcp_1);
131
132     }
133
134     /* close this client socket */
135     status = soc_close(sock_tcp_1);
136     if (status != ERROR)
137         printf("\nBSD Client Socket Closed %d\n",sock_tcp_1);
138     else
139         printf("\nError: BSD Client Socket close %d \n",sock_tcp_1);
140
141     /* End */
142 }
143
144
145
146 void tcpServer(void)
147 {
148
149 INT         status,status1,sock,sock_tcp_2,i;
150 struct      sockaddr_in echoServAddr; /* Echo server address */
```

```
151 struct      sockaddr_in ClientAddr;
152
153 INT         Clientlen;
154 UINT        echoServPort; /* Echo Server Port */
155
156 INT         maxfd;
157
158 /* Create BSD TCP Server Socket */
159     sock_tcp_2 = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP);
160     if (sock_tcp_2 == -1)
161     {
162         printf("Error: BSD TCP Server socket create\n");
163         return;
164     }
165     else
166         printf("BSD TCP Server socket created \n");
167
168     /* Now fill server side information */
169     echoServPort = 12;
170     memset(&echoServAddr, 0, sizeof(echoServAddr));
171     echoServAddr.sin_family = PF_INET;
172     echoServAddr.sin_addr.s_addr = htonl(0x01020304);
173     echoServAddr.sin_port = echoServPort;
174
175     /* Bind this server socket */
176         status = bind(sock_tcp_2, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr));
177     if (status < 0)
178     {
179         printf("Error: BSD TCP Server Socket Bind \n");
180         return;
181     }
182     else
183         printf("BSD TCP Server Socket bound \n");
184
185     FD_ZERO(&master_list);
186     FD_ZERO(&read_ready);
187     FD_SET(sock_tcp_2,&master_list);
188     maxfd = sock_tcp_2;
189
190     /* Now listen for any client connections for this server socket */
191     status = listen(sock_tcp_2,5);
192     if (status < 0)
193     {
194         printf("Error: BSD TCP Server Socket Listen\n");
195         return;
196     }
197     else
198         printf("BSD TCP Server Socket Listen complete, ");
199
200     /* All set to accept client connections */
201     printf("Now accepting client connections\n");
202
203     /* Loop to create and establish server connections. */
204     while(1)
205     {
206
207         read_ready = master_list;
208         tx_thread_sleep(2); /* Allow some time to other threads too */
209         status = select(maxfd+1,&read_ready,0,0,0);
210         if (status == ERROR)
211         {
212             continue;
213         }
214
215         status = FD_ISSET(sock_tcp_2,&read_ready);
216         if(status)
217         {
218             sock = accept(sock_tcp_2,(struct sockaddr*)&ClientAddr, &Clientlen);
219
```

```
            /* Add this new connection to our master list */
            FD_SET(sock,&master_list);
            if ( sock \maxfd)
            {
                maxfd = sock;
            }

            continue;
        }
        for (i = 0; i < (maxfd+1); i++)
        {
            if (( i != sock_tcp_2) && (FD_ISSET(i,&master_list)) &&
                (FD_ISSET(i,&read_ready)))
            {
                status1 = HandleClient(i);
                if (status1 == 0)
                {
                    status1 = soc_close(i);
                    if (status1 == OK)
                    {
                        FD_CLR(i,&master_list);
                        printf("\nBSD Server Socket:%d closed\n",i);
                    }
                    else
                        printf("\nError:BSD Server Socket:%d close\n",i);
                }

            }
        }

        /* Loop back to check any next client connection */

    } /* While(1) ends */

}

CHAR     rcvBuffer[128];

INT     HandleClient(INT sock)
{

INT     status;


    status = recv(sock, (VOID *)rcvBuffer, 128,0);
    if (status == ERROR )
    {
        printf("\n BSD Server Socket:%d receive \n",sock);
        return(ERROR);
    }

    /* a zero return from a recv() call indicates client is terminated! */
    if (status == 0)
    {
        /* Done with this client , close this secondary server socket */
        return(status);
    }

    /* print data received from the client */
    printf("\nBSD Server Socket:%d received %lu bytes: %s \n", sock, (ULONG)status,rcvBuffer);

    /* And echo the same data to the client */
    status = send(sock,rcvBuffer, status + 1, 0);
    if (status == ERROR )
    {
        printf("\nError: BSD Server Socket:%d send \n",sock);
        return(ERROR);
    }
    return(status);
```

```
    return(status);
}


/* 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 client thread. */
    tx_thread_create(&thread_0, "Client1", thread_0_entry, 0,
        pointer, DEMO_STACK_SIZE, 2, 2, TX_NO_TIME_SLICE, TX_AUTO_START);

    pointer = pointer + DEMO_STACK_SIZE;

    /* Create a server thread. */
    tx_thread_create(&thread_1, "Server", thread_1_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 a BSD packet pool. */
    status = nx_packet_pool_create(&bsd_pool,"NetX BSD Packet Pool", 128
                                   pointer, 16384);
    pointer = pointer + 16384;
    if (status)
    {
        error_counter++;
        printf("Error in creating BSD packet pool\n!");
    }

    /* Create an IP instance for BSD. */
    status = nx_ip_create(&bsd_ip, "NetX IP Instance 2", IP_ADDRESS(1, 2, 3, 4),
                          0xFFFFFF00UL, &bsd_pool, _nx_ram_network_driver,
                          pointer, 2048, 1);

    pointer = pointer + 2048;

    if (status)
    {
        error_counter++;
        printf("Error creating BSD IP instance\n!");
    }

    /* Enable ARP and supply ARP cache memory for BSD IP Instance */
    status = nx_arp_enable(&bsd_ip, (void *) pointer, 1024);
    pointer = pointer + 1024;

    /* Check ARP enable status. */
    if (status)
    {
        error_counter++;
        printf("Error in Enable ARP and supply ARP cache memory to BSD IP instance\n");
    }

    /* Enable TCP processing for BSD IP instances. */

    status = nx_tcp_enable(&bsd_ip);

    /* Check TCP enable status. */
```

```
353     if (status)
354     {
355         error_counter++;
356         printf("Error in Enable TCP \n");
357     }
358
359     /* Now initialize BSD Scoket Wrapper */
360     status = bsd_initialize(&bsd_ip, &bsd_pool,pointer, 2048, 1);
361 }
362
363
364 /* Define the test threads. */
365
366 void     thread_0_entry)ULONG thread_input)
367 {
368
369 CHAR     *msg0 = "Client 1:
                    "ABCDEFGHIJKLMNOPQRSTUVWXYZ<>ABCDEFGHIJKLMNOPQRSTUVWXYZ<> \
                    "ABCDEFGHIJKLMNOPQRSTUVWXYZ<>END";
370
371     /* Wait till Server side is all set */
372     tx_thread_sleep(2);
373     while (1)
374     {
375         tcpClient(msg0);
376         tx_thread_sleep(1);
377     }
378 }
379
380 /* Define the server thread entry function. */
381 void     thread_1_entry(ULONG thread_input)
382 {
383
384 UINT     status;
385 ULONG    actual_status;
386
387 /* Ensure the IP instance has been initialized. */
388 status = nx_ip_status_check(&bsd_ip, NX_IP_INITIALIZE_DONE, &actual_status, 100);
389
390 /* Check status... */
391 if (status != NX_SUCCESS)
392 {
393     error_counter++;
394     return;
395 }
396 /* Start a TCP Server */
397 tcpServer();
398 }
399
```

第 3 章 - Azure RTOS NetX BSD 服务

2021/4/29 •

本章按字母顺序介绍了下面列出的所有 Azure RTOS NetX BSD 基本服务。

```
INT accept(INT sockID, struct sockaddr *ClientAddress, INT *addressLength);

INT bind (INT sockID, struct sockaddr *localAddress, INT addressLength);

INT bsd_initialize(NX_IP *default_ip, NX_PACKET_POOL *default_pool, CHAR
                   *bsd_thread_stack_area, ULONG bsd_thread_stack_size,
                   *UINT bsd_thread_priority);

INT connect(INT sockID, struct sockaddr *remoteAddress, INT addressLength);

INT getpeername( INT sockID, struct sockaddr *remoteAddress, INT *addressLength);

INT getsockname( INT sockID, struct sockaddr *localAddress, INT *addressLength);

INT ioctl(INT sockID, INT command, INT *result);

in_addr_t inet_addr(const_CHAR *buffer);

INT inet_aton(const CHAR *cp_arg, struct in_addr *addr);

CHAR inet_ntoa(struct in_addr address_to_convert);

const CHAR *inet_ntop(INT af, const VOID *src, CHAR *dst, socklen_t size);

INT inet_pton(INT af, const CHAR *src, VOID *dst);

INT listen(INT sockID, INT backlog);

INT recvfrom(INT sockID, CHAR *buffer, INT buffersize, INT flags,
            struct sockaddr *fromAddr, INT *fromAddrLen);

INT recv(INT sockID, VOID *rcvBuffer, INT bufferLength, INT flags);
INT sendto(INT sockID, CHAR *msg, INT msgLength, INT flags,
            struct sockaddr *destAddr, INT destAddrLen);

INT send(INT sockID, const CHAR *msg, INT msgLength, INT flags);

INT select(INT nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            struct timeval *timeout);

INT soc_close ( INT sockID);

INT socket( INT protocolFamily, INT type, INT protocol);

INT fcntl(INT sock_ID, UINT flag_type, UINT f_options);

INT getsockopt(INT sockID, INT option_level, INT option_name, VOID *option_value,
                INT *option_length);

INT setsockopt(INT sockID, INT option_level, INT option_name,
                const VOID *option_value, INT option_length);

INT getaddrinfo(const CHAR *node, const CHAR *service, const struct addrinfo *hints,
                struct addrinfo **res);

VOID freeaddrinfo(struct addrinfo *res);
```

```
INT getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host,
                size_t hostlen, char *serv, size_t servlen, int flags);

VOID nx_bsd_set_service_list(struct NX_BSD_SERVICE_LIST *serv_list_ptr,
                             ULONG serv_list_len);

VOID FD_SET(INT fd, fd_set *fdset);

VOID FD_CLR(INT fd, fd_set *fdset);

INT FD_ISSET(INT fd, fd_set *fdset);

VOID FD_ZERO (fd_set *fdset);
```

第 1 章 - Azure RTOS NetX DHCP 客户端简介

2021/4/29 •

在 NetX 中，应用程序的 IP 地址是其中一个提供给 nx_ip_create 服务调用的参数。如果应用程序可以静态方式或通过用户配置知晓其 IP 地址，则提供该 IP 地址不会造成任何问题。但是，在某些情况下，应用程序不知道或不关心其 IP 地址。在这种情况下，应该提供零 IP 地址给 nx_ip_create 函数，并且应使用 Azure RTOS DHCP 客户端协议动态获取 IP 地址。

动态 IP 地址分配

用于从网络获取动态 IP 地址的基本服务是反向地址解析协议 (RARP)。此协议类似于 ARP，只不过它专门用于获取自身的 IP 地址，而不是查找其他网络节点的 MAC 地址。低级别的 RARP 消息在本地网络上广播，网络上的服务器负责作出 RARP 响应，该响应中包含动态分配的 IP 地址。

尽管 RARP 提供了动态分配 IP 地址的服务，但有几个缺点。最明显的缺陷是，RARP 仅提供 IP 地址动态分配。在大多数情况下，要让设备正确加入网络，需要更多的信息。除 IP 地址之外，大部分设备还需要网络掩码和网关 IP 地址。此外，可能还需要 DNS 服务器 IP 地址和其他网络信息。RARP 无法提供这些信息。

RARP 替代项

为克服 RARP 的缺陷，研究人员已开发出一种更为全面的 IP 地址分配机制，称为 Bootstrap 协议 (BOOTP)。此协议能够动态分配 IP 地址，还可以提供其他重要的网络信息。然而，BOOTP 有一个缺点，即专用于静态网络配置，不支持快速或自动化的地址分配，

而这正是动态主机配置协议 (DHCP) 的优势所在。DHCP 旨在扩展 BOOTP 的基本功能，实现完全自动的 IP 服务器分配和完全动态的 IP 地址分配，其中，IP 地址分配是通过将 IP 地址“租借”给客户端一段指定时间来实现。此外，还可将 DHCP 配置成以静态方式分配 IP 地址，就像 BOOTP 一样。

DHCP 消息

尽管 DHCP 极大地增强 BOOTP 的功能，但 DHCP 使用与 BOOTP 相同的消息格式，并支持与 BOOTP 相同的供应商选项。为执行其功能，DHCP 引入了 7 个新的 DHCP 专用选项，如下所示：

- DISCOVER (1)（由 DHCP 客户端发送）
- OFFER (2)（由 DHCP 服务器发送）
- REQUEST (3)（由 DHCP 客户端发送）
- DECLINE (4)（由 DHCP 客户端发送）
- ACK (5)（由 DHCP 服务器发送）
- NACK (6)（由 DHCP 服务器发送）
- RELEASE (7)（由 DHCP 客户端发送）
- INFORM (8)（由 DHCP 客户端发送）
- FORCERENEW (9)（由 DHCP 客户端发送）

DHCP 通信

DHCP 利用 UDP 协议发送请求和字段响应。在知晓 IP 地址之前，使用 IP 广播地址 255.255.255.255 来发送和接

收包含 DHCP 信息的 UDP 消息。

DHCP 客户端状态机

DHCP 客户端是以状态机的形式实现，该状态机由 nx_dhcp_create 处理期间创建的内部 DHCP 线程处理。DHCP 客户端的主要状态如下所示：

- NX_DHCP_STATE_BOOT：正在使用先前的 IP 地址启动
- NX_DHCP_STATE_INIT：正在没有先前 IP 地址值的情况下启动
- NX_DHCP_STATE_SELECTING：正在等待来自任何 DHCP 服务器的响应
- NX_DHCP_STATE_REQUESTING：已识别到 DHCP 服务器，已发送 IP 地址请求
- NX_DHCP_STATE_BOUND：已建立 DHCP IP 地址租约
- NX_DHCP_STATE_RENEWING：DHCP IP 地址租约续订时间已过，已请求续订
- NX_DHCP_STATE_REBINDING：DHCP IP 地址租约重新绑定时间已过，已请求续订
- NX_DHCP_STATE_FORCERENEW：已建立 DHCP IP 地址租约，由服务器强制续订（当前不支持），或者由调用 nx_dhcp_force_renew 的应用程序强制续订
- NX_DHCP_STATE_ADDRESS_PROBING：正在进行 DHCP IP 地址探测，发送 ARP 探测以检测 IP 地址冲突。

DHCP 客户端多接口支持

以前，实现的 DHCP 客户端仅在单个网络接口上运行。DHCP 客户端的默认行为是在主接口上运行，现在仍是如此。通过调用 nx_dhcp_set_interface_index，应用程序可以在辅助网络接口而非主接口上运行 DHCP，现在仍是如此。

现在，支持在多个接口上并行运行的 DHCP。有关如何同时在多个物理接口上运行 DHCP 客户端的具体详细信息，请参阅第二章中的“同时在多个接口上运行的 DHCP 客户端”。

DHCP 用户请求

一旦 DHCP 服务器授予 IP 地址，DHCP 客户端处理就可以使用 nx_dhcp_user_option_request 服务来请求其他参数（一次一个）。

DHCP 客户端套接字队列

DHCP 客户端会在等待服务器响应自身的同时，从其套接字接收队列中，自动清除来自 DHCP 服务器的发往其他 DHCP 客户端的广播数据包。在繁忙的网络中，如果不这样做，可能会导致丢弃要发往该客户端的数据包。

DHCP RFC

NetX DHCP 符合 RFC2132、RFC2131 以及相关 RFC。

第 2 章 - 安装和使用 Azure RTOS NetX DHCP 客户端

2021/4/29 •

本章介绍与安装、设置和使用 Azure RTOS NetX DHCP 组件相关的各种问题。

产品分发

可在 https://github.com/azure-rtos/netx 上获取 NetX DHCP。该程序包包含两个源文件和一个 PDF 文件（其中包含本文档），如下所示：

- **nx_dhcp.h**：NetX DHCP 的标头文件
- **nx_dhcp.c**：为 NetX DHCP 提供的 C 源文件
- **nx_dhcp.pdf**：NetX DHCP 的 PDF 描述
- **demo_netx_dhcp.c**：NetX DHCP 演示
- **demo_netx_multihome_dhcp_client.c**：在多个接口上运行 DHCP 的 NetX DHCP 客户端演示

安装 DHCP

若要使用 NetX DHCP，应将之前提到的全部分发文件复制到安装了 NetX 的目录中。例如，如果 NetX 安装在“\threadx\arm7\green”目录中，则应将 nx_dhcp.h 和 nx_dhcp.c 文件复制到该目录中。

使用 DHCP

NetX DHCP 使用起来非常简单。大致来说，必须在应用程序代码中加入 tx_api.h 和 nx_api.h，然后加入 nx_dhcp.h，才能分别使用 ThreadX 和 NetX 。在包含 nx_dhcp.h 之后，应用程序代码即可调用本指南后文所指定的 DHCP 函数。在生成过程中，还必须在应用程序中加入 nx_dhcp.c。此文件必须采用与其他应用程序文件相同的方式进行编译，并且文件的对象窗体必须与该应用程序的文件一起链接。这就是使用 NetX DHCP 所需的全部内容。

请注意，由于 DHCP 利用 NetX UDP 服务，因此在使用 DHCP 之前，必须通过 nx_udp_enable 调用启用 UDP。

若要获取先前分配的 IP 地址，DHCP 客户端可以使用“请求”消息和选项 50“请求的 IP 地址”针对 DHCP 服务器启动 DHCP 进程。如果 DHCP 服务器授予 IP 地址给客户端，则会以 ACK 消息响应，否则以 NACK 响应。在后一种情况下，DHCP 客户端会使用“发现”消息而不使用所请求的 IP 地址，以 Init 状态重启 DHCP 进程。主机应用程序会先创建 DHCP 客户端，然后调用 nx_dhcp_request_client_ip API 服务以设置所请求的 IP 地址，再使用 nx_dhcp_start 启动 DHCP 进程 。如需更多详细信息，请参阅本文档其他位置提供的示例 DHCP 应用程序。

处于绑定状态

当 DHCP 客户端处于绑定状态时，DHCP 客户端线程在 NX_DHCP_TIME_INTERVAL 所指定的每个时间间隔会处理一次客户端状态，并减少分配给客户端的 IP 租约的剩余时间。续订时间过去后，DHCP 客户端状态就会更新为 RENEW 状态，客户端随即向 DHCP 服务器请求续订。

将 DHCP 消息发送到服务器

DHCP 客户端具备 API 服务，可供主机应用程序用来将消息发送到 DHCP 服务器。请注意，这些服务并非供主机应用程序手动运行 DHCP 客户端协议使用，因为它们主要发送消息，而无需更新 DHCP 客户端的内部状态。

- nx_dhcp_release：当主机应用程序离开网络或需要放弃其 IP 地址时，此服务会将一条 RELEASE 消息发送到服务器。
- nx_dhcp_decline：如果主机应用程序独立于 DHCP 客户端确定其 IP 地址已被使用，则此服务会将一条 DECLINE 消息发送到服务器。
- nx_dhcp_forcerenew：此服务将 FORCERENEW 消息发送到服务器
- nx_dhcp_send_request：此服务使用 nxd_dhcp_client.h 中指定的 DHCP 消息类型作为参数，将消息发送到服务器 。此服务主要用于发送 DHCP INFORM 消息。

有关这些服务的详细信息，请参阅本文档其他位置的"DHCP 服务说明"。

启动和停止 DHCP 客户端

若要停止 DHCP 客户端，而不考虑其是否已达到绑定状态，主机应用程序可调用 nx_dhcp_stop。

若要重启 DHCP 客户端，主机应用程序必须先使用上述 nx_dhcp_stop 服务停止 DHCP 客户端。然后，主机应用程序可以调用 nx_dhcp_start 来恢复 DHCP 客户端。如果主机应用程序希望清除先前的 DHCP 客户端配置文件（例如，从另一网络上先前的 DHCP 服务器获取的配置文件），则应先调用 nx_dhcp_reinitialize 在内部执行此任务，然后再调用 nx_dhcp_start。

典型的序列可能是：

```
nx_dhcp_stop(&my_dhcp);

nx_dhcp_reinitialize(&my_dhcp);

nx_dhcp_start(&my_dhcp);
```

对于仅在单个 DHCP 接口上运行的 DHCP 应用程序，停止 DHCP 客户端还会停用 DHCP 客户端计时器。因此不再跟踪 IP 租约的剩余时间。在特定接口上停止 DHCP 客户端不会停用 DHCP 客户端计时器，但会停止计时器更新该接口上 IP 租约的剩余时间。

因此，除非主机应用程序需要重新启动或切换网络，否则不建议停止 DHCP 客户端。

将 DHCP 客户端与 Auto IP 配合使用

在 DHCP 和 Auto IP 可以保证提供地址，而 DHCP 服务器无法保证可用或有响应的应用程序中，NetX Duo DHCP 客户端与 Auto IP 协议并行工作。但是，如果主机无法检测到服务器或者无法获分配 IP 地址，则可以切换到 Auto IP 协议以获取本地 IP 地址。不过，这样做之前，建议在 Auto IP 经历"探测"和"防御"阶段期间暂时停止 DHCP 客户端。主机获分配 Auto IP 地址后，就可以重新启动 DHCP 客户端，如果 DHCP 服务器变为可用，则主机 IP 地址可以接受 DHCP 服务器在应用程序运行期间提供的 IP 地址。

NetX Auto IP 有地址更改通知，可供主机在 IP 地址更改时监视其活动。

小型示例系统

下面的图 1.1 举例说明了如何使用 NetX。在第 101 行创建"my_thread_entry"DHCP 客户端。成功创建后，在第 108 行对 nx_dhcp_start 的调用中启动 DHCP 进程。此时，将启动 DHCP 客户端尝试以联系 DHCP 服务器。在此过程中，应用程序代码在第 95 行使用 nx_ip_status_check 服务（若为辅助接口，则使用 nx_ip_interface_status_check）等待有效的 IP 地址向该 IP 实例注册 。此操作通常在使用较短等待选项的循环中完成。

在第 127 行之后，DHCP 已接收到有效的 IP 地址，应用程序可以继续操作，根据需要使用 NetX TCP/IP 服务。

```
#include  "tx_api.h"
```

```
#include  "nx_api.h"
#include  "nx_dhcp.h"

#define   DEMO_STACK_SIZE     4096
TX_THREAD        my_thread;
NX_PACKET_POOL     my_pool;
NX_IP          my_ip;
NX_DHCP         my_dhcp;

/* Define function prototypes. */

void  my_thread_entry(ULONG thread_input);
void  my_netx_driver(struct NX_IP_DRIVER_STRUCT *driver_req);

/* Define main entry point. */

intmain()
{

  /* 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 “my_thread”. */
    tx_thread_create(&my_thread, "my thread", my_thread_entry, 0,
             pointer, DEMO_STACK_SIZE,
             2, 2, TX_NO_TIME_SLICE, TX_AUTO_START);
  pointer = pointer + DEMO_STACK_SIZE;

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

  /* Create a packet pool. */
  status = nx_packet_pool_create(&my_pool, "NetX Main Packet Pool",
                     1024, pointer, 64000);
  pointer = pointer + 64000;

  /* Check for pool creation error. */
  if (status)
    error_counter++;

  /* Create an IP instance without an IP address. */
  status = nx_ip_create(&my_ip, "My NetX IP Instance", IP_ADDRESS(0,0,0,0),
      0xFFFFFF00, &my_pool, my_netx_driver, pointer,
      DEMO_STACK_SIZE, 1);
  pointer = pointer + DEMO_STACK_SIZE;

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

  /* Enable ARP and supply ARP cache memory for my IP Instance. */
  status = nx_arp_enable(&my_ip, (void *) pointer, 1024);
  pointer = pointer + 1024;

  /* Check for ARP 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    actual_status;
NX_PACKET  *my_packet;

  /* Wait for the link to come up. */
  do
  {

    /* Get the link status. */
    status = nx_ip_status_check(&my_ip, NX_IP_LINK_ENABLED,
                            &actual_status, 100);

  } while (status != NX_SUCCESS);

  /* Create a DHCP instance. */
  status = nx_dhcp_create(&my_dhcp, &my_ip, "My DHCP");

  /* Check for DHCP create error. */
  if (status)
    error_counter++;

  /* Start DHCP. */
  nx_dhcp_start(&my_dhcp);

  /* Check for DHCP start error. */
  if (status)
    error_counter++;

  /* Wait for IP address to be resolved through DHCP. */
  nx_ip_status_check(&my_ip, NX_IP_ADDRESS_RESOLVED,
                        (ULONG *) &status, 100000);

  /* Check to see if we have a valid IP address. */
  if (status)
  {
    error_counter++;
    return;
  }
  else
  {

        /* Yes, a valid IP address is now on lease… All NetX
       services are available.
  }
}
```

图 1.1 与 NetX 配合使用的 DHCP 的示例

多服务器环境

在有多台 DHCP 服务器的网络上，DHCP 客户端会接受其收到的第一条 DHCP 服务器 Offer 消息，进入“请求”状

态，并忽略所收到的任何其他 Offer 消息。

ARP 探测

可以配置 DHCP 客户端，使其在 DHCP 服务器分配 IP 地址后发送一个或多个 ARP 探测，验证该 IP 地址是否尚未被使用。RFC 2131 建议使用 ARP 探测步骤，此步骤在有多台 DHCP 服务器的环境中尤为重要。如果主机应用程序启用 NX_DHCP_CLIENT_SEND_ARP_PROBE 选项（请参阅第二章中的“配置选项”了解其他 ARP 探测选项），则 DHCP 客户端会发送“自寻址”ARP 探测，并等待指定的时间来获得响应。如果未收到任何响应，则 DHCP 客户端会前进到“绑定”状态。如果收到响应，则 DHCP 客户端会假设该地址已被使用。它会自动向服务器发送一条 DECLINE 消息，并重新初始化客户端，再次从 INIT 状态重启 DHCP 探测。这将重启 DHCP 状态机，并且客户端会向服务器发送另一条 DISCOVER 消息。

BOOTP 协议

DHCP 客户端还支持 BOOTP 协议和 DHCP 协议。若要启用此选项并使用 BOOTP 代替 DHCP，主机应用程序必须设置 NX_DHCP_BOOTP_ENABLE 配置选项。使用 BOOTP 协议时，主机应用程序仍可请求特定的 IP 地址。不过，有别于 BOOTP 有时用来加载主机操作系统，DHCP 客户端不支持该加载。

辅助接口上的 DHCP

NetX DHCP 客户端可以在辅助接口而不是默认的主接口上运行。

若要在辅助网络接口上运行 NetX DHCP 客户端，主机应用程序必须使用 nx_dhcp_set_interface_index API 服务，将 DHCP 客户端的接口索引设置为辅助接口。该接口必须已使用 nx_ip_interface_attach 服务来连接到主网络接口。有关连接辅助接口的详细信息，请参阅《NetX 用户指南》。

下面的图 1.2 是一个示例系统，其中，主机应用程序连接到 DHCP 服务器上的辅助接口。在第 65 行，该辅助接口连接到 IP 地址为 Null 的 IP 任务。在第 104 行，创建 DHCP 客户端实例之后，通过调用 nx_dhcp_set_interface_index，将 DHCP 客户端接口索引设置为 1（即，与自身索引为 0 的主接口的偏移量）。然后，DHCP 客户端已准备就绪，可以在第 108 行启动。

```
#include  "tx_api.h"
#include  "nx_api.h"
#include  "nx_dhcp.h"

#define   DEMO_STACK_SIZE     4096
TX_THREAD        my_thread;
NX_PACKET_POOL     my_pool;
NX_IP          my_ip;
NX_DHCP         my_dhcp;

/* Define function prototypes. */

void  my_thread_entry(ULONG thread_input);
void  my_netx_driver(struct NX_IP_DRIVER_STRUCT *driver_req);

/* Define main entry point. */

intmain()
{

  /* 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 “my_thread”. */
    tx_thread_create(&my_thread, "my thread", my_thread_entry, 0,
            pointer, DEMO_STACK_SIZE,
            2, 2, TX_NO_TIME_SLICE, TX_AUTO_START);
  pointer = pointer + DEMO_STACK_SIZE;

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

  /* Create a packet pool. */
  status = nx_packet_pool_create(&my_pool, "NetX Main Packet Pool",
                    1024, pointer, 64000);
  pointer = pointer + 64000;

  /* Check for pool creation error. */
  if (status)
    error_counter++;

  /* Create an IP instance without an IP address. */
  status = nx_ip_create(&my_ip, "My NetX IP Instance", IP_ADDRESS(0,0,0,0),
       0xFFFFFF00, &my_pool, my_netx_driver, pointer, STACK_SIZE, 1);
  pointer = pointer + DEMO_STACK_SIZE;

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

  status = _nx_ip_interface_attach(&ip_0, "port_2", IP_ADDRESS(0, 0, 0,0),
                            0xFFFFFF00UL, my_netx_driver);

  /* Enable ARP and supply ARP cache memory for my IP Instance. */
  status = nx_arp_enable(&my_ip, (void *) pointer, 1024);
  pointer = pointer + 1024;

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

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


void  my_thread_entry(ULONG thread_input)
{

UINT    status;
ULONG    status;
NX_PACKET  *my_packet;

  /* Wait for the link to come up. */
  do
  {

    /* Get the link status. */
    status = nx_ip_status_check(&my_ip,NX_IP_LINK_ENABLED,& status,100);
  } while (status != NX_SUCCESS);

  /* Create a DHCP instance. */
  status = nx_dhcp_create(&my_dhcp, &my_ip, "My DHCP");
```

```
    status = nx_dhcp_create(&my_dhcp, &my_ip, "My DHCP");

    /* Check for DHCP create error. */
    if (status)
      error_counter++;

    /* Set the DHCP client interface to the secondary interface.
      status = nx_dhcp_set_interface_index(&my_dhcp, 1);


    /* Start DHCP. */
    nx_dhcp_start(&my_dhcp);

    /* Check for DHCP start error. */
    if (status)
      error_counter++;

    /* Wait for IP address to be resolved through DHCP. */
    nx_ip_status_check(&my_ip, NX_IP_ADDRESS_RESOLVED,
                          (ULONG *) &status, 100000);

    /* Check to see if we have a valid IP address. */
    if (status)
    {
      error_counter++;
      return;
    }
    else
    {

          /* Yes, a valid IP address is now on lease… All NetX
        services are available.
    }
  }
```

图 1.2 具有多宿主支持的 NetX DHCP 示例

同时在多个接口上运行的 DHCP 客户端

若要在多个接口上运行 DHCP 客户端，nx_api.h 中的 NX_MAX_PHYSICAL_INTERFACES 必须设置为连接到设备的物理接口数。默认情况下，此值为 1（例如主接口）。若要向该 IP 实例注册其他接口，请使用 nx_ip_interface_attach 服务。有关连接辅助接口的详细信息，请参阅《NetX 用户指南》。

下一步是将 nx_dhcp.h 中的 NX_DHCP_CLIENT_MAX_RECORDS 设置为预期同时运行 DHCP 的最大接口数。请注意，NX_DHCP_CLIENT_MAX_RECORDS 不必等于 NX_MAX_PHYSICAL_INTERFACES。例如，NX_MAX_PHYSICAL_INTERFACES 可以为 3，而 NX_DHCP_CLIENT_MAX_RECORDS 等于 2。使用此配置时，三个物理接口在任何时刻都只能有两个接口（并且在任何时候都可以是三个物理接口中的任意两个接口）可以运行 DHCP。DHCP 客户端记录与网络接口之间没有一对一的映射，例如，客户端记录 1 不会自动与物理接口索引 1 关联。

NX_DHCP_CLIENT_MAX_RECORDS 也可以设置为大于 NX_MAX_PHYSICAL_INTERFACES，但这样会创建未使用的客户端记录，导致使用内存的效率低下。

应用程序必须通过调用 nx_dhcp_interface_enable 来启用任何接口，然后才能在这些接口上启动 DHCP。请注意，在 nx_dhcp_create 调用中自动启用的主接口除外（可以使用下面讨论的 nx_dhcp_interface_disable 服务禁用该接口）。

在任何时候，接口都可以禁用 DHCP，或者在该接口上独立于其他运行 DHCP 的接口停止 DHCP。

如上所述，若要让特定接口启用 DHCP，可使用 nx_dhcp_interface_enable 服务，并在输入参数中指定物理接口索引。最多可以启用 NX_DHCP_CLIENT_MAX_RECORDS 个接口，唯一的限制是接口索引输入参数必须小于 NX_MAX_PHYSICAL_INTERFACES。

若要在特定接口上启动 DHCP，请使用 nx_dhcp_interface_start 服务。若要在所有已启用的接口上启动 DHCP，请使用 nx_dhcp_start 服务。（已启动 DHCP 的接口不受 nx_dhcp_start 影响。）

若要在某一接口上停止 DHCP，请使用 nx_dhcp_interface_stop 服务。DHCP 必须已在该接口上启动，否则会返回错误状态。若要在所有已启用的接口上停止 DHCP，请使用 nx_dhcp_stop 服务。DHCP 随时可以独立于其他接口停止。

大部分现有 DHCP 客户端服务都有“接口”等效项，例如，nx_dhcp_interface_release 是 nx_dhcp_release 的接口特定等效项 。如果为单个接口配置 DHCP 客户端，则它们会执行相同的操作。

请注意，非接口特定 DHCP 客户端服务通常会应用于所有接口，但并非全都如此。在后一种情况下，非接口特定服务会应用于搜索 DHCP 客户端接口记录列表时，系统找到的第一个已启用 DHCP 的接口。如需了解多个接口启用 DHCP 时非接口特定服务的执行方式，请参阅第三章中的“服务说明”。

在以下示例序列中，IP 实例有两个网络接口，并且最初在辅助接口上运行 DHCP。稍后，在主接口上启动 DHCP。然后，释放主接口上的 IP 地址，并在主接口上重启 DHCP：

```
nx_dhcp_create(&my_dhcp_client);
/* By default this enables primary interface for DHCP. */

nx_dhcp_interface_enable(&my_dhcp_client, 1);
/* Secondary interface is enabled. */

nx_dhcp_interface_start(&my_dhcp_client, 1);
/* DHCP is started on secondary interface. */

/* Some time later… */

nx_dhcp_interface_start(&my_dhcp_client, 0);
/* DHCP is started on primary interface. */

nx_dhcp_interface_release(&my_dhcp_client, 0);

/* Some time later… */

nx_dhcp_interface_start(&my_dhcp_client, 0);
/* DHCP is restarted on primary interface. */
```

如需接口特定服务的完整列表，请参阅第三章中的“服务说明”。

配置选项

nx_dhcp.h 中的用户可配置 DHCP 选项允许主机应用程序根据其特定需求微调 DHCP 客户端。这些参数的列表如下所示：

- **NX_DHCP_ENABLE_BOOTP**：定义后，此选项用于启用 BOOTP 协议以代替 DHCP。默认已禁用此选项。
- **NX_DHCP_CLIENT_RESTORE_STATE**：定义后，此选项允许 DHCP 客户端保存其当前的 DHCP 客户端许可证“状态”，包括租约剩余时间，并在 DHCP 客户端应用程序重启之间还原该状态。默认值为“已禁用”。
- **NX_DHCP_CLIENT_USER_CREATE_PACKET_POOL**：设置此选项后，DHCP 客户端不会创建自己的数据包池。主机应用程序必须使用 nx_dhcp_packet_pool_set 服务来设置 DHCP 客户端数据包池。默认值为“已禁用”。
- **NX_DHCP_CLIENT_SEND_ARP_PROBE**：定义后，此选项允许 DHCP 客户端在 IP 地址分配后发送 ARP 探测，以确认所分配的 DHCP 地址不是由另一主机所拥有。默认情况下禁用此选项。
- **NX_DHCP_ARP_PROBE_WAIT**：定义 DHCP 客户端在发送 ARP 探测后等待响应的时间长度。默认值为 1 秒 (1 * NX_IP_PERIODIC_RATE)
- **NX_DHCP_ARP_PROBE_MIN**：定义发送 ARP 探测之间的时间间隔最小偏差。此值默认为 1 秒。

- **NX_DHCP_ARP_PROBE_MAX**：定义发送 ARP 探测之间的时间间隔最大偏差。此值默认为 2 秒。
- **NX_DHCP_ARP_PROBE_NUM**：定义为了确定 DHCP 服务器所分配的 IP 地址是否已被使用而发送的 ARP 探测数。此值默认为 3 个探测。
- **NX_DHCP_RESTART_WAIT**：定义分配给 DHCP 客户端的 IP 地址已被使用时，DHCP 客户端等待重启 DHCP 的时长。此值默认为 10 秒。
- **NX_DHCP_CLIENT_MAX_RECORDS**：指定要保存到 DHCP 客户端实例的最大接口记录数。DHCP 客户端接口记录是在特定接口上运行的 DHCP 客户端的记录。默认值设置为物理接口计数 (NX_MAX_PHYSICAL_INTERFACES)。
- **NX_DHCP_CLIENT_SEND_MAX_DHCP_MESSAGE_OPTION**：定义后，此选项允许 DHCP 客户端发送最大 DHCP 消息大小选项。默认情况下禁用此选项。
- **NX_DHCP_CLIENT_ENABLE_HOST_NAME_CHECK**：定义后，此选项允许 DHCP 客户端检查 nx_dhcp_create 调用中的输入主机名是否包含无效字符或者长度是否无效。默认情况下禁用此选项。
- **NX_DHCP_THREAD_PRIORITY**：DHCP 线程的优先级。默认情况下，此值指定 DHCP 线程以优先级 3 运行。
- **NX_DHCP_THREAD_STACK_SIZE**：DHCP 线程堆栈的大小。默认情况下，大小为 2048 个字节。
- **NX_DHCP_TIME_INTERVAL**：执行 DHCP 客户端计时器到期函数的时间间隔（以秒为单位）。例如，如果应重新传输消息或者 DHCP 客户端状态已更改，此函数会更新 DHCP 进程中的所有超时。默认情况下，此值为 1 秒。
- **NX_DHCP_OPTIONS_BUFFER_SIZE**：DHCP 选项缓冲区的大小。默认情况下，此值为 312。
- **NX_DHCP_PACKET_PAYLOAD**：指定 DHCP 客户端数据包有效负载的大小（以字节为单位）。默认值为 NX_DHCP_MINIMUM_IP_DATAGRAM + 物理标头大小。有线网络中的物理标头大小通常为以太网帧大小。
- **NX_DHCP_PACKET_POOL_SIZE**：指定 DHCP 客户端数据包池的大小。默认值为 (5 *NX_DHCP_PACKET_PAYLOAD)，这将提供四个数据包的空间，外加内部数据包池开销空间。
- **NX_DHCP_MIN_RETRANS_TIMEOUT**：指定在重新传输客户端消息之前，接收 DHCP 服务器对该消息的回复的最小等待时间选项。默认值为 RFC 2131 所建议的 4 秒。
- **NX_DHCP_MAX_RETRANS_TIMEOUT**：指定在重新传输客户端消息之前，接收 DHCP 服务器对该消息的回复的最大等待时间选项。默认值为 RFC 2131 所建议的 64 秒。
- **NX_DHCP_MIN_RENEW_TIMEOUT**：指定在将 DHCP 客户端与某一 IP 地址绑定之后，接收 DHCP 服务器消息和发送续订请求的最小等待时间选项。默认值为 60 秒。但是，在默认为最小续订超时值之前，DHCP 客户端会使用来自 DHCP 服务器消息的续订和重新绑定到期时间。
- **NX_DHCP_TYPE_OF_SERVICE**：DHCP UDP 请求所需的服务类型。默认情况下，此值定义为 NX_IP_NORMAL，表示正常的 IP 数据包服务。
- **NX_DHCP_FRAGMENT_OPTION**：根据 DHCP UDP 请求启用分段。默认情况下，此值为 NX_DONT_FRAGMENT，表示禁用 DHCP UDP 分段。
- **NX_DHCP_TIME_TO_LIVE**：指定数据包在被丢弃之前可通过的路由器数目。默认值设置为 0x80。
- **NX_DHCP_QUEUE_DEPTH**：指定接收队列的最大深度数值。默认值设置为 4。

第 3 章 - Azure RTOS NetX DHCP 客户端服务的说明

2021/4/30 •

本章包含以下按字母顺序列出的所有 Azure RTOS NetX DHCP 服务的说明。

在以下 API 说明中的"返回值"部分，以粗体显示的值不受定义用于禁用 API 错误检查的 NX_DISABLE_ERROR_CHECKING 影响，而非粗体值会完全禁用。

- nx_dhcp_create：创建 DHCP 实例
- nx_dhcp_clear_broadcast_flag：清除客户端消息上的广播标记
- nx_dhcp_delete：删除 DHCP 实例
- nx_dhcp_decline：向服务器发送拒绝消息
- nx_dhcp_force_renew：发送强制续订消息
- nx_dhcp_packet_pool_set：设置 DHCP 客户端数据包池
- nx_dhcp_release：向服务器发送释放消息
- nx_dhcp_reinitialize：清除 DHCP 客户端网络参数
- nx_dhcp_request_client_ip：指定特定 IP 地址
- nx_dhcp_send_request：向服务器发送 DHCP 消息
- nx_dhcp_server_address_get：检索 DHCP 客户端的 DHCP 服务器地址
- nx_dhcp_set_interface_index：指定客户端网络接口
- nx_dhcp_start：启动 DHCP 处理
- nx_dhcp_state_change_notify：向应用程序通知 DHCP 状态更改
- nx_dhcp_stop：停止 DHCP 处理
- nx_dhcp_user_option_retrieve：检索 DHCP 选项
- nx_dhcp_user_option_convert：将四字节转换为 ULONG

接口特定 DHCP 客户端服务：

- nx_dhcp_interface_clear_broadcast_flag：清除指定接口客户端消息上的广播标记
- nx_dhcp_interface_enable：启用接口以在指定接口上运行 DHCP
- nx_dhcp_interface_disable：禁用指定接口运行 DHCP
- nx_dhcp_interface_decline：在指定接口上向服务器发送拒绝消息
- nx_dhcp_interface_force_renew：在指定接口上发送强制续订消息
- nx_dhcp_interface_reinitialize：在指定接口上清除 DHCP 客户端网络参数
- nx_dhcp_interface_release：在指定接口上向服务器发送释放消息

- nx_dhcp_interface_request_client_ip：在指定接口上指定特定 IP 地址
- nx_dhcp_interface_send_request：在指定接口上向服务器发送 DHCP 消息
- nx_dhcp_interface_server_address_get：在指定接口上获取 DHCP 服务器 IP 地址
- nx_dhcp_interface_start：在指定接口上启动 DHCP 客户端处理
- nx_dhcp_interface_stop：在指定接口上停止 DHCP 客户端处理
- nx_dhcp_interface_state_change_notify：在指定接口上的 DHCP 状态更改时设置回调函数
- nx_dhcp_interface_user_option_retrieve：在指定接口上检索指定 DHCP 选项

定义 NX_DHCP_CLIENT_RESORE_STATE 时的 DHCP 客户端服务：

- nx_dhcp_resume：恢复以前确定的 DHCP 客户端状态
- nx_dhcp_suspend：暂停处理 DHCP 客户端状态
- nx_dhcp_client_get_record：创建 DHCP 客户端状态记录
- nx_dhcp_client_restore_record：将以前保存的记录还原到 DHCP 客户端
- nx_dhcp_client_update_time_remaining：更新当前 DHCP 状态中的剩余时间

定义 NX_DHCP_CLIENT_RESORE_STATE 时的接口特定 DHCP 客户端服务：

- nx_dhcp_client_interface_get_record：在指定接口上创建 DHCP 客户端状态记录
- nx_dhcp_client_interface_restore_record：在指定接口上将以前保存的记录还原到 DHCP 客户端
- nx_dhcp_client_interface_update_time_remaining：在指定接口上更新当前 DHCP 状态中的剩余时间

nx_dhcp_create

创建 DHCP 实例

原型

```
UINT nx_dhcp_create(NX_DHCP *dhcp_ptr, NX_IP *ip_ptr, CHAR *name_ptr);
```

说明

此服务可为以前创建的 IP 实例创建 DHCP 实例。默认情况下，已启用主接口来运行 DHCP。DHCP 客户端的 NetX 实现中虽不使用名称输入，但名称输入必须遵循主机名 RFC 1035 标准。总长度不能超过 255 个字符，以点分隔的标签必须以字母开头，以字母或数字结尾，并且可以包含连字符，但不能包含其他非字母数字字符。

如果应用程序想要在通过 IP 实例注册的另一个接口（使用 nx_ip_interface_attach）上运行 DHCP，则应用程序可以调用 nx_dhcp_set_interface_index，以便只在该接口上运行 DHCP，也可以调用 nx_dhcp_interface_enable，以便同时在该接口上运行 DHCP。有关更多详细信息，请参阅这些服务的说明。

> **NOTE**
> 应用程序必须确保 DHCP 客户端数据包池的有效负载支持 RFC 2131 第 2 节所指定的最小 DHCP 消息大小（DHCP 消息数据以及 UDP、IP 和物理网络框架标头的大小为 548 字节）。

输入参数

- dhcp_ptr：指向 DHCP 控制块的指针。
- ip_ptr 指向以前所创建 IP 实例的指针。

- name_ptr：指向 DHCP 实例主机名的指针。

返回值

- NX_SUCCESS (0x00)：DHCP 创建成功
- NX_DHCP_INVALID_NAME：(0xA8) 主机名无效
- NX_DHCP_INVALID_PAYLOAD (0x9C)：有效负载太小，不适用于 DHCP 消息
- NX_PTR_ERROR (0x16)：IP 或 DHCP 指针无效

获准方式

线程、初始化

示例

```
/* Create a DHCP instance. */
status =  nx_dhcp_create(&my_dhcp, &my_ip, "My-DHCP");

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

nx_dhcp_interface_enable

启用指定接口运行 DHCP

原型

```
UINT nx_dhcp_interface_enable(NX_DHCP *dhcp_ptr, UINT interface_index);
```

说明

此服务可启用指定接口来运行 DHCP。默认情况下，已启用主接口来运行 DHCP 客户端。此时，可以通过调用 nx_dhcp_interface_start，在此接口上启动 DHCP，也可以通过调用 nx_dhcp_start 在所有已启用接口上启动 DHCP。

请注意，应用程序必须先使用 nx_ip_interface_attach 在 IP 实例中注册此接口。

此外，还必须有可用 DHCP 客户端接口"记录"，才能将此接口添加到已启用接口列表中。默认情况下，NX_DHCP_CLIENT_MAX_RECORDS 定义为 1。可将此选项设置为预计同时运行 DHCP 客户端所使用接口的最大数量。通常 NX_DHCP_CLIENT_MAX_RECORDS 数量等于 NX_MAX_PHYSICAL_INTERFACES；但如果设备的物理接口比预计运行 DHCP 客户端的接口多，则可通过将 NX_DHCP_CLIENT_MAX_RECORDS 设置为比物理接口数小的数来节省内存。物理接口与 DHCP 客户端接口记录之间不存在一对一映射。

此服务与 nx_dhcp_set_interface_index 不同，后者仅设置单个接口来运行 DHCP，而此服务只将指定接口添加到为 DHCP 所启用客户端接口的列表中。

要禁用 DHCP 的接口，应用程序可以调用 nx_dhcp_interface_disable 服务。

输入参数

- dhcp_ptr：指向 DHCP 控制块的指针。
- interface_index 启用 DHCP 所使用接口的索引

返回值

- NX_SUCCESS (0x00) 成功启用 DHCP
- NX_DHCP_NO_RECORDS_AVAILABLE (0xA7) 无可用记录，无法再为 DHCP 启用接口
- NX_DHCP_INTERFACE_ALREADY_ENABLED (0xA3) 已为 DHCP 启用接口

- NX_PTR_ERROR (0x16)：IP 或 DHCP 指针无效
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程、初始化

示例

```
/* Enable DHCP on a secondary interface. It is already enabled on the primary
   interface. NX_DHCP_CLIENT_MAX_RECORDS is set to 2. */

status =  nx_dhcp_interface_enable(&my_dhcp, 1);
/* If status is NX_SUCCESS the interface was successfully enabled. */


status = nx_dhcp_start(&my_dhcp);
/* If status is NX_SUCCESS DHCP is running on interface 0 and 1. */
```

nx_dhcp_interface_disable

禁用指定接口运行 DHCP

原型

```
UINT nx_dhcp_interface_disable(NX_DHCP *dhcp_ptr,
                               UINT interface_index);
```

说明

此服务可禁用指定接口运行 DHCP。此服务可在此接口上重新初始化 DHCP 客户端。

若要重新启动 DHCP 客户端，应用程序必须使用 nx_dhcp_interface_enable 重新启用接口，并通过调用 nx_dhcp_interface_start 重新启动 DHCP。

输入参数

- dhcp_ptr：指向 DHCP 控制块的指针。
- interface_index 禁用 DHCP 所使用接口的索引

返回值

- NX_SUCCESS (0x00)：DHCP 创建成功
- NX_DHCP_INTERFACE_NOT_ENABLED (0xA4) 没有为 DHCP 启用接口
- NX_PTR_ERROR (0x16)：IP 或 DHCP 指针无效
- NX_CALLER_ERROR (0x11) 此服务的调用方无效
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程数

示例

```
/* Disable DHCP on a secondary interface.
. */

status = nx_dhcp_interface_disable(&my_dhcp, 1);
/* If status is NX_SUCCESS the interface is successfully disabled. */
```

nx_dhcp_clear_broadcast_flag

设置 DHCP 广播标记

原型

```
UINT nx_dhcp_clear_broadcast_flag(NX_DHCP *dhcp_ptr, UINT clear_flag);
```

说明

对于为 DHCP 启用的所有接口，此服务可设置或清除 DHCP 消息标头的广播标记。对于某些 DHCP 消息(例如发现)，设置广播标记进行广播是因为客户端没有 IP 地址。

clear_flag

- NX_TRUE 清除广播标记(请求单播响应)
- NX_FALSE 设置广播标记(请求广播响应)

此服务适用于必须通过路由器访问 DHCP 服务器，而路由器拒绝转发广播消息的 DHCP 客户端。

输入参数

- dhcp_ptr 指向 DHCP 控制块的指针
- clear_flag 广播标记的设置值

返回值

- NX_SUCCESS (0x00) 已成功更新广播标志
- NX_PTR_ERROR (0x16)：IP 或 DHCP 指针无效
- NX_CALLER_ERROR (0x11) 此服务的调用方无效

获准方式

线程、初始化

示例

```
/* Send DHCP Client messages with the broadcast flag cleared (e.g. request a
    unicast response). */
status =  nx_dhcp_clear_broadcast_flag(&my_dhcp, NX_TRUE);

/* If status is NX_SUCCESS the DHCP Client broadcast flag is updated. */
```

nx_dhcp_interface_clear_broadcast_flag

设置或清除指定接口上的广播标志

原型

```
UINT nx_dhcp_interface_clear_broadcast_flag(NX_DHCP *dhcp_ptr,
                                            UINT interface_index,
                                            UINT clear_flag);
```

说明

使用此服务，DHCP 客户端主机应用程序可设置或清除通过指定接口发送给 DHCP 服务器的 DHCP 客户端消息中的广播标记。有关更多详细信息，请参阅 nx_dhcp_clear_broadcast_flag

输入参数

- dhcp_ptr 指向 DHCP 控制块的指针
- interface_index 设置广播标记所使用接口的索引
- clear_flag 广播标记的设置值

返回值

- NX_SUCCESS (0x00) 已成功更新广播标志
- NX_DHCP_INTERFACE_NOT_ENABLED (0xA4) 没有为 DHCP 启用接口
- NX_PTR_ERROR (0x16)：IP 或 DHCP 指针无效
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程、初始化

示例

```
/* Send DHCP Client messages with the broadcast flag cleared (e.g. request a
   unicast response) on a previously attached secondary interface. */

iface_index = 1;

status =  nx_dhcp_interface_clear_broadcast_flag(&my_dhcp, iface_index, NX_TRUE);

/* If status is NX_SUCCESS the DHCP Client broadcast flag is updated. */
```

nx_dhcp_delete

删除 DHCP 实例

原型

```
UINT nx_dhcp_delete(NX_DHCP *dhcp_ptr);
```

说明

此服务可删除以前创建的 DHCP 实例。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。

返回值

- NX_SUCCESS (0x00) 成功删除 DHCP。
- NX_PTR_ERROR (0x16) DHCP 指针无效。

- NX_CALLER_ERROR (0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Delete a DHCP instance. */
status =  nx_dhcp_delete(&my_dhcp);

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

nx_dhcp_ force_renew

发送强制续订消息

原型

```
UINT nx_dhcp force_renew(NX_DHCP *dhcp_ptr);
```

说明

使用此服务，主机应用程序可在为 DHCP 启用的所有接口上发送强制续订消息。DHCP 客户端必须处于"绑定"状态。此函数可将状态设置为"续订"，以使 DHCP 客户端在 T1 超时时间到期之前尝试续订。

若要在为 DHCP 启用多个接口的情况下，在特定接口上发送强制续订消息，请使用 nx_dhcp_interface_force_renew。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。

返回值

- NX_SUCCESS (0x00) 已成功发送强制续订消息。
- NX_PTR_ERROR (0x16) DHCP 指针无效。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Send a force renew message from the Client. */
status =  nx_dhcp_force_renew(&my_dhcp);

/* If status is NX_SUCCESS the DHCP client state is the RENEWING state and the
   DHCP Client thread task will begin renewing before T1 is expired. */
```

nx_dhcp_interface_force_renew

在指定接口上发送强制续订消息

原型

```
UINT nx_dhcp_interface_force_renew(NX_DHCP *dhcp_ptr,
                                   UINT interface_index);
```

说明

只要已为 DHCP 启用输入接口，主机应用程序即可使用此服务，在该接口上发送强制续订消息（请参阅 nx_dhcp_interface_enable）。指定接口上的 DHCP 客户端必须处于"绑定"状态。此函数可将状态设置为"续订"，以使 DHCP 客户端在 T1 超时时间到期之前尝试续订。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。

返回值

- NX_SUCCESS (0x00) 已成功发送强制续订消息。
- NX_DHCP_INTERFACE_NOT_ENABLED (0xA4) 没有为 DHCP 启用接口
- NX_PTR_ERROR (0x16)：IP 或 DHCP 指针无效
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程数

示例

```
/* Send a force renew message to the server on interface 1. */
status =  nx_dhcp_interface_force_renew(&my_dhcp, 1);

/* If status is NX_SUCCESS the DHCP client state is the RENEWING state and the
   DHCP Client thread task will begin renewing before T1 is expired. */
```

nx_dhcp_packet_pool_set

设置 DHCP 客户端数据包池

原型

```
UINT nx_dhcp_packet_pool_set(NX_DHCP *dhcp_ptr,
                             NX_PACKET_POOL *packet_pool_ptr);
```

说明

使用此服务，应用程序可在此服务调用中传递指向先前所创建数据包池的指针，从而创建 DHCP 客户端数据包池。若要使用此功能，主机应用程序必须定义 NX_DHCP_CLIENT_USER_CREATE_PACKET_POOL。定义后，nx_dhcp_create 服务不能创建客户端的数据包池。请注意，在创建数据包池时，建议应用程序使用 DHCP 客户端数据包池有效负载的默认值，如 nx_dhcp.h 中的 NX_DHCP_PACKET_PAYLOAD 所定义。

输入参数

- dhcp_ptr：指向 DHCP 控制块的指针。
- packet_pool_ptr 指向以前所创建数据包池的指针

返回值

- NX_SUCCESS (0x00) 已设置 DHCP 客户端数据包池
- NX_NOT_ENABLED (0x14) 未启用服务
- NX_PTR_ERROR (0x16) DHCP 指针无效
- NX_DHCP_INVALID_PAYLOAD (0x9C) 有效负载太小

获准方式

应用程序代码

示例

```
/* Create the packet pool. */
status =  nx_packet_pool_create(&dhcp_pool, "DHCP Client Packet Pool",
        NX_DHCP_PACKET_PAYLOAD, pointer, (15 * NX_DHCP_PACKET_PAYLOAD));

/* Create the DHCP Client. */
status =  nx_dhcp_create(&dhcp_0, &ip_0, "janetsdhcp1");

/* Set the DHCP Client packet pool. */
status =  nx_dhcp_packet_pool_set(&my_dhcp, packet_pool_ptr);
/* If status is NX_SUCCESS packet pool was successfully set. */
```

nx_dhcp_request_client_ip

为 DHCP 实例设置请求的 IP 地址

原型

```
UINT nx_dhcp_request_client_ip(NX_DHCP *dhcp_ptr,
                               ULONG client_ip_address,
                               UINT skip_discover_message);
```

说明

此服务可在为 DHCP 客户端记录中 DHCP 启用的第一个接口上设置 DHCP 客户端向 DHCP 服务器请求的 IP 地址。如果已设置 skip_discover_message 标记，则 DHCP 客户端会跳过发现消息，并发送请求消息。

若要针对特定接口的 DHCP 消息设置特定 IP 请求，请使用 nx_dhcp_interface_request_client_ip 服务。

输入参数

- dhcp_ptr：指向 DHCP 控制块的指针。
- client_ip_address 向 DHCP 服务器请求的 IP 地址
- skip_discover_message 如果为 true，则 DHCP 客户端发送请求消息
  如果为 false，则发送发现消息。

返回值

- NX_SUCCESS：(0x00) 已设置请求的 IP 地址。
- NX_DHCP_INTERFACE_NOT_ENABLED (0xA4) 没有为 DHCP 启用接口
- NX_PTR_ERROR (0x16)：IP 或 DHCP 指针无效
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程数

示例

```
/* Set the DHCP Client requested IP address and skip the discover message. */

status =  nx_dhcp_request_client_ip(&my_dhcp, IP(192,168,0,6), NX_TRUE);

/* If status is NX_SUCCESS requested IP address was successfully set. */
```

nx_dhcp_interface_request_client_ip

在指定接口上为 DHCP 实例设置请求的 IP 地址

原型

```
UINT nx_dhcp_interface_request_client_ip(NX_DHCP *dhcp_ptr,
                                         UINT interface_index,
                                         ULONG client_ip_address,
                                         UINT skip_discover_message);
```

说明

如果已为 DHCP 启用指定接口，则此服务可在该接口上设置 DHCP 客户端向 DHCP 服务器请求的 IP 地址（请参阅 nx_dhcp_interface_enable）。如果已设置 skip_discover_message 标记，则 DHCP 客户端会跳过发现消息，并发送请求消息。

输入参数

- dhcp_ptr：指向 DHCP 控制块的指针。
- Interface_index 请求 IP 地址所使用接口的索引
- client_ip_address 向 DHCP 服务器请求的 IP 地址
- skip_discover_message 如果为 true，则 DHCP 客户端发送请求消息；否则发送发现消息。

返回值

- NX_SUCCESS：(0x00) 已设置请求的 IP 地址。
- NX_DHCP_INTERFACE_NOT_ENABLED (0xA4) 没有为 DHCP 启用接口
- NX_PTR_ERROR (0x16)：IP 或 DHCP 指针无效
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程数

示例

```
/* Set the DHCP Client requested IP address and skip the discover message on
   interface 0. */
status =  nx_dhcp_interface_request_client_ip(&my_dhcp, 0, IP(192,168,0,6), NX_TRUE);

/* If status is NX_SUCCESS requested IP address was successfully set. */
```

nx_dhcp_reinitialize

清除 DHCP 客户端网络参数

原型

```
UINT nx_dhcp_reinitialize(NX_DHCP *dhcp_ptr);
```

说明

此服务可清除主机应用程序网络参数（IP 地址、网络地址和网络掩码），并可清除为 DHCP 启用的所有接口的 DHCP 客户端状态。此服务与 nx_dhcp_stop 和 nx_dhcp_start 结合使用，可“重新启动”DHCP 状态机：

nx_dhcp_stop(&my_dhcp);
nx_dhcp_reinitialize(&my_dhcp);
nx_dhcp_start(&my_dhcp);

若要在为 DHCP 启用多个接口的情况下，重新初始化特定接口上的 DHCP 客户端，请使用 nx_dhcp_interface_reinitialize 服务。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。

返回值

- NX_SUCCESS (0x00) 已成功重新初始化 DHCP
- NX_PTR_ERROR (0x16) DHCP 指针无效

获准方式

线程数

示例

```
/* Reinitialize the previously started DHCP client. */
status =  nx_dhcp_reinitialize(&my_dhcp);

/* If status is NX_SUCCESS the host application successfully reinitialized its
network parameters and DHCP client state. */
```

nx_dhcp_interface_reinitialize

清除指定接口的 DHCP 客户端网络参数

原型

```
UINT nx_dhcp_interface_reinitialize(NX_DHCP *dhcp_ptr,
                                    UINT interface_index);
```

说明

如果已为 DHCP 启用指定接口，则此服务可清除该接口的网络参数（IP 地址、网络地址和网络掩码）（请参阅 nx_dhcp_interface_enable）。有关更多详细信息，请参阅 nx_dhcp_reinitialize。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针
- interface_index 重新初始化所使用接口的索引。

返回值

- NX_SUCCESS (0x00) 已成功重新初始化接口
- NX_DHCP_INTERFACE_NOT_ENABLED (0xA4) 没有为 DHCP 启用接口
- NX_PTR_ERROR (0x16) DHCP 指针无效
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程数

示例

```
/* Reinitialize the previously started DHCP client on interface 1. */
status =  nx_dhcp_interface_reinitialize(&my_dhcp, 1);

/* If status is NX_SUCCESS the host application successfully reinitialized its
network parameters and DHCP client state. */
```

nx_dhcp_release

发布租用的 IP 地址

原型

```
UINT nx_dhcp_release(NX_DHCP *dhcp_ptr);
```

说明

此服务可通过向 DHCP 服务器发送释放消息，释放从该服务器获取的 IP 地址。此服务随后会重新初始化 DHCP 客户端。此服务适用于为 DHCP 启用的所有接口。

应用程序可通过调用 nx_dhcp_start 来重新启动 DHCP 客户端。

若要在特定接口上将地址释放回 DHCP 服务器，请使用 nx_dhcp_interface_release 服务

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。

返回值

- NX_SUCCESS (0x00) DHCP 释放成功。
- NX_DHCP_NOT_BOUND：(0x94) 尚未租用 IP 地址，因此无法释放。
- NX_PTR_ERROR (0x16) DHCP 指针无效。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Release the previously leased IP address. */
status =  nx_dhcp_release(&my_dhcp);

/* If status is NX_SUCCESS the previous IP lease was successfully released. */
```

nx_dhcp_interface_release

在指定接口上释放 IP 地址

原型

```
UINT nx_dhcp_interface_release(NX_DHCP *dhcp_ptr,
                               UINT interface_index);
```

说明

此服务可在指定接口上释放从 DHCP 服务器获取的 IP 地址，并重新初始化 DHCP 客户端。用户可通过调用 nx_dhcp_start 来重新启动 DHCP 客户端。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。

返回值

- NX_SUCCESS (0x00) DHCP 释放成功。
- NX_DHCP_INTERFACE_NOT_ENABLED (0xA4) 没有为 DHCP 启用接口
- NX_DHCP_NOT_BOUND：(0x94) 尚未租用 IP 地址，因此无法释放。
- NX_PTR_ERROR (0x16) DHCP 指针无效
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程数

示例

```
/* Release the previously leased IP address on interface 1. */
status =  nx_dhcp_interface_release(&my_dhcp, 1);

/* If status is NX_SUCCESS the previous IP lease was successfully released. */
```

nx_dhcp_decline

拒绝 DHCP 服务器分配的 IP 地址

原型

```
UINT nx_dhcp_decline(NX_DHCP *dhcp_ptr);
```

说明

此服务可在为 DHCP 启用的所有接口上拒绝从 DHCP 服务器租用的 IP 地址。如果已定义 NX_DHCP_CLIENT_SEND_ ARP_PROBE，则 DHCP 客户端会在检测到 IP 地址已在使用中时发送拒绝消息。有关 NetX DHCP 客户端中 ARP 探测配置的详细信息，请参阅第一章的“ARP 探测”。

如果应用程序通过其他方式发现 IP 地址正在使用中，则可使用此服务拒绝此地址。

此服务会重新初始化 DHCP 客户端，以便用户通过调用 nx_dhcp_start 来重新启动此客户端。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。

返回值

- NX_SUCCESS (0x00) 已成功发送拒绝消息
- NX_DHCP_NOT_STARTED (0x96) 未启动 DHCP 实例
- NX_PTR_ERROR (0x16) DHCP 指针无效
- NX_CALLER_ERROR (0x11) 此服务的调用方无效

获准方式

线程数

示例

```
/* Decline the IP address offered by the DHCP server. */
status =  nx_dhcp_decline(&my_dhcp);

/* If status is NX_SUCCESS the previous IP address decline message was
   successfully trasnmitted. */
```

nx_dhcp_interface_decline

在指定接口上拒绝 DHCP 服务器分配的 IP 地址

原型

```
UINT nx_dhcp_interface_decline(NX_DHCP *dhcp_ptr,
                               UINT interface_index);
```

说明

此服务可向服务器发送拒绝消息，以拒绝 DHCP 服务器分配的 IP 地址。此服务还会重新初始化 DHCP 客户端。有关更多详细信息，请参阅 nx_dhcp_decline。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。
- Interface_index 拒绝 IP 地址所使用接口的索引

返回值

- NX_SUCCESS (0x00) 已发送 DHCP 拒绝消息
- NX_DHCP_NOT_BOUND (0x94) 未绑定 DHCP 客户端
- NX_DHCP_INTERFACE_NOT_ENABLED (0xA4) 没有为 DHCP 启用接口
- NX_PTR_ERROR (0x16) DHCP 指针无效
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程数

示例

```
/* Decline the IP address offered by the DHCP server on interface 2. */
status =  nx_dhcp_interface_decline(&my_dhcp, 2);

/* If status is NX_SUCCESS the previous IP address decline message was
   successfully trasnmitted. */
```

nx_dhcp_send_request

向服务器发送 DHCP 消息

原型

```
UINT nx_dhcp_send_request(NX_DHCP *dhcp_ptr, UINT dhcp_message_type);
```

说明

此服务将指定的 DHCP 消息发送到 dhcp 服务器上为 dhcp 客户端记录中找到的 DHCP 启用的第一个接口上的 DHCP 服务器。若要发送释放或拒绝消息，应用程序必须分别使用 nx_dhcp[_interface]_release() 或 nx_dhcp_interface_decline() 服务。

除发送 INFORM_REQUEST 消息类型以外，用户必须启动 DHCP 客户端才能使用此服务。

> **NOTE**
> 此服务不适用于"驱动"DHCP 客户端状态机的主机应用程序。

输入参数

- dhcp_ptr：指向 DHCP 控制块的指针。
- dhcp_message_type 消息请求（在 nx_dhcp.h 中定义）

返回值

- NX_SUCCESS (0x00) 已发送 DHCP 消息
- NX_DHCP_NOT_STARTED (0x96) 接口索引无效
- NX_DHCP_INVALID_MESSAGE (0x9B) 要发送的消息类型无效
- NX_PTR_ERROR (0x16) 指针输入无效

获准方式

线程数

示例

```
/* Send the DHCP INFORM REQUEST message to the server. */

status =  nx_dhcp_send_request(&my_dhcp, NX_DHCP_TYPE_DHCPINFORM);
/* If status is NX_SUCCESS a DHCP message was successfully sent. */
```

nx_dhcp_interface_send_request

在特定接口上向服务器发送 DHCP 消息

原型

```
UINT nx_dhcp_interface_send_request(NX_DHCP *dhcp_ptr,
                                    UINT interface_index,
                                    UINT dhcp_message_type);
```

说明

如果已为 DHCP 启用指定接口，则此服务可在该接口上向 DHCP 服务器发送消息。若要发送释放或拒绝消息，应用程序必须分别使用 nx_dhcp[_interface]_release() 或 nx_dhcp_interface_decline() 服务。

除发送 DHCP INFORM REQUEST 消息类型以外，用户必须启动 DHCP 客户端才能使用此服务。

此服务不适用于"驱动"DHCP 客户端状态机的主机应用程序。

输入参数

- dhcp_ptr：指向 DHCP 控制块的指针。
- Interface_index 发送消息所使用接口的索引
- dhcp_message_type 消息请求（在 nx_dhcp.h 中定义）

返回值

- NX_SUCCESS (0x00) 已发送 DHCP 消息
- NX_DHCP_NOT_STARTED (0x96) 接口索引无效
- NX_DHCP_INVALID_MESSAGE (0x9B) 要发送的消息类型无效
- NX_DHCP_INTERFACE_NOT_ENABLED (0xA4) 没有为 DHCP 启用接口
- NX_PTR_ERROR (0x16) DHCP 指针无效
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程数

示例

```
/* Send the INFORM REQUEST message to the server on the primary interface. */

status =  nx_dhcp_interface_send_request(&my_dhcp, 0, NX_DHCP_TYPE_DHCPINFORM);
/* If status is NX_SUCCESS a DHCP message was successfully sent. */
```

nx_dhcp_server_address_get

获取 DHCP 客户端的 DHCP 服务器 IP 地址

原型

```
UINT nx_dhcp_server_address_get(NX_DHCP *dhcp_ptr,
                                ULONG server_address);
```

说明

此服务可在为 DHCP 客户端记录中 DHCP 启用的第一个接口上检索 DHCP 客户端的 DHCP 服务器 IP 地址。仅当 DHCP 客户端绑定到 DHCP 服务器分配的 IP 地址后，调用方才能使用此服务。主机应用程序可使用 nx_ip_status_check 服务验证是否已设置 IP 地址，也可以使用 nx_dhcp_state_change_notify，并查询 DHCP 客户端状态是否为 NX_DHCP_STATE_BOUND。有关设置状态更改回调函数的更多详细信息，请参阅 nx_dhcp_state_change_notify。

若要在为 DHCP 客户端启用多个接口的情况下，在特定接口上查找 DHCP 服务器，请使用 nx_dhcp_interface_server_address_get 服务

输入参数

- dhcp_ptr：指向 DHCP 控制块的指针。
- server_address 指向服务器 IP 地址的指针

返回值

- NX_SUCCESS (0x00) 已返回 DHCP 服务器地址
- NX_PTR_ERROR (0x16) 输入指针无效

- NX_CALLER_ERROR (0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Use the state change notify service to determine the Client transition to the
bound state and get its DHCP server IP address.
/* void dhcp_state_change(NX_DHCP *dhcp_ptr, UCHAR new_state)
{

