Contents

Azure RTOS USBX 概述

2021/4/29 •

Azure RTOS USBX 是一种高性能的 USB 主机、设备和移动 (OTG) 嵌入式堆栈。Azure RTOS USBX 与 Azure RTOS ThreadX 完全集成，适用于所有支持 Azure RTOS ThreadX 的处理器。与 ThreadX 一样，Azure RTOS USBX 也采用占用空间小、性能高的设计，特别适用于需要与 USB 设备对接的深度嵌入式应用程序。

主机、设备、OTG 和广泛类支持

Azure RTOS USBX 主机/设备嵌入式 USB 协议堆栈是工业级的嵌入式 USB 解决方案，专门设计用于深度嵌入式应用程序、实时应用程序和 IoT 应用程序。Azure RTOS USBX 提供主机、设备和 OTG 支持以及广泛类支持。Azure RTOS USBX 与 ThreadX 实时操作系统、Azure RTOS FileX 嵌入式 FAT 兼容文件系统、Azure RTOS NetX 和 Azure RTOS NetX Duo 嵌入式 TCP/IP 堆栈完全集成。凭借所有这些以及占用空间极小、执行速度快、易于使用的优势，Azure RTOS USBX 已成为需要建立 USB 连接的、要求最高的嵌入式 IoT 应用程序的理想选择。

占用空间小

Azure RTOS USBX 的占用空间极小，它只需占用 10.5 KB 闪存和 5.1 KB RAM 来提供 Azure RTOS USBX 设备 CDC/ACM 支持。要提供 Azure RTOS USBX 主机 CDC/ACM 支持，它至少需要占用 18 KB 闪存和 25 KB RAM。

TCP 功能需要额外的 10 KB 到 13 KB 的指令区域内存。Azure RTOS USBX 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 ThreadX 一样，Azure RTOS USBX 的大小会根据应用程序实际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。

快速执行

Azure RTOS USBX 采用高速设计，提供极其精简的内部函数调用分层，并支持使用缓存和 DMA。所有这些以及面向性能的总体设计理念都有助于 Azure RTOS USBX 实现尽可能快的性能。

简单易用

Azure RTOS USBX 易于使用。Azure RTOS USBX API 既直观又功能强大。API 名称由实词组成，而不是其他文件系统产品中常见的“字母汤”或高度缩略的名称。所有 Azure RTOS USBX API 均具有前导“ux_”，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 均具有可选的超时期限，其功能对所有 API 都一致。

USB 互操作性验证

已使用 USB IF 标准测试工具 USBCV 对 Azure RTOS USBX 设备堆栈进行严格的测试，以确保完全符合 USB 规范以及与不同主机系统的互操作性。此外，Azure RTOS USBX OTG 堆栈已由位于中国台湾的独立测试实验室 Allion 进行验证和认证。

USB 主机控制器支持

Azure RTOS USBX 支持主要的 USB 标准，例如 OHCI 和 EHCI。此外，Azure RTOS USBX 支持 Atmel、Microchip、Philips、Renesas、ST、TI 和其他供应商提供的专有分立 USB 主机控制器。Azure RTOS USBX 还支持同一应用程序中的多个主机控制器。USB 设备控制器支持：Azure RTOS USBX 支持 Analog Devices、Atmel、Microchip、NXP、Philips、Renesas、ST、TI 和其他供应商提供的流行 USB 设备控制器。

广泛主机类支持

Azure RTOS USBX 主机支持大多数流行类，包括 ASIX、AUDIO、CDC/ACM、CDC/ECM、GSER、HID（键盘、鼠标和远程控制）、HUB、PIMA (PTP/MTP)、PRINTER、PROLIFIC 和 STORAGE。

广泛 USB 设备类支持

Azure RTOS USBX 设备支持大多数流行类，包括 CDC/ACM、CDC/ECM、DFU、HID、PIMA (PTP/MTP) (w/MTP)、RNDIS 和 STORAGE。此外还支持自定义类。

Pictbridge 支持

Azure RTOS USBX 在主机和设备上都支持完全 Pictbridge 实现。Pictbridge 位于两端上的 Azure RTOS USBX PIMA (PTP/MTP) 类的顶层。PictBridge 标准允许将数码照相机或智能手机直接连接到打印机，而不使用 PC，从而可以直接使用特定的 Pictbridge 感知打印机进行打印。当相机或手机连接到打印机时，打印机即为 USB 主机，照相机即为 USB 设备。然而，在使用 Pictbridge 时，相机显示为主机，而且命令是从相机驱动的。相机是存储服务器，打印机是存储客户端。相机是打印客户端，打印机当然是打印服务器。Pictbridge 使用 USB 作为传输层，但依赖于 PTP（图片传输协议）作为通信协议。

自定义类支持

Azure RTOS USBX 主机和设备支持自定义类。Azure RTOS USBX 分发包中提供了一个示例自定义类。此简单数据抽取类名为 DPUMP，可用作自定义应用程序类的模型。技术先进的 Azure RTOS USBX 主机和设备支持自定义类。Azure RTOS USBX 分发包中提供了一个示例自定义类。Azure RTOS USBX 属于先进技术，其中包括：

- 主机、设备和 OTG 支持
- USB 低速、全速和高速支持
- 自动缩放
- 与 ThreadX、Azure RTOS FileX 和 Azure RTOS NetX 完全集成
- 可选性能指标
- Azure RTOS TraceX 系统分析支持

最快面市时间

Azure RTOS USBX 占用空间极小，只需 9 KB 到 15 KB 即可提供基本的 IP 和 UDP 支持。Azure RTOS USBX 易于安装、学习、使用、调试、验证、认证和维护。因此，Azure RTOS USBX 是适用于嵌入式 IoT 设备的最流行 USB 解决方案之一。我们一贯的面市时间优势建立在以下基础之上：

- 优质文档 – 请查看我们的 Azure RTOS USBX 主机和设备用户指南，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，它还约定每个文件具有一个功能，正因如此，你可以轻松在源代码中导航。

支持最流行的体系结构

Azure RTOS USBX 可以直接在大多数主流 32 位/64 位微处理器上运行，已经过全面测试且完全受支持。这些微处理器包括：

- 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 和 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

Azure RTOS USBX API

Azure RTOS USBX 主机 API

Azure RTOS USBX 主机 API 是直观且一致的 API，遵循名词-动词命名约定。所有 API 带有前导 ux_host_*，可以轻松识别出它们是 USBX API。所有阻塞 API 具有可选的线程超时。

- ASIX
  - 占用空间极小，只需 0.3 KB 闪存，4 KB RAM
  - 自动缩放；通过 Azure RTOS TraceX 进行系统级跟踪
  - 以下形式的直观 Azure RTOS USBX 主机 API：ux_host_class_asix_*
- 音频
  - 占用空间极小，只需 1.2 KB 闪存，4 KB RAM
  - 自动缩放
  - 以下形式的直观 Azure RTOS USBX 主机 API：ux_host_class_audio_*
- CDC/ACM
  - 占用空间极小，只需 1.4 KB 闪存，4 KB RAM
  - 自动缩放
  - 通过 Azure RTOS TraceX 进行系统级跟踪
  - 以下形式的直观 Azure RTOS USBX 主机 API：ux_host_class_cdc_acm_*
- HID
  - 占用空间极小，只需 0.3 KB 闪存，4 KB RAM
  - 键盘、鼠标和远程支持
  - 自动缩放
  - 通过 Azure RTOS TraceX 进行系统级跟踪
  - 以下形式的直观 Azure RTOS USBX 主机 API：ux_host_class_hid_*
- HUB
  - 占用空间极小，只需 1.7 KB 闪存，2 KB RAM
  - 自动缩放
  - 通过 Azure RTOS TraceX 进行系统级跟踪
  - 以下形式的直观 Azure RTOS USBX 主机 API：ux_host_class_hub_*
- PIMA (PTP/MTP)
  - 占用空间极小，只需 0.9 KB 闪存，8 KB RAM
  - 自动缩放

    - 通过 Azure RTOS TraceX 进行系统级跟踪
    - 以下形式的直观 Azure RTOS USBX 主机 API：ux_host_class_pima_*
- PRINTER
    - 占用空间极小，只需 0.8 KB 闪存，8 KB RAM
    - 自动缩放
    - 通过 Azure RTOS TraceX 进行系统级跟踪
    - 以下形式的直观 Azure RTOS USBX 主机 API：ux_host_class_printer_*
- PROLIFIC
    - 占用空间极小，只需 1.5 KB 闪存，4 KB RAM
    - 自动缩放
    - 通过 Azure RTOS TraceX 进行系统级跟踪
    - 以下形式的直观 Azure RTOS USBX 主机 API：ux_host_class_prolific_*
- STORAG
    - 占用空间极小，只需 5.6 KB 闪存，4 KB RAM
    - 自动缩放
      与 Azure RTOS FileX 集成
    - 通过 Azure RTOS TraceX 进行系统级跟踪
    - 以下形式的直观 Azure RTOS USBX 主机 API：ux_host_class_storage_*
- USB 主机堆栈
    - 支持许多主机控制器
    - 占用空间极小，只需 18 KB 闪存，25 KB RAM
    - 自动缩放
    - 支持同一平台上的多个主机控制器
    - USB 低速、全速和高速支持
    - 通过 Azure RTOS TraceX 进行系统级跟踪
    - 以下形式的直观 Azure RTOS USBX 主机 API：ux_host_stack_ *
- OHCI、EHCI、专有主机控制器

Azure RTOS USBX 设备 API

Azure RTOS USBX 设备 API 是直观且一致的 API，遵循名词-动词命名约定。所有 API 带有前导 ux_device_*，可以轻松识别出它们是 USBX API。阻塞 API 具有可选的线程超时。有关更多详细信息，请参阅 Azure RTOS USBX 主机用户指南。

- CDC/ACM
    - 占用空间极小，只需 0.8 KB 闪存，2 KB RAM
    - 自动缩放
    - 通过 Azure RTOS TraceX 进行系统级跟踪
    - 以下形式的直观 Azure RTOS USBX 设备 API：*ux_device_class_cdc_acm_**。
- CDC/ECM
    - 占用空间极小，只需 1.5 KB 闪存，4 KB 到 8 KB RAM
    - 自动缩放
    - 通过 Azure RTOS TraceX 进行系统级跟踪
      以下形式的直观 Azure RTOS USBX 设备 API：*ux_device_class_cdc_ecm_**。
- DFU
    - 占用空间极小，只需 1.1 KB 闪存，2 KB RAM
    - 自动缩放
    - 通过 Azure RTOS TraceX 进行系统级跟踪
    - 以下形式的直观 Azure RTOS USBX 设备 API：ux_device_class_dfu_*

- GSER
  - 占用空间极小，只需 0.6 KB 闪存，4 KB RAM
  - 自动缩放
  - 通过 Azure RTOS TraceX 进行系统级跟踪
  - 以下形式的直观 Azure RTOS USBX 设备 API：ux_device_class_gser_*
- HID
  - 占用空间极小，只需 0.9 KB 闪存，2 KB RAM
  - 自动缩放
  - 通过 Azure RTOS TraceX 进行系统级跟踪
  - 以下形式的直观 Azure RTOS USBX 设备 API：ux_device_class_hid_* PIMA (PTP/MTP)
  - 占用空间极小，只需 5.2 KB 闪存，8 KB RAM
  - 自动缩放
  - 通过 Azure RTOS TraceX 进行系统级跟踪
  - 以下形式的直观 Azure RTOS USBX 设备 API：ux_device_class_pima_*
- STORAGE
  - 占用空间极小，只需 2.3 KB 闪存，4 KB RAM
  - 自动缩放
  - 通过 Azure RTOS TraceX 进行系统级跟踪
  - 以下形式的直观 Azure RTOS USBX 设备 API：ux_device_class_storage_*
- RNDIS
  - 占用空间极小，只需 2.3 KB 闪存，4 KB 到 8 KB RAM
  - 自动缩放
  - 与 Azure RTOS NetX 和 Azure RTOS NetX DUO 集成
  - 通过 Azure RTOS TraceX 进行系统级跟踪
  - 以下形式的直观 Azure RTOS USBX 设备 API：ux_device_class_rndls_*
- Azure RTOS USBX 设备堆栈
  - 占用空间极小，只需 2.3 KB 闪存，4 KB RAM
  - 自动缩放
  - 通过 Azure RTOS TraceX 进行系统级跟踪
  - 以下形式的直观 Azure RTOS USBX 设备 API：ux_device_class_storage_*
- 专有主机控制器

后续步骤

遵循我们的主机堆栈用户指南或设备堆栈用户指南开始使用 Azure RTOS USBX 主机和设备堆栈。

Azure RTOS USBX 设备堆栈用户指南

2021/4/29 •

本指南旨在全面介绍 Azure RTOS USBX（Microsoft 的高性能 USB 基础软件）。

本指南适用对象为嵌入式实时软件的开发人员。开发人员应熟悉标准实时操作系统函数、USB 规范和 C 编程语言。

有关与 USB 相关的技术信息，请参阅 USB 规范和 USB 类规范，这些规范可于 https://www.USB.org/developers 下载

组织

- **第 1 章** - Azure RTOS USBX 简介
- **第 2 章** - 介绍在 ThreadX 应用程序中安装和使用 Azure RTOS USBX 的基本步骤
- **第 3 章** - 介绍 Azure RTOS USBX 设备堆栈的功能组件
- **第 4 章** - 介绍 Azure RTOS USBX 设备堆栈服务
- **第 5 章** - 介绍每个 Azure RTOS USBX 设备类（包括它们的 API）

客户支持中心

请使用此处介绍的步骤，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. 详细说明发生问题前对应用程序和/或 Azure RTOS ThreadX 所作的任何更改。
3. 可在分发的 tx_port.h 文件中找到的 _tx_version_id 字符串的内容。此字符串将为我们提供有关运行时环境的重要信息。
4. RAM 中 _tx_build_options ULONG 变量的内容。此变量将为我们提供有关 Azure RTOS ThreadX 库生成方式的信息。

第 1 章 - Azure RTOS USBX 设备堆栈简介

2021/4/30 •

USBX 是适用于深度嵌入应用程序的全功能 USB 堆栈。本章引入了 USBX，介绍其应用程序和优点

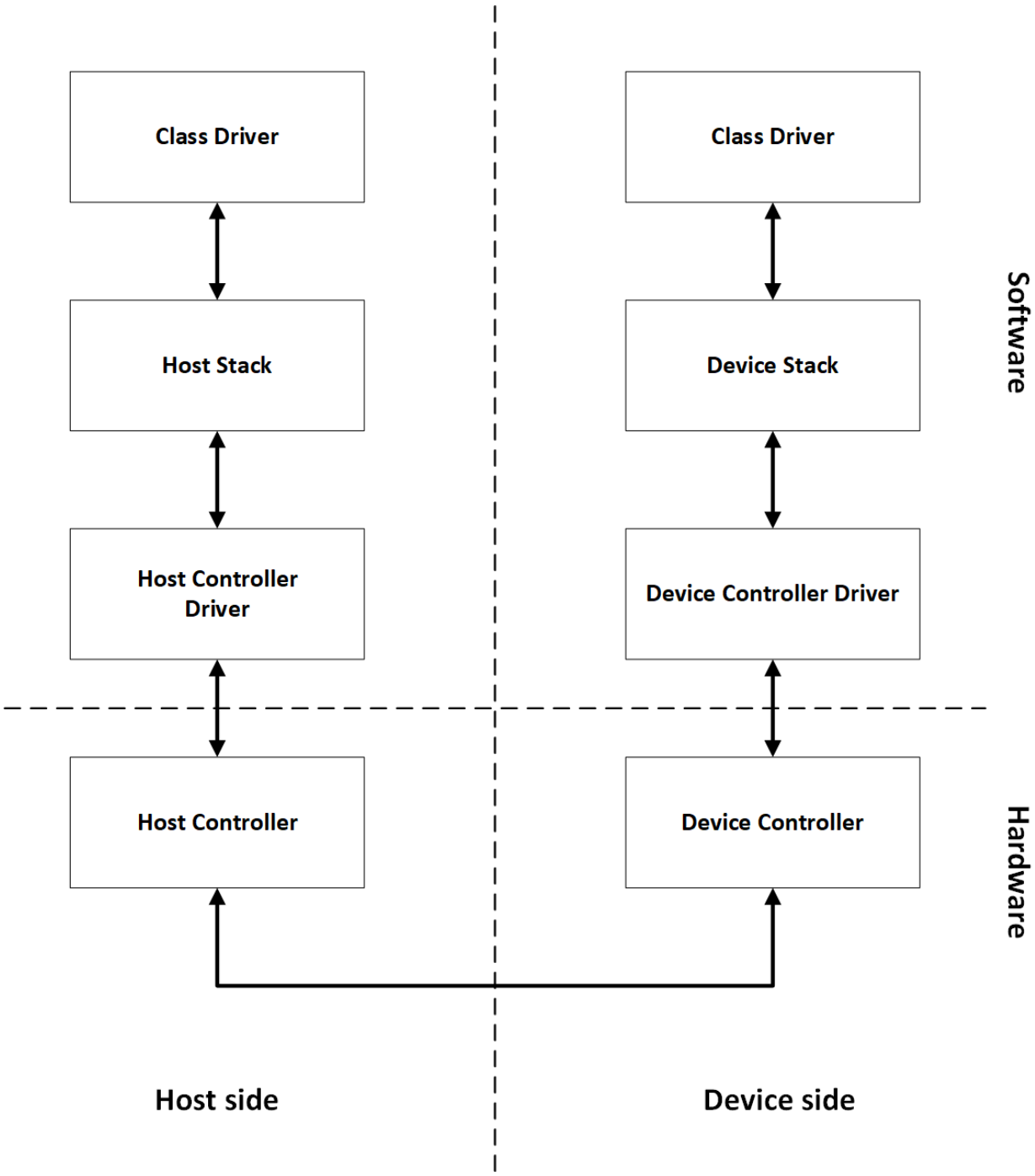
USBX 功能

USBX 支持三个现有的 USB 规范：1.1、2.0 和 OTG。它设计为能够缩放，适用于只有一个已连接设备的简单 USB 拓扑，以及具有多个设备和级联中心的复杂拓扑。USBX 支持 USB 协议的所有数据传输类型：控制、批量、中断和常时等量。

USBX 同时支持主机端和设备端。每一端都由三个层组成。

- 控制器层
- 堆栈层
- 类层

各个 USB 层之间的关系如下所示：



产品亮点

- 完整的 ThreadX 处理器支持
- 无版税
- 完整的 ANSI C 源代码
- 实时性能
- 快速响应的技术支持
- 多类支持
- 多类实例
- 各个类与 ThreadX、FileX 和 NetX 集成
- 支持具有多个配置的 USB 设备
- 支持 USB 复合设备
- 支持 USB 电源管理
- 支持 USB OTG

- 导出 TraceX 的跟踪事件

强大的 USBX 服务

完整的 USB 设备框架支持

USBX 可以支持要求最苛刻的 USB 设备，包括多个配置、多个接口和多个备用设置。

易于使用的 API

USBX 以一种易于理解和使用的方式提供了最佳的深度嵌入 USB 堆栈。USBX API 使服务直观且一致。通过使用提供的 USBX 类 API，用户应用程序无需了解 USB 协议的复杂性。

第 2 章 - Azure RTOS USBX 设备堆栈安装

2021/5/1 •

主机注意事项

计算机类型

嵌入式开发通常在 Windows PC 或 Unix 主机计算机上执行。在对应用程序进行编译和链接并将其放置在主机上之后，应用程序将下载到目标硬件进行执行。

下载接口

虽然并行接口、USB 接口和以太网接口变得越来越普遍，但目标下载通常通过 RS-232 串行接口进行。有关可用选项，请参阅开发工具文档。

调试工具

通常通过与程序映像下载相同的链接进行调试。存在多种调试器，包括在目标上运行的小型监视器程序、后台调试监视器 (BDM) 和在线仿真器 (ICE) 工具等。ICE 工具提供最可靠的实际目标硬件调试。

所需的硬盘空间

USBX 的源代码以 ASCII 格式提供，并要求主计算机的硬盘具有约 500 KB 可用空间。

目标注意事项

USBX 要求处于主机模式的目标具有 24 KB 到 64 KB 只读内存 (ROM)。所需的内存量取决于所使用的控制器类型和链接到 USBX 的 USB 类。USBX 全局数据结构和内存池要求目标具有额外的 32 KB 随机存取内存 (RAM)。还可以根据 USB 接口上预期连接的设备数和 USB 控制器的类型调整此内存池。USBX 设备端需要大约 10 K 到 12 K ROM，具体取决于设备控制器的类型。RAM 内存使用量取决于设备仿真的类的类型。

USBX 还依赖于 ThreadX 信号灯、互斥体和线程进行多线程保护、I/O 暂停和定期处理，以监视 USB 总线拓扑。

产品分发

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

下面列出了该存储库中的几个重要文件。

- ux_api.h：此 C 头文件包含所有系统等式、数据结构和服务原型。
- ux_port.h：此 C 头文件包含所有特定于开发工具的数据定义和结构。
- ux.lib：这是 USBX C 库的二进制版本，它随标准包一起分发。
- demo_usbx.c：包含简单 USBX 演示的 C 文件