ULONG server_address;
UINT  status;

/* Increment state changes counter. */
    state_changes++;

    if (dhcp_0.nx_dhcp_state == NX_DHCP_STATE_BOUND)
    {
            status = nx_dhcp_server_address_get(&dhcp_0, &server_address);
    }
```

nx_dhcp_interface_server_address_get

在指定接口上获取 DHCP 客户端的 DHCP 服务器 IP 地址

原型

```
UINT nx_dhcp_interface_server_address_get(NX_DHCP *dhcp_ptr,
                                          UINT interface_index,
                                          ULONG server_address);
```

说明

如果已为 DHCP 启用指定接口，则此服务可在该接口上检索 DHCP 客户端的 DHCP 服务器 IP 地址。DHCP 客户端必须处于"绑定"状态。在该接口上启动 DHCP 客户端后，主机应用程序可使用 nx_ip_status_check 服务验证是否已设置 IP 地址，也可以使用 DHCP 客户端状态更改回调函数，并查询 DHCP 客户端状态是否为 NX_DHCP_STATE_BOUND。有关设置状态更改回调函数的更多详细信息，请参阅 nx_dhcp_state_change_notify。

输入参数

- dhcp_ptr：指向 DHCP 控制块的指针。
- Interface_index 获取 IP 地址所使用接口的索引
- server_address 指向服务器 IP 地址的指针

返回值

- NX_SUCCESS (0x00) 已返回 DHCP 服务器地址
- NX_DHCP_NO_INTERFACES_ENABLED (0xA5) 没有为 DHCP 启用接口
- NX_DHCP_NOT_BOUND (0x94) 未绑定 DHCP 客户端
- NX_PTR_ERROR (0x16) DHCP 指针无效
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程数

示例

```
/* Use the state change notify service to determine the Client transition to the
bound state and get its DHCP server IP address.
/* void dhcp_state_change(NX_DHCP *dhcp_ptr, UCHAR new_state)
{

ULONG server_address;
UINT  status;

/* Increment state changes counter. */
    state_changes++;

    /* Get the DHCP server IP address on interface 1 */
    if (dhcp_0.nx_dhcp_state == NX_DHCP_STATE_BOUND)
    {
            status = nx_dhcp_interface_server_address_get(&dhcp_0, 1,
                                                          &server_address);
    }
```

nx_dhcp_set_interface_index

设置 DHCP 实例的网络接口

原型

```
UINT nx_dhcp_set_interface_index(NX_DHCP *dhcp_ptr, UINT index);
```

说明

在运行已配置单个网络接口的 DHCP 客户端时，此服务可为 DHCP 实例设置连接到 DHCP 服务器所使用的网络接口。

默认情况下，DHCP 客户端在主接口上运行。若要在辅助服务中运行 DHCP，请使用此服务设置辅助接口作为 DHCP 客户端接口。应用程序之前必须使用 nx_ip_interface_attach 服务在 IP 实例中注册指定接口。

请注意，此服务适用于仅在一个接口上运行 DHCP 客户端的应用程序。若要在多个接口上运行 DHCP，请参阅 nx_dhcp_interface_enable 了解更多详细信息。

输入参数

- dhcp_ptr：指向 DHCP 控制块的指针。
- index 设备网络接口的索引

返回值

- NX_SUCCESS (0x00) 已成功设置接口。
- NX_INVALID_INTERFACE (0x4C) 网络接口无效
- NX_DHCP_INTERFACE_ALREADY_ENABLED (0xA3) 已为 DHCP 启用接口
- NX_DHCP_NO_RECORDS_AVAILABLE (0xA7) 无可用记录，无法再启用一个
- NX_PTR_ERROR (0x16) DHCP 指针无效

获准方式

线程数

示例

```
/* Set the DHCP Client interface to the secondary interface (index 1). */
status =  nx_dhcp_set_interface_index(&my_dhcp, 1);
/* If status is NX_SUCCESS a DHCP interface was successfully set. */
```

nx_dhcp_start

启动 DHCP 处理

原型

```
UINT nx_dhcp_start(NX_DHCP *dhcp_ptr);
```

说明

此服务对为 DHCP 启用的所有接口启动 DHCP 处理。默认情况下，当应用程序调用 nx_dhcp_create 时，已为 DHCP 启用主接口。

若要验证 IP 实例何时在 DHCP 客户端接口上绑定 IP 地址，请使用 nx_ip_status_check 确认 IP 地址是否有效。

如果其他接口已在运行 DHCP，则此服务不会影响它们。

若要在启用多个接口的情况下，在特定接口上启动 DHCP，请使用 nx_dhcp_interface_start 服务。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。

返回值

- NX_SUCCESS (0x00) 成功启动 DHCP。
- NX_DHCP_ALREADY_STARTED (0x93) 已启动 DHCP。
- NX_PTR_ERROR (0x16) DHCP 指针无效。
- NX_CALLER_ERROR (0x11) 服务调用方无效。

获准方式

线程数

示例

```
/* Start the DHCP processing for this IP instance. */
status =  nx_dhcp_start(&my_dhcp);

/* If status is NX_SUCCESS the DHCP was successfully started. */
```

nx_dhcp_interface_start

在指定接口上启动 DHCP 处理

原型

```
UINT nx_dhcp_interface_start(NX_DHCP *dhcp_ptr, UINT interface_index);
```

说明

如果已为 DHCP 启用指定接口，则此服务可在该接口上启动 DHCP 处理。有关为 DHCP 启用接口的更多详细信

息，请参阅 nx_dhcp_interface_enable()。默认情况下，当应用程序调用 nx_dhcp_create 时，已为 DHCP 启用主接口。

如果没有其他接口运行 DHCP 客户端，则此服务将启动/恢复 DHCP 客户端线程，并（重新）激活 DHCP 客户端计时器。

应用程序应使用 nx_ip_status_check 验证是否已获取 IP 地址。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。
- Interface_index 启动 DHCP 客户端所使用接口的索引

返回值

- NX_SUCCESS (0x00) 成功启动 DHCP。
- NX_DHCP_ALREADY_STARTED (0x93) DHCP 实例已启动。
- NX_PTR_ERROR (0x16) DHCP 指针无效。
- NX_CALLER_ERROR (0x11) 服务调用方无效。
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程数

示例

```
/* Start the DHCP processing for this IP instance on interface 1. */
status =  nx_dhcp_interface_start(&my_dhcp, 1);

/* If status is NX_SUCCESS the DHCP was successfully started. */
```

nx_dhcp_state_change_notify

设置 DHCP 状态更改回调函数

原型

```
UINT nx_dhcp_state_change_notify(
          NX_DHCP *dhcp_ptr,
          VOID (*dhcp_state_change_notify)(NX_DHCP *dhcp_ptr,
                                           UCHAR new_state));
```

说明

此服务可注册指定回调函数 dhcp_state_change_notify，以通知应用程序 DHCP 状态更改。回调函数可提供 DHCP 客户端已转换的状态。

下面是与各种 DHCP 状态关联的值：

“	”t”
NX_DHCP_STATE_BOOT	1
NX_DHCP_STATE_INIT	2

“	“[”
NX_DHCP_STATE_SELECTING	3
NX_DHCP_STATE_REQUESTING	4
NX_DHCP_STATE_BOUND	5
NX_DHCP_STATE_RENEWING	6
NX_DHCP_STATE_REBINDING	7
NX_DHCP_STATE_FORCERENEW	8
NX_DHCP_STATE_ADDRESS_PROBING	9

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。
- dhcp_state_change_notify 状态更改回调函数指针

返回值

- **NX_SUCCESS** (0x00) 成功设置回调函数。
- NX_PTR_ERROR (0x16) DHCP 指针无效。
- NX_CALLER_ERROR (0x11) 服务调用方无效。

获准方式

线程、初始化

示例

```
/* Register the “my_state_change” function to be called on any DHCP state change,
   assuming DHCP has alreadybeen created. */
status =  nx_dhcp_state_change_notify(&my_dhcp, my_state_change);

/* If status is NX_SUCCESS the callback function was successfully
   registered. */
```

nx_dhcp_interface_state_change_notify

在指定接口上设置 DHCP 状态更改回调函数

原型

```
UINT nx_dhcp_interface_state_change_notify(
              NX_DHCP *dhcp_ptr,
              UINT interface_index,
              VOID (*dhcp_state_change_notify)(NX_DHCP *dhcp_ptr,
                                               UINT interface_index,
                                               UCHAR new_state));
```

说明

此服务可注册指定回调函数，以通知应用程序 DHCP 状态更改。回调函数输入参数是接口索引和 DHCP 客户端在该接口上已转换的状态。

有关状态更改函数的详细信息，请参阅 nx_dhcp_state_change_notify()。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。
- dhcp_interface_state_change_notify 应用程序回调函数指针

返回值

- **NX_SUCCESS** (0x00) 成功设置回调函数。
- NX_PTR_ERROR (0x16) DHCP 指针无效。

获准方式

线程、初始化

示例

```
/* Register the “my_state_change” function to be called on any DHCP state
   change, assuming DHCP has alreadybeen created. */

void dhcp_interstate_state_change(NX_DHCP *dhcp_ptr, UINT iface_index,
                                  UCHAR new_state);


status =  nx_dhcp_interstate_state_change_notify(&my_dhcp,
                                                 dhcp_interstate_state_change);

/* If status is NX_SUCCESS the callback function was successfully
   registered. */
```

nx_dhcp_stop

停止 DHCP 处理

原型

```
UINT nx_dhcp_stop(NX_DHCP *dhcp_ptr);
```

说明

此服务可在已启动 DHCP 处理的所有接口上停止 DHCP 处理。如果没有接口处理 DHCP，此服务会挂起 DHCP 客户端线程，并停用 DHCP 客户端计时器。

若要在为 DHCP 启用多个接口的情况下，在特定接口上停止 DHCP，请使用 nx_dhcp_interface_stop 服务。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。

返回值

- NX_SUCCESS (0x00) 成功停止 DHCP
- NX_DHCP_NOT_STARTED (0x96) 未启动 DHCP 实例。
- NX_PTR_ERROR (0x16) DHCP 指针无效。
- NX_CALLER_ERROR (0x11) 服务调用方无效。

获准方式

线程数

示例

```
/* Stop the DHCP processing for this IP instance. */
status =  nx_dhcp_stop(&my_dhcp);

/* If status is NX_SUCCESS the DHCP was successfully stopped. */
```

nx_dhcp_interface_stop

在指定接口上停止 DHCP 处理

原型

```
UINT nx_dhcp_interface_stop(NX_DHCP *dhcp_ptr, UINT interface_index);
```

说明

如果已启动 DHCP，则此服务可在指定接口上停止 DHCP 处理。如果没有其他接口运行 DHCP，则会挂起 DHCP 线程和计时器。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。
- Interface_index 停止 DHCP 处理所使用的接口

返回值

- NX_SUCCESS (0x00) 成功停止 DHCP
- NX_DHCP_NOT_STARTED (0x96) 未启动 DHCP。
- NX_PTR_ERROR (0x16) DHCP 指针无效。
- NX_CALLER_ERROR (0x11) 服务调用方无效。
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程数

示例

```
/* Stop DHCP processing for this IP instance on interface 1. */
status =  nx_dhcp_interface_stop(&my_dhcp, 1);

/* If status is NX_SUCCESS the DHCP was successfully stopped. */
```

nx_dhcp_user_option_retrieve

检索上次服务器响应中的 DHCP 选项

原型

```
UINT nx_dhcp_user_option_retrieve(NX_DHCP *dhcp_ptr,
            UINT request_option, UCHAR *destination_ptr,
            UINT *destination_size);
```

说明

此服务可在为 DHCP 客户端记录中 DHCP 启用的第一个接口上从选项缓冲区中检索指定 DHCP 选项。如果成功，则会将选项数据复制到指定缓冲区中。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。
- request_option RFC 所指定的 DHCP 选项。请参阅 nx_dhcp.h 中的 NX_DHCP_OPTION 选项。
- destination_ptr 指向响应字符串目标的指针。
- destination_size 指向目标大小的指针，返回值后，目标可容纳所返回字节数的内容。

返回值

- NX_SUCCESS (0x00) 成功检索选项。
- NX_DHCP_NOT_BOUND (0x94) 未绑定 DHCP 客户端。
- NX_DHCP_NO_INTERFACES_ENABLED (0xA5) 没有为 DHCP 启用接口
- NX_DHCP_DEST_TO_SMALL (0x95) 目标太小，无法容纳响应。
- NX_DHCP_PARSE_ERROR (0x97) 在服务器响应中找不到 DHCP 选项。
- NX_PTR_ERROR (0x16) 输入指针无效。
- NX_CALLER_ERROR (0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
UCHAR  dns_ip_string[4];
ULONG  size;

/* Obtain the IP address of the DNS server. */
size =    sizeof(dnx_ip_string);
status =  nx_dhcp_user_option_retrieve(&my_dhcp, NX_DHCP_OPTION_DNS_SVR,
        dns_ip_string, &size);

/* If status is NX_SUCCESS the DNS IP address is in dns_ip_string. */
```

nx_dhcp_interface_user_option_retrieve

在指定接口上从上次服务器响应中检索 DHCP 选项

原型

```
UINT nx_dhcp_interface_user_option_retrieve(NX_DHCP *dhcp_ptr,
                  UINT interface_index,
            UINT request_option, UCHAR *destination_ptr,
            UINT *destination_size);
```

说明

如果已为 DHCP 启用指定接口，则此服务可在该接口上从 DHCP 选项缓冲区中检索指定 DHCP 选项。如果成功，则会将选项数据复制到指定缓冲区中。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。

- Interface_index 检索指定选项所使用接口的索引
- request_option RFC 所指定的 DHCP 选项。请参阅 nx_dhcp.h 中的 NX_DHCP_OPTION 选项。
- destination_ptr 指向响应字符串目标的指针。
- destination_size 指向目标大小的指针，返回值后，目标可容纳所返回字节数的内容。

返回值

- NX_SUCCESS (0x00) 成功检索选项。
- NX_DHCP_NOT_BOUND (0x94) 未分配 IP 地址
- NX_DHCP_DEST_TO_SMALL (0x95) 缓冲区太小
- NX_DHCP_PARSE_ERROR (0x97) 在服务器响应中找不到 DHCP 选项。
- NX_PTR_ERROR (0x16) DHCP 指针无效。
- NX_CALLER_ERROR (0x11) 服务调用方无效。
- NX_INVALID_INTERFACE (0x4C) 网络接口无效

获准方式

线程数

示例

```
UCHAR  dns_ip_string[4];
ULONG  size;

/* Obtain the IP address of the DNS server on the prmary interface. */
size =    sizeof(dnx_ip_string);
status =  nx_dhcp_interface_user_option_retrieve(&my_dhcp, 0, NX_DHCP_OPTION_DNS_SVR,
        dns_ip_string, &size);

/* If status is NX_SUCCESS the DNS IP address is in dns_ip_string. */
```

nx_dhcp_user_option_convert

将四字节转换为 ULONG

原型

```
ULONG nx_dhcp_user_option_convert(UCHAR *option_string_ptr);
```

说明

此服务可将"option_string_ptr"指向的四个字符转换为无符号长值。在出现 IP 地址时，此服务特别有用。

输入参数

- option_string_ptr 指向以前所检索选项字符串的指针。

返回值

- Value 前四字节的值。

获准方式

线程数

示例

```
UCHAR  dns_ip_string[4];
ULONG  dns_ip;

/* Convert the first four bytes of “dns_ip_string” to an actual IP
address in “dns_ip.”  */
dns_ip=  nx_dhcp_user_option_convert(dns_ip_string);

/* If status is NX_SUCCESS the DNS IP address is in “dns_ip.”  */
```

nx_dhcp_user_option_add_callback_set

设置回调函数，以添加用户提供的选项

原型

```
ULONG nx_dhcp_user_option_add_callbcak_set(NX_DHCP *dhcp_ptr,
    UINT (*dhcp_user_option_add)(NX_DHCP *dhcp_ptr,
                                 UINT iface_index,
                                 UINT message_type,
                                 UCHAR *user_option_ptr,
                                 UINT *user_option_length));
```

说明

此服务可注册指定回调函数，以添加用户提供的选项。

如果已指定回调函数，则应用程序可通过 iface_index 和 message_type 将用户提供的选项添加到数据包中。

> **NOTE**
>
> 在用户的例程中，添加用户提供的选项时，应用程序必须遵循 DHCP 选项格式。用户选项的总大小必须小于或等于 user_option_length，并且必须按实际选项长度更新 user_option_length。如果成功添加选项，则返回 NX_TRUE，否则返回 NX_FALSE。

输入参数

- dhcp_ptr 指向以前所创建 DHCP 实例的指针。
- dhcp_user_option_add 指向用户选项添加函数的指针。

返回值

- **NX_SUCCESS** (0x00) 成功设置回调函数。
- NX_PTR_ERROR (0x16) 无效指针。

获准方式

线程数

示例

```
/* Register the “my_dhcp_user_option_add” function to be called when add DHCP
options, assuming DHCP has already been created. */

status =  nx_dhcp_user_option_add_callback_set(&my_dhcp, my_dhcp_user_option_add);

/* If status is NX_SUCCESS the callback function was successfully registered. */
```

附录 A - 关于还原状态功能的说明

2021/4/30 •

使用 Azure RTOS NetX DHDP 客户端配置选项 NX_DHCP_CLIENT_RESTORE_STATE，系统可以在系统重新启动之间的绑定状态下还原先前创建的 DHCP 客户端记录。

启用此选项后，应用程序可以挂起和恢复 DHCP 客户端线程。还有一种服务，可以用挂起和恢复线程之间的运行时间来更新 DHCP 客户端。

在重新启动之间还原 DHCP 客户端

在重新启动后且还原 DHCP 客户端之前，以前创建的 DHCP 客户端必须处于绑定状态，并且必须从 DHCP 服务器为其分配一个 IP 地址。在关机之前，DHCP 应用程序必须将当前 DHCP 客户端记录保存到非易失性内存。系统中还必须有一个独立的“计时器”，用以跟踪在此关机状态期间的运行时间。在启动时，应用程序会创建一个新的 DHCP 客户端实例，然后使用以前创建的 DHCP 客户端记录对其进行更新。通过“计时器”获取运行时间，然后将其应用到 DHCP 客户端租约的剩余时间中。请注意，这可能会导致 DHCP 客户端状态发生更改，例如，从“绑定”状态改为“续订”状态。此时，应用程序可以恢复 DHCP 客户端。

如果关机期间的运行时间将 DHCP 客户端状态置于“续订”或“重新绑定”状态，则 DHCP 客户端将自动启动请求续订或重新绑定 IP 地址租约的 DHCP 消息。如果 IP 地址已过期，DHCP 客户端将自动清除 IP 实例上的 IP 地址，并以 INIT 状态开始 DHCP 进程，请求新的 IP 地址。

这样一来，DHCP 客户端就可以在重新启动之间运行，如同未发生中断一样。

此功能的说明图示如下。假设 DHCP 客户端仅在主接口上运行。

```
/* On the power up, create an IP instance, DHCP Client, enable ICMP and UDP
   and other resources (not shown) for the DHCP Client/application
   in tx_application_define(). */

/* Define the DHCP application thread. */
void    thread_dhcp_client_entry(ULONG thread_input)
{

UINT        status;
UINT        time_elapsed = 0;
NX_DHCP_CLIENT_RECORD client_nv_record;


if (/* The application checks if there is a previously saved DHCP Client record. */)
{


  /* No previously saved Client record. Start the DHCP Client in the INIT state. */
  status =  nx_dhcp_start(&dhcp_0);

  if (status !=NX_SUCCESS)
    return;

  do
  {

    /* Wait for DHCP to assign the IP address. */
  } while (status != NX_SUCCESS);

  /* We have a valid IP address. */

  /* At some point decide we power down the system. */
```

```
  /* Save the Client state data which we will subsequently need to restore the DHCP
     Client. */
  status = nx_dhcp_client_get_record(&dhcp_0, &client_nv_record);

  /* Copy this memory to non-volatile memory (not shown). */

  /* Delete the IP and DHCP Client instances before powering down. */
  nx_dhcp_delete(&dhcp_0);

  nx_ip_delete(&ip_0);

  /* Ready to power down, having released other resources as necessary. */

}
else
{

  /* The application has determined there is a previously saved record. We will
     restore it to the current DHCP Client instance. */

  /* Get the previous Client state data from non-volatile memory. */

  /* Apply the record to the current Client instance. This will also
     update the IP instance with IP address, mask etc. */
  status = nx_dhcp_client_restore_record(&dhcp_0, &client_nv_record, time_elapsed);

     if (status != NX_SUCCESS)
      return;

     /* We are ready to resume the DHCP Client thread and use the assigned IP address. */
     status = nx_dhcp_resume(&dhcp_0);

     if (status != NX_SUCCESS)
      return;

}
```

挂起后恢复 DHCP 客户端线程

若要在不关机的情况下挂起 DHCP 客户端线程，应用程序将在置于绑定状态并具有有效 IP 地址的 DHCP 客户端上调用 nx_dhcp_suspend。当准备好恢复 DHCP 客户端时，它会首先调用 nx_dhcp_client_update_time_remaining 来更新 DHCP 地址租约中的剩余时间（从独立计时器获取运行时间）。然后，它会调用 nx_dhcp_resume 以恢复 DHCP 客户端线程。

如果运行时间将 DHCP 客户端状态置于“续订”或“重新绑定”状态，则 DHCP 客户端将自动启动请求续订或重新绑定 IP 地址租约的 DHCP 消息。如果 IP 地址已过期，DHCP 客户端将自动清除 IP 地址并以 INIT 状态开始 DHCP 进程，请求新的 IP 地址。

此功能的用法图示如下。

```
/* Create an IP instance, DHCP Client, enable ICMP and UDP
   and other resources (not shown) typically in tx_application_define(). */

/* Define the DHCP application thread. */
void    thread_dhcp_client_entry(ULONG thread_input)
{

  /* Start the DHCP Client. */
  status =  nx_dhcp_start(&dhcp_0);

  if (status !=NX_SUCCESS)
    return;

  while(1)
  {

    /* Wait for DHCP to obtain an IP address. */
  }

  /* Do tasks with the IP address e.g. send pings to another host on the
     network... */
  status =  nx_icmp_ping(…);

  if (status !=NX_SUCCESS)
          printf("Failed %d byte Ping!\n", length);

  /* At some later time, suspend the DHCP Client e.g. the device is going to low
   power mode (sleep) so we do not want any threads to wake it up. */

  nx_dhcp_suspend(&dhcp_0);

  /* During this suspended state, an independent timer is keeping track of the
     elapsed time. */


  /* At some point, we are ready to resume the DHCP Client thread. */

  /* Update the DHCP Client lease time remaining with the time elapsed. */
  status = nx_dhcp_client_update_time_remaining(&dhcp_0, time_elapsed);

  if (status != NX_SUCCESS)
       return;

  /* We now can resume the DHCP Client thread. */
  status = nx_dhcp_resume(&dhcp_0);

  if (status != NX_SUCCESS)
       return;

  /* Resume tasks e.g. ping another host. */
  status =  nx_icmp_ping(…);

}
```

下面列出从内存还原 DHCP 客户端状态、挂起和恢复 DHCP 客户端时使用的一系列服务。

nx_dhcp_client_get_record

创建当前 DHCP 客户端状态的记录

原型

```
ULONG nx_dhcp_ client_get_record(NX_DHCP *dhcp_ptr,
                                 NX_DHCP_CLIENT_RECORD *record_ptr);
```

说明

此服务将在 DHCP 客户端实例上为 DHCP 启用的第一个接口上运行的 DHCP 客户端保存到 record_ptr 指向的记录。这样，DHCP 客户端应用程序就可以在关机和重新启动后，还原其 DHCP 客户端状态。

在为 DHCP 启用了多个接口的情况下，如要将 DHCP 客户端记录保存在特定接口上，请使用 nx_dhcp_interface_client_get_record 服务。

输入参数

- dhcp_ptr：指向 DHCP 客户端的指针
- record_ptr：指向 DHCP 客户端记录的指针

返回值

- NX_SUCCESS (0x0)：创建的客户端记录
- NX_DHCP_NOT_BOUND (0x94)：客户端未处于绑定状态
- NX_DHCP_NO_INTERFACES_ENABLED (0xA5)：没有为 DHCP 启用接口
- NX_PTR_ERROR (0x16)：指针输入无效

获准方式

线程数

示例

```
NX_DHCP_CLIENT_RECORD dhcp_record;


/* Obtain a record of the current client state. */
status=  nx_dhcp_client_get_record(dhcp_ptr, &dhcp_record);

/* If status is NX_SUCCESS dhcp_record contains the current DHCP client record. */
```

nx_dhcp_interface_client_get_record

在指定的接口上创建当前 DHCP 客户端状态的记录

原型

```
ULONG nx_dhcp_interface_client_get_record(NX_DHCP *dhcp_ptr,
                                  UINT interface_index,
                                  NX_DHCP_CLIENT_RECORD *record_ptr);
```

说明

此服务会将在指定接口上运行的 DHCP 客户端保存到 record_ptr 指向的记录。这样，DHCP 客户端应用程序就可以在关机和重新启动后，还原其 DHCP 客户端状态。

输入参数

- dhcp_ptr：指向 DHCP 客户端的指针
- interface_index：要获取记录相关的索引
- record_ptr：指向 DHCP 客户端记录的指针

返回值

- NX_SUCCESS (0x0)：创建的客户端记录

- NX_DHCP_NOT_BOUND (0x94)：客户端未处于绑定状态
- NX_DHCP_BAD_INTERFACE_INDEX_ERROR (0x9A)：接口索引无效
- NX_PTR_ERROR (0x16)：DHCP 指针无效。
- NX_INVALID_INTERFACE (0x4C)：网络接口无效

获准方式

线程数

示例

```
NX_DHCP_CLIENT_RECORD dhcp_record;


/* Obtain a record of the current client state on interface 1. */
status=  nx_dhcp_interface_client_get_record(dhcp_ptr, 1, &dhcp_record);

/* If status is NX_SUCCESS dhcp_record contains the current DHCP client record. */
```

nx_dhcp_ client_restore_record

从以前保存的记录还原 DHCP 客户端

原型

```
ULONG nx_dhcp_client_restore_record(NX_DHCP *dhcp_ptr,
                                    NX_DHCP_CLIENT_RECORD
                                    *record_ptr, ULONG time_elapsed);
```

说明

此服务使应用程序能够使用 record_ptr 指向的 DHCP 客户端记录从以前的会话中还原其 DHCP 客户端。系统会将 Time_elapsed 输入值应用至 DHCP 客户端租约的剩余时间。

这要求 DHCP 客户端应用程序在关机之前创建 DHCP 客户端的记录，并将该记录保存到非易失存储器中。

如果为 DHCP 客户端启用了多个接口，则系统会将此服务应用于在 DHCP 客户端实例中找到的第一个有效接口。

输入参数

- dhcp_ptr：指向 DHCP 客户端的指针
- record_ptr：指向 DHCP 客户端记录的指针
- time_elapsed：要从输入客户端记录的剩余租用时间中减去的时间

返回值

- NX_SUCCESS (0x0)：还原的客户端记录
- NX_DHCP_NO_INTERFACES_ENABLED (0xA5)：没有运行 DHCP 的接口
- NX_PTR_ERROR (0x16)：指针输入无效

获准方式

线程数

示例

```
NX_DHCP_CLIENT_RECORD dhcp_record;
ULONG       time_elapsed;

/* Obtain time (timer ticks) elapsed from independent time keeper. */
Time_elapsed = /* to be determined by application */ 1000;


/* Obtain a record of the current client state. */
status=  nx_dhcp_client_restore_record(client_ptr, &dhcp_record, time_elapsed);

/* If status is NX_SUCCESS the current DHCP Client pointed to by dhcp_ptr
   contains the current client record updated for time elapsed during power down. */
```

nx_dhcp_interace_client_restore_record

从以前保存的记录中还原指定接口上的 DHCP 客户端

原型

```
ULONG nx_dhcp_interface_client_restore_record(NX_DHCP *dhcp_ptr,
                                              NX_DHCP_CLIENT_RECORD
                                              *record_ptr, ULONG time_elapsed);
```

说明

此服务使应用程序能够使用 record_ptr 指向的 DHCP 客户端记录来还原指定接口上的 DHCP 客户端。系统会将 Time_elapsed 输入值应用至 DHCP 客户端租约的剩余时间。

这要求 DHCP 客户端应用程序在关机之前创建 DHCP 客户端的记录，并将该记录保存到非易失存储器中。

如果为 DHCP 客户端启用了多个接口，则系统会将此服务应用于在 DHCP 客户端实例中找到的第一个有效接口。

输入参数

- dhcp_ptr：指向 DHCP 客户端的指针
- record_ptr：指向 DHCP 客户端记录的指针
- time_elapsed：要从输入客户端记录的剩余租用时间中减去的时间

返回值

- NX_SUCCESS (0x0)：还原的客户端记录
- NX_DHCP_NOT_BOUND (0x94)：客户端未绑定到 IP 地址
- NX_DHCP_BAD_INTERFACE_INDEX_ERROR (0x9A)：接口索引无效
- NX_PTR_ERROR (0x16)：DHCP 指针无效。
- NX_INVALID_INTERFACE (0x4C)：网络接口无效

获准方式

线程数

示例

```
NX_DHCP_CLIENT_RECORD dhcp_record;
ULONG       time_elapsed;

/* Obtain time (timer ticks) elapsed from independent time keeper. */
Time_elapsed = /* to be determined by application */ 1000;


/* Obtain a record of the current client state on the primary interface. */
status=  nx_dhcp_interface_client_restore_record(client_ptr, 0, &dhcp_record, time_elapsed);

/* If status is NX_SUCCESS the current DHCP Client pointed to by dhcp_ptr
   contains the current client record updated for time elapsed during power down. */
```

nx_dhcp_ client_update_time_remaining

更新 DHCP 客户端租约的剩余时间

原型

```
ULONG nx_dhcp_client_update_time_remaining(NX_DHCP *dhcp_ptr
                                           ULONG time_elapsed);
```

说明

此服务会按照 DHCP 客户端实例上为 DHCP 启用的第一个接口上的 time_elapsed 输入值，更新 DHCP 客户端 IP 地址租约的剩余时间。使用此服务之前，应用程序必须使用 nx_dhcp_suspend 挂起 DHCP 客户端线程。调用此服务后，应用程序可以通过调用 nx_dhcp_resume 来恢复 DHCP 客户端线程。

对于 DHCP 客户端应用程序而言，如果其需要在一段时间内挂起 DHCP 客户端线程并于其后更新剩余的 IP 地址租用时间，则可使用此服务。

> **NOTE**
> 此服务不应与前面所述的 nx_dhcp_client_get_record 和 nx_dhcp_client_restore_record 一起使用。本章节的前文介绍过这些服务。

输入参数

- dhcp_ptr：指向 DHCP 客户端的指针
- time_elapsed：要从 IP 地址租约剩余时间中减去的时间

返回值

- NX_SUCCESS (0x0)：客户端 IP 租约已更新
- NX_DHCP_NO_INTERFACES_ENABLED (0xA5)：没有为 DHCP 启用接口
- NX_PTR_ERROR (0x16)：指针输入无效

获准方式

线程数

示例

```
ULONG       time_elapsed;

/* Obtain time (timer ticks) elapsed from independent time keeper. */
time_elapsed = /* to be determined by application */ 1000;


/* Apply the elapsed time to the DHCP Client address lease. */
status=  nx_dhcp_client_update_time_remaining(client_ptr, time_elapsed);

/* If status is NX_SUCCESS the DHCP Client is updated for time elapsed. */
```

nx_dhcp_interface_client_update_time_remaining

更新指定接口上 DHCP 客户端租约的剩余时间

原型

```
ULONG nx_dhcp_interface_client_update_time_remaining(NX_DHCP *dhcp_ptr,
                                                     UINT interface_index,
                                                     ULONG time_elapsed);
```

说明

此服务以指定接口上的 time_elapsed 输入（如果已为 DHCP 启用该接口）来更新 DHCP 客户端 IP 地址租约的剩余时间。使用此服务之前，应用程序必须使用 nx_dhcp_suspend 挂起 DHCP 客户端线程。调用此服务后，应用程序可以通过调用 nx_dhcp_resume 来恢复 DHCP 客户端线程。请注意，挂起和恢复 DHCP 客户端线程适用于为 DHCP 启用的所有接口。

对于 DHCP 客户端应用程序而言，如果其需要在一段时间内挂起 DHCP 客户端线程并于其后更新剩余的 IP 地址租用时间，则可使用此服务。

> **NOTE**
> 此服务不应与前面所述的 nx_dhcp_client_get_record 和 nx_dhcp_client_restore_record 一起使用。本章节的前文介绍过这些服务。

输入参数

- dhcp_ptr：指向 DHCP 客户端的指针
- interface_index：要为其应用运行时间的接口索引
- time_elapsed：要从 IP 地址租约剩余时间中减去的时间

返回值

- NX_SUCCESS (0x0)：客户端 IP 租约已更新
- NX_DHCP_BAD_INTERFACE_INDEX_ERROR (0x9A)：接口索引无效
- NX_PTR_ERROR (0x16)：DHCP 指针无效。
- NX_INVALID_INTERFACE (0x4C)：网络接口无效

获准方式

线程数

示例

```
ULONG       time_elapsed;

/* Obtain time (timer ticks) elapsed from independent time keeper. */
time_elapsed = /* to be determined by application */ 1000;


/* Apply the elapsed time to the DHCP Client address lease on interface 1. */
status=  nx_dhcp_interface_client_update_time_remaining(client_ptr, 1, time_elapsed);

/* If status is NX_SUCCESS the DHCP Client is updated for time elapsed. */
```

nx_dhcp_suspend

挂起 DHCP 客户端线程

原型

```
ULONG nx_dhcp_suspend(NX_DHCP *dhcp_ptr);
```

说明

此服务将挂起当前的 DHCP 客户端线程。请注意，与 nx_dhcp_stop 不同的是，调用此服务时，系统不会更改 DHCP 客户端的状态。

此服务将挂起在为 DHCP 启用的所有接口上运行的 DHCP。

若要在 DHCP 客户端暂停时根据运行时间更新 DHCP 客户端状态，请参阅前面所述的 nx_dhcp_client_update_time_remaining。若要恢复挂起的 DHCP 客户端线程，应用程序应调用 nx_dhcp_resume。

输入参数

- dhcp_ptr：指向 DHCP 客户端的指针

返回值

- NX_SUCCESS (0x0)：客户端线程已挂起
- NX_PTR_ERROR (0x16)：指针输入无效

获准方式

线程数

示例

```
/* Pause the DHCP client thread. */
status=  nx_dhcp_suspend(client_ptr);

/* If status is NX_SUCCESS the current DHCP Client thread is paused. */
```

nx_dhcp_resume

恢复挂起的 DHCP 客户端线程

原型

```
ULONG nx_dhcp_resume(NX_DHCP *dhcp_ptr);
```

说明

此服务将恢复挂起的 DHCP 客户端线程。请注意，恢复客户端线程后，系统不会更改实际的 DHCP 客户端状态。若要将 DHCP 客户端 IP 地址租约中的剩余时间更新为调用 nx_dhcp_resume 之前的运行时间，请参阅前面所述的 nx_dhcp_client_update_time_remaining。

此服务将恢复在为 DHCP 启用的所有接口上运行的 DHCP。

输入参数

- dhcp_ptr：指向 DHCP 客户端的指针

返回值

- NX_SUCCESS (0x0)：客户端线程已恢复
- NX_PTR_ERROR (0x16)：指针输入无效

获准方式

线程数

示例

```
/* Resume the DHCP client thread. */
status=  nx_dhcp_resume(client_ptr);

/* If status is NX_SUCCESS the current DHCP Client thread is resumed. */
```

第 1 章 - Azure RTOS NetX DHCP 服务器简介

2021/4/29 •

在 NetX 中，应用程序的 IP 地址是其中一个提供给 nx_ip_create 服务调用的参数。如果应用程序可以静态方式或通过用户配置知晓其 IP 地址，则提供该 IP 地址不会造成任何问题。但是，在某些情况下，应用程序不知道或不关心其 IP 地址。在这种情况下，应该为 nx_ip_create 函数提供零 IP 地址，这样应用程序就会与其 Azure RTOS NetX DHCP 服务器建立通信，以动态请求并获取 IP 地址。

动态 IP 地址分配

用于从网络获取动态 IP 地址的基本服务是反向地址解析协议 (RARP)。此协议类似于 ARP，只不过它专门用于获取自身的 IP 地址，而不是查找其他网络节点的 MAC 地址。低级别的 RARP 消息在本地网络上广播，网络上的服务器负责作出 RARP 响应，该响应中包含动态分配的 IP 地址。

尽管 RARP 提供了动态分配 IP 地址的服务，但有几个缺点。最明显的缺陷是，RARP 仅提供 IP 地址动态分配。在大多数情况下，要让设备正确加入网络，需要更多的信息。除 IP 地址之外，大部分设备还需要网络掩码和网关 IP 地址。此外，可能还需要 DNS 服务器 IP 地址和其他网络信息。RARP 无法提供这些信息。

RARP 替代项

为克服 RARP 的缺陷，研究人员已开发出一种更为全面的 IP 地址分配机制，称为 Bootstrap 协议 (BOOTP)。此协议能够动态分配 IP 地址，还可以提供其他重要的网络信息。然而，BOOTP 有一个缺点，即专用于静态网络配置，不支持快速或自动化的地址分配，

而这正是动态主机配置协议 (DHCP) 的优势所在。DHCP 旨在扩展 BOOTP 的基本功能，实现完全自动的 IP 服务器分配和完全动态的 IP 地址分配，其中，IP 地址分配是通过将 IP 地址“租借”给客户端一段指定时间来实现。此外，还可将 DHCP 配置成以静态方式分配 IP 地址，就像 BOOTP 一样。

DHCP 消息

尽管 DHCP 极大地增强 BOOTP 的功能，但 DHCP 使用与 BOOTP 相同的消息格式，并支持与 BOOTP 相同的供应商选项。为执行其功能，DHCP 引入了 7 个新的 DHCP 专用选项，如下所示：

DISCOVER	(1)	（由 DHCP 客户端发送）
OFFER	(2)	（由 DHCP 服务器发送）
请求	(3)	（由 DHCP 客户端发送）
DECLINE	(4)	（由 DHCP 客户端发送）
ACK	(5)	（由 DHCP 服务器发送）
NACK	(6)	（由 DHCP 服务器发送）
RELEASE	(7)	（由 DHCP 客户端发送）
INFORM	(8)	（由 DHCP 客户端发送）

[[	[	[
FORCERENEW	(9)	（由 DHCP 客户端发送）

DHCP 通信

DHCP 服务器利用 UDP 协议来接收 DHCP 客户端请求并发送响应。在知晓 IP 地址之前，使用 IP 广播地址 255.255.255.255 来发送和接收包含 DHCP 信息的 UDP 消息。但是，如果客户端知道 DHCP 服务器的地址，则可能会使用单播消息发送 DHCP 消息。

DHCP 服务器状态机

DHCP 服务器是以两步骤状态机的形式实现，该状态机由 nx_dhcp_server_create 处理期间创建的内部 DHCP 线程处理。DHCP 服务器的主要状态为：1) 接收来自 DHCP 客户端的 DISCOVER 消息，2) 接收 REQUEST 消息。

相应的 DHCP 客户端状态如下所示：

- NX_DHCP_STATE_BOOT：正在使用先前的 IP 地址启动
- NX_DHCP_STATE_INIT：正在没有先前 IP 地址值的情况下启动
- NX_DHCP_STATE_SELECTING：正在等待来自任何 DHCP 服务器的响应
- NX_DHCP_STATE_REQUESTING：已识别到 DHCP 服务器，已发送 IP 地址请求
- NX_DHCP_STATE_BOUND：已建立 DHCP IP 地址租约
- NX_DHCP_STATE_RENEWING：DHCP IP 地址租约续订时间已过，已请求续订
- NX_DHCP_STATE_REBINDING：DHCP IP 地址租约重新绑定时间已过，已请求续订
- NX_DHCP_STATE_FORCERENEW：已建立 DHCP IP 地址租约，按服务器或应用程序强制续订
- NX_DHCP_STATE_FAILED：找不到服务器，或未收到来自服务器的响应

DHCP 其他参数

NetX DHCP 服务器有默认的选项参数列表，该列表在 nx_dhcp_server.h 中的可配置选项 NX_DHCP_DEFAULT_SERVER_OPTION_LIST 中设置，从而为 DHCP 客户端提供公共/关键的网络配置参数，例如，DHCP 客户端的路由器或网关地址以及 DNS 服务器。

DHCP RFC

NetX DHCP 服务器符合 RFC2132、RFC2131 以及相关 RFC。

第 2 章 - 安装和使用 Azure RTOS NetX DHCP 服务器

2021/4/30 •

本章介绍与安装、设置和使用 Azure RTOS NetX DHCP 组件相关的各种问题。

产品分发

可以从我们的公共源代码存储库获取 Azure RTOS NetX DHCP 服务器，网址为：https://github.com/azure-rtos/netx/。此软件包包含两个源文件和一个 PDF 文件，该 PDF 文件包含本文档，如下所示：

- nx_dhcp_server.h：NetX DHCP 服务器的头文件
- nx_dhcp_server.c：NetX FTP 服务器的 C 源文件
- nx_dhcp_server.pdf：NetX DHCP 服务器的 PDF 说明
- demo_netx_dhcp_server.c：NetX DHCP 服务器演示

DHCP 安装

若要使用 NetX DHCP 服务器，应将之前提到的全部分发文件复制到安装了 NetX 的同一目录中。例如，如果 NetX 安装在“\threadx\arm7\green”目录中，则应将 nx_dhcp_server.h 和 nx_dhpc_server.c 文件复制到该目录中。

使用 NetX DHCP 服务器

NetX DHCP 服务器易于使用。基本上来说，应用程序代码必须包含 nx_dhcp_server.h 和 tx_api.h 或 nx_api.h，才能使用 ThreadX 或 NetX。在包含 nx_dhcp_server.h 之后，应用程序代码即可发出本指南后文所指定的 DHCP 函数调用。在生成过程中，应用程序还必须包含 nx_dhcp_server.c。此文件必须采用与其他应用程序文件相同的方式进行编译，并且其对象窗体必须与应用程序文件一起链接。有关使用 NetX DHCP 服务器的更多详细信息，请参阅以下各节：NetX DHCP 服务器的要求和NetX DHCP 服务器的限制。

> **NOTE**
> 由于 DHCP 利用 NetX UDP 服务，因此在使用 DHCP 之前，必须通过 nx_udp_enable 调用启用 UDP。

NetX DHCP 服务器的要求

NetX DHCP 服务器要求将 UDP 套接字端口分配给众所周知的 DHCP 端口 67。若要创建 DHCP 服务器，应用程序必须创建一个数据包池，且其数据包有效负载至少为 548 字节加上 IP 标头、UDP 标头和以太网标头（总计 44 字节，4 字节对齐）。

假定服务器和客户端都使用以太网硬件地址设置：

- 硬件类型：1
- 硬件长度：6
- 跃点数：0

多个客户端会话

NetX DHCP 服务器可以处理多个客户端会话，具体方法是维护一个表并在其中记录处于活动状态的 DHCP 客户端以及客户端所处的状态（例如，DHCP 指示 INIT、BOOT、SELECTING、REQUESTING 或 RENEWING 等）。如果会话超时在收到下一条客户端消息之前过期，除非客户端绑定到某个 IP 租约，否则服务器将清除客户端会话数据

并将分配的 IP 地址返回给可用池。如果服务器从同一个客户端收到多条 DISCOVER 消息，则服务器会重置会话超时，并保留为客户端预留的 IP 地址以接受后续 REQUEST 消息。

NetX DHCP 服务器还接受单一状态客户端 DHCP 请求，例如，客户端只发送 REQUEST 消息。这假定之前已为客户端分配了来自 DHCP 服务器的 IP 租约。

设置特定于接口的网络参数服务器响应

应用程序可以使用 nx_dhcp_set_interface_network_parameters 服务为它处理 DHCP 客户端请求的每个接口设置路由器、子网掩码和 DNS 服务器参数。否则，这些参数将分别默认为服务器主接口的 IP 网关、其 DHCP 网络子网和 DHCP 服务器 IP 地址。

DHCP 服务器将这些参数包含在其发送给 DHCP 客户端的 DHCP 消息的选项数据中。

为客户端分配 IP 地址

如果客户端 DISCOVER 消息未指定所请求的 IP 地址，则 DHCP 服务器可以使用自己的池中的 IP 地址。如果服务器没有可用的 IP 地址，则会向客户端发送 NACK 消息。

只要所请求的 IP 地址可用并且可以在服务器 IP 地址数据库中找到，NetX DHCP 服务器就会在客户端 REQUEST 消息中授予该 IP 地址。应用程序将使用 nx_dhcp_create_server_ip_address_list 服务来创建服务器可分配给 DHCP 客户端的 IP 地址的列表。如果服务器没有所请求的 IP 地址或者该 IP 地址已分配给另一个主机，则会向客户端发送 NACK 消息。

当 DHCP 服务器收到客户端请求时，它将使用 DHCP 消息的"客户端 MAC 地址"字段中的客户端 MAC 地址来唯一标识该客户端。如果客户端更改其 MAC 地址或移动到另一个子网，则应向服务器发送 RELEASE 消息以将 IP 地址返回给可用池，并请求处于 INIT 状态的新 IP 地址。

有关详细信息，请参阅"小型示例系统"一节的图 1.1。保存到 DHCP 服务器实例的 IP 地址数限制为 DHCP 服务器控制块中服务器地址阵列的大小，并由可配置选项 NX_DHCP_IP_ADDRESS_MAX_LIST_SIZE 定义。

IP 地址租约时间

如果所请求的客户端租约时间小于可配置选项 NX_DHCP_DEFAULT_LEASE_TIME 中定义的服务器默认租约时间，则 DHCP 服务器还会接受该租约时间。分配给客户端的续订时间和重新绑定时间分别为租约时间的 50% 和 85%，除非租约时间为无限期 (0xFFFFFFFF)，在这种情况下，续订时间和重新绑定时间也设置为无限期。

DHCP 服务器超时

DHCP 服务器具有用户可配置的会话超时（中 NX_DHCP_CLIENT_SESSION_TIMEOUT 选项中定义），在此超时期限内，除非会话已完成，否则服务器会等待下一条 DHCP 客户端消息。当服务器从客户端收到下一条消息时，无论该消息是否与先前发送的消息相同，都会重置超时。

内部错误处理

DHCP 服务器使用 nx_dhcp_listen_for_messages 函数来接收和处理 DHCP 客户端数据包。如果数据包无效，或者 DHCP 服务器遇到内部错误，则此函数将停止处理当前的 DHCP 客户端数据包。nx_dhcp_listen_for_messages 将返回错误状态。在调用此函数以接收下一条 DHCP 客户端消息之前，DHCP 服务器线程会短暂让出对 ThreadX 计划程序的控制权。在当前版本中，对 nx_dhcp_listen_for_messages 返回的错误状态没有日志记录支持。

选项 **55**：**参数请求列表**

必须为 NetX DHCP 服务器配置一组选项，用于加载到服务器传输回客户端的 OFFER 消息和 DHCPACK 消息中的请求选项 (55) 列表。这些选项应包括客户端网络的关键网络配置数据，并且默认情况下定义为路由器 IP 地址、子网掩码和 DNS 服务器。该选项列表是一个空格分隔的列表，并在用户可配置选项 NX_DHCP_DEFAULT_SERVER_OPTION_LIST 中定义。请注意，该列表中指定的选项个数必须等于 NX_DHCP_DEFAULT_OPTION_LIST_SIZE，此选项也由用户定义。

NetX DHCP 服务器的限制

DHCP 消息

在向客户端授予 IP 地址之前，NetX DHCP 服务器不会验证该 IP 地址是否已分配给网络上的其他客户端。如果有

多个 DHCP 服务器，则确实是这样。根据 RFC 2131，客户端负责验证 IP 地址在其网络上是否唯一（例如，对该地址执行 ping 操作）。如果不是，则服务器应该会从客户端收到带有 IP 地址的 DECLINE 消息，提示其更新数据库。

NetX DHCP 服务器不会发出 FORCE_RENEW 消息。IP 地址租约由 DHCP 客户端续订。但是，DHCP 服务器会监视其数据库中所有已分配的 IP 地址的剩余时间。IP 地址租约过期后，该 IP 地址将返回给可用的 IP 地址池。因此，由客户端主动续订/重新绑定其 IP 地址租约。

只要为客户端授予（"绑定"）了 IP 地址租约（或续订了现有 IP 地址租约），就会清除会话数据。如果客户端数据包经证实为虚假，或者客户端在两次响应之间超时，则会清除会话数据。

在重启期间保存数据

NetX DHCP 服务器将客户端数据（包括 DHCP 请求参数）保存在客户端记录表中。此表不存储在非易失性内存中，因此，如果 DHCP 服务器主机必须重启，则重启期间不会保存这些信息。

NetX DHCP 服务器将 IP 地址租约数据保存在 IP 地址表中。此表不存储在非易失性内存中，因此，如果 DHCP 服务器主机必须重启，则重启期间不会保存这些信息。

中继代理

NetX DHCP 服务器的"中继代理"字段配置了全 0 IP 地址，因为它不支持网络外 DHCP 请求。

小型示例系统

下面的图 1.1 举例说明了 NetX DHCP Server 是多么易于使用。在此示例中，DHCP 包含文件 nx_dhcp_server.h 在第 5 行引入。DHCP 服务器线程堆栈大小、IP 线程堆栈大小和测试线程堆栈大小都在第 7 到第 13 行中定义。

首先，在第 57 行使用"test_thread_entry"函数创建用于停止、重启和最终删除 DHCP 服务器的可选测试线程任务。DHCP 服务器控制块"dhcp_server"在第 20 行定义为全局变量。请注意，将创建服务器数据包池，其数据包有效负载至少与标准 DHCP 消息一样大（548 字节加上 IP 标头字节数和 UDP 标头字节数）。成功创建 DHCP 服务器的 IP 实例后，应用程序将在第 96 行创建 DHCP 服务器。接下来，应用程序为服务器 IP 实例启用 UDP。在启动 DHCP 服务器之前，应用程序将使用 nx_dhcp_create_server_ip_address_list 服务在第 137 行创建可用 IP 地址列表。接下来，应用程序使用 nx_dhcp_set_interface_network_parameters 服务在第 138 行设置网络配置参数。当应用程序在第 141 行调用 nx_dhcp_server_start 时，DHCP 服务器将变为可用。测试线程任务演示了如何停止和重启 DHCP 服务器。

```
/* This is a small demo of NetX DHCP Server for the high-performance NetX TCP/IP stack. */

#include     "tx_api.h"
#include     "nx_api.h"
#include     "nx_dhcp_server.h"

#define     DEMO_TEST_STACK_SIZE       2048
#define     DEMO_SERVER_STACK_SIZE     2048
#define     SERVER_IP_ADDRESS_LIST     "192.168.2.10 192.168.2.11 192.168.2.12"
#define     PACKET_PAYLOAD             1000
#define     PACKET_POOL_SIZE           (PACKET_PAYLOAD * 10)
#define     SERVER_IP_THREAD_STACK     2048

/* Define the ThreadX and NetX object control blocks... */

TX_THREAD          test_thread;
NX_PACKET_POOL     server_pool;
NX_IP              server_ip;
NX_DHCP_SERVER     dhcp_server;

/* Define the counters used in the demo application... */

ULONG              state_changes;

/* Define thread prototypes. */
```

```
void     test_thread_entry(ULONG thread_input);
void     nx_etherDriver_mcf5485(struct NX_IP_DRIVER_STRUCT *driver_req);

/* Define main entry point. */

intmain()
{

    /* 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 the test thread. */
    status = tx_thread_create(&test_thread, "test thread", test_thread_entry, 0,
                pointer, TEST_STACK_SIZE, 1, 1, TX_NO_TIME_SLICE, TX_DONT_START);

    if (status)
    {
        printf("Error with DHCP test thread create. Status 0x%x\r\n", status);
        return;
    }

    pointer = pointer + DEMO_STACK_SIZE;

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

    /* Create the DHCP Server packet pool. */
    status = nx_packet_pool_create(&server_pool, "NetX Main Packet Pool",
                                PACKET_PAYLOAD, pointer, PACKET_POOL_SIZE);
                                pointer = pointer + PACKET_POOL_SIZE;

    /* Check for pool creation error. */
    if (status)
    {
        printf("Error with DHCP server packet pool create. Status 0x%x\r\n", status);
        return;
    }

    /* Create the DHCP Server IP instance. */
    status = nx_ip_create(&server_ip, "NetX DHCP Server IP", NX_DHCP_SERVER_IP_ADDRESS,
                        0xFFFFFF00UL, &server_pool, nx_etherDriver_mcf5485, pointer,
                        SERVER_IP_THREAD_STACK, 1);

    pointer = pointer + DEMO_IP_THREAD_STACK;

    /* Check for IP create errors. */
    if (status)
    {
        printf("Error with DHCP server IP task create. Status 0x%x\r\n", status);
        return;
    }

    /* Create the DHCP Server instance. */
    status = nx_dhcp_server_create(&dhcp_server, &server_ip, pointer,
                    DEMO_SERVER_STACK_SIZE,"DHCP Server", &server_pool);

    if (status)
```

```
    if (status)
    {
        printf("Error with DHCP server create. Status 0x%x\r\n", status);
        return;
    }

    pointer = pointer + DEMO_SERVER_STACK_SIZE;

    /* Enable ARP and supply ARP cache memory for IP Instance 0. */
    status = nx_arp_enable(&server_ip, (void *) pointer, 1024);
    pointer = pointer + 1024;

    /* Check for ARP enable errors. */
    if (status)
    {
        printf("Error with ARP enable. Status 0x%x\r\n", status);
        return;
    }

    /* Enable UDP traffic. */
    status = nx_udp_enable(&server_ip);

    /* Check for UDP enable errors. */
    if (status)
    {
        printf("Error with ICMP enable. Status 0x%x\r\n", status);
        return;
    }

    /* Enable ICMP to enable the ping utility. */
    status = nx_icmp_enable(&server_ip);

    /* Check for errors. */
    if (status)
    {
        printf("Error with ICMP enable. Status 0x%x\r\n", status);
    }

    status = nx_dhcp_create_server_ip_address_list(&dhcp_server, iface_index,
                START_IP_ADDRESS_LIST, END_IP_ADDRESS_LIST, &addresses_added);
    status = nx_dhcp_set_interface_network_parameters(&dhcp_server, iface_index,
                                        NX_DHCP_SUBNET_MASK, IP_ADDRESS(10,0,0,1),
                                        IP_ADDRESS(10,0,0,1));

    /* Start the DHCP Server. */
    status = nx_dhcp_server_start(&dhcp_server);

    tx_thread_resume(&test_thread);
}

/* Define the test thread. */
void     test_thread_entry(ULONG thread_input)
{

UINT     status;
UINT     keep_spinning;

    /* Just let the test thread be idle till we're ready to shut things down. */
    keep_spinning = 1;
    while(keep_spinning)
    {
        tx_thread_sleep(300);
    }

    printf("Stopping the server...\n");
    status = nx_dhcp_server_stop(&dhcp_server);
    if (status)
    {
        printf("Error with DHPC server stop. Status 0x%x\r\n", status);
```

```
        return;
    }

    tx_thread_sleep(500);

    printf("Starting the server...\n");
    status = nx_dhcp_server_start(&dhcp_server);
    if (status)
    {
        printf("Error with DHPC server start. Status 0x%x\r\n", status);
        return;
    }

    tx_thread_sleep(600);

    printf("Stopping the server for good...\n");
    status = nx_dhcp_server_stop(&dhcp_server);
    if (status)
    {
        printf("Error with DHPC server stop. Status 0x%x\r\n", status);
        return;
    }

    tx_thread_sleep(200);

    printf("Deleting the server...\n");
    status = nx_dhcp_server_delete(&dhcp_server);
    if (status)
    {
        printf("Error with DHCP server delete. Status 0x%x\r\n", status);
        return;
    }
}
```

图 1.1 示例 NetX DHCP 服务器应用程序

配置选项

有多个配置选项用于生成 NetX DHCP 服务器。以下列表详细介绍了每个配置选项：

- NX_DISABLE_ERROR_CHECKING：此选项用于禁用基本 DHCP 错误检查。在调试应用程序后，通常会使用此选项。
- NX_DHCP_SERVER_THREAD_PRIORITY：此选项用于指定 DHCP 服务器线程的优先级。默认情况下，此值指定 DHCP 线程以优先级 2 运行。
- NX_DHCP_TYPE_OF_SERVICE：此选项用于指定 DHCP UDP 请求所需的服务类型。默认情况下，此值定义为 NX_IP_NORMAL，表示正常的 IP 数据包服务。
- NX_DHCP_FRAGMENT_OPTION：为 DHCP UDP 请求启用分段。默认情况下，此值设置为 NX_DONT_FRAGMENT，表示禁用 UDP 分段。
- NX_DHCP_TIME_TO_LIVE：指定数据包在被丢弃之前可通过的路由器数目。默认值为 0x80。
- NX_DHCP_QUEUE_DEPTH：指定 DHCP 服务器套接字在刷新队列之前保留的数据包数。默认值为 5。
- NX_DHCP_PACKET_ALLOCATE_TIMEOUT：指定 NetX DHCP Server 等待分配数据包池中的数据包的超时（以计时器时钟周期为单位）。默认值设置为 NX_IP_PERIODIC_RATE。
- NX_DHCP_SUBNET_MASK：这是应为 DHCP 客户端配置的子网掩码。默认值设置为 0xFFFFFF00。
- NX_DHCP_FAST_PERIODIC_TIME_INTERVAL：这是 DHCP 服务器快速计时器检查会话剩余时间并处理已超时的会话的超时期限（以计时器时钟周期为单位）。
- NX_DHCP_SLOW_PERIODIC_TIME_INTERVAL：这是 DHCP 服务器慢速计时器检查 IP 地址租约剩余时间并处理已过期的租约的超时期限（以计时器时钟周期为单位）。
- NX_DHCP_CLIENT_SESSION_TIMEOUT：这是 DHCP 服务器等待接收下一条 DHCP 客户端消息的超时期限（以计时器时钟周期为单位）。

- NX_DHCP_DEFAULT_LEASE_TIME:这是分配给 DHCP 客户端的 IP 地址租约时间(以秒为单位),并用作计算分配给客户端的续订时间和重新绑定时间的基准。默认值设置为 0xFFFFFFFF(无限期)。
- NX_DHCP_IP_ADDRESS_MAX_LIST_SIZE:这是用于保存可分配给客户端的 IP 地址的 DHCP 服务器阵列的大小。默认值为 20。
- NX_DHCP_CLIENT_RECORD_TABLE_SIZE:这是用于保存客户端记录的 DHCP 服务器阵列的大小。默认值为 50。
- NX_DHCP_CLIENT_OPTIONS_MAX:这是 DHCP 客户端实例用于保存当前会话的参数请求列表中所有所请求选项的阵列的大小。默认值为 12。
- NX_DHCP_OPTIONAL_SERVER_OPTION_LIST:这是用于保存 DHCP 服务器在参数请求列表中提供给当前 DHCP 客户端的默认选项列表的缓冲区。默认值为 136。
- NX_DHCP_OPTIONAL_SERVER_OPTION_SIZE:这是用于保存 DHCP 服务器的默认选项列表的阵列的大小。默认值为 3。
- NX_DHCP_SERVER_HOSTNAME_MAX:这是用于保存服务器主机名的缓冲区的大小。默认值为 32。
- NX_DHCP_CLIENT_HOSTNAME_MAX:这是用于保存当前 DHCP 服务器客户端会话中的客户端主机名的缓冲区的大小。默认值为 32。

第 3 章 - Azure RTOS NetX DHCP 服务器服务的说明

2021/4/30 •

本章包含所有 NetX DHCP 服务器服务的说明。

在以下 API 说明的"返回值"部分中，以粗体显示的值不受定义用于禁用 API 错误检查的 NX_DISABLE_ERROR_CHECKING 影响，而不以粗体显示的值会被完全禁用。

- nx_dhcp_server_create：创建 DHCP 服务器实例
- nx_dhcp_set_interface_network_parameters：为指定接口的关键网络参数设置 DHCP 服务器选项
- nx_dhcp_create_server_ip_address_list：创建要分配给 DHCP 客户端接口的可用 IP 地址池
- nx_dhcp_clear_client_record：在服务器数据库中删除客户端记录
- nx_dhcp_server_delete：删除 DHCP 服务器实例
- nx_dhcp_server_start：启动或恢复 DHCP 服务器处理
- nx_dhcp_server_stop：停止 DHCP 服务器处理

nx_dhcp_server_create

创建 DHCP 服务器实例

原型

```
UINT nx_dhcp_server_create(NX_DHCP_SERVER *dhcp_ptr, NX_IP *ip_ptr,
                           VOID *stack_ptr, ULONG stack_size,
                           CHAR *input_address_list, CHAR *name_ptr,
                           NX_PACKET_POOL *packet_pool_ptr);
```

说明

此服务使用以前创建的 IP 实例创建 DHCP 服务器实例。

> **IMPORTANT**
> 应用程序必须确保为 IP 创建服务创建的数据包池的最小有效负载为 548 字节，其中不包括 UDP、IP 和以太网标头。

输入参数

- dhcp_ptr：指向 DHCP 服务器控制块的指针。
- ip_ptr：指向 DHCP 服务器 IP 实例的指针。
- stack_ptr：指向 DHCP 服务器堆栈位置的指针。
- stack_size：DHCP 服务器堆栈的大小
- input_address_list：指向服务器的 IP 地址列表的指针
- name_ptr：指向 DHCP 服务器名称的指针
- packet_pool_ptr：指向 DHCP 服务器数据包池的指针

返回值

- NX_SUCCESS：(0x00) 成功创建 DHCP 服务器。
- NX_DHCP_INADEQUATE_PACKET_POOL_PAYLOAD：(0XA9) 数据包有效负载太小错误
- NX_DHCP_NO_SERVER_OPTION_LIST：(0X96) 服务器选项列表为空

- NX_DHCP_PARAMETER_ERROR：(0x92) 无效的非指针输入
- NX_CALLER_ERROR：(0x11) 无效的服务调用方。
- NX_PTR_ERROR：(0x16) 无效的指针输入。

允许来自

应用程序

示例

```
/* Create a DHCP Server instance. */
status = nx_dhcp_server_create(&dhcp_server, &server_ip, pointer,
                    DEMO_SERVER_STACK_SIZE, SERVER_IP_ADDRESS_LIST,
                    "DHCP server", &server_pool);

/* If status is NX_SUCCESS a DHCP Server instance was successfully created. */
```

nx_dhcp_create_server_ip_address_list

创建 IP 地址池

原型

```
UINT nx_dhcp_create_server_ip_address_list(NX_DHCP_SERVER *dhcp_ptr,
                            UINT iface_index, ULONG start_ip_address,
                            ULONG end_ip_address, UINT *addresses_added);
```

说明

此服务为指定的 DHCP 服务器创建要分配的特定于网络接口的可用 IP 地址池。开始和结束 IP 地址必须与指定的网络接口匹配。如果 IP 地址列表不够大(在用户可配置的 NX_DHCP_IP_ADDRESS_MAX_LIST_SIZE 参数中设置)，添加的 IP 地址的实际数量可能小于总地址。

输入参数

- dhcp_ptr：指向 DHCP 服务器控制块的指针。
- iface_index：与网络接口对应的索引
- start_ip_address：第一个可用的 IP 地址
- end_ip_address：最后一个可用的 IP 地址
- addresses_added：添加到列表的 IP 地址数

返回值

- NX_SUCCESS：(0x00) 成功创建 DHCP 服务器。
- NX_DHCP_SERVER_BAD_INTERFACE_INDEX：(0XA1) 索引与地址不匹配
- NX_DHCP_INVALID_IP_ADDRESS_LIST：(0X99) 无效的地址输入
- NX_DHCP_INVALID_IP_ADDRESS：(0x9B) 不合逻辑的开始/结束地址
- NX_PTR_ERROR：(0x16) 无效的指针输入。

允许来自

应用程序

示例

```
/* Create a pool of available IP addresses to assign. */
status = nx_dhcp_create_server_ip_list (&dhcp_server, iface_index,
                START_IP_ADDRESS_LIST, END_IP_ADDRESS_LIST, &addresses_added);

/* If status is NX_SUCCESS aIP address list was successfully created.
addresses_added indicates how many IP addresses were actually added to the list. */
```

nx_dhcp_clear_client_record

从服务器数据库中删除客户端记录

原型

```
UINT nx_dhcp_clear_client_record (NX_DHCP_SERVER *dhcp_ptr,
                                  NX_DHCP_CLIENT *dhcp_client_ptr);
```

说明

此服务从服务器数据库中清除客户端记录。

输入参数

- dhcp_ptr：指向 DHCP 服务器控制块的指针。
- dhcp_client_ptr：指向要删除的 DHCP 客户端的指针

返回值

- NX_SUCCESS：(0x00) 成功创建 DHCP 服务器。
- NX_PTR_ERROR：(0x16) 无效的指针输入。
- NX_CALLER_ERROR：(0x11) 服务的非线程调用方

允许来自

应用程序

示例

```
/* Remove Client record from the server database. */
status = nx_dhcp_clear_client_record (&dhcp_server, &dhcp_client_ptr);

/* If status is NX_SUCCESS the specified Client was removed from the database. */
```

nx_dhcp_set_interface_network_parameters

设置 DHCP 选项的网络参数

原型

```
UINT nx_dhcp_set_interface_network_parameters(NX_DHCP_SERVER *dhcp_ptr,
                                    UINT iface_index, ULONG subnet_mask,
                                    ULONG default_gateway_address,
                                    ULONG dns_server_address);
```

说明

此服务为指定接口的网络关键参数设置默认值。DHCP 服务器会将这些选项包含在其提供给 DHCP 客户端的 OFFER 和 ACK 回复中。如果主机设置了运行 DHCP 服务器的接口参数，则这些参数的默认设置如下：路由器设置为 DHCP 服务器本身的主接口网关，DNS 服务器地址设置为 DHCP 服务器本身，子网掩码设置为与配置的 DHCP 服务器接口相同。

输入参数

- dhcp_ptr：指向 DHCP 服务器控制块的指针。
- iface_index：与网络接口对应的索引
- subnet_mask：客户端网络的子网掩码
- default_gateway_address：客户端的路由器 IP 地址
- dns_server_address：用于客户端网络的 DNS 服务器

返回值

- NX_SUCCESS：(0x00) 成功创建 DHCP 服务器。
- NX_DHCP_SERVER_BAD_INTERFACE_INDEX：(0XA1) 索引与地址不匹配
- NX_DHCP_INVALID_NETWORK_PARAMETERS：(0XA3) 无效的网络参数
- NX_PTR_ERROR：(0x16) 无效的指针输入。

允许来自

应用程序

示例

```
/* Set network parameters for a specific interface. */
status = nx_dhcp_set_interface_network_parameters(&dhcp_server, iface_index,
                                    NX_DHCP_SUBNET_MASK, IP_ADDRESS(10,0,0,1),
                                    IP_ADDRESS(10,0,0,1));

/* If status is NX_SUCCESS network parameters were successfully set. */
```

nx_dhcp_server_delete

删除 DHCP 服务器实例

原型

```
UINT nx_dhcp_server_delete(NX_DHCP_SERVER *dhcp_ptr);
```

说明

此服务删除先前创建的 DHCP 服务器实例。

输入参数

- dhcp_ptr：指向 DHCP 服务器实例的指针。

返回值

- NX_SUCCESS：(0x00) 成功删除 DHCP 服务器。
- NX_PTR_ERROR：(0x16) 无效的指针输入。
- NX_DHCP_PARAMETER_ERROR：(0x92) 无效的非指针输入
- NX_CALLER_ERROR：(0x11) 无效的服务调用方。

允许来自

线程数

示例

```
/* Delete a DHCP Server instance. */
status = nx_dhcp_server_delete(&dhcp_server);

/* If status is NX_SUCCESS the DHCP Server instance was successfully deleted. */
```

nx_dhcp_server_start

启动 DHCP 服务器处理

原型

```
UINT nx_dhcp_server_start(NX_DHCP_SERVER *dhcp_ptr);
```

说明

此服务启动 DHCP 服务器处理，其中包括创建服务器 UDP 套接字、绑定 DHCP 端口以及等待接收客户端 DHCP 请求。

输入参数

- dhcp_ptr：指向以前创建的 DHCP 实例的指针。

返回值

- NX_SUCCESS：(0x00) 成功启动服务器。
- NX_DHCP_ALREADY_STARTED：(0x93) DHCP 已启动。
- NX_PTR_ERROR：(0x16) 无效的指针输入。
- NX_CALLER_ERROR：(0x11) 无效的服务调用方。
- NX_DHCP_PARAMETER_ERROR：(0x92) 无效的非指针输入

允许来自

线程数

示例

```
/* Start the DHCP Server processing for this IP instance. */
status = nx_dhcp_server_start(&dhcp_server);

/* If status is NX_SUCCESS the DHCP Server was successfully started. */
```

另请参阅

nx_dhcp_create，nx_dhcp_delete，nx_dhcp_release，nx_dhcp_state_change_notify，nx_dhcp_stop，nx_dhcp_user_option_retrieve，nx_dhcp_user_option_convert

nx_dhcp_server_stop

停止 DHCP 服务器处理

原型

```
UINT nx_dhcp_server_stop(NX_DHCP_SERVER *dhcp_ptr);
```

说明

此服务停止 DHCP 服务器处理，其中包括接收 DHCP 客户端请求。

输入参数

- dhcp_ptr：指向 DHCP 服务器实例的指针。

返回值

- NX_SUCCESS：(0x00) 成功停止 DHCP。
- NX_DHCP_ALREADY_STARTED：(0x93) DHCP 已启动。
- NX_PTR_ERROR：(0x16) 无效的指针输入。
- NX_CALLER_ERROR：(0x11) 无效的服务调用方。
- NX_DHCP_PARAMETER_ERROR：(0x92) 无效的非指针输入。

允许来自

线程数

示例

```
/* Stop the DHCP Server processing for this IP instance. */
status = nx_dhcp_server_stop(&dhcp_server);

/* If status is NX_SUCCESS the DHCP Server was successfully stopped. */
```

第 1 章 - Azure RTOS NetX DNS 客户端简介

2021/4/29 •

Azure RTOS NetX DNS 提供了一个分布式数据库，其中包含域名与物理 IP 地址之间的映射。此数据库之所以被称为分布式数据库，是因为 Internet 上没有一个单独的实体包含完整的映射。维护部分映射的实体被称为“DNS 服务器”。Internet 是由许多 DNS 服务器组成的，每个服务器都包含此数据库的一个子集。DNS 服务器也会响应 DNS 客户端对域名映射信息的请求，但前提是服务器有所请求的映射信息。

NetX 的 DNS 客户端协议为应用程序提供了从一个或多个 DNS 服务器请求获取映射信息的服务。

DNS 客户端设置

为了能够正常运行，DNS 客户端包要求已经创建了 NetX IP 实例。

在创建 DNS 客户端后，应用程序必须将一个或多个 DNS 服务器添加到 DNS 客户端维护的服务器列表中。应用程序使用 nx_dns_server_add 服务来添加 DNS 服务器。

如果已启用 NX_DNS_IP_GATEWAY_SERVER 选项，并且 IP 实例网关地址不为零，则会将 IP 实例网关自动添加为主 DNS 服务器。如果 DNS 服务器信息不是静态已知的，它也可以通过 NetX 的动态主机配置协议 (DHCP) 获得。有关详细信息，请参阅 NetX DHCP 用户指南。

DNS 客户端需要一个数据包池来传输 DNS 消息。默认情况下，在调用 nx_dns_create 服务时，此数据包池由 DNS 客户端创建。使用配置选项 NX_DNS_PACKET_PAYLOAD 和 NX_DNS_PACKET_POOL_SIZE，应用程序可以分别确定此数据包池的数据包有效负载和数据包池大小（例如数据包数量）。第 2 章中的“配置选项”部分介绍了这些选项。

除了由 DNS 客户端创建自己的数据包池之外，还可以让应用程序创建数据包池，并使用 nx_dns_packet_pool_set 服务将它设置为 DNS 客户端的数据包池。为此，必须定义 NX_DNS_CLIENT_USER_CREATE_PACKET_POOL 选项。此选项还需要将之前使用 nx_packet_pool_create 创建的数据包池用作 nx_dns_packet_pool_set 的数据包池指针输入。当删除 DNS 客户端实例时，如果启用了 NX_DNS_CLIENT_USER_CREATE_PACKET_POOL，则由应用程序负责删除不再需要的 DNS 客户端数据包池。

> **NOTE**
>
> 对于选择使用 NX_DNS_CLIENT_USER_CREATE_PACKET_POOL 选项提供自己的数据包池的应用程序，数据包大小需要能够容纳 DNS 最大消息大小（512 字节）以及 UDP 头、IPv4 头和 MAC 头。

DNS 消息

DNS 有一种非常简单的机制来获取主机名和 IP 地址之间的映射。DNS 客户端为了获取映射会准备一条 DNS 查询消息，其中包含需要解析的域名或 IP 地址。然后，此消息会被发送到服务器列表中的第一个 DNS 服务器。如果服务器有这样的映射，则会使用包含所请求的映射信息的 DNS 响应消息来答复 DNS 客户端。如果服务器没有响应，那么 DNS 客户端会查询其列表上的下一个服务器，直到查询完其所有 DNS 服务器。如果所有 DNS 服务器都没有响应，则 DNS 客户端会利用重试逻辑来重新传输 DNS 消息。在重新发送 DNS 查询后，重新传输超时时间会加倍。此过程会一直持续，直到达到最大传输超时（在 nxd_dns.h 中定义为 NX_DNS_MAX_RETRANS_TIMEOUT），或直到收到来自相应服务器的成功响应。

NetX DNS 客户端可以通过调用 nx_dns_host_by_name_get 或 nx_dns_ipv4_address_by_name_get 来执行 IPv4 地址查找（类型 A）。DNS 客户端可以使用 nx_dns_host_by_address_get 来执行 IP 地址的反向查找（类型为 PTR 查询），从而获取 Web 主机名。

DNS 消息传递利用 UDP 协议来发送请求和处理响应。DNS 服务器侦听端口号 53，以确定是否有来自客户端的查询。因此，在 NetX 中，必须在之前创建的 IP 实例 (nx_ip_create) 上使用 nx_udp_enable 服务来启用 UDP 服务。

此时，DNS 客户端已就绪，可以接受来自应用程序的请求，并发出 DNS 查询。

扩展的 DNS 资源记录类型

如果启用了 NX_DNS_ENABLE_EXTENDED_RR_TYPES，那么 NetX DNS 客户端还支持以下记录类型查询：

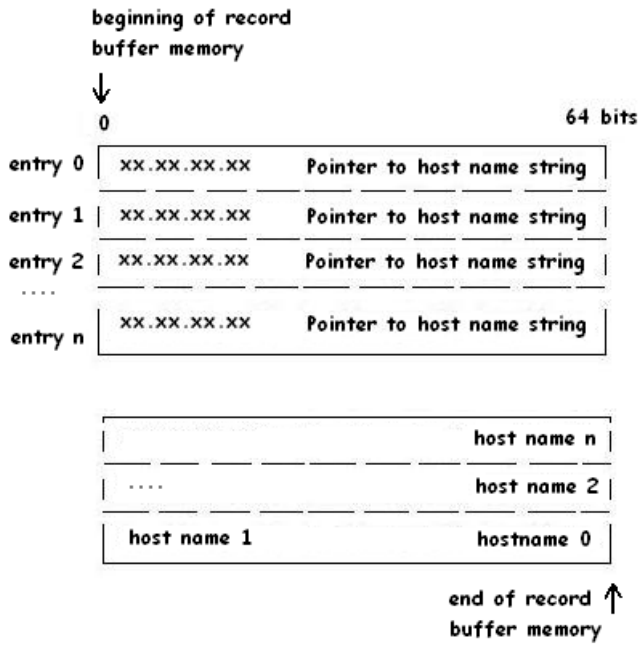
- CNAME：包含别名的规范名称
- TXT：包含文本字符串
- NS：包含授权名称服务器
- SOA：包含授权区域的起始部分
- MX：用于邮件交换
- SRV：包含域提供的服务的相关信息

应用程序必须提供 4 字节对齐的缓冲区来接收 DNS 数据记录（CNAME 和 TXT 记录类型除外）。

在 NetX DNS 客户端中，记录数据以这种方式存储是为了最高效地利用缓冲区空间。

对于记录类型的数据长度可变的查询（如主机名长度可变的 NS 记录），NetX DNS 客户端的数据保存方式如下：DNS 客户端查询中提供的缓冲区被组织成一个固定长度数据区域和一个非结构化内存区域。内存缓冲区的顶部被组织成 4 字节对齐的记录条目。每个记录条目都包含 IP 地址，以及指向此 IP 地址的可变长度数据的指针。每个 IP 地址的可变长度数据存储在从内存缓冲区末尾开始的非结构化区域内存中。每个后续记录条目的可变长度数据保存在与上一个记录条目变量数据相邻的下一个区域内存中。因此，变量数据向包含记录条目的结构化区域内存“增长”，直到没有足够的内存来存储另一个记录条目和变量数据。

如下图所示：



上面展示了 DNS 域名 (NS) 数据存储的示例。

使用记录存储格式的 NetX DNS 客户端查询返回保存到记录缓冲区中的记录数量。根据此信息，应用程序可以从记录缓冲区中提取 NS 记录。

下面展示了使用此记录存储格式来存储可变长度 DNS 数据的 DNS 客户端查询示例：

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

有关更多详情，请参阅第 3 章“DNS 客户端服务说明”。

DNS 缓存

如果启用了 NX_DNS_CACHE_ENABLE，则 NetX DNS 客户端支持 DNS 缓存功能。在创建 DNS 客户端后，应用程序可以调用 API nx_dns_cache_initialize() 来设置特殊的 DNS 缓存。如果启用了 DNS 缓存功能，那么 DNS 客户端在开始发送 DNS 查询之前会从 DNS 缓存中查找可用应答；如果找到可用应答，就会直接将应答返回给应用程序，否则 DNS 客户端会向 DNS 服务器发送查询消息并等待答复。当 DNS 客户端获得响应消息并且有可用缓存时，DNS 客户端会将应答返回给应用程序，并将应答作为资源记录添加到 DNS 缓存中。

每个应答在缓存中都有一个数据结构 NX_DNS_RR（资源记录）。记录中的字符串（资源记录名称和数据）是可变长度的，因此不会存储在 NX_DNS_RR 结构中。记录包含指向存储字符串的实际内存位置的指针。字符串表和记录共享缓存。记录从缓存的开头开始存储，然后向缓存的末尾增长。字符串表从缓存的末尾开始存储，然后向缓存的开头增长。字符串表中的每个字符串都有一个长度字段和一个计数器字段。当向字符串表添加字符串时，如果表中已有相同的字符串，则计数器值会递增，但不会为此字符串分配内存。如果无法向缓存添加更多资源记录或新字符串，则认为缓存已满。

DNS 客户端限制

DNS 客户端一次只支持一个 DNS 请求。尝试发出另一个 DNS 请求的线程会被暂时阻止，直到上一个 DNS 请求完成。

NetX DNS 客户端不使用授权应答中的数据来将额外 DNS 查询转发给其他 DNS 服务器。

DNS RFC

NetX DNS 符合以下 RFC：

- RFC1034 域名 - 概念和设施
- RFC1035 域名 - 实现和规范
- RFC1480 美国域
- RFC 2782 指定服务位置的 DNS RR (DNS SRV)

第 2 章 - Azure RTOS NetX DNS 客户端的安装和使用

2021/4/29 •

本章包含与安装、设置和使用 Azure RTOS NetX DNS 客户端相关的各种问题的说明。

产品分发

NetX DNS 客户端可从 https://github.com/azure-rtos/netx 获得。该包中包含以下文件：

- nx_dns.h：NetX DNS 客户端的头文件
- nx_dns.c：NetX DNS 客户端的 C 源文件
- nx_dns.pdf：NetX DNS 客户端的 PDF 说明

DNS 客户端安装

若要使用 NetX DNS 客户端，请将源代码文件 nx_dns.c 和 nx_dns.h 复制到 NetX 所安装在的目录中。例如，如果 NetX 安装在"\threadx\arm7\green"目录中，则应将 nx_dns.h 和 nx_dns.c 文件复制到此目录中。

使用 DNS 客户端

使用 NetX DNS 客户端非常简单。总体上，应用程序代码必须先包含 tx_api.h 和 nx_api.h，然后包含 nx_dns.h，才能分别使用 ThreadX 和 NetX。包含 nx_dns.h 之后，应用程序代码就可以进行本指南随后部分所述的 DNS 函数调用。应用程序还必须将 nx_dns.c 添加到生成过程中。此文件必须采用与其他应用程序文件相同的方式来编译，并且其对象窗体必须与该应用程序的文件一起链接。这就是使用 NetX DNS 所需的一切。

> **NOTE**
> 由于 DNS 利用了 NetX UDP 服务，因此必须先使用 nx_udp_enable 调用来启用 UDP，然后才能使用 DNS。

DNS 客户端的小型示例系统

在本部分所提供的示例 DNS 应用程序中，第 6 行包含 nx_dns.h。NX_DNS_CLIENT_USER_CREATE_PACKET_POOL（允许 DNS 客户端应用程序为 DNS 客户端创建数据包池）在第 21-23 行声明。此数据包池用于分配数据包，以便发送 DNS 消息。如果定义了 NX_DNS_CLIENT_USER_CREATE_PACKET_POOL，则会在第 71-91 行创建数据包池。如果未启用此选项，则 DNS 客户端将根据 nx_dns.h 中的配置参数所设置的数据包有效负载和池大小创建自己的数据包池，本章的其他部分将对此进行说明。

在第 93-105 行，创建用于客户端 IP 实例的另一个数据包池，以用于内部 NetX 操作。接下来，在第 107-119 行，使用 nx_ip_create 调用创建该 IP 实例。IP 任务和 DNS 客户端可以共享同一个数据包池，但 DNS 客户端所发送的消息通常比 IP 任务所发送的控制数据包要大，因此分别使用不同的数据包池有助于更高效地使用内存。

ARP 和 UDP（由 IPv4 网络使用）分别在第 122 行和第 134 行启用。

> **NOTE**
>
> 此演示使用"ram"驱动程序，该驱动程序在第 37 行声明，并在 nx_ip_create 调用中使用。该 ram 驱动程序随 NetX 源代码一起分发。若要实际运行 DNS 客户端，应用程序必须提供实际的物理网络驱动程序，以便将数据包传输到 DNS 服务器以及从 DNS 服务器接收数据包。

客户端线程入口函数 thread_client_entry 在 tx_application_define 函数下定义。最初，它会将控制权让给系统，以允许网络驱动程序初始化 IP 任务线程。

然后，在第 176-187 行创建 DNS 客户端，在第 189-200 行初始化缓存，并在第 202-217 行将先前创建的数据包池设置为 DNS 客户端实例的数据包池。接下来，在第 220-229 行添加 IPv4 DNS 服务器。

示例程序的其余部分使用 DNS 客户端服务进行 DNS 查询。第 240 行和第 262 行执行主机 IP 地址查找。nx_dns_host_by_name_get 和 nx_dns_ipv4_address_by_name_get 这两个服务之间的差别在于，前者只保存一个 IP 地址，而后者会在 DNS 服务器回复时保存多个地址。

第 354 行 (nx_dns_host_by_address_get) 执行反向查找（根据 IP 地址查找主机名）。

第 375 行和第 420 行分别演示另外两个用于 DNS 查找的服务，即 CNAME 和 TXT，它们用于发现输入域名的 CNAME 和 TXT。NetX DNS 客户端与适用于其他记录类型（例如，NS、MX、SRV 和 SOA）的类似服务用法相同。有关 NetX DNS 客户端中可用的所有记录类型查找的详细说明，请参阅第 3 章。

在第 594 行使用 nx_dns_delete 服务删除 DNS 客户端时，如果对应的数据包池由该 DNS 客户端自己创建，则会同时删除，否则会由应用程序在不再需要使用该数据包池时删除。

```
/* This is a small demo of DNS Client for the high-performance NetX TCP/IP stack.*/
#include "tx_api.h"
#include "nx_api.h"
#include "nx_udp.h"
#include "nx_dns.h"

#define     DEMO_STACK_SIZE         4096
#define     NX_PACKET_PAYLOAD       1536
#define     NX_PACKET_POOL_SIZE     30 * NX_PACKET_PAYLOAD
#define     LOCAL_CACHE_SIZE        2048

/* Define the ThreadX and NetX object control blocks... */

NX_DNS              client_dns;
TX_THREAD           client_thread;
NX_IP               client_ip;
NX_PACKET_POOL      main_pool;

#ifdef NX_DNS_CLIENT_USER_CREATE_PACKET_POOL
    NX_PACKET_POOL client_pool;
#endif

UCHAR       local_cache[LOCAL_CACHE_SIZE];
UINT        error_counter = 0;
#define     CLIENT_ADDRESS IP_ADDRESS(192,168,0,11)
#define     DNS_SERVER_ADDRESS IP_ADDRESS(192,168,0,1)

/* Define thread prototypes. */
void        thread_client_entry(ULONG thread_input);

/***** Substitute your ethernet driver entry function here *********/
extern     VOID _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr);

/* 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 the main thread. */
    tx_thread_create(&client_thread, "Client thread",
                     thread_client_entry, 0, pointer,
                     DEMO_STACK_SIZE, 4, 4, TX_NO_TIME_SLICE,
                     TX_AUTO_START);
    pointer = pointer + DEMO_STACK_SIZE;

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

#ifdef NX_DNS_CLIENT_USER_CREATE_PACKET_POOL

    /*Create the packet pool for the DNS Client to send packets. If the DNS Client is configured for letting
the host application create the DNS packet pool,
        (see NX_DNS_CLIENT_USER_CREATE_PACKET_POOL option), see 77 nx_dns_create() for guidelines on packet
payload size and pool size.
        packet traffic for NetX processes.
    */

    status = nx_packet_pool_create(&client_pool, "DNS Client Packet Pool",
                                   NX_DNS_PACKET_PAYLOAD, pointer,
                                   NX_DNS_PACKET_POOL_SIZE);

    pointer = pointer + NX_DNS_PACKET_POOL_SIZE;
    /* Check for pool creation error. */
    if (status)
    {
        error_counter++;
        return;
    }

#endif
/* Create the packet pool which the IP task will use to send packets. Also available to the host 94
application to send packet. */

    status = nx_packet_pool_create(&main_pool, "Main Packet Pool",
                                   NX_PACKET_PAYLOAD, pointer,
                                   NX_PACKET_POOL_SIZE);
    pointer = pointer + NX_PACKET_POOL_SIZE;

/* Check for pool creation error. */
    if (status)
    {
        error_counter++;
        return;
    }

/* Create an IP instance for the DNS Client. */
    status = nx_ip_create(&client_ip, "DNS Client IP Instance",
                          CLIENT_ADDRESS, 0xFFFFFF00UL,
                          &main_pool, _nx_ram_network_driver,
                          pointer, 2048, 1);
    pointer = pointer + 2048;

/* Check for IP create errors. */
    if (status)
    {
        error_counter++;
        return;
    }
/* Enable ARP and supply ARP cache memory for the DNS Client IP. */
```

```
/* Enable ARP and supply ARP cache memory for the DNS Client IP. */
    status = nx_arp_enable(&client_ip, (void *) pointer, 1024);
    pointer = pointer + 1024;

/* Check for ARP enable errors. */
    if (status)
    {
        error_counter++;
        return;
    }
/* Enable UDP traffic because DNS is a UDP based protocol. */

    status = nx_udp_enable(&client_ip);
/* Check for UDP enable errors. */
    if (status)
    {
        error_counter++;
        return;
    }

}

#define BUFFER_SIZE 200
#define RECORD_COUNT 10

/* Define the Client thread. */
void thread_client_entry(ULONG thread_input)
{
    UCHAR         record_buffer[200];
    UINT          record_count;
    UINT          status;
    ULONG         host_ip_address;
    UINT          i;
    ULONG         *ipv4_address_ptr[RECORD_COUNT];

#ifdef NX_DNS_ENABLE_EXTENDED_RR_TYPES
    NX_DNS_NS_ENTRY    *nx_dns_ns_entry_ptr[RECORD_COUNT];
    NX_DNS_MX_ENTRY    *nx_dns_mx_entry_ptr[RECORD_COUNT];
    NX_DNS_SRV_ENTRY    *nx_dns_srv_entry_ptr[RECORD_COUNT];
    NX_DNS_SOA_ENTRY    *nx_dns_soa_entry_ptr;
    ULONG                host_address;
    USHORT               host_port;
#endif

    /* Give NetX IP task a chance to get initialized . */
    tx_thread_sleep(100);
    /* Create a DNS instance for the Client. Note this function will create the DNS Client packet pool for
creating DNS message packets intended for querying its DNS server. */
    status = nx_dns_create(&client_dns, &client_ip, (UCHAR *)"DNS Client");

    /* Check for DNS create error. */
    if (status)
    {
        error_counter++;
        return;
    }

#ifdef NX_DNS_CACHE_ENABLE
    /* Initialize the cache. */
    status = nx_dns_cache_initialize(&client_dns, local_cache, LOCAL_CACHE_SIZE);

    /* Check for DNS cache error. */
    if (status)
    {
        error_counter++;
        return;
    }
#endif

/* Is the DNS client configured for the host application to create the packet pool? */
```

```
/* Is the DNS client configured for the host application to create the packet pool? */
#ifdef NX_DNS_CLIENT_USER_CREATE_PACKET_POOL

    /* Yes, use the packet pool created above which has appropriate payload size for DNS messages. */
    status = nx_dns_packet_pool_set(&client_dns, &client_pool);
    /* Check for set DNS packet pool error. */

    if (status)
    {
        error_counter++;
        return;
    }
#endif /* NX_DNS_CLIENT_USER_CREATE_PACKET_POOL */

    /* Add an IPv4 server address to the Client list. */
    status = nx_dns_server_add(&client_dns, DNS_SERVER_ADDRESS);
    /* Check for DNS add server error. */
    if (status)
    {
        error_counter++;
        return;
    }
/*****************************************************************************/
/* Type A */
/* Send A type DNS Query to its DNS server and get the IPv4 address. */
/*****************************************************************************/

    /* Look up an IPv4 address over IPv4. */
    status = nx_dns_host_by_name_get(&client_dns, (UCHAR *)"www.my_example.com", &host_ip_address, 400);

    /* Check for DNS query error. */
    if (status != NX_SUCCESS)
    {
        error_counter++;
    }
    else
    {
        printf("-----------------------------------------------------> n");
        printf("Test A: \n");
        printf("IP address: %lu.%lu.%lu.%lu\n",
        host_ip_address >> 24,
        host_ip_address >> 16 & 0xFF,
        host_ip_address >> 8 & 0xFF,
        host_ip_address & 0xFF);
    }
    /* Look up IPv4 addresses to record multiple IPv4 addresses in record_buffer and return the IPv4 address
count. */
    status = nx_dns_ipv4_address_by_name_get(&client_dns, (UCHAR *)"www.my_example.com",
                                             &record_buffer[0], BUFFER_SIZE, &record_count, 400);
    /* Check for DNS query error. */
    if (status != NX_SUCCESS)
    {
        error_counter++;
    }
    else
    {
        printf("-----------------------------------------------------> n");
        printf("Test A: ");
        printf("record_count = %d \n", record_count);
    }
    /* Get the IPv4 addresses of host. */
    for(i =0; i< record_count; i++)
    {
        ipv4_address_ptr[i] = (ULONG *)(record_buffer + i * sizeof(ULONG));
        printf("record %d: IP address: %lu.%lu.%lu.%lu\n", i,
               *ipv4_address_ptr[i] >> 24,
               *ipv4_address_ptr[i] >> 16 & 0xFF,
               *ipv4_address_ptr[i] >> 8 & 0xFF,
               *ipv4_address_ptr[i] & 0xFF);
    }
```

```
    }
/*****************************************************************************/
/* Type A + CNAME response */
/* Send A type DNS Query to its DNS server and get the IPv4 address. */
/*****************************************************************************/

    /* Look up an IPv4 address over IPv4. */
    status = nx_dns_host_by_name_get(&client_dns, (UCHAR *)"www.my_example.com", &host_ip_address, 400);

    /* Check for DNS query error. */
    if (status != NX_SUCCESS)
    {
        error_counter++;
    }
    else
    {
        printf("----------------------------------------------------> n");
        printf("Test A + CNAME response: \n");
        printf("IP address: %lu.%lu.%lu.%lu\n",
        host_ip_address >> 24,
        host_ip_address >> 16 & 0xFF,
        host_ip_address >> 8 & 0xFF,
        host_ip_address & 0xFF);
    }

    /* Look up IPv4 addresses to record multiple IPv4 addresses in record_buffer and return the IPv4 address
count. */
    status = nx_dns_ipv4_address_by_name_get(&client_dns, (UCHAR *)"www.my_example.com",
                                             &record_buffer[0], BUFFER_SIZE,
                                             &record_count, 400);

    /* Check for DNS query error. */
    if (status != NX_SUCCESS)
    {
        error_counter++;
    }
    else
    {
        printf("----------------------------------------------------> n");
        printf("Test Test A + CNAME response: ");
        printf("record_count = %d \n", record_count);
    }

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

/*****************************************************************************/
/* Type PTR */
/* Send PTR type DNS Query to its DNS server and get the host name. */
/*****************************************************************************/

    /* Look up host name over IPv4. */
    host_ip_address = IP_ADDRESS(74, 125, 71, 106);
    status = nx_dns_host_by_address_get(&client_dns, host_ip_address, &record_buffer[0], BUFFER_SIZE, 450);

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

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

#ifdef NX_DNS_ENABLE_EXTENDED_RR_TYPES
/*****************************************************************************/
/* Type CNAME */
/* Send CNAME type DNS Query to its DNS server and get the canonical name . */
/*****************************************************************************/

    /* Send CNAME type to record the canonical name of host in record_buffer. */

    status = nx_dns_cname_get(&client_dns, (UCHAR *)"www.my_example.com",
                              &record_buffer[0], BUFFER_SIZE, 400);

    /* Check for DNS query error. */
    if (status != NX_SUCCESS)
    {
        error_counter++;
    }
    else
    {
        printf("-----------------------------------------------------> n");
        printf("Test CNAME: %s\n", record_buffer);
    }
/*****************************************************************************/
/* Type TXT */
/* Send TXT type DNS Query to its DNS server and get descriptive text. */
/*****************************************************************************/

    /* Send TXT type to record the descriptive test of host in record_buffer. */
    status = nx_dns_host_text_get(&client_dns, (UCHAR *)"www.my_example.com", &record_buffer[0],
BUFFER_SIZE, 400);

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

/*****************************************************************************/
/* Type NS */
/* Send NS type DNS Query to its DNS server and get the domain name server. */
/*****************************************************************************/

    /* Send NS type to record multiple name servers in record_buffer and return the name server count. If
the DNS response includes the IPv4 addresses of name server, record it similarly in record_buffer. */

    status = nx_dns_domain_name_server_get(&client_dns, (UCHAR *)"www.my_example.com",
                                           &record_buffer[0], BUFFER_SIZE,
                                           &record_count, 400);

    /* Check for DNS query error. */
    if (status != NX_SUCCESS)
    {
        error_counter++;
    }
    else
    {
        printf("-----------------------------------------------------> n");
        printf("Test NS: ");
        printf("record_count = %d \n", record_count);
    }
```

```
    /* Get the name server. */
    for(i =0; i< record_count; i++)
    {
        nx_dns_ns_entry_ptr[i] = (NX_DNS_NS_ENTRY *)(record_buffer + i * sizeof(NX_DNS_NS_ENTRY));
        printf("record %d: IP address: %d.%d.%d.%d\n", i,
                nx_dns_ns_entry_ptr[i] -> nx_dns_ns_ipv4_address >> 24,
                nx_dns_ns_entry_ptr[i] -> nx_dns_ns_ipv4_address >> 16 & 0xFF,
                nx_dns_ns_entry_ptr[i] -> nx_dns_ns_ipv4_address >> 8 & 0xFF,
                nx_dns_ns_entry_ptr[i] -> nx_dns_ns_ipv4_address & 0xFF);

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

/*****************************************************************************/
/* Type MX */
/* Send MX type DNS Query to its DNS server and get the domain mail exchange. */
/*****************************************************************************/

    /* Send MX DNS query type to record multiple mail exchanges in record_buffer and return the mail
exchange count. If the DNS response includes the IPv4 addresses of mail exchange, record it similarly in
record_buffer. */

    status = nx_dns_domain_mail_exchange_get(&client_dns, (UCHAR *)"www.my_example.com",
                                             &record_buffer[0], BUFFER_SIZE, &record_count, 400);

    /* Check for DNS query error. */
    if (status != NX_SUCCESS)
    {
        error_counter++;
    }
    else
    {
        printf("-----------------------------------------------------> n");
        printf("Test MX: ");
        printf("record_count = %d \n", record_count);
    }
    /* Get the mail exchange. */
    for(i =0; i< record_count; i++)
    {
        nx_dns_mx_entry_ptr[i] = (NX_DNS_MX_ENTRY *)(record_buffer + i * sizeof(NX_DNS_MX_ENTRY));
        printf("record %d: IP address: %d.%d.%d.%d\n", i,
                nx_dns_mx_entry_ptr[i] -> nx_dns_mx_ipv4_address >> 24,
                nx_dns_mx_entry_ptr[i] -> nx_dns_mx_ipv4_address >> 16 & 0xFF,
                nx_dns_mx_entry_ptr[i] -> nx_dns_mx_ipv4_address >> 8 & 0xFF,
                nx_dns_mx_entry_ptr[i] -> nx_dns_mx_ipv4_address & 0xFF);

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

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

/*****************************************************************************/
/* Type SRV */
/* Send SRV type DNS Query to its DNS server and get the location of services. */
/*****************************************************************************/

    /* Send SRV DNS query type to record the location of services in record_buffer and return count. If the
DNS response includes the IPv4 addresses of service name, record it similarly in record_buffer. */

    status = nx_dns_domain_service_get(&client_dns, (UCHAR *)"www.my_example.com",
                                       &record_buffer[0], BUFFER_SIZE, &record_count, 400);

    /* Check for DNS query error. */
```

```
    if (status != NX_SUCCESS)
    {
        error_counter++;
    }
    else
    {
        printf("----------------------------------------------------> n");
        printf("Test SRV: ");
        printf("record_count = %d \n", record_count);
    }

    /* Get the location of services. */
    for(i =0; i< record_count; i++)
    {
        nx_dns_srv_entry_ptr[i] = (NX_DNS_SRV_ENTRY *)(record_buffer + i * sizeof(NX_DNS_SRV_ENTRY));
        printf("record %d: IP address: %d.%d.%d.%d\n", i,
                nx_dns_srv_entry_ptr[i] -> nx_dns_srv_ipv4_address >> 24,
                nx_dns_srv_entry_ptr[i] -> nx_dns_srv_ipv4_address >> 16 & 0xFF,
                nx_dns_srv_entry_ptr[i] -> nx_dns_srv_ipv4_address >> 8 & 0xFF,
                nx_dns_srv_entry_ptr[i] -> nx_dns_srv_ipv4_address & 0xFF);

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

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

    /* Get the service info, NetX old API.*/
    status = nx_dns_info_by_name_get(&client_dns, (UCHAR *)"www.my_example.com",
                                     &host_address, &host_port, 200);
    /* Check for DNS add server error. */
    if (status != NX_SUCCESS)
    {
        error_counter++;
    }
    else
    {
        printf("----------------------------------------------------> n");
        printf("Test SRV: ");
        printf("IP address: %d.%d.%d.%d\n",
                host_address >> 24,
                host_address >> 16 & 0xFF,
                host_address >> 8 & 0xFF,
                host_address & 0xFF);
        printf("port number = %d\n", host_port);
    }

/*****************************************************************************/
/* Type SOA */
/* Send SOA type DNS Query to its DNS server and get zone of start of authority.*/
/*****************************************************************************/

    /* Send SOA DNS query type to record the zone of start of authority in record_buffer. */

    status = nx_dns_authority_zone_start_get(&client_dns, (UCHAR *)"www.my_example.com",
                                             &record_buffer[0], BUFFER_SIZE, 400);

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

    /* Get the loc*/
    nx_dns_soa_entry_ptr = (NX_DNS_SOA_ENTRY *) record_buffer;
```

```
    printf("-------------------------------------------------------> n");
    printf("Test SOA: \n");
    printf("serial = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_serial );
    printf("refresh = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_refresh );
    printf("retry = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_retry );
    printf("expire = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_expire );
    printf("minmum = %d\n", nx_dns_soa_entry_ptr -> nx_dns_soa_minmum );
    if(nx_dns_soa_entry_ptr -> nx_dns_soa_host_mname_ptr)
        printf("host mname = %s\n", nx_dns_soa_entry_ptr -> nx_dns_soa_host_mname_ptr);
    else
        printf("host mame is not set\n");
    if(nx_dns_soa_entry_ptr -> nx_dns_soa_host_rname_ptr)
        printf("host rname = %s\n", nx_dns_soa_entry_ptr -> nx_dns_soa_host_rname_ptr);
    else
        printf("host rname is not set\n");
    #endif

    /* Shutting down...*/
    /* Terminate the DNS Client thread. */
    status = nx_dns_delete(&client_dns);
    return;
}
```

配置选项

可通过几个配置选项生成适用于 NetX 的 DNS。这些选项可以在 nx_dns.h 中重新定义。以下列表对每个配置选项进行详细说明：

- NX_DNS_TYPE_OF_SERVICE：DNS UDP 请求所需的服务类型。默认情况下，此值定义为 NX_IP_NORMAL，代表普通 IP 数据包服务。
- NX_DNS_TIME_TO_LIVE：指定数据包在丢弃之前可以经过的最大路由器数。默认值为 0x80
- NX_DNS_FRAGMENT_OPTION：设置套接字属性，以允许或禁止将传出数据包分段。默认值为 NX_DONT_FRAGMENT。
- NX_DNS_QUEUE_DEPTH：设置要存储在套接字接收队列上的数据包的最大数目。默认值为 5。
- NX_DNS_MAX_SERVERS：指定客户端服务器列表中的最大 DNS 服务器数目。
- NX_DNS_MESSAGE_MAX：用于发送 DNS 查询的最大 DNS 消息大小。默认值为 512，这也是 RFC 1035 的 2.3.4 部分中指定的最大大小。
- NX_DNS_PACKET_PAYLOAD_UNALIGNED：如果未定义，则为客户端数据包有效负载的大小，其中包括以太网、IP（或 IPv6）和 UDP 标头，再加上 NX_DNS_MESSAGE_MAX 所指定的最大 DNS 消息大小。无论是否定义，数据包有效负载均为 4 字节对齐，并存储在 NX_DNS_PACKET_PAYLOAD 中。
- NX_DNS_PACKET_POOL_SIZE：用于发送 DNS 查询的客户端数据包池的大小（如果未定义 NX_DNS_CLIENT_USER_CREATE_PACKET_POOL）。默认值足以容纳 16 个具有 NX_DNS_PACKET_PAYLOAD 所定义的有效负载大小的数据包，并且为 4 字节对齐。
- NX_DNS_MAX_RETRIES：DNS 客户端查询当前 DNS 服务器的最大次数，达到此次数之后，它会尝试其他服务器或中止 DNS 查询。
- NX_DNS_MAX_RETRANS_TIMEOUT：对特定 DNS 服务器执行的 DNS 查询的最大重新传输超时。默认值为 64 秒 (64 *NX_IP_PERIODIC_RATE)。
- NX_DNS_IP_GATEWAY_AND_DNS_SERVER：如果已定义，而且客户端 IPv4 网关地址为非零值，则 DNS 客户端会将 IPv4 网关设置为客户端的主 DNS 服务器。默认值为“已禁用”。
- NX_DNS_PACKET_ALLOCATE_TIMEOUT：此选项设置从 DNS 客户端数据包池分配数据包时的超时选项。默认值为 1 秒 (1*NX_IP_PERIODIC_RATE)。

- NX_DNS_CLIENT_USER_CREATE_PACKET_POOL：此选项使 DNS 客户端可以让应用程序创建和设置 DNS 客户端数据包池。默认情况下，此选项已禁用，DNS 客户端会在 nx_dns_create 中创建自己的数据包池。
- NX_DNS_CLIENT_CLEAR_QUEUE：此选项使 DNS 客户端能够先从接收队列中清除旧的 DNS 消息，然后再发送新查询。通过删除来自先前 DNS 查询的这些数据包，可以防止 DNS 客户端套接字队列溢出并丢弃有效的数据包。
- NX_DNS_ENABLE_EXTENDED_RR_TYPES：此选项使 DNS 客户端可以在 CNAME、NS、MX、SOA、SRV 和 TXT 中查询其他 DNS 记录类型。
- NX_DNS_CACHE_ENABLE：此选项使 DNS 客户端可以将应答记录存储到 DNS 缓存中。

第 3 章 - Azure RTOS NetX DNS 客户端服务说明

2021/4/29 •

本章按字母顺序提供所有 Azure RTOS NetX DNS 服务说明（下面列出的）。

在以下 API 说明的"返回值"部分，以粗体显示的值不受 NX_DISABLE_ERROR_CHECKING 定义（用于禁用 API 错误检查）影响，而对于非粗体值，则会完全禁用此检查。

- nx_dns_authority_zone_start_get：查找与指定主机名相关联授权的区域开头
- nx_dns_cache_initialize：初始化 DNS 缓存。
- nx_dns_cache_notify_clear：清除"缓存已满"通知函数。
- nx_dns_cache_notify_set：设置"缓存已满"通知函数。
- nx_dns_cname_get：查找输入域名别名的规范域名
- nx_dns_create：创建 DNS 客户端实例
- nx_dns_delete：删除 DNS 客户端实例
- nx_dns_domain_name_server_get：查找输入域区域的授权名称服务器
- nx_dns_domain_mail_exchange_get：查找与指定主机名关联的邮件交换。
- nx_dns_domain_service_get：查找与指定主机名关联的服务
- nx_dns_get_serverlist_size：返回 DNS 客户端服务器列表的大小
- nx_dns_info_by_name_get：通过查询输入主机名返回 IP 地址、端口
- nx_dns_ipv4_address_by_name_get：根据指定主机名查找 IPv4 地址
- nx_dns_host_by_address_get：根据指定 IP 地址查找主机名
- nx_dns_host_by_name_get：根据指定主机名查找 IPv4 地址
- nx_dns_host_text_get：查找输入域名的文本数据
- nx_dns_packet_pool_set：设置 DNS 客户端数据包池
- nx_dns_server_add：将指定地址的 DNS 服务器添加到客户端列表中
- nx_dns_server_get：返回客户端列表中的 DNS 服务器
- nx_dns_server_remove：从客户端列表中删除 DNS 服务器
- nx_dns_server_remove_all：从客户端列表中删除所有 DNS 服务器

nx_dns_authority_zone_start_get

查找输入主机授权的区域开头

原型

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

说明

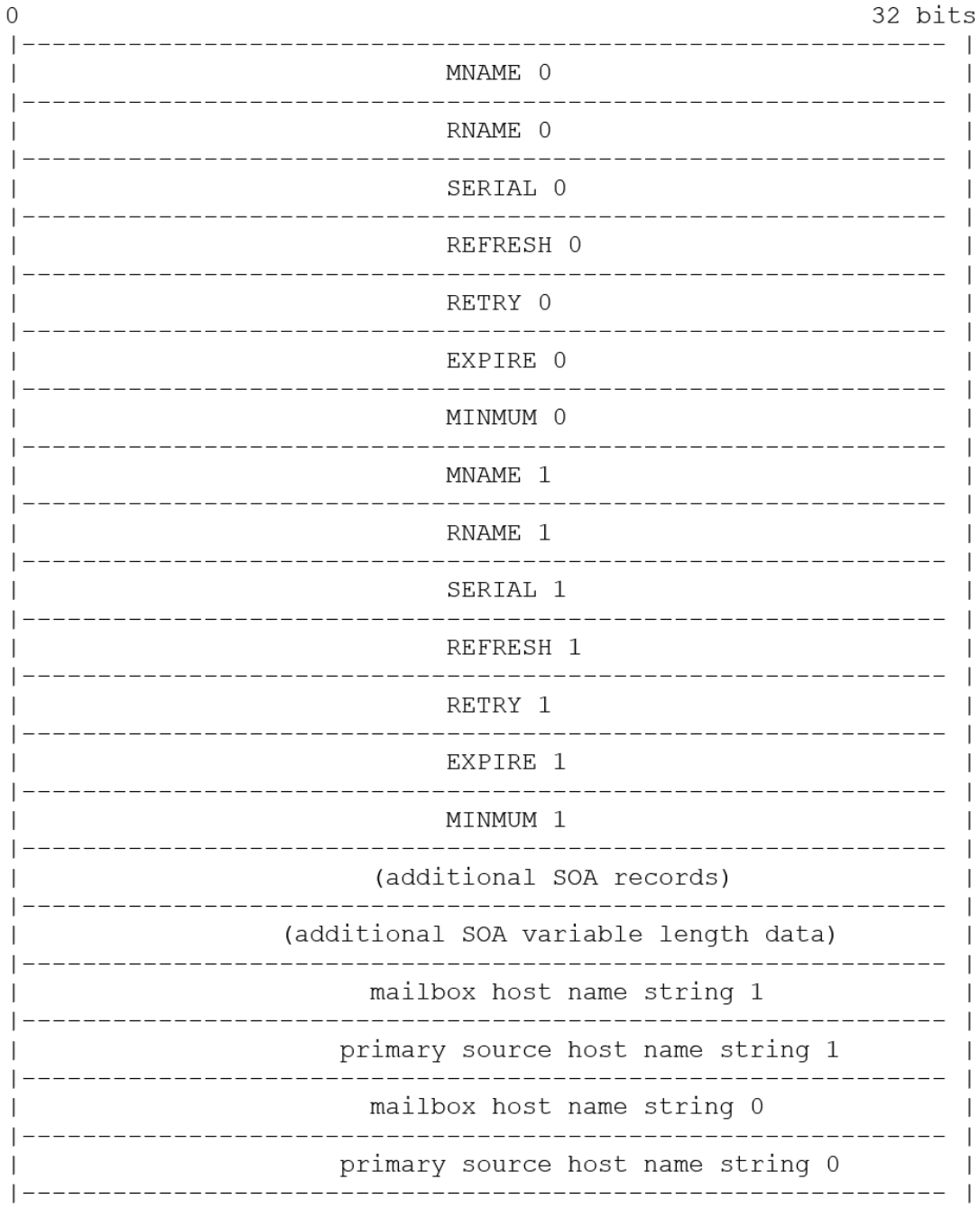
如果已定义 NX_DNS_ENABLE_EXTENDED_RR_TYPES，此服务可使用指定域名发送类型为 SOA 的查询，以便获取输入域名授权的区域开头。DNS 客户端会将 DNS 服务器响应中返回的 SOA 记录复制到 record_buffer 内存位置。

> **NOTE**
>
> record_buffer 接收数据时必须 4 字节对齐。

在 NetX DNS 客户端中，SOA 记录类型 NX_DNS_SOA_ENTRY 保存为七个 4 字节参数（总计 28 字节）：

- nx_dns_soa_host_mname_ptr：指向此区域主数据源的指针
- nx_dns_soa_host_rname_ptr：指向此区域专用邮箱的指针
- nx_dns_soa_serial：区域版本号
- nx_dns_soa_refresh：刷新间隔
- nx_dns_soa_retry：SOA 查询重试间隔
- nx_dns_soa_expire：SOA 过期时的持续时间
- nx_dns_soa_minmum：SOA 主机名 DNS 回复消息中的最小 TTL 字段

下面显示了两个 SOA 记录的存储方式。从缓冲区的顶部开始输入包含固定长度数据的 SOA 记录。指针 MNAME 和 RNAME 指向可变长度数据（主机名），这类数据存储在缓冲区底部。其他 SOA 记录（"其他 SOA 记录..."）会输入到第一条记录之后，而其可变长度数据（"其他 SOA 可变长度数据"）会存储到最后一项可变长度数据的上方：



如果输入 record_buffer 无法容纳服务器回复中的所有 SOA 数据，则 record_buffer 会容纳尽可能多的记录，并返回缓冲区中的记录数。

使用 *record_count 中返回的 SOA 记录数，应用程序可分析 record_buffer 中的数据，并提取授权主机名字符串的区域开头。

输入参数

- dns_ptr：指向 DNS 客户端的指针。
- host_name：指向主机名以获取其 SOA 数据的指针
- record_buffer：指向相应位置以从中提取 SOA 数据的指针
- buffer_size：保存 SOA 数据所用缓冲区的大小
- record_count：指向所检索 SOA 记录数的指针
- wait_option：用于接收 DNS 服务器响应的等待选项

返回值

- **NX_SUCCESS**：(0x00) 成功获取 SOA 数据

- **NX_DNS_NO_SERVER**:(0xA1) 客户端服务器列表为空
- **NX_DNS_QUERY_FAILED**:(0xA3) 未收到有效 DNS 响应
- NX_PTR_ERROR:(0x07) IP 或 DNS 指针无效
- NX_CALLER_ERROR:(0x11) 此服务的调用方无效
- NX_DNS_PARAM_ERROR:(0xA8) 非指针输入无效

获准方式

线程数

示例

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

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

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

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

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

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

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

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

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

nx_dns_cache_initialize

初始化 DNS 缓存

原型

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

说明

此服务可创建并初始化 DNS 缓存。

输入参数

- dns_ptr：指向 DNS 控制块的指针。
- cache_ptr：指向 DNS 缓存的指针。
- cache_size：DNS 缓存大小（以字节为单位）。

返回值

- **NX_SUCCESS**：(0x00) 成功初始化 DNS 缓存
- NX_DNS_ERROR：(0xA0) 缓存不是 4 字节对齐。
- NX_DNS_PARAM_ERROR：(0xA8) DNS ID 无效。
- NX_PTR_ERROR：(0x07) DNS 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

获准方式

线程数

示例

```
UCHAR          dns_cache [2048];

/* Initialize the DNS Cache.  */
status =  nx_dns_cache_initialize(&my_dns, dns_cache, 2048);
/* If status is NX_SUCCESS DNS Cache was successfully initialized.  */
```

nx_dns_cache_notify_clear

清除“DNS 缓存已满”通知函数

原型

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

说明

此服务可清除“缓存已满”通知函数。

输入参数

- dns_ptr：指向 DNS 控制块的指针。

返回值

- **NX_SUCCESS**：(0x00) 成功设置 DNS 缓存通知
- NX_DNS_PARAM_ERROR：(0xA8) DNS ID 无效。
- NX_PTR_ERROR：(0x07) DNS 指针无效。

获准方式

线程数

示例

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

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

nx_dns_cache_notify_set

设置“DNS 缓存已满”通知函数

原型

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

说明

此服务可设置“缓存已满”通知函数。

输入参数

- dns_ptr：指向 DNS 控制块的指针。
- cache_full_notify_cb：在缓存已满时要调用的回调函数。

返回值

- **NX_SUCCESS**：(0x00) 成功设置 DNS 缓存通知
- NX_DNS_PARAM_ERROR：(0xA8) DNS ID 无效。
- NX_PTR_ERROR：(0x07) DNS 指针无效。

获准方式

线程数

示例

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

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

nx_dns_cname_get

查找输入主机名的规范名称

原型

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

说明

如果已在 nx_dns.h 中定义 NX_DNS_ENABLE_EXTENDED_RR_TYPES，此服务可使用指定域名发送类型为 CNAME 的查询，以便获取规范域名。DNS 客户端会将 DNS 服务器响应中返回的 CNAME 字符串复制到 record_buffer 内存位置。

输入参数

- dns_ptr：指向 DNS 客户端的指针。
- host_name：指向主机名以获取其 CNAME 数据的指针
- record_buffer：指向相应位置以从中提取 CNAME 数据的指针
- buffer_size：保存 CNAME 数据所用缓冲区的大小
- wait_option：用于接收 DNS 服务器响应的等待选项

返回值

- **NX_SUCCESS**：(0x00) 成功获取 CNAME 数据
- **NX_DNS_NO_SERVER**：(0xA1) 客户端服务器列表为空
- **NX_DNS_QUERY_FAILED**：(0xA3) 未收到有效 DNS 响应
- NX_PTR_ERROR：(0x07) IP 或 DNS 指针无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效
- NX_DNS_PARAM_ERROR：(0xA8) 非指针输入无效

获准方式

线程数

示例

```
CHAR            record _buffer[50];

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

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

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

```
Test CNAME: **my_example**.com
```

nx_dns_create

创建 DNS 客户端实例

原型

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

说明

此服务可为以前创建的 IP 实例创建 DNS 客户端实例。

> **NOTE**
> 应用程序必须确保 DNS 客户端所用数据包池的数据包有效负载足够大，以便容纳最大 512 字节的 DNS 消息，以及 UDP、IP 和以太网标头。如果 DNS 客户端创建其自己的数据包池，则此服务由 NX_DNS_PACKET_POOL_SIZE 和 NX_DNS_PACKET_PAYLOAD 进行定义。如果 DNS 客户端应用程序喜欢提供以前创建的数据包池，则 IPv4 DNS 客户端的有效负载应为 512 字节的最大 DNS，以及 20 字节的 IP 标头、8 字节的 UDP 标头和 14 字节的以太网标头。

输入参数

- dns_ptr：指向 DNS 客户端的指针。
- ip_ptr：指向以前所创建 IP 实例的指针。
- domain_name：指向 DNS 实例域名的指针。

返回值

- **NX_SUCCESS**：(0x00) DNS 创建成功
- **NX_DNS_ERROR**：(0xA0) DNS 创建错误
- NX_PTR_ERROR：(0x07) IP 或 DNS 指针无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

获准方式

线程数

示例

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

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

nx_dns_delete

删除 DNS 客户端实例

原型

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

说明

此服务可删除以前创建的 DNS 客户端实例，并释放其资源。

> **NOTE**
>
> 如果已定义 NX_DNS_CLIENT_USER_CREATE_PACKET_POOL，并为 DNS 客户端分配了用户定义的数据包池，则应用程序可在不再需要 DNS 客户端数据包池时将其删除。

输入参数

- dns_ptr：指向以前所创建 DNS 客户端实例的指针。

返回值

- **NX_SUCCESS**：(0x00) DNS 客户端删除成功。
- NX_PTR_ERROR：(0x07) IP 或 DNS 客户端指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

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

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

nx_dns_domain_name_server_get

查找输入域区域的授权名称服务器

原型

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

说明

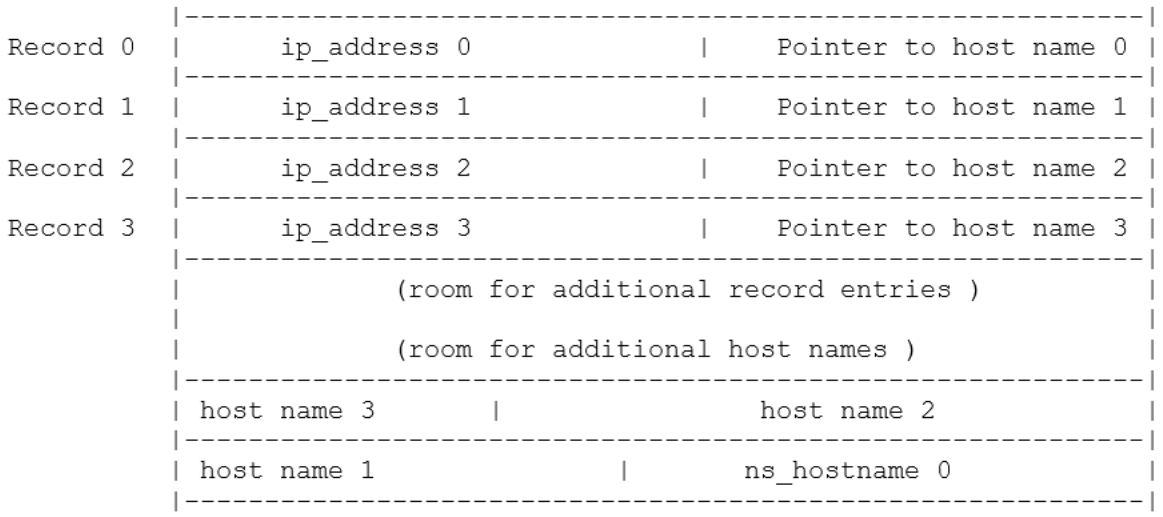
如果已定义 NX_DNS_ENABLE_EXTENDED_RR_TYPES，此服务可使用指定域名发送类型为 NS 的查询，以便获取输入域名的名称服务器。DNS 客户端会将 DNS 服务器响应中返回的 NS 记录复制到 record_buffer 内存位置。

> **NOTE**
>
> record_buffer 接收数据时必须 4 字节对齐。

在 NetX DNS 客户端中，NS 数据类型 NX_DNS_NS_ENTRY 保存为两个 4 字节参数：

- nx_dns_ns_ipv4_address：名称服务器的 IPv4 地址
- nx_dns_ns_hostname_ptr：指向名称服务器主机名的指针

下面显示的缓冲区包含四个 NX_DNS_NS_ENTRY 记录。每个条目中指向主机名字符串的指针均会指向缓冲区下半部分的相应主机名字符串：



如果输入 record_buffer 无法容纳服务器回复中的所有 NS 数据，则 record_buffer 会容纳尽可能多的记录，并返回缓冲区中的记录数。

使用 *record_count 中返回的 NS 记录数，应用程序可分析 record_buffer 中每个记录的 IP 地址和主机名。

输入参数

- dns_ptr：指向 DNS 客户端的指针。
- host_name：指向主机名以获取其 NS 数据的指针
- record_buffer：指向相应位置以从中提取 NS 数据的指针
- buffer_size：保存 NS 数据所用缓冲区的大小
- record_count：指向所检索 NS 记录数的指针
- wait_option：用于接收 DNS 服务器响应的等待选项

返回值

- **NX_SUCCESS**：(0x00) 成功获取 NS 数据
- **NX_DNS_NO_SERVER**：(0xA1) 客户端服务器列表为空
- **NX_DNS_QUERY_FAILED**：(0xA3) 未收到有效 DNS 响应
- NX_DNS_PARAM_ERROR：(0xA8) DNS ID 无效。
- NX_PTR_ERROR：(0x07) IP 或 DNS 指针无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

获准方式

线程数

示例

```
#define RECORD_COUNT        10

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

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

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

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

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

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

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

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

nx_dns_domain_mail_exchange_get

查找输入主机名的邮件交换

原型

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

说明

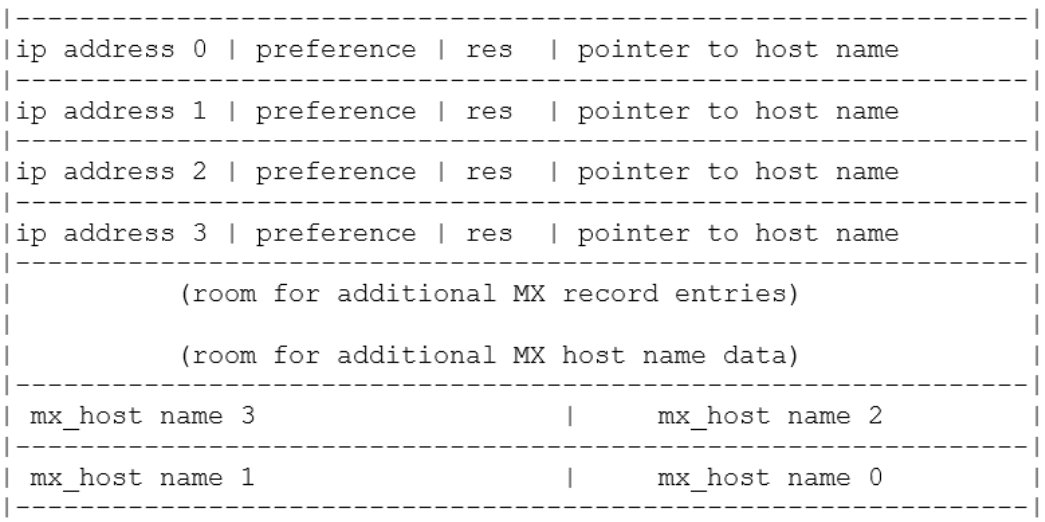
如果已定义 NX_DNS_ENABLE_EXTENDED_RR_TYPES，此服务可使用指定域名发送类型为 MX 的查询，以便获取输入域名的邮件交换。DNS 客户端会将 DNS 服务器响应中返回的 MX 记录复制到 record_buffer 内存位置。

> **NOTE**
>
> record_buffer 接收数据时必须 4 字节对齐。

在 NetX DNS 客户端中，邮件交换记录类型 NX_DNS_MAIL_EXCHANGE_ENTRY 保存为四个参数，总计 12 字节：

- nx_dns_mx_ipv4_address：邮件交换 IPv4 地址（4 字节）
- nx_dns_mx_preference：首选项（2 字节）
- nx_dns_mx_reserved0：保留 2 字节
- nx_dns_mx_hostname_ptr：指向邮件交换服务器主机名的指针（4 字节）

下面显示的缓冲区包含四个 MX 记录。每个记录的固定长度数据都显示在列表上方。指向邮件交换服务器主机名的指针会指向缓冲区底部的相应主机名。



如果输入 record_buffer 无法容纳服务器回复中的所有 MX 数据，则 record_buffer 会容纳尽可能多的记录，并返回缓冲区中的记录数。

使用 *record_count 中返回的 MX 记录数，应用程序可分析 MX 参数，包括 record_buffer 中每个记录的邮件主机名。

输入参数

- dns_ptr：指向 DNS 客户端的指针。
- host_name：指向主机名以获取其 MX 数据的指针
- record_buffer：指向相应位置以从中提取 MX 数据的指针
- buffer_size：保存 MX 数据所用缓冲区的大小
- record_count：指向所检索 MX 记录数的指针

- wait_option：用于接收 DNS 服务器响应的等待选项

返回值

- **NX_SUCCESS**：(0x00) 成功获取 MX 数据
- **NX_DNS_NO_SERVER**：(0xA1) 客户端服务器列表为空
- **NX_DNS_QUERY_FAILED**：(0xA3) 未收到有效 DNS 响应
- NX_DNS_PARAM_ERROR：(0xA8) DNS ID 无效。
- NX_PTR_ERROR：(0x07) IP 或 DNS 指针无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

获准方式

线程数

示例

```
#define           MAX_RECORD_COUNT 10

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

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

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

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

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


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

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

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

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

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

nx_dns_domain_service_get

按输入主机名查找所提供服务

原型

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

说明

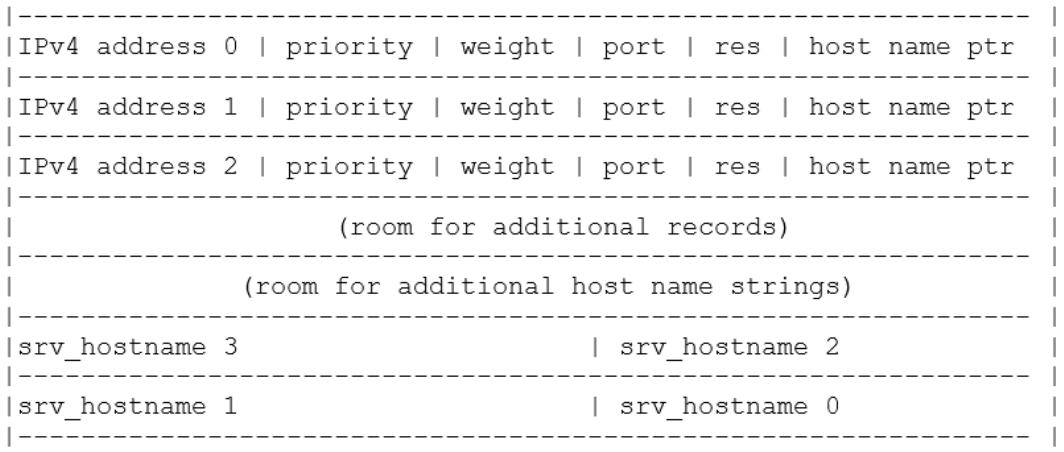
如果已定义 NX_DNS_ENABLE_EXTENDED_RR_TYPES，此服务可使用指定域名发送类型为 SRV 的查询，以便查找与指定域关联的服务及其端口号。DNS 客户端会将 DNS 服务器响应中返回的 SRV 记录复制到 record_buffer 内存位置。

> **NOTE**
> record_buffer 接收数据时必须 4 字节对齐。

在 NetX DNS 客户端中，服务记录类型 NX_DNS_SRV_ ENTRY 保存为六个参数，总计 16 字节。这样，可将可变长度 SRV 数据高效存储在内存中：

- 服务器 IPv4 地址：nx_dns_srv_ipv4_address（4 字节）
- 服务器优先级：nx_dns_srv_priority（2 字节）
- 服务器权重：nx_dns_srv_weight（2 字节）
- 服务端口号：nx_dns_srv_port_number（2 字节）
- 为实现 4 字节对齐保留：nx_dns_srv_reserved0（2 字节）
- 指向服务器主机名的指针：*nx_dns_srv_hostname_ptr（4 字节）

所提供缓冲区中已存储四个 SRV 记录。每个 NX_DNS_SRV_ENTRY 记录均包含指针 nx_dns_srv_hostname_ptr，可指向记录缓冲区底部的相应主机名字符串：



如果输入 record_buffer 无法容纳服务器回复中的所有 SRV 数据，则 record_buffer 会容纳尽可能多的记录，并返回缓冲区中的记录数。

使用 *record_count 中返回的 SRV 记录数，应用程序可分析 SRV 参数，包括 record_buffer 中每个记录的服务器主机名。

输入参数

- dns_ptr：指向 DNS 客户端的指针。
- host_name：指向主机名以获取其 SRV 数据的指针
- record_buffer：指向相应位置以从中提取 SRV 数据的指针
- buffer_size：保存 SRV 数据所用缓冲区的大小
- record_count：指向所检索 SRV 记录数的指针
- wait_option：用于接收 DNS 服务器响应的等待选项

返回值

- **NX_SUCCESS**：(0x00) 成功获取 SRV 数据
- **NX_DNS_NO_SERVER**：(0xA1) 客户端服务器列表为空
- **NX_DNS_QUERY_FAILED**：(0xA3) 未收到有效 DNS 响应
- NX_DNS_PARAM_ERROR：(0xA8) DNS ID 无效。
- NX_PTR_ERROR：(0x07) IP 或 DNS 指针无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

获准方式

线程数

示例

```
#define MAX_RECORD_COUNT  10

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

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

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

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

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


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

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

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

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

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

nx_dns_get_serverlist_size

返回 DNS 客户端服务器列表的大小

原型

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

说明

此服务可返回客户端列表中有效 DNS 服务器的数量。

输入参数

- dns_ptr：指向 DNS 控制块的指针
- size：返回列表中服务器的数量

返回值

- **NX_SUCCESS**：(0x00) 成功返回 DNS 服务器列表大小
- NX_PTR_ERROR：(0x07) IP 或 DNS 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

获准方式

线程数

示例

```
UINT my_listsize;

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

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

nx_dns_info_by_name_get

按主机名返回 DNS 服务器的 IP 地址和端口

原型

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

说明

此服务可通过 DNS 查询，按输入主机名返回服务器 IP 和端口（服务记录）。如果找不到服务记录，此例程会使用输入地址指针返回零 IP 地址，并返回非零错误状态，以指示错误。

输入参数

- dns_ptr：指向 DNS 控制块的指针
- host_name：指向主机名缓冲区的指针
- host_address_ptr：指向要返回地址的指针
- host_port_ptr：指向端口以返回 DNS 响应 wait_option 等待选项的指针

返回值

- **NX_SUCCESS**：(0x00) 成功返回 DNS 服务器记录
- **NX_DNS_NO_SERVER**：(0xA1) 未在客户端上注册发送主机名查询的 DNS 服务器
- **NX_DNS_QUERY_FAILED**：(0xA3) DNS 查询失败；客户端列表中的任何 DNS 服务器均未响应，或者未对此输入主机名提供服务记录。
- NX_PTR_ERROR：(0x07) IP 或 DNS 指针无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

获准方式

线程数

示例

```
ULONG         ip_address;
USHORT         port;

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

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

nx_dns_ipv4_address_by_name_get

查找输入主机名的 IPv4 地址

原型

```
UINT nx_dns_ipv4_address_by_name_get (NX_DNS *dns_ptr,
                                      UCHAR *host_name_ptr, VOID *buffer,
                                      UINT buffer_size,
                                      UINT *record_count,
                                      ULONG wait_option);
```

说明

此服务可使用指定主机名发送类型为 A 的查询，以便获取输入主机名的 IP 地址。DNS 客户端会将 DNS 服务器响应中返回的 A 记录 IPv4 地址复制到 record_buffer 内存位置。

NOTE

record_buffer 接收数据时必须 4 字节对齐。

下面显示的 4 字节对齐缓冲区中已存储多个 IPv4 地址：

```
|-------------------------------------------------------------- |
| Address 0 | Address 1 | Address 2 |  .  . . . . . |  Address n   |
|-------------------------------------------------------------- |
```

如果所提供缓冲区无法容纳所有 IP 地址数据，则 record_buffer 中不会存储剩余 A 记录。这样，应用程序可检索到服务器回复中的一个、部分或全部可用 IP 地址数据。

使用 *record_count 中返回的 A 记录数，应用程序可分析 record_buffer 中的 IPv4 地址数据。

输入参数

- dns_ptr：指向 DNS 客户端的指针。
- host_name_ptr：指向主机名以获取 IPv4 地址缓冲区的指针，指向相应位置以从中提取 IPv4 数据的指针
- buffer_size：保存 IPv4 数据所用缓冲区的大小
- wait_option：用于接收 DNS 服务器响应的等待选项

返回值

- **NX_SUCCESS**：(0x00) 成功获取 IPv4 数据
- **NX_DNS_NO_SERVER**：(0xA1) 客户端服务器列表为空
- **NX_DNS_QUERY_FAILED**：(0xA3) 未收到有效 DNS 响应
- NX_DNS_PARAM_ERROR：(0xA8) 输入参数无效。
- NX_PTR_ERROR：(0x07) IP 或 DNS 指针无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

获准方式

线程数

示例

```
#define MAX_RECORD_COUNT  20

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

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

        /* Check for DNS query error.  */
if (status != NX_SUCCESS)
{
        error_counter++;
}
else
{
    /* If status is NX_SUCCESS a DNS query was successfully completed the IPv4 address(es) is returned in
record_buffer.  */
    printf("------------------------------------------------------\n");
    printf("Test A: ");
    printf("record_count = %d \n", record_count);


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

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

nx_dns_host_by_address_get

根据 IP 地址查找主机名

原型

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

说明

此服务可请求根据应用程序以前指定的一个或多个 DNS 服务器所提供 IP 地址来解析名称。如果成功，host_name_ptr 所指定字符串中会返回以 NULL 结尾的主机名。

输入参数

- dns_ptr：指向以前所创建 DNS 实例的指针。
- ip_address：要解析为名称的 IP 地址
- host_name_ptr：指向主机名目标区域的指针
- max_host_name_size：主机名目标区域的大小
- wait_option：定义服务在每次 DNS 查询和查询重试后要等待 DNS 服务器响应的时长（以计时器刻度为单位）。等待选项定义如下：
  - timeout value：(0x00000001-0xFFFFFFFE) 选择数值 (1-0xFFFFFFFE) 可指定等待 DNS 解析时服务保持挂起状态的最大计时器刻度数。
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 DNS 服务器响应请求。

返回值

- **NX_SUCCESS**：(0x00) DNS 解析成功
- **NX_DNS_TIMEOUT**：(0xA2) 获取 DNS 互斥体时超时
- **NX_DNS_NO_SERVER**：(0xA1) 未指定 DNS 服务器地址
- **NX_DNS_QUERY_FAILED**：(0xA3) 未收到查询响应
- **NX_DNS_BAD_ADDRESS_ERROR**：(0xA4) 空输入地址
- **NX_DNS_INVALID_ADDRESS_TYPE**：(0xB2) 索引指向无效地址类型（例如 IPv6）
- **NX_DNS_PARAM_ERROR**：(0xA8) 非指针输入无效
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

获准方式

线程数

示例

```
#define BUFFER_SIZE   200

UCHAR   resolved_name[200];

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

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

nx_dns_host_by_name_get

根据主机名查找 IP 地址

原型

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

说明

此服务可请求根据应用程序以前指定的一个或多个 DNS 服务器所提供且 host_name 所指向的名称来解析名称。如果成功，则 host_address_ptr 所指向的目标中会返回关联的 IP 地址。

输入参数

- dns_ptr：指向以前所创建 DNS 实例的指针。
- host_name：指向主机名的指针
- host_address_ptr：指向 DNS 主机 IP 地址的指针
- wait_option：定义服务等待 DNS 解析的时长。等待选项定义如下：
  - timeout value：(0x00000001 - 0xFFFFFFFE) 选择数值 (1-0xFFFFFFFE) 可指定等待 DNS 解析时服务保持挂起状态的最大计时器刻度数。
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 DNS 服务器响应请求。

返回值

- **NX_SUCCESS**：(0x00) DNS 解析成功。
- **NX_DNS_NO_SERVER**：(0xA1) 未指定 DNS 服务器地址
- **NX_DNS_QUERY_FAILED**：(0xA3) 未收到查询响应
- NX_DNS_PARAM_ERROR：(0xA8) 非指针输入无效
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

获准方式

线程数

示例

```
ULONG ip_address;

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

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

    else
    {

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

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

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

nx_dns_host_text_get

查找输入域名的文本字符串

原型

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

说明

此服务可使用指定域名和缓冲区发送类型为 TXT 的查询，以便获取任意字符串数据。

DNS 客户端会将 DNS 服务器响应中的 TXT 记录文本字符串复制到 record_buffer 内存位置。

> **NOTE**
>
> record_buffer 接收数据时不必 4 字节对齐。

输入参数

- dns_ptr：指向 DNS 客户端的指针。
- host_name：指向要搜索主机所用名称的指针
- record_buffer：指向相应位置以从中提取 TXT 数据的指针
- buffer_size：保存 TXT 数据所用缓冲区的大小
- wait_option：用于接收 DNS 服务器响应的等待选项

返回值

- **NX_SUCCESS**：(0x00) 成功获取 TXT 字符串
- **NX_DNS_NO_SERVER**：(0xA1) 客户端服务器列表为空
- **NX_DNS_QUERY_FAILED**：(0xA3) 未收到有效 DNS 响应
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效
- NX_DNS_PARAM_ERROR：(0xA8) 非指针输入无效

获准方式

线程数

示例

```
CHAR            record_buffer[50];

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


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

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

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

nx_dns_packet_pool_set

设置 DNS 客户端数据包池

原型

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

说明

此服务可将以前创建的数据包池设置为 DNS 客户端数据包池。DNS 客户端会使用此数据包池发送 DNS 查询，因此数据包有效负载不能小于包括以太网帧、IP 和 UDP 标头的 NX_DNS_PACKET_PAYLOAD_UNALIGNED，并且需在 nx_dns.h 中进行定义。

> **NOTE**
> 删除 DNS 客户端时，不会删除其数据包池，应用程序可在不再需要数据包池时将其删除。

> **NOTE**
> 此服务仅在 nx_dns.h 中定义了配置选项 NX_DNS_CLIENT_USER_CREATE_PACKET_POOL 时才可用

输入参数

- dns_ptr：指向以前所创建 DNS 客户端实例的指针。
- pool_ptr：指向以前所创建数据包池的指针

返回值

- **NX_SUCCESS**：(0x00) 成功完成。
- **NX_NOT_ENABLED**：(0x14) 未在客户端中配置此选项
- NX_PTR_ERROR：(0x07) IP 或 DNS 客户端指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

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


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

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

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

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

nx_dns_server_add

添加 DNS 服务器 IP 地址

原型

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

说明

此服务可向服务器列表添加 IPv4 DNS 服务器。

输入参数

- dns_ptr：指向 DNS 控制块的指针。
- server_address：DNS 服务器的 IP 地址

返回值

- **NX_SUCCESS**：(0x00) 成功添加服务器
- **NX_DNS_DUPLICATE_ENTRY 或 NX_NO_MORE_ENTRIES**：(0x17) 不允许添加更多 DNS 服务器（列表已满）
- **NX_DNS_PARAM_ERROR**：(0xA8) 非指针输入无效
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效
- NX_DNS_BAD_ADDRESS_ERROR：(0xA4) 空服务器地址输入