所有文件名均为小写。此命名约定使你可以更轻松地将命令转换为 Unix 开发平台命令。

USBX 安装

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

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

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。

你还可以在联机存储库的首页上找到有关生成 USBX 库的说明。

以下一般说明适用于几乎所有安装：

1. 使用之前在主机硬盘驱动器上安装 ThreadX 的同一目录。所有 USBX 名称都是唯一的，不会干扰以前的 USBX 安装。
2. 在 tx_application_define 的开头或附近添加对 ux_system_initialize 的调用。这是初始化 USBX 资源的位置。
3. 添加对 ux_host_stack_initialize 的调用。
4. 添加一个或多个调用，以初始化所需的 USBX 类（主机和/或设备类）
5. 添加一个或多个调用，以初始化系统中可用的设备控制器。
6. 可能需要修改 tx_low_level_initialize.c 文件，以添加低级别硬件初始化并中断矢量路由。此说明特定于硬件平台，本文不做讨论。|
7. 编译应用程序源代码并将其与 USBX 运行时库和 ThreadX 运行时库（如果要编译 USB 存储类和/或 USB 网络类，则可能还需要 FileX 和/或 Netx）、ux.a（或 ux.lib）以及 tx.a（或 tx.lib）链接起来。生成的结果可下载到目标并执行！

配置选项

有多个配置选项用于生成 USBX 库。所有选项都位于 ux_user.h 中。

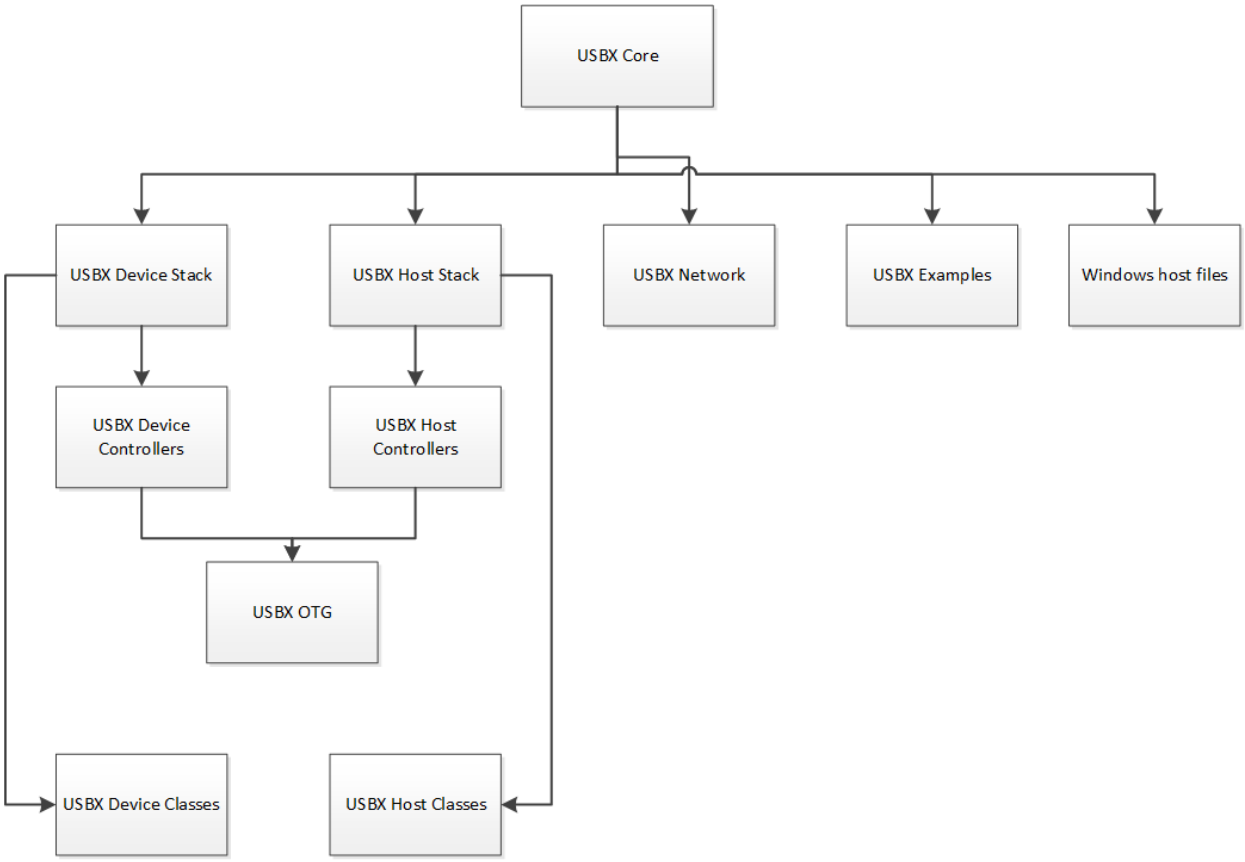
以下列表详细介绍了每个配置选项。

‘‘‘‘	‘‘
UX_PERIODIC_RATE	此值表示特定硬件平台每秒的时钟周期数。默认值为 1000，表示每毫秒 1 个时钟周期。
UX_THREAD_STACK_SIZE	此值为 USBX 线程的堆栈大小（以字节为单位）。它通常可以是 1024 字节或 2048 字节，具体取决于所用的处理器和主机控制器。
UX_THREAD_PRIORITY_ENUM	这是用于监视总线拓扑的 USBX 枚举线程的 ThreadX 优先级值。
UX_THREAD_PRIORITY_CLASS	这是标准 USBX 线程的 ThreadX 优先级值。
UX_THREAD_PRIORITY_KEYBOARD	这是 USBX HID 键盘类的 ThreadX 优先级值。
UX_THREAD_PRIORITY_DCD	这是设备控制器线程的 ThreadX 优先级值。
UX_NO_TIME_SLICE	此值实际上定义将用于线程的时间片。例如，如果定义为 0，则 ThreadX 目标端口不使用时间片。
UX_MAX_SLAVE_CLASS_DRIVER	这是可通过 ux_device_stack_class_register 注册的 USBX 类的最大数目。
UX_MAX_SLAVE_LUN	此值表示设备存储类驱动程序中表示的当前 SCSI 逻辑单元数。
UX_SLAVE_CLASS_STORAGE_INCLUDE_MMC	如果已定义，则存储类将处理多媒体命令 (MMC)，即 DVD-ROM 命令。
UX_DEVICE_CLASS_CDC_ECM_NX_PKPOOL_ENTRIES	此值表示 CDC-ECM 类的数据包池中的 NetX 数据包数。默认值为 16。

UX_SLAVE_REQUEST_CONTROL_MAX_LENGTH	此值表示在设备堆栈中的控制终结点上收到的最大字节数。 默认值为 256 字节，但内存受限的环境中可以减少。
UX_DEVICE_CLASS_HID_EVENT_BUFFER_LENGTH	此值表示 HID 报告的最大长度（以字节为单位）。
UX_DEVICE_CLASS_HID_MAX_EVENTS_QUEUE	此值表示一次可以排队的最大 HID 报告数。
UX_SLAVE_REQUEST_DATA_MAX_LENGTH	此值表示在设备堆栈中的大量终结点上收到的最大字节数。 默认值为 4096 字节，但内存受限的环境中可以减少。

源代码树

USBX 文件在多个目录中提供。



为了使文件可通过其名称识别，已采用以下约定：

ux_host_stack	usbx 主机堆栈核心文件
ux_host_class	usbx 主机堆栈类文件
ux_hcd	usbx 主机堆栈控制器驱动程序文件
ux_device_stack	usbx 设备堆栈核心文件
ux_device_class	usbx 设备堆栈类文件

ux_dcd	usbx 设备堆栈控制器驱动程序文件
ux_otg	usbx otg 控制器驱动程序相关文件
ux_pictbridge	usbx pictbridge 文件
ux_utility	usbx 实用工具函数
demo_usbx	USBX 的演示文件

USBX 资源的初始化

USBX 有自己的内存管理器。在初始化 USBX 的主机或设备端之前，需要将内存分配给 USBX。USBX 内存管理器可以容纳可缓存内存的系统。

以下函数可将 USBX 内存资源初始化为具有 128 K 常规内存且没有用于缓存安全内存的单独池：

```
/* Initialize USBX Memory */
ux_system_initialize(memory_pointer,(128*1024),UX_NULL,0);
```

ux_system_initialize 的原型如下所示：

```
UINT ux_system_initialize(VOID *regular_memory_pool_start,
        ULONG regular_memory_size,
        VOID *cache_safe_memory_pool_start,
        ULONG cache_safe_memory_size);
```

输入参数：

VOID *regular_memory_pool_start	常规内存池的开头
ULONG regular_memory_size	常规内存池的大小
VOID *cache_safe_memory_pool_start	缓存安全内存池的开头
ULONG cache_safe_memory_size	缓存安全内存池的大小

并非所有系统都需要定义缓存安全内存。在此类系统中，在初始化过程中为内存指针传递的值将设置为 UX_NULL，并且池的大小会设置为 0。然后，USBX 将使用常规内存池来代替缓存安全池。

在常规内存不是缓存安全内存的系统中，如果控制器需要执行 DMA 内存，则需要在缓存安全区域中定义一个内存池。

USBX 资源的取消初始化

可以通过释放 USBX 的资源来将其终止。在终止 USBX 之前，需要正确终止所有类和控制器资源。以下函数将取消初始化 USBX 内存资源：

```
/* Unitialize USBX Resources */

ux_system_uninitialize();
```

ux_system_initialize 的原型如下所示：

```
UINT ux_system_uninitialize(VOID);
```

USB 设备控制器的定义

在任何时候，只能定义一个以设备模式运行的 USB 设备控制器。应用程序初始化文件应包含此定义。以下行将定义通用 USB 控制器：

```
ux_dcd_controller_initialize(0x7BB00000, 0, 0xB7A00000);
```

USB 设备初始化具有以下原型：

```
UINT ux_dcd_controller_initialize(ULONG dcd_io,
    ULONG dcd_irq, ULONG dcd_vbus_address);
```

使用以下参数：

“	“
ULONG dcd_io	控制器 IO 的地址
ULONG dcd_irq	控制器使用的中断
ULONG dcd_vbus_address	VBUS GPIO 的地址

以下示例演示如何使用存储设备类和通用控制器初始化处于设备模式的 USBX：

```
/* Initialize USBX Memory */

ux_system_initialize(memory_pointer,(128*1024), 0, 0);

/* The code below is required for installing the device portion of USBX */
status = ux_device_stack_initialize(&device_framework_high_speed,
    DEVICE_FRAMEWORK_LENGTH_HIGH_SPEED, &device_framework_full_speed,
    DEVICE_FRAMEWORK_LENGTH_FULL_SPEED, &string_framework,
    STRING_FRAMEWORK_LENGTH, &language_id_framework,
    LANGUAGE_ID_FRAMEWORK_LENGTH, UX_NULL);

/* If status equals UX_SUCCESS, installation was successful. */

/* Store the number of LUN in this device storage instance: single LUN. */
storage_parameter.ux_slave_class_storage_parameter_number_lun = 1;

/* Initialize the storage class parameters for reading/writing to the Flash Disk. */
storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_last_lba = 0x1e6bfe;
storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_block_length = 512;
storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_type = 0;
storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_removable_flag =
0x80;
storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_read =
tx_demo_thread_flash_media_read;
storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_write =
tx_demo_thread_flash_media_write;
storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_status =
tx_demo_thread_flash_media_status;

/* Initialize the device storage class. The class is connected with interface 0 */
status = ux_device_stack_class_register(ux_system_slave_class_storage_name ux_device_class_storage_entry,
    ux_device_class_storage_thread,0, (VOID *)&storage_parameter);

/* Register the device controllers available in this system */
status = ux_dcd_controller_initialize(0x7BB00000, 0, 0xB7A00000);

/* If status equals UX_SUCCESS, registration was successful. */
```

疑难解答

USBX 附带演示文件和模拟环境。最好先让演示平台在目标硬件或特定演示平台上运行。

USBX 版本 ID

当前版本的 USBX 在运行时可供用户和应用程序软件使用。程序员可以通过检查 ux_port.h 文件来获取 USBX 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检查全局字符串 *ux_version_id（在 ux_port.h 中定义）来获取 USBX 版本*。

第 3 章 - USBX 设备堆栈的功能组件

2021/4/29 •

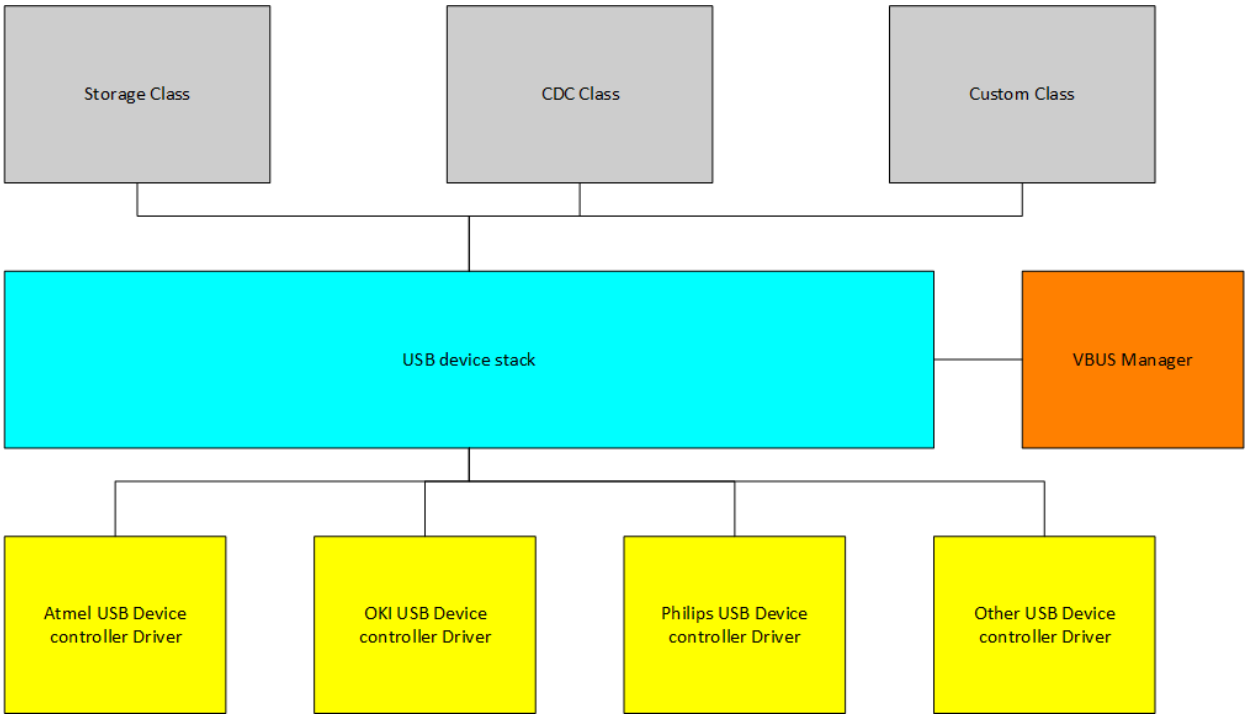
本章从功能角度介绍了高性能 USBX 嵌入式 USB 设备堆栈。

执行概述

设备的 USBX 由多个组件组成。

- 初始化
- 应用程序接口调用
- 设备类
- USB 设备堆栈
- 设备控制器
- VBUS 管理器

下图展示了 USBX 设备堆栈。



初始化

若要激活 USBX，必须调用函数 ux_system_initialize。此函数初始化 USBX 的内存资源。

若要激活 USBX 设备功能，必须调用函数 ux_device_stack_initialize。此函数反过来会对 USBX 设备堆栈使用的所有资源（如 ThreadX 线程、互斥和信号灯）进行初始化。

应该通过应用程序初始化来激活 USB 设备控制器和一个或多个 USB 类。与 USB 主机端相反，设备端每次只能运行一个 USB 控制器驱动程序。当类已注册到堆栈并且设备控制器初始化函数已被调用时，总线将处于活动状态，堆栈会响应总线重置和主机枚举命令。

应用程序接口调用

USBX 中有两个级别的 API。

- USB 设备堆栈 API

- USB 设备类 API

通常，USBX 应用程序不必调用任何 USB 设备堆栈 API。大多数应用程序将只访问 USB 类 API。

USB 设备堆栈 API

设备堆栈 API 负责注册 USBX 设备组件（例如类和设备框架）。

USB 设备类 API

类 API 严格特定于每个 USB 类。USB 类的大多数常见 API 提供了诸如打开/关闭设备以及从设备读取和向设备写入之类的服务。这些 API 在本质上类似于主机端。

设备框架

USB 设备端负责设备框架的定义。设备框架分为三个类别，如以下部分所述。

设备框架的组件的定义

设备框架的每个组件的定义与设备和设备使用的资源的性质有关。下面是主要类别。

- 设备描述符
- 配置描述符
- 接口描述符
- 终结点描述符

USBX 支持高速和全速的设备组件定义（对低速的处理方式与全速相同）。这允许设备在连接到高速或全速主机时以不同的方式运行。典型区别在于每个终结点的大小和设备消耗的电量。

设备组件的定义采用遵循 USB 规范的字节字符串形式。定义是连续的，框架在内存中的呈现顺序将与枚举期间返回到主机的顺序相同。

下面是高速 USB 闪存磁盘的设备框架示例。

```
#define DEVICE_FRAMEWORK_LENGTH_HIGH_SPEED 60
UCHAR device_framework_high_speed[] = {
    /* Device descriptor */
    0x12, 0x01, 0x00, 0x02, 0x00, 0x00, 0x00, 0x40, 0x0a, 0x07, 0x25, 0x40, 0x01, 0x00, 0x01, 0x02, 0x03,
0x01,

    /* Device qualifier descriptor */
    0x0a, 0x06, 0x00, 0x02, 0x00, 0x00, 0x00, 0x40, 0x01, 0x00,

    /* Configuration descriptor */
    0x09, 0x02, 0x20, 0x00, 0x01, 0x01, 0x00, 0xc0, 0x32,

    /* Interface descriptor */
    0x09, 0x04, 0x00, 0x00, 0x02, 0x08, 0x06, 0x50, 0x00,

    /* Endpoint descriptor (Bulk Out) */
    0x07, 0x05, 0x01, 0x02, 0x00, 0x02, 0x00,

    /* Endpoint descriptor (Bulk In) */
    0x07, 0x05, 0x82, 0x02, 0x00, 0x02, 0x00
};
```

设备框架的字符串的定义

字符串在设备中是可选的。它们的用途是通过 Unicode 字符串让 USB 主机知道设备的制造商、产品名称和修订号。

主字符串是设备描述符中嵌入的索引。其他字符串索引可以嵌入到各个接口中。

假设上面的设备框架在设备描述符中嵌入了三个字符串索引，则字符串框架定义可能类似于下面的示例代码。

```
/* String Device Framework:
    Byte 0 and 1: Word containing the language ID: 0x0904 for US
    Byte 2 : Byte containing the index of the descriptor
    Byte 3 : Byte containing the length of the descriptor string
*/

#define STRING_FRAMEWORK_LENGTH 38
UCHAR string_framework[] = {
    /* Manufacturer string descriptor: Index 1 */
    0x09, 0x04, 0x01, 0x0c,
    0x45, 0x78, 0x70, 0x72, 0x65, 0x73, 0x20, 0x4c,
    0x6f, 0x67, 0x69, 0x63,

    /* Product string descriptor: Index 2 */
    0x09, 0x04, 0x02, 0x0c,
    0x4D, 0x4C, 0x36, 0x39, 0x36, 0x35, 0x30, 0x30,
    0x20, 0x53, 0x44, 0x4B,

    /* Serial Number string descriptor: Index 3 */
    0x09, 0x04, 0x03, 0x04,
    0x30, 0x30, 0x30, 0x31
};
```

如果必须为每种速度使用不同的字符串，则必须使用不同的索引，因为索引与速度无关。

字符串的编码基于 UNICODE。有关 UNICODE 编码标准的详细信息，请参阅以下出版物：

Unicode 标准，全球字符编码版本 1 第 1 卷和第 2 卷，Unicode 联合会，Addison-Wesley Publishing Company，Reading MA。

设备针对每个字符串支持的语言的定义

USBX 能够支持多种语言，但英语是默认语言。字符串描述符的每种语言的定义采用语言定义数组形式，定义如下。

```
#define LANGUAGE_ID_FRAMEWORK_LENGTH 2
UCHAR language_id_framework[] = {
    /* English. */
    0x09, 0x04
};
```

若要支持更多语言，只需在默认英语代码之后添加语言代码双字节定义。Microsoft 已在文档中定义了语言代码。

针对 Windows 95 和 Windows NT 开发国际软件：Nadine Kano，Microsoft Press，Redmond WA

VBUS 管理器

在大多数 USB 设备设计中，VBUS 不属于 USB 设备核心，而是连接到用于监视线路信号的外部 GPIO。

因此，VBUS 必须与设备控制器驱动程序分开管理。

由应用程序向设备控制器提供 VBUS IO 的地址。在设备控制器初始化之前，必须对 VBUS 进行初始化。

根据监视 VBUS 的平台规范，可以在 VBUS IO 初始化后让控制器驱动程序处理 VBUS 信号。如果这不可能，则应用程序必须提供用于处理 VBUS 的代码。

如果应用程序希望自行处理 VBUS，则它唯一的要求是在检测到设备已被提取时调用函数 ux_device_stack_disconnect。在插入设备时无需通知控制器，因为当检测到总线复位断言/解除断言信号时，控制器会被唤醒。

USBX 设备服务的说明

2021/4/29 •

ux_device_stack_alternate_setting_get

获取接口值的当前备用设置

原型

```
UINT ux_device_stack_alternate_setting_get(ULONG interface_value);
```

说明

USB 主机使用此函数获取特定接口值的当前备用设置。收到 GET_INTERFACE 请求时，控制器驱动程序将调用此函数。

输入参数

- Interface_value：要查询其当前备用设置的接口值

返回值

- UX_SUCCESS：(0x00) 已完成数据传输。
- UX_ERROR：(0xFF) 错误的接口值。

示例

```
ULONG interface_value;
UINT status;

/* The following example illustrates this service. */
status = ux_device_stack_alternate_setting_get(interface_value);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_alternate_setting_set

设置接口值的当前备用设置

原型