获准方式

线程数

示例

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

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

nx_dns_server_get

返回客户端列表中的 IPv4 DNS 服务器

原型

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

说明

此服务可返回服务器列表中指定索引处的 IPv4 DNS 服务器地址。请注意，该索引从零开始。如果输入索引超出 DNS 客户端列表大小，则会返回错误。可以先调用 nx_dns_get_serverlist_size 服务，以获取客户端列表中的 DNS 服务器数量。

输入参数

- dns_ptr：指向 DNS 控制块的指针
- index：DNS 客户端服务器列表索引
- dns_server_address：指向 DNS 服务器 IP 地址的指针

返回值

- **NX_SUCCESS**：(0x00) 成功返回服务器
- **NX_DNS_SERVER_NOT_FOUND**：(0xA9) 索引指向空槽
- **NX_DNS_BAD_ADDRESS_ERROR**：(0xA4) 索引指向空地址
- **NX_DNS_PARAM_ERROR**：(0xA8) 索引超出列表大小
- NX_PTR_ERROR：(0x07) IP 或 DNS 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

获准方式

线程数

示例

```
ULONG     my_server_address;

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

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

nx_dns_server_remove

从客户端列表中删除 IPv4 DNS 服务器

原型

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

说明

此服务可从客户端列表中删除 IPv4 DNS 服务器。

输入参数

- dns_ptr：指向 DNS 控制块的指针。
- server_address：DNS 服务器的 IP 地址。

返回值

- **NX_SUCCESS**：(0x00) 成功删除 DNS 服务器

- **NX_DNS_SERVER_NOT_FOUND**: (0xA9) 客户端列表中无服务器
- **NX_DNS_BAD_ADDRESS_ERROR**: (0xA4) 空服务器地址输入
- NX_PTR_ERROR: (0x07) IP 或 DNS 指针无效。
- NX_CALLER_ERROR: (0x11) 此服务的调用方无效

获准方式

线程数

示例

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

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

nx_dns_server_remove_all

从客户端列表中删除所有 DNS 服务器

原型

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

说明

此服务可从客户端列表中删除所有 DNS 服务器。

输入参数

- dns_ptr: 指向 DNS 控制块的指针。

返回值

- **NX_SUCCESS**: (0x00) 成功删除 DNS 服务器
- **NX_DNS_ERROR**: (0xA0) 无法获取保护互斥体
- NX_PTR_ERROR: (0x07) IP 或 DNS 指针无效。
- NX_CALLER_ERROR: (0x11) 此服务的调用方无效

获准方式

线程数

示例

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

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

第 1 章 - Azure RTOS NetX FTP 简介

2021/4/29 •

文件传输协议 (FTP) 是为文件传输而设计的协议。FTP 利用可靠的传输控制协议 (TCP) 服务来实现文件传输功能。因此，FTP 是一种高度可靠的文件传输协议。FTP 也是高性能的。实际的 FTP 文件传输是在专用的 FTP 连接上执行的。

FTP 要求

为了能够正常运行，Azure RTOS NetX FTP 包要求已经创建了 NetX IP 实例。此外，必须在同一个 IP 实例上启用 TCP。NetX FTP 包的 FTP 客户端部分没有进一步的要求。

NetX FTP 包的 FTP 服务器部分有几个额外的要求。首先，它要求完全访问 TCP 已知端口 21 来处理所有客户端 FTP 命令请求，以及完全访问已知端口 20 来处理所有客户端 FTP 数据传输。FTP 服务器还设计为与 FileX 嵌入式文件系统配合使用。如果 FileX 不可用，用户可以将使用的 FileX 部分移植到他们自己的环境中。本指南后面的几部分将对此进行讨论。

FTP 约束

关于文件数据的表示，FTP 标准提供了许多选择。与 Unix 实现类似，NetX FTP 假设有以下文件格式约束：

- 文件类型：二进制
- 文件格式：仅非打印
- 文件结构：仅文件结构
- 传输模式：仅流模式

FTP 文件名

FTP 文件名应采用目标文件系统（通常是 FileX）的格式。它们应为以 NULL 结尾的 ASCII 字符串，如果需要，还应该包含完整的路径信息。在 NetX FTP 实现中，没有指定对 FTP 文件名大小的限制。但是，数据包池有效负载大小应能够容纳最大路径和/或文件名。

FTP 客户端命令

FTP 有一种用于打开连接以及执行文件和目录操作的简单机制。基本上有一组标准 FTP 命令，客户端会在已知 TCP 端口 21 上成功建立连接后发出这些命令。下面展示了一些基本 FTP 命令：

FTP 命令和含义

- CWD path：更改工作目录
- DELE filename：删除指定的文件名
- LIST directory：获取目录列表
- MKD directory：生成新目录
- NLST directory：获取目录列表
- NOOP：无操作，返回成功
- PASS password：提供登录密码
- PASV：请求被动传输模式
- PWD path：拾取当前目录路径
- QUIT：终止客户端连接
- RETR filename：读取指定文件

- RMD directory：删除指定目录
- RNFR oldfilename：指定要重命名的文件
- RNTO newfilename：将文件重命名为所提供的文件名
- STOR filename：写入指定文件
- TYPE I：选择二进制文件图像
- USER username：提供登录用户名
- PORT ip_address,port：提供 IP 地址和客户端数据端口

这些 ASCII 命令是 NetX FTP 客户端软件内部使用的，用于与 FTP 服务器进行 FTP 操作。

FTP 服务器响应

FTP 服务器利用已知 TCP 端口 21 来处理客户端命令请求。在处理客户端命令后，FTP 服务器就会返回 3 位数的 ASCII 数字响应，后跟一个可选的 ASCII 字符串。FTP 客户端软件使用此数字响应来确定操作是成功还是失败。下面列出了 FTP 服务器对客户端命令的各种响应：

第一个数字字段和含义

- 1xx：积极的初步状态 – 即将返回另一个答复。
- 2xx：积极的完成状态。
- 3xx：积极的初步状态 – 必须发送另一个命令。
- 4xx：临时错误状态。
- 5xx：错误状态。

第二个数字字段和含义

- x0x：命令中出现语法错误。
- x1x：信息性消息。
- x2x：与连接相关。
- x3x：与身份验证相关。
- x4x：未指定。
- x5x：与文件系统相关。

例如，如果断开连接成功，那么使用 QUIT 命令断开 FTP 连接的客户端请求通常会收到来自服务器的“221”响应代码。

FTP 被动传输模式

默认情况下，NetX FTP 客户端使用主动传输模式通过数据套接字与 FTP 服务器交换数据。这种安排的问题是，它要求 FTP 客户端打开 TCP 服务器套接字，以便 FTP 服务器连接。这样做可能会带来安全风险，并可能会被客户端防火墙阻止。被动传输模式与主动传输模式的不同之处在于，它是由 FTP 服务器在数据连接上创建 TCP 服务器套接字。这消除了安全风险（对于 FTP 客户端）。

为了启用被动数据传输，应用程序在之前创建的 FTP 客户端上调用 nx_ftp_client_passive_mode_set（其中第二个参数设置为 NX_TRUE）。此后，所有后续用于传输数据的 NetX FTP 客户端服务（NLST、RETR、STOR）都将尝试采用被动传输模式。

首先，FTP 客户端发送 PASV 命令（无参数）。如果 FTP 服务器支持此请求，它就会返回 227“OK”响应。然后，客户端发送请求（例如 RETR）。如果服务器拒绝被动传输模式，则 NetX FTP 客户端服务就会返回错误状态。

为了禁用被动传输模式并恢复主动传输模式，应用程序调用 nx_ftp_client_passive_mode_set（其中第二个参数设置为 NX_FALSE）。

FTP 通信

FTP 服务器利用已知 TCP 端口 21 来处理客户端请求。FTP 客户端可以使用任何可用的 TCP 端口。FTP 事件的一般顺序如下：

FTP 读取文件请求

1. 客户端发起与服务器端口 21 进行 TCP 连接。
2. 服务器发送"220"响应来指明成功。
3. 客户端发送带有"username"的"USER"消息。
4. 服务器发送"331"响应来指明成功。
5. 客户端发送带有"password"的"PASS"消息。
6. 服务器发送"230"响应来指明成功。
7. 客户端发送"TYPE I"消息进行二进制传输。
8. 服务器发送"200"响应来指明成功。
9. 客户端发送带有 IP 地址和端口的"PORT"消息。
10. 服务器发送"200"响应来指明成功。
11. 客户端发送带有要读取的文件名的"RETR"消息。
12. 服务器创建数据套接字，并连接到"PORT"命令中指定的客户端数据端口。
13. 服务器发送"125"响应来指明已开始读取文件。
14. 服务器通过数据连接发送文件内容。此过程一直继续到文件完全传输完毕。
15. 完成后，服务器断开数据连接。
16. 服务器发送"250"响应来指明已成功读取文件。
17. 客户端发送"QUIT"来终止 FTP 连接。
18. 服务器发送"221"响应来指明已成功断开连接。
19. 服务器断开 FTP 连接。

如前所述，通过 IPv4 和 IPv6 运行 FTP 的唯一区别是，对于 IPv6，PORT 命令被替换为 EPRT 命令

如果 FTP 客户端使用被动传输模式发出读取请求，则命令顺序如下所示（以粗体显示的行表示与主动传输模式不同的步骤）：

1. 客户端发起与服务器端口 21 进行 TCP 连接。
2. 服务器发送"220"响应来指明成功。
3. 客户端发送带有"username"的"USER"消息。
4. 服务器发送"331"响应来指明成功。
5. 客户端发送带有"password"的"PASS"消息。
6. 服务器发送"230"响应来指明成功。
7. 客户端发送"TYPE I"消息进行二进制传输。
8. 服务器发送"200"响应来指明成功。
9. **客户端发送"PASV"消息。**
10. **服务器发送"227"响应以及客户端要连接到的 IP 地址和端口来指明成功。**
11. 客户端发送带有要读取的文件名的"RETR"消息。
12. **服务器创建数据服务器套接字，并侦听此套接字上使用"227"响应中指定端口的客户端连接请求。**
13. **服务器在控制套接字上发送"150"响应来指明已开始读取文件。**
14. 服务器通过数据连接发送文件内容。此过程一直继续到文件完全传输完毕。
15. 完成后，服务器断开数据连接。
16. **服务器在控制套接字上发送"226"响应来指明已成功读取文件。**
17. 客户端发送"QUIT"来终止 FTP 连接。
18. 服务器发送"221"响应来指明已成功断开连接。
19. 服务器断开 FTP 连接。

FTP 写入请求

1. 客户端发起与服务器端口 21 进行 TCP 连接。
2. 服务器发送“220”响应来指明成功。
3. 客户端发送带有“username”的“USER”消息。
4. 服务器发送“331”响应来指明成功。
5. 客户端发送带有“password”的“PASS”消息。
6. 服务器发送“230”响应来指明成功。
7. 客户端发送“TYPE I”消息进行二进制传输。
8. 服务器发送“200”响应来指明成功。
9. 客户端发送带有 IP 地址和端口的“PORT”消息。
10. 服务器发送“200”响应来指明成功。
11. 客户端发送带有要写入的文件名的“STOR”消息。
12. 服务器创建数据套接字，并连接到“PORT”命令中指定的客户端数据端口。
13. 服务器发送“125”响应来指明已开始写入文件。
14. 客户端通过数据连接发送文件内容。此过程一直继续到文件完全传输完毕。
15. 完成后，客户端断开数据连接。
16. 服务器发送“250”响应来指明已成功写入文件。
17. 客户端发送“QUIT”来终止 FTP 连接。
18. 服务器发送“221”响应来指明已成功断开连接。
19. 服务器断开 FTP 连接。

如果 FTP 客户端使用被动传输模式发出写入请求，则命令顺序如下所示（以粗体显示的行表示与主动传输模式不同的步骤）：

1. 客户端发起与服务器端口 21 进行 TCP 连接。
2. 服务器发送“220”响应来指明成功。
3. 客户端发送带有“username”的“USER”消息。
4. 服务器发送“331”响应来指明成功。
5. 客户端发送带有“password”的“PASS”消息。
6. 服务器发送“230”响应来指明成功。
7. 客户端发送“TYPE I”消息进行二进制传输。
8. 服务器发送“200”响应来指明成功。
9. **客户端发送“PASV”消息。**
10. **服务器发送“227”响应以及客户端要连接到的 IP 地址和端口来指明成功。**
11. 客户端发送带有要写入的文件名的“STOR”消息。
12. **服务器创建数据服务器套接字，并侦听此套接字上使用“227”响应中指定端口的客户端连接请求。**
13. **服务器在控制套接字上发送“150”响应来指明已开始写入文件。**
14. 客户端通过数据连接发送文件内容。此过程一直继续到文件完全传输完毕。
15. 完成后，客户端断开数据连接。
16. **服务器在控制套接字上发送“226”响应来指明已成功写入文件。**
17. 客户端发送“QUIT”来终止 FTP 连接。
18. 服务器发送“221”响应来指明已成功断开连接。
19. 服务器断开 FTP 连接。

FTP 身份验证

每当进行 FTP 连接时，客户端都必须为服务器提供“username”和“password”。一些 FTP 站点允许所谓的“匿名 FTP”，即允许在没有特定用户名和密码的情况下进行 FTP 访问。对于这种类型的连接，应提供“匿名”用户名，密码应为完整的电子邮件地址。

用户负责向 NetX FTP 提供登录和注销身份验证例程。这些是在 nx_ftp_server_create*_ 函数期间提供的，并通过密码处理来调用。如果 login_* 函数返回 NX_SUCCESS，则表示连接已经过身份验证，允许执行 FTP 操作。否则，如果 login 函数返回的不是 NX_SUCCESS，则连接尝试会被拒绝。

FTP 多线程支持

可以从多个线程同时调用 NetX FTP 客户端服务。但是，对于特定 FTP 客户端实例的读取或写入请求应通过同一个线程依次执行。

FTP RFC

NetX FTP 符合 RFC959 和相关 RFC。

第 2 章 - 安装和使用 Azure RTOS NetX FTP

2021/4/30 •

本章介绍与安装、设置和使用 Azure RTOS NetX FTP 组件相关的各种问题。

产品分发

可以从我们的公共源代码存储库获取 Azure RTOS NetX，网址为：https://github.com/azure-rtos/netx/。此软件包包含两个源文件和一个 PDF 文件，该 PDF 文件包含本文档，如下所示：

- nx_ftp.h：NetX FTP 的头文件
- nx_ftp_client.c：NetX FTP 客户端的 C 源文件
- nx_ftp_server.c：NetX FTP 服务器的 C 源文件
- filex_stub.h：存根文件，如果 FileX 不存在
- nx_ftp.pdf：NetX FTP 的 PDF 说明
- demo_netx_ftp.c：FTP 演示系统

FTP 安装

若要使用 NetX FTP，应将之前提到的全部分发文件复制到安装了 NetX 的同一目录中。例如，如果 NetX 安装在“\threadx\arm7\green”目录中，则应将 nx_ftp.h、nx_ftp_client.c 和 nx_ftp_server.c 文件复制到该目录中。

使用 FTP

NetX FTP 易于使用。基本上来说，应用程序代码必须包含 nx_ftp.h 和 tx_api.h、fx_api.h 或 nx_api.h，才能使用 ThreadX、FileX 或 NetX。在包含 nx_ftp.h 之后，应用程序代码即可发出本指南后文所指定的 FTP 函数调用。在生成过程中，应用程序还必须包含 nx_ftp_client.c 和 nx_ftp_server.c。这些文件必须采用与其他应用程序文件相同的方式进行编译，并且其对象窗体必须与应用程序文件一起链接。这就是使用 NetX FTP 所需的一切。

> **NOTE**
> 由于 FTP 利用 NetX TCP 服务，因此在使用 FTP 之前，必须通过 nx_tcp_enable 调用启用 TCP。

小型示例系统

下面的图 1.1 举例说明了 NetX FTP 是多么易于使用。

> **NOTE**
> 此示例适用于具有单个网络接口的主机设备。

在此示例中，FTP 包含文件 nx_ftp_client.h 和 nx_ftp_server.h 分别在第 10 行和第 11 行引入。接下来，在第 134 行的“tx_application_define”中创建 FTP 服务器。请注意，FTP 服务器控制块“Server”以前在第 31 行定义为全局变量。创建成功后，将在第 363 行启动 FTP 服务器。在第 183 行创建 FTP 客户端。最后，客户端在第 229 行写入文件，并在第 318 行读回文件。

```
/* This is a small demo of NetX FTP on the high-performance NetX TCP/IP stack. This demo
relies on ThreadX, NetX, and FileX to show a simple file transfer from the client
and then back to the server. */
```

```
#include     "tx_api.h"
#include     "fx_api.h"
#include     "nx_api.h"
#include     "nx_ftp_client.h"
#include     "nx_ftp_server.h"

#define     DEMO_STACK_SIZE     4096

/* Define the ThreadX, NetX, and FileX object control blocks... */

TX_THREAD          server_thread;
TX_THREAD          client_thread;
NX_PACKET_POOL     server_pool;
NX_IP              server_ip;
NX_PACKET_POOL     client_pool;
NX_IP              client_ip;
FX_MEDIA           ram_disk;

/* Define the NetX FTP object control blocks. */

NX_FTP_CLIENT     ftp_client;
NX_FTP_SERVER     ftp_server;

/* Define the counters used in the demo application... */

ULONG     error_counter = 0;

/* Define the memory area for the FileX RAM disk. */

UCHAR     ram_disk_memory[32000];
UCHAR     ram_disk_sector_cache[512];

#define FTP_SERVER_ADDRESS IP_ADDRESS(1,2,3,4)
#define FTP_CLIENT_ADDRESS IP_ADDRESS(1,2,3,5)

extern UINT _fx_media_format(FX_MEDIA *media_ptr, VOID (*driver)(FX_MEDIA *media),
                    VOID *driver_info_ptr, UCHAR *memory_ptr, UINT memory_size,
                    CHAR *volume_name, UINT number_of_fats, UINT directory_entries,
                    UINT hidden_sectors, ULONG total_sectors, UINT bytes_per_sector,
                    UINT sectors_per_cluster, UINT heads, UINT sectors_per_track);

/* Define the FileX and NetX driver entry functions. */
VOID     _fx_ram_driver(FX_MEDIA *media_ptr);

/* Replace the 'ram' driver with your own Ethernet driver. */
VOID     _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr);

void     client_thread_entry(ULONG thread_input);
void     thread_server_entry(ULONG thread_input);

/* Define server login/logout functions. These are stubs for functions that would
validate a client login request. */

UINT     server_login(struct NX_FTP_SERVER_STRUCT *ftp_server_ptr,
                ULONG client_ip_address, UINT client_port, CHAR *name,
                CHAR *password, CHAR *extra_info);

UINT     server_logout(struct NX_FTP_SERVER_STRUCT *ftp_server_ptr,
                    ULONG client_ip_address, UINT client_port, CHAR *name,
                    CHAR *password, CHAR *extra_info);

/* Define main entry point. */

int main()
{

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

```
    return(0);
}

/* Define what the initial system looks like. */

void     tx_application_define(void *first_unused_memory)
{

UINT      status;
UCHAR     *pointer;

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

    /* Create a helper thread for the server. */
    tx_thread_create(&server_thread, "FTP Server thread", thread_server_entry,
                     0, pointer, DEMO_STACK_SIZE,
                     4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);

    pointer = pointer + DEMO_STACK_SIZE;

    /* Initialize NetX. */
    nx_system_initialize();

    /* Create the packet pool for the FTP Server. */
    status = nx_packet_pool_create(&server_pool, "NetX Server Packet Pool",
                                   256, pointer, 8192);
    pointer = pointer + 8192;

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

    /* Create the IP instance for the FTP Server. */
    status = nx_ip_create(&server_ip, "NetX Server IP Instance",
                        FTP_SERVER_ADDRESS, 0xFFFFFF00UL, &server_pool,
                        _nx_ram_network_driver, pointer, 2048, 1);
    pointer = pointer + 2048;

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

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

    /* Enable TCP. */
    nx_tcp_enable(&server_ip);

    /* Create the FTP server. */
    status = nx_ftp_server_create(&ftp_server, "FTP Server Instance",
                     &server_ip, &ram_disk, pointer, DEMO_STACK_SIZE,
                     &server_pool, server_login, server_logout);
    pointer = pointer + DEMO_STACK_SIZE;

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

    /* Now set up the FTP Client. */

    /* Create the main FTP client thread. */
```

```
    status = tx_thread_create(&client_thread, "FTP Client thread ",
                              client_thread_entry, 0,
                              pointer, DEMO_STACK_SIZE,
                              6, 6, TX_NO_TIME_SLICE, TX_AUTO_START);
    pointer = pointer + DEMO_STACK_SIZE;

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

    /* Create a packet pool for the FTP client. */
    status = nx_packet_pool_create(&client_pool, "NetX Client Packet Pool",
                                   256, pointer, 8192);
    pointer = pointer + 8192;

    /* Create an IP instance for the FTP client. */
    status = nx_ip_create(&client_ip, "NetX Client IP Instance", FTP_CLIENT_ADDRESS,
            0xFFFFFF00UL, &client_pool, _nx_ram_network_driver, pointer, 2048, 1);
    pointer = pointer + 2048;

    /* Enable ARP and supply ARP cache memory for the FTP Client IP. */
    nx_arp_enable(&client_ip, (void *) pointer, 1024);

    pointer = pointer + 1024;

    /* Enable TCP for client IP instance. */
    nx_tcp_enable(&client_ip);

    return;
}

/* Define the FTP client thread. */
void     client_thread_entry(ULONG thread_input)
{

NX_PACKET     *my_packet;
UINT          status;

    /* Format the RAM disk - the memory for the RAM disk was defined above. */
    status = _fx_media_format(&ram_disk,
            _fx_ram_driver, /* Driver entry */
            ram_disk_memory, /* RAM disk memory pointer */
            ram_disk_sector_cache, /* Media buffer pointer */
            sizeof(ram_disk_sector_cache), /* 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 status. */
    if (status != NX_SUCCESS)
    {
        error_counter++;
        return;
    }

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

    /* Check status. */
```

```
if (status != NX_SUCCESS)
{
    error_counter++;
    return;
}

 /* Let the IP threads and driver initialize the system. */
tx_thread_sleep(100);

/* Create an FTP client. */
status = nx_ftp_client_create(&ftp_client, "FTP Client", &client_ip, 2000, &client_pool);

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

    error_counter++;
    return;
}

printf("Created the FTP Client\n");

/* Now connect with the NetX FTP (IPv4) server. */
status = nx_ftp_client_connect(&ftp_client, FTP_SERVER_ADDRESS,
                               "name", "password", 100);

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

    error_counter++;
    return;
}

printf("Connected to the FTP Server\n");

/* Open a FTP file for writing. */
status = nx_ftp_client_file_open(&ftp_client, "test.txt", NX_FTP_OPEN_FOR_WRITE, 100);

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

    error_counter++;
    return;
}

printf("Opened the FTP client test.txt file\n");

/* Allocate a FTP packet. */
status = nx_packet_allocate(&client_pool, &my_packet, NX_TCP_PACKET, 100);

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

    error_counter++;
    return;
}

/* Write ABCs into the packet payload! */
memcpy(my_packet -> nx_packet_prepend_ptr, "ABCDEFGHIJKLMNOPQRSTUVWXYZ ", 28);

/* Adjust the write pointer. */
my_packet -> nx_packet_length = 28;
my_packet -> nx_packet_append_ptr = my_packet -> nx_packet_prepend_ptr + 28;

/* Write the packet to the file test.txt. */
status = nx_ftp_client_file_write(&ftp_client, my_packet, 100);
```

```
    status = nx_ftp_client_file_write(&ftp_client, my_packet, 100);

    /* Check status. */
    if (status != NX_SUCCESS)
    {
        error_counter++;
    }
    else
        printf("Wrote to the FTP client test.txt file\n");

    /* Close the file. */
    status = nx_ftp_client_file_close(&ftp_client, 100);

    /* Check status. */
    if (status != NX_SUCCESS)
        error_counter++;
    else
        printf("Closed the FTP client test.txt file\n");

    /* Now open the same file for reading. */
    status = nx_ftp_client_file_open(&ftp_client, "test.txt", NX_FTP_OPEN_FOR_READ, 100);

    /* Check status. */
    if (status != NX_SUCCESS)
        error_counter++;

    else
        printf("Reopened the FTP client test.txt file\n");

    /* Read the file. */
    status = nx_ftp_client_file_read(&ftp_client, &my_packet, 100);

    /* Check status. */
    if (status != NX_SUCCESS)
        error_counter++;
    else
    {
        printf("Reread the FTP client test.txt file\n");
        nx_packet_release(my_packet);
    }

    /* Close this file. */
    status = nx_ftp_client_file_close(&ftp_client, 100);

    if (status != NX_SUCCESS)
        error_counter++;

    /* Disconnect from the server. */
    status = nx_ftp_client_disconnect(&ftp_client, 100);

    /* Check status. */
    if (status != NX_SUCCESS)
        error_counter++;

    /* Delete the FTP client. */
    status = nx_ftp_client_delete(&ftp_client);

    /* Check status. */
    if (status != NX_SUCCESS)
        error_counter++;
}

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

UINT     status;

    /* Wait till the IP thread and driver have initialized the system. */
    tx_thread_sleep(100);
```

```
    tx_thread_sleep(100);

    /* OK to start the FTP Server. */
    status = nx_ftp_server_start(&ftp_server);

    if (status != NX_SUCCESS)
        error_counter++;

    printf("Server started!\n");

    /* FTP server ready to take requests! */

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

    return;
}

UINT     server_login(struct NX_FTP_SERVER_STRUCT *ftp_server_ptr,
                ULONG client_ip_address, UINT client_port,
                CHAR *name, CHAR *password, CHAR *extra_info)
{

    printf("Logged in!\n");
    /* Always return success. */
    return(NX_SUCCESS);
}

UINT     server_logout(struct NX_FTP_SERVER_STRUCT *ftp_server_ptr,
                    ULONG client_ip_address, UINT client_port,
                    CHAR *name, CHAR *password, CHAR *extra_info)
{
    printf("Logged out!\n");

    /* Always return success. */
    return(NX_SUCCESS);
}
```

图 1.1 NetX FTP 客户端和服务器的示例（单网络接口主机）

配置选项

有多个配置选项用于生成 NetX FTP。以下列表详细介绍了每个配置选项：

- NX_FTP_SERVER_PRIORITY：FTP 服务器线程的优先级。默认情况下，此值定义为 16，表示将优先级指定为 16。
- NX_FTP_MAX_CLIENTS：服务器可同时处理的最大客户端数。默认情况下，此值为 4，表示同时支持 4 个客户端。
- NX_FTP_SERVER_MIN_PACKET_PAYLOAD：服务器数据包池有效负载的最小大小（以字节为单位），包括 TCP、IP 和网络帧标头以及 HTTP 数据。默认值为 256（FileX 中的最大文件名长度为）+ 12 个字节（用于存储文件信息）+ NX_PHYSICAL_TRAILER。
- NX_FTP_NO_FILEX：如果定义了此选项，则可为 FileX 依赖项提供存根。如果定义了此选项，则无需进行任何更改，FTP 客户端即可正常工作。FTP 服务器将需要进行修改，或者用户必须创建几个 FileX 服务，FTP 服务器才能正常工作。
- NX_FTP_CONTROL_TOS：FTP TCP 控制请求所需的服务类型。默认情况下，此值定义为 NX_IP_NORMAL，表示正常的 IP 数据包服务。此定义可以在包含 nx_ftp.h 之前由应用程序进行设置。
- NX_FTP_DATA_TOS：FTP TCP 数据请求所需的服务类型。默认情况下，此值定义为 NX_IP_NORMAL，表示正常的 IP 数据包服务。此定义可以在包含 nx_ftp.h 之前由应用程序进行设置。

- NX_FTP_FRAGMENT_OPTION：为 FTP TCP 请求启用分段。默认情况下，此值为 NX_DONT_FRAGMENT，表示禁用 FTP TCP 分段。此定义可以在包含 nx_ftp.h 之前由应用程序进行设置。
- NX_FTP_CONTROL_WINDOW_SIZE：控制套接字窗口大小。默认情况下，此值为 400 个字节。此定义可以在包含 nx_ftp.h 之前由应用程序进行设置。
- NX_FTP_DATA_WINDOW_SIZE：数据套接字窗口大小。默认情况下，此值为 2048 个字节。此定义可以在包含 nx_ftp.h 之前由应用程序进行设置。
- NX_FTP_TIME_TO_LIVE：指定数据包在被丢弃之前可通过的路由器数目。默认值设置为 0x80，但在包含 nx_ftp.h 之前可以重新定义该值。
- NX_FTP_SERVER_TIMEOUT：指定内部服务将暂停的 ThreadX 时钟周期数。默认值设置为 100，但在包含 nx_ftp.h 之前可以重新定义该值。
- NX_FTP_USERNAME_SIZE：指定客户端提供的 username 中允许的字节数。默认值设置为 20，但在包含 nx_ftp.h 之前可以重新定义该值。
- NX_FTP_PASSWORD_SIZE：指定客户端提供的 password 中允许的字节数。默认值设置为 20，但在包含 nx_ftp.h 之前可以重新定义该值。
- NX_FTP_ACTIVITY_TIMEOUT：指定不活动时保持客户端连接的秒数。默认值设置为 240，但在包含 nx_ftp.h 之前可以重新定义该值。
- NX_FTP_TIMEOUT_PERIOD：指定服务器两次检查客户端非活动状态之间间隔的秒数。默认值设置为 60，但在包含 nx_ftp.h 之前可以重新定义该值。

第 3 章 - Azure RTOS NetX FTP 服务说明

2021/4/29 •

本章按字母顺序提供所有 Azure RTOS NetX FTP 服务说明(下面列出的)。

在以下 API 说明的"返回值"部分，以粗体显示的值不受 NX_DISABLE_ERROR_CHECKING 定义(用于禁用 API 错误检查)影响，而对于非粗体值，则会完全禁用此检查。

- nx_ftp_client_connect:连接到 FTP 服务器
- nx_ftp_client_create:创建 FTP 客户端实例
- nx_ftp_client_delete:删除 FTP 客户端实例
- nx_ftp_client_directory_create:在服务器上创建目录
- nx_ftp_client_directory_default_set:在服务器上设置默认目录
- nx_ftp_client_directory_delete:在服务器上删除目录
- nx_ftp_client_directory_listing_get:从服务器获取目录列表
- nx_ftp_client_directory_listing_continue:从服务器继续获取目录列表
- nx_ftp_client_disconnect:断开与 FTP 服务器的连接
- nx_ftp_client_file_close:关闭客户端文件
- nx_ftp_client_file_delete:删除服务器上的文件
- nx_ftp_client_file_open:打开客户端文件
- nx_ftp_client_file_read:从文件中读取
- nx_ftp_client_file_rename:重命名服务器上的文件
- nx_ftp_client_file_write:写入到文件
- nx_ftp_server_create:创建 FTP 服务器
- nx_ftp_server_delete:删除 FTP 服务器
- nx_ftp_server_start:启动 FTP 服务器
- nx_ftp_server_stop:停止 FTP 服务器

nx_ftp_client_connect

连接到 FTP 服务器

原型

```
UINT nx_ftp_client_connect(NX_FTP_CLIENT *ftp_client_ptr,
        ULONG server_ip, CHAR *username, CHAR *password,
        ULONG wait_option);
```

说明

此服务将以前创建的 FTP 客户端实例连接到所提供 IP 地址的 FTP 服务器。

输入参数

- ftp_client_ptr:指向 FTP 客户端控制块的指针。
- server_ip:FTP 服务器的 IP 地址。
- username:用于身份验证的客户端用户名。
- password:用于身份验证的客户端密码。
- wait_option:定义服务等待 FTP 客户端连接的时长。等待选项定义如下:
  - timeout value:(0x00000001 至 0xFFFFFFFE)

- TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 连接成功。
- **NX_TFTP_EXPECTED_22X_CODE**：(0xDB) 未获得 22X（正常）响应
- **NX_FTP_EXPECTED_23X_CODE**：(0xDC) 未获得 23X（正常）响应
- **NX_FTP_EXPECTED_33X_CODE**：(0xDE) 未获得 33X（正常）响应
- **NX_FTP_NOT_DISCONNECTED**：(0xD4) 客户端已连接。
- NX_PTR_ERROR：(0x07) 指针输入输出无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。
- NX_IP_ADDRESS_ERROR：(0x21) IP 地址无效。

获准方式

线程数

示例

```
/* Connect the FTP Client instance "my_client" to the FTP Server at
    IP address 1.2.3.4. */
status = nx_ftp_client_connect(&my_client, IP_ADDRESS(1,2,3,4), NULL, NULL, 100);

/* If status is NX_SUCCESS an FTP Client instance was successfully
connected to the FTP Server. */
```

nx_ftp_client_create

创建 FTP 客户端实例

原型

```
UINT nx_ftp_client_create(NX_FTP_CLIENT *ftp_client_ptr,
    CHAR *ftp_client_name, NX_IP *ip_ptr, ULONG window_size,
    NX_PACKET_POOL *pool_ptr);
```

说明

此服务可创建 FTP 客户端实例。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- ftp_client_name：FTP 客户端的名称。
- ip_ptr：指向以前所创建 IP 实例的指针。
- window_size：用于此 FTP 客户端 TCP 套接字的播发窗口大小。
- pool_ptr：指向此 FTP 客户端默认数据包池的指针。请注意，最小数据包有效负载必须足够大，以便容纳完整路径和文件名或目录名。

返回值

- **NX_SUCCESS**：(0x00) FTP 客户端创建成功。
- NX_PTR_ERROR：(0x16) FTP、IP 指针或数据包池指针无效。密码指针。

获准方式

初始化和线程

示例

```
/* Create the FTP Client instance "my_client." */
status = nx_ftp_client_create(&my_client, "My Client", &client_ip,
                              2000, &client_pool);

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

nx_ftp_client_delete

删除 FTP 客户端实例

原型

```
UINT nx_ftp_client_delete(NX_FTP_CLIENT *ftp_client_ptr);
```

说明

此服务可删除 FTP 客户端实例。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。

返回值

- **NX_SUCCESS**：(0x00) FTP 客户端删除成功。
- **NX_FTP_NOT_DISCONNECTED**：(0xD4) FTP 客户端删除错误。
- NX_PTR_ERROR：(0x16) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Delete the FTP Client instance "my_client." */
status = nx_ftp_client_delete(&my_client);

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

nx_ftp_client_directory_create

在 FTP 服务器上创建目录

原型

```
UINT nx_ftp_client_directory_create(NX_FTP_CLIENT *ftp_client_ptr,
                        CHAR *directory_name, ULONG wait_option);
```

说明

此服务可在连接到指定 FTP 客户端的 FTP 服务器上创建指定目录。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。

- directory_name：要创建目录的名称。
- wait_option：定义服务等待 FTP 目录创建的时长。等待选项定义如下：
  - timeout value：（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 目录创建成功。
- **NX_FTP_NOT_CONNECTED**：(0xD3) FTP 客户端未连接。
- **NX_FTP_EXPECTED_2XX_CODE**：(0xDA) 未获得 2XX（正常）响应
- NX_PTR_ERROR：(0x07) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Create the directory "my_dir" on the FTP Server connected to
    the FTP Client instance "my_client." */
status = nx_ftp_client_directory_create(&my_client, "my_dir", 200);

/* If status is NX_SUCCESS the directory "my_dir" was successfully
    created. */
```

nx_ftp_client_directory_default_set

在 FTP 服务器上设置默认目录

原型

```
UINT nx_ftp_client_directory_default_set(NX_FTP_CLIENT *ftp_client_ptr,
                                CHAR *directory_path, ULONG wait_option);
```

说明

此服务可在连接到指定 FTP 客户端的 FTP 服务器上设置默认目录。此默认目录仅适用于此客户端的连接。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- directory_path：要设置目录路径的名称。
- wait_option：定义服务等待 FTP 默认目录设置的时长。等待选项定义如下：
  - timeout value：（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 默认设置成功。
- **NX_FTP_NOT_CONNECTED**：(0xD3) FTP 客户端未连接。
- **NX_FTP_EXPECTED_2XX_CODE**：(0xDA) 未获得 2XX（正常）响应
- NX_PTR_ERROR：(0x07) FTP 指针无效。

- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Set the default directory to "my_dir" on the FTP Server connected to
    the FTP Client instance "my_client." */
status = nx_ftp_client_directory_default_set(&my_client, "my_dir", 200);

/* If status is NX_SUCCESS the directory "my_dir" is the default directory. */
```

nx_ftp_client_directory_delete

在 FTP 服务器上删除目录

原型

```
UINT nx_ftp_client_directory_delete(NX_FTP_CLIENT *ftp_client_ptr,
                        CHAR *directory_name, ULONG wait_option);
```

说明

此服务可在连接到指定 FTP 客户端的 FTP 服务器上删除指定目录。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- directory_name：要删除目录的名称。
- wait_option：定义服务等待 FTP 目录删除的时长。等待选项定义如下：
  - timeout value：（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 目录删除成功。
- **NX_FTP_NOT_CONNECTED**：(0xD3) FTP 客户端未连接。
- **NX_FTP_EXPECTED_2XX_CODE**：(0xDA) 未获得 2XX（正常）响应
- NX_PTR_ERROR：(0x07) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Delete directory "my_dir" on the FTP Server connected to
    the FTP Client instance "my_client." */
status = nx_ftp_client_directory_delete(&my_client, "my_dir", 200);

/* If status is NX_SUCCESS the directory "my_dir" is deleted. */
```

nx_ftp_client_directory_listing_get

从 FTP 服务器获取目录列表

原型

```
UINT nx_ftp_client_directory_listing_get(NX_FTP_CLIENT *ftp_client_ptr,
                            CHAR *directory_name, NX_PACKET **packet_ptr,
                            ULONG wait_option);
```

说明

此服务可在连接到指定 FTP 客户端的 FTP 服务器上获取指定目录的内容。提供的数据包指针将包含一个或多个目录项。各目录项用 <cr/lf\combination 进行分隔。用户应调用 nx_ftp_client_directory_listing_continue 来完成目录获取操作。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- directory_name：要获取其内容的目录的名称。
- packet_ptr：指向目标数据包指针的指针。如果成功，数据包有效负载将包含一个或多个目录项。
- wait_option：定义服务等待 FTP 目录列出的时长。等待选项定义如下：
  - timeout value：(0x00000001 至 0xFFFFFFFE)
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 目录列出成功。
- **NX_FTP_NOT_CONNECTED**：(0xD3) FTP 客户端未连接。
- **NX_NOT_ENABLED**：(0x14) 服务 (IPv6) 未启用
- **NX_FTP_EXPECTED_1XX_CODE**：(0xD9) 未获得 1XX（正常）响应
- **NX_FTP_EXPECTED_2XX_CODE**：(0xDA) 未获得 2XX（正常）响应
- NX_PTR_ERROR：(0x07) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Get the contents of directory "my_dir" on the FTP Server connected to
    the FTP Client instance "my_client." */
status = nx_ftp_client_directory_listing_get(&my_client, "my_dir", &my_packet,
                                             200);

/* If status is NX_SUCCESS, one or more entries of the directory "my_dir"
    can be found in "my_packet". */
```

nx_ftp_client_directory_listing_continue

从 FTP 服务器继续获取目录列表

原型

```
UINT nx_ftp_client_directory_listing_continue(NX_FTP_CLIENT
                    *ftp_client_ptr, NX_PACKET **packet_ptr,
                    ULONG wait_option);
```

说明

此服务可在连接到指定 FTP 客户端的 FTP 服务器上继续获取指定目录的内容。此服务应紧随 nx_ftp_client_directory_listing_get 调用之后。如果成功，提供的数据包指针将包含一个或多个目录项。用户应在收到 NX_FTP_END_OF_LISTING 状态之前，调用此例程。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- packet_ptr：指向目标数据包指针的指针。如果成功，数据包有效负载将包含一个或多个用 <cr/lf> 分隔的目录项。
- wait_option：定义服务等待 FTP 目录列出的时长。等待选项定义如下：
  - timeout value：（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 目录列出成功。
- **NX_FTP_END_OF_LISTING**：(0xD8) 此目录中没有更多条目。
- **NX_FTP_NOT_CONNECTED**：(0xD3) FTP 客户端未连接。
- **NX_FTP_EXPECTED_2XX_CODE**：(0xDA) 未获得 2XX（正常）响应
- NX_PTR_ERROR：(0x07) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Continue getting the contents of directory "my_dir" on the FTP Server
    connected to the FTP Client instance "my_client." */
status = nx_ftp_client_directory_listing_continue(&my_client, &my_packet,
                                                  200);

/* If status is NX_SUCCESS, one or more entries of the directory "my_dir"
    can be found in "my_packet". */
```

nx_ftp_client_disconnect

断开与 FTP 服务器的连接

原型

```
UINT nx_ftp_client_disconnect(NX_FTP_CLIENT *ftp_client_ptr,
                              ULONG wait_option);
```

说明

此服务可断开以前建立的 FTP 服务器与指定 FTP 客户端的连接。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- wait_option：定义服务等待 FTP 客户端断开连接的时长。等待选项定义如下：
  - timeout value：（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 断开连接成功。
- **NX_FTP_NOT_CONNECTED**：(0xD3) FTP 客户端未连接。
- **NX_FTP_EXPECTED_2XX_CODE**：(0xDA) 未获得 2XX（正常）响应
- NX_PTR_ERROR：(0x07) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Disconnect "my_client" from the FTP Server. */
status = nx_ftp_client_disconnect(&my_client, 200);

/* If status is NX_SUCCESS, "my_client" has been disconnected. */
```

nx_ftp_client_file_close

关闭客户端文件

原型

```
UINT nx_ftp_client_file_close(NX_FTP_CLIENT *ftp_client_ptr,
                              ULONG wait_option);
```

说明

此服务可关闭以前在 FTP 服务器上打开的文件。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- wait_option：定义服务等待 FTP 客户端关闭文件的时长。等待选项定义如下：
  - timeout value：（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 文件关闭成功。
- **NX_FTP_NOT_CONNECTED**：(0xD3) FTP 客户端未连接。
- **NX_FTP_NOT_OPEN (0xD5)** ：未打开文件；无法关闭
- **NX_FTP_EXPECTED_2XX_CODE**：(0xDA) 未获得 2XX（正常）响应
- NX_PTR_ERROR：(0x07) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Close previously opened file of client "my_client" on the FTP Server. */
status = nx_ftp_client_file_close(&my_client, 200);

/* If status is NX_SUCCESS, the file opened previously in the "my_client" FTP
    connection has been closed. */
```

nx_ftp_client_file_delete

删除 FTP 服务器上的文件

原型

```
UINT nx_ftp_client_file_delete(NX_FTP_CLIENT *ftp_client_ptr,
                        CHAR *file_name, ULONG wait_option);
```

说明

此服务可删除 FTP 服务器上的指定文件。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- file_name：要删除文件的名称。
- wait_option：定义服务等待 FTP 客户端删除文件的时长。等待选项定义如下：
  - timeout value：（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 文件删除成功。
- **NX_FTP_NOT_CONNECTED**：(0xD3) FTP 客户端未连接。
- **NX_FTP_EXPECTED_2XX_CODE**：(0xDA) 未获得 2XX（正常）响应
- NX_PTR_ERROR：(0x07) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Delete the file "my_file.txt" on the FTP Server using the previously
    connected client "my_client." */
status = nx_ftp_client_file_delete(&my_client, "my_file.txt", 200);

/* If status is NX_SUCCESS, the file "my_file.txt" on the FTP Server is
    deleted. */
```

nx_ftp_client_file_open

打开 FTP 服务器上的文件

原型

```
UINT nx_ftp_client_file_open(NX_FTP_CLIENT *ftp_client_ptr,
        CHAR *file_name, UINT open_type, ULONG wait_option);
```

说明

此服务可在以前连接到指定客户端实例的 FTP 服务器上打开指定文件，以供读取或写入。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- file_name：要打开文件的名称。
- open_type：NX_FTP_OPEN_FOR_READ 或 NX_FTP_OPEN_FOR_WRITE。
- wait_option：定义服务等待 FTP 客户端打开文件的时长。等待选项定义如下：
  - timeout value：（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 文件打开成功。
- **NX_OPTION_ERROR**：(0x0A) 打开类型无效。
- **NX_FTP_NOT_CONNECTED**：(0xD3) FTP 客户端未连接。
- **NX_FTP_NOT_CLOSED**：(0xD6) FTP 客户端已打开。
- **NX_NO_FREE_PORTS**：(0x45) 无可分配 TCP 端口
- NX_PTR_ERROR：(0x07) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Open the file "my_file.txt" for reading on the FTP Server using the previously
    connected client "my_client." */
status = nx_ftp_client_file_open(&my_client, "my_file.txt", NX_FTP_OPEN_FOR_READ,
                                 200);

/* If status is NX_SUCCESS, the file "my_file.txt" on the FTP Server is
    open for reading. */
```

nx_ftp_client_file_read

从文件中读取

原型

```
UINT nx_ftp_client_file_read(NX_FTP_CLIENT *ftp_client_ptr,
                NX_PACKET **packet_ptr, ULONG wait_option);
```

说明

此服务可从以前打开的文件中读取数据包。在收到 NX_FTP_END_OF_FILE 状态之前，应重复调用此服务。

请注意，调用方不会为此服务分配数据包。只需要提供指向数据包指针的指针。此服务将更新该数据包指针，以

指向从套接字接收队列中检索到的数据包。如果返回"成功"状态，则表示有可用数据包，并且调用方负责在使用后释放数据包。

如果返回非零状态(错误状态或 NX_FTP_END_OF_FILE)，调用方无法释放数据包。否则，在数据包指针为 NULL 时，会生成错误。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- packet_ptr：从队列中检索到的数据包指针所指向目标的指针。如果成功，数据包数据会包含部分或全部文件。
- wait_option：定义服务等待 FTP 客户端读取文件的时长。等待选项定义如下：
  - timeout value：(0x00000001 至 0xFFFFFFFE)
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 文件读取成功。
- **NX_FTP_NOT_OPEN**：(0xD5) FTP 客户端未打开。
- **NX_FTP_END_OF_FILE**：(0xD7) 文件结尾条件。
- NX_PTR_ERROR：(0x07) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
       NX_PACKET   *my_packet;

/* Read a packet of data from the file "my_file.txt" that was previously opened
    from the client "my_client." */

status = nx_ftp_client_file_read(&my_client, &my_packet, 200);

        /* Check status.  */
        if (status != NX_SUCCESS)
        {
            error_counter++;
        }
        else
        {
            /* Release packet when done with it. */
            nx_packet_release(my_packet);
        }

/* If status is NX_SUCCESS, the packet pointer, "my_packet" points to the packet
that contains the next bytes from the file. If the file is completely
downloaded, an NX_FTP_END_OF_FILE status is returned (no packet retrieved). */
```

nx_ftp_client_file_rename

重命名 FTP 服务器上的文件

原型

```
UINT nx_ftp_client_file_rename(NX_FTP_CLIENT *ftp_ptr, CHAR *filename,
                               CHAR *new_filename, ULONG wait_option);
```

说明

此服务可重命名 FTP 服务器上的文件。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- filename：文件的当前名称。
- new_filename：文件的新名称。
- wait_option：定义服务等待 FTP 客户端重命名文件的时长。等待选项定义如下：
  - timeout value：（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 文件重命名成功。
- **NX_FTP_NOT_CONNECTED**：(0xD3) FTP 客户端未连接。
- **NX_FTP_EXPECTED_3XX_CODE**：(0XDD) 未获得 3XX（正常）响应
- **NX_FTP_EXPECTED_2XX_CODE**：(0xDA) 未获得 2XX（正常）响应
- NX_PTR_ERROR：(0x07) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Rename file "my_file.txt" to "new_file.txt" on the previously connected
    Client instance "my_client." */

status = nx_ftp_client_file_rename(&my_client, "my_file.txt", "new_file.txt",
                                   200);

/* If status is NX_SUCCESS, the file has been renamed. */
```

nx_ftp_client_file_write

写入到文件

原型

```
UINT nx_ftp_client_file_write(NX_FTP_CLIENT *ftp_client_ptr,
                    NX_PACKET *packet_ptr, ULONG wait_option);
```

说明

此服务可将数据包的数据写入 FTP 服务器上以前打开的文件。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- packet_ptr：包含要写入数据的数据包指针。

- wait_option：定义服务等待 FTP 客户端写入文件的时长。等待选项定义如下：
  - timeout value：（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER：(0xFFFFFFFF) 选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 FTP 服务器响应请求。选择数值 (1-0xFFFFFFFE) 可指定等待 FTP 服务器响应时调用线程保持挂起状态的最大计时器刻度数。

返回值

- **NX_SUCCESS**：(0x00) FTP 文件写入成功。
- **NX_FTP_NOT_OPEN**：(0xD5) FTP 客户端未打开。
- NX_PTR_ERROR：(0x07) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Write the data contained in "my_packet" to the previously opened file
    "my_file.txt". */

status = nx_ftp_client_file_write(&my_client, my_packet, 200);

/* If status is NX_SUCCESS, the file has been written to. */
```

nx_ftp_client_passive_mode_set

启用或禁用被动传输模式

原型

```
UINT nx_ftp_client_passive_mode_set(NX_FTP_CLIENT *ftp_client_ptr,
                                    UINT passive_mode_enabled);
```

说明

如果在以前创建的 FTP 客户端实例中 passive_mode_enabled 输入设置为 NX_TRUE，则此服务会启用被动传输模式，以使读取或写入文件（RETR、STOR）或下载目录列表 (NLST) 的后续调用操作均在传输模式下完成。若要禁用被动传输模式并返回主动传输模式默认行为，请在调用此函数时，将 passive_mode_enabled 输入设置为 NX_FALSE。

输入参数

- ftp_client_ptr：指向 FTP 客户端控制块的指针。
- passive_mode_enabled：
  - 如果设置为 NX_TRUE，则启用被动模式。
  - 如果设置为 NX_FALSE，则禁用被动模式。

返回值

- **NX_SUCCESS**：(0x00) 被动模式设置成功。
- NX_PTR_ERROR：(0x16) FTP 指针无效。
- NX_INVALID_PARAMETERS：(0x4D) 非指针输入无效

获准方式

线程数

示例

```
/* Enable the FTP Client to exchange data with the FTP server in passive mode. */

status = nx_ftp_client_passive_mode_set(&my_client, NX_TRUE);

/* If status is NX_SUCCESS, the FTP client is in passive transfer mode. */
```

nx_ftp_server_create

创建 FTP 服务器

原型

```
UINT nx_ftp_server_create(NX_FTP_SERVER *ftp_server_ptr,
        CHAR *ftp_server_name, NX_IP *ip_ptr,
        FX_MEDIA *media_ptr, VOID *stack_ptr, ULONG stack_size,
        NX_PACKET_POOL *pool_ptr,
        UINT (*ftp_login)(struct NX_FTP_SERVER_STRUCT
                *ftp_server_ptr, ULONG client_ip_address,
                UINT client_port, CHAR *name, CHAR *password,
                CHAR *extra_info),
        UINT (*ftp_logout)(struct NX_FTP_SERVER_STRUCT
                *ftp_server_ptr, ULONG client_ip_address,
                UINT client_port, CHAR *name, CHAR *password,
                CHAR *extra_info));
```

说明

此服务可在以前创建的指定 NetX IP 实例中创建 FTP 服务器实例。请注意，FTP 服务器需要先通过调用 nx_ftp_server_start 启动，然后才能开始运行。

输入参数

- ftp_server_ptr：指向 FTP 服务器控制块的指针。
- ftp_server_name：FTP 服务器的名称。
- ip_ptr：指向关联 NetX IP 实例的指针。请注意，IP 实例只能有一个 FTP 服务器。
- media_ptr：指向关联 FileX 媒体实例的指针。
- stack_ptr：指向内部 FTP 服务器线程堆栈区域内存的指针。
- stack_size：stack_ptr 所指定堆栈区域的大小。
- pool_ptr：指向默认 NetX 数据包池的指针。请注意，池中数据包的有效负载必须足够大，以便容纳最长文件名/路径。
- ftp_login：指向应用程序登录函数的函数指针。此函数用于获取请求连接的客户端的用户名和密码。如果此数据有效，应用程序登录函数应返回 NX_SUCCESS。
- ftp_logout：指向应用程序注销函数的函数指针。此函数用于获取请求断开连接的客户端的用户名和密码。如果此数据有效，应用程序登录函数应返回 NX_SUCCESS。

返回值

- **NX_SUCCESS**：(0x00) FTP 服务器创建成功。
- NX_PTR_ERROR：(0x16) FTP 指针无效。

获准方式

初始化和线程

示例

```
/* Create the FTP Server "my_server" on the IP instance "ip_0" using the
    "ram_disk" media. */

status = nx_ftp_server_create(&my_server, "My Server Name", &ip_0, &ram_disk,
                                stack_ptr, stack_size, &pool_0,
                                my_login, my_logout);

/* If status is NX_SUCCESS, the FTP Server has been created. */
```

nx_ftp_server_delete

删除 FTP 服务器

原型

```
UINT nx_ftp_server_delete(NX_FTP_SERVER *ftp_server_ptr);
```

说明

此服务可删除以前创建的 FTP 服务器实例。

输入参数

- ftp_server_ptr：指向 FTP 服务器控制块的指针。

返回值

- **NX_SUCCESS**：(0x00) FTP 服务器删除成功。
- NX_PTR_ERROR：(0x16) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Delete the FTP Server "my_server". */

status = nx_ftp_server_delete(&my_server);

/* If status is NX_SUCCESS, the FTP Server has been deleted. */
```

nx_ftp_server_start

启动 FTP 服务器

原型

```
UINT nx_ftp_server_start(NX_FTP_SERVER *ftp_server_ptr);
```

说明

此服务可启动以前创建的 FTP 服务器实例。

输入参数

- ftp_server_ptr：指向 FTP 服务器控制块的指针。

返回值

- **NX_SUCCESS**：(0x00) FTP 服务器启动成功。
- NX_PTR_ERROR：(0x16) FTP 指针无效。

获准方式

初始化和线程

示例

```
/* Start the FTP Server "my_server". */

status = nx_ftp_server_start(&my_server);

/* If status is NX_SUCCESS, the FTP Server has been started. */
```

nx_ftp_server_stop

停止 FTP 服务器

原型

```
UINT nx_ftp_server_stop(NX_FTP_SERVER *ftp_server_ptr);
```

说明

此服务可停止以前创建和启动的 FTP 服务器实例。

输入参数

- ftp_server_ptr：指向 FTP 服务器控制块的指针。

返回值

- **NX_SUCCESS**：(0x00) FTP 服务器停止成功。
- NX_PTR_ERROR：(0x16) FTP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

获准方式

线程数

示例

```
/* Stop the FTP Server "my_server". */

status = nx_ftp_server_stop(&my_server);

/* If status is NX_SUCCESS, the FTP Server has been stopped. */
```

第 1 章 - NetX HTTP 简介

2021/4/30 •

超文本传输协议 (Hypertext Transfer Protocol, HTTP) 是设计用于在 Web 上传输内容的协议。HTTP 是一种简单协议，它利用可靠的传输控制协议 (Transmission Control Protocol, TCP) 服务来执行其内容传输功能。因此，HTTP 是高度可靠的内容传输协议。HTTP 是最常用的应用程序协议之一。Web 上的所有操作都利用 HTTP 协议。

HTTP 要求

若要正常运行，NetX HTTP 包要求安装 NetX（版本 5.2 或更高版本）。此外，必须已创建一个 IP 实例，且必须在该 IP 实例上启用 TCP。第 2 章的“小型示例系统”一节中的演示文件将演示如何执行此操作。

NetX HTTP 包的 HTTP 客户端部分没有其他要求。

NetX HTTP 包的 HTTP 服务器部分有几个附加要求。首先，它需要对众所周知的 TCP 端口 80 的完全访问权限，该端口用于处理所有客户端 HTTP 请求。HTTP 服务器还设计为与 FileX 嵌入式文件系统一起使用。如果 FileX 不可用，则用户可将要使用的 FileX 部分移植到自己的环境。本指南后面各节将对此进行讨论。

HTTP 限制

NetX HTTP 协议实现 HTTP 1.0 标准。但是，存在以下限制：

1. 不支持持续连接
2. 不支持请求流水线操作
3. HTTP 服务器支持基本身份验证和 MD5 摘要式身份验证，但不支持 MD5-sess 身份验证。目前，HTTP 客户端仅支持基本身份验证。
4. 不支持内容压缩。
5. 不支持 TRACE、OPTIONS 和 CONNECT 请求。
6. 与 HTTP 服务器或客户端关联的数据包池必须足够大以容纳完整的 HTTP 标头。
7. HTTP 客户端服务仅用于内容传输，此包中未提供任何显示实用工具。

HTTP URL（资源名称）

HTTP 协议设计用于在 Web 上传输内容。请求的内容由统一资源定位器 (URL) 指定。这是每个 HTTP 请求的主要组件。URL 始终以“/”字符开头，通常与 HTTP 服务器上的文件相对应。常见 HTTP 文件扩展名如下所示：

- .htm（或 .html）：超文本标记语言 (Hypertext Markup Language, HTML)
- .txt：ASCII 纯文本
- .gif：二进制 GIF 图像
- .xbm：二进制 Xbitmap 图像

HTTP 客户端请求

HTTP 采用简单机制来请求 Web 内容。基本上来说，成功在众所周知的 TCP 端口 80 上建立连接后，客户端会发出一组标准 HTTP 命令。下面显示了一些基本 HTTP 命令：

- GET resource HTTP/1.0：获取指定的资源

- POST resource HTTP/1.0：获取指定的资源并将附加的输入传递到 HTTP 服务器
- HEAD resource HTTP/1.0：与 GET 类似，但 HTTP 服务器不会返回任何内容
- PUT resource HTTP/1.0：在 HTTP 服务器上放置资源
- DELETE resource HTTP/1.0：删除服务器上的资源

这些 ASCII 命令由 Web 浏览器和 NetX HTTP 客户端服务在内部生成，以通过 HTTP 服务器执行 HTTP 操作。

> **NOTE**
>
> HTTP 客户端应用程序的连接端口默认设置为端口 80。但是，应用程序可以在运行时使用 nx_http_client_set_connect_port 服务来更改与 HTTP 服务器的连接端口。有关此服务的更多详细信息，请参阅第 4 章。此服务适用于偶尔将备用端口用于客户端连接的 Web 服务器。

HTTP 服务器响应

HTTP 服务器还使用众所周知的 TCP 端口 80 来发送客户端命令响应。HTTP 服务器在处理客户端命令后，会返回 ASCII 响应字符串，其中包含 3 位数状态代码。HTTP 客户端软件使用数值响应来确定操作是成功还是失败。下面列出了对客户端命令的各种 HTTP 服务器响应：

- 200：请求成功
- 400：请求的格式不正确
- 401：未经授权的请求，客户端需要发送身份验证
- 404：找不到在请求中指定的资源
- 500：内部 HTTP 服务器错误
- 501：HTTP 服务器未实现该请求
- 502：服务不可用

例如，当放置文件“test.htm”的客户端请求成功时，响应消息为“HTTP/1.0 200 OK”。

HTTP 通信

如前所述，HTTP 服务器使用众所周知的 TCP 端口 80 来处理客户端请求。HTTP 客户端可以使用任何可用的 TCP 端口。HTTP 事件的一般顺序如下：

HTTP GET 请求：

1. 客户端向服务器端口 80 发起 TCP 连接。
2. 客户端发送“GET resource HTTP/1.0”请求（以及其他标头信息）。
3. 服务器生成“HTTP/1.0 200 OK”消息和附加信息，后面紧跟资源内容（如果有）。
4. 服务器断开连接。
5. 客户端断开连接。

HTTP PUT 请求：

1. 客户端向服务器端口 80 发起 TCP 连接。
2. 客户端发送“PUT resource HTTP/1.0”请求，以及其他标头信息，后跟资源内容。
3. 服务器生成“HTTP/1.0 200 OK”消息和附加信息，后面紧跟资源内容。
4. 服务器断开连接。
5. 客户端断开连接。

NOTE

如前所述，对于使用备用端口连接到客户端的 Web 服务器，HTTP 客户端可以使用 nx_http_client_set_connect_port 将默认连接端口从 80 更改为其他端口。

HTTP 身份验证

HTTP 身份验证是可选的，并非所有 Web 请求都需要。身份验证有两种形式，即基本身份验证和摘要式身份验证。基本身份验证相当于许多协议中的名称和密码身份验证。在 HTTP 基本身份验证中，名称和密码采用 base64 格式进行连接和编码。基本身份验证的主要缺点是，名称和密码会在请求中公开传输。这样，名称和密码容易被盗用。摘要式身份验证从不在请求中传输名称和密码，因此解决了这个问题。相反，此身份验证会使用某种算法根据名称、密码和其他信息派生一个 128 位的密钥或摘要。NetX HTTP 服务器支持标准 MD5 摘要算法。

何时需要身份验证？ 基本上来说，请求的资源是否需要身份验证由 HTTP 服务器决定。如果需要身份验证，并且客户端请求不包含适当的身份验证，则服务器会向客户端发送"HTTP/1.0 401 Unauthorized"响应和所需的身份验证类型。因此，客户端需要构建具有适当身份验证的新请求。

HTTP 身份验证回调

如前所述，HTTP 身份验证是可选的，并非所有 Web 传输都需要。此外，身份验证通常与资源相关。访问服务器上的某些资源需要身份验证，而访问其中的另一些资源不需要身份验证。使用 NetX HTTP 服务器包，应用程序可以指定（通过 nx_http_server_create 调用）身份验证回调例程，以便在开始处理每个 HTTP 客户端请求时进行调用。

回调例程可向 NetX HTTP 服务器提供与资源关联的用户名、密码和领域字符串，并返回所需的身份验证类型。如果资源不需要身份验证，则身份验证回调应返回 NX_HTTP_DONT_AUTHENTICATE 的值。如果指定的资源需要基本身份验证，则此例程应返回 NX_HTTP_BASIC_AUTHENTICATE。最后，如果资源需要 MD5 摘要式身份验证，则回调例程应返回 NX_HTTP_DIGEST_AUTHENTICATE。如果 HTTP 服务器提供的所有资源都不需要身份验证，则不需要回调，并且可以向 nx_http_server_create 调用提供空指针。

应用程序身份验证回调例程的格式很简单，定义如下：

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

输入参数定义如下：

- request_type - 指定 HTTP 客户端请求，有效请求定义为：
  - 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 - 请求的特定资源。
- name - 指向所需用户名的指针。
- password - 指向所需密码的指针。
- realm - 指向此身份验证领域的指针。

身份验证例程的返回值指定是否需要身份验证。如果身份验证回调例程返回 NX_HTTP_DONT_AUTHENTICATE，则不使用 name 指针、password 指针和 realm 指针。否则，HTTP 服务器开发人员必须确保 nx_http_server.h 中

定义的 NX_HTTP_MAX_USERNAME 和 NX_HTTP_MAX_PASSWORD 足够大，以便容纳身份验证回调中指定的用户名和密码。这两个选项都默认设置为 20 个字符。

HTTP 用户名/密码无效回调

如果 HTTP 服务器收到的客户端请求中的用户名和密码组合无效，则会调用 NetX HTTP 服务器中可选的 HTTP 用户名/密码无效回调。如果 HTTP 服务器应用程序向 HTTP 服务器中注册了回调，则会在 nx_http_server_get_process、nx_http_server_put_process 或 nx_http_server_delete_process 中的基本身份验证或摘要式身份验证失败时调用该回调。

为了向 HTTP 服务器注册回调，NetX HTTP 服务器中定义了以下服务。

```
UINT nx_http_server_invalid_userpassword_notify_set (NX_HTTP_SERVER *http_server_ptr,
                                                     UINT *invalid_username_password_callback)
                                                     (CHAR *resource,
                                                     ULONG *client_nx_address,
                                                     UINT request_type));
```

请求类型定义如下：

- 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 插入 GMT 日期标头回调

NetX HTTP 服务器提供用于在其响应消息中插入日期标头的可选回调。当 HTTP 服务器响应 put 或 get 请求时，将调用此回调

为了向 HTTP 服务器注册 GMT 日期回调，NetX HTTP 服务器中定义了以下服务。

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

NX_HTTP_SERVER_DATE 数据类型定义如下：

```
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 缓存信息获取回调

HTTP 服务器提供用于从 HTTP 应用程序请求特定资源的最长期限和日期的回调。此信息用于确定 HTTP 服务器是否在对客户端 Get 请求的响应中发送整页。如果在客户端请求中找不到“If-Modified-Since”，或者它与获取缓存回调所返回的“Last-Modified”日期不匹配，则会发送整页。

为了向 HTTP 服务器注册回调，定义了以下服务：

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

HTTP 多部分支持

多用途 Internet 邮件扩展 (MIME) 最初适用于 SMTP 协议，但现已适用于 HTTP。MIME 允许同一消息中包含混合的消息类型（例如，图像/jpg 和文本/纯文本）。NetX HTTP 服务器添加了一些服务，用于确定来自客户端的包含 MIME 的 HTTP 消息中的内容类型。若要启用 HTTP 多部分支持并使用这些服务，必须定义配置选项 NX_HTTP_MULTIPART_ENABLE。

```
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);
```

有关使用这些服务的更多详细信息，请参阅第 3 章“HTTP 服务说明”中的说明。

HTTP 多线程支持

用户可以同时从多个线程调用 NetX HTTP 客户端服务。但在同一个线程中，对特定 HTTP 客户端实例的读取或写入请求应按顺序发出。

HTTP RFC

NetX HTTP 符合 RFC1945“超文本传输协议 1.0”、RFC 2581“TCP 拥塞控制”、RFC 1122“Internet 主机要求”和相关的 RFC。

第 2 章 - NetX HTTP 的安装和使用

2021/4/29 •

本章介绍与安装、设置和使用 NetX HTTP 组件相关的各种问题。

产品分发

可以从我们的公共源代码存储库获取 Azure RTOS NetX，网址为：https://github.com/azure-rtos/netx。

- nx_http_client.h：用于 NetX 的 HTTP 客户端的头文件
- nx_http_server.h：用于 NetX 的 HTTP 服务器的头文件
- nx_http_client.c：用于 NetX 的 HTTP 客户端的 C 源文件
- nx_http_server.c：用于 NetX 的 HTTP 服务器的 C 源文件
- nx_md5.c：MD5 摘要算法
- filex_stub.h：存根文件（如果 FileX 不存在）
- nx_http.pdf：NetX HTTP 的说明
- demo_netx_http.c：NetX HTTP 演示

HTTP 安装

若要使用 NetX HTTP，应将之前提到的全部分发文件复制到安装了 NetX 的目录中。例如，如果 NetX 安装在目录"\threadx\arm7\green"中，则应将 NetX HTTP 客户端应用程序的 nx_http_client.h 和 nx_http_client.c、NetX HTTP 服务器应用程序的 nx_http_server.h 和 nx_http_server.c，以及 nx_md5.c 复制到此目录中。对于演示的"RAM 驱动程序"应用程序，应将 NetX HTTP 客户端和服务器文件复制到相同的目录中。

使用 HTTP

NetX HTTP 易于使用。基本上来说，应用程序代码必须包括 nx_http_client.h 和/或 nx_http_server.h。之后，为了分别使用 ThreadX、FileX 和 NetX，还需要包括 tx_api.h、fx_api.h 和 nx_api.h。应用程序代码包含 HTTP 头文件后，就能够执行本指南后面部分指定的 HTTP 函数调用。应用程序还必须在生成过程中包括 nx_http_client.c、nx_http_server.c 和 md5.c。这些文件的编译方式必须与其他应用程序文件相同，并且其对象窗体必须与应用程序的文件链接起来。这就是使用 NetX HTTP 所需的全部内容。

> **NOTE**
> 如果在生成过程中未指定 NX_HTTP_DIGEST_ENABLE，则无需将 md5.c 文件添加到应用程序中。同样，如果不需要 HTTP 客户端功能，则可以省略 nx_http_client.c 文件。

> **NOTE**
> 由于 HTTP 采用 NetX TCP 服务，因此在使用 HTTP 之前，必须通过 nx_tcp_enable 调用来启用 TCP。

小型示例系统

下面的图 1.1 举例说明了 NetX HTTP 是多么易于使用。在此示例中，在第 8 行引入了 HTTP include 文件 nx_http_client.h 和 nx_http_server.h。接下来，在第 131 行的"tx_application_define"中创建了 HTTP 服务器。

NOTE

HTTP 服务器控制块“Server”以前在第 25 行定义为全局变量。

创建成功后，将在第 136 行启动 HTTP 服务器。在第 149 行创建 HTTP 客户端。最后，客户端在第 157 行写入文件，并在第 195 行读回文件。

```
/* This is a small demo of HTTP on the high-performance NetX TCP/IP stack.
This demo relies on ThreadX, NetX, and FileX to show a simple HTML
transfer from the client and then back from the server. */

#include "tx_api.h"
#include "fx_api.h"
#include "nx_api.h"
#include "nx_http_client.h"
#include "nx_http_server.h"
#define     DEMO_STACK_SIZE     4096

/* Define the ThreadX and NetX object control blocks... */

TX_THREAD         thread_0;
TX_THREAD         thread_1;
NX_PACKET_POOL    pool_0;
NX_PACKET_POOL    pool_1;
NX_IP             ip_0;
NX_IP             ip_1;
FX_MEDIA          ram_disk;

/* Define HTTP objects. */

NX_HTTP_SERVER    my_server;
NX_HTTP_CLIENT    my_client;

/* Define the counters used in the demo application... */

ULONG             error_counter;

/* Define the RAM disk memory. */

UCHAR             ram_disk_memory[32000];

/* Define function prototypes. */

void     thread_0_entry(ULONG thread_input);
VOID     _fx_ram_driver(FX_MEDIA *media_ptr) ;
void     _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);
UINT     authentication_check(NX_HTTP_SERVER *server_ptr, UINT request_type,
                              CHAR *resource, CHAR **name, CHAR **password,
                              CHAR **realm);

/* 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 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;

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

    /* Create the main thread. */
    tx_thread_create(&thread_0, "thread 0", thread_0_entry, 0,
                    pointer, DEMO_STACK_SIZE,
                    2, 2, TX_NO_TIME_SLICE, TX_AUTO_START);
                    pointer = pointer + DEMO_STACK_SIZE;

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

    /* Create packet pool. */
    nx_packet_pool_create(&pool_0, "NetX Packet Pool 0",
                        600, pointer, 8192);
                        pointer = pointer + 8192;