```
UINT ux_device_stack_alternate_setting_set(
    ULONG interface_value,
    ULONG alternate_setting_value);
```

说明

USB 主机使用此函数设置特定接口值的当前备用设置。收到 SET_INTERFACE 请求时，控制器驱动程序将调用此函数。完成 SET_INTERFACE 后，会将备用设置的值应用于类。

设备堆栈将向拥有此接口的类发出 UX_SLAVE_CLASS_COMMAND_CHANGE，以反映备用设置的更改。

参数

- Interface_value：要设置其当前备用设置的接口值。
- alternate_setting_value：新的备用设置值。

返回值

- UX_SUCCESS：(0x00) 已完成数据传输。
- UX_INTERFACE_HANDLE_UNKNOWN：(0x52) 未附加接口。
- UX_FUNCTION_NOT_SUPPORTED：(0x54) 未配置设备。
- UX_ERROR：(0xFF) 错误的接口值。

示例

```
ULONG interface_value;

ULONG alternate_setting_value;

/* The following example illustrates this service. */
status = ux_device_stack_alternate_setting_set(interface_value, alternate_setting_value);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_class_register

注册新的 USB 设备类

原型

```
UINT ux_device_stack_class_register(
    UCHAR *class_name,
    UINT (*class_entry_function)(struct UX_SLAVE_CLASS_COMMAND_STRUCT *),
    ULONG configuration_number,
    ULONG interface_number,
    VOID *parameter);
```

说明

应用程序使用此函数注册新的 USB 设备类。此注册将启动类容器，而不是启动类的实例。类应具有活动线程，并且应附加到特定的接口。

某些类需要一个参数或参数列表。例如，设备存储类需要它正在尝试仿真的存储设备的几何结构。因此，参数字段依赖于类要求，可以是某个值，或者是指向填充了类值的结构的指针。

> **NOTE**
> class_name 的 C 字符串必须以 NULL 结尾，其长度（不包括 NULL 终止符本身）不能大于 UX_MAX_CLASS_NAME_LENGTH。

参数

- class_name：类名
- class_entry_function：类的入口函数。
- configuration_number：此类附加到的配置编号。
- interface_number：此类附加到的接口编号。
- parameter：指向类特定的参数列表的指针。

返回值

- UX_SUCCESS：(0x00) 类已注册
- UX_MEMORY_INSUFFICIENT：(0x12) 类表中未保留条目。
- UX_THREAD_ERROR：(0x16) 无法创建类线程。

示例

```
UINT status;

/* The following example illustrates this service. */

/* Initialize the device storage class. The class is connected with interface 1 */
status = ux_device_stack_class_register(_ux_system_slave_class_storage_name ux_device_class_storage_entry,
    1, 1, (VOID *)&parameter);
```

ux_device_stack_class_unregister

注销 USB 设备类

原型

```
UINT ux_device_stack_class_unregister(
    UCHAR *class_name,
    UINT (*class_entry_function)(struct UX_SLAVE_CLASS_COMMAND_STRUCT*));
```

说明

应用程序使用此函数注销 USB 设备类。

> **NOTE**
>
> class_name 的 C 字符串必须以 NULL 结尾，其长度（不包括 NULL 终止符本身）不能大于 UX_MAX_CLASS_NAME_LENGTH。

参数

- class_name：类名
- class_entry_function：类的入口函数。

返回值

- UX_SUCCESS：(0x00) 类已注销。
- UX_NO_CLASS_MATCH：(0x57) 类未注册。

示例

```
/* The following example illustrates this service. */

/* Unitialize the device storage class. */
status = ux_device_stack_class_unregister(_ux_system_slave_class_storage_name,
ux_device_class_storage_entry);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_configuration_get

获取当前配置

原型

```
UINT ux_device_stack_configuration_get(VOID);
```

说明

主机使用此函数获取设备中运行的当前配置。

输入参数

无

返回值

- UX_SUCCESS：(0x00) 已完成数据传输。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_device_stack_configuration_get();

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_configuration_set

设置当前配置

原型

```
UINT ux_device_stack_configuration_set(ULONG configuration_value);
```

说明

主机使用此函数设置设备中运行的当前配置。收到此命令后，USB 设备堆栈将激活已连接到此配置的每个接口的备用设置 0。

输入参数

- configuration_value：主机选择的配置值。

返回值

- UX_SUCCESS：(0x00) 已成功设置配置。

示例

```
ULONG configuration_value;
UINT status;

/* The following example illustrates this service. */
status = ux_device_stack_configuration_set(configuration_value);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_descriptor_send

将描述符发送到主机

原型

```
UINT ux_device_stack_descriptor_send(
    ULONG descriptor_type,
    ULONG request_index,
    ULONG host_length);
```

说明

设备端使用此函数向主机返回描述符。此描述符可以是设备描述符、配置描述符或字符串描述符。

参数

- descriptor_type：描述符的类型。必须是以下值之一。

- UX_DEVICE_DESCRIPTOR_ITEM
- UX_CONFIGURATION_DESCRIPTOR_ITEM
- UX_STRING_DESCRIPTOR_ITEM
- UX_DEVICE_QUALIFIER_DESCRIPTOR_ITEM
- UX_OTHER_SPEED_DESCRIPTOR_ITEM

- request_index：描述符的索引。
- host_length：主机所需的长度。

返回值

- UX_SUCCESS：(0x00) 已完成数据传输。
- UX_ERROR：(0xFF) 传输未完成。

示例

```
ULONG descriptor_type;
ULONG request_index;
ULONG host_length;
UINT status;

/* The following example illustrates this service. */
status = ux_device_stack_descriptor_send(descriptor_type, request_index, host_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_disconnect

断开连接设备堆栈

原型

```
UINT ux_device_stack_disconnect(VOID);
```

说明

当设备断开连接时，VBUS 管理器将调用此函数。设备堆栈将告知所有已注册到此设备的类，然后释放所有设备资源。

输入参数

无

返回值

- UX_SUCCESS：(0x00) 设备已断开连接。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_device_stack_disconnect();

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_endpoint_stall

请求终结点停滞状态

原型

```
UINT ux_device_stack_endpoint_stall(UX_SLAVE_ENDPOINT*endpoint);
```

说明

当终结点应向主机返回停滞状态时，USB 设备类将调用此函数。

输入参数

- endpoint：在其上请求停滞状态的终结点。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_ERROR：(0xFF) 设备处于无效状态。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_device_stack_endpoint_stall(endpoint);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_host_wakeup

唤醒主机

原型

```
UINT ux_device_stack_host_wakeup(VOID);
```

说明

当设备想要唤醒主机时，将调用此函数。此命令仅在设备处于挂起模式时才有效。由设备应用程序决定何时要唤醒 USB 主机。例如，当 USB 调制解调器在电话线路上检测到 RING 信号时，它便可以唤醒主机。

输入参数

无

返回值

- UX_SUCCESS：(0x00) 调用成功。
- UX_FUNCTION_NOT_SUPPORTED：(0x54) 调用失败（设备可能不处于挂起模式）。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_device_stack_host_wakeup();

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_initialize

初始化 USB 设备堆栈

原型

```
UINT ux_device_stack_initialize(
    UCHAR *device_framework_high_speed,
    ULONG device_framework_length_high_speed,
    UCHAR *device_framework_full_speed,
    ULONG device_framework_length_full_speed,
    UCHAR *string_framework,
    ULONG string_framework_length,
    UCHAR *language_id_framework,
    ULONG language_id_framework_length),
    UINT (*ux_system_slave_change_function)(ULONG)));
```

说明

应用程序调用此函数来初始化 USB 设备堆栈。它不会初始化任何类或任何控制器。应使用单独的函数调用来执行此操作。此调用的主要目的是为堆栈提供 USB 函数的设备框架。它支持高速和全速，对于每种速度，它可能提供完全不同的设备框架。支持字符串框架和多种语言。

参数

- device_framework_high_speed：指向高速框架的指针。
- device_framework_length_high_speed：高速框架的长度。
- device_framework_full_speed：指向全速框架的指针。
- device_framework_length_full_speed：全速框架的长度。
- string_framework：指向字符串框架的指针。
- string_framework_length：字符串框架的长度。
- language_id_framework：指向字符串语言框架的指针。
- language_id_framework_length：字符串语言框架的长度。
- ux_system_slave_change_function：设备状态更改时要调用的函数。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_MEMORY_INSUFFICIENT：(0x12) 内存不足，无法初始化堆栈。
- UX_DESCRIPTOR_CORRUPTED：(0x42) 描述符无效。

示例

```
/* Example of a device framework */

#define DEVICE_FRAMEWORK_LENGTH_FULL_SPEED 50

UCHAR device_framework_full_speed[] = {
    /* Device descriptor */
    0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x08,
    0xec, 0x08, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x01,

    /* Configuration descriptor */
    0x09, 0x02, 0x20, 0x00, 0x01, 0x01, 0x00, 0xc0,
    0x32,

    /* Interface descriptor */
    0x09, 0x04, 0x00, 0x00, 0x02, 0x08, 0x06, 0x50,
    0x00,

    /* Endpoint descriptor (Bulk Out) */
    0x07, 0x05, 0x01, 0x02, 0x40, 0x00, 0x00,

    /* Endpoint descriptor (Bulk In) */
    0x07, 0x05, 0x82, 0x02, 0x40, 0x00, 0x00
};
```

```
#define DEVICE_FRAMEWORK_LENGTH_HIGH_SPEED 60

UCHAR device_framework_high_speed[] = {
    /* Device descriptor */
    0x12, 0x01, 0x00, 0x02, 0x00, 0x00, 0x00, 0x40,
    0x0a, 0x07, 0x25, 0x40, 0x01, 0x00, 0x01, 0x02,
    0x03, 0x01,

    /* Device qualifier descriptor */
    0x0a, 0x06, 0x00, 0x02, 0x00, 0x00, 0x00, 0x40,
    0x01, 0x00,

    /* Configuration descriptor */
    0x09, 0x02, 0x20, 0x00, 0x01, 0x01, 0x00, 0xc0,
    0x32,

    /* Interface descriptor */
    0x09, 0x04, 0x00, 0x00, 0x02, 0x08, 0x06, 0x50,
    0x00,

    /* Endpoint descriptor (Bulk Out) */
    0x07, 0x05, 0x01, 0x02, 0x00, 0x02, 0x00,

    /* Endpoint descriptor (Bulk In) */
    0x07, 0x05, 0x82, 0x02, 0x00, 0x02, 0x00
};

/* String Device Framework:
    Byte 0 and 1: Word containing the language ID: 0x0904 for US
    Byte 2 : Byte containing the index of the descriptor
    Byte 3 : Byte containing the length of the descriptor string */

#define STRING_FRAMEWORK_LENGTH 38 UCHAR string_framework[] = {
    /* Manufacturer string descriptor: Index 1 */
    0x09, 0x04, 0x01, 0x0c,
    0x45, 0x78, 0x70, 0x72,0x65, 0x73, 0x20, 0x4c,
    0x6f, 0x67, 0x69, 0x63,

    /* Product string descriptor: Index 2 */
    0x09, 0x04, 0x02, 0x0c,
    0x4D, 0x4C, 0x36, 0x39, 0x36, 0x35, 0x30, 0x30,
    0x20, 0x53, 0x44, 0x4B,

    /* Serial Number string descriptor: Index 3 */
    0x09, 0x04, 0x03, 0x04,
    0x30, 0x30, 0x30, 0x31
};

/* Multiple languages are supported on the device, to add a language besides English,
  the Unicode language code must be appended to the language_id_framework array
  and the length adjusted accordingly. */

#define LANGUAGE_ID_FRAMEWORK_LENGTH 2

UCHAR language_id_framework[] = {
    /* English. */
    0x09, 0x04
};
```

当控制器更改了其状态时，应用程序可以请求回调。控制器的两种主要状态为：

- UX_DEVICE_SUSPENDED
- UX_DEVICE_RESUMED

如果应用程序不需要暂停/继续信号，则会提供 UX_NULL 函数。

```
UINT status;

/* The code below is required for installing the device portion of USBX.
    There is no call back for device status change in this example. */
status = ux_device_stack_initialize(&device_framework_high_speed,
    DEVICE_FRAMEWORK_LENGTH_HIGH_SPEED, &device_framework_full_speed,
    DEVICE_FRAMEWORK_LENGTH_FULL_SPEED, &string_framework,
    STRING_FRAMEWORK_LENGTH, &language_id_framework,
    LANGUAGE_ID_FRAMEWORK_LENGTH, UX_NULL);

/* If status equals UX_SUCCESS, initialization was successful. */
```

ux_device_stack_interface_delete

删除堆栈接口

原型

```
UINT ux_device_stack_interface_delete(UX_SLAVE_INTERFACE*interface);
```

说明

应删除接口时将调用此函数。在提取设备时、总线重置后，或者有新的备用设置时，将删除接口。

输入参数

- interface：指向要删除的接口的指针。

返回值

- UX_SUCCESS：(0x00) 此操作成功。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_device_stack_interface_delete(interface);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_interface_get

获取当前接口值

原型

```
UINT ux_device_stack_interface_get(UINT interface_value);
```

说明

当主机查询当前接口时，将调用此函数。设备将返回当前接口值。

> **NOTE**
> 不推荐使用此函数。此函数适用于旧式软件，新式软件应改用 ux_device_stack_alternate_setting_get 函数。

输入参数

- interface_value：要返回的接口值。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_ERROR：(0xFF) 不存在接口。

示例

```
ULONG interface_value;

UINT status;

/* The following example illustrates this service. */
status = ux_device_stack_interface_get(interface_value);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_interface_set

更改接口的备用设置

原型

```
UINT ux_device_stack_interface_set(
    UCHAR *device_framework,
    ULONG device_framework_length,
    ULONG alternate_setting_value);
```

说明

当主机请求更改接口的备用设置时，将调用此函数。

参数

- device_framework：此接口的设备框架的地址。
- device_framework_length：设备框架的长度。
- alternate_setting_value：此接口要使用的备用设置值。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_ERROR：(0xFF) 不存在接口。

示例

```
UCHAR * device_framework
ULONG device_framework_length;
ULONG alternate_setting_value;
UINT status;

/* The following example illustrates this service. */
status = ux_device_stack_interface_set(device_framework,
    device_framework_length, alternate_setting_value);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_interface_start

开始搜索类以拥有接口实例

原型

```
UINT ux_device_stack_interface_start(UX_SLAVE_INTERFACE*interface);
```

说明

当主机已选择某个接口，并且设备堆栈需要搜索设备类以拥有此接口实例时，将调用此函数。

输入参数

- interface：指向已创建的接口的指针。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_NO_CLASS_MATCH：(0x57) 此接口不存在任何类。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_device_stack_interface_start(interface);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_transfer_request

请求将数据传输到主机

原型

```
UINT ux_device_stack_transfer_request(
    UX_SLAVE_TRANSFER *transfer_request,
    ULONG slave_length,
    ULONG host_length);
```

说明

当类或堆栈要将数据传输到主机时，将调用此函数。主机始终轮询设备，但设备可以提前准备数据。

参数

- transfer_request：指向传输请求的指针。
- slave_length：设备要返回的长度。
- host_length：主机已请求的长度。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_TRANSFER_NOT_READY：(0x25) 设备处于无效状态；状态必须是 ATTACHED、CONFIGURED 或 ADDRESSED 。
- UX_ERROR：(0xFF) 传输错误。

示例

```
UINT status;

/* The following example illustrates how to transfer more data than an application requests. */
while(total_length) {
    /* How much can we send in this transfer? */
    if (total_length > UX_SLAVE_CLASS_STORAGE_BUFFER_SIZE)
        transfer_length = UX_SLAVE_CLASS_STORAGE_BUFFER_SIZE;
    else
        transfer_length = total_length;

   /* Copy the Storage Buffer into the transfer request memory. */
   ux_utility_memory_copy(transfer_request ->  ux_slave_transfer_request_data_pointer,
       media_memory, transfer_length);

   /* Send the data payload back to the caller. */
   status = ux_device_transfer_request(transfer_request,
       transfer_length, transfer_length);

   /* If status equals UX_SUCCESS, the operation was successful. */

   /* Update the buffer address.  */
    media_memory += transfer_length;

   /* Update the length to remain. */
    total_length -= transfer_length;
}
```

ux_device_stack_transfer_abort

取消转移请求

原型

```
UINT ux_device_stack_transfer_abort(
    UX_SLAVE_TRANSFER *transfer_request,
    ULONG completion_code);
```

说明

当应用程序需要取消传输请求，或者堆栈需要中止与某个终结点关联的传输请求时，将调用此函数。

参数

- transfer_request：指向传输请求的指针。
- completion_code：要向正在等待此传输请求完成的类返回的错误代码。

返回值

- UX_SUCCESS：(0x00) 此操作成功。

示例

```
UINT status;

/* The following example illustrates how to abort a transfer when a
    bus reset has been detected on the bus. */
status = ux_device_stack_transfer_abort(transfer_request, UX_TRANSFER_BUS_RESET);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_device_stack_uninitialize

取消初始化堆栈

原型

```
UINT ux_device_stack_uninitialize();
```

说明

当应用程序需要取消初始化 USBX 设备堆栈时，将调用此函数 – 所有设备堆栈资源均会释放。应在已通过 ux_device_stack_class_unregister 注销所有类之后调用此函数。

参数

无

返回值

UX_SUCCESS：(0x00) 此操作成功。

第 5 章 - USBX 设备类注意事项

2021/4/29 •

设备类注册

每个设备类都遵循相同的注册原则。包含特定类参数的结构将传递给类初始化函数。

```
/* Set the parameters for callback when insertion/extraction of a HID device. */

hid_parameter.ux_slave_class_hid_instance_activate = tx_demo_hid_instance_activate;

hid_parameter.ux_slave_class_hid_instance_deactivate = tx_demo_hid_instance_deactivate;

/* Initialize the hid class parameters for the device. */
hid_parameter.ux_device_class_hid_parameter_report_address = hid_device_report;

hid_parameter.ux_device_class_hid_parameter_report_length = HID_DEVICE_REPORT_LENGTH;

hid_parameter.ux_device_class_hid_parameter_report_id = UX_TRUE;
hid_parameter.ux_device_class_hid_parameter_callback = demo_thread_hid_callback;

/* Initialize the device hid class. The class is connected with interface 0 */

status = ux_device_stack_class_register(_ux_system_slave_class_hid_name,
    ux_device_class_hid_entry,1,0, (VOID *)&hid_parameter);
```

当激活类的实例时，每个类都可以注册一个回调函数。然后，设备堆栈调用该回调，以通知应用程序已创建实例。

应用程序的主体中将有 2 个用于激活和停用的函数，如下面的示例中所示。

```
VOID tx_demo_hid_instance_activate(VOID *hid_instance)
{
    /* Save the HID instance. */
    hid_slave = (UX_SLAVE_CLASS_HID *) hid_instance;
}

VOID tx_demo_hid_instance_deactivate(VOID *hid_instance)
{
    /* Reset the HID instance. */
    hid_slave = UX_NULL;
}
```

不建议在这些函数中执行任何操作，但要记住类的实例，并与应用程序的其余部分同步。

USB 设备存储类

USB 设备存储类允许在系统中嵌入的存储设备对 USB 主机可见。

USB 设备存储类本身不提供存储解决方案。它仅接受并解释来自主机的 SCSI 请求。当其中一个请求为读取或写入命令时，它将调用一个预定义的回调到实际存储设备处理程序，例如 ATA 设备驱动程序或闪存设备驱动程序。

初始化设备存储类时，会向包含所有必需信息的类提供指针结构。下面给出了一个示例。

```
/* Initialize the storage class parameters to customize vendor strings. */

storage_parameter.ux_slave_class_storage_parameter_vendor_id
    = demo_ux_system_slave_class_storage_vendor_id;

storage_parameter.ux_slave_class_storage_parameter_product_id
    = demo_ux_system_slave_class_storage_product_id;

storage_parameter.ux_slave_class_storage_parameter_product_rev
    = demo_ux_system_slave_class_storage_product_rev;

storage_parameter.ux_slave_class_storage_parameter_product_serial
    = demo_ux_system_slave_class_storage_product_serial;

/* Store the number of LUN in this device storage instance: single LUN. */
storage_parameter.ux_slave_class_storage_parameter_number_lun = 1;

/* Initialize the storage class parameters for reading/writing to the Flash Disk. */

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_last_lba = 0x1e6bfe;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_block_length = 512;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_type = 0;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_removable_flag =
0x80;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_read_only_flag
    = UX_FALSE;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_read
    = tx_demo_thread_flash_media_read;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_write
    = tx_demo_thread_flash_media_write;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_status
    = tx_demo_thread_flash_media_status;

/* Initialize the device storage class. The class is connected with interface 0 */
status = ux_device_stack_class_register(_ux_system_slave_class_storage_name,
    ux_device_class_storage_entry, ux_device_class_storage_thread, 0, (VOID *)&storage_parameter);
```

在此示例中，通过将字符串指针分配给相应的参数来自定义驱动程序存储字符串。如果将任何一个字符串指针留给 UX_NULL，则使用默认的 Azure RTOS 字符串。

在此示例中，将提供驱动器的最后一个块地址或 LBA 以及逻辑扇区大小。LBA 是介质 (1) 中可用的扇区数。常规存储介质中的块长度设置为 512。对于光驱，可以将其设置为 2048。