    /* Create an IP instance. */
    nx_ip_create(&ip_0, "NetX IP Instance 0", IP_ADDRESS(1, 2, 3, 4),
                0xFFFFFF00UL, &pool_0, _nx_ram_network_driver,
                pointer, 4096, 1);
                pointer = pointer + 4096;

    /* Create another packet pool. */
    nx_packet_pool_create(&pool_1, "NetX Packet Pool 1", 600, pointer, 8192);
                        pointer = pointer + 8192;

    /* Create another IP instance. */
    nx_ip_create(&ip_1, "NetX IP Instance 1", IP_ADDRESS(1, 2, 3, 5),
                0xFFFFFF00UL, &pool_1, _nx_ram_network_driver,
                pointer, 4096, 1);
                pointer = pointer + 4096;

    /* Enable ARP and supply ARP cache memory for IP Instance 0. */
    nx_arp_enable(&ip_0, (void *) pointer, 1024);
                 pointer = pointer + 1024;

    /* Enable ARP and supply ARP cache memory for IP Instance 1. */
    nx_arp_enable(&ip_1, (void *) pointer, 1024);
                 pointer = pointer + 1024;

    /* Enable TCP processing for both IP instances. */
    nx_tcp_enable(&ip_0);
    nx_tcp_enable(&ip_1);

    /* Open the RAM disk. */
                 _fx_ram_driver, ram_disk_memory, pointer, 4096) ;
                 pointer += 4096;

    /* Create the NetX HTTP Server. */
    nx_http_server_create(&my_server, "My HTTP Server", &ip_1, &ram_disk,
                        pointer, 4096, &pool_1, authentication_check, NX_NULL);
```

```
                        pointer = pointer + 4096;

    /* Start the HTTP Server. */
    nx_http_server_start(&my_server);
}

/* Define the test thread. */
void     thread_0_entry(ULONG thread_input)
{

NX_PACKET     *my_packet;
UINT          status;

    /* Create an HTTP client instance. */
    status = nx_http_client_create(&my_client, "My Client", &ip_0,
                                   &pool_0, 600);

    /* Check status. */
    if (status)
        error_counter++;

    /* Prepare to send the simple 103-byte HTML file to the Server. */
    status = nx_http_client_put_start(&my_client, IP_ADDRESS(1,2,3,5),
                                      "/test.htm", "name", "password", 103, 50);
    /* Check status. */
    if (status)
        error_counter++;

    /* Allocate a packet. */
    status = nx_packet_allocate(&pool_0, &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,
                        &pool_0, NX_WAIT_FOREVER);
    nx_packet_data_append(my_packet,
                        "<HEAD><TITLE>NetX HTTP Test</TITLE></HEAD>\r\n", 44,
                        &pool_0, NX_WAIT_FOREVER);
    nx_packet_data_append(my_packet, "<BODY>\r\n", 8,
                        &pool_0, NX_WAIT_FOREVER);
    nx_packet_data_append(my_packet, "<H1>NetX Test Page</H1>\r\n", 25,
                        &pool_0, NX_WAIT_FOREVER);
    nx_packet_data_append(my_packet, "</BODY>\r\n", 9,
                        &pool_0, NX_WAIT_FOREVER);
    nx_packet_data_append(my_packet, "</HTML>\r\n", 9,
                        &pool_0, 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)
        error_counter++;

    /* Now GET the file back! */
    status = nx_http_client_get_start(&my_client, IP_ADDRESS(1,2,3,5),
                                      "/test.htm", NX_NULL, 0, "name",
                                      "password", 50);

    /* Check status. */
    if (status)
        error_counter++;

    /* Get a packet. */
    status = nx_http_client_get_packet(&my_client, &my_packet, 20);
```

```
    /* Check for an error. */
    if ((status) || (my_packet -> nx_packet_length != 103))
        error_counter++;

    /* Check to see if we have a packet. */
    if (status == NX_SUCCESS)
    {

        /* Yes, release it! */
        nx_packet_release(my_packet);
    }

    /* Flush the media. */
     fx_media_flush(&ram_disk);
}
```

图 1.1 与 NetX 配合使用的 HTTP 的示例

配置选项

生成 NetX HTTP 时，有多个配置选项可用。下面是所有选项的列表，其中包含每个选项的详细说明。此处列出了默认值，但你可以在包括 nx_http_client.h 和 nx_http_server.h 之前重新定义这些值：

- NX_DISABLE_ERROR_CHECKING：如果定义此选项，则会删除基本的 HTTP 错误检查。通常会在调试应用程序后使用此选项。
- NX_HTTP_SERVER_PRIORITY：HTTP 服务器线程的优先级。默认情况下，此值定义为 16，表示将优先级指定为 16。
- NX_HTTP_NO_FILEX：如果定义了此选项，则可为 FileX 依赖项提供存根。如果定义了此选项，则无需进行任何更改，HTTP 客户端即可正常工作。HTTP 服务器将需要进行修改，或者用户必须创建几个 FileX 服务，FTP 服务器才能正常工作。
- NX_HTTP_TYPE_OF_SERVICE：HTTP TCP 请求所需的服务类型。默认情况下，此值定义为 NX_IP_NORMAL，表示正常的 IP 数据包服务。
- NX_HTTP_SERVER_THREAD_TIME_SLICE：在向相同优先级的线程让步之前，服务器线程获允运行的计时器计时周期数。默认值为 2。
- NX_HTTP_FRAGMENT_OPTION：为 HTTP TCP 请求启用分段。默认情况下，此值为 NX_DONT_FRAGMENT，表示禁用 HTTP TCP 分段。
- NX_HTTP_SERVER_WINDOW_SIZE：服务器套接字窗口大小。默认情况下，此值为 2048 个字节。
- NX_HTTP_TIME_TO_LIVE：指定数据包在被丢弃之前可通过的路由器数目。默认值设置为 0x80。
- NX_HTTP_SERVER_TIMEOUT：指定内部服务将暂停的 ThreadX 时钟周期数。默认值设置为 10 秒 (10 * NX_IP_PERIODIC_RATE)。
- NX_HTTP_SERVER_TIMEOUT_ACCEPT：指定内部服务在内部 nx_tcp_server_socket_accept 调用中将挂起的 ThreadX 时钟周期数。默认值设置为 (10 * NX_IP_PERIODIC_RATE)。
- NX_HTTP_SERVER_TIMEOUT_DISCONNECT：指定内部服务在内部 nx_tcp_socket_disconnect 调用中将挂起的 ThreadX 时钟周期数。默认值设置为 10 秒 (10 * NX_IP_PERIODIC_RATE)。
- NX_HTTP_SERVER_TIMEOUT_RECEIVE：指定内部服务在内部 nx_tcp_socket_receive 调用中将挂起的 ThreadX 时钟周期数。默认值设置为 10 秒 (10 * NX_IP_PERIODIC_RATE)。
- NX_HTTP_SERVER_TIMEOUT_SEND：指定内部服务在内部 nx_tcp_socket_send 调用中将挂起的 ThreadX 时钟周期数。默认值设置为 10 秒 (10 * NX_IP_PERIODIC_RATE)。
- NX_HTTP_MAX_HEADER_FIELD：指定 HTTP 标头字段的最大大小。默认值为 256。
- NX_HTTP_MULTIPART_ENABLE：如果已定义，则允许 HTTP 服务器支持多部分 HTTP 请求。
- NX_HTTP_SERVER_MAX_PENDING：指定可以排队等待 HTTP 服务器的连接数。默认值设置为 5。
- NX_HTTP_MAX_RESOURCE：指定客户端提供的"资源名称"中允许包含的字节数。默认值设置为 40。
- NX_HTTP_MAX_NAME：指定客户端提供的"用户名"中允许包含的字节数。默认值设置为 20。
- NX_HTTP_MAX_PASSWORD：指定客户端提供的"密码"中允许包含的字节数。默认值设置为 20。

- NX_HTTP_SERVER_MIN_PACKET_SIZE：指定在创建服务器时指定的池中的数据包的最小大小。为了确保完整的 HTTP 标头可以包含在一个数据包中，需要提供最小大小。默认值设置为 600。
- NX_HTTP_CLIENT_MIN_PACKET_SIZE：指定在创建客户端时指定的池中的数据包的最小大小。为了确保完整的 HTTP 标头可以包含在一个数据包中，需要提供最小大小。默认值设置为 300。
- NX_HTTP_SERVER_RETRY_SECONDS：设置服务器套接字重新传输超时值（以秒为单位）。默认值设置为 2。
- NX_HTTP_SERVER_RETRY_MAX：这会设置服务器套接字上的最大重新传输次数。默认值设置为 10。
- NX_HTTP_SERVER_RETRY_SHIFT：此值用于设置下一个重新传输超时值。系统将当前超时值乘以到目前为止的重新传输次数，再按套接字超时移位值进行移位。默认值设置为 1，可将超时值加倍。
- NX_HTTP_ SERVER_TRANSMIT_QUEUE_DEPTH：这会指定可在服务器套接字重新传输队列中排队的最大数据包数量。如果排队的数据包数量达到此数量，则无法再发送更多数据包，除非释放一个或多个排队的数据包。默认值设置为 20。

第 3 章 - NetX HTTP 服务的说明

2021/4/29 •

本章按字母顺序介绍了所有 NetX HTTP 服务（如下所列）。

在以下 API 说明中的"返回值"部分，以粗体显示的值不受用于禁用 API 错误检查的 NX_DISABLE_ERROR_CHECKING 定义影响，而不以粗体显示的值则已完全禁用 。

HTTP 客户端服务：

- nx_http_client_create：创建 HTTP 客户端实例
- nx_http_client_delete：删除 HTTP 客户端实例
- nx_http_client_get_start：启动 HTTP GET 请求
- nx_http_client_get_start_extended：启动 HTTP GET 请求
- nx_http_client_get_packet：获取下一个资源数据包
- nx_http_client_put_start：启动 HTTP PUT 请求
- nx_http_client_put_start_extended：启动 HTTP PUT 请求
- nx_http_client_put_packet：发送下一个资源数据包
- nx_http_client_set_connect_port：更改用于连接到 HTTP 服务器的端口

HTTP 服务器服务：

- nx_http_server_cache_info_callback_set：设置回调以检索指定 URL 的期限和上次修改日期
- nx_http_server_callback_data_send：从回调函数发送 HTTP 数据
- nx_http_server_callback_generate_response_header：在回调函数中创建响应头
- nx_http_server_callback_generate_response_header_extended：在回调函数中创建响应头
- nx_http_server_callback_packet_send：从 HTTP 回调发送 HTTP 数据包
- nx_http_server_callback_response_send：从回调函数发送响应
- nx_http_server_callback_response_send_extended：从回调函数发送响应
- nx_http_server_content_get：获取请求中的内容
- nx_http_server_content_get_extended：获取请求中的内容；支持空（零内容长度）请求
- nx_http_server_content_length_get：获取请求中的内容长度
- nx_http_server_content_length_get_extended：获取请求中的内容长度；支持空（零内容长度）请求
- nx_http_server_create：创建 HTTP 服务器实例
- nx_http_server_delete：删除 HTTP 服务器实例
- nx_http_server_get_entity_content：返回 URL 中实体内容的大小和位置
- nx_http_server_get_entity_header：将 URL 实体头提取到指定的缓冲区中
- nx_http_server_gmt_callback_set：设置回调以检索 GMT 日期和时间
- nx_http_server_invalid_userpassword_notify_set：设置回调以便在客户端请求中收到无效用户名和密码时发出通知
- nx_http_server_mime_maps_additional_set：为 HTML 定义其他 mime 映射
- nx_http_server_packet_content_find：提取 HTTP 头中的内容长度，并设置指向内容数据开始位置的指针
- nx_http_server_packet_get：直接接收客户端数据包
- nx_http_server_param_get：获取请求中的参数
- nx_http_server_query_get：获取请求中的查询
- nx_http_server_start：启动 HTTP 服务器
- nx_http_server_stop：停止 HTTP 服务器

- nx_http_server_type_get：从头中提取 HTTP 类型，例如 text/plain
- nx_http_server_type_get_extended：从头中提取 HTTP 类型，例如 text/plain
- nx_http_server_digest_authenticate_notify_set：设置摘要身份验证回调函数
- nx_http_server_authentication_check_set：设置身份验证检查回调函数

nx_http_client_create

创建 HTTP 客户端实例

原型

```
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);
```

说明

此服务在指定的 IP 实例上创建 HTTP 客户端实例。

输入参数

- client_ptr：指向 HTTP 客户端控制块的指针。
- client_name：HTTP 客户端实例的名称。
- ip_ptr：指向 IP 实例的指针。
- pool_ptr：指向默认数据包池的指针。请注意，此池中的数据包必须具有足够大的有效负载才能处理完整响应头。这是通过 nx_http_client.h 中的 NX_HTTP_CLIENT_MIN_PACKET_SIZE 定义的。
- window_size：客户端 TCP 套接字接收窗口的大小。

返回值

- NX_SUCCESS：(0x00) 已成功创建 HTTP 客户端
- NX_PTR_ERROR：(0x16) HTTP、ip_ptr 或数据包池指针无效
- NX_HTTP_POOL_ERROR：(0xE9) 数据包池中的有效负载大小无效

允许来自

初始化、线程

示例

```
/* 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

删除 HTTP 客户端实例

原型

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

说明

此服务用于删除以前创建的 HTTP 客户端实例。

输入参数

- client_ptr：指向 HTTP 客户端控制块的指针。

返回值

- NX_SUCCESS：(0x00) 已成功删除 HTTP 客户端
- NX_PTR_ERROR (0x16)：HTTP 指针无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

允许来自

线程数

示例

```
/* 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

启动 HTTP GET 请求

原型

```
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);
```

说明

此服务尝试获取以前创建的 HTTP 客户端实例上的“resource”指针指定的资源。如果此例程返回 NX_SUCCESS，则应用程序可以向 nx_http_client_get_packet 发出多次调用，以检索对应于所请求资源内容的数据包。

请注意，资源字符串可以引用本地文件（例如“/index.htm”）；或者如果 HTTP 服务器指示其支持引用 GET 请求，则资源字符串也可以引用其他 URL（例如 `http://abc.website.com/index.htm`）。

此服务已弃用。建议开发人员迁移到 nx_http_client_get_start_extended()

输入参数

- client_ptr：指向 HTTP 客户端控制块的指针。
- ip_address：HTTP 服务器的 IP 地址。
- resource：指向所请求资源的 URL 字符串的指针。
- input_ptr：指向 GET 请求的附加数据的指针。此为可选项。如果有效，则会将指定的输入放在消息的内容区域中，并使用 POST 而不是 GET 操作。
- input_size：input_ptr 指向的可选附加输入中的字节数。
- username：指向用于身份验证的可选用户名的指针。
- password：指向用于身份验证的可选密码的指针。-wait_option：定义服务等待 HTTP 客户端 GET 启动请求的时长。等待选项的定义方式如下：
  - 超时值（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER (0xFFFFFFFF)

选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 HTTP 服务器响应请求。
选择数值 (0x1-0xFFFFFFFE) 可指定等待 HTTP 服务器响应时调用线程保持挂起状态的最大计时器时钟周期数。

返回值

- NX_SUCCESS：(0x00) 已成功发送 HTTP 客户端 GET 启动消息
- NX_HTTP_ERROR：(0xE0) 内部 HTTP 客户端错误
- NX_HTTP_NOT_READY：(0xEA) HTTP 客户端未准备就绪
- NX_HTTP_FAILED：(0xE2) HTTP 客户端与 HTTP 服务器通信时出错。
- NX_HTTP_AUTHENTICATION_ERROR：(0xEB) 名称和/或密码无效。
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

线程数

示例

```
/* 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, sizeof(POST_MESSAGE),
                                   “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. */
```

nx_http_client_get_start_extended

启动 HTTP GET 请求

原型

```
UINT nx_http_client_get_start_extended(NX_HTTP_CLIENT *client_ptr,
     ULONG ip_address, CHAR *resource, UINT resource_length,
     CHAR *input_ptr, UINT input_size, CHAR *username,
     UINT username_length, CHAR *password, UINT password_length,
     ULONG wait_option);
```

说明

此服务尝试获取以前创建的 HTTP 客户端实例上的“resource”指针指定的资源。如果此例程返回 NX_SUCCESS，则应用程序可以向 nx_http_client_get_packet 发出多次调用，以检索对应于所请求资源内容的数据包。

请注意，资源字符串可以引用本地文件（例如“/index.htm”）；或者如果 HTTP 服务器指示其支持引用 GET 请求，则资源字符串也可以引用其他 URL（例如 `http://abc.website.com/index.htm`）。

此服务取代了 nx_http_client_get_start()。它要求调用方指定资源、用户名和密码的长度。

输入参数

- client_ptr：指向 HTTP 客户端控制块的指针。
- ip_address：HTTP 服务器的 IP 地址。
- resource：指向所请求资源的 URL 字符串的指针。
- resource_length：所请求资源的 URL 字符串的长度。
- input_ptr：指向 GET 请求的附加数据的指针。此为可选项。如果有效，则会将指定的输入放在消息的内容区域中，并使用 POST 而不是 GET 操作。
- input_size：input_ptr 指向的可选附加输入中的字节数。
- username：指向用于身份验证的可选用户名的指针。
- username_length：用于身份验证的可选用户名的长度。
- password：指向用于身份验证的可选密码的指针。
- password_length：用于身份验证的可选密码的长度。
- wait_option：定义服务等待 HTTP 客户端 GET 启动请求的时长。等待选项的定义方式如下：
  - 超时值（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER (0xFFFFFFFF)
    选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 HTTP 服务器响应请求。
    选择数值 (0x1-0xFFFFFFFE) 可指定等待 HTTP 服务器响应时调用线程保持挂起状态的最大计时器时钟周期数。

返回值

- NX_SUCCESS：(0x00) 已成功发送 HTTP 客户端 GET 启动消息
- NX_HTTP_ERROR：(0xE0) 内部 HTTP 客户端错误
- NX_HTTP_NOT_READY：(0xEA) HTTP 客户端未准备就绪
- NX_HTTP_FAILED：(0xE2) HTTP 客户端与 HTTP 服务器通信时出错。
- NX_HTTP_AUTHENTICATION_ERROR：(0xEB) 名称和/或密码无效。
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

线程数

示例

```
/* Start the GET operation on the HTTP Client “my_client.”  */
status =  nx_http_client_get_start_extended(&my_client, IP_ADDRESS(1,2,3,5),
                                            “/TEST.HTM”,
                                            9, NX_NULL, 0, “myname”, 6,
                                            “mypassword”, 10, 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_extended(&my_client, IP_ADDRESS(1,2,3,5),
                                            “/TEST.HTM”,
                                            9, POST_MESSAGE, sizeof(POST_MESSAGE),
                                            “myname”, 6, “mypassword”, 10, 1000);

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

nx_http_client_get_packet

获取下一个资源数据包

原型

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

说明

此服务检索前一次 nx_http_client_get_start 调用请求的资源的下一个内容数据包。对此例程的后续调用应在收到 NX_HTTP_GET_DONE 的返回状态之后发出。

输入参数

- client_ptr：指向 HTTP 客户端控制块的指针。
- packet_ptr：包含部分资源内容的数据包指针的目标。
- wait_option：定义服务等待 HTTP 客户端获取数据包的时长。等待选项的定义方式如下：
  - 超时值（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER (0xFFFFFFFF)
    选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 HTTP 服务器响应请求。
    选择数值 (0x1-0xFFFFFFFE) 可指定等待 HTTP 服务器响应时调用线程保持挂起状态的最大计时器时钟周期数。

返回值

- NX_SUCCESS：(0x00) HTTP 客户端获取数据包成功。
- NX_HTTP_GET_DONE：(0xEC) HTTP 客户端获取数据包已完成
- NX_HTTP_NOT_READY：(0xEA) HTTP 客户端不处于 get 模式。
- NX_HTTP_BAD_PACKET_LENGTH：(0xED) 数据包长度无效
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

允许来自

线程数

示例

```
/* 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

启动 HTTP PUT 请求

原型

```
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);
```

说明

此服务尝试使用指定的资源向位于所提供的 IP 地址的 HTTP 服务器发送 PUT 请求。如果此例程成功，应用程序代码应向 nx_http_client_put_packet 例程发出后续调用，以将资源内容实际发送到 HTTP 服务器。

请注意，资源字符串可以引用本地文件（例如"/index.htm"）；或者如果 HTTP 服务器指示其支持引用 PUT 请求，则资源字符串也可以引用其他 URL（例如 `http://abc.website.com/index.htm`）。

此服务已弃用。建议开发人员迁移到 nx_http_client_put_start_extended()。

输入参数

- client_ptr：指向 HTTP 客户端控制块的指针。
- ip_address：HTTP 服务器的 IP 地址。
- resource：指向要发送到服务器的资源的 URL 字符串的指针。
- username：指向用于身份验证的可选用户名的指针。
- password：指向用于身份验证的可选密码的指针。
- total_bytes：正在发送的资源的总字节数。请注意，通过对 nx_http_client_put_packet 进行后续调用发送的所有数据包的总长度必须等于此值。
- wait_option：定义服务等待 HTTP 客户端 PUT 启动的时长。等待选项的定义方式如下：
  - 超时值（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER (0xFFFFFFFF)
    选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 HTTP 服务器响应请求。
    选择数值 (0x1-0xFFFFFFFE) 可指定等待 HTTP 服务器响应时调用线程保持挂起状态的最大计时器时钟周期数。

返回值

- NX_SUCCESS：(0x00) 已成功发送 PUT 请求
- NX_HTTP_USERNAME_TOO_LONG
- (0xF1) 用户名太大，无法放入缓冲区
- NX_HTTP_NOT_READY：(0xEA) HTTP 客户端未准备就绪
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_SIZE_ERROR：(0x09) 资源的总大小无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

允许来自

线程数

示例

```
/* 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. */
```

nx_http_client_put_start_extended

启动 HTTP PUT 请求

原型

```
UINT nx_http_client_put_start_extended(NX_HTTP_CLIENT *client_ptr,
     ULONG ip_address, CHAR *resource, UINT resource_length,
     CHAR *username,UINT username_length, CHAR *password,
     UINT password_length, ULONG total_bytes, ULONG wait_option);
```

说明

此服务尝试使用指定的资源向位于所提供的 IP 地址的 HTTP 服务器发送 PUT 请求。如果此例程成功，应用程序代码应向 nx_http_client_put_packet 例程发出后续调用，以将资源内容实际发送到 HTTP 服务器。

请注意，资源字符串可以引用本地文件（例如"/index.htm"）；或者如果 HTTP 服务器指示其支持引用 PUT 请求，则资源字符串也可以引用其他 URL（例如 `http://abc.website.com/index.htm`）。

此服务取代了 nx_http_client_put_start()。它要求调用方指定资源、用户名和密码的长度。

输入参数

- client_ptr：指向 HTTP 客户端控制块的指针。
- ip_address：HTTP 服务器的 IP 地址。
- resource：指向要发送到服务器的资源的 URL 字符串的指针。
- resource_length：要发送到服务器的资源的 URL 字符串长度。
- username：指向用于身份验证的可选用户名的指针。
- username_length：用于身份验证的可选用户名的长度。
- password：指向用于身份验证的可选密码的指针。
- password_length：用于身份验证的可选密码的长度。
- total_bytes：正在发送的资源的总字节数。请注意，通过对 nx_http_client_put_packet 进行后续调用发送的所有数据包的总长度必须等于此值。
- wait_option：定义服务等待 HTTP 客户端 PUT 启动的时长。等待选项的定义方式如下：
  - 超时值（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER (0xFFFFFFFF)
    选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 HTTP 服务器响应请求。
    选择数值 (0x1-0xFFFFFFFE) 可指定等待 HTTP 服务器响应时调用线程保持挂起状态的最大计时器时钟周期数。

返回值

- NX_SUCCESS：(0x00) 已成功发送 PUT 请求
- NX_HTTP_USERNAME_TOO_LONG：(0xF1) 用户名太大，无法放入缓冲区
- NX_HTTP_NOT_READY：(0xEA) HTTP 客户端未准备就绪
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_SIZE_ERROR：(0x09) 资源的总大小无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

允许来自

线程数

示例

```
/* 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_extended(&my_client, IP_ADDRESS(1, 2, 3, 5),
                                           “/TEST.HTM”, 9, “myname”, 6,
                                           “mypassword”,
                                           10, 20, NX_WAIT_FOREVER);

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

nx_http_client_put_packet

发送下一个资源数据包

原型

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

说明

此服务尝试将下一个资源内容数据包发送到 HTTP 服务器。请注意，应反复调用此例程，直到发送的数据包总长度等于前一次 nx_http_client_put_start() 调用中指定的“total_bytes”为止。

输入参数

- client_ptr：指向 HTTP 客户端控制块的指针。
- packet_ptr：指向要发送到 HTTP 服务器的资源的下一个内容的指针。
- wait_option：定义服务在内部等待处理 HTTP 客户端 PUT 数据包的时长。等待选项的定义方式如下：
  - 超时值（0x00000001 至 0xFFFFFFFE）
  - TX_WAIT_FOREVER (0xFFFFFFFF)
    选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到 HTTP 服务器响应请求。
    选择数值 (0x1-0xFFFFFFFE) 可指定等待 HTTP 服务器响应时调用线程保持挂起状态的最大计时器时钟周期数。

返回值

- NX_SUCCESS：(0x00) 已成功发送 HTTP 客户端数据包。
- NX_HTTP_NOT_READY：(0xEA) HTTP 客户端未准备就绪
- NX_HTTP_REQUEST_UNSUCCESSFUL_CODE：(0xEE) 收到了服务器错误代码。
  NX_HTTP_BAD_PACKET_LENGTH：(0xED) 数据包长度无效
- NX_HTTP_AUTHENTICATION_ERROR：(0xEB) 名称和/或密码无效
- NX_HTTP_INCOMPLETE_PUT_ERROR：(0xEF) 服务器在 PUT 完成之前做出响应
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_INVALID_PACKET：(0x12) 数据包太小，无法用于 TCP 头
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

允许来自

线程数

示例

```
/* 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

设置用于连接到服务器的端口

原型

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

说明

在运行时，此服务在连接到 HTTP 服务器时会将连接端口更改为指定的端口。否则，连接端口默认为 80。必须在 nx_http_client_get_start() 和 nx_http_client_put_start() 之前调用此服务，例如，在 HTTP 客户端连接到服务器时。

输入参数

- client_ptr：指向 HTTP 客户端控制块的指针。
- port：用于连接到服务器的端口。

返回值

- NX_SUCCESS：(0x00) 已成功更改连接端口
- NX_INVALID_PORT：(0x46) 端口超出最大值 (0xFFFF) 或为零。
- NX_PTR_ERROR：(0x07) 指针输入无效

允许来自

线程、初始化

示例

```
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

设置回调以检索 URL 最大期限和日期

原型

```
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));
```

说明

此服务设置调用的回调服务，以获取指定资源的最大期限和上次修改日期。

输入参数

- server_ptr：指向 HTTP 服务器控制块的指针。
- cache_info_get：指向回调的指针
- max_age：指向资源最大期限的指针
- data：指向返回的上次修改日期的指针。

返回值

- NX_SUCCESS：(0x00) 已成功设置回调
- NX_PTR_ERROR (0x07)：指针输入无效

允许来自

初始化

示例

```
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

从回调函数发送数据

原型

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

说明

此服务从应用程序的回调例程发送提供的数据包中的数据。它通常用于发送与 GET/POST 请求关联的动态数据。请注意，如果使用此函数，则回调例程将负责以正确的格式发送整个响应。此外，回调例程必须返回 NX_HTTP_CALLBACK_COMPLETED 状态。

输入参数

- server_ptr：指向 HTTP 服务器控制块的指针。
- data_ptr：指向要发送的数据的指针。
- data_length：要发送的字节数。

返回值

- NX_SUCCESS：(0x00) 已成功发送服务器数据
- NX_PTR_ERROR (0x07)：指针输入无效

允许来自

线程数

示例

```
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

在回调函数中创建响应头

原型

```
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);
```

说明

当 HTTP 服务器响应客户端 get、put 和 delete 请求时，此服务将调用内部函数 nx_http_server_generate_response_header。它适合在 HTTP 服务器回调函数中使用（当 HTTP 服务器应用程序正在设计要向客户端做出的响应时）。

此服务已弃用。建议开发人员迁移到 nxd_http_server_callback_generate_response_header_extended()。

输入参数

- server_ptr：指向 HTTP 服务器控制块的指针。
- packet_pptr：指向分配给消息的数据包指针的指针
- status_code：指示资源的状态。示例：
- NX_HTTP_STATUS_OK

- NX_HTTP_STATUS_MODIFIED
- NX_HTTP_STATUS_INTERNAL_ERROR
- content_length：内容大小，以字节为单位
- content_type：HTTP 的类型，例如“text/plain”
- additional_header：指向附加头文本的指针

返回值

- NX_SUCCESS：(0x00) 已成功创建 HTML 头
- NX_PTR_ERROR (0x07)：指针输入无效

允许来自

线程数

示例

```
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 = sizeof(demotestbuffer) - 1;

/* 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],
             length, 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_generate_response_header_extended

在回调函数中创建响应头

原型

```
UINT nx_http_server_callback_generate_response_header_extended(
     NX_HTTP_SERVER *server_ptr,
     NX_PACKET **packet_pptr,
     CHAR *status_code, UINT status_code_length,
     UINT content_length,
     CHAR *content_type, UINT content_type_length,
     CHAR *additional_header,
     UINT additional_header_length);
```

说明

当 HTTP 服务器响应客户端 get、put 和 delete 请求时，此服务将调用内部函数 nx_http_server_generate_response_header()。它适合在 HTTP 服务器回调函数中使用（当 HTTP 服务器应用程序正在设计要向客户端做出的响应时）。

此服务取代了 nx_http_server_callback_generate_response_header()。此版本向回调函数提供附加长度信息。

输入参数

- server_ptr：指向 HTTP 服务器控制块的指针。
- packet_pptr：指向分配给消息的数据包指针的指针
- status_code：指示资源的状态。示例：
  - NX_HTTP_STATUS_OK
  - NX_HTTP_STATUS_MODIFIED
  - NX_HTTP_STATUS_INTERNAL_ERROR
- status_code：状态代码的长度
- content_length：内容大小，以字节为单位
- content_type：HTTP 的类型，例如“text/plain”
- content_type_length：HTTP 类型的长度
- additional_header：指向附加头文本的指针
- additional_header_length：附加头文本的长度

返回值

- NX_SUCCESS：(0x00) 已成功创建头
- NX_PTR_ERROR (0x07)：指针输入无效

允许来自

线程数

示例

```
  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 = sizeof(demotestbuffer) - 1;

    /* Derive the client request type from the client request. */
       string_length = (ULONG) nx_http_server_type_retrieve(server_ptr, server_ptr >>
                       nx_http_server_request_resource, temp_string,
                       sizeof(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_extended(
               http_server_ptr, &resp_packet_ptr, NX_HTTP_STATUS_OK,
               sizeof(NX_HTTP_STATUS_OK) - 1, length,
               temp_string, string_length, NX_NULL, 0);

    /* 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],
                length, 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

从回调函数发送 HTTP 数据包

原型

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

说明

此服务从 HTTP 回调发送完整的 HTTP 服务器响应。HTTP 服务器将在指定 NX_HTTP_SERVER_TIMEOUT_SEND 的情况下发送数据包。HTTP 头和数据必须追加到数据包。如果返回状态指示错误，则 HTTP 应用程序必须释放该数据包。

回调应返回 NX_HTTP_CALLBACK_COMPLETED。

有关更详细的示例，请参阅“nx_http_server_callback_generate_response_header()”。

输入参数

- server_ptr：指向 HTTP 服务器控制块的指针
- packet_ptr：指向要发送的数据包的指针

返回值

- NX_SUCCESS：(0x00) 已成功发送 HTTP 服务器数据包
- NX_PTR_ERROR (0x07)：指针输入无效

允许来自

线程数

示例

```
/* 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

从回调函数发送响应

原型

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

说明

此服务从应用程序的回调例程发送提供的响应信息。它通常用于发送与 GET/POST 请求关联的自定义响应。请注意，如果使用此函数，则回调例程必须返回 NX_HTTP_CALLBACK_COMPLETED 状态。

此服务已弃用。建议开发人员迁移到 nx_http_server_callback_response_send_extended()。

输入参数

- server_ptr：指向 HTTP 服务器控制块的指针。

- header：指向响应头字符串的指针。
- information：指向信息字符串的指针。
- additional_info：指向附加信息字符串的指针。

返回值

- NX_SUCCESS：(0x00) 已成功发送 HTTP 服务器响应

允许来自

线程数

示例

```
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_callback_response_send_extended

从回调函数发送响应

原型

```
UINT nx_http_server_callback_response_send_extended(
     NX_HTTP_SERVER *server_ptr, CHAR *heade
     UINT header_length, CHAR *information,
     UINT information_length, CHAR *additional_info,
     UINT additional_info_length);
```

说明

此服务从应用程序的回调例程发送提供的响应信息。它通常用于发送与 GET/POST 请求关联的自定义响应。请注意，如果使用此函数，则回调例程必须返回 NX_HTTP_CALLBACK_COMPLETED 状态。

此服务取代了 nx_http_server_callback_response_send()。此版本采用长度信息作为输入自变量。

输入参数

- server_ptr：指向 HTTP 服务器控制块的指针。
- header：指向响应头字符串的指针。
- header_length：响应头字符串的长度。
- information：指向信息字符串的指针。

- information_length：信息字符串的长度。
- additional_info：指向附加信息字符串的指针。
- additional_info_length：附加信息字符串的长度。

返回值

- NX_SUCCESS：(0x00) 已成功发送服务器响应

允许来自

线程数

示例

```
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_extended(server_ptr,
                                                          "HTTP/1.0 404 ", 12,
                                                          "NetX HTTP Server unable to find
                                                          file: ", 38, resource, 9);
        /* Return completion status. */
           return(NX_HTTP_CALLBACK_COMPLETED);
    }
    return(NX_SUCCESS);
}
```

nx_http_server_content_get

从请求获取内容

原型

```
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);
```

说明

此服务尝试从 POST 或 PUT HTTP 客户端请求中检索指定数量的内容。应从创建 HTTP 服务器 (nx_http_server_create()) 期间指定的应用程序请求通知回调调用此服务。

此服务已弃用。建议开发人员迁移到 nx_http_server_content_get_extended()。

输入参数

- server_ptr：指向 HTTP 服务器控制块的指针。
- packet_ptr：指向 HTTP 客户端请求数据包的指针。请注意，请求通知回调不得释放此数据包。
- byte_offset：要在内容区域中偏移的字节数。
- destination_ptr：指向内容目标区域的指针。

- destination_size：目标区域中可用的最大字节数。
- actual_size：指向目标变量的指针，该变量将设置为已复制内容的实际大小。

返回值

- NX_SUCCESS：(0x00) 已成功获取 HTTP 服务器内容
- NX_HTTP_ERROR：(0xE0) HTTP 服务器内部错误
- NX_HTTP_DATA_END：(0xE7) 请求内容结束
- NX_HTTP_TIMEOUT：(0xE1) HTTP 服务器在获取下一个内容数据包时超时
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

允许来自

线程数

示例

```
/* 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

从请求中获取内容/支持零长度内容

原型

```
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);
```

说明

此服务与 nx_http_server_content_get() 几乎完全相同；它尝试从 POST 或 PUT HTTP 客户端请求中检索指定数量的内容。但是，它会将内容长度值为零的请求（“空请求”）作为有效请求进行处理。应从创建 HTTP 服务器 (nx_http_server_create()) 期间指定的应用程序请求通知回调调用此服务。

此服务取代了 nx_http_server_content_get()。此版本要求调用方提供附加长度信息。

输入参数

- server_ptr：指向 HTTP 服务器控制块的指针。
- packet_ptr：指向 HTTP 客户端请求数据包的指针。请注意，请求通知回调不得释放此数据包。
- byte_offset：要在内容区域中偏移的字节数。
- destination_ptr：指向内容目标区域的指针。
- destination_size：目标区域中可用的最大字节数。
- actual_size：指向目标变量的指针，该变量将设置为已复制内容的实际大小。

返回值

- NX_SUCCESS：(0x00) 已成功获取 HTTP 内容
- NX_HTTP_ERROR：(0xE0) HTTP 服务器内部错误
- NX_HTTP_DATA_END：(0xE7) 请求内容结束
- NX_HTTP_TIMEOUT：(0xE1) HTTP 服务器在获取下一个数据包时超时
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

允许来自

线程数

示例

```
/* 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

获取请求中内容的长度

原型

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

说明

此服务尝试在提供的数据包中检索 HTTP 内容长度。如果没有 HTTP 内容，此例程将返回零值。应从创建 HTTP 服务器 (nx_http_server_create()) 期间指定的应用程序请求通知回调调用此服务。

此服务已弃用。建议开发人员迁移到 nx_http_server_content_length_get_extended()。

输入参数

- packet_ptr：指向 HTTP 客户端请求数据包的指针。请注意，请求通知回调不得释放此数据包。

返回值

- 内容长度：出错时将返回零值

允许来自

线程数

示例

```
/* 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

获取请求中内容的长度/支持零值内容长度

原型

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

说明

此服务类似于 nx_http_server_content_length_get()；尝试在提供的数据包中检索 HTTP 内容长度。但是，返回值指示成功完成状态，并且实际长度值是在输入指针 content_length 中返回的。如果没有 HTTP 内容/内容长度 = 0，则此例程仍会返回成功完成状态，并且 content_length 输入指针将指向有效长度（零）。应从创建 HTTP 服务器 (nx_http_server_create()) 期间指定的应用程序请求通知回调调用此服务。

此服务取代了 nx_http_server_content_length_get()。

输入参数

- packet_ptr：指向 HTTP 客户端请求数据包的指针。请注意，请求通知回调不得释放此数据包。
- content_length：指向从内容长度字段检索到的值的指针

返回值

- NX_SUCCESS：(0x00) 已成功获取 HTTP 服务器内容
- NX_HTTP_INCOMPLETE_PUT_ERROR：(0xEF) HTTP 头格式不正确
- NX_PTR_ERROR：(0x07) 指针输入无效

允许来自

线程数

示例

```
/* 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

创建 HTTP 服务器实例

原型

```
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));
```

说明

此服务创建 HTTP 服务器实例，该实例在其自己的 ThreadX 线程上下文中运行。可选的 authentication_check 和 request_notify 应用程序回调例程可让应用程序软件控制 HTTP 服务器的基本操作 。

输入参数

- http_server_ptr：指向 HTTP 服务器控制块的指针。
- http_server_name：指向 HTTP 服务器名称的指针。
- ip_ptr：指向以前创建的 IP 实例的指针。
- media_ptr：指向以前创建的 FileX 媒体实例的指针。
- stack_ptr：指向 HTTP 服务器线程堆栈区域的指针。
- stack_size：指向 HTTP 服务器线程堆栈大小的指针。
- authentication_check：指向应用程序身份验证检查例程的函数指针。如果已指定，将对每个 HTTP 客户端请求调用此例程。如果此参数为 NULL，则不执行任何身份验证。
- request_notify：指向应用程序请求通知例程的函数指针。如果已指定，则会在 HTTP 服务器处理请求之前调用此例程。这样便可以在完成 HTTP 客户端请求之前，重定向资源名称或更新资源中的字段。

返回值

- NX_SUCCESS：(0x00) 已成功创建 HTTP 服务器。
- NX_PTR_ERROR：(0x07) HTTP 服务器、IP、媒体、堆栈或数据包池指针无效。
- NX_HTTP_POOL_ERROR：(0xE9) 池的数据包有效负载不够大，无法包含整个 HTTP 请求。

允许来自

初始化、线程

示例

```
/* 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

删除 HTTP 服务器实例

原型

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

说明

此服务删除以前创建的 HTTP 服务器实例。

输入参数

- http_server_ptr：指向 HTTP 服务器控制块的指针。

返回值

- NX_SUCCESS：(0x00) 已成功删除 HTTP 服务器
- NX_PTR_ERROR：(0x07) HTTP 服务器指针无效

- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

允许来自

线程数

示例

```
/* 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

检索实体数据的位置和长度

原型

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

说明

此服务确定已收到的客户端消息中当前多部分实体内的数据开始位置，以及不包括边界字符串的数据的长度。HTTP 服务器在内部更新其自己的偏移量，以便可以对包含多个实体的消息的相同客户端数据报再次调用此函数。数据包指针将更新为指向下一个数据包，该数据包中的客户端消息是多数据包数据报。

请注意，必须启用 NX_HTTP_MULTIPART_ENABLE 才能使用此服务。

有关更多详细信息，请参阅“nx_http_server_get_entity_header”。

输入参数

- server_ptr：指向 HTTP 服务器的指针
- packet_pptr：指向数据包指针位置的指针。请注意，应用程序不应释放此数据包。
- available_offset：指向实体数据与数据包预置指针之间的偏移量的指针
- available_length：指向实体数据长度的指针

返回值

- NX_SUCCESS：(0x00) 已成功检索到实体内容的大小和位置
- NX_HTTP_BOUNDARY_ALREADY_FOUND：(0xF4) 已找到 HTTP 服务器内部多部分标记的内容
- NX_HTTP_ERROR：(0xE0) HTTP 服务器内部错误
- NX_PTR_ERROR：(0x07) 指针输入无效

允许来自

线程数

示例

```
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

检索实体头的内容

原型

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

说明

此服务将实体头检索到指定的缓冲区中。HTTP 服务器在内部更新其自己的指针，以在具有多个实体头的客户端数据报中找到下一个多部分实体。数据包指针将更新为指向下一个数据包，该数据包中的客户端消息是多数据包数据报。

请注意，必须启用 NX_HTTP_MULTIPART_ENABLE 才能使用此服务。

输入参数

- server_ptr：指向 HTTP 服务器的指针
- packet_pptr：指向数据包指针位置的指针。请注意，应用程序不应释放此数据包。
- entity_header_buffer：指向实体头存储位置的指针
- buffer_size：输入缓冲区的大小

返回值

- NX_SUCCESS：(0x00) 已成功检索到实体头
- NX_HTTP_NOT_FOUND：(0xE6) 找不到实体头字段
- NX_HTTP_TIMEOUT：(0xE1) 接收多数据包客户端消息的下一个数据包时超过了过期时间
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效
- NX_HTTP_ERROR：(0xE0) 内部 HTTP 错误

允许来自

线程数

示例

```
/* 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: NetX 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

设置回调以获取 GMT 日期和时间

原型

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

说明

此服务设置回调以使用以前创建的 HTTP 服务器获取 GMT 日期和时间。此服务是当 HTTP 服务器正在创建返回给客户端的 HTTP 服务器响应中的头时调用的。

输入参数

- server_ptr：指向 HTTP 服务器的指针
- gmt_get：指向 GMT 回调的指针
- date：指向检索到的日期的指针

返回值

- NX_SUCCESS：(0x00) 已成功设置回调
- NX_PTR_ERROR：(0x07) 数据包或参数指针无效。

允许来自

线程数

示例

```
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

设置回调以处理无效的用户/密码

原型

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

说明

此服务设置当客户端 get、put 或 delete 请求中收到无效的用户名和密码时，要通过摘要身份验证或基本身份验证调用的回调。HTTP 服务器必须事先已创建。

输入参数

- server_ptr：指向 HTTP 服务器的指针

- invalid_username_password_callback：指向无效用户/密码回调的指针
- resource：指向客户端指定的资源的指针
- client_address：客户端地址
- request_type：指示客户端请求类型。可以是：
  - 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

返回值

- NX_SUCCESS：(0x00) 已成功设置回调
- NX_PTR_ERROR：(0x07) 指针输入无效

允许来自

线程数

示例

```
NX_HTTP_SERVER my_server;
VOID invalid_username_password_callback (NX_ CHAR *resource,
                                         ULONG 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

为 **HTML** 设置其他 **MIME** 映射

原型

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

说明

使用此服务，HTTP 应用程序开发人员可以添加除 NetX HTTP 服务器所提供的默认 MIME 类型以外的其他 MIME 类型（有关已定义类型的列表，请参阅“nx_http_server_get_type”）。

收到客户端请求（例如 GET 请求）时，HTTP 服务器将优先使用设置的附加 MIME 映射来分析 HTTP 头中请求的文件类型，如果找不到匹配项，则它会在 HTTP 服务器的默认 MIME 映射中查找匹配项。如果未找到匹配项，MIME 类型将默认为“text/plain”。

如果已将请求通知函数注册到 HTTP 服务器，则请求通知回调可以调用 nx_http_server_type_get 来分析文件类型。

输入参数

- server_ptr：指向 HTTP 服务器实例的指针
- mime_maps：指向 MIME 映射数组的指针
- mime_map_num：数组中 MIME 映射的数目

返回值

- NX_SUCCESS：(0x00) 已成功设置 HTTP 服务器 MIME 映射
- NX_PTR_ERROR：(0x07) 指针输入无效

允许来自

初始化、线程

示例

```
/* 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

提取内容长度，并设置指向数据开始位置的指针

原型

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

说明

此服务从 HTTP 头中提取内容长度。它还会按如下所述更新提供的数据包：将数据包预置指针（要写入到的数据包缓冲区的开始位置）设置为紧接在 HTTP 头后面的 HTTP 内容（数据）。

如果在当前数据包中找不到内容开头位置，该函数将使用 NX_HTTP_SERVER_TIMEOUT_RECEIVE 等待选项等待接收下一个数据包。

请注意，不应在调用 nx_http_server_get_entity_header() 之前调用此函数，因为它会修改实体头后面的预置指针。

输入参数

- server_ptr：指向 HTTP 服务器实例的指针
- packet_ptr：指向数据包指针的指针，用于通过已更新的预置指针返回数据包
- content_length：指向提取到的 content_length 的指针

返回值

- NX_SUCCESS：(0x00) 已找到 HTTP 内容长度，并且已成功更新数据包
- NX_HTTP_TIMEOUT：(0xE1) 等待下一个数据包时超过了过期时间
- NX_PTR_ERROR：(0x07) 指针输入无效

允许来自

线程数

示例

```
/* 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
registere 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

接收下一个 HTTP 数据包

原型

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

说明

此服务返回 HTTP 服务器套接字上收到的下一个数据包。用于接收数据包的等待选项是 NX_HTTP_SERVER_TIMEOUT_RECEIVE。

输入参数

- server_ptr：指向 HTTP 服务器实例的指针
- packet_ptr：指向已接收的数据包的指针

返回值

- NX_SUCCESS：(0x00) 已成功接收下一个 HTTP 数据包
- NX_HTTP_TIMEOUT：(0xE1) 等待下一个数据包时超过了过期时间
- NX_PTR_ERROR：(0x07) 指针输入无效

允许来自

线程数

示例

```
/* 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

获取请求中的参数

原型

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

说明

此服务尝试在提供的请求数据包中检索指定的 HTTP URL 参数。如果请求的 HTTP 参数不存在，此例程将返回 NX_HTTP_NOT_FOUND 状态。应从创建 HTTP 服务器 (nx_http_server_create()) 期间指定的应用程序请求通知回调调用此例程。

输入参数

- packet_ptr：指向 HTTP 客户端请求数据包的指针。请注意，应用程序不应释放此数据包。
- param_number：参数按照从左到右的顺序在参数列表中的逻辑编号（从零开始）。
- param_ptr：要将参数复制到的目标区域。
- max_param_size：参数目标区域的最大大小。

返回值

- NX_SUCCESS：(0x00) 已成功获取 HTTP 服务器参数
- NX_HTTP_NOT_FOUND：(0xE6) 找不到指定的参数
- NX_HTTP_IMPROPERLY_TERMINATED_PARAM：(0xF3) 请求参数未正确终止
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

允许来自

线程数

示例

```
/* 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

获取请求中的查询

原型

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

说明

此服务尝试在提供的请求数据包中检索指定的 HTTP URL 查询。如果请求的 HTTP 查询不存在，此例程将返回 NX_HTTP_NOT_FOUND 状态。应从创建 HTTP 服务器 (nx_http_server_create()) 期间指定的应用程序请求通知回调调用此例程。

输入参数

- packet_ptr：指向 HTTP 客户端请求数据包的指针。请注意，应用程序不应释放此数据包。
- query_number：参数按照从左到右的顺序在查询列表中的逻辑编号（从零开始）。
- query_ptr：要将查询复制到的目标区域。
- max_query_size：查询目标区域的最大大小。

返回值

- NX_SUCCESS：(0x00) 已成功获取 HTTP 服务器查询
- NX_HTTP_FAILED：(0xE2) 查询太小。
- NX_HTTP_NOT_FOUND：(0xE6) 找不到指定的查询
- NX_HTTP_NO_QUERY_PARSED：(0xF2) 客户端请求中没有查询
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

允许来自

线程数

示例

```
/* 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

启动 HTTP 服务器

原型

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

说明

此服务启动以前创建的 HTTP 服务器实例。

输入参数

- http_server_ptr：指向 HTTP 服务器实例的指针。

返回值

- NX_SUCCESS：(0x00) 已成功启动 HTTP 服务器
- NX_PTR_ERROR：(0x07) 指针输入无效

允许来自

初始化、线程

示例

```
/* 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

停止 HTTP 服务器

原型

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

说明

此服务停止以前创建的 HTTP 服务器实例。应在删除 HTTP 服务器实例之前调用此例程。

输入参数

- http_server_ptr：指向 HTTP 服务器实例的指针。

返回值

- NX_SUCCESS：(0x00) 已成功停止 HTTP 服务器
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效

允许来自

线程数

示例

```
/* 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

从客户端 HTTP 请求中提取文件类型

原型

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

说明

此服务从输入缓冲区名称（通常是 URL）将 HTTP 请求类型提取到缓冲区 http_type_string 中，并将其长度提取到返回值中 。如果找不到 MIME 映射，则默认为“text/plain”类型。否则，此服务会将提取的类型与 HTTP 服务器默认 MIME 映射进行比较，以查找匹配项。NetX HTTP 服务器中的默认 MIME 映射为：

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

如果已提供，则它还会搜索用户定义的一组附加 MIME 映射。有关用户定义的映射的更多详细信息，请参阅“nx_http_server_mime_maps_addtional_set()”。

此服务已弃用。建议开发人员迁移到 nx_http_server_type_get_extended()。

输入参数

- http_server_ptr：指向 HTTP 服务器实例的指针
- name：指向要搜索的缓冲区的指针
- http_type_string：指向已提取的 HTML 类型的指针

返回值

- 以字节为单位的字符串长度：非零值表示成功
- 零表示出错

允许来自

应用程序

示例

```
/* 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. */
```

有关更详细的示例，请参阅以下说明

nx_http_server_callback_generate_response_header

nx_http_server_type_get_extended

从客户端 HTTP 请求中提取文件类型

原型

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

说明

此服务从输入缓冲区名称（通常是 URL）将 HTTP 请求类型提取到缓冲区 http_type_string 中，并将其长度提取到返回值中 。如果找不到 MIME 映射，则默认为“text/plain”类型。否则，此服务会将提取的类型与 HTTP 服务器默认 MIME 映射进行比较，以查找匹配项。NetX Duo HTTP 服务器中的默认 MIME 映射为：

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

如果已提供，则它还会搜索用户定义的一组附加 MIME 映射。有关用户定义的映射的更多详细信息，请参阅“nx_http_server_mime_maps_addtional_set()”。

此服务取代了 nx_http_server_type_get()。此版本提供附加长度信息。

输入参数

- http_server_ptr：指向 HTTP 服务器实例的指针
- name：指向要搜索的缓冲区的指针
- name_length：要搜索的缓冲区的长度
- http_type_string：指向已提取的 HTML 类型的指针
- http_type_string_max_size

http_type_string 缓冲区的大小

返回值

- 以字节为单位的字符串长度：非零值表示成功
  零表示出错

允许来自

应用程序

示例

```
/* 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;

/* Get the length of request resource. */
    if (_nx_utility_string_length_check(
       my_server.nx_http_server_request_resource, &resource_length,
       sizeof(my_server.nx_http_server_request_resource) - 1))
    {
        return;
    }
/* Extract the HTTP type. */
string_length = nx_http_server_type_get_extended(&my_server,
                my_server.nx_http_server_request_resource, resource_length,
                temp_string, sizeof(temp_string));

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

有关更详细的示例，请参阅以下说明

nx_http_server_callback_generate_response_header

nx_http_server_digest_authenticate_notify_set

设置摘要身份验证回调函数

原型

```
UINT nx_http_server_digest_authenticate_notify_set(
     NX_HTTP_SERVER *http_server_ptr,
     UINT (*digest_authenticate_callback)(NX_HTTP_SERVER *server_ptr,
                                          CHAR *name_ptr,
                                          CHAR *realm_ptr,
                                          CHAR *password_ptr,
                                          CHAR *method,
                                          CHAR *authorization_uri,
                                          CHAR *authorization_nc,
                                          CHAR *authorization_cnonce
                                          ));
```

说明

此服务设置执行摘要身份验证时调用的回调。

输入参数

- http_server_ptr：指向 HTTP 服务器实例的指针
- digest_authenticate_callback：指向摘要身份验证回调的指针

返回值

- NX_SUCCESS：(0x00) 已成功设置回调
- NX_PTR_ERROR：(0x07) 指针输入无效
- NX_NOT_SUPPORTED：(0x4B) 摘要身份验证未启用

允许来自

应用程序

示例

```
UINT digest_authenticate_callback(NX_HTTP_SERVER *server_ptr, CHAR *name_ptr,
                                  CHAR *realm_ptr, CHAR *password_ptr,
                                  CHAR *method,CHAR *authorization_uri,
                                  CHAR *authorization_nc,
                                  CHAR *authorization_cnonce)
{
    return(NX_SUCCESS);
}

NX_HTTP_SERVER my_server;

/* 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
digest authenticate callback: */

status = nx_http_server_digest_authenticate_notify_set (&my_server,
                                                        digest_authenticate_callback);

/* If status equals NX_SUCCESS, the digest_authenticate_callback function
will be called when the HTTP server performs digest authenticate. */
```

nx_http_server_authentication_check_set

设置身份验证检查回调函数

原型

```
UINT nx_http_server_authentication_check_set(
     NX_HTTP_SERVER *http_server_ptr,
     UINT (*authentication_check_extended)(
                                        NX_HTTP_SERVER *server_ptr,
                                        UINT request_type,
                                        CHAR *resource,
                                        CHAR **name,
                                        UINT *name_length,
                                        CHAR **password,
                                        UINT *password_length,
                                        CHAR **realm,
                                        UINT *realm_length
                                        ));
```

说明

此服务设置身份验证检查的回调函数。

输入参数

- http_server_ptr：指向 HTTP 服务器实例的指针
- authentication_check_extended：指向应用程序身份验证检查的指针

返回值

- NX_SUCCESS：(0x00) 已成功设置回调
- NX_PTR_ERROR：(0x07) 指针输入无效

允许来自

应用程序

示例

```
static UINT authentication_check_extended(NX_HTTP_SERVER *server_ptr,
                                          UINT request_type, CHAR *resource,
                                          CHAR **name, UINT *name_length,
                                          CHAR **password,
                                          UINT *password_length,
                                          CHAR **realm, UINT *realm_length)
{

    /* Just use a simple name, password, and realm for all
    requests and resources. */

    *name = "name";
    *password = "password";
    *realm = "NetX Duo HTTP demo";
    *name_length = 4;
    *password_length = 8;
    *realm_length = 18;

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

NX_HTTP_SERVER my_server;

/* 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 authentication checking callback: */

nx_http_server_authentication_check_set (&my_server,
                                         authentication_check_extended);
```

第 1 章 - Azure RTOS NetX LWM2M 简介

2021/4/30 •

Azure RTOS NetX LWM2M 协议实现轻型计算机到计算机协议标准的客户端部分。

NetX LWM2M 要求

若要正常运行，必须已为 NetX LWM2M 运行时库创建 NetX IP 实例。NetX LWM2M 软件包没有其他要求。

NetX LWM2M RFC

NetX LWM2M 符合 OMA-TS-LightweightM2M-V1_0-20170208-A 和以下与受限制应用程序协议 (Constrained Application Protocol, CoAP) 相关的 RFC：

- RFC 7252：受限制应用程序协议 (CoAP)
- RFC 7641：使用受限制应用程序协议 (CoAP) 观察资源
- RFC 6690：受限制 RESTful 环境 (Constrained RESTful Environments, CoRE) 链路格式

第 2 章 - 安装和使用 Azure RTOS NetX LWM2M

2021/4/30 •

本章介绍与安装、设置和使用 Azure RTOS NetX LWM2M 组件相关的各种问题。

产品分发

NetX LWM2M 在以下网址提供 https://github.com/azure-rtos/netx。此包包含三个源文件、一个包含文件，以及一个包含此文档的 PDF 文件，如下所示：

- nx_lwm2m_client.h：NetX LWM2M 客户端的头文件
- nx_lwm2m_*.c/h：NetX LWM2M 的 C/H 源文件
- demo_netx_lwm2m_client.c：NetX LWM2M 客户端演示的 C 源文件
- NetX_LWM2M_User_Guide.pdf：NetX LWM2M 产品的 PDF 说明

NetX LWM2M 安装

若要使用 NetX LWM2M，应将之前提到的全部分发文件复制到安装 NetX 的相同目录。例如，如果 NetX 安装在“\threadx\arm7\green”目录中，则应将 nx_lwm2m*.* 文件复制到该目录中。

私有 NetX LWM2M

使用 NetX LWM2M 很简单。大致说来，必须在应用程序代码中加入 nx_lwm2m_client.h，然后加入 tx_api.h 和 nx_api.h，才能使用 ThreadX 和 NetX。包含 nx_lwm2m_client.h 之后，应用程序代码即可进行本指南后续部分指定的 NetX LWM2M 函数调用。应用程序还必须将 nx_lwm2m*.* 文件导入 NetX 库。

配置选项

使用 LWM2M 客户端生成 LWM2M 客户端库和应用程序时，有几个配置选项。除非另有说明，否则可通过命令行在应用程序源中定义配置选项。

NX_LWM2M_CLIENT_DISABLE_ERROR_CHECKING

定义后，将删除基本 LWM2M 客户端错误检查 API 并提高性能。不受禁用错误检查影响的 API 返回代码将在 API 定义中以粗体字样列出。

NX_LWM2M_CLIENT_DISABLE_FLOAT

定义后，将删除对浮点数值的支持。禁用后，LWM2M 客户端无法支持浮点数据类型的资源。

NX_LWM2M_CLIENT_DISABLE_FLOAT64

定义后，将删除对 64 位浮点数值的支持。LWM2M 客户端仍可接收包含 64 位浮点数的 TLV 消息，但这些值将转换为 32 位浮点进行处理。

NX_LWM2M_CLIENT_PRIORITY

指定 LWM2M 客户端线程的优先级。默认情况下，此值定义为 16，指定优先级为 16。

NX_LWM2M_CLIENT_MAX_DEVICE_ERRORS

指定设备对象存储的最大错误代码数。默认值为 8。

NX_LWM2M_CLIENT_MAX_SECURITY_INSTANCES

指定安全对象实例的最大数量。默认值为 2，表示支持启动服务器和标准服务器。

NX_LWM2M_CLIENT_MAX_SERVER_INSTANCES

指定服务器对象实例的最大数量。默认值为 1，表示支持单个标准服务器。

NX_LWM2M_CLIENT_MAX_SERVER_URI

指定服务器 URI 的最大长度，包括终止零字符。默认值为 128。

NX_LWM2M_CLIENT_MAX_ACCESS_CONTROL_INSTANCES

指定访问控制实例的最大数量。默认值为 0，这会禁用访问控制。如果应用程序支持多个 LWM2M 服务器，则必须将访问控制实例的最大数量设置为 LWM2M 客户端支持的对象实例的最大数量，因为必须为每个对象实例（安全对象实例除外）创建一个访问控制实例。

NX_LWM2M_CLIENT_MAX_ACCESS_CONTROL_ACLS

指定每个访问控制实例的 ACL 资源的最大数量。默认值为 4。

NX_LWM2M_CLIENT_MAX_NOTIFICATIONS

指定 LWM2M 客户端支持的最大通知数。LWM2M 服务器可以设置有关对象、对象实例和资源的通知。默认值为 8。

NX_LWM2M_CLIENT_MAX_RESOURCES

指定每个对象的最大资源数。默认值为 32。

NX_LWM2M_CLIENT_BOOTSTRAP_IDLE_TIMER

指定启动启动会话时在中止会话之前等待启动服务器请求的最长时间。默认值为 60 秒。

NX_LWM2M_CLIENT_DTLS_START_TIMEOUT

指定等待 DTLS 握手完成的最长时间。默认值为 30 秒。

NX_LWM2M_CLIENT_DTLS_END_TIMEOUT

指定等待 DTLS 关闭完成的最长时间。默认值为 5 秒。

第 3 章 - Azure RTOS NetX LWM2M 功能说明

2021/4/29 •

本章包含 Azure RTOS NetX LWM2M 的功能说明。

LWM2M 客户端初始化

通过调用 nx_lwm2m_client_create 服务来初始化 LWM2M 客户端。LWM2M 客户端在其自己的线程中运行，可以通过使用回调或通过调用应用程序实现的自定义对象的方法，将某些事件报告给应用程序。

此外，必须通过调用用于实现与一个或多个服务器通信的 nx_lwm2m_client_session_create 来创建 LWM2M 客户端会话。一个会话可与两种不同类型的服务器通信：启动服务器或 LWM2M 服务器（设备管理）。

启动服务器会话

与启动服务器建立的通信会话用于在 LWM2M 客户端中预配基本信息，使 LWM2M 客户端能够对一个或多个 LWM2M 服务器执行“Register”操作。在客户端发起的启动和服务器发起的启动模式下，将使用这种类型的服务器。

应用程序可以通过调用 nx_lwm2m_client_session_bootstrap 或 nx_lwm2m_client_session_bootstrap_dtls 来启动启动会话，必须提供服务器的 IP 地址和端口号，以及可选的安全对象实例 ID。nx_lwm2m_client_session_bootstrap 函数使用非安全通信，而 nx_lwm2m_client_session_bootstrap_dtls 则与服务器建立安全的 DTLS 连接。

如果启动操作成功，则启动服务器应已为启动服务器和 LWM2M 服务器创建安全对象实例，并已为 LWM2M 服务器创建服务器对象实例。应用程序必须使用此信息来与 LWM2M 服务器建立会话。

应用程序应将启动数据保存到非易失存储器，以便在下次重新启动设备时配置 LWM2M 客户端。

LWM2M 服务器会话

与 LWM2M 服务器建立的通信会话用于注册、设备管理和服务启用。

应用程序可以通过调用 nx_lwm2m_client_session_register 或 nx_lwm2m_client_session_register_dtls 将 LWM2M 客户端注册到服务器，必须提供服务器的 IP 地址和端口号，以及对应于现有服务器对象实例的短服务器 ID。nx_lwm2m_client_session_register 函数使用非安全通信，而 nx_lwm2m_client_session_register_dtls 则与服务器建立安全的 DTLS 连接。

应用程序可以通过调用 nx_lwm2m_client_session_deregister 来注销 LWM2M 客户端，并要求客户端通过调用 nx_lwm2m_client_session_update 来发送“Update”消息。

会话状态回调

创建会话时，应用程序将注册一个在会话状态更新时调用的回调。回调函数 NX_LWM2M_CLIENT_SESSION_STATE_CALLBACK 具有以下原型：

```
typedef VOID (*NX_LWM2M_CLIENT_SESSION_STATE_CALLBACK)
        (NX_LWM2M_CLIENT_SESSION *session_ptr, UINT state);
```

定义了以下状态：

- NX_LWM2M_CLIENT_SESSION_INIT：会话在创建后的初始状态。
- NX_LWM2M_CLIENT_SESSION_BOOTSTRAP_WAITING：客户端正在等待“暂缓”计时器或服务器发起的启动过期。

- NX_LWM2M_CLIENT_SESSION_BOOTSTRAP_REQUESTING：客户端已将“Request”消息发送到启动服务器（客户端发起的启动）。
- NX_LWM2M_CLIENT_SESSION_BOOTSTRAP_INITIATED：客户端正在从启动服务器接收数据。
- NX_LWM2M_CLIENT_SESSION_BOOTSTRAP_FINISHED：启动服务器已发送“Finished”消息。
- NX_LWM2M_CLIENT_SESSION_BOOTSTRAP_ERROR：启动会话失败。
- NX_LWM2M_CLIENT_SESSION_REGISTERING：客户端已向 LWM2M 服务器发送“Register”消息。
- NX_LWM2M_CLIENT_SESSION_REGISTERED：客户端已注册到 LWM2M 服务器。
- NX_LWM2M_CLIENT_SESSION_UPDATING：客户端已向 LWM2M 服务器发送“Update”消息。
- NX_LWM2M_CLIENT_SESSION_DEREGISTERING：客户端已向 LWM2M 服务器发送“De-register”消息。
- NX_LWM2M_CLIENT_SESSION_DEREGISTERED：客户端已从 LWM2M 服务器中注销。
- NX_LWM2M_CLIENT_SESSION_DISABLED：已禁用 LWM2M 服务器。禁用计时器过期后将发送“Register”。
- NX_LWM2M_CLIENT_SESSION_ERROR：对 LWM2M 服务器执行注册或更新操作失败。
- NX_LWM2M_CLIENT_SESSION_DELETED：已删除对应于 LWM2M 服务器的服务器对象实例。如果出错，应用程序可以通过调用 nx_lwm2m_client_session_error_get 来检索出错原因。

本地设备管理

应用程序可以使用本地设备管理功能来访问 LWM2M 客户端的对象。提供了以下函数：

- nx_lwm2m_client_object_read：读取对象实例中的资源。
- nx_lwm2m_client_object_discover：获取对象实例的资源列表。
- nx_lwm2m_client_object_write：向对象实例写入资源
- nx_lwm2m_client_object_execute：对对象实例的资源执行 Execute 操作。
- nx_lwm2m_client_object_create：创建新的对象实例。
- nx_lwm2m_client_object_delete：删除现有的对象实例。
- nx_lwm2m_client_object_get_next：获取 LWM2M 客户端实现的下一个对象 ID。
- nx_lwm2m_client_object_instance_get_next：获取对象的下一个实例。

资源信息

从对象读取和写入到对象时，资源由 NX_LWM2M_RESOURCE 结构表示。此结构包含资源/实例的 ID 及其值。值的编码取决于该资源/实例的类型和源（应用程序或网络）。

NX_LWM2M_RESOURCE 结构具有以下定义：

```
typedef struct NX_LWM2M_RESOURCE_STRUCT
{
    NX_LWM2M_ID     nx_lwm2m_resource_id;
    UCHAR           nx_lwm2m_resource_type;
    union
    {
        struct
        {
            const VOID *     nx_lwm2m_resource_buffer_ptr;
            UINT             nx_lwm2m_resource_buffer_length;
        } nx_lwm2m_resource_bufferdata;
        const CHAR *         nx_lwm2m_resource_stringdata;
        NX_LWM2M_INT32       nx_lwm2m_resource_integer32data;
        NX_LWM2M_INT64       nx_lwm2m_resource_integer64data;
        NX_LWM2M_FLOAT32     nx_lwm2m_resource_float32data;
        NX_LWM2M_FLOAT64     nx_lwm2m_resource_float64data;
        NX_LWM2M_BOOL        nx_lwm2m_resource_booleandata;
        NX_LWM2M_OBJLNK      nx_lwm2m_resource_objlnkdata;

        struct
        {
            const VOID *     nx_lwm2m_resource_multiple_ptr;
            UINT             nx_lwm2m_resource_multiple_dim;
        } nx_lwm2m_resource_multipledata;
    } nx_lwm2m_resource_value;
} NX_LWM2M_RESOURCE;
```

- nx_lwm2m_resource_id：资源或实例的 ID。
- nx_lwm2m_resource_type：值的类型，请参阅下文。
- nx_lwm2m_resource_value：资源的值，取决于值的类型。

定义了以下类型的值：

- NX_LWM2M_RESOURCE_NONE：空资源。
- NX_LWM2M_RESOURCE_STRING：存储在 nx_lwm2m_resource_stringdata 中的以 null 结尾的 UTF-8 字符串值。
- NX_LWM2M_RESOURCE_INTEGER32：存储在 nx_lwm2m_resource_integer32data 中的 32 位整数值。
- NX_LWM2M_RESOURCE_INTEGER64：存储在 nx_lwm2m_resource_integer64data 中的 64 位整数值。
- NX_LWM2M_RESOURCE_FLOAT32：存储在 nx_lwm2m_resource_float32data 中的 32 位浮点值。
- NX_LWM2M_RESOURCE_FLOAT64：存储在 nx_lwm2m_resource_float64data 中的 64 位浮点值。
- NX_LWM2M_RESOURCE_BOOLEAN：存储在 nx_lwm2m_resource_booleandata 中的布尔值。
- NX_LWM2M_RESOURCE_OPAQUE：nx_lwm2m_resource_bufferdata 定义的不透明值。
- NX_LWM2M_RESOURCE_OBJLNK：存储在 nx_lwm2m_resource_objlnkdata 中的对象链接值。
- NX_LWM2M_RESOURCE_TLV：nx_lwm2m_resource_bufferdata 定义的 TLV 编码值。
- NX_LWM2M_RESOURCE_TEXT：nx_lwm2m_resource_bufferdata 定义的纯文本编码值。
- NX_LWM2M_RESOURCE_MULTIPLE：nx_lwm2m_resource_multipledata 定义的多重资源。nx_lwm2m_resource_multiple_ptr 是指向 NX_LWM2M_RESOURCE 结构数组的指针，该数组包含有关每个资源实例的信息。
- NX_LWM2M_RESOURCE_MULTIPLE_TLV：nx_lwm2m_resource_multipledata 定义的多重资源。nx_lwm2m_resource_multiple_ptr 是指向 TLV 编码缓冲区的指针。

提供了便捷函数用于检索值并检查其类型，从 NX_LWM2M_RESOURCE 结构获取值时，应用程序绝对不应直接访问 nx_lwm2m_resource_value 字段。定义了以下函数：

- nx_lwm2m_resource_get_：获取具有给定类型的值。
- nx_lwm2m_resource_multiple_get_：从多重资源获取具有给定类型的值。

宏 NX_LWM2M_RESOURCE_IS_MULTIPLE(type) 可用于检查某个资源类型是否为多重资源。

对象实现

LWM2M 客户端实现必需的 OMA LWM2M 对象：安全性 (0)、服务器 (1)、访问控制 (2) 和设备 (3)。其他特定于设备的对象必须由应用程序实现。

使用两个数据结构来定义对象：NX_LWM2M_OBJECT 结构定义对象实现（包括对象 ID 和对象方法），NX_LWM2M_OBJECT_INSTANCE 结构包含对象实例的数据。

NX_LWM2M_OBJECT 结构具有以下定义：

```
typedef struct NX_LWM2M_OBJECT_STRUCT
{
    NX_LWM2M_OBJECT * nx_lwm2m_object_next;
    NX_LWM2M_ID nx_lwm2m_object_id;
    UINT (*nx_lwm2m_object_read)(NX_LWM2M_OBJECT *object_ptr,
        NX_LWM2M_OBJECT_INSTANCE *instance_ptr, UINT num_values,
        NX_LWM2M_RESOURCE *values);
    UINT (*nx_lwm2m_object_discover)(NX_LWM2M_OBJECT *object_ptr,
        NX_LWM2M_OBJECT_INSTANCE *instance_ptr, UINT *num_resources,
        NX_LWM2M_RESOURCE_INFO *resources);
    UINT (*nx_lwm2m_object_write)(NX_LWM2M_OBJECT *object_ptr,
        NX_LWM2M_OBJECT_INSTANCE *instance_ptr, UINT num_values, const
        NX_LWM2M_RESOURCE *values, UINT write_op);
    UINT (*nx_lwm2m_object_execute)(NX_LWM2M_OBJECT *object_ptr,
        NX_LWM2M_OBJECT_INSTANCE *instance_ptr, NX_LWM2M_ID resource_id,
        const CHAR *args_ptr, UINT args_length);
    UINT (*nx_lwm2m_object_create)(NX_LWM2M_OBJECT * object_ptr,
        NX_LWM2M_ID instance_id, UINT num_values, const NX_LWM2M_RESOURCE
        *values, NX_LWM2M_OBJECT_INSTANCE **instance_ptr);
    UINT (*nx_lwm2m_object_delete)(NX_LWM2M_OBJECT *object_ptr,
        NX_LWM2M_OBJECT_INSTANCE *instance_ptr);
    NX_LWM2M_OBJECT_INSTANCE * nx_lwm2m_object_instances;
} NX_LWM2M_OBJECT;
```

- nx_lwm2m_object_next：列表中的下一个对象。
- nx_lwm2m_object_id：对象 ID。
- nx_lwm2m_object_read："Read"方法，请参阅下文。
- nx_lwm2m_object_discover："Discover"方法，请参阅下文。
- nx_lwm2m_object_write："Write"方法，请参阅下文。
- nx_lwm2m_object_execute："Execute"方法，请参阅下文。
- nx_lwm2m_object_create："Create"方法，请参阅下文。
- nx_lwm2m_object_delete："Delete"方法，请参阅下文。
- nx_lwm2m_object_instances：对象实例的以 NULL 结尾的列表。

NX_LWM2M_OBJECT_INSTANCE 结构具有以下定义：

```
typedef struct NX_LWM2M_OBJECT_INSTANCE_STRUCT
{
    NX_LWM2M_OBJECT_INSTANCE * nx_lwm2m_object_instance_next;
    NX_LWM2M_ID nx_lwm2m_object_instance_id;
} NX_LWM2M_OBJECT_INSTANCE;
```

- nx_lwm2m_object_instance_next: 列表中的下一个实例。
- nx_lwm2m_object_instance_id: 对象实例 ID。

对象必须实现与 LWM2M 设备管理接口定义的操作对应的方法: “Read”、“Discover”、“Write”、“Execute”、“Create”和“Delete”。如果对象不支持动态创建实例，则“Create”和“Delete”方法可设置为 NULL。

“Read”方法

“Read”方法用于从对象实例读取资源值。函数具有以下定义:

```
UINT nx_lwm2m_object_read(NX_LWM2M_OBJECT *object_ptr,
    NX_LWM2M_OBJECT_INSTANCE *instance_ptr, UINT num_values,
    NX_LWM2M_RESOURCE *values_ptr);
```

输入参数定义如下:

- object_ptr: 指向对象实现的指针。
- instance_ptr: 指向对象实例的指针。
- num_values: 要读取的资源数。
- values_ptr: 指向 NX_LWM2M_RESOURCE 数组的指针，该数组包含要读取的资源的 ID。返回时该数组中填充了对应的类型和值。

“Discover”方法

“Discover”方法用于检索对象实现的所有资源的列表。函数具有以下定义:

```
UINT nx_lwm2m_object_discover(NX_LWM2M_OBJECT *object_ptr,
    NX_LWM2M_OBJECT_INSTANCE *instance_ptr, UINT *num_resources_ptr,
    NX_LWM2M_RESOURCE_INFO *resources_ptr);
```

输入参数定义如下:

- object_ptr: 指向对象实现的指针。
- instance_ptr: 指向对象实例的指针。
- num_resources_ptr: 输入时，这是目标缓冲区的大小，输出时，这是写入缓冲区的元素数目。
- resources_ptr: 指向目标缓冲区的指针。

资源信息在如下定义的 NX_LWM2M_RESOURCE_INFO 结构中返回:

```
typedef struct NX_LWM2M_RESOURCE_INFO_STRUCT
{
    NX_LWM2M_ID     nx_lwm2m_resource_info_id;
    USHORT          nx_lwm2m_resource_info_flags;
    UINT            nx_lwm2m_resource_info_dim;
} NX_LWM2M_RESOURCE_INFO;
```

- nx_lwm2m_resource_info_id: 资源的 ID。

- nx_lwm2m_resource_info_flags：请参阅下文。
- nx_lwm2m_resource_info_dim：如果设置了标志 NX_LWM2M_RESOURCE_INFO_MULTIPLE，则这是多重资源的维度。

字段 nx_lwm2m_resource_flags 可包含以下值：

- 0：单个可读资源。
- NX_LWM2M_RESOURCE_INFO_MULTIPLE：多重资源，必须定义 nx_lwm2m_resource_info_dim。
- NX_LWM2M_RESOURCE_INFO_EXECUTABLE：可执行文件或不可读资源。

“Write”方法

“Write”方法用于更新或替换对象实例的资源。函数具有以下定义：

```
UINT nx_lwm2m_object_write(NX_LWM2M_OBJECT *object_ptr,
    NX_LWM2M_OBJECT_INSTANCE *instance_ptr, UINT num_values,
    const NX_LWM2M_RESOURCE *values_ptr, UINT write_op);
```

输入参数定义如下：

- object_ptr：指向对象实现的指针。
- instance_ptr：指向对象实例的指针。
- num_values：要写入的资源数。
- values_ptr：指向资源值的指针。
- write_op：写入操作的类型。

可对 write_op 参数指定以下写入操作：

- 0 — 部分更新：添加或更新在新值中提供的资源，其他现有资源保持不变。
- NX_LWM2M_OBJECT_WRITE_REPLACE_INSTANCE — 替换实例：将对象实例替换为新提供的资源值。
- NX_LWM2M_OBJECT_WRITE_REPLACE_RESOURCE — 替换资源：将资源替换为新提供的资源值（用于替换多重资源）。
- NX_LWM2M_OBJECT_WRITE_CREATE — 创建实例：使用提供的资源值（从 nx_lwm2m_object_create 方法调用）初始化新建的对象实例。
- NX_LWM2M_OBJECT_WRITE_BOOTSTRAP — 启动写入：在启动序列期间调用。

“Execute”方法

“Execute”方法实现对象资源的执行。函数具有以下定义：

```
UINT nx_lwm2m_object_execute(NX_LWM2M_OBJECT *object_ptr,
    NX_LWM2M_OBJECT_INSTANCE *instance_ptr, NX_LWM2M_ID resource_id,
    const CHAR *arguments_ptr);
```

输入参数定义如下：

- object_ptr：指向对象实现的指针
- instance_ptr：指向对象实例的指针。
- resource_id：资源 ID。
- arguments_ptr：指向执行操作的自变量的指针。如果 arguments_length 为零，则此参数可为 NULL。
- arguments_length：自变量的长度。

如果资源 ID 不存在，则该函数必须返回 NX_LWM2M_NOT_FOUND；如果它不支持执行，则必须返回

NX_LWM2M_METHOD_NOT_ALLOWED。

“Create”方法

“Create”方法实现新对象实例的创建。函数具有以下定义：

```
UINT nx_lwm2m_object_create(NX_LWM2M_OBJECT * object_ptr,
    NX_LWM2M_ID instance_id, UINT num_values, const NX_LWM2M_RESOURCE *values_ptr,
    NX_LWM2M_OBJECT_INSTANCE **instance_ptr, NX_LWM2M_BOOL bootstrap);
```

输入参数定义如下：

- object_ptr：指向对象实现的指针。
- instance_id：新实例的 ID。
- num_values：要初始化的资源数。
- values_ptr：指向资源值的指针。
- instance_ptr：指向所创建实例的目标指针的指针。
- bootstrap：如果从启动序列调用，则为 True。

对象必须使用提供的资源值列表分配并初始化新对象实例。

“Delete”方法

“Delete”方法实现对象实例的删除。函数具有以下定义：

```
UINT nx_lwm2m_object_delete(NX_LWM2M_OBJECT *object_ptr,
                NX_LWM2M_OBJECT_INSTANCE *instance_ptr);
```

输入参数定义如下：

- object_ptr：指向对象实现的指针。
- instance_ptr：指向对象实例的指针。

成功时，对象必须释放实例分配的对象实例数据和任何其他资源。

将对象实现和实例添加到 LWM2M 客户端

应用程序可以通过调用 nx_lwm2m_client_object_add 服务，将新的对象实现添加到 LWM2M 客户端。

如果对象不支持动态创建实例（例如，如果它是仅限单个实例的对象），则对象结构的 nx_lwm2m_object_instances 字段应指向静态实例结构的列表。

如果对象支持动态创建实例，应将 nx_lwm2m_object_instances 设置为 NULL，并且应改用 nx_lwm2m_client_object_create 服务，以调用对象的“Create”方法。

LWM2M 客户端应用程序的示例

以下代码是一个简单的 LWM2M 客户端应用程序示例，该示例实现由温度传感器和灯光开关组成的自定义设备。

该设备允许服务器读取温度传感器的值以及灯光开关的布尔状态，并将灯光开关设置为开/关。

```
#include "nx_lwm2m_client.h"

/* Custom Object implementation */
/* Define the Custom Object Instance structure */
typedef struct
{
    /* The LWM2M Object Instance */
    NX_LWM2M_OBJECT_INSTANCE     lwm2m;
```

```
    /* Resources Data */
    NX_LWM2M_FLOAT32            temperature;
    NX_LWM2M_BOOL               light;
} MYOBJECT_INSTANCE;

/* Define the Resources IDs */
#define MYOBJECT_RES_TEMPERATURE     0 /* temperature sensor */
#define MYOBJECT_RES_LIGHT           1 /* light switch */

/* Define the 'Read' Method */
UINT myobject_read(NX_LWM2M_OBJECT *object_ptr,
    NX_LWM2M_OBJECT_INSTANCE *instance_ptr,
    UINT num_values, NX_LWM2M_RESOURCE *values)
{
    UINT i;
    for (i=0; i<num_values; i++)
    {
        switch (values[i].nx_lwm2m_resource_id)
        {
        case MYOBJECT_RES_TEMPERATURE:

            /* return the temperature value */
            values[i].nx_lwm2m_resource_type = NX_LWM2M_RESOURCE_FLOAT32;
            values[i].nx_lwm2m_resource_value.nx_lwm2m_resource_float32data =
                ((MYOBJECT_INSTANCE *) instance_ptr)->temperature;
            break;
        case MYOBJECT_RES_LIGHT:

            /* return the state of the light switch */
            values[i].nx_lwm2m_resource_type = NX_LWM2M_RESOURCE_BOOLEAN;
            values[i].nx_lwm2m_resource_value.nx_lwm2m_resource_booleandata =
                ((MYOBJECT_INSTANCE *) instance_ptr)->light;
            break;

        default:

            /* unknown resource ID */
            return NX_LWM2M_NOT_FOUND;
        }
    }
    return NX_SUCCESS;
}

/* Define the 'Discover' method */
UINT myobject_discover(NX_LWM2M_OBJECT *object_ptr, NX_LWM2M_OBJECT_INSTANCE *instance_ptr,
                                    UINT *num_resources, NX_LWM2M_RESOURCE_INFO *resources)
{
    if (*num_resources < 2)
    {
        return NX_LWM2M_BUFFER_TOO_SMALL;
    }

    /* return the list of supported resources IDs */
    *num_resources = 2;
    resources[0].nx_lwm2m_resource_info_id        = MYOBJECT_RES_TEMPERATURE;
    resources[0].nx_lwm2m_resource_info_flags     = 0;
    resources[1].nx_lwm2m_resource_info_id        = MYOBJECT_RES_LIGHT;
    resources[1].nx_lwm2m_resource_info_flags     = 0;
    return NX_SUCCESS;
}

/* Define the 'Write' method */
UINT myobject_write(NX_LWM2M_OBJECT *object_ptr,
    NX_LWM2M_OBJECT_INSTANCE *instance_ptr, UINT num_values,
    const NX_LWM2M_RESOURCE *values, UINT flags)
{
    UINT i;
    for (i=0; i<num_values; i++)
```

```
    for (i = 0; i<num_values; i++)
    {
        UINT ret;
        switch (values[i].nx_lwm2m_resource_id)
        {

        case MYOBJECT_RES_TEMPERATURE:

            /* read-only resource */
            return NX_LWM2M_METHOD_NOT_ALLOWED;

        case MYOBJECT_RES_LIGHT:

            /* assign boolean value */

            ret = nx_lwm2m_resource_get_boolean(&values[i],
                &((MYOBJECT_INSTANCE *) instance_ptr)->light);

            if (ret != NX_SUCCESS)
            {

                /* invalid value type */
                return ret;
            }
            break;

        default:

            /* unknown resource ID */
            return NX_LWM2M_NOT_FOUND;
        }
    }
    return NX_SUCCESS;
}

/* Define the 'Execute' method */
UINT myobject_execute(NX_LWM2M_OBJECT *object_ptr,
    NX_LWM2M_OBJECT_INSTANCE *instance_ptr, NX_LWM2M_ID resource_id,
    const CHAR *args_ptr, UINT args_length)
{
    switch (resource_id)
    {

    case MYOBJECT_RES_TEMPERATURE:
    case MYOBJECT_RES_LIGHT:

        /* read-only resource */
        return NX_LWM2M_METHOD_NOT_ALLOWED;

    default:

        /* unknown resource ID */
        return NX_LWM2M_NOT_FOUND;

    }
}

/* NetX data */
NX_IP                       ip;
NX_PACKET_POOL              packet_pool;

/* LWM2M Client data */
NX_LWM2M_CLIENT             client;
ULONG                       client_stack[4096 / sizeof(ULONG)];
NX_LWM2M_CLIENT_SESSION     session;

/* Custom Object Data */
NX_LWM2M_OBJECT             myobject;
MYOBJECT_INSTANCE           myinstance;
```

```
/* Define the session state callback */
void session_callback(NX_LWM2M_CLIENT_SESSION *session_ptr, UINT state)
{
    switch (state)
    {

    case NX_LWM2M_CLIENT_SESSION_BOOTSTRAP_FINISHED:

        /* Bootstrap session done, we can register to the LWM2M Server */
        {
            NX_LWM2M_ID security_id;

            /* find the Security Object Instance of the LWM2M Server */
            security_id = NX_LWM2M_RESERVED_ID;
            while (nx_lwm2m_client_object_instance_get_next(&client,
                NX_LWM2M_SECURITY_OBJECT_ID, &security_id) == NX_SUCCESS)
            {
                NX_LWM2M_RESOURCE res[3];

                /* retrieve instance data: */
                /* Bootstrap server flag */
                res[0].nx_lwm2m_resource_id = NX_LWM2M_SECURITY_BOOTSTRAP_ID;

                /* URI of server */
                res[1].nx_lwm2m_resource_id = NX_LWM2M_SECURITY_URI_ID;

                /* Short Server ID */
                res[2].nx_lwm2m_resource_id = NX_LWM2M_SECURITY_SHORT_SERVER_ID;

                /* Read Object Instance: */
                nx_lwm2m_client_object_read(&client,
                    NX_LWM2M_SECURITY_OBJECT_ID, security_id, 3, res);

                /* Not a bootstrap server? */
                if (!res[0].nx_lwm2m_resource_value.nx_lwm2m_resource_booleandata)
                {
                    ULONG ip_addr;
                    UINT udp_port;

                    /* get IP address and UDP port from server URI */
                    parse_uri(res[1].nx_lwm2m_resource_value.nx_lwm2m_resource_stringdata, &ip_addr,
&udp_port);

                    /* Start registration to the LWM2M server */
                    nx_lwm2m_client_session_register(&session,
                        (NX_LWM2M_ID) res[2].nx_lwm2m_resource_value.
                        nx_lwm2m_resource_integer32data,
                        ip_addr, udp_port);
                    break;
                }
            }
        }
        break;

    case NX_LWM2M_CLIENT_SESSION_BOOTSTRAP_ERROR:

        /* Failed to Bootstrap the LWM2M Client. */
        break;

    case NX_LWM2M_CLIENT_SESSION_REGISTERED:

        /* Registration to the LWM2M Client done. */
        break;

    case NX_LWM2M_CLIENT_SESSION_ERROR:

        /* Failed to register to the LWM2M Client. */
        break;
    }
```

```
    }
}

/* Application main thread */
void application_thread(ULONG info)
{
    NX_LWM2M_RESOURCE res[4];
    NX_LWM2M_ID security_id;

    /* Create the LWM2M client */
    nx_lwm2m_client_create(&client, &ip, &packet_pool, NX_LWM2M_COAP_PORT,
        "mylwm2mclient", NULL, NX_LWM2M_BINDING_U,
        client_stack, sizeof(client_stack));

    /* Define our custom object */
    myobject.nx_lwm2m_object_id           = 1024;
    myobject.nx_lwm2m_object_read         = myobject_read;
    myobject.nx_lwm2m_object_discover     = myobject_discover;
    myobject.nx_lwm2m_object_write        = myobject_write;
    myobject.nx_lwm2m_object_execute      = myobject_execute;
    myobject.nx_lwm2m_object_create       = NULL;
    myobject.nx_lwm2m_object_delete       = NULL;

    /* Define a single instance */
    myobject.nx_lwm2m_object_instances             = (NX_LWM2M_OBJECT_INSTANCE *) &myinstance;
    myinstance.lwm2m.nx_lwm2m_object_instance_id   = 0;
    myinstance.lwm2m.nx_lwm2m_object_instance_next = NULL;
    myinstance.temperature                         = 22.5f;
    myinstance.light                               = NX_FALSE;

    /* Add the object to the LWM2M Client */
    nx_lwm2m_client_object_add(&client, &myobject);

    /* Create a security entry for the bootstrap server */
    security_id = 0;

    /* set the URI of the server */
    res[0].nx_lwm2m_resource_id                                 = NX_LWM2M_SECURITY_URI_ID;
    res[0].nx_lwm2m_resource_type                               = NX_LWM2M_RESOURCE_STRING;
    res[0].nx_lwm2m_resource_value.nx_lwm2m_resource_stringdata = "coap://1.2.3.4";

    /* set the Bootstrap flag */
    res[1].nx_lwm2m_resource_id                                  = NX_LWM2M_SECURITY_BOOTSTRAP_ID;
    res[1].nx_lwm2m_resource_type                                = NX_LWM2M_RESOURCE_BOOLEAN;
    res[1].nx_lwm2m_resource_value.nx_lwm2m_resource_booleandata = NX_TRUE;

    /* set the security mode */
    res[2].nx_lwm2m_resource_id                                   = NX_LWM2M_SECURITY_MODE_ID;
    res[2].nx_lwm2m_resource_type                                 = NX_LWM2M_RESOURCE_INTEGER32;
    res[2].nx_lwm2m_resource_value.nx_lwm2m_resource_integer32data =
    NX_LWM2M_SECURITY_MODE_NOSEC;

    /* set the Hold Off timer */
    res[3].nx_lwm2m_resource_id                                   = NX_LWM2M_SECURITY_HOLD_OFF_ID;
    res[3].nx_lwm2m_resource_type                                 = NX_LWM2M_RESOURCE_INTEGER32;
    res[3].nx_lwm2m_resource_value.nx_lwm2m_resource_integer32data = 10;
    nx_lwm2m_client_object_create(&client, NX_LWM2M_SECURITY_OBJECT_ID,
        &security_id, 4, res);

    /* Create a session */
    nx_lwm2m_client_session_create(&session, &client, session_callback);

    /* start bootstrap session */
    nx_lwm2m_client_session_bootstrap(&session, security_id,
                            IP_ADDRESS(1, 2, 3, 4), 5683);

    /* Application main loop */
    while (1)
    {
```

```
        /* ...application code... */
    }

    /* Terminate the LWM2M Client */
    nx_lwm2m_client_delete(&client);
}
```

第 4 章 - Azure RTOS NetX LWM2M 服务说明

2021/4/29 •

本章提供下面按字母顺序列出的所有 Azure RTOS NetX LWM2M 服务的说明。

在以下 API 说明的"返回值"部分，以 **粗体** 显示的值不受 NX_DISABLE_ERROR_CHECKING 定义（用于禁用 API 错误检查）影响，而对于非粗体值，则会完全禁用该检查。

LWM2M 客户端管理

- nx_lwm2m_client_create：创建 LWM2M 客户端
- nx_lwm2m_client_delete：删除 LWM2M 客户端
- nx_lwm2m_client_lock：锁定 LWM2M 客户端
- nx_lwm2m_client_object_add：向 LWM2M 客户端添加对象实现
- nx_lwm2m_client_unlock：解锁 LWM2M 客户端

LWM2M 客户端会话管理

- nx_lwm2m_client_session_bootstrap：启动与启动服务器的会话
- nx_lwm2m_client_session_bootstrap_dtls：启动与启动服务器的安全会话
- nx_lwm2m_client_session_create：创建 LWM2M 客户端会话
- nx_lwm2m_client_session_delete：删除 LWM2M 客户端会话
- nx_lwm2m_client_session_deregister：终止与 LWM2M 服务器的会话
- nx_lwm2m_client_session_error_get：获取会话的最后一个错误
- nx_lwm2m_client_session_register：启动与 LWM2M 服务器的会话
- nx_lwm2m_client_session_register_dtls：启动与 LWM2M 服务器的安全会话
- nx_lwm2m_client_session_update：更新 LWM2M 服务器的会话

安全对象实现

- nx_lwm2m_client_security_key_callbacks_set：设置安全密钥管理回调

设备对象实现

- nx_lwm2m_client_device_callbacks_set：设置设备对象应用程序回调
- nx_lwm2m_client_device_error_push：向设备对象添加错误代码
- nx_lwm2m_client_device_error_reset：重置设备对象的错误代码
- nx_lwm2m_client_device_resource_changed：指出设备对象资源已更改

自定义对象实现

- nx_lwm2m_object_resource_changed：指出对象实例的资源值已更改

本地设备管理

- nx_lwm2m_client_object_create：创建新的对象实例
- nx_lwm2m_client_object_delete：删除对象实例
- nx_lwm2m_client_object_discover：发现对象实例的资源
- nx_lwm2m_client_object_execute：执行对象实例的资源
- nx_lwm2m_client_object_get_next：获取 LWM2M 客户端所实现的对象列表
- nx_lwm2m_client_object_instance_get_next：获取对象的实例列表
- nx_lwm2m_client_object_read：读取对象实例的资源值
- nx_lwm2m_client_object_write：更改对象实例的资源值

资源值解码

- nx_lwm2m_resource_get_boolean：获取布尔资源的值
- nx_lwm2m_resource_get_float32：获取 32 位浮点资源的值
- nx_lwm2m_resource_get_float64：获取 64 位浮点资源的值
- nx_lwm2m_resource_get_integer32：获取 32 位整数资源的值
- nx_lwm2m_resource_get_integer64：获取 64 位整数资源的值
- nx_lwm2m_resource_get_objlnk：获取对象链接资源的值
- nx_lwm2m_resource_get_opaque：获取不透明资源的值
- nx_lwm2m_resource_get_string：获取字符串资源的值
- nx_lwm2m_resource_multiple_get_boolean：获取布尔资源多重资源实例的值
- nx_lwm2m_resource_multiple_get_float32：获取 32 位浮点多重资源实例的值
- nx_lwm2m_resource_multiple_get_float64：获取 64 位浮点多重资源实例的值
- nx_lwm2m_resource_multiple_get_integer32：获取 32 位整数多重资源实例的值
- nx_lwm2m_resource_multiple_get_integer64：获取 64 位整数多重资源实例的值
- nx_lwm2m_resource_multiple_get_objlnk：获取对象链接多重资源实例的值
- nx_lwm2m_resource_multiple_get_opaque：获取不透明多重资源实例的值
- nx_lwm2m_resource_multiple_get_string：获取字符串多重资源实例的值

固件更新对象

- nx_lwm2m_firmware_create：创建固件更新对象
- nx_lwm2m_firmware_package_info_set：设置固件更新包信息
- nx_lwm2m_firmware_result_set：设置固件更新结果
- nx_lwm2m_firmware_state_set：设置固件更新状态

nx_lwm2m_client_create

创建 LWM2M 客户端

原型

```
UINT nx_lwm2m_client_create(NX_LWM2M_CLIENT *client_ptr, NX_IP *ip_ptr,
    NX_PACKET_POOL *packet_pool_ptr, UINT local_port, const CHAR *name_ptr,
    const CHAR *msisdn_ptr, UCHAR binding_modes, VOID *stack_ptr, ULONG stack_size);
```

说明

此服务创建 LWM2M 客户端实例，该实例在其自己的 ThreadX 线程的上下文中运行。

LWM2M 客户端实现了以下 OMA LWM2M 对象：安全性 (0)、服务器 (1)、访问控制 (2) 和设备 (3)。其他对象实现必须由应用程序添加。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- ip_ptr：指向以前所创建 IP 实例的指针。
- packet_pool_ptr：指向此 LWM2M 客户端的默认数据包池的指针。
- local_port：用于非安全通信的本地 UDP 端口。
- name_ptr：指向 LWM2M 客户端终结点名称的指针。
- msisdn_ptr：指向 MSISDN 的指针，从中可以访问 LWM2M 客户端以便与 SMS 绑定配合使用，如果不支持 SMS 绑定，则可为 NULL。
- binding_modes：LWM2M 客户端所支持的绑定和模式，必须是 NX_LWM2M_BINDING_U、NX_LWM2M_BINDING_UQ、NX_LWM2M_BINDING_S 和 NX_LWM2M_BINDING_SQ 标志的组合。

- stack_ptr：指向 LWM2M 客户端线程堆栈区域的指针。
- stack_size：LWM2M 客户端线程堆栈大小。

返回值

- NX_SUCCESS：已成功创建 LWM2M 客户端。
- NX_LWM2M_ERROR：创建 LWM2M 客户端出错。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_delete

删除 LWM2M 客户端

原型

```
UINT nx_lwm2m_client_delete(NX_LWM2M_CLIENT *client_ptr);
```

说明

此服务删除以前创建的 LWM2M 客户端实例。

此调用还会删除所有当前已附加该客户端的会话。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。

返回值

- NX_SUCCESS：已成功删除 LWM2M 客户端。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_device_callbacks_set

设置设备对象应用程序回调

原型

```
UINT nx_lwm2m_client_device_callbacks_set(NX_LWM2M_CLIENT *client_ptr,
    NX_LWM2M_CLIENT_DEVICE_READ_CALLBACK read_callback,
    NX_LWM2M_CLIENT_DEVICE_DISCOVER_CALLBACK discover_callback,
    NX_LWM2M_CLIENT_DEVICE_WRITE_CALLBACK write_callback,
    NX_LWM2M_CLIENT_DEVICE_EXECUTE_CALLBACK execute_callback);
```

说明

此服务安装应用程序回调，用于在并非由 LWM2M 客户端处理的 LWM2M 设备对象资源上实现操作。

LWM2M 客户端实现了以下资源：错误代码 (11)、重置错误代码 (12) 以及支持的绑定和模式 (16)，针对其他资源的操作将重定向到应用程序。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- read_callback：“读取”方法回调。
- discover_callback：“发现”方法回调。
- write_callback：“写入”方法回调。
- execute_callback：“执行”方法回调。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_device_error_push

向设备对象添加错误代码

原型

```
UINT nx_lwm2m_client_device_error_push(NX_LWM2M_CLIENT *client_ptr, UCHAR code);
```

说明

此服务将一个新实例添加到对象设备的错误代码 (11) 资源。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- code：新的错误代码。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BUFFER_TOO_SMALL：已达到存储的错误代码的最大数目。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_device_error_reset

重置设备对象的错误代码

原型

```
UINT nx_lwm2m_client_device_error_reset(NX_LWM2M_CLIENT *client_ptr);
```

说明

此服务从设备对象中删除所有的错误代码资源实例。这等同于执行资源“重置错误代码”(12)。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_device_resource_changed

指出设备对象资源已更改

原型

```
UINT nx_lwm2m_client_device_resource_changed(NX_LWM2M_CLIENT *client_ptr,
                                     const NX_LWM2M_RESOURCE *resource);
```

说明

应用程序使用该服务向 LWM2M 客户端发送信号，指出对象设备的资源已更改。当 LWM2M 服务器观察到资源时，LWM2M 客户端就会发送通知。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- resource：指向一个结构的指针，该结构说明已更改的资源。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_lock

锁定 LWM2M 客户端

原型

```
UINT nx_lwm2m_client_lock(NX_LWM2M_CLIENT *client_ptr);
```

说明

此服务锁定 LWM2M 客户端，以防止从服务器或其他应用程序线程并发访问 LWM2M 对象。

如果 LWM2M 客户端当前由另一线程锁定，则该函数会被阻止，直到 LWM2M 客户端解锁为止。

nx_lwm2m_client_lock()/nx_lwm2m_client_unlock() 对的调用可以嵌套。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_object_add

向 LWM2M 客户端添加对象实现

原型

```
UINT nx_lwm2m_client_object_add(NX_LWM2M_CLIENT *client_ptr,
                                NX_LWM2M_OBJECT *object_ptr);
```

说明

此服务将新的对象实现添加到 LWM2M 客户端。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- object_ptr：指向一个结构的指针，该结构定义对象实现。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_ALREADY_EXIST：该对象 ID 已存在。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_object_create

创建新的对象实例

原型

```
UINT nx_lwm2m_client_object_create(NX_LWM2M_CLIENT *client_ptr,
            NX_LWM2M_ID object_id, NX_LWM2M_ID *instance_id_ptr,
            UINT num_values, const NX_LWM2M_RESOURCE *values_ptr);
```

说明

此服务对 LWM2M 客户端的对象执行"创建"操作，以创建新的对象实例。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- object_id：对象 ID。
- instance_id_ptr：指向新实例的 ID 的指针，如果 LWM2M 客户端必须分配实例 ID，则此参数可以为 NULL。如果该 ID 为保留值 65535，则 LWM2M 客户端会分配实例 ID。如果不为 NULL，则执行该调用后就会返回所分配的 ID。
- num_values：要设置的值的数目。
- values_ptr：指向资源值数组的指针，这些资源值用于初始化新对象实例。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_ALREADY_EXIST：该对象实例 ID 已存在。
- NX_LWM2M_METHOD_NOT_ALLOWED：该对象不支持实例创建。
- NX_LWM2M_NO_MEMORY：无法分配内存来存储新实例。
- NX_LWM2M_NOT_FOUND：该对象 ID 不存在。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_object_delete

删除对象实例

原型

```
UINT nx_lwm2m_client_object_delete(NX_LWM2M_CLIENT *client_ptr,
                NX_LWM2M_ID object_id, NX_LWM2M_ID instance_id);
```

说明

此服务对 LWM2M 客户端的对象实例执行"删除"操作。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- object_id：对象 ID。
- instance_id：对象实例 ID。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_METHOD_NOT_ALLOWED：该对象不支持实例删除。
- NX_LWM2M_NOT_FOUND：该对象 ID 或对象实例 ID 不存在。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_object_discover

发现对象实例的资源

原型

```
UINT nx_lwm2m_client_object_discover(NX_LWM2M_CLIENT *client_ptr,
        NX_LWM2M_ID object_id, NX_LWM2M_ID instance_id,
        UINT *num_resources_ptr, NX_LWM2M_RESOURCE_INFO *resources_ptr);
```

说明

此服务对 LWM2M 客户端的对象实例执行"发现"操作，这将返回该对象所实现的资源列表。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- object_id：对象 ID。
- instance_id：对象实例 ID。
- num_resources_ptr：输入时，这是目标缓冲区的大小，输出时，这是写入缓冲区的元素数目。
- resources_ptr：指向目标缓冲区的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BUFFER_TOO_SMALL：资源缓冲区太小，无法存储资源列表。
- NX_LWM2M_NOT_FOUND：该对象 ID 或对象实例 ID 不存在。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_object_execute

执行对象实例的资源

原型

```
UINT nx_lwm2m_client_object_execute(NX_LWM2M_CLIENT *client_ptr,
    NX_LWM2M_ID object_id, NX_LWM2M_ID instance_id, NX_LWM2M_ID resource_id,
    const CHAR *arguments_ptr, UINT arguments_length);
```

说明

此服务对 LWM2M 客户端的对象实例资源执行"执行"操作。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- object_id：对象 ID。
- instance_id：对象实例 ID。
- resource_id：资源 ID。
- arguments_ptr：指向执行操作参数的指针。如果 arguments_length 为零，则此参数可以为 NULL。
- arguments_length：参数的长度。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_METHOD_NOT_ALLOWED：该资源不支持执行操作。
- NX_LWM2M_NOT_FOUND：该对象 ID、对象实例 ID 或资源 ID 不存在。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_object_get_next

获取 LWM2M 客户端所实现的对象列表

原型

```
UINT nx_lwm2m_client_object_get_next(NX_LWM2M_CLIENT *client_ptr,
                                     NX_LWM2M_ID *object_id_ptr);
```

说明

此服务返回 LWM2M 客户端所实现的下一个对象的 ID。如果当前对象 ID 设置为 NX_LWM2M_RESERVED_ID (65535)，则会返回第一个对象 ID。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- object_id_ptr：指向当前对象 ID 的指针。返回时，包含 LWM2M 客户端所实现的下一个对象 ID。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_NOT_FOUND：给定的对象 ID 是数据库中的最后一个 ID。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_object_instance_get_next

获取对象的实例列表

原型

```
UINT nx_lwm2m_client_object_instance_get_next(NX_LWM2M_CLIENT *client_ptr,
                    NX_LWM2M_ID object_id, NX_LWM2M_ID *instance_id_ptr);
```

说明

此服务返回给定对象的下一个对象实例的 ID。如果当前实例 ID 设置为 NX_LWM2M_RESERVED_ID (65535)，则会返回第一个实例的 ID。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- object_id：对象 ID。
- instance_id_ptr：指向当前对象实例 ID 的指针。返回时，包含该对象的下一个实例 ID。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_NOT_FOUND：给定的实例 ID 是该对象的最后一个实例 ID，或者该对象 ID 尚未实现。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_object_read

读取对象实例的资源值

原型

```
UINT nx_lwm2m_client_object_read(NX_LWM2M_CLIENT *client_ptr,
        NX_LWM2M_ID object_id, NX_LWM2M_ID instance_id,
        UINT num_values, NX_LWM2M_RESOURCE *values);
```

说明

此服务对 LWM2M 客户端的对象实例执行“读取”操作。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- object_id：对象 ID。
- instance_id：对象实例 ID。
- num_values：要读取的资源数目。
- values_ptr：指向 NX_LWM2M_RESOURCE 数组的指针，其中，NX_LWM2M_RESOURCE 包含要读取的资源的 ID。返回时，该数组中填充有相应的类型和值。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_METHOD_NOT_ALLOWED：资源不支持读取操作。
- NX_LWM2M_NOT_FOUND：该对象 ID、对象实例 ID 或资源 ID 不存在。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_object_write

更改对象实例的资源值

原型

```
UINT nx_lwm2m_client_object_write(NX_LWM2M_CLIENT *client_ptr,
        NX_LWM2M_ID object_id, NX_LWM2M_ID instance_id, UINT num_values,
        const NX_LWM2M_RESOURCE *values_ptr, UINT write_op);
```

说明

此服务对 LWM2M 客户端的对象实例执行"写入"操作。

可以对 write_op 参数指定以下写入操作：

- 0 — 部分更新：添加或更新在新值中提供的资源，其他现有资源保持不变。
- NX_LWM2M_OBJECT_WRITE_REPLACE_INSTANCE — 替换实例：将对象实例替换为新提供的资源值。
- NX_LWM2M_OBJECT_WRITE_REPLACE_RESOURCE — 替换资源：将资源替换为新提供的资源值（用于替换多个资源）。
- NX_LWM2M_OBJECT_WRITE_BOOTSTRAP — 启动写入：指示从启动序列中执行的调用。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- object_id：对象 ID。
- instance_id：对象实例 ID。
- num_values：要写入的资源数目。
- values_ptr：指向 NX_LWM2M_RESOURCE 数组的指针，其中，NX_LWM2M_RESOURCE 包含要写入的资源 ID、类型和值。
- write_op：写入操作的类型。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_ENCODING：资源类型无效。
- NX_LWM2M_BUFFER_TOO_SMALL：值的长度太大，无法存储在该实例中。
- NX_LWM2M_METHOD_NOT_ALLOWED：资源不支持写入操作。
- NX_LWM2M_NO_MEMORY：无法分配内存来存储资源值。

- NX_LWM2M_NOT_ACCEPTABLE：某个资源的值无效。
- NX_LWM2M_NOT_FOUND：该对象 ID、对象实例 ID 或资源 ID 不存在。
- NX_OPTION_ERROR：写入操作类型无效。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_security_key_callbacks_set

设置安全对象密钥管理回调

原型

```
UINT nx_lwm2m_client_security_key_callbacks_set(NX_LWM2M_CLIENT *client_ptr,
                NX_LWM2M_CLIENT_SECURITY_KEY_WRITE_CALLBACK write_callback,
                NX_LWM2M_CLIENT_SECURITY_KEY_DELETE_CALLBACK delete_callback);
```

说明

此服务安装应用程序回调，用于在 LWM2M 安全对象资源上实现与安全密钥相关的操作。

在启动会话过程中，LWM2M 客户端会将安全密钥管理委托给应用程序，当启动服务器在安全对象实例上写入或删除以下资源时，就会调用这些回调：公钥或身份 (3)、服务器公钥 (4) 和密钥 (5)。

应用程序负责存储密钥数据，以及配置 LWM2M 客户端所使用的 DTLS 会话。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。
- write_callback："写入"密钥方法回调。
- delete_callback："删除"密钥方法回调。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_session_bootstrap

启动与启动服务器的会话

原型

```
UINT nx_lwm2m_client_session_bootstrap(NX_LWM2M_CLIENT_SESSION
    *session_ptr, NX_LWM2M_ID security_id, ULONG ip_address, UINT port);
```

说明

此服务启动与启动服务器的会话。该服务器在安全对象中应该有相应的安全实例。

在与此服务器相关联的安全实例中，如果"延迟"资源不为零，则该会话会等待服务器所启动的启动，如果服务器在这段时间过后仍未启动任何启动，则会执行客户端所启动的启动。

此调用会中止所有的当前活动会话，并将其替换为新的启动服务器会话。

参数

- session_ptr：指向 LWM2M 客户端会话控制块的指针。
- security_id：如果启动服务器未定义有任何安全实例，则启动服务器的安全实例 ID 必须设置为 NX_LWM2M_RESERVED_ID (65535)。
- ip_address：服务器的 IP 地址。

- port：服务器的 UDP 端口。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_NOT_FOUND：没有与安全实例 ID 相对应的安全对象实例。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_session_bootstrap_dtls

启动与启动服务器的安全会话

原型

```
UINT nx_lwm2m_client_session_bootstrap_dtls(NX_LWM2M_CLIENT_SESSION *session_ptr,
    NX_LWM2M_ID security_id, ULONG ip_address, UINT port, NX_SECURE_DTLS_SESSION
    *dtls_session_ptr, UINT dtls_local_port);
```

说明

此服务使用安全 DTLS 连接来启动与启动服务器的会话。该服务器在安全对象中应该有相应的安全实例。

在调用此函数之前，必须使用正确的密码套件和密钥材料来配置 DTLS 会话。必须定义 NX_SECURE_ENABLE_DTLS。

在与此服务器相关联的安全实例中，如果"延迟"资源不为零，则该会话会等待服务器所启动的启动，如果服务器在这段时间过后仍未启动任何启动，则会执行客户端所启动的启动。

此调用会中止所有的当前活动会话，并将其替换为新的启动服务器会话。

参数

- session_ptr：指向 LWM2M 客户端会话控制块的指针。
- security_id：如果启动服务器未定义有任何安全实例，则启动服务器的安全实例 ID 必须设置为 NX_LWM2M_RESERVED_ID (65535)。
- ip_address：服务器的 IP 地址。
- port：服务器的 UDP 端口。
- dtls_session_ptr：要用于启动会话的 DTLS 会话。
- dtls_local_port：用于 DTLS 会话的本地 UDP 端口。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_NOT_FOUND：没有与安全实例 ID 相对应的安全对象实例。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_session_create

创建 LWM2M 客户端会话

原型

```
UINT nx_lwm2m_client_session_create(NX_LWM2M_CLIENT_SESSION *session_ptr,
         NX_LWM2M_CLIENT *client_ptr,
         NX_LWM2M_CLIENT_SESSION_STATE_CALLBACK state_callback);
```

说明

此服务创建新的 LWM2M 客户端会话，该会话会附加到现有的 LWM2M 客户端。LWM2M 客户端使用该会话与

启动服务器或 LWM2M 服务器进行通信。

应用程序必须提供一个回调函数，以在会话状态更新时调用该函数。

参数

- session_ptr：指向 LWM2M 客户端会话控制块的指针。
- client_ptr：指向以前所创建 LWM2M 客户端的指针。
- state_callback：会话状态更改时调用的应用程序回调。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_session_delete

删除 LWM2M 客户端会话

原型

```
UINT nx_lwm2m_client_session_delete(NX_LWM2M_CLIENT_SESSION *session_ptr);
```

说明

此服务删除 LWM2M 客户端会话。

通过调用 nx_lwm2m_client_delete 删除 LWM2M 客户端时，还会删除所有附加到该客户端的会话。

参数

- session_ptr：指向 LWM2M 客户端会话控制块的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_session_deregister

终止与 LWM2M 服务器的会话

原型

```
UINT nx_lwm2m_client_session_deregister(NX_LWM2M_CLIENT_SESSION *session_ptr);
```

说明

此服务对 LWM2M 服务器执行“注销”操作。

参数

- session_ptr：指向 LWM2M 客户端会话控制块的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_NOT_REGISTERED：该客户端未向服务器注册。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_session_error_get

获取会话的最后一个错误

原型

```
UINT nx_lwm2m_client_session_error_get(NX_LWM2M_CLIENT_SESSION *session_ptr);
```

说明

当会话处于错误状态时，此服务会返回该会话的错误代码。

参数

- session_ptr：指向 LWM2M 客户端会话控制块的指针。

返回值

- NX_SUCCESS：该会话并非处于错误状态。
- NX_LWM2M_ADDRESS_ERROR：服务器地址无效。
- NX_LWM2M_BUFFER_TOO_SMALL：请求有效负载在网络缓冲区中放不下。
- NX_LWM2M_DTLS_ERROR：未能与服务器建立安全连接。
- NX_LWM2M_ERROR：未指定的错误。
- NX_LWM2M_FORBIDDEN：注册被服务器拒绝。
- NX_LWM2M_NOT_FOUND：执行更新/注销时，服务器找不到客户端。
- NX_LWM2M_SERVER_INSTANCE_DELETED：与该会话相对应的服务器对象实例已删除。
- NX_LWM2M_TIMED_OUT：服务器无应答。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_session_register

启动与 LWM2M 服务器的会话

原型

```
UINT nx_lwm2m_client_session_register(NX_LWM2M_CLIENT_SESSION *session_ptr,
                    NX_LWM2M_ID server_id, ULONG ip_address, UINT port);
```

说明

此服务对 LWM2M 服务器执行"注册"操作。该服务器在服务器对象中必须有相应的服务器实例。

如果注册成功，则 LWM2M 客户端会处理来自服务器的进一步操作，并且会定期发送"更新"消息，直到该客户端注销为止。

此调用会中止所有的当前活动会话，并将其替换为新的 LWM2M 服务器会话。

参数

- session_ptr：指向 LWM2M 客户端会话控制块的指针。
- server_id：LWM2M 服务器的短服务器 ID。
- ip_address：服务器的 IP 地址。
- port：服务器的 UDP 端口。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_NOT_FOUND：没有与该短服务器 ID 相对应的服务器对象实例。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_session_register_dtls

启动与 LWM2M 服务器的安全会话

原型

```
UINT nx_lwm2m_client_session_register_dtls(NX_LWM2M_CLIENT_SESSION *session_ptr,
    NX_LWM2M_ID server_id, ULONG ip_address, UINT port, NX_SECURE_DTLS_SESSION
    *dtls_session_ptr, UINT dtls_local_port);
```

说明

此服务使用安全 DTLS 连接对 LWM2M 服务器执行"注册"操作。该服务器在服务器对象中必须有相应的服务器实例。

在调用此函数之前，必须使用正确的密码套件和密钥材料来配置 DTLS 会话。必须定义 NX_SECURE_ENABLE_DTLS。

每个 DTLS 会话都使用自己的 UDP 套接字进行通信，因此，当应用程序创建多个安全会话时，各个会话的本地端口 dtls_local_port 不得相同。

如果注册成功，则 LWM2M 客户端会处理来自服务器的进一步操作，并且会定期发送"更新"消息，直到该客户端注销为止。

此调用会中止所有的当前活动会话，并将其替换为新的 LWM2M 服务器会话。

参数

- session_ptr：指向 LWM2M 客户端会话控制块的指针。
- server_id：LWM2M 服务器的短服务器 ID。
- ip_address：服务器的 IP 地址。
- port：服务器的 UDP 端口。
- dtls_session_ptr：要用于 LWM2M 会话的 DTLS 会话。
- dtls_local_port：用于 DTLS 会话的本地 UDP 端口。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_NOT_FOUND：没有与该短服务器 ID 相对应的服务器对象实例。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_session_update

更新与 LWM2M 服务器的会话

原型

```
UINT nx_lwm2m_client_session_update(NX_LWM2M_CLIENT_SESSION *session_ptr);
```

说明

此服务对 LWM2M 服务器执行"更新"操作。

参数

- session_ptr：指向 LWM2M 客户端会话控制块的指针。

返回值

- NX_SUCCESS：操作成功。

- NX_LWM2M_NOT_REGISTERED：该客户端未向服务器注册。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_client_unlock

解锁 LWM2M 客户端

原型

```
UINT nx_lwm2m_client_unlock(NX_LWM2M_CLIENT *client_ptr);
```

说明

此服务通过调用 nx_lwm2m_client_unlock()，将以前锁定的 LWM2M 客户端解锁。

参数

- client_ptr：指向 LWM2M 客户端控制块的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_firmware_create

创建固件更新对象

原型

```
UINT nx_lwm2m_firmware_create(NX_LWM2M_FIRMWARE *firmware_ptr,
    NX_LWM2M_CLIENT *client_ptr, UINT protocols,
    NX_LWM2M_FIRMWARE_PACKAGE_CALLBACK package_callback,
    NX_LWM2M_FIRMWARE_PACKAGE_URI_CALLBACK package_uri_callback,
    NX_LWM2M_FIRMWARE_UPDATE_CALLBACK update_callback);
```

说明

此服务初始化固件更新对象，并将该对象添加到以前创建的 LWM2M 客户端。该固件更新对象实现了用于与 LWM2M 服务器通信的资源，但应用程序必须提供回调，才能在固件上实现实际的操作（下载、存储以及更新固件）。

protocols 参数指出应用程序支持哪些协议来通过“包 URI”资源检索固件，其中定义了以下标志：

NX_LWM2M_FIRMWARE_PROTOCOL_COAP、NX_LWM2M_FIRMWARE_PROTOCOL_COAPS、NX_LWM2M_FIRMWARE_PROTOCOL_HTTP 和 NX_LWM2M_FIRMWARE_PROTOCOL_HTPPS。

参数

- firmware_ptr：指向固件对象控制块的指针。
- client_ptr：指向以前所创建 LWM2M 客户端的指针。
- protocols：这是一些标志，它们指出“包 URI”资源支持哪些协议。
- package_callback：必须为 NULL [待定]。
- package_uri_callback：用于实现“包 URI”资源的回调。
- update_callback：用于实现“更新”资源的回调。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_firmware_package_info_set

设置固件更新包信息

原型

```
UINT nx_lwm2m_firmware_package_info_set(NX_LWM2M_FIRMWARE *firmware_ptr,
                                const CHAR *name, const CHAR *version);
```

说明

此服务更改固件更新对象的“包名称”(6) 和“包版本”(7) 资源的值。

参数

- firmware_ptr：指向固件更新对象的指针。
- name：包名称的新值。
- version：包版本的新值。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_firmware_result_set

设置固件更新结果

原型

```
UINT nx_lwm2m_firmware_result_set(NX_LWM2M_FIRMWARE *firmware_ptr, UCHAR result);
```

说明

此服务更改固件更新对象的“更新结果”(5) 资源的值。

参数

- firmware_ptr：指向固件更新对象的指针。
- result：“更新结果”资源的新值。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_firmware_state_set

设置固件更新状态

原型

```
UINT nx_lwm2m_firmware_state_set(NX_LWM2M_FIRMWARE *firmware_ptr, UCHAR state);
```

说明

此服务更改固件更新对象的“状态”(3) 资源的值。

参数

- firmware_ptr：指向固件更新对象的指针。

- state：“状态”资源的新值。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_object_resource_changed

指出对象实例的资源值已更改

原型

```
UINT nx_lwm2m_object_resource_changed(NX_LWM2M_OBJECT *object_ptr,
    NX_LWM2M_OBJECT_INSTANCE *instance_ptr, const NX_LWM2M_RESOURCE *resource);
```

说明

此服务由对象实现用来向 LWM2M 客户端发出信号，指出其某个资源值已更改。当 LWM2M 服务器观察到资源时，LWM2M 客户端就会发送通知。

参数

- object_ptr：指向对象实现的指针。
- instance_ptr：指向对象实例的指针。
- resource：指向一个结构的指针，该结构说明已更改的资源。

返回值

- NX_SUCCESS：操作成功。
- NX_PTR_ERROR：指针无效。

nx_lwm2m_resource_get_boolean

获取布尔资源的值

原型

```
UINT nx_lwm2m_resource_get_boolean(const NX_LWM2M_RESOURCE *value,
                                   NX_LWM2M_BOOL *bool_ptr);
```

说明

此服务检索布尔资源的值。

参数

- value：指向资源值说明的指针。
- bool_ptr：指向目标布尔值的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_ENCODING：该资源值不是布尔值。

nx_lwm2m_resource_get_float32

获取 32 位浮点资源的值

原型

```
UINT nx_lwm2m_resource_get_float32(const NX_LWM2M_RESOURCE *value,
                                   NX_LWM2M_FLOAT32 *float32_ptr);
```

说明

此服务检索 32 位浮点资源的值。

参数

- value：指向资源值说明的指针。
- float32_ptr：指向目标 32 位浮点值的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_ENCODING：该资源值不是浮点值。

nx_lwm2m_resource_get_float64

获取 64 位浮点资源的值

原型

```
UINT nx_lwm2m_resource_get_float64(const NX_LWM2M_RESOURCE *value,
                                   NX_LWM2M_FLOAT64 *float64_ptr);
```

说明

此服务检索 64 位浮点资源的值。

参数

- value：指向资源值说明的指针。
- float64_ptr：指向目标 64 位浮点值的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_ENCODING：该资源值不是浮点值。

nx_lwm2m_resource_get_integer32

获取 32 位整数资源的值

原型

```
UINT nx_lwm2m_resource_get_integer32(const NX_LWM2M_RESOURCE *value,
                                     NX_LWM2M_INT32 *int32_ptr);
```

说明

此服务检索 32 位整数资源的值。

参数

- value：指向资源值说明的指针。
- int32_ptr：指向目标 32 位整数值的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_ENCODING：该资源值不是整数值，或者该整数值在 32 位数字中放不下。

nx_lwm2m_resource_get_integer64

获取 64 位整数资源的值

原型

```
UINT nx_lwm2m_resource_get_integer64(const NX_LWM2M_RESOURCE *value,
                                     NX_LWM2M_INT64 *int64_ptr);
```

说明

此服务检索 64 位整数资源的值。

参数

- value：指向资源值说明的指针。
- int64_ptr：指向目标 64 位整数值的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_ENCODING：该资源值不是整数值。

nx_lwm2m_resource_get_objlnk

获取对象链接资源的值

原型

```
UINT nx_lwm2m_resource_get_objlnk(const NX_LWM2M_RESOURCE *value,
                                  NX_LWM2M_OBJLNK *objlnk_ptr);
```

说明

此服务检索对象链接资源的值。

参数

- value：指向资源值说明的指针。
- objlnk_ptr：指向目标对象链接值的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_ENCODING：该资源值不是对象链接值。

nx_lwm2m_resource_get_opaque

获取不透明资源的值

原型

```
UINT nx_lwm2m_resource_get_opaque(const NX_LWM2M_RESOURCE *value,
            const VOID **opaque_ptr_ptr, UINT *opaque_length_ptr);
```

说明

此服务检索不透明资源的值。

不透明资源值是由指向缓冲区的指针以及长度所组成。

参数

- value：指向资源值说明的指针。
- opaque_ptr_ptr：指向目标不透明缓冲区指针的指针。
- opaque_length_ptr：指向目标不透明缓冲区长度的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_ENCODING：该资源值不是不透明值。

nx_lwm2m_resource_get_string

获取字符串资源的值

原型

```
UINT nx_lwm2m_resource_get_string(const NX_LWM2M_RESOURCE *value,
            const CHAR **string_ptr_ptr, UINT *string_length_ptr);
```

说明

此服务检索字符串资源的值。

参数

- value：指向资源值说明的指针。
- string_ptr_ptr：指向目标字符串指针的指针。
- string_length_ptr：指向目标字符串长度的指针。可以为 NULL（如果该字符串以 null 结尾）。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_ENCODING：该资源值不是字符串值。

nx_lwm2m_resource_multiple_get_boolean

获取布尔资源多重资源实例的值

原型

```
UINT nx_lwm2m_resource_multiple_get_boolean(const NX_LWM2M_RESOURCE *value,
        int index, NX_LWM2M_ID *instance_id_ptr, NX_LWM2M_BOOL *bool_ptr);
```

说明

此服务从多重资源检索布尔资源实例的值。

参数

- value：指向多重资源值说明的指针。
- index：要检索的实例在资源值数组中的下标。
- instance_id_ptr：指向目标实例 ID 的指针。
- bool_ptr：指向目标布尔值的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_PARAMETER：下标超出范围。
- NX_LWM2M_BAD_ENCODING：该资源不是多重资源，或资源值不是布尔值。

nx_lwm2m_resource_multiple_get_float32

获取 32 位浮点多重资源实例的值

原型

```
UINT nx_lwm2m_resource_multiple_get_float32(const NX_LWM2M_RESOURCE *value,
    int index, NX_LWM2M_ID *instance_id_ptr, NX_LWM2M_FLOAT32 *float32_ptr);
```

说明

此服务从多重资源检索 32 位浮点资源实例的值。

参数

- value：指向多重资源值说明的指针。
- index：要检索的实例在资源值数组中的下标。
- instance_id_ptr：指向目标实例 ID 的指针。
- float32_ptr：指向目标 32 位浮点值的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_PARAMETER：下标超出范围。
- NX_LWM2M_BAD_ENCODING：该资源不是多重资源，或资源值不是浮点值。

nx_lwm2m_resource_multiple_get_float64

获取 64 位浮点多重资源实例的值

原型

```
UINT nx_lwm2m_resource_multiple_get_float64(const NX_LWM2M_RESOURCE *value,
    int index, NX_LWM2M_ID *instance_id_ptr, NX_LWM2M_FLOAT64 *float64_ptr);
```

说明

此服务从多重资源检索 64 位浮点资源实例的值。

参数

- value：指向多重资源值说明的指针。
- index：要检索的实例在资源值数组中的下标。
- instance_id_ptr：指向目标实例 ID 的指针。
- float64_ptr：指向目标 64 位浮点值的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_PARAMETER：下标超出范围。
- NX_LWM2M_BAD_ENCODING：该资源不是多重资源，或资源值不是浮点值。

nx_lwm2m_resource_multiple_get_integer32

获取 32 位整数多重资源实例的值

原型

```
UINT nx_lwm2m_resource_multiple_get_integer32(const NX_LWM2M_RESOURCE *value,
        int index, NX_LWM2M_ID *instance_id_ptr, NX_LWM2M_INT32 *int32_ptr);
```

说明

此服务从多重资源检索 32 位整数资源实例的值。

参数

- value：指向多重资源值说明的指针。
- index：要检索的实例在资源值数组中的下标。
- instance_id_ptr：指向目标实例 ID 的指针。
- int32_ptr：指向目标 32 位整数值的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_PARAMETER：下标超出范围。
- NX_LWM2M_BAD_ENCODING：该资源不是多重资源，或资源值不是整数值，或者该整数值在 32 位数字中放不下。

nx_lwm2m_resource_multiple_get_integer64

获取 64 位整数多重资源实例的值

原型

```
UINT nx_lwm2m_resource_multiple_get_integer64(const NX_LWM2M_RESOURCE *value,
        int index, NX_LWM2M_ID *instance_id_ptr, NX_LWM2M_INT64 *int64_ptr);
```

说明

此服务从多重资源检索 64 位整数资源实例的值。

参数

- value：指向多重资源值说明的指针。
- index：要检索的实例在资源值数组中的下标。
- instance_id_ptr：指向目标实例 ID 的指针。
- int64_ptr：指向目标 64 位整数值的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_PARAMETER：下标超出范围。
- NX_LWM2M_BAD_ENCODING：该资源不是多重资源，或资源值不是整数值。

nx_lwm2m_resource_multiple_get_objlnk

获取对象链接多重资源实例的值

原型

```
UINT nx_lwm2m_resource_multiple_get_objlnk(const NX_LWM2M_RESOURCE *value,
    int index, NX_LWM2M_ID *instance_id_ptr, NX_LWM2M_OBJLNK *objlnk_ptr);
```

说明

此服务从多重资源检索对象链接资源实例的值。

参数

- value：指向多重资源值说明的指针。
- index：要检索的实例在资源值数组中的下标。
- instance_id_ptr：指向目标实例 ID 的指针。
- objlnk_ptr：指向目标对象链接值的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_PARAMETER：下标超出范围。
- NX_LWM2M_BAD_ENCODING：该资源不是多重资源，或资源值不是对象链接值。

nx_lwm2m_resource_multiple_get_opaque

获取不透明多重资源实例的值

原型

```
UINT nx_lwm2m_resource_multiple_get_opaque(const NX_LWM2M_RESOURCE *value,
    int index, NX_LWM2M_ID *instance_id_ptr, const VOID **opaque_ptr_ptr,
    UINT *opaque_length_ptr);
```

说明

此服务从多重资源检索不透明资源实例的值。

不透明资源值是由指向缓冲区的指针以及长度所组成。

参数

- value：指向多重资源值说明的指针。
- index：要检索的实例在资源值数组中的下标。
- instance_id_ptr：指向目标实例 ID 的指针。
- opaque_ptr_ptr：指向目标不透明缓冲区指针的指针。
- opaque_length_ptr：指向目标不透明缓冲区长度的指针。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_PARAMETER：下标超出范围。
- NX_LWM2M_BAD_ENCODING：该资源不是多重资源，或资源值不是不透明值。

nx_lwm2m_resource_multiple_get_string

获取字符串多重资源实例的值

原型

```
UINT nx_lwm2m_resource_multiple_get_string(const NX_LWM2M_RESOURCE *value,
    int index, NX_LWM2M_ID *instance_id_ptr, const CHAR **string_ptr_ptr,
    UINT *string_length_ptr);
```

说明

此服务从多重资源检索字符串资源实例的值。

参数

- value：指向多重资源值说明的指针。

- index：要检索的实例在资源值数组中的下标。
- instance_id_ptr：指向目标实例 ID 的指针。
- string_ptr_ptr：指向目标字符串指针的指针。
- string_length_ptr：指向目标字符串长度的指针。可以为 NULL（如果该字符串以 null 结尾）。

返回值

- NX_SUCCESS：操作成功。
- NX_LWM2M_BAD_PARAMETER：下标超出范围。
- NX_LWM2M_BAD_ENCODING：该资源不是多重资源，或实例值不是字符串值。

第 1 章 - Azure RTOS NetX POP3 客户端简介

2021/4/29 •

邮局协议版本 3 (POP3) 是一种协议，旨在为小型工作站提供邮件传输系统，用于访问 POP3 服务器上的客户端 maildrop 以检索客户端邮件。POP3 使用传输控制协议 (TCP) 服务来执行邮件传输。因此，POP3 是高度可靠的内容传输协议。

但是，POP3 不提供广泛的邮件处理操作。通常情况下，邮件由客户端下载，然后会从服务器的 maildrop 中删除。

NetX POP3 客户端要求

客户端要求

Azure RTOS NetX POP3 客户端 API 需要一个事先使用 nx_ip_create 创建的 NetX IP 实例，以及一个事先使用 nx_packet_pool_create 创建的 NetX 数据包池 。由于 NetX POP3 客户端使用 TCP 服务，因此必须先使用 nx_tcp_enable 调用启用 TCP，然后才能在同一 IP 实例上使用 NetX POP3 客户端服务。POP3 客户端使用 TCP 套接字通过服务器的 POP3 端口连接到 POP3 服务器。该端口通常设置为熟知的端口 110，但不要求 POP3 客户端和服务器使用该端口。

创建 POP3 客户端时使用的数据包池的大小可由用户根据数据包有效负载和可用数据包数量进行配置。如果数据包仅在 POP3 客户端创建服务中使用，则数据包有效负载不必超过 100-120 个字节，具体取决于用户名和密码或 APOP 摘要的长度。使用本地主机用户名的 USER 命令可能是 POP3 客户端发送的最大消息。可以在 nx_ip_create 中共享同一数据包池(IP 默认数据包池)，因为 IP 内部操作不需要使用非常大的数据包有效负载来发送和接收 TCP 控制数据。

但是，这在网络驱动程序使用与 POP3 客户端数据包池相同的数据包池的情况下可能并无益处。通常，接收数据包池的有效负载设置为网络接口的 IP 实例 MTU(通常为 1500 个字节)，这比 POP3 客户端消息大得多。传入 POP3 消息的数据通常比传出 POP3 客户端消息大得多

NetX POP3 客户端创建

POP3 客户端创建服务 nx_pop3_client_create 可创建 TCP 套接字并与 POP3 服务器相连接。

连接到 POP3 服务器后，POP3 客户端应用程序可以调用 nx_pop3_client_mail_items_get 来获取 maildrop 邮箱中的邮件项数：

```
UINT nx_pop3_client_mail_items_get(NX_POP3_CLIENT *client_ptr,
                                   UINT *number_mail_items,
                                   ULONG *maildrop_total_size)
```

如果一个或多个项位于客户端 maildrop 中，应用程序可以使用 nx_pop3_client_get_mail_item 服务来获取特定邮件项的大小：

```
UINT nx_pop3_client_mail_item_get(NX_POP3_CLIENT *client_ptr,
                                  UINT mail_item,
                                  ULONG *item_size)
```

maildrop 中的第一个邮件项位于索引 1 处。

若要获取实际的邮件，应用程序可以调用 nx_pop3_client_mail_item_get_message_data 服务来检索邮件数据包，直到服务指示 final_packet 输入参数接收到最后一个数据包为止：

```
UINT nx_pop3_client_mail_item_message_get(
                        NX_POP3_CLIENT *client_ptr,
                        NX_PACKET **recv_packet_ptr,
                        ULONG *bytes_retrieved,
                        UINT *final_packet)
```

若要删除特定邮件项，应用程序将调用与前面的 nx_pop3_client_get_mail_item 调用中使用的同一个索引来调用 nx_pop3_client_mail_item_delete。

可以使用 nx_pop3_client_delete 服务来删除客户端。请注意，应由应用程序使用 nx_packet_pool_delete 服务删除其不再使用的 POP3 客户端数据包池。

NetX POP3 客户端限制

NetX POP3 客户端实现中存在一些限制：

1. 虽然 NetX POP3 客户端确实将 DIGEST-MD5 用于客户端服务器身份验证交换来实现 APOP 身份验证，但却不支持身份验证命令。
2. NetX POP3 客户端并不会实现所有 POP3 命令（例如，TOP 命令或 UIDL 命令）。下面是它支持的命令列表：
   - NOOP
   - RSET

NetX POP3 客户端登录

NetX POP3 客户端必须向 POP3 服务器对自身进行身份验证（登录到服务器）才能访问 maildrop。为此，它可以使用 USER/PASS 命令并提供 POP3 服务器已知的用户名和密码，或者使用下面所述的 APOP 命令和 MD5 摘要。

用户名通常是完全限定的域名（包含本地部分和域名，用“@”字符分隔）。使用 POP3 命令 USER 和 PASS 时，客户端将以未加密方式通过 Internet 发送其用户名和密码。

若要避免明文用户名和密码带来的安全风险，可以通过在 nx_pop3_client_create 服务中设置 APOP_authentication 参数，将 NetX POP3 客户端配置为使用 APOP 身份验证：

```
UINT  nxd_pop3_client_create(NX_POP3_CLIENT *client_ptr,
                             UINT APOP_authentication,
                             NX_IP *ip_ptr,
                             NX_PACKET_POOL *packet_pool_ptr,
                             NXD_ADDRESS *server_ip_address, ULONG server_port,
                             CHAR *client_name,
                             CHAR *client_password)
```

对于仅接受 IPv4 的应用程序，则在 nx_pop3_client_create 服务中设置该参数：

```
UINT  nx_pop3_client_create(NX_POP3_CLIENT *client_ptr,
                            UINT APOP_authentication,
                            NX_IP *ip_ptr,
                            NX_PACKET_POOL *packet_pool_ptr,
                            ULONG server_ip_address,
                            ULONG server_port, CHAR *client_name,
                            CHAR *client_password)
```

当客户端发送 APOP 命令时，它会采用 MD5 摘要，其中包含服务器域、从服务器问候语中提取的本地时间和进程 ID，以及客户端密码。POP3 服务器将创建包含相同信息的 MD5 摘要，如果其 MD5 摘要与客户端的 MD5 摘要匹配，则客户端身份验证将通过。

如果 APOP 身份验证失败，NetX POP3 客户端将尝试使用 USER/PASS 命令进行身份验证。

POP3 客户端 maildrop

客户端邮件存储在 POP3 服务器上的邮箱或"maildrop"中。POP3 服务器上的客户端 maildrop 表示为从 1 开始的邮件项列表。也就是说，在 maildrop 列表中，每封邮件都由对应的索引表示，并且第一个邮件项位于索引 1（而不是零）处。POP3 命令通过此列表中的对应索引引用特定的邮件项。

POP3 协议状态机

POP3 协议要求客户端和服务器都维护 POP3 会话的状态。首先，客户端将尝试连接到 POP3 服务器。如果成功，它将进入 POP3 协议，该协议具有由 RFC 1939 定义的三种不同状态。初始状态为授权状态，在此状态下，它必须向服务器标识自身。在授权状态下，POP3 客户端只能发出 USER 和 PASS 命令（按照该顺序）或 APOP 命令。

POP3 客户端身份验证通过后，客户端会话将进入事务状态。在此状态下，客户端可以下载邮件和请求删除邮件。事务状态下允许的命令有 LIST、STAT、RETR、DELE、RSET 和 QUIT。通常，POP3 客户端会发送一个 STAT 命令，然后发送一系列 RETR 命令（每个命令分别针对 maildrop 中的一个邮件项）。

客户端发出 QUIT 命令后，POP3 会话将进入更新状态，在此状态下，它会发起从服务器断开 TCP 连接。以后若要下载邮件，POP3 客户端应用程序可以随时调用 `nx_pop3_client_mail_items_get` 来检查 maildrop 中是否有新邮件。

POP3 服务器回复代码

- +OK：服务器使用此回复来接受客户端命令。服务器可以在"+OK"后面加入其他信息但不能假定客户端将处理这些信息，除非在下载邮件数据或者发出 LIST 命令或 DELE 命令的情况下。在后一种情况下，命令后面的"参数"引用邮件项在客户端 maildrop 中的索引。
- -ERR：服务器使用此回复来拒绝客户端命令。服务器可以在"-ERR"后面发送其他信息，但不能假定客户端将处理这些信息。

示例 POP3 客户端-服务器会话

使用 USER/PASS 的基本 POP3 示例：

```
S: <wait for connection on TCP port 110>
C: <open connection>
S: +OK POP3 server ready <1896.697170952@dbc.mtview.ca.us>
C: USER mrose
S: +OK mrose is valid
C: PASS mvan99
S: +OK mrose is logged in
C: STAT
S: +OK 2 320
C: RETR 1
S: +OK 120 octets
S: <the POP3 server sends message 1>
S: .
C: DELE 1
S: +OK message 1 deleted
C: RETR 2
S: +OK 200 octets
S: <the POP3 server sends message 2>
S: .
C: DELE 2
S: +OK message 2 deleted
C: QUIT
S: +OK POP3 server signing off (maildrop empty)
C: <close connection>
S: <wait for next connection>
```

使用 APOP（和 LIST 而不是 STAT）的基本 POP3 示例：

```
S: <wait for connection on TCP port 110>
C: <open connection>
S: +OK POP3 server ready <1896.697170952@dbc.mtview.ca.us>
C: APOP mrose c4c9334bac560ecc979e58001b3e22fb
S: +OK mrose's maildrop has 2 messages (320 octets)
C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
C: RETR 1
S: +OK 120 octets
S: <the POP3 server sends message 1>
S: .
C: DELE 1
S: +OK message 1 deleted
C: RETR 2
S: +OK 200 octets
S: <the POP3 server sends message 2>
S: .
C: DELE 2
S: +OK message 2 deleted
C: QUIT
S: +OK dewey POP3 server signing off (maildrop empty)
C: <close connection>
S: <wait for next connection>
```

NetX POP3 客户端支持的 RFC

NetX POP3 客户端符合 RFC 1939。

第 2 章 - 安装和使用 Azure RTOS NetX POP3 客户端

2021/4/29 •

NetX POP3 客户端包括一个源文件、一个头文件和一个演示文件。其中有两个用于 MD5 摘要式服务的附加文件。此外，还有一个用户指南 PDF 文件（本文档）。