应用程序需要传递三个回调函数指针，以允许存储类读取、写入和获取介质状态。

下面的示例展示了 read 和 write 函数的原型。

```
UINT media_read(
    VOID *storage,
    ULONG lun,
    UCHAR *data_pointer,
    ULONG number_blocks,
    ULONG lba,
    ULONG *media_status);

UINT media_write(
    VOID *storage,
    ULONG lun,
    UCHAR *data_pointer,
    ULONG number_blocks,
    ULONG lba,
    ULONG *media_status);
```

其中：

- storage 是存储类的实例。
- lun 是命令定向到的 LUN。
- data_pointer 是要用于读取或写入的缓冲区的地址。
- number_blocks 是要读取/写入的扇区数。
- lba 是要读取的扇区地址。
- media_status 应完全按照介质状态回调的返回值来填写。

返回值可以是 UX_SUCCESS 或 UX_ERROR 的值，表示操作成功或不成功。这些操作不需要返回任何其他错误代码。如果任何操作中出现错误，则存储类将调用状态回调函数。

此函数具有以下原型。

```
ULONG media_status(
    VOID *storage,
    ULONG lun,
    ULONG media_id,
    ULONG *media_status);
```

当前未使用调用参数 media_id，它应始终为 0。将来，可以使用它来区分多个存储设备或具有多个 SCSI LUN 的存储设备。此版本的存储类不支持多个具有多个 SCSI LUN 的存储类或存储设备实例。

返回值是一个可以采用以下格式的 SCSI 错误代码。

- **Bits 0-7** Sense_key
- **Bits 8-15** 其他感知代码
- **Bits 16-23** 其他感知代码限定符

下表提供了可能的感知/ASC/ASCQ 组合。

IIII	ASC	ASCQ	II
00	00	00	无感知
01	17	01	重试后恢复的数据
01	18	00	使用 ECC 恢复数据
02	04	01	逻辑驱动器未就绪 - 准备就绪

IIII	ASC	ASCQ	II
02	04	02	逻辑驱动器未就绪 - 需要初始化
02	04	04	逻辑单元未就绪 - 正在进行格式设置
02	04	FF	逻辑驱动器未就绪 - 设备繁忙
02	06	00	找不到引用位置
02	08	00	逻辑单元通信失败
02	08	01	逻辑单元通信超时
02	08	80	逻辑单元通信超限
02	3A	00	不存在介质
02	54	00	USB 到主机系统接口故障
02	80	00	资源不足
02	FF	FF	未知错误
03	02	00	无查找完成
03	03	00	写入错误
03	10	00	ID CRC 错误
03	11	00	未恢复的读取错误
03	12	00	找不到 ID 字段的地址标记
03	13	00	找不到数据字段的地址标记
03	14	00	找不到记录的实体
03	30	01	无法读取介质 - 未知格式
03	31	01	格式命令失败
04	40	NN	组件 NN(80H-FFH)的诊断故障
05	1A	00	参数列表长度错误
05	20	00	无效的命令操作代码
05	21	00	逻辑块地址超出范围

ιιιι	ASC	ASCQ	ιι
05	24	00	命令包中的无效字段
05	25	00	逻辑单元不受支持
05	26	00	参数列表中的无效字段
05	26	01	参数不受支持
05	26	02	参数值无效
05	39	00	保存参数不受支持
06	28	00	未准备好进行转换 - 介质已更改
06	29	00	发生开机重置或总线设备重置
06	2F	00	其他发起程序清除的命令
07	27	00	写入受保护的介质
0B	4E	00	尝试重叠的命令

应用程序可以实现另外两种可选的回调;一种用于响应 GET_STATUS_NOTIFICATION 命令，另一种用于响应 SYNCHRONIZE_CACHE 命令 。

如果应用程序要处理主机中的 GET_STATUS_NOTIFICATION 命令，则应使用以下原型实现回调。

```
UINT ux_slave_class_storage_media_notification(
    VOID *storage,
    ULONG lun,
    ULONG media_id,
    ULONG notification_class,
    UCHAR **media_notification,
    ULONG *media_notification_length);
```

其中：

- storage 是存储类的实例。
- 当前未使用 media_id。notification_class 指定通知的类。
- media_notification 应由应用程序设置为包含通知响应的缓冲区。
- media_notification_length 应由应用程序设置为包含响应缓冲区的长度。

返回值指示命令是否成功 - 应为 UX_SUCCESS 或 UX_ERROR 。

如果应用程序未实现此回调，则在接收到 GET_STATUS_NOTIFICATION 命令后，USBX 将通知主机未实现该命令。

如果应用程序利用缓存从主机写入，则应处理 SYNCHRONIZE_CACHE 命令。如果主机知道存储设备即将断开连接，则主机可以发送此命令，例如，在 Windows 中，如果右键单击工具栏中的闪存驱动器图标并选择“弹出 [存储设备名称]”，Windows 将向该设备发出 SYNCHRONIZE_CACHE 命令。

如果应用程序要处理主机中的 GET_STATUS_NOTIFICATION 命令，则应使用以下原型实现回调。

```
UINT ux_slave_class_storage_media_flush(
    VOID *storage,
    ULONG lun,
    ULONG number_blocks,
    ULONG lba,
    ULONG *media_status);
```

其中：

- storage 是存储类的实例。
- lun 参数指定命令所指向的 LUN。
- number_blocks 指定要同步的块的数目。
- lba 是要同步的第一个块的扇区地址。
- media_status 应完全按照介质状态回调的返回值来填写。

返回值指示命令是否成功 - 应为 UX_SUCCESS 或 UX_ERROR 。

多个 SCSI LUN

USBX 设备存储类支持多个 LUN。因此，可以创建一个充当 CD-ROM 的存储设备，并同时创建一个闪存盘。在这种情况下，初始化将略有不同。下面是闪存盘和 CD-ROM 的示例：

```
/* Store the number of LUN in this device storage instance. */
storage_parameter.ux_slave_class_storage_parameter_number_lun = 2;

/* Initialize the storage class parameters for reading/writing to the Flash Disk. */
storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_last_lba = 0x7bbff;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_block_length = 512;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_type = 0;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_removable_flag =
0x80;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_read =
tx_demo_thread_flash_media_read;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_write =
tx_demo_thread_flash_media_write;

storage_parameter.ux_slave_class_storage_parameter_lun[0].ux_slave_class_storage_media_status =
tx_demo_thread_flash_media_status;

/* Initialize the storage class LUN parameters for reading/writing to the CD-ROM. */

storage_parameter.ux_slave_class_storage_parameter_lun[1].ux_slave_class_storage_media_last_lba = 0x04caaf;

storage_parameter.ux_slave_class_storage_parameter_lun[1].ux_slave_class_storage_media_block_length = 2048;

storage_parameter.ux_slave_class_storage_parameter_lun[1].ux_slave_class_storage_media_type = 5;

storage_parameter.ux_slave_class_storage_parameter_lun[1].ux_slave_class_storage_media_removable_flag =
0x80;

storage_parameter.ux_slave_class_storage_parameter_lun[1].ux_slave_class_storage_media_read =
tx_demo_thread_cdrom_media_read;

storage_parameter.ux_slave_class_storage_parameter_lun[1].ux_slave_class_storage_media_write =
tx_demo_thread_cdrom_media_write;

storage_parameter.ux_slave_class_storage_parameter_lun[1].ux_slave_class_storage_media_status =
tx_demo_thread_cdrom_media_status;

/* Initialize the device storage class for a Flash disk and CD-ROM. The class is connected with interface 0
*/ status =
ux_device_stack_class_register(_ux_system_slave_class_storage_name,ux_device_class_storage_entry,
    ux_device_class_storage_thread,0, (VOID *) &storage_parameter);
```

USB 设备 CDC-ACM 类

USB 设备 CDC-ACM 类允许 USB 主机系统将设备作为串行设备进行通信。此类基于 USB 标准，并且是 CDC 标准的子集。

符合 CDC-ACM 的设备框架需要由设备堆栈声明。下面是一个示例。

```
unsigned char device_framework_full_speed[] = {

    /*
    Device descriptor 18 bytes
    0x02 bDeviceClass: CDC class code
    0x00 bDeviceSubclass: CDC class sub code 0x00 bDeviceProtocol: CDC Device protocol
    idVendor & idProduct - https://www.linux-usb.org/usb.ids
    */

    0x12, 0x01, 0x10, 0x01,
    0xEF, 0x02, 0x01, 0x08,
    0x84, 0x84, 0x00, 0x00,
    0x00, 0x01, 0x01, 0x02,
    0x03, 0x01,

    /* Configuration 1 descriptor 9 bytes */
    0x09, 0x02, 0x4b, 0x00, 0x02, 0x01, 0x00,0x40, 0x00,

    /* Interface association descriptor. 8 bytes. */
    0x08, 0x0b, 0x00,
    0x02, 0x02, 0x02, 0x00, 0x00,

    /* Communication Class Interface Descriptor Requirement. 9 bytes. */
    0x09, 0x04, 0x00, 0x00,0x01,0x02, 0x02, 0x01, 0x00,

    /* Header Functional Descriptor 5 bytes */
    0x05, 0x24, 0x00,0x10, 0x01,

    /* ACM Functional Descriptor 4 bytes */
    0x04, 0x24, 0x02,0x0f,

    /* Union Functional Descriptor 5 bytes */
    0x05, 0x24, 0x06, 0x00,

    /* Master interface */
    0x01, /* Slave interface */

    /* Call Management Functional Descriptor 5 bytes */
    0x05, 0x24, 0x01,0x03, 0x01, /* Data interface */

    /* Endpoint 1 descriptor 7 bytes */
    0x07, 0x05, 0x83, 0x03,0x08, 0x00, 0xFF,

    /* Data Class Interface Descriptor Requirement 9 bytes */
    0x09, 0x04, 0x01, 0x00, 0x02,0x0A, 0x00, 0x00, 0x00,

    /* First alternate setting Endpoint 1 descriptor 7 bytes*/
    0x07, 0x05, 0x02,0x02,0x40, 0x00,0x00,

    /* Endpoint 2 descriptor 7 bytes */
    0x07, 0x05, 0x81,0x02,0x40, 0x00, 0x00,

};
```

CDC-ACM 类使用复合设备框架对接口（控件和数据）进行分组。因此，在定义设备描述符时应保持谨慎。USBX 依赖于 IAD 描述符来了解如何在内部绑定接口。IAD 描述符应在接口之前声明，并包含 CDC-ACM 类的第一个接口和附加的接口数量。

CDC-ACM 类还使用联合功能描述符，该描述符执行与较新的 IAD 描述符相同的功能。尽管出于历史原因和与主机端兼容的考虑，必须声明联合功能描述符，但它并不被 USBX 使用。

CDC-ACM 类的初始化需要以下参数。

```
/* Set the parameters for callback when insertion/extraction of a CDC device. */

parameter.ux_slave_class_cdc_acm_instance_activate = tx_demo_cdc_instance_activate;

parameter.ux_slave_class_cdc_acm_instance_deactivate = tx_demo_cdc_instance_deactivate;

parameter.ux_slave_class_cdc_acm_parameter_change = tx_demo_cdc_instance_parameter_change;

/* Initialize the device cdc class. This class owns both interfaces starting with 0. */
status = ux_device_stack_class_register(_ux_system_slave_class_cdc_acm_name,ux_device_class_cdc_acm_entry,
    1,0, &parameter);
```

定义的 2 个参数是指向用户应用程序的回调指针，这些指针在堆栈激活或停用此类时被调用。

定义的第三个参数是指向用户应用程序的回调指针，该指针在行编码或行状态参数发生更改时被调用。例如，当主机请求将 DTR 状态改为 TRUE 时，回调被调用，在回调中用户应用程序可以通过 IOCTL 函数检查行状态，以了解主机是否可用于通信。

CDC-ACM 基于 USB-IF 标准，可被 MACO 和 Linux 操作系统自动识别。在 Windows 平台上，此类需要在 Windows 10 之前的版本中使用 .inf 文件。Windows 10 不需要任何 .inf 文件。我们提供了 CDC-ACM 类的模板，可以在 usbx_windows_host_files 目录中找到它。对于较新版本的 Windows，应使用 CDC_ACM_Template_Win7_64bit.inf 文件（Win10 除外）。需要对此文件进行修改，以反映设备使用的 PID/VID。将公司和产品注册到 USB-IF 后，该 PID/VID 将特定于最终客户。在 inf 文件中，要修改的字段位于此处。

```
[DeviceList]
%DESCRIPTION%=DriverInstall, USB\VID_8484&PID_0000

[DeviceList.NTamd64]
%DESCRIPTION%=DriverInstall, USB\VID_8484&PID_0000
```

在 CDC-ACM 设备的设备框架中，PID/VID 存储在设备描述符（请参见上面声明的设备描述符）中。

USB 主机系统发现 USB CDC-ACM 设备后，它将装载一个串行类，并且该设备可以与任何串行终端程序一起使用。有关参考信息，请参阅“主机操作系统”。

下面定义了 CDC-ACM 类 API 函数。

ux_device_class_cdc_acm_ioctl

在 CDC-ACM 接口上执行 IOCTL

原型

```
UINT ux_device_class_cdc_acm_ioctl (
    UX_SLAVE_CLASS_CDC_ACM *cdc_acm,
    ULONG ioctl_function,
    VOID *parameter);
```

说明

应用程序需要执行对 cdc acm 接口的各种 ioctl 调用时，将调用此函数

参数

- **cdc_acm**：指向 cdc 类实例的指针。
- **ioctl_function**：要执行的 ioctl 函数。
- **parameter**：指向特定于 ioctl 的参数的指针。

返回值

- **UX_SUCCESS** (0x00) 此操作成功。

- **UX_ERROR** (0xFF) 函数出错

示例

```
/* Start cdc acm callback transmission. */

status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
    UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START, &callback);

if(status != UX_SUCCESS)
    return;
```

Ioctl 函数:

tt	"t"
UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_CODING	1
UX_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING	2
UX_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_STATE	3
UX_SLAVE_CLASS_CDC_ACM_IOCTL_ABORT_PIPE	4
UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_STATE	5
UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START	6
UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_STOP	7

ux_device_class_cdc_acm_ioctl:**UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_CODING**

在 CDC-ACM 接口上执行 IOCTL 设置行编码

原型

```
UINT ux_device_class_cdc_acm_ioctl (
    UX_SLAVE_CLASS_CDC_ACM*cdc_acm,
    ULONG ioctl_function,
    VOID *parameter);
```

说明

应用程序需要设置行编码参数时，将调用此函数。

参数

- **cdc_acm**:指向 cdc 类实例的指针。
- **ioctl_function**:ux_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_CODING
- **参数**:指向行参数结构的指针:

```
typedef struct UX_SLAVE_CLASS_CDC_ACM_LINE_CODING_PARAMETER_STRUCT
{
    ULONG ux_slave_class_cdc_acm_parameter_baudrate;
    UCHAR ux_slave_class_cdc_acm_parameter_stop_bit;
    UCHAR ux_slave_class_cdc_acm_parameter_parity;
    UCHAR ux_slave_class_cdc_acm_parameter_data_bit;
} UX_SLAVE_CLASS_CDC_ACM_LINE_CODING_PARAMETER;
```

返回值

UX_SUCCESS (0x00) 此操作成功。

示例

```
/* Change the line coding values. */

line_coding.ux_slave_class_cdc_acm_line_coding_dter = 9600;
line_coding.ux_slave_class_cdc_acm_line_coding_stop_bit =
    UX_HOST_CLASS_CDC_ACM_LINE_CODING_STOP_BIT_15;

line_coding.ux_slave_class_cdc_acm_line_coding_parity =
    UX_HOST_CLASS_CDC_ACM_LINE_CODING_PARITY_EVEN;

line_coding.ux_slave_class_cdc_acm_line_coding_data_bits = 5;

status = _ux_slave_class_cdc_acm_ioctl(cdc_acm,
    UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_CODING, &line_coding);

if (status != UX_SUCCESS)
    break;
```

ux_device_class_cdc_acm_ioctl：UX_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING

在 CDC-ACM 接口上执行 IOCTL 获取行编码

原型

```
device_class_cdc_acm_ioctl (
    UX_SLAVE_CLASS_CDC_ACM *cdc_acm,
    ULONG ioctl_function,
    VOID *parameter);
```

说明

应用程序需要获取行编码参数时，将调用此函数。

参数

- **cdc_acm**：指向 cdc 类实例的指针。
- **ioctl_function**：ux_SLAVE_CLASS_CDC_ACM_IOCTL_GET_ LINE_CODING
- **参数**：指向行参数结构的指针：

```
typedef struct UX_SLAVE_CLASS_CDC_ACM_LINE_CODING_PARAMETER_STRUCT
{
    ULONG ux_slave_class_cdc_acm_parameter_baudrate;
    UCHAR ux_slave_class_cdc_acm_parameter_stop_bit;
    UCHAR ux_slave_class_cdc_acm_parameter_parity;
    UCHAR ux_slave_class_cdc_acm_parameter_data_bit;
} UX_SLAVE_CLASS_CDC_ACM_LINE_CODING_PARAMETER;
```

返回值

- UX_SUCCESS (0x00) 此操作成功。

示例

```
/* This is to retrieve BAUD rate. */

status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
    UX_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING, &line_coding);

/* Any error ? */
if (status == UX_SUCCESS)
{
    /* Decode BAUD rate. */
    switch (line_coding.ux_slave_class_cdc_acm_parameter_baudrate)
    {
        case 1200 :
            status = tx_demo_thread_slave_cdc_acm_response("1200", 4);
            break;
        case 2400 :
            status = tx_demo_thread_slave_cdc_acm_response("2400", 4);
            break;
        case 4800 :
            status = tx_demo_thread_slave_cdc_acm_response("4800", 4);
            break;
        case 9600 :
            status = tx_demo_thread_slave_cdc_acm_response("9600", 4);
            break;
        case 115200 :
            status = tx_demo_thread_slave_cdc_acm_response("115200", 6);
            break;
    }
}
```

ux_device_class_cdc_acm_ioctl：**UX_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_STATE**

在 CDC-ACM 接口上执行 IOCTL 获取行状态

原型

```
UINT ux_device_class_cdc_acm_ioctl (
    UX_SLAVE_CLASS_CDC_ACM*cdc_acm,
    ULONG ioctl_function,
    VOID *parameter);
```

说明

应用程序需要获取行状态参数时，将调用此函数。

参数

- **cdc_acm**：指向 cdc 类实例的指针。
- **ioctl_function**：ux_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_STATE
- **参数**：指向行参数结构的指针：

```
typedef struct UX_SLAVE_CLASS_CDC_ACM_LINE_STATE_PARAMETER_STRUCT
{
    UCHAR ux_slave_class_cdc_acm_parameter_rts;
    UCHAR ux_slave_class_cdc_acm_parameter_dtr;
} UX_SLAVE_CLASS_CDC_ACM_LINE_STATE_PARAMETER;
```

返回值

- **UX_SUCCESS** (0x00) 此操作成功。

示例

```
/* This is to retrieve RTS state. */
status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
    UX_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_STATE, &line_state);

/* Any error ? */
if (status == UX_SUCCESS)
{
/* Check state. */
    if (line_state.ux_slave_class_cdc_acm_parameter_rts == UX_TRUE)
        /* State is ON. */
        status = tx_demo_thread_slave_cdc_acm_response("RTS ON", 6);
    else
        /* State is OFF. */
        status = tx_demo_thread_slave_cdc_acm_response("RTS OFF", 7);
}
```

ux_device_class_cdc_acm_ioctl:**UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_STATE**

在CDC-ACM 接口上执行 IOCTL 设置行状态

原型

```
UINT ux_device_class_cdc_acm_ioctl (
    UX_SLAVE_CLASS_CDC_ACM *cdc_acm,
    ULONG ioctl_function,
    VOID *parameter);
```

说明

应用程序需要获取行状态参数时，将调用此函数

参数

- **cdc_acm**：指向 cdc 类实例的指针。
- **ioctl_function**：ux_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_STATE
- **参数**：指向行参数结构的指针：

```
typedef struct UX_SLAVE_CLASS_CDC_ACM_LINE_STATE_PARAMETER_STRUCT
{
    UCHAR ux_slave_class_cdc_acm_parameter_rts;
    UCHAR ux_slave_class_cdc_acm_parameter_dtr;
} UX_SLAVE_CLASS_CDC_ACM_LINE_STATE_PARAMETER;
```

返回值

- **UX_SUCCESS** (0x00) 此操作成功。

示例

```
/* This is to set RTS state. */

line_state.ux_slave_class_cdc_acm_parameter_rts = UX_TRUE;
status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
    UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_STATE, &line_state);

/* If status is UX_SUCCESS, the operation was successful. */
```

ux_device_class_cdc_acm_ioctl:**UX_SLAVE_CLASS_CDC_ACM_IOCTL_ABORT_PIPE**

在 CDC-ACM 接口上执行 IOCTL 中止管道

原型

```
UINT ux_device_class_cdc_acm_ioctl (
    UX_SLAVE_CLASS_CDC_ACM *cdc_acm,
    ULONG ioctl_function,
    VOID *parameter);
```

说明

应用程序需要中止管道时，将调用此函数。例如，若要中止正在进行的写入，UX_SLAVE_CLASS_CDC_ACM_ENDPOINT_XMIT 应作为参数传递。

参数

- **cdc_acm**：指向 cdc 类实例的指针。
- **ioctl_function**：ux_SLAVE_CLASS_CDC_ACM_IOCTL_ABORT_PIPE
- **参数**：管道方向：

```
UX_SLAVE_CLASS_CDC_ACM_ENDPOINT_XMIT 1

UX_SLAVE_CLASS_CDC_ACM_ENDPOINT_RCV 2
```

返回值

- **UX_SUCCESS** (0x00) 此操作成功。
- **UX_ENDPOINT_HANDLE_UNKNOWN** (0x53) 管道方向无效。

示例

```
/* This is to abort the Xmit pipe. */

status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
    UX_SLAVE_CLASS_CDC_ACM_IOCTL_ABORT_PIPE,
    UX_SLAVE_CLASS_CDC_ACM_ENDPOINT_XMIT);

/* If status is UX_SUCCESS, the operation was successful. */
```

ux_device_class_cdc_acm_ioctl：UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START

在 CDC-ACM 接口上执行 IOCTL 传输启动

原型

```
UINT ux_device_class_cdc_acm_ioctl (
    UX_SLAVE_CLASS_CDC_ACM *cdc_acm,
    ULONG ioctl_function,
    VOID *parameter);
```

说明

应用程序想要使用带回叫的传输时，将调用此函数。

参数

- **cdc_acm**：指向 cdc 类实例的指针。
- **ioctl_function**：ux_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START
- **参数**：指向开始传输参数结构的指针：

```
typedef struct UX_SLAVE_CLASS_CDC_ACM_CALLBACK_PARAMETER_STRUCT
{
    UINT (*ux_device_class_cdc_acm_parameter_write_callback)(struct UX_SLAVE_CLASS_CDC_ACM_STRUCT *cdc_acm,
        UINT status, ULONG length);
    UINT (*ux_device_class_cdc_acm_parameter_read_callback)(struct UX_SLAVE_CLASS_CDC_ACM_STRUCT *cdc_acm,
        UINT status, UCHAR *data_pointer, ULONG length);
} UX_SLAVE_CLASS_CDC_ACM_CALLBACK_PARAMETER;
```

返回值

- **UX_SUCCESS** (0x00) 此操作成功。
- **UX_ERROR** (0xFF) 传输已开始。
- **UX_MEMORY_INSUFFICIENT** (0x12) 内存分配失败。

示例

```
/* Set the callback parameter. */

callback.ux_device_class_cdc_acm_parameter_write_callback
    = tx_demo_thread_slave_write_callback;

callback.ux_device_class_cdc_acm_parameter_read_callback
    = tx_demo_thread_slave_read_callback;

/* Program the start of transmission. */
status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
    UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START, &callback);

/* If status is UX_SUCCESS, the operation was successful. */
```

ux_device_class_cdc_acm_ioctl: **UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_STOP**

在 CDC-ACM 接口上执行 IOCTL 传输停止

原型

```
UINT ux_device_class_cdc_acm_ioctl(
    UX_SLAVE_CLASS_CDC_ACM *cdc_acm,
    ULONG ioctl_function,
    VOID *parameter);
```

说明

应用程序想要停止使用带回叫的传输时，将调用此函数。

参数

- **cdc_acm**：指向 cdc 类实例的指针。
- **ioctl_function**：ux_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSI ON_STOP
- **参数**：未使用

返回值

- **UX_SUCCESS** (0x00) 此操作成功。
- **UX_ERROR** (0xFF) 无正在进行的传输。

示例

```
/* Program the stop of transmission. */

status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
    UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_STOP, UX_NULL);

/* If status is UX_SUCCESS, the operation was successful. */
```

ux_device_class_cdc_acm_read

从 CDC-ACM 管道读取

原型

```
UINT ux_device_class_cdc_acm_read(
    UX_SLAVE_CLASS_CDC_ACM *cdc_acm,
    UCHAR *buffer,
    ULONG requested_length,
    ULONG *actual_length);
```

说明

应用程序需要从输出数据管道(从主机输出，从设备输入)读取数据时，将调用此函数。它正在阻止。

参数

- **cdc_acm**：指向 cdc 类实例的指针。
- **buffer**：将要存储数据的缓冲区地址。
- **requested_length**：所需的最大长度。
- **actual_length**：返回到缓冲区的长度。

返回值

- **UX_SUCCESS** (0x00) 此操作成功。
- **UX_CONFIGURATION_HANDLE_UNKNOWN** (0x51) 设备不再处于已配置状态。
- **UX_TRANSFER_NO_ANSWER** (0x22) 设备无应答。传输挂起时，设备可能已断开连接。

示例

```
/* Read from the CDC class. */

status = ux_device_class_cdc_acm_read(cdc, buffer, UX_DEMO_BUFFER_SIZE, &actual_length);

if(status != UX_SUCCESS) return;
```

ux_device_class_cdc_acm_write

写入 CDC-ACM 管道

原型

```
UINT ux_device_class_cdc_acm_write(
    UX_SLAVE_CLASS_CDC_ACM *cdc_acm,
    UCHAR *buffer,
    ULONG requested_length,
    ULONG *actual_length);
```

说明

应用程序需要写入输入数据管道(从主机输入，从设备输出)时，将调用此函数。它正在阻止。

参数

- **cdc_acm**：指向 cdc 类实例的指针。
- **buffer**：存储数据的缓冲区地址。
- **requested_length**：要写入的缓冲区的长度。
- **actual_length**：执行写入后返回到缓冲区的长度。

返回值

- **UX_SUCCESS** (0x00) 此操作成功。
- **UX_CONFIGURATION_HANDLE_UNKNOWN** (0x51) 设备不再处于已配置状态。
- **UX_TRANSFER_NO_ANSWER** (0x22) 设备无应答。传输挂起时，设备可能已断开连接。

示例

```
/* Write to the CDC class bulk in pipe. */

status = ux_device_class_cdc_acm_write(cdc, buffer, UX_DEMO_BUFFER_SIZE, &actual_length);

if(status != UX_SUCCESS)
    return;
```

ux_device_class_cdc_acm_write_with_callback

写入带回叫的 CDC-ACM 管道

原型

```
UINT ux_device_class_cdc_acm_write_with_callback(
    UX_SLAVE_CLASS_CDC_ACM *cdc_acm,
    UCHAR *buffer,
    ULONG requested_length);
```

说明

应用程序需要写入输入数据管道（从主机输入，从设备输出）时，将调用此函数。此函数是非阻塞的，将通过 UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START 中设置的回调完成。

参数

- **cdc_acm**：指向 cdc 类实例的指针。
- **buffer**：存储数据的缓冲区地址。
- **requested_length**：要写入的缓冲区的长度。
- **actual_length**：执行写入后返回到缓冲区的长度

返回值

- **UX_SUCCESS** (0x00) 此操作成功。
- **UX_CONFIGURATION_HANDLE_UNKNOWN** (0x51) 设备不再处于已配置状态。
- **UX_TRANSFER_NO_ANSWER** (0x22) 设备无应答。传输挂起时，设备可能已断开连接。

示例

```
/* Write to the CDC class bulk in pipe non blocking mode. */
status = ux_device_class_cdc_acm_write_with_callback(cdc, buffer, UX_DEMO_BUFFER_SIZE);

if(status != UX_SUCCESS)
    return;
```

USB 设备 CDC-ECM 类

USB 设备 CDC-ECM 类允许 USB 主机系统作为以太网设备与设备进行通信。此类基于 USB 标准，并且是 CDC

标准的子集。

符合 CDC-ACM 的设备框架需要由设备堆栈声明。下面是一个示例。

```
unsigned char device_framework_full_speed[] = {

    /* Device descriptor 18 bytes
    0x02 bDeviceClass: CDC_ECM class code
    0x06 bDeviceSubclass: CDC_ECM class sub code
    0x00 bDeviceProtocol: CDC_ECM Device protocol
    idVendor & idProduct - https://www.linux-usb.org/usb.ids
    0x3939 idVendor: Azure RTOS test.
    */

    0x12, 0x01, 0x10, 0x01,
    0x02, 0x00, 0x00, 0x08,
    0x39, 0x39, 0x08, 0x08, 0x00, 0x01, 0x01, 0x02, 03,0x01,

    /* Configuration 1 descriptor 9 bytes. */
    0x09, 0x02, 0x58, 0x00,0x02, 0x01, 0x00,0x40, 0x00,

    /* Interface association descriptor. 8 bytes. */

    0x08, 0x0b, 0x00, 0x02, 0x02, 0x06, 0x00, 0x00,

    /* Communication Class Interface Descriptor Requirement 9 bytes */
    0x09, 0x04, 0x00, 0x00,0x01,0x02, 0x06, 0x00, 0x00,

    /* Header Functional Descriptor 5 bytes */
    0x05, 0x24, 0x00, 0x10, 0x01,

    /* ECM Functional Descriptor 13 bytes */
    0x0D, 0x24, 0x0F, 0x04,0x00, 0x00, 0x00, 0x00, 0xEA, 0x05, 0x00,
    0x00,0x00,

    /* Union Functional Descriptor 5 bytes */
    0x05, 0x24, 0x06, 0x00,0x01,

    /* Endpoint descriptor (Interrupt) */
    0x07, 0x05, 0x83, 0x03, 0x08, 0x00, 0x08,

    /* Data Class Interface Descriptor Alternate Setting 0, 0 endpoints. 9 bytes */
    0x09, 0x04, 0x01, 0x00, 0x00, 0x0A, 0x00, 0x00, 0x00,

    /* Data Class Interface Descriptor Alternate Setting 1, 2 endpoints. 9 bytes */
    0x09, 0x04, 0x01, 0x01, 0x02, 0x0A, 0x00, 0x00,0x00,

    /* First alternate setting Endpoint 1 descriptor 7 bytes */
    0x07, 0x05, 0x02, 0x02, 0x40, 0x00, 0x00,

    /* Endpoint 2 descriptor 7 bytes */
    0x07, 0x05, 0x81, 0x02, 0x40, 0x00,0x00

};
```

CDC-ECM 类使用与 CDC-ACM 非常相似的设备描述符方法，并且还需要 IAD 描述符。有关定义，请参见 CDC-ACM 类。

除常规设备框架外，CDC-ECM 还需要特殊的字符串描述符。下面给出了一个示例。

```
unsigned char string_framework[] = {
    /* Manufacturer string descriptor: Index 1 - "Azure RTOS" */
    0x09, 0x04, 0x01, 0x0c,
    0x45, 0x78, 0x70, 0x72, 0x65, 0x73, 0x20, 0x4c,
    0x6f, 0x67, 0x69, 0x63,

    /* Product string descriptor: Index 2 - "EL CDCECM Device" */
    0x09, 0x04, 0x02, 0x10,
    0x45, 0x4c, 0x20, 0x43, 0x44, 0x43, 0x45, 0x43,
    0x4d, 0x20, 0x44, 0x65, 0x76, 0x69, 0x63, 0x64,

    /* Serial Number string descriptor: Index 3 - "0001" */
    0x09, 0x04, 0x03, 0x04,
    0x30, 0x30, 0x30, 0x31,

    /* MAC Address string descriptor: Index 4 - "001E5841B879" */
    0x09, 0x04, 0x04, 0x0C,
    0x30, 0x30, 0x31, 0x45, 0x35, 0x38,
    0x34, 0x31, 0x42, 0x38, 0x37, 0x39

};
```

CDC-ECM 类使用 MAC 地址字符串描述符来响应主机查询，就像设备基于 TCP/IP 协议响应 MAC 地址一样。它可以设置为设备选项，但必须在此处定义。

CDC-ECM 类的初始化如下所示。

```
/* Set the parameters for callback when insertion/extraction of a CDC device. Set to NULL.*/
cdc_ecm_parameter.ux_slave_class_cdc_ecm_instance_activate = UX_NULL;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_instance_deactivate = UX_NULL;

/* Define a NODE ID. */
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[0] = 0x00;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[1] = 0x1e;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[2] = 0x58;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[3] = 0x41;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[4] = 0xb8;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[5] = 0x78;

/* Define a remote NODE ID. */
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_remote_node_id[0] = 0x00;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_remote_node_id[1] = 0x1e;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_remote_node_id[2] = 0x58;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_remote_node_id[3] = 0x41;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_remote_node_id[4] = 0xb8;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_remote_node_id[5] = 0x79;

/* Initialize the device cdc_ecm class. */
status = ux_device_stack_class_register(_ux_system_slave_class_cdc_ecm_name,
    ux_device_class_cdc_ecm_entry, 1,0,&cdc_ecm_parameter);
```

此类的初始化在此练习中设置为 NULL 以便不执行回调，但实际上，激活和停用该初始化需要使用相同的函数回调。

后面的参数用于定义节点 ID。CDC-ECM 需要 2 个节点：本地节点和远程节点。本地节点指定设备的 MAC 地址，而远程节点指定主机的 MAC 地址。远程节点必须与设备框架字符串描述符中声明的节点相同。

CDC-ECM 类中的内置 API 可用于通过两种方式传输数据，但由于用户应用程序将通过 NetX 与 USB 以太网设备进行通信，因此它们对应用程序是隐藏的。

USBX CDC-ECM 类与 Azure RTOS NetX 网络堆栈密切相关。使用 NetX 和 USBX CDC-ECM 类的应用程序将以普通方式激活 NetX 网络堆栈，但还需要激活 USB 网络堆栈，如下所示。

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

/* Perform the initialization of the network driver. This will initialize the USBX network layer.*/
ux_network_driver_init();
```

USB 网络堆栈只需激活一次，且不特定于 CDCECM，但需要 NetX 服务的任何 USB 类都需要此堆栈。

CDC-ECM 类将由 MAC OS 和 Linux 主机识别。但是，Microsoft Windows 不提供任何驱动程序来识别本地 CDC-ECM。对于 Windows 平台而言，确实存在一些商业产品并且它们有自己的 .inf 文件。但需要修改此文件（方法与 CDC-ACM inf 模板相同），以匹配 USB 网络设备的 PID/VID。

USB 设备 HID 类

USB 设备 HID 类允许 USB 主机系统使用特定的 HID 客户端功能连接到 HID 设备。

与主机端相比，USBX HID 设备类相对简单。它与设备及其 HID 描述符的行为密切相关。

任何 HID 客户端都需要首先定义一个 HID 设备框架，如以下示例所示。

```
UCHAR device_framework_full_speed[] = {
    /* Device descriptor */
    0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x08,
    0x81, 0x0A, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x01,

    /* Configuration descriptor */
    0x09, 0x02, 0x22, 0x00, 0x01, 0x01, 0x00, 0xc0, 0x32,

    /* Interface descriptor */
    0x09, 0x04, 0x00, 0x00, 0x01, 0x03, 0x00, 0x00, 0x00,

    /* HID descriptor */
    0x09, 0x21, 0x10, 0x01, 0x21, 0x01, 0x22, 0x3f, 0x00,

    /* Endpoint descriptor (Interrupt) */
    0x07, 0x05, 0x81, 0x03, 0x08, 0x00, 0x08

};
```

HID 框架包含一个接口描述符，用于描述 HID 类和 HID 设备子类。HID 接口可以是独立的类，也可以是复合设备的一部分。

目前，USBX HID 类不支持多个报告 ID，因为大多数应用程序仅需要一个 ID（ID 为零）。如果需要使用多个报告 ID，请联系我们。

可使用 USB 键盘初始化 HID 类，如以下所示。

```
/* Initialize the hid class parameters for a keyboard. */
hid_parameter.ux_device_class_hid_parameter_report_address = hid_keyboard_report;
hid_parameter.ux_device_class_hid_parameter_report_length = HID_KEYBOARD_REPORT_LENGTH;
hid_parameter.ux_device_class_hid_parameter_callback = tx_demo_thread_hid_callback;
hid_parameter.ux_device_class_hid_parameter_report_id = 0;

/* Initialize the device hid class. The class is connected with interface 0 */

status = ux_device_stack_class_register(_ux_system_slave_class_hid_name,
    ux_device_class_hid_entry, 1,0,(VOID *)&hid_parameter);
if (status!=UX_SUCCESS)
    return;
```

应用程序需要将 HID 报告描述符及其长度传递给 HID 类。报告描述符是描述设备的项的集合。有关 HID 语法的详细信息，请参阅 HID USB 类规范。

除报告描述符外，应用程序在发生 HID 事件时还会传递回调。

USBX HID 类支持主机中的以下标准 HID 命令。

‘‘‘‘	“‘”	‘‘
UX_DEVICE_CLASS_HID_COMMAND_GET_REPORT	0x01	从设备获取报告
UX_DEVICE_CLASS_HID_COMMAND_GET_IDLE	0x02	获取中断终结点的空闲频率
UX_DEVICE_CLASS_HID_COMMAND_GET_PROTOCOL	0x03	获取设备上运行的协议
UX_DEVICE_CLASS_HID_COMMAND_SET_REPORT	0x09	为设备设置报告
UX_DEVICE_CLASS_HID_COMMAND_SET_IDLE	0x0a	设置中断终结点的空闲频率
UX_DEVICE_CLASS_HID_COMMAND_SET_PROTOCOL	0x0b	获取设备上运行的协议

获取和设置报告是 HID 最常用的命令，用于在主机和设备之间来回传输数据。最常见的情况是，主机在控制终结点上发送数据，但也可以在中断终结点上或者通过发出 GET_REPORT 命令来接收数据以在控制终结点上提取数据。

HID 类可以通过使用 ux_device_class_hid_event_set 函数将数据发送回中断终结点上的主机。

ux_device_class_hid_event_set

为 HID 类设置事件

原型

```
UINT ux_device_class_hid_event_set(
    UX_SLAVE_CLASS_HID *hid,
    UX_SLAVE_CLASS_HID_EVENT *hid_event);
```

说明

应用程序需要将一个 HID 事件发送回主机时，将调用此函数。该函数不会阻塞，它只是把报告放入循环队列，然后返回到应用程序。

参数

- **hid**：指向 hid 类实例的指针。
- **hid_event**：指向 hid 事件的结构的指针。

返回值

- **UX_SUCCESS** (0x00) 此操作成功。
- **UX_ERROR** (0xFF) 错误，循环队列中无可用空间。

示例

```
/* Insert a key into the keyboard event. Length is fixed to 8. */
hid_event.ux_device_class_hid_event_length = 8;

/* First byte is a modifier byte. */
hid_event.ux_device_class_hid_event_buffer[0] = 0;

/* Second byte is reserved. */
hid_event.ux_device_class_hid_event_buffer[1] = 0;

/* The 6 next bytes are keys. We only have one key here. */
hid_event.ux_device_class_hid_event_buffer[2] = key;

/* Set the keyboard event. */
ux_device_class_hid_event_set(hid, &hid_event);
```

在初始化 HID 类时定义的回调将执行与发送事件相反的任务。它获取由主机发送的事件作为输入。回调的原型如下所示。

hid_callback

从 HID 类获取事件

原型

```
UINT hid_callback(
    UX_SLAVE_CLASS_HID *hid,
    UX_SLAVE_CLASS_HID_EVENT *hid_event);
```

说明

主机向应用程序发送 HID 报告时，将调用此函数。

参数

- **hid**：指向 hid 类实例的指针。
- **hid_event**：指向 hid 事件的结构的指针。

示例

以下示例演示了如何解释 HID 键盘事件：

```
UINT tx_demo_thread_hid_callback(UX_SLAVE_CLASS_HID *hid, UX_SLAVE_CLASS_HID_EVENT *hid_event
{
/* There was an event. Analyze it. Is it NUM LOCK ? */

if (hid_event -\ux_device_class_hid_event_buffer[0] & HID_NUM_LOCK_MASK)
    /* Set the Num lock flag. */
    num_lock_flag = UX_TRUE;
else
    /* Reset the Num lock flag. */
    num_lock_flag = UX_FALSE;

/* There was an event. Analyze it. Is it CAPS LOCK ? */

if (hid_event -\ux_device_class_hid_event_buffer[0] & HID_CAPS_LOCK_MASK)
    /* Set the Caps lock flag. */
    caps_lock_flag = UX_TRUE;
else
    /* Reset the Caps lock flag. */
    caps_lock_flag = UX_FALSE;
}
```

第 1 章 - USBX 设备堆栈用户指南补充简介

2021/4/29 •

本文档是对 USBX 设备堆栈用户指南的补充。它包含主要用户指南中未包括的未认证 USBX 设备类的文档。

组织

- 第 1 章包含 USBX 简介
- **第 2 章**：USBX 主机类 API
- **第 3 章**：USBX DPUMP 类注意事项
- **第 4 章**：USBX Pictbridge 实现
- **第 5 章**：USBX OTG

第 2 章 - USBX 设备类注意事项

2021/5/1 •

USB 设备 RNDIS 类

USB 主机系统可以通过 USB 设备 RNDIS 类来与充当以太网设备的设备通信。此类基于 Microsoft 专有的实现，并特定于 Windows 平台。

RNDIS 合规的设备框架需要由设备堆栈声明。下面提供了一个示例。

```
unsigned char device_framework_full_speed[] = {
    /* VID: 0x04b4
    PID: 0x1127
    */

    /* Device Descriptor */
    0x12, /* bLength */
    0x01, /* bDescriptorType */
    0x10, 0x01, /* bcdUSB */
    0x02, /* bDeviceClass - CDC */
    0x00, /* bDeviceSubClass */
    0x00, /* bDeviceProtocol */
    0x40, /* bMaxPacketSize0 */
    0xb4, 0x04, /* idVendor */
    0x27, 0x11, /* idProduct */
    0x00, 0x01, /* bcdDevice */
    0x01, /* iManufacturer */
    0x02, /* iProduct */
    0x03, /* iSerialNumber */
    0x01, /* bNumConfigurations */

    /* Configuration Descriptor */
    0x09, /* bLength */
    0x02, /* bDescriptorType */
    0x38, 0x00, /* wTotalLength */
    0x02, /* bNumInterfaces */
    0x01, /* bConfigurationValue */
    0x00, /* iConfiguration */
    0x40, /* bmAttributes - Self-powered */
    0x00, /* bMaxPower */