- nx_pop3_client.c：用于 NetX POP3 客户端 API 的 C 源文件
- nx_pop3_client.h：用于 NetX POP3 客户端 API 的 C 头文件
- demo_netxduo_pop3_client.c：用于 POP3 客户端创建和会话启动的演示文件
- nx_md5.c：定义 MD5 摘要式服务的 C 源文件
- nx_md5.h：定义 MD5 摘要式服务的 C 头文件
- nx_pop3_client.pdf：NetX POP3 客户端用户指南

若要使用 NetX POP3 客户端，可将之前提到的全部分发文件复制到安装 NetX 的同一目录中。例如，如果 NetX 安装在目录“\threadx\mcf5272\green”中，则应将 nx_md5.h、nx_md5.c、nx_pop3_client.h 和 nx_pop3_client.c 文件复制到此目录中。

使用 NetX POP3 客户端

若要使用 NetX POP3 客户端服务，应用程序必须将 nx_pop3_client.c 添加到其生成项目中。应用程序代码的 tx_api.h 和 nx_api.h 后面必须加入 nx_md5.h、nx_pop3.h 和 nx_pop3_client.h，才能使用 ThreadX 和 NetX。

这些文件的编译方式必须与其他应用程序文件相同，并且对象代码必须与应用程序的文件一起进行链接。这就是使用 NetX POP3 客户端所需执行的所有操作。

使用 NetX POP3 客户端的小示例

下面图 1 介绍如何使用 NetX POP3 客户端服务的示例。此演示在第 37 行和第 38 行设置两个回调，以便提供邮件下载和会话完成通知。在第 76 行创建 POP3 客户端数据包池。在第 88 行创建 IP 线程任务。请注意，此数据包池也可用于 POP3 客户端数据包池。在第 107 行的 IP 任务中启用 TCP。

在第 133 行的应用程序线程入口函数 demo_thread_entry 中创建 POP3 客户端。这是因为 nx_pop3_client_create 服务也会尝试与 POP3 服务器建立 TCP 连接。如果成功，应用程序会使用第 149 行的 nx_pop3_client_mail_items_get 服务，在 POP3 服务器上查询其邮箱中的项目数。

如果有一个或多个项目，则应用程序会对每个邮件项反复执行 while 循环，以下载邮件。在第 149 行，通过 nx_pop3_client_mail_item_get 调用发出 RETR 请求。如果成功，应用程序会使用第 177 行的 nx_pop3_client_mail_item_message_get 服务下载数据包，直至在收到的消息中检测到最后一个数据包（第 196 行）。最后，如果成功下载，应用程序会通过第 199 行的 nx_pop3_client_mail_item_delete 调用删除邮件项。RFC 1939 建议 POP3 客户端指示服务器删除下载的邮件项，以防客户端邮箱中的邮件堆积。不过服务器可能会自动执行此操作。

下载所有邮件项后，或在 POP3 客户端服务调用失败的情况下，应用程序会退出循环，并使用 nx_pop3_client_delete 服务删除 POP3 客户端（第 217 行）。

```
/*
    demo_netxduo_pop3.c

    This is a small demo of POP3 Client on the NetX TCP/IP stack.
    This demo relies on Thread, NetX and POP3 Client API to conduct
```

```
    a POP3 mail session.
*/

#include "tx_api.h"
#include "nx_api.h"
#include "nx_pop3_client.h"

#define DEMO_STACK_SIZE        4096
#define CLIENT_ADDRESS         IP_ADDRESS(192,2,2,61)
#define SERVER_ADDRESS         IP_ADDRESS(192,2,2,89)
#define SERVER_PORT            110

/* Replace the 'ram' driver with your own Ethernet driver. */
void _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);

/* Set up the POP3 Client. */

TX_THREAD          demo_client_thread;
NX_POP3_CLIENT     demo_client;
NX_PACKET_POOL     client_packet_pool;
NX_IP              client_ip;

#define PAYLOAD_SIZE 500

/* Set up Client thread entry point. */
void     demo_thread_entry(ULONG info);

/* Shared secret is the same as password. */

#define LOCALHOST              "recipient@domain.com"
#define LOCALHOST_PASSWORD     "testpwd"

/* 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)
{

UINT      status;
UCHAR     *free_memory_pointer;

    /* Setup the working pointer. */
    free_memory_pointer = first_unused_memory;

    /* Create a client thread. */
    tx_thread_create(&demo_client_thread, "Client", demo_thread_entry, 0,
                     free_memory_pointer, DEMO_STACK_SIZE, 1, 1,
                     TX_NO_TIME_SLICE, TX_AUTO_START);

    free_memory_pointer = free_memory_pointer + DEMO_STACK_SIZE;

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

    /* The demo client username and password is the authentication
    data used when the server attempts to authentication the client. */

    /* Create Client packet pool. */
    status = nx_packet_pool_create(&client_packet_pool,"POP3 Client Packet Pool",
                        PAYLOAD_SIZE, free_memory_pointer, (PAYLOAD_SIZE * 10));
    if (status != NX_SUCCESS)
    {
        return;
    }
```

```
    /* Update pointer to unallocated (free) memory. */
    free_memory_pointer = free_memory_pointer + (PAYLOAD_SIZE * 10);

    /* Create IP instance for demo Client */
    status = nx_ip_create(&client_ip, "POP3 Client IP Instance",
                          CLIENT_ADDRESS, 0xFFFFFF00UL, &client_packet_pool,
                          _nx_ram_network_driver, free_memory_pointer,
                          2048, 1);

    if (status != NX_SUCCESS)
    {
        return;
    }

    /* Update pointer to unallocated (free) memory. */
    free_memory_pointer = free_memory_pointer + 2048;

    /* Enable ARP and supply ARP cache memory. */
    nx_arp_enable(&client_ip, (void **) free_memory_pointer, 1024);

    /* Update pointer to unallocated (free) memory. */
    free_memory_pointer = free_memory_pointer + 1024;

    /* Enable TCP and ICMP for Client IP. */
    nx_tcp_enable(&client_ip);
    nx_icmp_enable(&client_ip);

    return;
}

/* Define the application thread entry function. */

void demo_thread_entry(ULONG info)
{

UINT          status;
UINT          mail_item, number_mail_items;
UINT          bytes_downloaded = 0;
UINT          final_packet = NX_FALSE;
ULONG         total_size, mail_item_size, bytes_retrieved;
NX_PACKET     *packet_ptr;

    /* Let the IP instance get initialized with driver parameters. */
    tx_thread_sleep(40);

    /* Create a NetX POP3 Client instance with no byte or block memory pools.
    Note that it uses its password for its APOP shared secret. */
    status = nx_pop3_client_create(&demo_client,
                        NX_TRUE,
                        &client_ip, &client_packet_pool, SERVER_ADDRESS,
                        SERVER_PORT, LOCALHOST, LOCALHOST_PASSWORD);

    /* Check for error. */
    if (status != NX_SUCCESS)
    {
        status = nx_pop3_client_delete(&demo_client);

        /* Abort. */
        return;
    }

    /* Find out how many items are in our mailbox. */
    status = nx_pop3_client_mail_items_get(&demo_client, &number_mail_items, &total_size);

    printf("Got %d mail items, total size%d \n", number_mail_items, total_size);

    /* If nothing in the mailbox, disconnect. */
    if (number_mail_items == 0)
```

```
    {
        nx_pop3_client_delete(&demo_client);

        return;
    }

    /* Download all mail items. */
    mail_item = 1;

    while (mail_item <= number_mail_items)
    {

        /* This submits a RETR request and gets the mail message size. */
        status = nx_pop3_client_mail_item_get(&demo_client, mail_item, &mail_item_size);

        /* Loop to get all mail message packets until the mail item is completely downloaded. */

        while((final_packet == NX_FALSE) && (status == NX_SUCCESS))
        {
            status = nx_pop3_client_mail_item_message_get(&demo_client, &packet_ptr,
                                                          &bytes_retrieved,
                                                          &final_packet);

            if (status != NX_SUCCESS)
            {
                break;
            }

            if (bytes_retrieved != 0)
            {

            printf("Received %d bytes of data for item %d: %s\n",
                    packet_ptr -> nx_packet_length,
                    mail_item, packet_ptr -> nx_packet_prepend_ptr);
            }

            nx_packet_release(packet_ptr);

            /* Determine if this is the last data packet. */
            if (final_packet)
            {
                /* It is. Let the server know it can delete this mail item. */
                status = nx_pop3_client_mail_item_delete(&demo_client, mail_item);
            }

            /* Keep track of how much mail message data is left. */
            bytes_downloaded += bytes_retrieved;
        }

        /* Get the next mail item. */
        mail_item++;

        tx_thread_sleep(100);
    }

    /* Disconnect from the POP3 server. */
    status = nx_pop3_client_quit(&demo_client);

    /* Delete the POP3 Client. This will not delete the Client packet pool. */
    status = nx_pop3_client_delete(&demo_client);

}
```

图 1. NetX POP3 客户端应用程序示例

POP3 客户端配置选项

NetX POP3 客户端有多个配置选项。下面是详细描述的所有选项的列表：

- NX_POP3_CLIENT_PACKET_TIMEOUT：此选项可定义 POP3 客户端分配数据包时的等待选项（以秒为单位）。默认值为 1 秒钟。
- NX_POP3_CLIENT_CONNECTION_TIMEOUT：此选项可定义 POP3 客户端连接到 POP3 服务器时的等待选项（以秒为单位）。默认值为 30 秒。
- NX_POP3_CLIENT_DISCONNECT_TIMEOUT：此选项可定义 POP3 客户端断开与 POP3 服务器的连接时的等待选项（以秒为单位）。默认值为 2 秒。
- NX_POP3_TCP_SOCKET_SEND_WAIT：此选项可在 nx_tcp_socket_send 服务调用中设置等待选项（以秒为单位）。默认值为 2 秒。
- NX_POP3_SERVER_REPLY_TIMEOUT：此选项可在 nx_tcp_socket_receive 服务调用中设置服务器回复客户端请求时的等待选项。默认值为 10 秒。
- NX_POP3_CLIENT_TCP_WINDOW_SIZE：此选项可设置客户端 TCP 接收窗口的大小。此选项应设置为 IP 实例 MTU 大小减去 IP 和 TCP 标头的值。默认值为 1460。
- NX_POP3_MAX_USERNAME：此选项可设置 POP3 客户端用户名的缓冲区大小。默认值为 40 字节。
- NX_POP3_MAX_PASSWORD：此选项可设置 POP3 客户端密码的缓冲区大小。默认值为 20 字节。

第 3 章 - Azure RTOS NetX POP3 客户端服务说明

2021/4/29 •

本章按字母顺序介绍了所有 Azure RTOS NetX POP3 客户端服务（如下所列）。

在以下 API 说明的"返回值"部分中，以粗体显示的值不受定义用于禁用 API 错误检查的 NX_DISABLE_ERROR_CHECKING 影响，而不以粗体显示的值则完全禁用。

- nx_pop3_client_create：创建 POP3 客户端实例
- nx_pop3_client_delete：删除 POP3 客户端实例
- nx_pop3_client_mail_item_get：从服务器 maildrop 中删除客户端邮件项
- nx_pop3_client_mail_item_get：检索特定大小的邮件
- nx_pop3_client_mail_items_get：获取 maildrop 中的邮件项数
- nx_pop3_client_mail_item_message_get：下载特定邮件
- nx_pop3_client_mail_item_size_get：获取特定邮件项的大小

nx_pop3_client_create

创建 POP3 客户端实例

原型

```
UINT nx_pop3_client_create(NX_POP3_CLIENT *client_ptr,
                        UINT APOP_authentication, NX_IP *ip_ptr,
                        NX_PACKET_POOL *packet_pool_ptr,
                        ULONG server_ip_address,
                        ULONG server_port,
                        CHAR *client_name, CHAR *client_password);
```

说明

此服务用于创建 POP3 客户端实例，并使用输入参数来设置其配置。

输入参数

- client_ptr：指向要创建的客户端的指针
- APOP_authentication：启用 APOP 身份验证
- ip_ptr：指向 IP 实例的指针
- packet_pool_ptr：指向客户端数据包池的指针
- server_ip_address：POP3 服务器地址
- server_port：POP3 服务器端口
- client_name：指向客户端名称的指针
- client_password：指向客户端密码的指针

返回值

- **NX_SUCCESS** (0x00)：已成功创建客户端
- **status**：NetX 和 ThreadX 服务调用处于完成状态
- NX_PTR_ERROR (0x07)：输入指针参数无效
- NX_POP3_PARAM_ERROR (0xB1)：非指针输入无效

允许调用自

应用程序代码

示例

```
/* Create the POP3 Client. */

/* Create username and password for our POP3 Client mail drop. */
#define LOCALHOST                   "recipient@expresslogic.com"
#define LOCALHOST_PASSWORD          "pass"
#define POP3_SERVER_ADDRESS         IP_ADDRESS(192,2,2,92
#define POP3_SERVER_PORT            110

/* Create a NetX POP3 Client instance. */
ULONG server_ip_address = IP_ADDRESS(1,2,3,4);
status = nx_pop3_client_create(&demo_client,
        NX_FALSE /* disable APOP authentication */,
        &client_ip, &client_packet_pool,
        server_ip_address, POP3_SERVER_PORT,
        LOCALHOST, LOCALHOST_PASSWORD);

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

nx_pop3_client_delete

删除 POP3 客户端实例

原型

```
UINT nx_pop3_client_delete(NX_POP3_CLIENT *client_ptr)
```

说明

此服务用于删除之前创建的 POP3 客户端。请注意，此服务不会删除 POP3 客户端数据包池。如果不再需要使用数据包池，则设备应用程序必须单独删除此资源。

输入参数

- client_ptr：指向要删除的客户端的指针

返回值

- **NX_SUCCESS** (0x00)：已成功删除客户端
- NX_PTR_ERROR (0x07)：输入指针参数无效

允许调用自

应用程序代码

示例

```
/* Delete the POP3 Client. */
status = nx_pop3_client_delete (&demo_client);

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

nx_pop3_client_mail_item_delete

从客户端 maildrop 中删除指定的邮件项

原型

```
UINT nx_pop3_client_mail_items_delete(NX_POP3_CLIENT *client_ptr,
                                      UINT mail_index)
```

说明

此服务用于从客户端 maildrop 中删除指定的邮件项。虽然一些 POP3 服务器可能会在收到客户端请求后自动删除邮件项，但此服务是在下载邮件项后使用的。

输入参数

- client_ptr：指向客户端实例的指针
- mail_index：客户端 maildrop 的索引

返回值

- **NX_SUCCESS** (0x00)：已成功执行删除请求
- **NX_POP3_INVALID_MAIL_ITEM** (0xB2)：邮件项索引无效
- **NX_POP3_INSUFFICIENT_PACKET_PAYLOAD** (0xB6)：对于 POP3 请求，客户端数据包有效负载太小。
- **NX_POP3_SERVER_ERROR_STATUS** (0xB4)：服务器答复了错误状态
- NX_POP3_CLIENT_INVALID_INDEX (0xB8)：邮件索引输入无效
- NX_PTR_ERROR (0x07)：输入指针参数无效

允许调用自

应用程序代码

示例

```
ULONG     item_index;

/* Delete the POP3 Client mail item. */
status = nx_pop3_client_mail_item_delete(&demo_client, item_index);

/* If the server accepts the DELE request (and deletes the mail item), status =
   NX_SUCCESS. */
```

nx_pop3_client_mail_item_get

检索指定的邮件项

原型

```
UINT nx_pop3_client_mail_item_get(NX_POP3_CLIENT *client_ptr,
                                  UINT mail_item, ULONG *item_size)
```

说明

此服务用于发出 RETR 请求，以从客户端 maildrop 中检索由索引 mail_item 指定的邮件项。在发出 RETR 请求并从服务器收到肯定响应后，客户端可以使用 nx_pop3_client_mail_item_message_get 服务开始下载邮件。请注意，此服务还提供从服务器答复中提取的所请求邮件项的大小。

输入参数

- client_ptr：指向客户端实例的指针
- mail_item：客户端 maildrop 的索引
- item_size：指向邮件大小的指针

返回值

- **NX_SUCCESS** (0x00)：已成功检索邮件项

- **NX_POP3_INVALID_MAIL_ITEM** (0xB2)：邮件项索引无效
- **NX_POP3_INSUFFICIENT_PACKET_PAYLOAD** (0xB6)：对于 POP3 请求，客户端数据包有效负载太小。
- **NX_POP3_SERVER_ERROR_STATUS** (0xB4)：服务器答复了错误状态
- NX_POP3_CLIENT_INVALID_INDEX (0xB8)：邮件索引输入无效
- NX_PTR_ERROR (0x07)：输入指针参数无效

允许调用自

应用程序代码

示例

```
ULONG     item_size;

/* Retrieve the POP3 Client mail item. */
status = nx_pop3_client_mail_item_get (&demo_client, 1, &item_size);

/* If the mail item was successfully obtained, status = NX_SUCCESS. */
```

nx_pop3_client_mail_items_get

检索 maildrop 中的邮件项数

原型

```
UINT nx_pop3_client_mail_items_get(NX_POP3_CLIENT *client_ptr,
                                   UINT *number_mail_items,
                                   ULONG *maildrop_total_size)
```

说明

此服务用于发出 STAT 请求，以从客户端 maildrop 中检索邮件项数和邮件数据总大小。

输入参数

- client_ptr：指向客户端实例的指针
- number_mail_item：客户端 maildrop 中的邮件数
- maildrop_total_size：指向邮件总大小的指针

返回值

- **NX_SUCCESS** (0x00)：已成功检索邮件项
- **NX_POP3_INVALID_MAIL_ITEM** (0xB2)：邮件项索引无效
- **NX_POP3_INSUFFICIENT_PACKET_PAYLOAD** (0xB6)：对于 POP3 请求，客户端数据包有效负载太小。
- **NX_POP3_SERVER_ERROR_STATUS** (0xB4)：服务器答复了错误状态
- NX_PTR_ERROR (0x07)：输入指针参数无效

允许调用自

应用程序代码

示例

```
UINT      number_mail_items;
ULONG     maildrop_total_size;

/* Retrieve the size and number of items in POP3 Client maildrop. */
status = nx_pop3_client_mail_item_get (&demo_client, 1, &number_mail_items,
                                       &maildrop_total_size);

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

nx_pop3_client_mail_item_message_get

检索指定的邮件项

原型

```
UINT nx_pop3_client_mail_item_message_get(
                NX_POP3_CLIENT *client_ptr,
                NX_PACKET **recv_packet_ptr,
                ULONG *bytes_retrieved,
                UINT *final_packet)
```

说明

此服务用于检索邮件项、邮件大小以及是否为邮件中的最后一个数据包。如果 `final_packet` 为 NX_TRUE，则 `recv_packet_ptr` 指向的数据包是邮件项中的最后一个数据包。

输入参数

- client_ptr：指向客户端实例的指针
- recv_packet_ptr：接收到的邮件数据包
- number_mail_item：客户端 maildrop 中的邮件数
- maildrop_total_size：指向邮件总大小的指针

返回值

- **NX_SUCCESS** (0x00)：已成功检索邮件项
- **NX_POP3_CLIENT_INVALID_STATE** (0xB7)：对于 POP3 请求，客户端数据包有效负载太小。
- NX_PTR_ERROR (0x07)：输入指针参数无效

允许调用自

应用程序代码

示例

```
NX_PACKET     *recv_packet_ptr;
ULONG         bytes_retrieved;
UINT          final_packet;

/* Retrieve the size and number of items in POP3 Client maildrop. */
status = nx_pop3_client_mail_item_message_get (&demo_client, &recv_packet_ptr,
                                               bytes_retrieved, final_packet);

/* If the maildrop message packet was successfully obtained, status = NX_SUCCESS. */
```

nx_pop3_client_mail_item_size_get

检索指定邮件项的大小

原型

```
UINT nx_pop3_client_mail_item_size_get(
                NX_POP3_CLIENT *client_ptr,
                UINT mail_item, ULONG *size)
```

说明

此服务用于发出 LIST 请求，以获取指定邮件项的大小。

输入参数

- client_ptr：指向客户端实例的指针
- mail_item：客户端 maildrop 的索引
- size：指向邮件大小的指针

返回值

- **NX_SUCCESS** (0x00)：已成功检索邮件项
- **NX_POP3_INVALID_MAIL_ITEM** (0xB2)：邮件项索引无效
- **NX_POP3_INSUFFICIENT_PACKET_PAYLOAD** (0xB6)：对于 POP3 请求，客户端数据包有效负载太小。
- **NX_POP3_SERVER_ERROR_STATUS** (0xB4)：服务器答复了错误状态
- NX_POP3_CLIENT_INVALID_INDEX (0xB8)：邮件索引输入无效
- NX_PTR_ERROR (0x07)：输入指针参数无效

允许调用自

应用程序代码

示例

```
ULONG     size;
UINT      mail_item;

/* Retrieve the size of the specified mail item in POP3 Client maildrop. */
status = nx_pop3_client_mail_item_size_get (&demo_client, mail_item, &size);

/* If the maildrop message packet was successfully obtained, status = NX_SUCCESS. */
```

第 1 章 - Azure RTOS NetX 点对点协议 (PPP) 简介

2021/4/29 •

通常，NetX 应用程序通过以太网连接到实际的物理网络。这提供了快速、高效的网络访问。但是，在某些情况下，应用程序无法访问以太网。在这种情况下，应用程序仍然可以通过直接连接到另一个网络成员的串行接口连接到网络。用于管理此类连接的最常见软件协议是点对点协议 (PPP)。

尽管串行通信相对简单，但 PPP 有点复杂。PPP 实际上是由多个协议组成的，如链接控制协议 (LCP)、Internet 协议控制协议 (IPCP)、密码身份验证协议 (PAP) 和质询握手身份验证协议 (CHAP)。LCP 是 PPP 的主要协议。其中，链接的基本组件以点对点的方式进行动态协商。一旦链接的基本特征被成功协商，PAP 和/或 CHAP 就会被用来确保连接的对等节点是有效的。如果两个对等节点都有效，则使用 IPCP 来协商对等节点使用的 IP 地址。在 IPCP 完成后，PPP 就能发送和接收 IP 数据包。

NetX 主要将 PPP 视为设备驱动程序。nx_ppp_driver 函数被提供给 NetX IP 创建函数 nx_ip_create。否则，NetX 对 PPP 没有任何直接的了解。

PPP 串行通信

NetX PPP 包要求应用程序提供串行通信驱动程序。此驱动程序必须支持 8 位字符，也可以使用软件流控制。应用程序负责初始化此驱动程序，这应在创建 PPP 实例之前完成。

为了发送 PPP 数据包，必须将串行驱动程序输出字节例程提供给 PPP（在 nx_ppp_create 函数中指定）。为了传输整个 PPP 数据包，此串行驱动程序字节输出例程将被重复调用。串行驱动程序负责缓冲输出。在接收端，只要有新字节到达，应用程序的串行驱动程序就必须调用 PPP nx_ppp_byte_receive 函数。这通常是在中断服务例程 (ISR) 的上下文中完成的。nx_ppp_byte_receive 函数将传入字节置于循环缓冲区中，并提醒 PPP 接收线程它的存在。

PPP Over Ethernet 通信

NetX PPP 也可以通过以太网传输 PPP 消息，在这种情况下，NetX PPP 包需要应用程序提供以太网通信驱动程序。

为了通过以太网发送 PPP 数据包，必须将输出例程提供给 PPP（在 nx_ppp_packet_send_set 函数中指定）。为了传输整个 PPP 数据包，此输出例程将被重复调用。在接收端，只要有新数据包到达，应用程序的接收器就必须调用 PPP nx_ppp_packet_receive 函数。

PPP 数据包

PPP 利用 AHDLC 帧（HDLC 的子集）来封装所有 PPP 协议控制和用户数据。AHDLC 帧如下所示：

ᛁᛁ	ADDR	CTRL	ᛁᛁ	CRC	ᛁᛁ
7E	FF	03	[0-1502 字节]	2 字节	7E

每个 PPP 帧都有这样的整体外观。信息字段的前两个字节包含 PPP 协议类型。有效值的定义如下：

- C021：LCP
- 8021：IPCP
- C023：PAP
- C223：CHAP
- 0021：IP 数据包

如果有 0x0021 协议类型，则 IP 数据包紧随其后。否则，如果存在其他协议之一，则以下字节对应于相应的特定协议。

为了确保唯一的 0x7E 开始/结束帧标记并支持软件流控制，AHDLC 使用转义序列来表示各种字节值。0x7D 值指定对后面的字符进行编码，这基本上是与 0x20 互斥的原始字符。例如，头中 Ctrl 字段的 0x03 值由两个字节序列表示：7D 23。默认情况下，小于 0x20 的值以及"信息"字段中的 0x7E 和 0x7D 值都被转换为转义序列。请注意，转义序列也适用于 CRC 字段。

链接控制协议 (LCP)

LCP 既是主要的 PPP 协议，也是第一个运行的协议。LCP 负责协商各种 PPP 参数，包括最大接收单元 (MRU) 和要使用的身份验证协议（PAP、CHAP 或 NONE）。在 LCP 的两端都同意 PPP 参数之后，身份验证协议（若有）就开始运行。

密码身份验证协议 (PAP)

PAP 是一种相对简单的协议，它依赖于连接一端所提供的用户名和密码（如在 LCP 期间所协商）。然后，另一端验证此信息。如果正确，接受消息就会返回给发送方，然后 PPP 可以进入 IPCP 状态机。否则，如果用户名或密码不正确，连接就会被拒绝。

> **NOTE**
> 接口的两端都可以请求 PAP，但 PAP 通常只在一个方向上使用。

质询握手身份验证协议 (CHAP)

CHAP 是一种比 PAP 更复杂的身份验证协议。CHAP 身份验证器向它的对等节点提供名称和值。然后，对等节点使用所提供的名称来查找两个实体之间的共享"机密"。然后，对 ID、值和"机密"进行计算。此计算的结果在响应中返回。如果正确，PPP 就可以进入 IPCP 状态机。否则，如果结果不正确，连接就会被拒绝。

CHAP 的另一个有趣的地方是，它可以在连接建立后以随机的间隔发生。这用来防止连接在经过身份验证后被劫持。如果质询在这些随机时间之一失败，那么连接会立即终止。

> **NOTE**
> 接口的两端都可以请求 CHAP，但 CHAP 通常只在一个方向上使用。

Internet 协议控制协议 (IPCP)

IPCP 是在 PPP 通信可用于 NetX IP 数据传输之前要执行的最后一个协议。此协议的主要目的是让一个对等节点将其 IP 地址告知给另一对等节点。在 IP 地址设置后，NetX IP 数据传输就启用了。

数据传输

如前所述，NetX IP 数据包驻留在协议 ID 为 0x0021 的 PPP 帧中。所有收到的数据包都放置在一个或多个 NX_PACKET 结构中，并传输到 NetX 接收处理。在传输时，NetX 数据包内容被置于 AHDLC 帧中并传输。

PPP RFC

NetX PPP 符合 RFC1332、RFC1334、RFC1661、RFC1994 和相关 RFC。

第 2 章 - 安装和使用 Azure RTOS NetX 点对点协议 (PPP)

2021/4/30 •

本章介绍与安装、设置和使用 Azure RTOS NetX 点对点协议 (PPP) 组件相关的各种问题。

产品分发

可从 https://github.com/azure-rtos/netx 获取 Azure RTOS NetX 点对点协议 (PPP) 软件包。该软件包中包含以下文件：

- nx_ppp.h：NetX PPP 的头文件
- nx_ppp.c：NetX PPP 的 C 源文件
- nx_ppp.pdf：NetX PPP 的 PDF 说明
- demo_netx_ppp.c：NetX PPP 演示

PPP 安装

若要使用 NetX PPP，应将之前提到的全部分发文件复制到安装了 NetX 的同一目录中。例如，如果 NetX 安装在“\threadx\arm7\green”目录中，则应将 nx_ppp.h 和 nx_ppp.c 文件复制到该目录中。

使用 PPP

NetX PPP 易于使用。基本上来说，应用程序代码必须包含 nx_ppp.h 和 tx_api.h 或 nx_api.h，才能使用 ThreadX 或 NetX。在包含 nx_ppp.h 之后，应用程序代码即可发出本指南后文所指定的 PPP 函数调用。在生成过程中，应用程序还必须包含 nx_ppp.c。此文件必须采用与其他应用程序文件相同的方式进行编译，并且其对象窗体必须与应用程序文件一起链接。这就是使用 NetX PPP 所需的一切。

使用调制解调器

如果需要调制解调器才能连接到 Internet，则需要注意一些特别注意事项才能使用 NetX PPP 产品。基本上，使用调制解调器会引入其他初始化逻辑和适用于解决通信中断问题的逻辑。此外，大多数其他调制解调器逻辑均在 NetX PPP 上下文之外完成。将 NetX PPP 与调制解调器一起使用的基本流程如下所示：

1. 初始化调制解调器
2. 向 Internet 服务提供商 (ISP) 拨号
3. 等待连接
4. 等待用户 ID 提示
5. 启动 NetX PPP [PPP 运行中]
6. 通信中断
7. 停止 NetX PPP（或者通过 nx_ppp_restart 重启）

初始化调制解调器

调制解调器使用应用程序的低级别串行输出例程，通过一系列 ASCII 字符命令进行初始化（有关更多详细信息，请参阅调制解调器文档）。

向 Internet 服务提供商拨号

调制解调器使用应用程序的低级别串行输出例程，接受指示向 ISP 拨号。例如，用于向 ISP（号码 123-4567）拨号的典型 ASCII 字符串如下所示：

“ATDT123456\r”

等待连接

此时，应用程序会等待从调制解调器接收已建立连接的指示。这可通过在应用程序的低级别串行输入例程中查找字符来实现。通常，在建立连接后，调制解调器会返回 ASCII 字符串“CONNECT”。

等待用户 ID 提示

建立连接后，应用程序现在必须等待来自 ISP 的初始登录请求。这通常采用 ASCII 字符串形式，例如“Login?”

启动 NetX PPP

此时，可以启动 NetX PPP。这可通过先调用 nx_ppp_create 服务，再调用 nx_ip_create 服务来实现。可能还需要其他服务来启用 PAP 并设置 PPP IP 地址。有关更多信息，请参阅本指南的以下各节。

通信中断

启动 PPP 后，任何非 PPP 信息都会传递至应用程序指定给 nx_ppp_create 服务的“无效数据包处理”例程。通常，当与 ISP 中断通信时，调制解调器会发送 ASCII 字符串，例如“NO CARRIER”。当应用程序收到包含此类信息的非 PPP 数据包时，应停止 NetX PPP 实例，或通过 nx_ppp_restart API 重启 PPP 状态机。

停止 NetX PPP

停止 NetX PPP 非常简单。基本上来说，必须取消绑定并删除已创建的所有套接字。接下来，通过 nx_ip_delete 服务删除 IP 实例。删除 IP 实例后，应调用 nx_ppp_delete 服务以完成停止 PPP 的过程。此时，应用程序可以尝试与 ISP 重新建立通信。

小型示例系统

下面的图 1.1 举例说明了 NetX PPP 是多么易于使用。在此示例中，PPP 包含文件 nx_ppp.h 在第 3 行引入。接下来，在第 56 行的“tx_application_define”中创建 PPP。PPP 控制块“my_ppp”以前在第 9 行定义为全局变量。

> **NOTE**
>
> PPP 应该在创建 IP 实例之前创建。成功创建 PPP 和 IP 后，线程“my_thread”在第 98 行等待 PPP 链接生效。在第 104 行，PPP 和 NetX 可完全正常运行。

应用程序的串行字节接收 ISR 未在此示例中显示。它需要以“my_ppp”和接收到的字节作为输入参数调用 nx_ppp_byte_receive。

```
#include   "tx_api.h"
#include   "nx_api.h"
#include   "nx_ppp.h"

#define     DEMO_STACK_SIZE         4096
TX_THREAD               my_thread;
NX_PACKET_POOL          my_pool;
NX_IP                   my_ip;
NX_PPP                  my_ppp;

/* Define function prototypes. */

void    my_thread_entry(ULONG thread_input);
void    my_serial_driver_byte_output(UCHAR byte);
void    my_invalid_packet_handler(NX_PACKET *packet_ptr);

/* Define main entry point. */
intmain()
```

```
{

    /* 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 “my_thread”. */
    tx_thread_create(&my_thread, "my thread", my_thread_entry, 0,
                  pointer, DEMO_STACK_SIZE,
                  2, 2, TX_NO_TIME_SLICE, TX_AUTO_START);
    pointer =  pointer + DEMO_STACK_SIZE;

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

    /* Create a packet pool. */
    status =  nx_packet_pool_create(&my_pool, "NetX Main Packet Pool",
                                    1024, pointer, 64000);
    pointer = pointer + 64000;

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

}
```

配置选项

有多个配置选项可用于生成 NetX PPP。以下列表详细介绍了每个配置选项：

- NX_DISABLE_ERROR_CHECKING：如果定义此选项，则会删除基本的 PPP 错误检查。通常会在调试应用程序后使用此选项。
- NX_PPP_PPPOE_ENABLE：如果定义此选项，则 PPP 可以通过以太网传输数据包
- NX_PPP_BASE_TIMEOUT：此选项定义唤醒 PPP 线程任务以检查 PPP 事件的周期速率（以计时器时钟周期为单位）。默认值为 1*NX_IP_PERIODIC_RATE（时钟周期数为 100）。
- NX_PPP_DISABLE_INFO：如果定义此选项，则会禁用内部 PPP 信息收集。
- NX_PPP_DEBUG_LOG_ENABLE：如果定义此选项，则会启用内部 PPP 调试日志。
- NX_PPP_DEBUG_LOG_PRINT_ENABLE：如果定义此选项，则会启用内部 PPP 调试日志 printf 至 stdio。此选项仅在调试日志启用时有效。
- NX_PPP_DEBUG_LOG_SIZE：调试日志的大小（调试日志中的条目数）。在到达最后一个条目时，调试捕获会换行到第一个条目，并覆盖先前捕获的所有数据。默认值为 50。
- NX_PPP_DEBUG_FRAME_SIZE：从接收到的数据包有效负载中捕获并保存到调试输出的最大数据量。默认值为 50。
- NX_PPP_DISABLE_CHAP：如果定义此选项，则会删除内部 PPP CHAP 逻辑，包括 MD5 摘要逻辑。
- NX_PPP_DISABLE_PAP：如果定义此选项，则会删除内部 PPP PAP 逻辑。
- NX_PPP_DNS_OPTION_DISABLE：如果定义此选项，则会在 IPCP 响应中禁用 DNS 选项。默认情况下，未定义此选项（DNS 选项已设置）。
- NX_PPP_DNS_ADDRESS_MAX_RETRIES：此选项指定 PPP 主机在 IPCP 状态下向对等方请求 DNS 服务器地址的次数。如果已定义 NX_PPP_DNS_OPTION_DISABLE，则该选项不起作用。默认值为 2。
- NX_PPP_HASHED_VALUE_SIZE：指定 CHAP 身份验证中使用的“哈希值”字符串的大小。默认值设置为 16 字节，但在包含 nx_ppp .h之前可以重新定义该值。
- NX_PPP_MAX_LCP_PROTOCOL_RETRIES：此选项定义 PPP 在发送另一条 LCP 配置请求消息之前超时的最大重试次数。达到此数目后，PPP 握手将中止，并且断开链接状态。默认值为 20。
- NX_PPP_MAX_PAP_PROTOCOL_RETRIES：此选项定义 PPP 在发送另一条 PAP 身份验证请求消息之前超时的最大重试次数。达到此数目后，PPP 握手将中止，并且断开链接状态。默认值为 20。
- NX_PPP_MAX_CHAP_PROTOCOL_RETRIES：此选项定义 PPP 在发送另一条 CHAP 质询消息之前超时的最大重试次数。达到此数目后，PPP 握手将中止，并且断开链接状态。默认值为 20。
- NX_PPP_MAX_IPCP_PROTOCOL_RETRIES：此选项定义 PPP 在发送另一条 IPCP 配置请求消息之前超时的最大重试次数。达到此数目后，PPP 握手将中止，并且断开链接状态。默认值为 20。
- NX_PPP_MRU：指定 PPP 的最大接收单元 (MRU)。默认情况下，此值为 1,500 个字节（最小值）。此定义可以在包含 nx_ppp.h之前由应用程序进行设置。

- NX_PPP_MINIMUM_MRU：指定 LCP 配置请求消息中接收到的最小 MRU。默认情况下，此值为 1,500 个字节（最小值）。此定义可以在包含 nx_ppp.h之前由应用程序进行设置。
- NX_PPP_NAME_SIZE：指定在身份验证中使用的"名称"字符串的大小。默认值设置为 32 字节，但在包含 *nx_ppp.h 之前可以重新定义该值。
- NX_PPP_PASSWORD_SIZE：指定在身份验证中使用的"密码"字符串的大小。默认值设置为 32 字节，但在包含 nx_ppp.h. 之前可以重新定义该值。
- NX_PPP_PROTOCOL_TIMEOUT：此选项定义 PPP 任务接收对 PPP 协议请求消息的响应的等待选项（以秒为单位）。默认值为 4 秒。
- NX_PPP_RECEIVE_TIMEOUTS：此选项定义 PPP 线程任务等待接收 PPP 消息流中下一个字符时的超时次数。此后，PPP 会释放数据包，并开始等待接收下一条 PPP 消息。默认值为 4。
- NX_PPP_SERIAL_BUFFER_SIZE：指定接收字符串行缓冲区的大小。默认情况下，此值为 3,000 字节。此定义可以在包含 nx_ppp.h之前由应用程序进行设置。
- NX_PPP_TIMEOUT：此选项定义分配数据包以传输数据以及将 PPP 串行数据缓冲到要发送至 IP 层的数据包中的等待选项（以计时器时钟周期为单位）。默认值为 4*NX_IP_PERIODIC_RATE（时钟周期数为 400）。
- NX_PPP_THREAD_TIME_SLICE：PPP 线程的时间片选项。默认情况下，此值为 TX_NO_TIME_SLICE。此定义可以在包含 nx_ppp.h之前由应用程序进行设置。
- NX_PPP_VALUE_SIZE：指定 CHAP 身份验证中使用的"值"字符串的大小。默认值设置为 32 字节，但在包含 nx_ppp.h. 之前可以重新定义该值。

第 3 章 - Azure RTOS NetX 点对点协议 (PPP) 服务说明

2021/4/29 •

本章按字母顺序提供了所有 Azure RTOS NetX PPP 服务（如下所列）的说明。

在以下 API 说明中的"返回值"部分，以粗体显示的值不受用于禁用 API 错误检查的 NX_DISABLE_ERROR_CHECKING 定义影响，而不以粗体显示的值则已完全禁用 。

- nx_ppp_byte_receive：从串行 ISR 接收字节
- nx_ppp_chap_challenge：生成 CHAP 质询
- nx_ppp_chap_enable：启用 CHAP 身份验证
- nx_ppp_create：创建 PPP 实例
- nx_ppp_delete：删除 PPP 实例
- nx_ppp_dns_address_get：获取 DNS IP 地址
- nx_ppp_dns_address_set：设置 DNS 服务器 IP 地址
- nx_ppp_secondary_dns_address_get：获取次要 DNS 服务器 IP 地址
- nx_ppp_secondary_dns_address_set：设置次要 DNS 服务器 IP 地址
- nx_ppp_interface_index_get：获取 IP 接口索引
- nx_ppp_ip_address_assign：为 IPCP 分配 IP 地址
- nx_ppp_link_down_notify：在链接断开时通知应用程序
- nx_ppp_link_up_notify：在链接恢复时通知应用程序
- nx_ppp_nak_authentication_notify：收到身份验证 NAK 时通知应用程序
- nx_ppp_pap_enable：启用 PAP 身份验证
- nx_ppp_ping_request：发送 LCP 回显请求
- nx_ppp_raw_string_send：发送非 PPP 字符串
- nx_ppp_restart：重启 PPP 处理
- nx_ppp_start：启动 PPP 处理
- nx_ppp_status_get：获取当前 PPP 状态
- nx_ppp_stop：停止 PPP 处理
- nx_ppp_packet_receive：接收 PPP 数据包
- nx_ppp_packet_send_set：设置 PPP 数据包发送函数

nx_ppp_byte_receive

从串行 ISR 接收字节

原型

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

说明

通常，从应用程序的串行驱动程序中断服务例程 (ISR) 调用此服务，以将接收的字节传输到 PPP。调用时，此例程将接收的字节置于循环字节缓冲区中，并通知相应的 PPP 线程进行处理。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- byte：从串行设备接收到的字节

返回值

- NX_SUCCESS：(0x00) 已成功接收 PPP 字节。
- NX_PPP_BUFFER_FULL：(0xB1) PPP 串行缓冲区已满。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

允许来自

线程、ISR

示例

```
/* 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

生成 CHAP 质询

原型

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

说明

此服务用于在已启动并运行 PPP 连接后启动 CHAP 质询。这使得应用程序能够定期验证连接的真实性。如果质询失败，则将关闭 PPP 链接。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。

返回值

- NX_SUCCESS：(0x00) 已成功启动 PPP 质询。
- NX_PPP_FAILURE：(0xB0) PPP 质询无效，仅为响应启用了 CHAP。
- NX_NOT_IMPLEMENTED：(0x80) 通过 NX_PPP_DISABLE_CHAP 禁用了 CHAP 逻辑。
- NX_PTR_ERROR：(0x07) PPP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

线程数

示例

```
/* 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

启用 CHAP 身份验证

原型

```
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));
```

说明

此服务用于为指定的 PPP 实例启用质询握手身份验证协议 (CHAP)。

如果指定了"get_challenge_values"*和"get_verification_values"*函数指针，则此 PPP 实例需要使用 CHAP。否则，CHAP 仅响应对等方的质询请求。

在所需的回调函数中，引用如下数据项。数据项 secret、name 和 system 应为以 NULL 结尾的字符串，其最大大小为 NX_PPP_NAME_SIZE-1。数据项 rand_value 应为以 NULL 结尾的字符串，其最大大小为 NX_PPP_VALUE_SIZE-1。数据项 id 是简单的无符号字符类型。

> **NOTE**
> 必须在 nx_ppp_create 之后且在 nx_ip_create 或 nx_ip_interface_attach 之前调用此函数。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- get_challenge_values：指向应用程序函数的指针，以检索用于质询的值。请注意，必须将 rand_value、id 和 secret 值复制到所提供的目标中。
- get_responder_values：指向应用程序函数的指针，该函数检索用于响应质询的值。请注意，必须将 system、name 和 secret 值复制到所提供的目标中。
- get_verification_values：指向应用程序函数的指针，该函数检索用于验证质询响应的值。请注意，必须将 system、name 和 secret 值复制到所提供的目标中。

返回值

- NX_SUCCESS：(0x00) 已成功启用 PPP CHAP
- NX_NOT_IMPLEMENTED：(0x80) 通过 NX_PPP_DISABLE_CHAP 禁用了 CHAP 逻辑。
- NX_PTR_ERROR：(0x07) PPP 指针或回调函数指针无效。请注意，如果指定了 get_challenge_values，则还必须提供 get_verification_values 函数。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

初始化、线程

示例

```
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. */
```

```
/* 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);
```

```
    return(NX_SUCCESS);
}
```

nx_ppp_create

创建 PPP 实例

原型

```
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));
```

说明

此服务用于为指定的 NetX IP 实例创建 PPP 实例。

> **NOTE**
>
> 通常，创建 NetX IP 线程的优先级最好高于创建 PPP 线程的优先级。有关指定 IP 线程优先级的详细信息，请参阅“nx_ip_create”服务。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- name：此 PPP 实例的名称。
- ip_ptr：指向尚未创建的 IP 实例的控制块的指针。
- stack_memory_ptr：指向 PPP 线程堆栈区域起始位置的指针。
- stack_size：线程堆栈的大小（以字节为单位）。
- pool_ptr：指向默认数据包池的指针。
- thread_priority：内部 PPP 线程的优先级 (1-31)。
- ppp_invalid_packet_handler：指向所有非 PPP 数据包的应用程序处理程序的函数指针。NetX PPP 通常会在初始化期间调用此例程。在这种情况下，应用程序可以响应调制解调器命令，或者对于 Windows XP，NetX PPP 应用程序可以通过使用“客户端服务器”响应 Windows XP 发送的初始“客户端”来启动 PPP。
- ppp_byte_send：指向应用程序串行字节输出例程的函数指针。

返回值

- NX_SUCCESS：(0x00) 已成功创建 PPP。
- NX_PTR_ERROR：(0x07) PPP、IP 或字节输出函数指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

初始化、线程

示例

```
/* 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

删除 PPP 实例

原型

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

说明

此服务用于删除先前创建的 PPP 实例。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。

返回值

- NX_SUCCESS：(0x00) 已成功删除 PPP。
- NX_PTR_ERROR：(0x07) PPP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

线程数

示例

```
/* 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

获取 DNS IP 地址

原型

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

说明

此服务用于检索对等方提供的 DNS IP 地址。如果对等方未提供 IP 地址，则返回的 IP 地址为 0。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- dns_address_ptr：DNS IP 地址的目标

返回值

- NX_SUCCESS：(0x00) 已成功获取 PPP 地址。
- NX_PPP_NOT_ESTABLISHED：(0xB5) PPP 尚未完成与对等方的协商。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

允许来自

初始化、线程、计时器、ISR

示例

```
ULONG  my_dns_address;

/* Get IP Server 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_secondary_dns_address_get

获取次要 DNS 服务器 IP 地址

原型

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

说明

此服务用于检索 IPCP 握手中对等方提供的次要 DNS IP 地址。如果对等方未提供 IP 地址，则返回的 IP 地址为 0。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- dns_address_ptr：次要 DNS 服务器地址的目标

返回值

- NX_SUCCESS：(0x00) 已成功获取 DNS 地址。
- NX_PPP_NOT_ESTABLISHED：(0xB5) PPP 尚未完成与对等方的协商。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

允许来自

初始化、线程、计时器、ISR

示例

```
ULONG  my_dns_address;

/* Get secondary DNS Server address supplied by peer. */
status =  nx_ppp_secondary_dns_address_get(&my_ppp, &my_dns_address);

/* If status is NX_SUCCESS the “my_dns_address” contains the secondary DNS Server address – if the peer
supplied one. */
```

nx_ppp_dns_address_set

设置主 DNS 服务器 IP 地址

原型

```
UINT nx_ppp_dns_address_set(NX_PPP *ppp_ptr, ULONG dns_address);
```

说明

此服务用于设置 DNS 服务器 IP 地址。如果对等方在 IPCP 状态下发送 DNS 服务器选项请求，则此主机将提供这些信息。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- dns_address：DNS 服务器地址

返回值

- NX_SUCCESS：(0x00) 已成功设置 DNS 地址。
- NX_PPP_NOT_ESTABLISHED：(0xB5) PPP 尚未完成与对等方的协商。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

允许来自

初始化、线程

示例

```
ULONG  my_dns_address = IP_ADDRESS(1,2,3,1);

/* Set DNS Server address. */
status =  nx_ppp_dns_address_set(&my_ppp, my_dns_address);

/* If status is NX_SUCCESS the “my_dns_address” will be the DNS Server address provided if the peer requests
one. */
```

nx_ppp_secondary_dns_address_set

设置次要 DNS 服务器 IP 地址

原型

```
UINT nx_ppp_secondary_dns_address_set(NX_PPP *ppp_ptr, ULONG dns_address);
```

说明

此服务用于设置次要 DNS 服务器 IP 地址。如果对等方在 IPCP 状态下发送次要 DNS 服务器选项请求，则此主机将提供这些信息。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- dns_address：次要 DNS 服务器地址

返回值

- NX_SUCCESS：(0x00) 已成功设置 DNS 地址。
- NX_PPP_NOT_ESTABLISHED：(0xB5) PPP 尚未完成与对等方的协商。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

允许来自

初始化、线程

示例

```
ULONG  my_dns_address = IP_ADDRESS(1,2,3,1);

/* Set DNS Server address. */
status =  nx_ppp_secondary_dns_address_set(&my_ppp, my_dns_address);

/* If status is NX_SUCCESS the “my_dns_address” will be the secondary DNS Server address provided if the
peer requests one. */
```

nx_ppp_interface_index_get

获取 IP 接口索引

原型

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

说明

此服务用于检索与该 PPP 实例关联的 IP 接口索引。仅当 PPP 实例不是 IP 实例的主接口时，此服务才有用。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- index_ptr：接口索引的目标

返回值

- NX_SUCCESS：(0x00) 已成功获取 PPP 索引。
- NX_IN_PROGRESS：(0x37) PPP 尚未完成初始化。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

允许来自

初始化、线程、计时器、ISR

示例

```
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

为 IPCP 分配 IP 地址

原型

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

说明

此服务用于设置本地和对等方 IP 地址，以用于 Internet 协议控制协议 (IPCP)。PPP 应用程序应使用自身和另一对等方的有效 IP 地址在 PPP 实例上调用此服务。如果未在 PPP 实例中注册有效地址，则该应用程序必须依赖

PPP 对等方来为其定义 IP 地址。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- local_ip_address：本地 IP 地址。
- peer_ip_address：对等方 IP 地址。

返回值

- NX_SUCCESS：(0x00) 已成功分配 PPP 地址。
- NX_PTR_ERROR：(0x07) PPP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

初始化、线程

示例

```
/* 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

链接断开时通知应用程序

原型

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

说明

此服务使用指定 PPP 实例，注册应用程序的链接断开通知回调。如果为非 NULL，则只要链接断开，即会调用应用程序的链接断开回调函数。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- link_down_callback：应用程序的链接断开通知函数指针。如果为 NULL，则禁用链接断开通知。

返回值

- NX_SUCCESS：(0x00) 已成功注册链接断开通知回调。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

允许来自

初始化、线程、计时器、ISR

示例

```
/* 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

链接恢复时通知应用程序

原型

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

说明

此服务使用指定 PPP 实例，注册应用程序的链接恢复通知回调。如果为非 NULL，则只要出现链接，即会调用应用程序的链接恢复回调函数。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- link_up_callback：应用程序的链接恢复通知函数指针。如果为 NULL，则禁用链接恢复通知。**

返回值

- NX_SUCCESS：(0x00) 已成功注册链接恢复通知回调。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

允许来自

初始化、线程、计时器、ISR

示例

```
/* 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

收到身份验证 NAK 时通知应用程序

原型

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

说明

此服务使用指定 PPP 实例，注册应用程序的身份验证 NAK 通知回调。如果为非 NULL，则只要 PPP 实例在身份验证期间收到 NAK，即会调用此回调函数。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- nak_authentication_notify：PPP 实例接收身份验证 NAK 时所调用函数的指针。如果为 NULL，则禁用此通知。

返回值

- NX_SUCCESS：(0x00) 已成功注册通知回调。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

允许来自

初始化、线程、计时器、ISR

示例

```
/* 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

启用 PAP 身份验证

原型

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

说明

此服务为指定的 PPP 实例启用密码身份验证协议 (PAP)。如果指定了“verify_login”函数指针，则此 PPP 实例需要使用 PAP。否则，PAP 仅响应在 LCP 协商期间指定的对等方的 PAP 要求。

在所需的回调函数中，引用如下数据项。数据项 name 应为以 NULL 结尾的字符串，其最大大小为 NX_PPP_NAME_SIZE-1。数据项 password 也应为以 NULL 结尾的字符串，其最大大小为 NX_PPP_PASSWORD_SIZE-1。

NOTE

必须在 nx_ppp_create 之后且在 nx_ip_create 或 nx_ip_interface_attach 之前调用此函数。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- generate_login：指向应用程序函数的指针，该函数将生成供对等方进行身份验证的 name 和 password 。请注意，必须将 name 和 password 值复制到所提供的目标中。
- verify_login：指向应用程序函数的指针，该函数会验证对等方提供的 name 和 password 。此例程必须将提供的 name 和 password 进行比较。如果此例程返回 NX_SUCCESS，则 name 和 password 正确，且 PPP 可以继续下一步。否则，此例程返回 NX_PPP_ERROR，PPP 只能等待其他 name 和 password。

返回值

- NX_SUCCESS：(0x00) 已成功启用 PPP PAP。
- NX_NOT_IMPLEMENTED：(0x80) 通过 NX_PPP_DISABLE_PAP 禁用 PAP 逻辑。
- NX_PTR_ERROR：(0x07) PPP 指针或应用程序函数指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

初始化、线程

示例

```
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

发送 LCP ping 请求

原型

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

说明

此服务用于发送 LCP ping 请求并设置表明 PPP 设备正在等待回显响应的标志。发送请求后就会立即返回此服务。无需等待响应。

接收到匹配的回显响应时，PPP 线程任务将清除该标志。PPP 设备必须已完成 PPP 协商的 LCP 部分。

此服务对于 PPP 设置非常有用，在 PPP 设置中，可能无法轻松轮询硬件以获取链接状态。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- data：指向要在回显请求中发送的数据的指针。
- data_size：要发送的数据的大小。wait_option：等待发送 LCP 回显消息的时间。

返回值

- NX_SUCCESS：(0x00) 已成功发送回显请求。
- NX_PPP_NOT_ESTABLISHED：(0xB5) 未建立 PPP 连接。
- NX_PTR_ERROR：(0x07) PPP 指针或应用程序函数指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

应用程序线程

示例

```
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

发送原始 ASCII 字符串

原型

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

说明

此服务直接从 PPP 接口发送非 PPP ASCII 字符串。通常在 PPP 收到包含调制解调器控制信息的非 PPP 数据包后使用此服务。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- string_ptr：指向要发送的字符串的指针。

返回值

- NX_SUCCESS：(0x00) 已成功发送 PPP 原始字符串。
- NX_PTR_ERROR：(0x07) PPP 指针或字符串指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

线程数

示例

```
/* 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

重启 PPP 处理

原型

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

说明

此服务用于重启 PPP 处理。通常，当需要从链接断开回调或通过指示通信丢失的非 PPP 调制解调器消息重新建立链接时，会调用此服务。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。

返回值

- NX_SUCCESS：(0x00) 已成功发起 PPP 重启。
- NX_PTR_ERROR：(0x07) PPP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

线程数

示例

```
/* 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

启动 PPP 处理

原型

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

说明

此服务用于启动 PPP 处理。通常在调用 nx_ppp_stop() 之后调用此服务。

> **NOTE**
> 启用链接后，PPP 会自动启动 PPP 处理。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。

返回值

- NX_SUCCESS：(0x00) 已成功发起 PPP 启动。
- NX_PPP_ALREADY_STARTED：(0xb9) 已启动 PPP。
- NX_PTR_ERROR：(0x07) PPP 指针无效。
- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

线程数

示例

```
/* 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

获取当前 PPP 状态

原型

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

说明

此服务用于获取指定 PPP 实例的当前状态。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。

- status_ptr：PPP 状态的目标，其可能的状态值如下：
  - 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**
> 此状态仅在 API 返回 NX_SUCCESS 时有效。此外，如果返回任何 *_FAILED 状态值，则会有效地停止 PPP 处理，直到应用程序再次重启 PPP 处理。

返回值

- NX_SUCCESS：(0x00) 已成功请求 PPP 状态。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

允许来自

初始化、线程、计时器、ISR

示例

```
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

启动 PPP 处理

原型

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

说明

此服务用于停止 PPP 处理。如果需要，用户还可以调用 nx_ppp_start() 以启动 PPP 处理。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。

返回值

- NX_SUCCESS：(0x00) 已成功发起 PPP 启动。
- NX_PPP_ALREADY_STOPPED：(0xb8) PPP 已停止。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

- NX_CALLER_ERROR：(0x11) 此服务的调用方无效。

允许来自

线程数

示例

```
/* Stop the PPP instance “my_ppp”.  */
status =  nx_ppp_stop(&my_ppp);

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

nx_ppp_packet_receive

接收 PPP 数据包

原型

```
UINT  nx_ppp_packet_receive(NX_PPP *ppp_ptr, NX_PACKET *packet_ptr);
```

说明

此服务用于接收 PPP 数据包。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- packet_ptr：指向 PPP 数据包的指针。

返回值

- NX_SUCCESS：(0x00) 已成功请求 PPP 状态。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

允许来自

初始化、线程

示例

```
/* Receive the PPP packet of PPP instance “my_ppp”.  */
status =  nx_ppp_packet_receive(&my_ppp, packet_ptr);

/* If status is NX_SUCCESS the PPP packet has received. */
```

nx_ppp_packet_send_set

设置 PPP 数据包发送函数

原型

```
UINT  nx_ppp_packet_send_set(NX_PPP *ppp_ptr,
                             VOID (*nx_ppp_packet_send)(NX_PACKET *packet_ptr));
```

说明

此服务用于设置 PPP 数据包发送函数。

输入参数

- ppp_ptr：指向 PPP 控制块的指针。
- nx_ppp_packet_send：发送 PPP 数据包的例程。

返回值

- NX_SUCCESS：(0x00) 已成功请求 PPP 状态。
- NX_PTR_ERROR：(0x07) PPP 指针无效。

允许来自

初始化、线程

示例

```
/* Set the PPP packet send function of PPP instance “my_ppp”.  */
status =  nx_ppp_packet_send_set(&my_ppp, nx_ppp_packet_send);

/* If status is NX_SUCCESS the PPP packet send function has set. */
```

第 1 章 - Azure RTOS NetX PPPoE 客户端简介

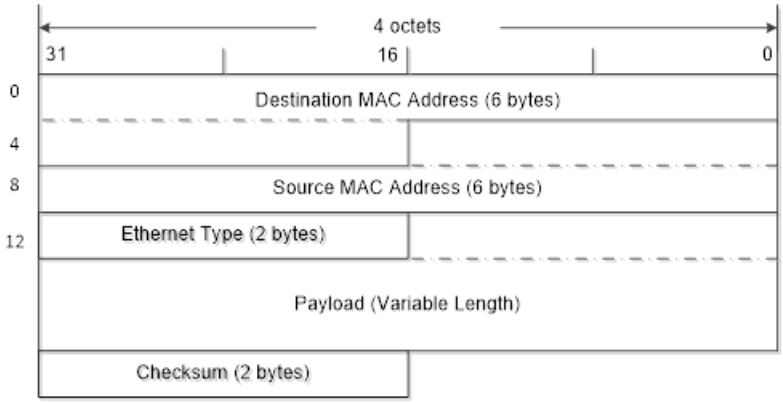
2021/4/29 •

以太网 PPP (PPPoE) 允许主机通过以太网（而非传统的基于字符的串行线路通信）连接到 PPP 服务器。“RFC 2516：一种传输以太网 PPP (PPPoE) 的方法”中介绍了 PPPoE 的技术详细信息。本文档重点介绍 Azure RTOS NetX PPPoE 模块的详细信息。

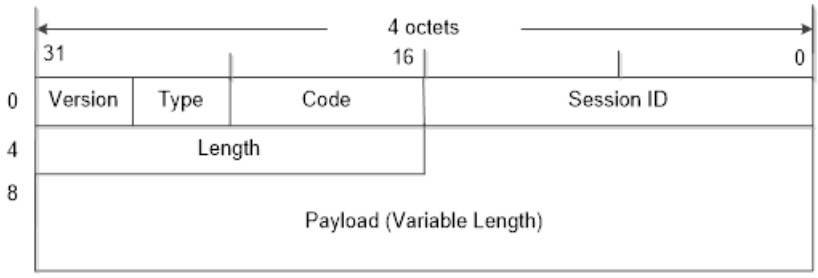
若要通过以太网提供点到点连接，每个 PPP 会话必须了解远程对等方的以太网地址，并建立唯一的会话标识符。

根据 RFC 2516，PPPoE 包含两个阶段：“发现”阶段和“PPPoE 会话”阶段。当主机（客户端）希望启动 PPP 会话时，它必须首先执行发现步骤来查找 PPPoE 服务器。此步骤还允许服务器和客户端识别彼此的以太网 MAC 地址和 SESSION_ID，它们将用于 PPP 会话的其余部分。

以太网帧如下所示：



PPPoE 的以太网有效负载如下所示：



PPPoE 发现阶段

PPPoE 发现阶段允许客户端从网络上的所有可用服务器中选择一台服务器，并高效地在交换 PPP 帧之前创建会话。在发现阶段结束时，客户端和服务器都应同意使用唯一的会话 ID，同时双方都需要知道对等方的 MAC 地址。

PPPoE 主动发现开始 (PADI)	0x09	从客户端到广播
PPPoE 主动发现要约 (PADO)	0x07	从服务器到客户端
PPPoE 主动发现请求 (PADR)	0x19	从客户端到服务器

[illegible]	[illegible]	[illegible]
PPPOE 主动发现会话-确认 (PADS)	0x65	从服务器到客户端
PPPoE 主动发现终止 (PADT)	0xa7	可从服务器或客户端启动

所有发现以太网帧都将 ETHER_TYPE 字段设置为值 0x8863。

PPPoE 会话消息

在客户端和服务器创建会话后，可以将 PPP 帧作为 PPPoE 会话消息进行传输。在会话期间，SESSION_ID 不能更改，并且必须是在发现阶段由服务器分配的值。

所有 PPPoE 会话以太网帧都将 ETHER_TYPE 字段设置为值 0x8864。

第 2 章 - Azure RTOS NetX PPPoE 客户端的安装和使用

2021/4/29 •

本章包含与安装、设置和使用 Azure RTOS NetX PPPoE 客户端组件相关的各种问题的说明。

产品分发

适用于 NetX 的 PPPoE 客户端可从 https://github.com/azure-rtos/netx 获得。该包中包含以下文件：

- nx_pppoe_client.h：适用于 NetX 的 PPPoE 客户端的头文件
- nx_pppoe_client.c：适用于 NetX 的 PPPoE 客户端的 C 源文件
- nx_pppoe_client.pdf：适用于 NetX 的 PPPoE 客户端的 PDF 说明
- demo_netx_pppoe_client.c：NetX PPPoE 客户端演示

PPPoE 客户端安装

要使用适用于 NetX 的 PPPoE 客户端，应将前面提到的整个分发包复制到 NetX 所安装在的目录中。例如，如果 NetX 安装在“\threadx\arm7\green”目录中，则应将 nx_pppoe_client.h 和 nx_pppoe_client.c 文件复制到该目录中。

使用 PPPoE 客户端

使用适用于 NetX 的 PPPoE 客户端非常简单。总体上，应用程序代码必须先包含 tx_api.h 和 nx_api.h，然后包含 nx_pppoe_client.h，才能分别使用 ThreadX 和 NetX。包含 nx_pppoe_client.h 之后，应用程序代码就可以进行本指南随后部分所述的 PPPoE 客户端函数调用。应用程序还必须在生成过程中包含 nx_pppoe_client.c。此文件必须采用与其他应用程序文件相同的方式来编译，并且其对象窗体必须与该应用程序的文件一起链接。这就是使用 NetX PPPoE 客户端所需的一切。

小型示例系统

下面的示例演示如何使用 NetX PPPoE 客户端，如图 1.1 所述。在此示例中，第 50 行引入 PPPoE 客户端包含文件 nx_pppoe_client.h。接下来，在“thread_0_entry”中的第 238 行创建 PPPoE 客户端。请注意，应该先创建 IP 实例，然后再创建 PPPoE 客户端。在第 142-220 行，创建并初始化 IP 实例和 PPP 实例。在之前的第 75 行，已将 PPPoE 客户端控制块“pppoe_client”定义为全局变量。send 和 receive 函数在第 238 行进行设置。

通常，PPPoE 模块应与 PPP 模块配合使用。在此示例中，第 49 行引入 PPP 客户端包含文件 nx_ppp.h。接下来，在第 164 行创建 PPP 客户端。第 172 行设置用于发送 PPP 数据包的函数。第 179-190 行设置 IP 地址，并定义 pap 协议。第 104-129 行设置 pap 协议的用户名和密码。

PPPoE 会话建立之后，在第 264 行，应用程序可以调用 nx_pppoe_client_session_get，获取会话信息（服务器 MAC 地址和会话 id）。在第 283 行，PPP 或应用程序可以调用 nx_pppoe_client_session_packet_send，发送 PPPoE 数据包。

当应用程序不再处理 PPP 流量时，应用程序可以调用 nx_pppoe_client_session_terminate 终止 PPPoE 会话。

请注意，在此示例中，PPPoE 客户端与普通 IP 堆栈同时运行，并共享一个以太网驱动程序。在第 155 行，传递同一以太网驱动程序以创建普通 IP 实例，并在第 298 行传递该驱动程序以创建 PPPoE 客户端实例。

NOTE

将 NX_PHYSICAL_HEADER 重新定义为 24，确保有足够的空间可用于填充物理标头。物理标头：14（以太网标头）+ 6（PPPoE 标头）+ 2（PPP 标头）+ 2（四字节对齐）。

```
/**************************************************************************/
/**************************************************************************/
/**                                                                       */
/** NetX PPPoE Client stack Component                                     */
/**                                                                       */
/**   This is a small demo of the high-performance NetX PPPoE Client      */
/**   stack. This demo includes IP instance, PPPoE Client and PPP Client  */
/**   stack. Create one IP instance includes two interfaces to support    */
/**   for normal IP stack and PPPoE Client, PPPoE Client can use the      */
/**   mutex of IP instance to send PPPoE message when share one Ethernet  */
/**   driver. PPPoE Client work with normal IP instance at the same time. */
/**                                                                       */
/**   Note1: Substitute your Ethernet driver instead of                   */
/**   _nx_ram_network_driver before run this demo                         */
/**                                                                       */
/**   Note2: Prerequisite for using PPPoE.                                */
/**   Redefine NX_PHYSICAL_HEADER to 24 to ensure enough space for filling*/
/**   in physical header. Physical header:14(Ethernet header)             */
/**    + 6(PPPoE header) + 2(PPP header) + 2(four-byte aligment)          */
/**                                                                       */
/**************************************************************************/
/**************************************************************************/


      /*********************************************************************/
      /*                          NetX Stack                               */
      /*********************************************************************/

                                        /*************************/
                                        /*        PPP Client     */
                                        /*************************/

                                        /*************************/
                                        /*       PPPoE Client    */
                                        /*************************/
      /*************************/       /*************************/
      /*    Normal Ethernet Type */       /*    PPPoE Ethernet Type  */
      /*************************/       /*************************/
      /*************************/       /*************************/
      /*       Interface 0     */       /*       Interface 1     */
      /*************************/       /*************************/

      /*********************************************************************/
      /*                       Ethernet Dirver                             */
      /*********************************************************************/

#include   "tx_api.h"
#include   "nx_api.h"
#include   "nx_ppp.h"
#include   "nx_pppoe_client.h"

/* Defined NX_PPP_PPPOE_ENABLE if use Express Logic's PPP, since PPP module has been modified
    to match PPPoE moduler under this definition.  */
#ifdef NX_PPP_PPPOE_ENABLE

/* If the driver is not initialized in other module, define NX_PPPOE_CLIENT_INITIALIZE_DRIVER_ENABLE
    to initialize the driver in PPPoE module .
   In this demo, the driver has been initialized in IP module.  */
#ifndef NX_PPPOE_CLIENT_INITIALIZE_DRIVER_ENABLE

/* Define the block size.  */
```

```
59 /* Define the block size.  */
60 #define     NX_PACKET_POOL_SIZE     ((1536 + sizeof(NX_PACKET)) * 30)
61 #define     DEMO_STACK_SIZE         2048
62 #define     PPPOE_THREAD_SIZE       2048
63
64 /* Define the ThreadX and NetX object control blocks...  */
65 TX_THREAD               thread_0;
66
67 /* Define the packet pool and IP instance for normal IP instnace.  */
68 NX_PACKET_POOL          pool_0;
69 NX_IP                   ip_0;
70
71 /* Define the PPP Client instance.  */
72 NX_PPP                  ppp_client;
73
74 /* Define the PPPoE Client instance.  */
75 NX_PPPOE_CLIENT         pppoe_client;
76
77 /* Define the counters.  */
78 CHAR                    *pointer;
79 ULONG                   error_counter;
80
81 /* Define thread prototypes.  */
82 void    thread_0_entry(ULONG thread_input);
83
84 /***** Substitute your PPP driver entry function here *********/
85 extern void    _nx_ppp_driver(NX_IP_DRIVER *driver_req_ptr);
86
87 /***** Substitute your Ethernet driver entry function here *********/
88 extern void    _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr);
89
90 /* Define the porting layer function for Express Logic's PPP to simulate TTP's PPP.
91    Functions to be provided by PPP for calling by the PPPoE Stack.  */
92 void    ppp_client_packet_send(NX_PACKET *packet_ptr);
93 void    pppoe_client_packet_receive(NX_PACKET *packet_ptr);
94
95 /* Define main entry point.  */
96
97 int main()
98 {
99
100     /* Enter the ThreadX kernel.  */
101     tx_kernel_enter();
102 }
103
104 UINT generate_login(CHAR *name, CHAR *password)
105 {
106
107     /* Make a name and password, called "myname" and "mypassword".  */
108     name[0] = 'm';
109     name[1] = 'y';
110     name[2] = 'n';
111     name[3] = 'a';
112     name[4] = 'm';
113     name[5] = 'e';
114     name[6] = (CHAR) 0;
115
116     password[0] = 'm';
117     password[1] = 'y';
118     password[2] = 'p';
119     password[3] = 'a';
120     password[4] = 's';
121     password[5] = 's';
122     password[6] = 'w';
123     password[7] = 'o';
124     password[8] = 'r';
125     password[9] = 'd';
126     password[10] = (CHAR) 0;
127
128     return(NX_SUCCESS);
```

```
128     return(NX_SUCCESS);
129 }
130
131 /* Define what the initial system looks like.  */
132
133 void    tx_application_define(void *first_unused_memory)
134 {
135
136 UINT    status;
137
138     /* Setup the working pointer.  */
139     pointer =  (CHAR *) first_unused_memory;
140
141     /* Initialize the NetX system.  */
142     nx_system_initialize();
143
144     /* Create a packet pool for normal IP instance.  */
145     status = nx_packet_pool_create(&pool_0, "NetX Main Packet Pool",
146                                    (1536 + sizeof(NX_PACKET)),
147                                    pointer, NX_PACKET_POOL_SIZE);
148     pointer = pointer + NX_PACKET_POOL_SIZE;
149
150     /* Check for error.  */
151     if (status)
152         error_counter++;
153
154     /* Create an normal IP instance.  */
155     status = nx_ip_create(&ip_0, "NetX IP Instance", IP_ADDRESS(192, 168, 100, 44), 0xFFFFFF00UL,
156                           &pool_0, _nx_ram_network_driver, pointer, 2048, 1);
157     pointer = pointer + 2048;
158
159     /* Check for error.  */
160     if (status)
161         error_counter++;
162
163     /* Create the PPP instance.  */
164     status = nx_ppp_create(&ppp_client, "PPP Instance", &ip_0, pointer, 2048, 1,
165                            &pool_0, NX_NULL, NX_NULL); pointer = pointer + 2048;
166
167     /* Check for PPP create error.   */
168     if (status)
169         error_counter++;
170
171     /* Set the PPP packet send function.  */
172     status = nx_ppp_packet_send_set(&ppp_client, ppp_client_packet_send);
173
174     /* Check for PPP packet send function set error.   */
175     if (status)
176         error_counter++;
177
178     /* Define IP address. This PPP instance is effectively the client since it doesn't have
           any IP addresses. */
179     status = nx_ppp_ip_address_assign(&ppp_client, IP_ADDRESS(0, 0, 0, 0), IP_ADDRESS(0, 0, 0, 0));
180
181     /* Check for PPP IP address assign error.   */
182     if (status)
183         error_counter++;
184
185     /* Setup PAP, this PPP instance is effectively the since it generates the name and password
           for the peer..  */
186     status = nx_ppp_pap_enable(&ppp_client, generate_login, NX_NULL);
187
188     /* Check for PPP PAP enable error.  */
189     if (status)
190         error_counter++;
191
192     /* Attach an interface for PPP.  */
193     status = nx_ip_interface_attach(&ip_0, "Second Interface For PPP", IP_ADDRESS(0, 0, 0, 0), 0,
                                        nx_ppp_driver);
```

```
194
195     /* Check for error.  */
196     if (status)
197         error_counter++;
198
199     /* Enable ARP and supply ARP cache memory for Normal IP Instance.  */
200     status = nx_arp_enable(&ip_0, (void *) pointer, 1024);
201     pointer = pointer + 1024;
202
203     /* Check for ARP enable errors.  */
204     if (status)
205         error_counter++;
206
207     /* Enable ICMP */
208     status = nx_icmp_enable(&ip_0);
209     if(status)
210         error_counter++;
211
212     /* Enable UDP traffic.  */
213     status =  nx_udp_enable(&ip_0);
214     if (status)
215         error_counter++;
216
217     /* Enable TCP traffic.  */
218     status =  nx_tcp_enable(&ip_0);
219     if (status)
220         error_counter++;
221
222     /* Create the main thread.  */
223     tx_thread_create(&thread_0, "thread 0", thread_0_entry, 0,
224                      pointer, DEMO_STACK_SIZE,
225                      4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);
226     pointer =  pointer + DEMO_STACK_SIZE;
227
228 }
229
230 /* Define the test threads.  */
231
232 void    thread_0_entry(ULONG thread_input)
233 {
234 UINT    status;
235 ULONG   ip_status;
236
237     /* Create the PPPoE instance.  */
238     status =  nx_pppoe_client_create(&pppoe_client, (UCHAR *)"PPPoE Client",  &ip_0,  0,
        &pool_0, pointer, PPPOE_THREAD_SIZE, 4, _nx_ram_network_driver, pppoe_client_packet_receive);
239     pointer = pointer + PPPOE_THREAD_SIZE;
240     if (status)
241     {
242         error_counter++;
243         return;
244     }
245
246     /* Establish PPPoE Client sessione.  */
247     status = nx_pppoe_client_session_connect(&pppoe_client, NX_WAIT_FOREVER);
248     if (status)
249     {
250         error_counter++;
251         return;
252     }
253
254     /* Wait for the link to come up.  */
255     status = nx_ip_interface_status_check(&ip_0, 1, NX_IP_ADDRESS_RESOLVED,
                                              &ip_status, NX_WAIT_FOREVER);
256     if (status)
257     {
258         error_counter++;
259         return;
260     }
```

```
261
262     /* Get the PPPoE Server physical address and Session ID after establish PPPoE Session.  */
263     /*
264     status = nx_pppoe_client_session_get(&pppoe_client, &server_mac_msw,
                                             &server_mac_lsw, &session_id);
265     if (status)
266         error_counter++;
267     */
268 }
269
270 /* PPPoE Client receive function.  */
271 void    pppoe_client_packet_receive(NX_PACKET *packet_ptr)
272 {
273
274     /* Call PPP Client to receive the PPP data fame.  */
275     nx_ppp_packet_receive(&ppp_client, packet_ptr);
276 }
277
278 /* PPP Client send function.  */
279 void    ppp_client_packet_send(NX_PACKET *packet_ptr)
280 {
281
282     /* Directly Call PPPoE send function to send out the data through PPPoE module.  */
283     nx_pppoe_client_session_packet_send(&pppoe_client, packet_ptr);
284 }
285 #endif /* NX_PPPOE_CLIENT_INITIALIZE_DRIVER_ENABLE  */
286
287 #endif /* NX_PPP_PPPOE_ENABLE  */
```

图 1.1 与 NetX 配合使用的 PPPoE 客户端示例

配置选项

可通过几个配置选项生成构建适用于 NetX 的 PPPoE 客户端。以下列表对每个配置选项进行详细说明：

- NX_DISABLE_ERROR_CHECKING：定义后，此选项会删除基本的 PPPoE 客户端错误检查，通常在调试应用程序完成后使用。
- NX_PPPOE_CLIENT_INITIALIZE_DRIVER_ENABLE：定义后，此选项会启用在 PPPoE 模块中初始化以太网驱动程序的功能，默认情况下则会禁用该功能。
- NX_PPPOE_CLIENT_THREAD_TIME_SLICE：PPPoE 客户端线程的时间片选项。默认情况下，此值为 TX_NO_TIME_SLICE。
- NX_PPPOE_CLIENT_PADI_INIT_TIMEOUT：此选项定义初始重新传输 PADI 数据包的等待部分。默认情况下，此值为 1 秒。
- NX_PPPOE_CLIENT_PADI_COUNT：此选项定义在将连接视为中断之前允许的 PADI 传输停用次数。默认情况下，此值为 4。
- NX_PPPOE_CLIENT_PADR_INIT_TIMEOUT：此选项定义初始重新传输 PADR 数据包的等待部分。默认情况下，此值为 1 秒。
- NX_PPPOE_CLIENT_PADR_COUNT：此选项定义在将连接视为中断之前允许的 PADR 传输停用次数。默认情况下，此值为 4。
- NX_PPPOE_CLIENT_MAX_AC_NAME_SIZE：此选项定义 AC-Name 的最大大小。默认情况下，此值为 32。
- NX_PPPOE_CLIENT_MAX_AC_COOKIE_SIZE：此选项定义 AC-Cookie 的最大大小。默认情况下，此值为 32。
- NX_PPPOE_CLIENT_MAX_RELAY_SESSION_ID_SIZE：此选项定义 Relay-Session-Id 的最大大小。默认情况下，此值为 12。
- NX_PPPOE_CLIENT_MIN_PACKET_PAYLOAD_SIZE：指定 PPPoE 客户端的最小数据包有效负载大小。如果数据包有效负载大小大于此值，则可以避免数据包链接。默认情况下，此值为 1520（以太网的最大有效负载大小 1500 + 以太网标头 14 + CRC 2 + 四字节对齐 4）。

第 3 章 - Azure RTOS NetX PPPoE 客户端服务说明

2021/4/29 •

本章旨在按字母顺序介绍如下所列的所有 Azure RTOS NetX PPPoE 客户端服务。

在以下 API 说明的"返回值"部分，以粗体显示的值不受 NX_DISABLE_ERROR_CHECKING 定义（用于禁用 API 错误检查）的影响，而非粗体值则完全遭到禁用。

- nx_pppoe_client_create 创建 PPPoE 客户端实例
- nx_pppoe_client_delete 删除 PPPoE 客户端实例
- nx_pppoe_client_host_uniq_set 设置 PPPoE 客户端主机唯一标识
- nx_pppoe_client_host_uniq_set_extended 设置 PPPoE 客户端主机唯一标识
- nx_pppoe_client_service_name_set 设置 PPPoE 客户端服务名称
- nx_pppoe_client_service_name_set_extended 设置 PPPoE 客户端服务名称
- nx_pppoe_client_session_connect 将 PPPoE 客户端会话连接至 PPPoE 服务器
- nx_pppoe_client_session_packet_send 发送 PPPoE 会话数据包
- nx_pppoe_client_session_terminate 终止 PPPoE 会话
- nx_pppoe_client_session_get 获取指定 PPPoE 会话信息

nx_pppoe_client_create

创建 PPPoE 客户端实例

原型

```
UINT  nx_pppoe_client_create(NX_PPPOE_CLIENT *pppoe_client_ptr,
                             CHAR *name, NX_IP *ip_ptr,
                             UINT interface_index,
                             NX_PACKET_POOL *pool_ptr,
                             VOID *stack_ptr, ULONG stack_size,
                             UINT priority,
                             VOID (*pppoe_link_driver)
                             (struct NX_IP_DRIVER_STRUCT *)
                             VOID (*pppoe_packet_receive)
                             (NX_PACKET *packet_ptr));
```

说明

此服务使用用户提供的链接驱动程序，为指定的 NetX IP 实例创建 PPPoE 客户端实例。如果尚未初始化和启用链接驱动程序，则 PPPoE 客户端软件会负责初始化链接驱动程序。

此外，应用程序必须为 PPPoE 客户端实例提供先前创建的数据包池，以用于内部数据包分配。

> **NOTE**
> 通常情况下，创建 NetX IP 线程的优先级最好高于创建 PPPoE 客户端线程的优先级。有关指定 IP 线程优先级的详细信息，请参阅 nx_ip_create 服务。

输入参数

- pppoe_client_ptr 指向 PPPoE 客户端控制块的指针。
- name 此 PPPoE 客户端实例的名称。
- ip_ptr 指向 IP 实例控制块的指针。

- interface_index 接口索引。
- pool_ptr 指向数据包池的指针。
- stack_ptr 指向 PPPoE 客户端线程堆栈区域起始位置的指针。
- stack_size 线程堆栈的大小（以字节为单位）。
- priority 内部 PPPoE 客户端线程的优先级 (1-31)。
- pppoe_link_driver 用户提供的链接驱动程序。
- pppoe_packet_receive 用户提供的数据包接收函数。

返回值

- NX_PPPOE_CLIENT_SUCCESS (0x00) 成功创建 PPPoE 客户端。
- NX_PPPOE_CLIENT_PTR_ERROR (0xD1) PPPoE 客户端、IP、数据包池或堆栈指针无效。
- NX_PPPOE_CLIENT_INVALID_INTERFACE (0xD2) 接口无效。
- NX_PPPOE_CLIENT_PACKET_PAYLOAD_ERROR (0xD3) 数据包池的有效负载大小无效。
- NX_PPPOE_CLIENT_MEMORY_SIZE_ERROR (0xD4) 内存大小无效。
- NX_PPPOE_CLIENT_PRIORITY_ERROR (0xD5) PPPoE 客户端线程的优先级无效。

支持的来源

初始化、线程

示例

```
/* Create “my_pppoe_client” for IP instance “my_ip”. */
status =  nx_pppoe_client_create(&my_pppoe_client, “my PPPoE Client”, &my_ip,
0, &my_pool, stack_start, 1024, 2,
link_driver, packet_receive);

/* If status is NX_PPPOE_CLIENT_SUCCESS, the “my_pppoe_client” was successfully created. */
```

nx_pppoe_client_delete

删除 PPPoE 客户端实例

原型

```
UINT nx_pppoe_client_delete(NX_PPPOE_CLIENT *pppoe_client_ptr);
```

说明

此服务用于删除先前创建的 PPPoE 客户端实例。

输入参数

- pppoe_client_ptr 指向 PPPoE 客户端控制块的指针

返回值

- NX_PPPOE_CLIENT_SUCCESS (0x00) 成功删除 PPPoE 客户端。
- NX_PPPOE_CLIENT_PTR_ERROR (0xD1) PPPoE 客户端指针无效。

支持的来源

线程数

示例

```
/* Delete PPPoE Client instance “my_pppoe_client”. */
status =  nx_pppoe_client_delete(&my_pppoe_client);

/* If status is NX_PPPOE_CLIENT_SUCCESS, the “my_pppoe_client” was successfully deleted. */
```

nx_pppoe_client_host_uniq_set

设置 PPPoE 客户端主机唯一标识

原型

```
UINT nx_pppoe_client_host_uniq_set(
                        NX_PPPOE_CLIENT *pppoe_client_ptr,
                        UCHAR *host_uniq);
```

说明

此服务用于设置 PPPoE 客户端主机唯一标识。主机使用 Host-Uniq，以将访问集中器与特定主机请求进行唯一关联。Host-Uniq 可以是主机选择的任何值和长度的二进制数据。

> **NOTE**
> 主机唯一标识必须为以 Null 结尾的字符串。

> **NOTE**
> 此服务已弃用。鼓励开发人员迁移至 nx_pppoe_client_host_uniq_set_extended()。

输入参数

- pppoe_client_ptr 指向 PPPoE 客户端控制块的指针。
- host_uniq 指向主机唯一标识的指针。主机唯一标识必须为以 Null 结尾的字符串。

返回值

- NX_PPPOE_CLIENT_SUCCESS (0x00) 成功设置 PPPoE 客户端主机唯一标识。
- NX_PPPOE_CLIENT_PTR_ERROR (0xD1) PPPoE 客户端指针无效。

支持的来源

初始化、线程

示例

```
/* Set service name for PPPoE Client instance “my_pppoe_client”. */
status =  nx_pppoe_client_host_uniq_set(&my_pppoe_client,
“220000000000000041000000”);

/* If status is NX_PPPOE_CLIENT_SUCCESS, the “my_pppoe_client” host unique has set. */
```

nx_pppoe_client_host_uniq_set_extended

设置 PPPoE 客户端主机唯一标识

原型

```
UINT nx_pppoe_client_host_uniq_set_extended(
                        NX_PPPOE_CLIENT *pppoe_client_ptr,
                        UCHAR *host_uniq,UINT host_uniq_length);
```

说明

此服务用于设置 PPPoE 客户端主机唯一标识。主机使用 Host-Uniq，以将访问集中器与特定主机请求进行唯一关联。Host-Uniq 可以是主机选择的任何值和长度的二进制数据。

> **NOTE**
>
> 主机唯一标识必须为以 Null 结尾的字符串。

> **NOTE**
>
> 此服务用于替代 nx_pppoe_client_host_uniq_set()。这一版本向此服务提供其他长度信息。

输入参数

- pppoe_client_ptr 指向 PPPoE 客户端控制块的指针。
- host_uniq 指向主机唯一标识的指针。主机唯一标识必须为以 Null 结尾的字符串。
- host_uniq_length host_uniq 的长度

返回值

- NX_PPPOE_CLIENT_SUCCESS (0x00) 成功设置 PPPoE 客户端主机唯一标识。
- NX_PPPOE_CLIENT_PTR_ERROR (0xD1) PPPoE 客户端指针无效。
- NX_SIZE_ERROR (0X09) 无法检查 host_uniq_length

支持的来源

初始化、线程

示例

```
/* Set service name for PPPoE Client instance “my_pppoe_client”. */
status =  nx_pppoe_client_host_uniq_set_extended(&my_pppoe_client,
“220000000000000041000000”,24);

/* If status is NX_PPPOE_CLIENT_SUCCESS, the “my_pppoe_client” host unique has set. */
```

nx_pppoe_client_service_name_set

设置 PPPoE 客户端服务名称

原型

```
UINT nx_pppoe_client_service_name_set(
                        NX_PPPOE_CLIENT *pppoe_client_ptr,
                        UCHAR *service_name);
```

说明

此服务用于设置 PPPoE 客户端服务名称。服务名称表示主机正在请求的服务。如果未设置服务名称，则表示主机将接受任意数量的服务。

> **NOTE**
> 服务名称必须为以 Null 结尾的字符串

> **NOTE**
> 此服务已弃用。鼓励开发人员迁移至 nx_pppoe_client_service_name_set_extended()。

输入参数

- pppoe_client_ptr 指向 PPPoE 客户端控制块的指针。
- service_name 指向服务名称的指针。服务名称必须为以 Null 结尾的字符串。

返回值

- NX_PPPOE_CLIENT_SUCCESS (0x00) 成功设置 PPPoE 客户端服务名称。
- NX_PPPOE_CLIENT_PTR_ERROR (0xD1) PPPoE 客户端指针无效。

支持的来源

初始化、线程

示例

```
/* Set service name for PPPoE Client instance “my_pppoe_client”. */
status =  nx_pppoe_client_service_name_set(&my_pppoe_client,
“BRAS”);

/* If status is NX_PPPOE_CLIENT_SUCCESS, the “my_pppoe_client” service name has set. */
```

nx_pppoe_client_service_name_set_extended

设置 PPPoE 客户端服务名称

原型

```
UINT nx_pppoe_client_service_name_set_extended(
                        NX_PPPOE_CLIENT *pppoe_client_ptr,
                        UCHAR *service_name,UINT service_name_length);
```

说明

此服务用于设置 PPPoE 客户端服务名称。服务名称表示主机正在请求的服务。如果未设置服务名称，则表示主机将接受任意数量的服务。

> **NOTE**
> 服务名称必须为以 Null 结尾的字符串。

> **NOTE**
> 此服务用于替代 nx_pppoe_client_service_name_set()。这一版本向此函数提供其他服务名称长度信息。

输入参数

- pppoe_client_ptr 指向 PPPoE 客户端控制块的指针。

- service_name 指向服务名称的指针。服务名称必须为以 Null 结尾的字符串。
- service_name_length service_name 的长度

返回值

- NX_PPPOE_CLIENT_SUCCESS (0x00) 成功设置 PPPoE 客户端服务名称。
- NX_PPPOE_CLIENT_PTR_ERROR (0xD1) PPPoE 客户端指针无效。
- NX_SIZE_ERROR (0X09) 无法检查 service_name_length

支持的来源

初始化、线程

示例

```
/* Set service name for PPPoE Client instance “my_pppoe_client”. */
status =  nx_pppoe_client_service_name_set_extended(&my_pppoe_client,
“BRAS”,4);

/* If status is NX_PPPOE_CLIENT_SUCCESS, the “my_pppoe_client” service name has set. */
```

nx_pppoe_client_session_connect

连接 PPPoE 客户端会话

原型

```
UINT nx_pppoe_client_session_connect(NX_PPPOE_CLIENT *pppoe_client_ptr,
                                     ULONG wait_option);
```

说明

此服务使用先前创建的 PPPoE 客户端，与指定的 PPPoE 服务器建立 PPPoE 会话连接。

> **NOTE**
> 必须在 nx_pppoe_client_create、nx_pppoe_client_host_uniq_set 和 nx_pppoe_client_service_name_set 之后调用此函数。

输入参数

- pppoe_client_ptr 指向 PPPoE 客户端控制块的指针。
- wait_option 建立连接时的等待选项。

返回值

- NX_PPPOE_CLIENT_SUCCESS (0x00) 成功连接 PPPoE 客户端会话。
- NX_PPPOE_CLIENT_PTR_ERROR (0xD1) PPPoE 客户端指针无效。

支持的来源

初始化、线程

示例

```
/* Enable PPPoE Client instance “my_pppoe_client”. */
status =  nx_pppoe_client_session_connect(&my_pppoe_client);

/* If status is NX_PPPOE_CLIENT_SUCCESS, the “my_pppoe_client” session connection has connected. */
```

nx_pppoe_client_session_packet_send

将 PPPoE 客户端数据包发送至指定会话

原型

```
UINT nx_pppoe_client_session_packet_send(
                                    NX_PPPOE_CLIENT *pppoe_client_ptr,
                                    NX_PACKET *packet_ptr);
```

说明

此服务使用指定的会话 ID 发送 PPPoE 数据包。

输入参数

- pppoe_client_ptr 指向 PPPoE 客户端控制块的指针。
- packet_ptr 指向 PPPoE 数据包的指针。

返回值

- NX_PPPOE_CLIENT_SUCCESS (0x00) 成功发送 PPPoE 客户端数据包。
- NX_PPPOE_CLIENT_PTR_ERROR (0xD1) PPPoE 客户端指针无效。
- NX_PPPOE_CLIENT_PACKET_PAYLOAD_ERROR (0xD3) PPPoE 客户端数据包无效。
- NX_PPPOE_CLIENT_SESSION_NOT_ESTABLISHED (0xD8) 未建立 PPPoE 会话。

支持的来源

初始化、线程

示例

```
/* Send PPPoE client packet to specified session, PPPoE Client instance “my_pppoe_client”. */
status =  nx_pppoe_client_session_packet_send(&my_pppoe_client, packet_ptr);

/* If status is NX_PPPOE_CLIENT_SUCCESS, the “my_pppoe_client” packet has sent. */
```

nx_pppoe_client_session_terminate

终止 PPPoE 客户端会话

原型

```
UINT nx_pppoe_client_session_terminate(
                                    NX_PPPOE_CLIENT *pppoe_client_ptr);
```

说明

此服务用于终止指定的 PPPoE 会话。

输入参数

- pppoe_client_ptr 指向 PPPoE 客户端控制块的指针。

返回值

- NX_PPPOE_CLIENT_SUCCESS (0x00) 成功终止 PPPoE 客户端会话。
- NX_PPPOE_CLIENT_PTR_ERROR (0xD1) PPPoE 客户端指针无效。

支持的来源

初始化、线程

示例

```
/* Terminates the specified PPPoE session, PPPoE Client instance “my_pppoe_client”. */
status =  nx_pppoe_client_session_terminate(&my_pppoe_client);

/* If status is NX_PPPOE_CLIENT_SUCCESS, the session has terminated. */
```

nx_pppoe_client_session_get

获取指定的 PPPoE 会话信息

原型

```
UINT nx_pppoe_client_session_get(NX_PPPOE_CLIENT *pppoe_client_ptr,
                                 ULONG *server_mac_msw,
                                 ULONG *server_mac_lsw,
                                 ULONG *session_id);
```

说明

此服务用于获取指定的 PPPoE 会话信息、服务器物理地址和会话 ID。

输入参数

- pppoe_server_ptr 指向 PPPoE 客户端控制块的指针。
- server_mac_msw 服务器物理地址 MSW 指针。
- server_mac_lsw 服务器物理地址 LSW 指针。
- session_id 会话 ID 指针。

返回值

- NX_PPPOE_CLIENT_SUCCESS (0x00) 成功获取 PPPoE 客户端会话。
- NX_PPPOE_CLIENT_PTR_ERROR (0xD1) PPPoE 客户端指针无效。
- NX_PPPOE_CLIENT_SESSION_NOT_ESTABLISHED (0xD8) 未建立 PPPoE 会话。

支持的来源

初始化、线程

示例

```
/* Gets the specified PPPoE session information, PPPoE Client instance “my_pppoe_client”. */
status =  nx_pppoe_client_session_get (&my_pppoe_client, &server_mac_msw, &server_mac_lsw, &session_id);

/* If status is NX_PPPOE_CLIENT_SUCCESS, the server physical address and session id
   of the session has got. */
```

第 1 章 - Azure RTOS NetX PPPoE 服务器简介

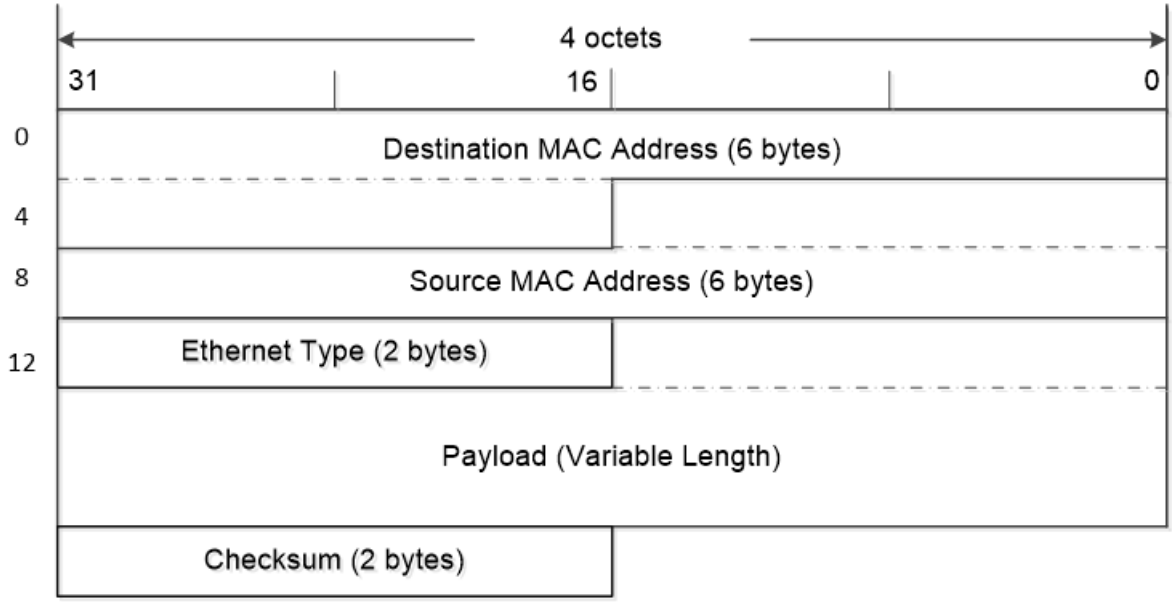
2021/4/29 •

以太网 PPP (PPPoE) 允许主机通过以太网（而非传统的基于字符的串行线路通信）连接到 PPP 服务器。“RFC 2516：一种传输以太网 PPP (PPPoE) 的方法”中介绍了 PPPoE 的技术详细信息。本文档重点介绍了 Azure RTOS NetX PPPoE 模块的详细信息。

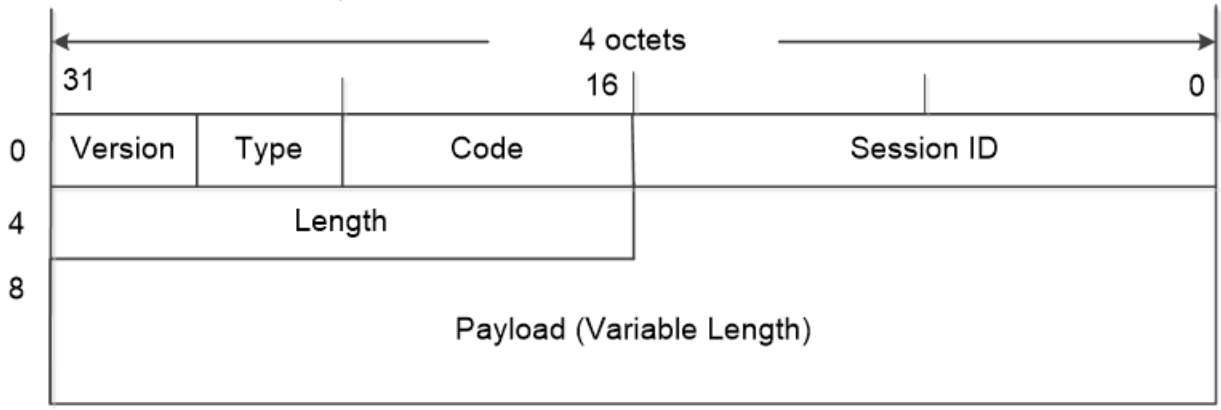
若要通过以太网提供点到点连接，每个 PPP 会话必须了解远程对等方的以太网地址，并建立唯一的会话标识符。

根据 RFC 2516，PPPoE 包含两个阶段：“发现”阶段和“PPPoE 会话”阶段。当主机（客户端）希望启动 PPP 会话时，它必须首先执行发现步骤来查找 PPPoE 服务器。此步骤还允许服务器和客户端识别彼此的以太网 MAC 地址和 SESSION_ID，它们将用于 PPP 会话的其余部分。

以太网帧如下所示：



PPPoE 的以太网有效负载如下所示：



PPPoE 发现阶段

PPPoE 发现阶段允许客户端从网络上的所有可用服务器中选择一台服务器，并高效地在交换 PPP 帧之前创建会话。在发现阶段结束时，客户端和服务器都应同意使用唯一的会话 ID，同时双方都需要知道对等方的 MAC 地址。

[illegible]	[illegible]	[illegible]
PPPoE 主动发现开始 (PADI)	0x09	从客户端到广播
PPPoE 主动发现要约 (PADO)	0x07	从服务器到客户端
PPPoE 主动发现请求 (PADR)	0x19	从客户端到服务器
PPPOE 主动发现会话-确认 (PADS)	0x65	从服务器到客户端
PPPoE 主动发现终止 (PADT)	0xa7	可从服务器或客户端启动

所有发现以太网帧都将 ETHER_TYPE 字段设置为值 0x8863。

PPPoE 会话消息

在客户端和服务器创建会话后，可以将 PPP 帧作为 PPPoE 会话消息进行传输。在会话期间，SESSION_ID 不能更改，并且必须是在发现阶段由服务器分配的值。

所有 PPPoE 会话以太网帧都将 ETHER_TYPE 字段设置为值 0x8864。

第 2 章 - 安装和使用 Azure RTOS NetX PPPoE 服务器

2021/4/29 •

本章介绍了与安装、设置和使用 Azure RTOS NetX PPPoE 服务器组件相关的各种问题。

产品分发

https://github.com/azure-rtos/netx 中提供了 NetX PPPoE 服务器。该程序包包含两个源文件和一个 PDF 文件（其中包含本文档），如下所示：

- nx_pppoe_server.h：NetX PPPoE 服务器的头文件
- nx_pppoe_server.c：NetX PPPoE 服务器的 C 源文件
- nx_pppoe_server.pdf：NetX PPPoE 服务器的 PDF 说明
- demo_netx_pppoe_server.c：NetX PPPoE 服务器演示

PPPoE 服务器安装

若要使用 NetX PPPoE 服务器，应将前面提到的整个分发包复制到安装了 NetX 的目录。例如，如果 NetX 安装在“\threadx\arm7\green”目录中，则应将 nx_pppoe_server.h 和 nx_pppoe_server.c 文件复制到该目录中。

使用 PPPoE 服务器

使用 NetX PPPoE 服务器非常容易。总体上，应用程序代码必须先包含 tx_api.h 和 nx_api.h，然后包含 nx_pppoe_server.h，才能分别使用 ThreadX 和 NetX。包含 nx_pppoe_server.h 之后，应用程序代码就可以进行本指南后面部分所述的 PPPoE 服务器函数调用。在生成过程中，应用程序还必须包含 nx_pppoe_server.c。此文件必须采用与其他应用程序文件相同的方式进行编译，并且其对象窗体必须与该应用程序的文件一起链接。这就是使用 NetX PPPoE 服务器所需的一切。

小型示例系统

下面的示例演示了如何使用 NetX PPPoE 服务器，如图 1.1 所述。在此示例中，第 50 行引入了 PPPoE 服务器 include 文件 nx_pppoe_server.h。接下来，在第 248 行的“thread_0_entry”中创建了 PPPoE 服务器。请注意，应该先创建 IP 实例，然后再创建 PPPoE 服务器。IP 实例是在第 165 行中创建并初始化的。HTTP 服务器控制块“pppoe_server”在前面的第 79 行中定义为全局变量。notify 函数是在第 257 行设置的。请注意，必须设置 pppoe_session_data_receive notify 函数。成功创建 IP 和 PPPoE 服务器后，在第 272 行，PPPoE 服务器通过调用 nx_pppoe_server_enable 建立 PPPoE 会话。

通常，PPPoE 模块应与 PPP 模块配合使用。在此示例中，在第 49 行引入了 PPP 服务器 include 文件 nx_ppp.h。接下来，在第 174 行创建了 PPP 服务器。第 182 行设置用于发送 PPP 数据包的函数。第 188-200 行设置 IP 地址，并定义 pap 协议。第 115-139 行设置 pap 协议的用户名和密码。

在 PPPoE 会话建立之后，在第 281 行，应用程序可以调用 nx_pppoe_server_session_get 来获取会话信息（客户端 MAC 地址和会话 ID）。

当应用程序不再处理 PPP 流量时，应用程序可以调用 PppCloseInd 或 nx_pppoe_server_session_terminate 来终止 PPPoE 会话。

NOTE

在此示例中，PPPoE 服务器与普通 IP 堆栈同时运行，并共享同一个以太网驱动程序。为普通 IP 实例（第 165 行）和 PPPoE 服务器实例（第 248 行）传递相同的以太网驱动程序。

NOTE

函数由 PPPoE 实现提供，供已定义的 NX_PPPoE_SERVER_SESSION_CONTROL_ENABELE 下的软件调用。

如果定义了此项，则会启用控制 PPPoE 会话的功能。

在应用程序调用特定 API 之前，PPPoE 服务器不会自动响应请求。如果未定义此项，则 PPPoE 服务器会自动响应请求。默认情况下，它在 nx_pppoe_server.h 中启用。

请注意，将 NX_PHYSICAL_HEADER 重新定义为 24，确保有足够的空间可用于填充物理标头。物理标头：14（以太网标头）+ 6（PPPoE 标头）+ 2（PPP 标头）+ 2（四字节对齐）。

```
/**************************************************************************/
/**************************************************************************/
/**                                                                       */
/** NetX PPPoE Server stack Component                                     */
/**                                                                       */
/** This is a small demo of the high-performance NetX PPPoE Server        */
/** stack. This demo includes IP instance, PPPoE Server and PPP Server    */
/** stack. Create one IP instance includes two interfaces to support      */
/** for normal IP stack and PPPoE Server, PPPoE Server can use the        */
/** mutex of IP instance to send PPPoE message when share one Ethernet    */
/** driver. PPPoE Server work with normal IP instance at the same time.   */
/**                                                                       */
/** Note1: Substitute your Ethernet driver instead of                     */
/** _nx_ram_network_driver before run this demo                           */
/**                                                                       */
/** Note2: Prerequisite for using PPPoE.                                  */
/** Redefine NX_PHYSICAL_HEADER to 24 to ensure enough space for filling  */
/** in physical header. Physical header:14(Ethernet header)               */
/** + 6(PPPoE header) + 2(PPP header) + 2(four-byte aligment)             */
/**                                                                       */
/**************************************************************************/
/**************************************************************************/


/****************************************************************/
/*                          NetX Stack                          */
/****************************************************************/

                                /*************************/
                                /* PPP Server            */
                                /*************************/

                                /*************************/
                                /* PPPoE Server          */
                                /*************************/
/*************************/     /*************************/
/* Normal Ethernet Type  */     /* PPPoE Ethernet Type   */
/*************************/     /*************************/
/*************************/     /*************************/
/* Interface 0           */     /* Interface 1           */
/*************************/     /*************************/

/****************************************************************/
/*                    Ethernet Dirver                           */
/****************************************************************/
```

```
#include     "tx_api.h"
#include     "nx_api.h"
#include     "nx_ppp.h"
#include     "nx_pppoe_server.h"

/* Defined NX_PPP_PPPOE_ENABLE if use Express Logic's PPP, since PPP module has been modified to match PPPoE
moduler under this definition. */
#ifdef NX_PPP_PPPOE_ENABLE

/*   If the driver is not initialized in other module, define */
/*   NX_PPPOE_SERVER_INITIALIZE_DRIVER_ENABLE to initialize the driver in PPPoE module. */
/*   In this demo, the driver has been initialized in IP module. */

#ifdef NX_PPPOE_SERVER_INITIALIZE_DRIVER_ENABLE

/*   NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE: If defined, enables the feature that */
/*   controls the PPPoE session. PPPoE server does not automatically response to the */
/*   request until application call specific API. */

/* Define the block size. */
#define     NX_PACKET_POOL_SIZE     ((1536 + sizeof(NX_PACKET)) * 30)
#define     DEMO_STACK_SIZE         2048
#define     PPPOE_THREAD_SIZE       2048

/* Define the ThreadX and NetX object control blocks... */
TX_THREAD           thread_0;

/* Define the packet pool and IP instance for normal IP instnace. */
NX_PACKET_POOL      pool_0;
NX_IP               ip_0;

/* Define the PPP Server instance. */
NX_PPP              ppp_server;

/* Define the PPPoE Server instance. */
NX_PPPOE_SERVER     pppoe_server;

/* Define the counters. */
CHAR                *pointer;
ULONG               error_counter;

/* Define thread prototypes. */
void     thread_0_entry(ULONG thread_input);

/***** Substitute your PPP driver entry function here *********/
extern void     _nx_ppp_driver(NX_IP_DRIVER *driver_req_ptr);

/***** Substitute your Ethernet driver entry function here *********/
extern void     _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr);

/* Define the callback functions. */
void     PppDiscoverReq(UINT interfaceHandle);
void     PppOpenReq(UINT interfaceHandle, ULONG length, UCHAR *data);
void     PppCloseRsp(UINT interfaceHandle);
void     PppCloseReq(UINT interfaceHandle);
void     PppTransmitDataReq(UINT interfaceHandle, ULONG length, UCHAR *data, UINT packet_id);
void     PppReceiveDataRsp(UINT interfaceHandle, UCHAR *data);

/* Define the porting layer function for Express Logic's PPP to simulate TTP's PPP. */
/* Functions to be provided by PPP for calling by the PPPoE Stack. */
void     ppp_server_packet_send(NX_PACKET *packet_ptr);

/* Define main entry point. */

int main()
{

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

```
    tx_kernel_enter();
}

UINT verify_login(CHAR *name, CHAR *password)
{

    if ((name[0] == 'm') &&
        (name[1] == 'y') &&
        (name[2] == 'n') &&
        (name[3] == 'a') &&
        (name[4] == 'm') &&
        (name[5] == 'e') &&
        (name[6] == (CHAR) 0) &&
        (password[0] == 'm') &&
        (password[1] == 'y') &&
        (password[2] == 'p') &&
        (password[3] == 'a') &&
        (password[4] == 's') &&
        (password[5] == 's') &&
        (password[6] == 'w') &&
        (password[7] == 'o') &&
        (password[8] == 'r') &&
        (password[9] == 'd') &&
        (password[10] == (CHAR) 0))
            return(NX_SUCCESS);
    else
        return(NX_PPP_ERROR);
}

/* Define what the initial system looks like. */

void tx_application_define(void *first_unused_memory)
{

    UINT status;

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

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

    /* Create a packet pool for normal IP instance. */
    status = nx_packet_pool_create(&pool_0, "NetX Main Packet Pool",
                                   (1536 + sizeof(NX_PACKET)),
                                   pointer, NX_PACKET_POOL_SIZE);
                                   pointer = pointer + NX_PACKET_POOL_SIZE;

    /* Check for error. */
    if (status)
        error_counter++;

    /* Create an normal IP instance. */
    status = nx_ip_create(&ip_0, "NetX IP Instance", IP_ADDRESS(192, 168, 100, 43),
                        0xFFFFFF00UL, &pool_0, _nx_ram_network_driver,
                        pointer, 2048, 1);
                        pointer = pointer + 2048;

    /* Check for error. */
    if (status)
        error_counter++;

    /* Create the PPP instance. */
    status = nx_ppp_create(&ppp_server, "PPP Instance", &ip_0, pointer, 2048, 1,
                           &pool_0, NX_NULL, NX_NULL);
    pointer = pointer + 2048;

    /* Check for PPP create error. */
    if (status)
```

```
        error_counter++;

    /* Set the PPP packet send function. */
    status = nx_ppp_packet_send_set(&ppp_server, ppp_server_packet_send);

    /* Check for PPP packet send function set error. */
    if (status)
        error_counter++;

    /* Define IP address. This PPP instance is effectively the server since it has both IP addresses. */
    status = nx_ppp_ip_address_assign(&ppp_server, IP_ADDRESS(192, 168, 10, 43),
IP_ADDRESS(192, 168, 10, 44));

    /* Check for PPP IP address assign error. */
    if (status)
        error_counter++;

    /* Setup PAP, this PPP instance is effectively the server since it will verify the name and password. */
    status = nx_ppp_pap_enable(&ppp_server, NX_NULL, verify_login);

    /* Check for PPP PAP enable error. */
    if (status)
        error_counter++;

    /* Attach an interface for PPP. */
    status = nx_ip_interface_attach(&ip_0, "Second Interface For PPP",
                            IP_ADDRESS(0, 0, 0, 0), 0, nx_ppp_driver);

    /* Check for error. */
    if (status)
        error_counter++;

    /* Enable ARP and supply ARP cache memory for Normal IP Instance. */
    status = nx_arp_enable(&ip_0, (void *) pointer, 1024);
    pointer = pointer + 1024;

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

    /* Enable ICMP */
    status = nx_icmp_enable(&ip_0);
    if(status)
        error_counter++;

    /* Enable UDP traffic. */
    status = nx_udp_enable(&ip_0);
    if (status)
        error_counter++;

    /* Enable TCP traffic. */
    status = nx_tcp_enable(&ip_0);
    if (status)
        error_counter++;

    /* Create the main thread. */
    tx_thread_create(&thread_0, "thread 0", thread_0_entry, 0,
                    pointer, DEMO_STACK_SIZE,
                    4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);
                    pointer = pointer + DEMO_STACK_SIZE;
}

/* Define the test threads. */

void     thread_0_entry(ULONG thread_input)
{
UINT     status;
ULONG    ip_status;
```

```
    /* Create the PPPoE instance. */
    status = nx_pppoe_server_create(&pppoe_server, (UCHAR *)"PPPoE Server", &ip_0, 0,
                    _nx_ram_network_driver, &pool_0, pointer, PPPOE_THREAD_SIZE, 4);
    pointer = pointer + PPPOE_THREAD_SIZE;
    if (status)
    {
        error_counter++;
        return;
    }

    /* Set the callback notify function. */
    status = nx_pppoe_server_callback_notify_set(&pppoe_server, PppDiscoverReq,
        PppOpenReq, PppCloseRsp, PppCloseReq, PppTransmitDataReq, PppReceiveDataRsp);

    if (status)
    {
        error_counter++;
        return;
    }

    #ifdef NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE
        /* Call function function to set the default service Name. */
        /* PppInitInd(length, aData); */
    #endif /* NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE */

    /* Enable PPPoE Server. */
    status = nx_pppoe_server_enable(&pppoe_server);
    if (status)
    {
        error_counter++;
        return;
    }

    /* Get the PPPoE Client physical address and Session ID after establish PPPoE Session. */
    /*
        status = nx_pppoe_server_session_get(&pppoe_server, interfaceHandle, &client_mac_msw,
&client_mac_lsw, &session_id);
        if (status)
            error_counter++;
    */

    /* Wait for the link to come up. */
    status = nx_ip_interface_status_check(&ip_0, 1, NX_IP_ADDRESS_RESOLVED,
                                          &ip_status, NX_WAIT_FOREVER);
    if (status)
    {
        error_counter++;
        return;
    }

    #ifdef NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE
        /* Call PPPoE function to terminate the PPPoE Session. */
        /* PppCloseInd(interfaceHandle, causeCode); */
    #endif /* NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE */

}

void     PppDiscoverReq(UINT interfaceHandle)
{

    /* Receive the PPPoE Discovery Initiation Message. */

    #ifdef NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE
        /* Call PPPoE function to allow TTP's software to define the Service Name field of the PADO packet.
*/
        PppDiscoverCnf(0, NX_NULL, interfaceHandle);
    #endif /* NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE */
}
```

```
void     PppOpenReq(UINT interfaceHandle, ULONG length, UCHAR *data)
{

    /* Get the notify that receive the PPPoE Discovery Request Message. */

    #ifdef NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE
        /* Call PPPoE function to allow TTP's software to accept the PPPoE session. */
        PppOpenCnf(NX_TRUE, interfaceHandle);
    #endif /* NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE */
}

void     PppCloseRsp(UINT interfaceHandle)
{

    /* Get the notify that receive the PPPoE Discovery Terminate Message. */

    #ifdef NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE
        /* Call PPPoE function to allow TTP's software to confirm that the handle has been freed. */
        PppCloseCnf(interfaceHandle);
    #endif /* NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE */
}

void     PppCloseReq(UINT interfaceHandle)
{

    /* Get the notify that PPPoE Discovery Terminate Message has been sent. */

}

void     PppTransmitDataReq(UINT interfaceHandle, ULONG length, UCHAR *data, UINT packet_id)
{

    NX_PACKET *packet_ptr;

    /* Get the notify that receive the PPPoE Session data. */

    /* Call PPP Server to receive the PPP data fame. */
    packet_ptr = (NX_PACKET *)(packet_id);
    nx_ppp_packet_receive(&ppp_server, packet_ptr);

    #ifdef NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE
        /* Call PPPoE function to confirm that the data has been processed. */
        PppTransmitDataCnf(interfaceHandle, data, packet_id);
    #endif /* NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE */
}

void     PppReceiveDataRsp(UINT interfaceHandle, UCHAR *data)
{

    /* Get the notify that the PPPoE Session data has been sent. */

}

/* PPP Server send function. */
void     ppp_server_packet_send(NX_PACKET *packet_ptr)
{

    /* For Express Logic's PPP test, the session should be the first session, so set interfaceHandle as 0.
*/
    UINT interfaceHandle = 0;

    #ifdef NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE
        while(packet_ptr)
        {

            /* Call functions to be provided by PPPoE for TTP. */
            PppReceiveDataInd(interfaceHandle, (packet_ptr -> nx_packet_append_ptr -
            packet_ptr -> nx_packet_prepend_ptr), packet_ptr -> nx_packet_prepend_ptr);
```

```
            /* Move to the next packet structure. */
            packet_ptr = packet_ptr -> nx_packet_next;
        }
    #else
        /* Directly Call PPPoE send function to send out the data through PPPoE module. */
        nx_pppoe_server_session_packet_send(&pppoe_server, interfaceHandle, packet_ptr);
    #endif /* NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE */
}
#endif /* NX_PPPOE_SERVER_INITIALIZE_DRIVER_ENABLE */

#endif /* NX_PPP_PPPOE_ENABLE */
```

图 1.1 与 NetX 配合使用的 PPPoE 服务器示例

配置选项

可通过几个配置选项生成适用于 NetX 的 PPPoE 服务器。以下列表详细介绍了每个配置选项：

- NX_DISABLE_ERROR_CHECKING：定义后，此选项会删除基本的 PPPoE 服务器错误检查。通常会在调试应用程序后使用此选项。
- NX_PPPOE_SERVER_SESSION_CONTROL_ENABLE：如果定义了此项，则会启用控制 PPPOE 会话的功能。在应用程序调用特定 API 之前，PPPoE 服务器不会自动响应请求。
- NX_PPPOE_SERVER_INITIALIZE_DRIVER_ENABLE：如果定义了此项，则会启用在 PPPoE 模块中初始化以太网驱动程序的功能，默认情况下会禁用该功能。
- NX_PPPOE_SERVER_THREAD_TIME_SLICE：PPPOE 服务器线程的时间切片选项。默认情况下，此值为 TX_NO_TIME_SLICE。
- NX_PPPOE_SERVER_MAX_CLIENT_SESSION_NUMBER：此项定义并发客户端会话的最大数量。默认情况下，此值为 10。
- NX_PPPOE_SERVER_MAX_HOST_UNIQ_SIZE：此项定义 Host-Uniq 的最大大小。默认情况下，此值为 32。
- NX_PPPOE_SERVER_MAX_RELAY_SESSION_ID_SIZE：此项定义 Relay-Session-Id 的最大大小。默认情况下，此值为 12。
- NX_PPPOE_SERVER_MIN_PACKET_PAYLOAD_SIZE：指定 PPPoE 服务器的最小数据包有效负载大小。如果数据包有效负载大小大于此值，则可以避免数据包链接。默认情况下，此值为 1520（以太网的最大有效负载大小 1500 + 以太网标头 14 + CRC 2 + 四字节对齐 4）。
- NX_PPPOE_SERVER_PACKET_TIMEOUT：此项定义分配数据包时或将数据追加到数据包时的等待部分（以时钟周期为单位）。默认情况下，此值为 NX_IP_PERIODIC_RATE（100 个时钟周期）。
- NX_PPPOE_SERVER_START_SESSION_ID：此项定义分配到 PPPoE 会话的起始会话 ID。默认情况下，此值为 0X4944（ID 的 ASCII 值）。

第 3 章 - Azure RTOS NetX PPPoE 服务器服务的说明

2021/4/29 •

本章旨在按字母顺序介绍如下所列的所有 Azure RTOS NetX PPPoE 服务器服务。

在以下 API 说明中的"返回值"部分，以粗体显示的值不受用于禁用 API 错误检查的 NX_DISABLE_ERROR_CHECKING 定义影响，而不以粗体显示的值则已完全禁用 。

- nx_pppoe_server_create：创建 PPPoE 服务器实例
- nx_pppoe_server_ac_name_set：设置访问集中器名称
- nx_pppoe_server_delete：删除 PPPoE 服务器实例
- nx_pppoe_server_enable：启用 PPPoE 服务器服务
- nx_pppoe_server_disable：禁用 PPPoE 服务器服务
- nx_pppoe_server_callback_notify_set：设置 PPPoE 服务器回调通知功能
- nx_pppoe_server_service_name_set：设置 PPPoE 服务器服务名称
- nx_pppoe_server_session_send：将 PPPoE 服务器数据发送到指定的会话
- nx_pppoe_server_session_packet_send：将 PPPoE 服务器数据包发送到指定的会话
- nx_pppoe_server_session_terminate：终止指定的 PPPoE 会话
- nx_pppoe_server_session_get：获取指定的会话信息

nx_pppoe_server_create

创建 PPPoE 服务器实例

原型

```
UINT nx_pppoe_server_create(NX_PPPOE_SERVER *pppoe_server_ptr,
                            CHAR *name, NX_IP *ip_ptr,
                            UINT interface_index,
                            VOID (*pppoe_link_driver)
                            (struct NX_IP_DRIVER_STRUCT *)
                            NX_PACKET_POOL *pool_ptr,
                            VOID *stack_ptr, ULONG stack_size,
                            UINT priority);
```

说明

此服务使用用户提供的链接驱动程序，为指定的 NetX IP 实例创建 PPPoE 服务器实例。如果尚未初始化和启用链接驱动程序，则 PPPoE 服务器软件会负责初始化链接驱动程序。

此外，应用程序必须为 PPPoE 服务器实例提供先前创建的数据包池，以用于内部数据包分配。

请注意，通常情况下，创建 NetX IP 线程的优先级最好高于创建 PPPoE 服务器线程的优先级。有关指定 IP 线程优先级的详细信息，请参阅"nx_ip_create"服务。

输入参数

- pppoe_server_ptr：指向 PPPoE 服务器控制块的指针。
- name：此 PPPoE 服务器实例的名称。
- ip_ptr：指向 IP 实例控制块的指针。
- interface_index：接口索引。
- pppoe_link_driver：用户提供的链接驱动程序。
- pool_ptr：指向数据包池的指针。
- stack_ptr：指向 PPPoE 服务器线程堆栈区域起始位置的指针。
- stack_size：线程堆栈的大小（以字节为单位）。
- priority：内部 PPPoE 服务器线程的优先级 (1-31)。

返回值

- NX_PPPOE_SERVER_SUCCESS：(0x00) 已成功创建 PPPoE 服务器。
- NX_PPPOE_SERVER_PTR_ERROR：(0xC1) PPPoE 服务器、IP、数据包池或堆栈指针无效。
- NX_PPPOE_SERVER_INVALID_INTERFACE：(0xC2) 接口无效。
- NX_PPPOE_SERVER_PACKET_PAYLOAD_ERROR：(0xC3) 数据包池的有效负载大小无效。
- NX_PPPOE_SERVER_MEMORY_SIZE_ERROR：(0xC4) 内存大小无效。
- NX_PPPOE_SERVER_PRIORITY_ERROR：(0xC5) PPPoE 服务器线程的优先级无效。

允许来自

初始化、线程

示例

```
/* Create "my_pppoe_server" for IP instance "my_ip". */
status = nx_pppoe_server_create(&my_pppoe_server, "my PPPoE Server", &my_ip,
                                &my_pool, stack_start, 1024, 2);

/* If status is NX_PPPOE_SERVER_SUCCESS, the "my_pppoe_server" was successfully created. */
```

nx_pppoe_server_delete

删除 PPPoE 服务器实例

原型

```
UINT nx_pppoe_server_delete(NX_PPPOE_SERVER *pppoe_server_ptr);
```

说明

此服务用于删除先前创建的 PPPoE 服务器实例。

输入参数

- pppoe_server_ptr：指向 PPPoE 服务器控制块的指针。

返回值

- NX_PPPOE_SERVER_SUCCESS：(0x00) 已成功删除 PPPoE 服务器。
- NX_PPPOE_SERVER_PTR_ERROR：(0xC1) PPPoE 服务器指针无效。

允许来自

线程数

示例

```
/* Delete PPPoE Server instance "my_pppoe_server". */
status = nx_pppoe_server_delete(&my_pppoe_server);

/* If status is NX_PPPOE_SERVER_SUCCESS, the "my_pppoe_server" was successfully deleted. */
```

nx_pppoe_server_enable

启用 PPPoE 服务器服务

原型

```
UINT nx_pppoe_server_enable(NX_PPPOE_SERVER *pppoe_server_ptr);
```

说明

此服务用于启用 PPPoE 服务器服务。

> **NOTE**
> 必须在 nx_pppoe_server_create 和 nx_pppoe_server_callback_notify_set 之后调用此函数 。

输入参数

- pppoe_server_ptr：指向 PPPoE 服务器控制块的指针。

返回值

- NX_PPPOE_SERVER_SUCCESS：(0x00) 已成功删除 PPPoE 服务器。
- NX_PPPOE_SERVER_PTR_ERROR：(0xC1) PPPoE 服务器指针无效。

允许来自

初始化、线程

示例

```
/* Enable PPPoE Server instance "my_pppoe_server". */
status = nx_pppoe_server_enable(&my_pppoe_server);

/* If status is NX_PPPOE_SERVER_SUCCESS, the "my_pppoe_server" service has enabled. */
```

nx_pppoe_server_disable

禁用 PPPoE 服务器服务

原型

```
UINT nx_pppoe_server_disable(NX_PPPOE_SERVER *pppoe_server_ptr);
```

说明

此服务用于禁用 PPPoE 服务器服务。

输入参数

- pppoe_server_ptr：指向 PPPoE 服务器控制块的指针。

返回值

- NX_PPPOE_SERVER_SUCCESS：(0x00) 已成功删除 PPPoE 服务器。

- NX_PPPOE_SERVER_PTR_ERROR：(0xC1) PPPoE 服务器指针无效。

允许来自

初始化、线程

示例

```
/* Disable PPPoE Server instance "my_pppoe_server". */
status = nx_pppoe_server_disable(&my_pppoe_server);

/* If status is NX_PPPOE_SERVER_SUCCESS, the "my_pppoe_server" service has disabled. */
```

nx_pppoe_server_callback_notify_set

设置 PPPoE 服务器回调通知函数

原型

```
UINT nx_pppoe_server_callback_notify_set(
        NX_PPPOE_SERVER *pppoe_server_ptr,
        VOID (* pppoe_discover_initiation_notify)(UINT session_index,
                ULONG length, UCHAR *data),
        VOID (* pppoe_discover_request_notify)(UINT session_index),
        VOID (* pppoe_discover_terminate_notify)(UINT session_index),
        VOID (* pppoe_discover_terminate_confirm)(UINT session_index),
        VOID (* pppoe_data_receive_notify)(UINT session_index,
                ULONG length, UCHAR *data, UINT packet_id),
        VOID (* pppoe_discover_notify)(UINT session_index, UCHAR *data))
```

说明

此服务用于设置 PPPoE 服务器回调通知函数。

> **NOTE**
> 必须在 nx_pppoe_server_enable 之前调用此函数，并且必须设置 pppoe_data_receive_notify 函数指针，否则 nx_pppoe_server_enable 将会失败 。

输入参数

- pppoe_server_ptr：指向 PPPoE 服务器控制块的指针。
- pppoe_discover_initiation_notify：每当收到 PPPoE 发现启动消息时要调用的应用程序函数。如果此值为 NULL，则会禁用发现启动回调函数。
- pppoe_discover_request_notify：每当收到 PPPoE 发现请求消息时要调用的应用程序函数。如果此值为 NULL，则会禁用发现请求回调函数。
- pppoe_discover_terminate_notify：每当收到 PPPoE 发现终止消息时要调用的应用程序函数。如果此值为 NULL，则会禁用发现终止回调函数。
- pppoe_discover_terminate_confirm：每当发送 PPPoE 发现终止消息时要调用的应用程序函数。如果此值为 NULL，则会禁用发现终止回调函数。
- pppoe_data_receive_notify：每当收到 PPPoE 数据消息时要调用的应用程序函数。此值不得为零。
- pppoe_data_send_notify：每当发送 PPPoE 数据消息时要调用的应用程序函数。如果此值为 NULL，则会禁用数据发送回调函数。

返回值

- NX_PPPOE_SERVER_SUCCESS：(0x00) 已成功删除 PPPoE 服务器。
- NX_PPPOE_SERVER_PTR_ERROR：(0xC1) PPPoE 服务器指针或函数指针无效。

允许来自

初始化、线程

示例

```
/* Set PPPoE Server callback notify functions, PPPoE Server instance "my_pppoe_server". */

status = nx_pppoe_server_disable(&my_pppoe_server,
                pppoe_discovery_initiation_notify,
                pppoe_discovery_request_notify,
                pppoe_discovery_terminate_notify,
                pppoe_discovery_terminate_confirm,
                pppoe_data_receive_notify,
                pppoe_data_send_notify);

/* If status is NX_PPPOE_SERVER_SUCCESS, the callback notify functions for "my_pppoe_server" service has
set. */
```

nx_pppoe_server_ac_name_set

设置访问集中器名称

原型

```
UINT nx_pppoe_server_ac_name_set(
    NX_PPPOE_SERVER *pppoe_server_ptr,
    CHAR *ac_name, UINT ac_name_length,
);
```

说明

此函数用于设置访问集中器名称函数调用。

> **NOTE**
> ac_name 的字符串必须以 NULL 结尾，并且 ac_name 的长度与自变量列表中指定的长度匹配。

输入参数

- pppoe_server_ptr：指向 PPPoE 服务器控制块的指针。
- ac_name：访问集中器名称。
- ac_name_length：ac_ame 的长度。

返回值

- NX_PPPOE_SERVER_SUCCESS：(0x00) 已成功设置 PPPoE 服务器。
- NX_PPPOE_SERVER_PTR_ERROR：(0xC1) PPPoE 服务器、IP、数据包池或堆栈指针无效。
- NX_SIZE_ERROR：(0x09) 检查 name_length 失败。

允许来自

初始化、线程

示例

```
/* Set "my PPPoE ac name" for Server instance "my_pppoe_server". */
status = nx_pppoe_server_ac_name_set(&my_pppoe_server, "my PPPoE ac name",16);

/* If status is NX_PPPOE_SERVER_SUCCESS, the "my PPPoE ac name" was successfully set. */
```

nx_pppoe_server_service_name_set

设置 PPPoE 服务器服务名称

原型

```
UINT nx_pppoe_server_service_name_set(
        NX_PPPOE_SERVER *pppoe_server_ptr,
        UCHAR **service_name, UINT service_name_count);
```

说明

此服务用于设置 PPPoE 服务器服务名称。

输入参数

- pppoe_server_ptr：指向 PPPoE 服务器控制块的指针。

返回值

- NX_PPPOE_SERVER_SUCCESS：(0x00) 已成功删除 PPPoE 服务器。
- NX_PPPOE_SERVER_PTR_ERROR：(0xC1) PPPoE 服务器指针无效。

允许来自

初始化、线程

示例

```
CHAR *nx_pppoe_service_name[] =
{
    "XBB",
    "PRINTER",
    NX_NULL
};

/* Set service name for PPPoE Server instance "my_pppoe_server". */
status = nx_pppoe_server_service_namet_set(&my_pppoe_server,
                                    nx_pppoe_service_name, 2);

/* If status is NX_PPPOE_SERVER_SUCCESS, the "my_pppoe_server" service name has set. */
```

nx_pppoe_server_session_send

将 PPPoE 服务器数据发送到指定的会话

原型

```
UINT nx_pppoe_server_session_send (NX_PPPOE_SERVER *pppoe_server_ptr,
                UINT session_index, UCHAR *data_ptr, UINT data_length);
```

说明

此服务使用指定的会话 ID 发送 PPPoE 帧。

输入参数

- pppoe_server_ptr：指向 PPPoE 服务器控制块的指针。
- session_index：会话的索引。
- data_ptr：指向 PPPoE 服务器数据帧起始位置的指针。
- data_length：PPPoE 服务器数据帧的长度。

返回值

- NX_PPPOE_SERVER_SUCCESS：(0x00) 已成功删除 PPPoE 服务器。
- NX_PPPOE_SERVER_PTR_ERROR：(0xC1) PPPoE 服务器指针无效。
- NX_PPPOE_SERVER_NOT_ENABLED：(0xC6) 未启用 PPPoE 服务器服务。
- NX_PPPOE_SERVER_INVALID_SESSION：(0xC7) PPPoE 会话索引无效。
- NX_PPPOE_SERVER_SESSION_NOT_ESTABLISHED：(0xC8) 未建立 PPPoE 会话。

允许来自

初始化、线程

示例

```
/* Send PPPoE Server data to specified session, PPPoE Server instance "my_pppoe_server". */
status = nx_pppoe_server_session_send(&my_pppoe_server, 0, my_data_ptr, 1400);

/* If status is NX_PPPOE_SERVER_SUCCESS, the "my_pppoe_server" data has sent. */
```

nx_pppoe_server_session_packet_send

将 PPPoE 服务器数据包发送到指定的会话

原型

```
UINT nx_pppoe_server_session_packet_send (
        NX_PPPOE_SERVER *pppoe_server_ptr,
        UINT session_index, NX_PACKET *packet_ptr);
```

说明

此服务使用指定的会话 ID 发送 PPPoE 数据包。

输入参数

- pppoe_server_ptr：指向 PPPoE 服务器控制块的指针。
- session_index：会话的索引。
- packet_ptr：指向 PPPoE 数据包的指针。

返回值

- NX_PPPOE_SERVER_SUCCESS：(0x00) 已成功删除 PPPoE 服务器。
- NX_PPPOE_SERVER_PTR_ERROR：(0xC1) PPPoE 服务器指针无效。
- NX_PPPOE_SERVER_PACKET_PAYLOAD_ERROR：(0xC3) PPPoE 服务器数据包无效。
- NX_PPPOE_SERVER_NOT_ENABLED：(0xC6) 未启用 PPPoE 服务器服务。
- NX_PPPOE_SERVER_INVALID_SESSION：(0xC7) PPPoE 会话索引无效。
- NX_PPPOE_SERVER_SESSION_NOT_ESTABLISHED：(0xC8) 未建立 PPPoE 会话。

允许来自

初始化、线程

示例

```
/* Send PPPoE Server data to specified session, PPPoE Server instance "my_pppoe_server". */
status = nx_pppoe_server_session_packet_send(&my_pppoe_server, 0, packet_ptr);

/* If status is NX_PPPOE_SERVER_SUCCESS, the "my_pppoe_server" packet has sent. */
```

nx_pppoe_server_session_terminate

终止指定的 PPPoE 会话

原型

```
UINT nx_pppoe_server_session_terminate(
        NX_PPPOE_SERVER *pppoe_server_ptr,
        UINT session_index);
```

说明

此服务用于终止指定的 PPPoE 会话。

输入参数

- pppoe_server_ptr：指向 PPPoE 服务器控制块的指针。
- session_index：会话的索引。

返回值

- NX_PPPOE_SERVER_SUCCESS：(0x00) 已成功删除 PPPoE 服务器。
- NX_PPPOE_SERVER_PTR_ERROR：(0xC1) PPPoE 服务器指针无效。
- NX_PPPOE_SERVER_NOT_ENABLED：(0xC6) 未启用 PPPoE 服务器服务。
- NX_PPPOE_SERVER_INVALID_SESSION：(0xC7) PPPoE 会话索引无效。
- NX_PPPOE_SERVER_SESSION_NOT_ESTABLISHED：(0xC8) 未建立 PPPoE 会话。

允许来自

初始化、线程

示例

```
/* Terminates the specified PPPoE session, PPPoE Server instance "my_pppoe_server". */
status = nx_pppoe_server_session_send(&my_pppoe_server, 0);

/* If status is NX_PPPOE_SERVER_SUCCESS, the session indexed with 0 has terminated. */
```

nx_pppoe_server_session_get

获取指定的 PPPoE 会话信息

原型

```
UINT nx_pppoe_server_session_get(NX_PPPOE_SERVER *pppoe_server_ptr,
                                 UINT session_index
                                 ULONG *client_mac_msw,
                                 ULONG *client_mac_lsw,
                                 ULONG *session_id);
```

说明

此服务用于获取指定的 PPPoE 会话信息、客户端物理地址和会话 ID。

输入参数

- pppoe_server_ptr：指向 PPPoE 服务器控制块的指针。
- session_index：会话的索引。
- client_mac_msw：客户端物理地址 MSW 指针。
- client_mac_lsw：客户端物理地址 MSW 指针。

- session_id：会话 ID 指针。

返回值

- NX_PPPOE_SERVER_SUCCESS：(0x00) 已成功删除 PPPoE 服务器。
- NX_PPPOE_SERVER_PTR_ERROR：(0xC1) PPPoE 服务器指针无效。
- NX_PPPOE_SERVER_INVALID_SESSION：(0xC7) PPPoE 会话索引无效。
- NX_PPPOE_SERVER_SESSION_NOT_ESTABLISHED：(0xC8) 未建立 PPPoE 会话。

允许来自

初始化、线程

示例

```
/* Gets the specified PPPoE session information, PPPoE Server instance "my_pppoe_server". */
status = nx_pppoe_server_session_get (&my_pppoe_server, 0, &client_mac_msw, &client_mac_lsw, &session_id);

/* If status is NX_PPPOE_SERVER_SUCCESS, the client physical address and session id of the session indexed
with 0 has got. */
```

PppInitInd

配置默认服务名称

原型

```
VOID PppInitnd(UINT length, UCHAR *aData);
```

说明

PPPoE 软件将公开此函数，以允许 TTP 的软件配置 PPPoE 在筛选传入 PADI 请求时应该使用的“默认服务名称”。PPPoE 软件应记住此信息，如果收到了包含匹配服务名称的 PADI 数据包，则它应该调用 PppDiscoverReq。

输入参数

- length：默认服务名称的长度。
- aData：默认服务名称。

返回值

无

允许来自

初始化、线程

示例

```
/* Configure the default Service Name. */
PppInitInd (3, "XBB");
```

PppDiscoverCnf

定义 PADO 数据包的“服务名称”字段

原型

```
VOID PppDiscoverCnf (UINT length, UCHAR *aData, UINT interfaceHandle);
```

说明

PPPoE 软件将公开此函数，以允许 TTP 的软件定义 PADO 数据包的"服务名称"字段。在调用 PppDiscoverCnf 之前，PPPoE 软件不应发送 PADO。

PADO 数据包应该包含 PPPoE 软件初始化时定义的访问集中器名称（使用 RFC2516 中定义的标记 ID 0x0102）。

可在 aData 中传递多个服务名称，其中每个名称以 null 结尾。

Null 字符用作分隔符，以便在需要传入其他命令作为服务名称的一部分时提供最大的灵活性。

输入参数

- length：默认服务名称的长度。
- aData：默认服务名称。
- interfaceHandle：接口句柄。

返回值

无

允许来自

初始化、线程

示例

```
/* Define the Service Name field of the PADO packet. */
PppDiscoverCnf (3, "XBB", 0);
```

PppOpenCnf

接受或拒绝 PPPoE 会话

原型

```
VOID PppOpenCnf (UCHAR accept, UINT interfaceHandle);
```

说明

PPPoE 软件将公开此函数，以允许 TTP 的软件接受或拒绝 PPPoE 会话。作为响应，PPPoE 堆栈应接受连接，并分配一个与 interfaceHandle 关联的唯一 PPPoE Session_ID 编号。

输入参数

- accept：如果要接受连接，则此项为 NX_TRUE。
- interfaceHandle：接口句柄。

返回值

无

允许来自

初始化、线程

示例

```
/* Accept the connection. */
PppOpenCnf(NX_TRUE, 0);
```

PppCloseInd

关闭 PPPoE 会话

原型

```
VOID PppCloseInd (UINT interfaceHandle, UCHAR *causeCode);
```

说明

PPPoE 软件将公开此函数，以允许 TTP 的软件关闭 PPPoE 会话。

PPPoE 软件将在 PADT 消息的 Generic-Error 标记 (0x0203) 中指示原因代码字符串

输入参数

- interfaceHandle：接口句柄。
- causeCode：以 Null 结尾的字符串，用于发送有关从 PPPoE 服务器关闭连接的原因的信息。

返回值

无

允许来自

初始化、线程

示例

```
/* Close a PPPoE session. */
PppCloseInd(0, NX_NULL);
```

PppCloseCnf

确认已释放句柄

原型

```
VOID PppCloseCnf (UINT interfaceHandle);
```

说明

PPPoE 软件将公开此函数，以允许 TTP 的软件确认句柄已释放。

输入参数

- interfaceHandle：接口句柄。

返回值

无

允许来自

初始化、线程

示例

```
/* Confirm that the handle has been freed. */
PppCloseCnf(0);
```

PppTransmitDataCnf

允许确认先前的 PPP 数据

原型

```
VOID PppTransmitDataCnf (UINT interfaceHandle, UCHAR *aData,
                         UINT packet_id);
```

说明

PPPoE 软件将公开此函数，以允许确认先前的 PppTransmitDataReq。这意味着，TTP 的软件已经为来自 PPPoE 的新 PPP 帧做好了准备。

输入参数

- interfaceHandle：接口句柄。
- aData：指向已接受的 PPP 数据缓冲区的指针。
- Packet_id：数据包标识符。

返回值

无

允许来自

初始化、线程

示例

```
UINT packet_id = 0x20015429

/* Allow a preceding PPP data to be acknowledged, let PPPoE Server release the packet with same packet
identifier. */
PppTransmitDataCnf(0, NX_NULL, packet_id);
```

PppReceiveDataInd

接收通过以太网传输的数据

原型

```
VOID PppReceiveDataInd(UINT interfaceHandle, UINT length, UCHAR *aData);
```

说明

PPPoE 软件将公开此函数，以接收通过以太网传输的数据

输入参数

- interfaceHandle：接口句柄。
- length：aData 中的字节数。
- aData：包含 PPP 数据帧的数据缓冲区。

返回值

无

允许来自

初始化、线程

示例

```
/* Receive data from transmission over Ethernet, data start pointer is aData.
The number of bytes in aData is 1480. */
PppReceiveDataInd (0, 1480, aData);
```