    /* Interface Association Descriptor */
    0x08, /* bLength */
    0x0b, /* bDescriptorType */
    0x00, /* bFirstInterface */
    0x02, /* bInterfaceCount */
    0x02, /* bFunctionClass - CDC - Communication */
    0xff, /* bFunctionSubClass - Vendor Defined – In this case, RNDIS */
    0x00, /* bFunctionProtocol - No class specific protocol required */
    0x00, /* iFunction */

    /* Interface Descriptor */
    0x09, /* bLength */
    0x04, /* bDescriptorType */
    0x00, /* bInterfaceNumber */
    0x00, /* bAlternateSetting */
    0x01, /* bNumEndpoints */
    0x02, /* bInterfaceClass - CDC - Communication */
    0xff, /* bInterfaceSubClass - Vendor Defined – In this case, RNDIS */
    0x00, /* bInterfaceProtocol - No class specific protocol required */
    0x00, /* iInterface */
```

```
    /* Endpoint Descriptor */
    0x07, /* bLength */
    0x05, /* bDescriptorType */
    0x83, /* bEndpointAddress */
    0x03, /* bmAttributes - Interrupt */
    0x08, 0x00, /* wMaxPacketSize */
    0xff, /* bInterval */

    /* Interface Descriptor */
    0x09, /* bLength */
    0x04, /* bDescriptorType */
    0x01, /* bInterfaceNumber */
    0x00, /* bAlternateSetting */
    0x02, /* bNumEndpoints */
    0x0a, /* bInterfaceClass - CDC - Data */
    0x00, /* bInterfaceSubClass - Should be 0x00 */
    0x00, /* bInterfaceProtocol - No class specific protocol required */
    0x00, /* iInterface */

    /* Endpoint Descriptor */
    0x07, /* bLength */
    0x05, /* bDescriptorType */
    0x02, /* bEndpointAddress */
    0x02, /* bmAttributes - Bulk */
    0x40, 0x00, /* wMaxPacketSize */
    0x00, /* bInterval */

    /* Endpoint Descriptor */
    0x07, /* bLength */
    0x05, /* bDescriptorType */
    0x81, /* bEndpointAddress */
    0x02, /* bmAttributes - Bulk */
    0x40, 0x00, /* wMaxPacketSize */
    0x00, /* bInterval */
};
```

RNDIS 类对 CDC-ACM 和 CDC-ECM 使用非常类似的设备描述符方法，此外还需要 IAD 描述符。有关设备描述符的定义和要求，请参阅 CDC-ACM 类。

RNDIS 类的激活方式如下所示。

```
/* Set the parameters for callback when insertion/extraction of a CDC device. Set to NULL.*/

parameter.ux_slave_class_rndis_instance_activate = UX_NULL;
parameter.ux_slave_class_rndis_instance_deactivate = UX_NULL;

/* Define a local NODE ID. */

parameter.ux_slave_class_rndis_parameter_local_node_id[0] = 0x00;
parameter.ux_slave_class_rndis_parameter_local_node_id[1] = 0x1e;
parameter.ux_slave_class_rndis_parameter_local_node_id[2] = 0x58;
parameter.ux_slave_class_rndis_parameter_local_node_id[3] = 0x41;
parameter.ux_slave_class_rndis_parameter_local_node_id[4] = 0xb8;
parameter.ux_slave_class_rndis_parameter_local_node_id[5] = 0x78;

/* Define a remote NODE ID. */

parameter.ux_slave_class_rndis_parameter_remote_node_id[0] = 0x00;
parameter.ux_slave_class_rndis_parameter_remote_node_id[1] = 0x1e;
parameter.ux_slave_class_rndis_parameter_remote_node_id[2] = 0x58;
parameter.ux_slave_class_rndis_parameter_remote_node_id[3] = 0x41;
parameter.ux_slave_class_rndis_parameter_remote_node_id[4] = 0xb8;
parameter.ux_slave_class_rndis_parameter_remote_node_id[5] = 0x79;

/* Set extra parameters used by the RNDIS query command with certain OIDs. */

parameter.ux_slave_class_rndis_parameter_vendor_id = 0x04b4 ;
parameter.ux_slave_class_rndis_parameter_driver_version = 0x1127;
ux_utility_memory_copy(parameter.ux_slave_class_rndis_parameter_vendor_description,
    "ELOGIC RNDIS", 12);

/* Initialize the device rndis class. This class owns both interfaces. */
status = ux_device_stack_class_register(_ux_system_slave_class_rndis_name,
    ux_device_class_rndis_entry, 1,0, &parameter);
```

对于 CDC-ECM，RNDIS 类需要 2 个节点（一个是本地节点，一个是远程节点），但不要求使用字符串描述符来描述远程节点。

不过，由于 Microsoft 专有的消息传递机制，还需要提供一些额外的参数。首先必须传递供应商 ID。同理，需要传递 RNDIS 的驱动程序版本。此外，必须指定供应商字符串。

RNDIS 类具有用于双向传输数据的内置 API，但这些 API 对于应用程序是隐藏的，因为用户应用程序通过 NetX 来与 USB 以太网设备通信。

USBX RNDIS 类与 Azure RTOS NetX 网络堆栈密切相关。使用 NetX 和 USBX RNDIS 类的应用程序将以普通方式激活 NetX 网络堆栈，但此外还需要激活 USB 网络堆栈，如下所示。

```
/* Initialize the NetX system. */

nx_system_initialize();

/* Perform the initialization of the network driver. This will initialize the USBX network layer.*/
ux_network_driver_init();
```

USB 网络堆栈只需激活一次，且不特定于 RNDIS；但是，需要 NetX 服务的任何 USB 类都需要此堆栈。

MAC OS 和 Linux 主机无法识别 RNDIS 类，因为此类特定于 Microsoft 操作系统。在 Windows 平台上，与设备描述符匹配的主机上需有一个 .inf 文件。Microsoft 为 RNDIS 类提供了一个模板，可在 usbx_windows_host_files 目录中找到该模板。对于较新的 Windows 版本，应使用文件 RNDIS_Template.inf。需要对此文件进行修改，以反映设备使用的 PID/VID。将公司和产品注册到 USB-IF 后，该 PID/VID 将特定于最终客户。在 inf 文件中，要修改的字段位于此处。

```
[DeviceList]
%DeviceName%=DriverInstall, USB\\VID_xxxx&PID_yyyy&MI_00

[DeviceList.NTamd64]
%DeviceName%=DriverInstall, USB\\VID_xxxx&PID_yyyy&MI_00
```

在 RNDIS 设备的设备框架中，PID/VID 存储在设备描述符中（请参阅上面声明的设备描述符）

当 USB 主机系统发现 USB RNDIS 设备时，它会装载一个网络接口，然后，可以配合网络协议堆栈使用该设备。有关参考信息，请参阅"主机操作系统"。

USB 设备 DFU 类

USB 主机系统可以通过 USB 设备 DFU 类基于主机应用程序更新设备固件。DFU 类是一个 USB-IF 标准类。

USBX DFU 类相对较为简单。它的设备描述符只需包含控制终结点，此外不需要包含其他任何内容。大多数情况下，此类将嵌入到 USB 复合设备中。设备可以是任何设备，例如存储设备或通信设备；添加的 DFU 接口可让主机知道设备能够动态更新其固件。

DFU 类的工作方式分为 3 个步骤。首先，设备像往常一样使用导出的类进行装载。主机（Windows 或 Linux）上的应用程序将执行 DFU 类，并发送一个将设备重置为 DFU 模式的请求。该设备将在总线中消失较短的一段时间（此时间足以让主机和设备检测到 RESET 序列），重启后，该设备将以独占方式处于 DFU 模式，等待主机应用程序发送固件升级。完成固件升级后，主机应用程序将重置设备，重新枚举后，该设备将恢复正常运行并使用新固件。

DFU 设备框架如下所示。

```
UCHAR device_framework_full_speed[] = {

    /* Device descriptor */
    0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x40,
    0x99, 0x99, 0x00, 0x00, 0x00, 0x00, 0x01, 0x02,
    0x03, 0x01,

    /* Configuration descriptor */
    0x09, 0x02, 0x1b, 0x00, 0x01, 0x01, 0x00, 0xc0,
    0x32,

    /* Interface descriptor for DFU. */
    0x09, 0x04, 0x00, 0x00, 0x00, 0xFE, 0x01, 0x01, 0x00,

    /* Functional descriptor for DFU. */
    0x09, 0x21, 0x0f, 0xE8, 0x03, 0x40, 0x00, 0x00,
    0x01,

};
```

在此示例中，DFU 描述符不与任何其他类相关联。它有一个简单的接口描述符，并且其上未附加其他终结点。有一个功能描述符描述设备的 DFU 功能细节。

下面是 DFU 功能的描述。

	OFFSET		TYPE	

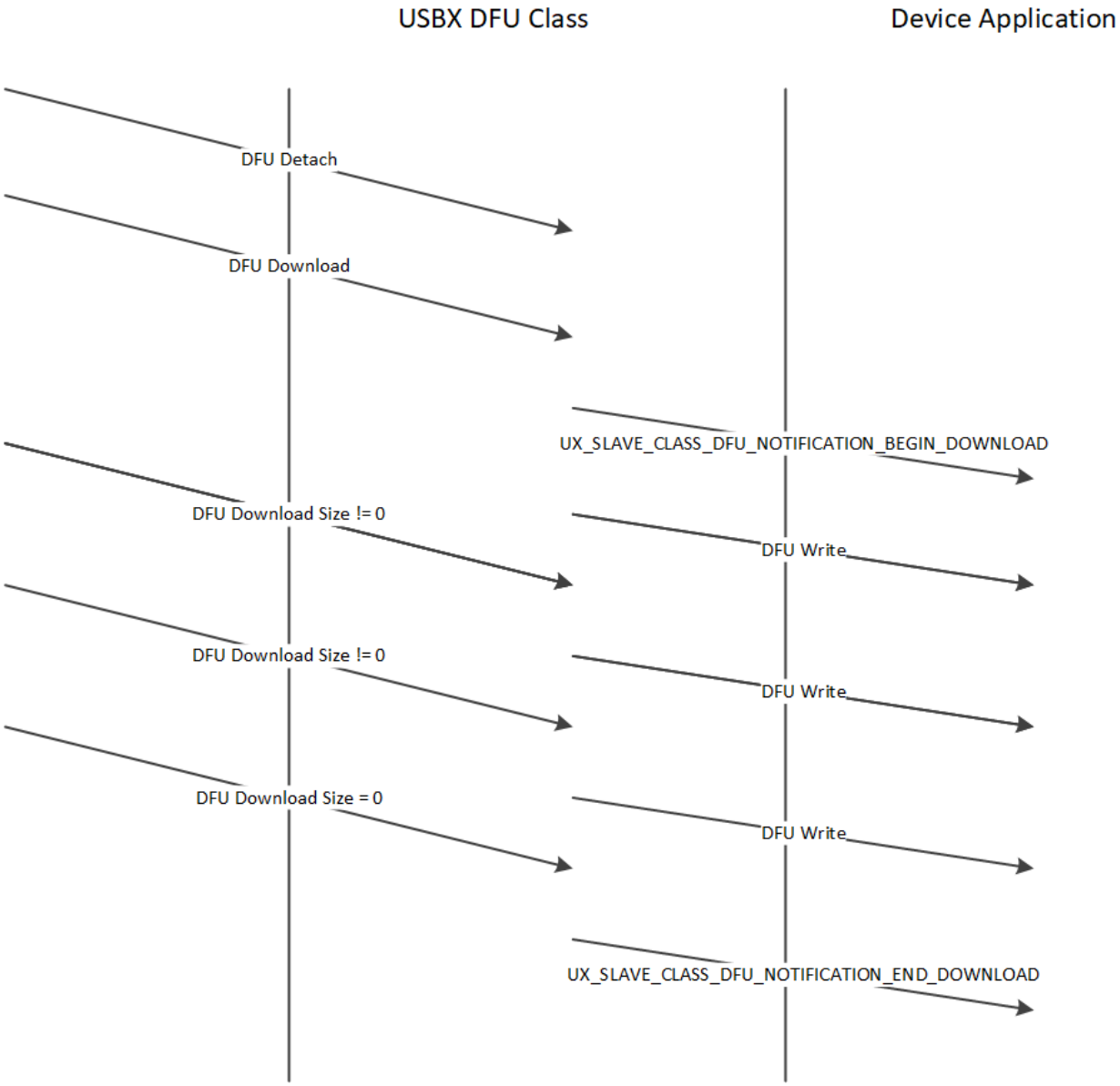
𝐼𝐼	OFFSET	𝐼𝐼	TYPE	𝐼𝐼
bmAttributes	2	1	位域	位 3：当设备收到 DFU_DETACH 请求时，将执行总线拆离-附加序列。主机不得发出 USB 重置命令。(bitWillDetach) 0 = 否，1 = 是。位 2：设备在“具体化”阶段之后可以通过 USB 进行通信。(bitManifestationTolerant) 0 = 否，必须看到总线重置，1 = 是。位 1：支持上传。(bitCanUpload) 0 = 否，1 = 是。位 0：支持下载。(bitCanDnload) 0 = 否，1 = 是
wDetachTimeOut	3	2	数字	设备在收到 DFU_DETACH 请求后等待的时间，以毫秒为单位。如果此时间已消逝但未发生 USB 重置，则设备将终止“重新配置”阶段并恢复正常运行。这表示设备可以等待的最长时间（根据其计时器等）。USBX 将此值设置为 1000 毫秒。
wTransferSize	5	2	数字	设备在每个控制-写入操作中可以接受的最大字节数。USBX 将此值设置为 64 字节。

DFU 类的声明如下：

```
/* Store the DFU parameters. */

dfu_parameter.ux_slave_class_dfu_parameter_instance_activate =
    tx_demo_thread_dfu_activate;

dfu_parameter.ux_slave_class_dfu_parameter_instance_deactivate =
    tx_demo_thread_dfu_deactivate;

dfu_parameter.ux_slave_class_dfu_parameter_read =
    tx_demo_thread_dfu_read;

dfu_parameter.ux_slave_class_dfu_parameter_write =
    tx_demo_thread_dfu_write;

dfu_parameter.ux_slave_class_dfu_parameter_get_status =
    tx_demo_thread_dfu_get_status;

dfu_parameter.ux_slave_class_dfu_parameter_notify =
    tx_demo_thread_dfu_notify;

dfu_parameter.ux_slave_class_dfu_parameter_framework =
    device_framework_dfu;

dfu_parameter.ux_slave_class_dfu_parameter_framework_length =
    DEVICE_FRAMEWORK_LENGTH_DFU;

/* Initialize the device dfu class. The class is connected with interface
1 on configuration 1. */
status = ux_device_stack_class_register(_ux_system_slave_class_dfu_name,
    ux_device_class_dfu_entry, 1, 0,
    (VOID *)&dfu_parameter);

if (status!=UX_SUCCESS) return;
```

DFU 类需要使用特定于目标的设备固件应用程序。因此，它定义了多次回调来读取和写入固件块，并从固件更新应用程序获取状态。DFU 类还具有一个通知回调函数，用于在固件传输开始和结束时告知应用程序。

下面是典型 DFU 应用程序流的说明。



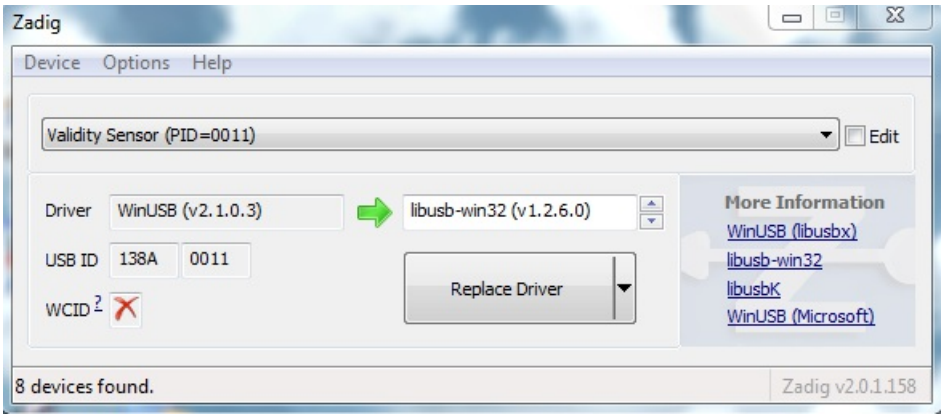
DFU 类的主要挑战在于如何使主机上的适当应用程序执行固件下载。Microsoft 或 USB-IF 不提供这种应用程序。有一些共享软件，它们可以在 Linux 上非常合理地运行，但在 Windows 上其运行合理程度要低一些。

在 Linux 上，用户可以使用以下链接中提供的 dfu-util：https://wiki.openmoko.org/wiki/Dfu-util。在以下链接中也可以找到有关 dfu-util 的大量信息：https://www.libusb.org/wiki/windows_backend

DFU 的 Linux 实现在主机与设备之间正确执行重置序列，因此设备无需执行此操作。Linux 可以接受 bmAttributes bitWillDetach 为 0。另一方面，Windows 要求设备执行重置。

在 Windows 上，USB 注册表必须能够将 USB 设备与其 PID/VID 和 USB 库相关联，然后，DFU 应用程序又可以使用该库。可以使用以下链接中提供的免费实用工具 Zadig 轻松实现此目的：https://sourceforge.net/projects/libwdi/files/zadig/。

首次运行 Zadig 时会显示以下屏幕：



在设备列表中找到你的设备，并将其与 libusb Windows 驱动程序相关联。这会将设备的 PID/VID 与 DFU 实用工具使用的 Windows USB 库绑定在一起。

若要运行 DFU 命令，只需将压缩的 DFU 实用工具解压缩到某个目录，并确保 libusb dll 也在同一目录中。必须在 DOSBox 中的命令行下运行 DFU 实用工具。

首先，键入命令 dfu-util -l 确定是否列出了该设备。如果未列出，请运行 Zadig 来确保设备将被列出并与 USB 库相关联。应会看到如下所示的屏幕：

```
C:\usb specs\DFU\dfu-util-0.6&gt;dfu-util -l dfu-util 0.6

Copyright 2005-2008 Weston Schmidt, Harald Welte and OpenMoko Inc.
Copyright 2010-2012 Tormod Volden and Stefan Schmidt
This program is Free Software and has ABSOLUTELY NO WARRANTY
Found Runtime: [0a5c:21bc] devnum=0, cfg=1, intf=3, alt=0, name="UNDEFINED"
```

下一步是准备要下载的文件。USBX DFU 类不会对此文件执行任何验证，并且不知道此文件的内部格式。此固件文件与目标密切相关，但与 DFU 和 USBX 不相关。

然后，可以键入以下命令来指示 dfu-util 发送文件：

```
dfu-util –R –t 64 -D file_to_download.hex
```

dfu-util 应显示文件下载过程，直到固件已完全下载。

USB 设备 PIMA 类(PTP 响应方)

USB 主机系统(发起方)可以通过 USB 设备 PIMA 类连接到

PIMA 设备(响应方)以传输媒体文件。USBX PIMA 类遵从 USB-IF PIMA 15740 类(也称为 PTP(图片传输协议)类)的规范。

USBX 设备端 PIMA 类支持以下操作。

‖‖‖	‖	‖
UX_DEVICE_CLASS_PIMA_OC_GET_DEVICE_INFO	0x1001	获取设备支持的操作和事件
UX_DEVICE_CLASS_PIMA_OC_OPEN_SESSION	0x1002	打开主机与设备之间的会话
UX_DEVICE_CLASS_PIMA_OC_CLOSE_SESSION	0x1003	关闭主机与设备之间的会话

[illegible]	[illegible]	[illegible]
UX_DEVICE_CLASS_PIMA_OC_GET_STORAGE_IDS	0x1004	返回设备的存储 ID。USBX PIMA 仅使用一个存储 ID
UX_DEVICE_CLASS_PIMA_OC_GET_STORAGE_INFO	0x1005	返回有关存储对象的信息，例如最大容量和可用空间
UX_DEVICE_CLASS_PIMA_OC_GET_NUM_OBJECTS	0x1006	返回存储设备中包含的对象数
UX_DEVICE_CLASS_PIMA_OC_GET_OBJECT_HANDLES	0x1007	返回存储设备上对象的句柄数组
UX_DEVICE_CLASS_PIMA_OC_GET_OBJECT_INFO	0x1008	返回有关对象的信息，例如对象的名称、创建日期和修改日期
UX_DEVICE_CLASS_PIMA_OC_GET_OBJECT	0x1009	返回与特定对象相关的数据
UX_DEVICE_CLASS_PIMA_OC_GET_THUMB	0x100A	发送有关对象的缩略图（如果有）
UX_DEVICE_CLASS_PIMA_OC_DELETE_OBJECT	0x100B	删除媒体上的对象
UX_DEVICE_CLASS_PIMA_OC_SEND_OBJECT_INFO	0x100C	将有关对象的信息发送到设备，以便在媒体上创建该对象
UX_DEVICE_CLASS_PIMA_OC_SEND_OBJECT	0x100D	将对象数据发送到设备
UX_DEVICE_CLASS_PIMA_OC_FORMAT_STORE	0x100F	清理设备媒体
UX_DEVICE_CLASS_PIMA_OC_RESET_DEVICE	0x0110	重置目标设备

[illegible]	[illegible]	[illegible]
UX_DEVICE_CLASS_PIMA_EC_CANCEL_TRANSACTION	0x4001	取消当前事务
UX_DEVICE_CLASS_PIMA_EC_OBJECT_ADDED	0x4002	对象已被添加到设备介质中，并且可以被主机检索\。
UX_DEVICE_CLASS_PIMA_EC_OBJECT_REMOVED	0x4003	已从设备媒体中删除对象
UX_DEVICE_CLASS_PIMA_EC_STORE_ADDED	0x4004	已将媒体添加到设备
UX_DEVICE_CLASS_PIMA_EC_STORE_REMOVED	0x4005	已从设备中删除媒体

[[[[	[	[[
UX_DEVICE_CLASS_PIMA_EC_DEVICE_PROP_CHANGED	0x4006	已更改设备属性
UX_DEVICE_CLASS_PIMA_EC_OBJECT_INFO_CHANGED	0x4007	已更改对象信息
UX_DEVICE_CLASS_PIMA_EC_DEVICE_INFO_CHANGE	0x4008	已更改设备
UX_DEVICE_CLASS_PIMA_EC_REQUEST_OBJECT_TRANSFER	0x4009	设备请求从主机传输对象
UX_DEVICE_CLASS_PIMA_EC_STORE_FULL	0x400A	设备报告媒体已满
UX_DEVICE_CLASS_PIMA_EC_DEVICE_RESET	0x400B	设备报告它已重置
UX_DEVICE_CLASS_PIMA_EC_STORAGE_INFO_CHANGED	0x400C	已更改设备上的存储信息
UX_DEVICE_CLASS_PIMA_EC_CAPTURE_COMPLETE	0x400D	已完成捕获

USBX PIMA 设备类使用 TX 线程侦听来自主机的 PIMA 命令。

PIMA 命令由命令块、数据块和状态阶段组成。

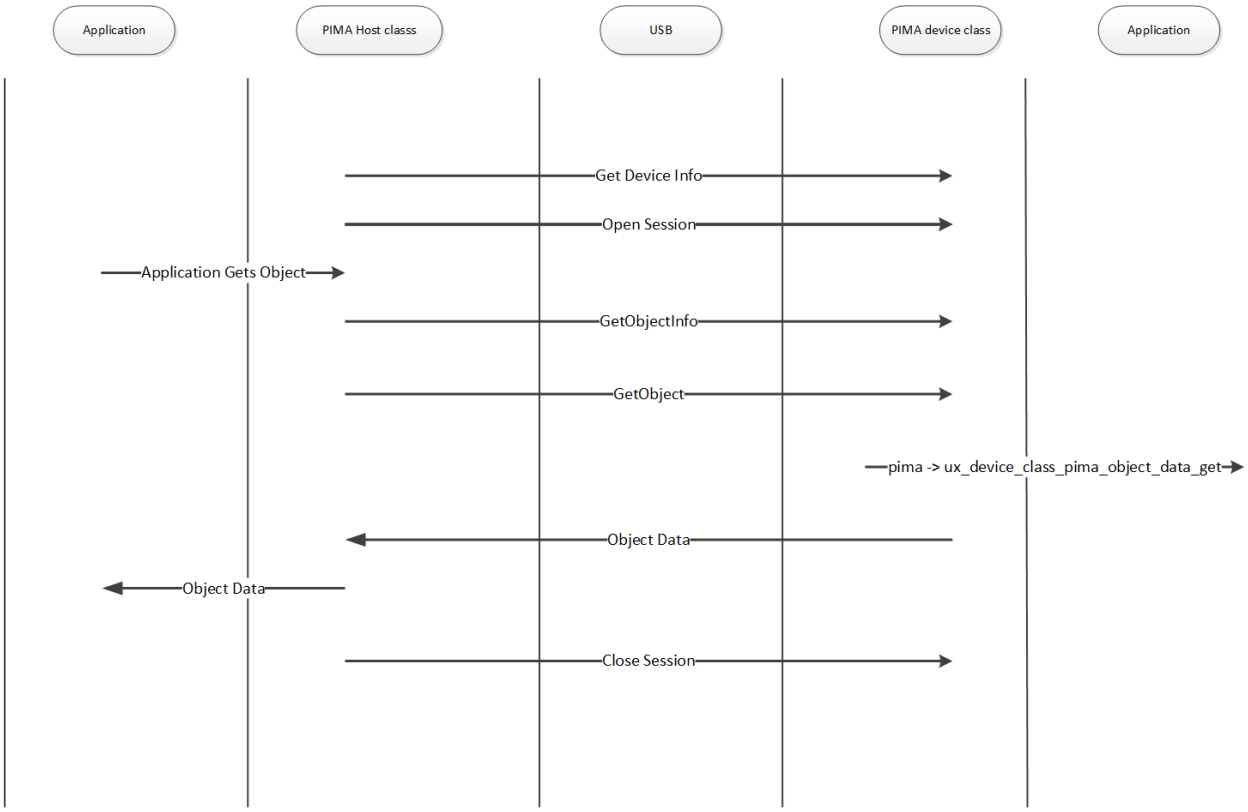
函数 ux_device_class_pima_thread 向堆栈发送请求，以从主机端接收 PIMA 命令。将解码 PIMA 命令并验证其内容。如果命令块有效，则它会分支到相应的命令处理程序。

大多数 PIMA 命令仅在主机已打开会话时才能执行。只有命令 UX_DEVICE_CLASS_PIMA_OC_GET_DEVICE_INFO 例外。使用 USBX PIMA 实现时，在任意时间只能在发起方与响应方之间打开一个会话。单个会话中的所有事务将会阻塞，在前一个事务完成之前无法开始新的事务。

PIMA 事务由 3 个阶段组成：命令阶段、可选的数据阶段，以及响应阶段。如果存在数据阶段，此阶段只能单向进行。

发起方始终会确定 PIMA 操作流，但响应方可以向发起方反向发起事件，以告知在会话期间发生的状态更改。

下图显示了主机与 PIMA 设备类之间的数据对象传输。



PIMA 设备类的初始化

在初始化期间，PIMA 设备类需要应用程序提供的一些参数。

以下参数描述设备和存储信息。

- `ux_device_class_pima_manufacturer`
- `ux_device_class_pima_model`
- `ux_device_class_pima_device_version`
- `ux_device_class_pima_serial_number`
- `ux_device_class_pima_storage_id`
- `ux_device_class_pima_storage_type`
- `ux_device_class_pima_storage_file_system_type`
- `ux_device_class_pima_storage_access_capability`
- `ux_device_class_pima_storage_max_capacity_low`
- `ux_device_class_pima_storage_max_capacity_high`
- `ux_device_class_pima_storage_free_space_low`
- `ux_device_class_pima_storage_free_space_high`
- `ux_device_class_pima_storage_free_space_image`
- `ux_device_class_pima_storage_description`
- `ux_device_class_pima_storage_volume_label`

PIMA 类还要求在应用程序中注册回调，以告知应用程序发生了某些事件，或者从/向本地媒体检索/存储数据。回调如下所示。

- `ux_device_class_pima_object_number_get`
- `ux_device_class_pima_object_handles_get`
- `ux_device_class_pima_object_info_get`
- `ux_device_class_pima_object_data_get`

- `ux_device_class_pima_object_info_send`
- `ux_device_class_pima_object_data_send`
- `ux_device_class_pima_object_delete`

以下示例演示如何初始化 PIMA 的客户端。此示例使用 Pictbridge 作为 PIMA 的客户端。

```
/* Initialize the first XML object valid in the pictbridge instance.

Initialize the handle, type and file name.

The storage handle and the object handle have a fixed value of 1 in our implementation. */

object_info = pictbridge -> ux_pictbridge_object_client;

object_info -> ux_device_class_pima_object_format = UX_DEVICE_CLASS_PIMA_OFC_SCRIPT;
object_info -> ux_device_class_pima_object_storage_id = 1;
object_info -> ux_device_class_pima_object_handle_id = 2;

ux_utility_string_to_unicode(_ux_pictbridge_ddiscovery_name,
    object_info -> ux_device_class_pima_object_filename);

/* Initialize the head and tail of the notification round robin buffers.
   At first, the head and tail are pointing to the beginning of the array.
*/

pictbridge -> ux_pictbridge_event_array_head =
    pictbridge -> ux_pictbridge_event_array;

pictbridge -> ux_pictbridge_event_array_tail =
    pictbridge -> ux_pictbridge_event_array;

pictbridge -> ux_pictbridge_event_array_end =
    pictbridge -> ux_pictbridge_event_array +
    UX_PICTBRIDGE_MAX_EVENT_NUMBER;

/* Initialialize the pima device parameter. */
pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_manufacturer =
    pictbridge -> ux_pictbridge_dpslocal.ux_pictbridge_devinfo_vendor_name;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_model =
    pictbridge -> ux_pictbridge_dpslocal.ux_pictbridge_devinfo_product_name;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_serial_number =
    pictbridge -> ux_pictbridge_dpslocal.ux_pictbridge_devinfo_serial_no;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_storage_id = 1;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_storage_type =
    UX_DEVICE_CLASS_PIMA_STC_FIXED_RAM;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_storage_file_system_type =
    UX_DEVICE_CLASS_PIMA_FSTC_GENERIC_FLAT;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_storage_access_capability =
    UX_DEVICE_CLASS_PIMA_AC_READ_WRITE;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_storage_max_capacity_low =
    pictbridge -> ux_pictbridge_dpslocal.ux_pictbridge_devinfo_storage_size;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_storage_max_capacity_high = 0;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_storage_free_space_low =
    pictbridge -> ux_pictbridge_dpslocal.ux_pictbridge_devinfo_storage_size;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_storage_free_space_high = 0;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_storage_free_space_image = 0;
```

```
pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_storage_free_space_image = 0;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_storage_description =
    _ux_pictbridge_volume_description;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_storage_volume_label =
    _ux_pictbridge_volume_label;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_object_number_get =
    ux_pictbridge_dpsclient_object_number_get;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_object_handles_get =
    ux_pictbridge_dpsclient_object_handles_get;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_object_info_get =
    ux_pictbridge_dpsclient_object_info_get;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_object_data_get =
    ux_pictbridge_dpsclient_object_data_get;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_object_info_send =
    ux_pictbridge_dpsclient_object_info_send;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_object_data_send =
    ux_pictbridge_dpsclient_object_data_send;

pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_object_delete =
    ux_pictbridge_dpsclient_object_delete;

/* Store the instance owner. */
pictbridge -> ux_pictbridge_pima_parameter.ux_device_class_pima_parameter_application =
    (VOID *) pictbridge;

/* Initialize the device pima class. The class is connected with interface 0 */

status = ux_device_stack_class_register(_ux_system_slave_class_pima_name,
    ux_device_class_pima_entry, 1, 0, (VOID *)&pictbridge -> ux_pictbridge_pima_parameter);

/* Check status. */
if (status != UX_SUCCESS)
```

ux_device_class_pima_object_add

添加对象并将事件发送到主机

原型

```
UINT ux_device_class_pima_object_add(
    UX_SLAVE_CLASS_PIMA *pima,
    ULONG object_handle);
```

说明

当 PIMA 类需要添加对象并告知主机时，将调用此函数。

参数

- pima：指向 PIMA 类实例的指针
- object_handle：对象的句柄。

示例

```
/* Send the notification to the host that an object has been added. */

status = ux_device_class_pima_object_add(pima, UX_PICTBRIDGE_OBJECT_HANDLE_CLIENT_REQUEST);
```

ux_device_class_pima_object_number_get

从应用程序获取对象数

原型

```
UINT ux_device_class_pima_object_number_get(
    UX_SLAVE_CLASS_PIMA *pima,
    ULONG *object_number);
```

说明

当 PIMA 类需要检索本地系统中的对象数并将其发回到主机时，将调用此函数。

参数

- pima：指向 PIMA 类实例的指针
- object_number：要返回的对象数的地址

示例

```
UINT ux_pictbridge_dpsclient_object_number_get(UX_SLAVE_CLASS_PIMA *pima, ULONG *number_objects)
{
    /* We force the number of objects to be 1 only here. This will be the XML scripts. */
    *number_objects = 1;
    return(UX_SUCCESS);
}
```

ux_device_class_pima_object_handles_get

返回对象句柄数组

原型

```
UINT **ux_device_class_pima_object_handles_get**(
    UX_SLAVE_CLASS_PIMA_STRUCT *pima,
    ULONG object_handles_format_code,
    ULONG object_handles_association,
    ULONG *object_handles_array,
    ULONG object_handles_max_number);
```

说明

当 PIMA 类需要检索本地系统中的对象句柄数组并将其发回到主机时，将调用此函数。

参数

- pima：指向 PIMA 类实例的指针。
- object_handles_format_code：句柄的格式代码
- object_handles_association：对象关联代码
- object_handle_array：将句柄存储到的地址
- object_handles_max_number：数组中的最大句柄数

示例

```
UINT ux_pictbridge_dpsclient_object_handles_get(UX_SLAVE_CLASS_PIMA *pima,
    ULONG object_handles_format_code,
    ULONG object_handles_association,
    ULONG *object_handles_array,
    ULONG object_handles_max_number)
{

    UX_PICTBRIDGE *pictbridge;
    UX_SLAVE_CLASS_PIMA_OBJECT *object_info;

    /* Get the pointer to the Pictbridge instance. */
    pictbridge = (UX_PICTBRIDGE *) pima -> ux_device_class_pima_application;

    /* Set the pima pointer to the pictbridge instance. */
    pictbridge -> ux_pictbridge_pima = (VOID *) pima;

    /* We say we have one object but the caller might specify different format code and associations. */
    object_info = pictbridge -> ux_pictbridge_object_client;

    /* Insert in the array the number of found handles so far: 0. */
    ux_utility_long_put((UCHAR *)object_handles_array, 0);

    /* Check the type demanded. */

    if (object_handles_format_code == 0 || object_handles_format_code ==
        0xFFFFFFFF || object_info -> ux_device_class_pima_object_format ==
        object_handles_format_code)
    {
        /* Insert in the array the number of found handles. This handle is for the client XML script. */
        ux_utility_long_put((UCHAR *)object_handles_array, 1);

        /* Adjust the array to point after the number of elements. */
        object_handles_array++;

        /* We have a candicate. Store the handle. */
        ux_utility_long_put((UCHAR *)object_handles_array, object_info ->
            ux_device_class_pima_object_handle_id);
    }

    return(UX_SUCCESS);
}
```

ux_device_class_pima_object_info_get

返回对象信息

原型

```
UINT ux_device_class_pima_object_info_get(
    struct UX_SLAVE_CLASS_PIMA_STRUCT *pima,
    ULONG object_handle,
    UX_SLAVE_CLASS_PIMA_OBJECT **object);
```

说明

当 PIMA 类需要检索本地系统中的对象句柄数组并将其发回到主机时，将调用此函数。

参数

- pima：指向 PIMA 类实例的指针。
- object_handles：对象的句柄
- object：对象指针地址

示例

```
UINT ux_pictbridge_dpsclient_object_info_get(UX_SLAVE_CLASS_PIMA *pima,
    ULONG object_handle, UX_SLAVE_CLASS_PIMA_OBJECT **object)
{
    UX_PICTBRIDGE *pictbridge;
    UX_SLAVE_CLASS_PIMA_OBJECT *object_info;
    /* Get the pointer to the Pictbridge instance. */
    pictbridge = (UX_PICTBRIDGE *)pima -> ux_device_class_pima_application;

    /* Check the object handle. If this is handle 1 or 2 , we need to return the XML script object.
       If the handle is not 1 or 2, this is a JPEG picture or other object to be printed. */
    if ((object_handle == UX_PICTBRIDGE_OBJECT_HANDLE_HOST_RESPONSE) ||
        (object_handle == UX_PICTBRIDGE_OBJECT_HANDLE_CLIENT_REQUEST)) {

        /* Check what XML object is requested. It is either a request script or a response. */
        if (object_handle == UX_PICTBRIDGE_OBJECT_HANDLE_HOST_RESPONSE)
            object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge -> ux_pictbridge_object_host;
        else
            object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
                ux_pictbridge_object_client;
    } else
        /* Get the object info from the job info structure. */
        object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
            ux_pictbridge_jobinfo.ux_pictbridge_jobinfo_object;
    /* Return the pointer to this object. */
    *object = object_info;

    /* We are done. */
    return(UX_SUCCESS);
}
```

ux_device_class_pima_object_data_get

返回对象数据

原型

```
UINT ux_device_class_pima_object_info_get(
    UX_SLAVE_CLASS_PIMA *pima,
    ULONG object_handle,
    UCHAR *object_buffer,
    ULONG object_offset,
    ULONG object_length_requested,
    ULONG *object_actual_length);
```

说明

当 PIMA 类需要检索本地系统中的对象数据并将其发回到主机时，将调用此函数。

参数

- pima：指向 PIMA 类实例的指针。
- object_handle：对象的句柄
- object_buffer：对象缓冲区地址
- object_length_requested：客户端向应用程序请求的对象数据长度
- object_actual_length：应用程序返回的对象数据长度

示例

```
UINT ux_pictbridge_dpsclient_object_data_get(UX_SLAVE_CLASS_PIMA *pima,
    ULONG object_handle, UCHAR *object_buffer, ULONG object_offset,
    ULONG object_length_requested, ULONG *object_actual_length)
```

```
{

    UX_PICTBRIDGE *pictbridge;
    UX_SLAVE_CLASS_PIMA_OBJECT *object_info;
    UCHAR *pima_object_buffer;
    ULONG actual_length;
    UINT status;

    /* Get the pointer to the Pictbridge instance. */
    pictbridge = (UX_PICTBRIDGE *)pima -> ux_device_class_pima_application;

    /* Check the object handle. If this is handle 1 or 2 , we need to return the XML script object.
       If the handle is not 1 or 2, this is a JPEG picture or other object to be printed. */
    if ((object_handle == UX_PICTBRIDGE_OBJECT_HANDLE_HOST_RESPONSE) ||
        (object_handle == UX_PICTBRIDGE_OBJECT_HANDLE_CLIENT_REQUEST))
    {
        /* Check what XML object is requested. It is either a request script or a response. */
        if (object_handle == UX_PICTBRIDGE_OBJECT_HANDLE_HOST_RESPONSE)
            object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
                ux_pictbridge_object_host;
        else
            object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
                ux_pictbridge_object_client;

       /* Is this the corrent handle ? */
       if (object_info -> ux_device_class_pima_object_handle_id == object_handle)
       {
           /* Get the pointer to the object buffer. */
           pima_object_buffer = object_info -> ux_device_class_pima_object_buffer;

           /* Copy the demanded object data portion. */
           ux_utility_memory_copy(object_buffer, pima_object_buffer +
               object_offset, object_length_requested);

           /* Update the length requested. for a demo, we do not do any checking. */
           *object_actual_length = object_length_requested;

           /* What cycle are we in ? */
           if (pictbridge -> ux_pictbridge_host_client_state_machine &
               UX_PICTBRIDGE_STATE_MACHINE_HOST_REQUEST)
            {
                /* Check if we are blocking for a client request. */
                if (pictbridge -> ux_pictbridge_host_client_state_machine &
                    UX_PICTBRIDGE_STATE_MACHINE_CLIENT_REQUEST_PENDING)

                    /* Yes we are pending, send an event to release the pending request. */
                    ux_utility_event_flags_set(&pictbridge ->
                        ux_pictbridge_event_flags_group,
                        UX_PICTBRIDGE_EVENT_FLAG_STATE_MACHINE_READY, TX_OR);

               /* Since we are in host request, this indicates we are done with the cycle. */
               pictbridge -> ux_pictbridge_host_client_state_machine =
                   UX_PICTBRIDGE_STATE_MACHINE_IDLE;

           }

           /* We have copied the requested data. Return OK. */
           return(UX_SUCCESS);

       }
    }
    else
    {

        /* Get the object info from the job info structure. */
        object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
            ux_pictbridge_jobinfo.ux_pictbridge_jobinfo_object;

        /* Obtain the data from the application jobinfo callback. */
```

```
        /* Obtain the data from the application jobinfo callback. */
        status = pictbridge -> ux_pictbridge_jobinfo.
            ux_pictbridge_jobinfo_object_data_read(pictbridge, object_buffer, object_offset,
            object_length_requested, &actual_length);

        /* Save the length returned. */
        *object_actual_length = actual_length;

        /* Return the application status. */
        return(status);

    }

    /* Could not find the handle. */

    return(UX_DEVICE_CLASS_PIMA_RC_INVALID_OBJECT_HANDLE);
}
```

ux_device_class_pima_object_info_send

主机发送对象信息

原型

```
UINT ux_device_class_pima_object_info_send(
    UX_SLAVE_CLASS_PIMA *pima,
    UX_SLAVE_CLASS_PIMA_OBJECT *object,
    ULONG *object_handle);
```

说明

当 PIMA 类需要接收本地系统中的对象信息以便将来进行存储时，将调用此函数。

参数

- pima：指向 PIMA 类实例的指针
- object：指向对象的指针
- object_handle：对象的句柄

示例

```
UINT ux_pictbridge_dpsclient_object_info_send(UX_SLAVE_CLASS_PIMA *pima,
    UX_SLAVE_CLASS_PIMA_OBJECT *object, ULONG *object_handle)
{
    UX_PICTBRIDGE *pictbridge;
    UX_SLAVE_CLASS_PIMA_OBJECT *object_info; UCHAR
    string_discovery_name[UX_PICTBRIDGE_MAX_FILE_NAME_SIZE];

    /* Get the pointer to the Pictbridge instance. */
    pictbridge = (UX_PICTBRIDGE *)pima -> ux_device_class_pima_application;

    /* We only have one object. */
    object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
        ux_pictbridge_object_host;

    /* Copy the demanded object info set. */
    ux_utility_memory_copy(object_info, object,
        UX_SLAVE_CLASS_PIMA_OBJECT_DATA_LENGTH);

    /* Store the object handle. In Pictbridge we only receive XML scripts so the handle is hardwired to 1.
*/
    object_info -> ux_device_class_pima_object_handle_id = 1;
    *object_handle = 1;

    /* Check state machine. If we are in discovery pending mode, check file name of this object. */
```

```
    if (pictbridge -> ux_pictbridge_discovery_state ==
        UX_PICTBRIDGE_DPSCLIENT_DISCOVERY_PENDING)
    {
        /* We are in the discovery mode. Check for file name. It must match
           HDISCVRY.DPS in Unicode mode. */

        /* Check if this is a script. */
        if (object_info -> ux_device_class_pima_object_format ==
            UX_DEVICE_CLASS_PIMA_OFC_SCRIPT)
        {

            /* Yes this is a script. We need to search for the HDISCVRY.DPS file name. Get the file name in
a ascii format. */
            ux_utility_unicode_to_string(object_info ->
                ux_device_class_pima_object_filename,
                string_discovery_name);

            /* Now, compare it to the HDISCVRY.DPS file name. Check length first. */
            if (ux_utility_string_length_get(_ux_pictbridge_hdiscovery_name)
                == ux_utility_string_length_get(string_discovery_name))
            {

                /* So far, the length of name of the files are the same. Compare names now. */
                if(ux_utility_memory_compare(_ux_pictbridge_hdiscovery_name,
                    string_discovery_name,
                    ux_utility_string_length_get(string_discovery_name)) == UX_SUCCESS)
                {
                    /* We are done with discovery of the printer. We can now send notifications when the
camera wants to print an object. */
                    pictbridge -> ux_pictbridge_discovery_state =
                        UX_PICTBRIDGE_DPSCLIENT_DISCOVERY_COMPLETE;

                    /* Set an event flag if the application is listening. */
                    ux_utility_event_flags_set(&pictbridge ->
                        ux_pictbridge_event_flags_group,
                        UX_PICTBRIDGE_EVENT_FLAG_DISCOVERY, TX_OR);

                    /* There is no object during th discovery cycle. */
                    return(UX_SUCCESS);
                }
            }
        }
    }
    /* What cycle are we in ? */
    if (pictbridge -> ux_pictbridge_host_client_state_machine ==
        UX_PICTBRIDGE_STATE_MACHINE_IDLE)

        /* Since we are in idle state, we must have received a request from the host. */
        pictbridge -> ux_pictbridge_host_client_state_machine =
            UX_PICTBRIDGE_STATE_MACHINE_HOST_REQUEST;

    /* We have copied the requested data. Return OK. */
    return(UX_SUCCESS);
}
```

ux_device_class_pima_object_data_send

主机发送对象数据

原型

```
UINT ux_device_class_pima_object_data_send(
    UX_SLAVE_CLASS_PIMA *pima,
    ULONG object_handle,
    ULONG phase,
    UCHAR *object_buffer,
    ULONG object_offset,
    ULONG object_length);
```

说明

当 PIMA 类需要接收本地系统中的对象数据以便进行存储时，将调用此函数。

参数

- pima：指向 PIMA 类实例的指针
- object_handle：对象的句柄
- phase：传输阶段（活动或完成）
- object_buffer：对象缓冲区地址
- object_offset：数据的地址
- object_length：应用程序发送的对象数据长度

示例

```
UINT ux_pictbridge_dpsclient_object_data_send(UX_SLAVE_CLASS_PIMA *pima,
    ULONG object_handle,
    ULONG phase,
    UCHAR *object_buffer,
    ULONG object_offset,
    ULONG object_length)
{
    UINT status;
    UX_PICTBRIDGE *pictbridge;
    UX_SLAVE_CLASS_PIMA_OBJECT *object_info;
    ULONG event_flag;
    UCHAR *pima_object_buffer;

    /* Get the pointer to the Pictbridge instance. */
    pictbridge = (UX_PICTBRIDGE *)pima -> ux_device_class_pima_application;

    /* Get the pointer to the pima object. */
    object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
        ux_pictbridge_object_host;

    /* Is this the corrent handle ? */
    if (object_info -> ux_device_class_pima_object_handle_id == object_handle)
    {
        /* Get the pointer to the object buffer. */
        pima_object_buffer = object_info ->
            ux_device_class_pima_object_buffer;

        /* Check the phase. We should wait for the object to be completed and the response sent back before
parsing the object. */
        if (phase == UX_DEVICE_CLASS_PIMA_OBJECT_TRANSFER_PHASE_ACTIVE)
        {
            /* Copy the demanded object data portion. */
            ux_utility_memory_copy(pima_object_buffer + object_offset,
                object_buffer, object_length);

            /* Save the length of this object. */
            object_info -> ux_device_class_pima_object_length = object_length;

            /* We are not done yet. */
            return(UX_SUCCESS);
        }
        else
        {
            /* Completion of transfer. We are done. */
            return(UX_SUCCESS);
        }
    }
}
```

ux_device_class_pima_object_delete

删除本地对象

原型

```
UINT ux_device_class_pima_object_delete(
    UX_SLAVE_CLASS_PIMA *pima,
    ULONG object_handle);
```

说明

当 PIMA 类需要删除本地存储中的对象时，将调用此函数。

参数

- pima：指向 PIMA 类实例的指针
- object_handle：对象的句柄

示例

```
UINT ux_pictbridge_dpsclient_object_delete(UX_SLAVE_CLASS_PIMA *pima,
    ULONG object_handle)
{
    /* Delete the object pointer by the handle. */

}
```

USB 设备音频类

USB 主机系统可以通过 USB 设备音频类来与充当音频设备的设备通信。此类基于 USB 标准以及 USB 音频类 1.0 或 2.0 标准。

USB 音频合规的设备框架需要由设备堆栈声明。音频 2.0 扬声器的示例如下：

```
unsigned char device_framework_high_speed[] = {

    /* --- Device Descriptor 18 bytes
    0x00 bDeviceClass: Refer to interface
    0x00 bDeviceSubclass: Refer to interface
    0x00 bDeviceProtocol: Refer to interface

    idVendor & idProduct - https://www.linux-usb.org/usb.ids
    */

    /* 0 bLength, bDescriptorType */ 18, 0x01,
    /* 2 bcdUSB : 0x200 (2.00) */ 0x00, 0x02,
    /* 4 bDeviceClass : 0x00 (see interface) */ 0x00,
    /* 5 bDeviceSubClass : 0x00 (see interface) */ 0x00,
    /* 6 bDeviceProtocol : 0x00 (see interface) */ 0x00,
    /* 7 bMaxPacketSize0 */ 0x08,
    /* 8 idVendor, idProduct */ 0x84, 0x84, 0x03, 0x00,
    /* 12 bcdDevice */ 0x00, 0x02,
    /* 14 iManufacturer, iProduct, iSerialNumber */ 0, 0, 0,
    /* 17 bNumConfigurations */ 1,
    /* ---------------- Device Qualifier Descriptor */
    /* 0 bLength, bDescriptorType */ 10, 0x06,
    /* 2 bcdUSB : 0x200 (2.00) */ 0x00,0x02,
    /* 4 bDeviceClass : 0x00 (see interface) */ 0x00,
    /* 5 bDeviceSubClass : 0x00 (see interface) */ 0x00,
    /* 6 bDeviceProtocol : 0x00 (see interface) */ 0x00,
    /* 7 bMaxPacketSize0 */ 8,
    /* 8 bNumConfigurations */ 1,
    /* 9 bReserved */ 0,
    /* --- Configuration Descriptor (9+8+73+55=145, 0x91) */
    /* 0 bLength, bDescriptorType */ 9, 0x02,
    /* 2 wTotalLength */ 145, 0,
    /* 4 bNumInterfaces, bConfigurationValue */ 2, 1,
    /* 6 iConfiguration */ 0,
    /* 7 bmAttributes, bMaxPower */ 0x80, 50,
    /* ----------- Interface Association Descriptor */
    /* 0 bLength, bDescriptorType */ 8, 0x0B,
    /* 2 bFirstInterface, bInterfaceCount */ 0, 2,
    /* 4 bFunctionClass : 0x01 (Audio) */ 0x01,
    /* 5 bFunctionSubClass : 0x00 (UNDEFINED) */ 0x00,
    /* 6 bFunctionProtocol : 0x20 (VERSION_02_00) */ 0x20,
    /* 7 iFunction */ 0,
    /* --- Interface Descriptor #0: Control (9+64=73) */
    /* 0 bLength, bDescriptorType */ 9, 0x04,
    /* 2 bInterfaceNumber, bAlternateSetting */ 0, 0,
```

```
/* 4 bNumEndpoints */ 0,
/* 5 bInterfaceClass : 0x01 (Audio) */ 0x01,
/* 6 bInterfaceSubClass : 0x01 (AudioControl) */ 0x01,
/* 7 bInterfaceProtocol : 0x20 (VERSION_02_00) */ 0x20,
/* 8 iInterface */ 0,
/* --- Audio 2.0 AC Interface Header Descriptor (9+8+17+18+12=64, 0x40) */
/* 0 bLength */ 9,
/* 1 bDescriptorType, bDescriptorSubtype */ 0x24, 0x01,
/* 3 bcdADC */ 0x00, 0x02,
/* 5 bCategory : 0x08 (IO Box) */ 0x08,
/* 6 wTotalLength */ 64, 0,
/* 8 bmControls */ 0x00,
/* -------- Audio 2.0 AC Clock Source Descriptor */
/* 0 bLength */ 8,
/* 1 bDescriptorType, bDescriptorSubtype */ 0x24, 0x0A,
/* 3 bClockID */ 0x10,
/* 4 bmAttributes : 0x05 (Sync|InternalFixedClk) */ 0x05,
/* 5 bmControls : 0x01 (FreqReadOnly) */ 0x01,
/* 6 bAssocTerminal, iClockSource */ 0x00, 0,
/* ------ Audio 2.0 AC Input Terminal Descriptor */
/* 0 bLength */ 17,
/* 1 bDescriptorType, bDescriptorSubtype */ 0x24, 0x02, /* 3 bTerminalID */ 0x04,
/* 4 wTerminalType : 0x0101 (USB Streaming) */ 0x01, 0x01,
/* 6 bAssocTerminal, bCSourceID */ 0x00, 0x10,
/* 8 bNrChannels */ 2,
/* 9 bmChannelConfig */ 0x00, 0x00, 0x00, 0x00,
/* 13 iChannelNames, bmControls, iTerminal */ 0, 0x00, 0x00, 0,
/* -------- Audio 2.0 AC Feature Unit Descriptor */
/* 0 bLength */ 18,
/* 1 bDescriptorType, bDescriptorSubtype */ 0x24, 0x06,
/* 3 bUnitID, bSourceID */ 0x05, 0x04,
/* 5 bmaControls(0) : 0x0F (VolumeRW|MuteRW) */ 0x0F, 0x00, 0x00, 0x00,
/* 9 bmaControls(1) : 0x00000000 */ 0x00, 0x00, 0x00, 0x00,
/* 13 bmaControls(1) : 0x00000000 */ 0x00, 0x00, 0x00, 0x00,
/* . iFeature */ 0,
/* ----- Audio 2.0 AC Output Terminal Descriptor */
/* 0 bLength */ 12,
/* 1 bDescriptorType, bDescriptorSubtype */ 0x24, 0x03, /* 3 bTerminalID */ 0x06,
/* 4 wTerminalType : 0x0301 (Speaker) */ 0x01, 0x03,
/* 6 bAssocTerminal, bSourceID, bCSourceID */ 0x00, 0x05, 0x10,
/* 9 bmControls, iTerminal */ 0x00, 0x00, 0,
/* --- Interface Descriptor #1: Stream OUT (9+9+16+6+7+8=55) */
/* 0 bLength, bDescriptorType */ 9, 0x04,
/* 2 bInterfaceNumber, bAlternateSetting */ 1, 0,
/* 4 bNumEndpoints */ 0,
/* 5 bInterfaceClass : 0x01 (Audio) */ 0x01,
/* 6 bInterfaceSubClass : 0x01 (AudioStream) */ 0x02,
/* 7 bInterfaceProtocol : 0x20 (VERSION_02_00) */ 0x20,
/* 8 iInterface */ 0,
/* ---------------------- Interface Descriptor */
/* 0 bLength, bDescriptorType */ 9, 0x04,
/* 2 bInterfaceNumber, bAlternateSetting */ 1, 1,
/* 4 bNumEndpoints */ 1,
/* 5 bInterfaceClass : 0x01 (Audio) */ 0x01,
/* 6 bInterfaceSubClass : 0x01 (AudioStream) */ 0x02,
/* 7 bInterfaceProtocol : 0x20 (VERSION_02_00) */ 0x20,
/* 8 iInterface */ 0,
/* ----------- Audio 2.0 AS Interface Descriptor */
/* 0 bLength */ 16,
/* 1 bDescriptorType, bDescriptorSubtype */ 0x24, 0x01,
/* 3 bTerminalLink, bmControls */ 0x04, 0x00,
/* 5 bFormatType : 0x01 (FORMAT_TYPE_I) */ 0x01,
/* 6 bmFormats : 0x000000001 (PCM) */ 0x01, 0x00, 0x00, 0x00, /* 10 bNrChannels */ 2,
/* 11 bmChannelConfig */ 0x00, 0x00, 0x00, 0x00,
/* 15 iChannelNames */ 0, /* --------- Audio 2.0 AS Format Type Descriptor */
/* 0 bLength */ 6,
/* 1 bDescriptorType, bDescriptorSubtype */ 0x24, 0x02,
/* 3 bFormatType : 0x01 (FORMAT_TYPE_I) */ 0x01,
/* 4 bSubslotSize, bBitResolution */ 2, 16,
```

```
    /* ------------------------ Endpoint Descriptor */
    /* 0 bLength, bDescriptorType */ 7, 0x05,
    /* 2 bEndpointAddress */ 0x02,
    /* 3 bmAttributes : 0x0D (Sync|ISO) */ 0x0D,
    /* 4 wMaxPacketSize : 0x0100 (256) */ 0x00, 0x01,
    /* 6 bInterval : 0x04 (1ms) */ 4,
    /* - Audio 2.0 AS ISO Audio Data Endpoint Descriptor */
    /* 0 bLength */ 8,
    /* 1 bDescriptorType, bDescriptorSubtype */ 0x25, 0x01,
    /* 3 bmAttributes, bmControls */ 0x00, 0x00,
    /* 5 bLockDelayUnits, wLockDelay */ 0x00, 0x00, 0x00,
};
```

音频类使用复合设备框架对接口进行分组(控制和流式处理)。因此，在定义设备描述符时应保持谨慎。USBX 依赖于 IAD 描述符来了解如何在内部绑定接口。IAD 描述符应在接口之前声明(一个 AudioControl 接口后接一个或多个 AudioStreaming 接口)，并包含音频类的第一个接口(AudioControl 接口)以及附加的接口数。

音频类的工作方式取决于设备是发送还是接收音频，但这两种情况都使用 FIFO 来存储音频帧缓冲区：如果设备向主机发送音频，则应用程序会将音频帧缓冲区添加到 FIFO，然后，这些缓冲区由 USBX 发送到主机；如果设备从主机接收音频，则 USBX 会将从主机收到的音频帧缓冲区添加到 FIFO，然后，应用程序将读取这些缓冲区。每个音频流都有自己的 FIFO，每个音频帧缓冲区由多个样本构成。

音频类的初始化需要以下组成部分。

1. 音频类需要以下流式处理参数：

```
/* Set the parameters for Audio streams. */
/* Set the application-defined callback that is invoked when the
   host requests a change to the alternate setting. */
audio_stream_parameter[0].ux_device_class_audio_stream_parameter_callbacks
    .ux_device_class_audio_stream_change = demo_audio_read_change;

/* Set the application-defined callback that is invoked whenever
   a USB packet (audio frame) is sent to or received from the host. */
audio_stream_parameter[0].ux_device_class_audio_stream_parameter_callbacks
    .ux_device_class_audio_stream_frame_done = demo_audio_read_done;

/* Set the number of audio frame buffers in the FIFO. */
audio_stream_parameter[0].ux_device_class_audio_stream_parameter_max_frame _buffer_nb =
UX_DEMO_FRAME_BUFFER_NB;

/* Set the maximum size of each audio frame buffer in the FIFO. */
audio_stream_parameter[0].ux_device_class_audio_stream_parameter_max_frame _buffer_size =
UX_DEMO_MAX_FRAME_SIZE;

/* Set the internally-defined audio processing thread entry pointer. If the application wishes to
receive audio from the host
   (which is the case in this example), ux_device_class_audio_read_thread_entry should be used;
   if the application wishes to send data to the host, ux_device_class_audio_write_thread_entry
should be used. */
audio_stream_parameter[0].ux_device_class_audio_stream_parameter_thread_entry =
ux_device_class_audio_read_thread_entry;
```

2. 音频类需要以下函数参数。

```
/* Set the parameters for Audio device. */

/* Set the number of streams. */
audio_parameter.ux_device_class_audio_parameter_streams_nb = 1;

/* Set the pointer to the first audio stream parameter.
   Note that we initialized this parameter in the previous section.
   Also note that for more than one streams, this should be an array. */
audio_parameter.ux_device_class_audio_parameter_streams = audio_stream_parameter;

/* Set the application-defined callback that is invoked when the audio class
   is activated i.e. device is connected to host. */
audio_parameter.ux_device_class_audio_parameter_callbacks
    .ux_slave_class_audio_instance_activate = demo_audio_instance_activate;

/* Set the application-defined callback that is invoked when the audio class
   is deactivated i.e. device is disconnected from host. */

audio_parameter.ux_device_class_audio_parameter_callbacks
    .ux_slave_class_audio_instance_deactivate = demo_audio_instance_deactivate;

/* Set the application-defined callback that is invoked when the stack receives a control request
from the host.
   See below for more details.
*/
audio_parameter.ux_device_class_audio_parameter_callbacks
    .ux_device_class_audio_control_process = demo_audio20_request_process;

/* Initialize the device Audio class. This class owns interfaces starting with 0. */
status = ux_device_stack_class_register(_ux_system_slave_class_audio_name,
    ux_device_class_audio_entry, 1, 0, &audio_parameter);
if(status!=UX_SUCCESS)
    return;
```

当堆栈接收来自主机的控制请求时，将调用应用程序定义的控制请求回调（ux_device_class_audio_control_process；已在前面的示例中设置）。如果已接受并处理请求（已确认或已停滞），则该回调必须返回成功结果，否则应返回错误。

类特定的控制请求进程定义为应用程序定义的回调，控制请求根据 USB 音频版本的不同而有很大的差异，并且请求进程的很大一部分与设备框架有关。应用程序应正确处理请求才能使设备正常运行。

由于对于音频设备而言，音量、静音和采样频率是常见的控制请求，因此，后续部分介绍了不同 USB 音频版本的、可供应用程序使用的简单且可在内部定义的回调。有关更多详细信息，请参阅“ux_device_class_audio10_control_process”和“ux_device_class_audio_control_request”。

在音频设备的设备框架中，PID/VID 存储在设备描述符中（请参阅上面声明的设备描述符）。

当 USB 主机系统发现 USB 音频设备并装载音频类时，可将该设备与任何音频播放器或录制器（具体取决于框架）配合使用。有关参考信息，请参阅“主机操作系统”。

下面定义了音频类 API。

ux_device_class_audio_read_thread_entry

用于读取音频函数数据的线程条目。

原型

```
VOID ux_device_class_audio_read_thread_entry(ULONG audio_stream);
```

说明

如果需要从主机读取音频，则将此函数传递给音频流初始化参数。在内部，使用此函数作为入口函数来创建线程；线程本身通过 Audio 函数中的常时等量 OUT 终结点来读取音频数据。

参数

- audio_stream：指向音频流实例的指针。

示例

```
/* Set parameter to initialize a stream for reading. */
audio_stream_parameter[0].ux_device_class_audio_stream_parameter_thread_entry
     = ux_device_class_audio_read_thread_entry;
```

ux_device_class_audio_write_thread_entry

用于写入音频函数数据的线程条目

原型

```
VOID ux_device_class_audio_write_thread_entry(ULONG audio_stream);
```

说明

如果需要将音频写入主机，则将此函数传递给音频流初始化参数。在内部，使用此函数作为入口函数来创建线程；线程本身通过 Audio 函数中的常时等量 IN 终结点来写入音频数据。

参数

- audio_stream：指向音频流实例的指针。

示例

```
/* Set parameter to initialize as stream for writing. */
audio_stream_parameter[0].ux_device_class_audio_stream_parameter_thread_en
    try = ux_device_class_audio_write_thread_entry;
```

ux_device_class_audio_stream_get

获取音频函数的特定流实例

原型

```
UINT ux_device_class_audio_stream_get(
    UX_DEVICE_CLASS_AUDIO *audio,
    ULONG stream_index,
    UX_DEVICE_CLASS_AUDIO_STREAM **stream);
```

说明

此函数用于获取音频类的流实例。

参数

- audio：指向音频实例的指针
- stream_index：从 0 开始的流实例索引
- stream：指向用于存储音频流实例指针的缓冲区的指针

返回值

- UX_SUCCESS：(0x00) 此操作成功

- UX_ERROR：(0xFF) 函数出错

示例

```
/* Get audio stream instance. */
status = ux_device_class_audio_stream_get(audio, 0, &stream);

if(status != UX_SUCCESS)
    return;
```

ux_device_class_audio_reception_start

开始接收音频流的音频数据

原型

```
UINT ux_device_class_audio_reception_start(UX_DEVICE_CLASS_AUDIO_STREAM *stream);
```

说明

此函数用于在音频流中开始读取音频数据。

参数

- stream：指向音频流实例的指针。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_CONFIGURATION_HANDLE_UNKNOWN：(0x51) 接口已关闭。
- UX_BUFFER_OVERFLOW：(0x5d) FIFO 缓冲区已满。
- UX_ERROR：(0xFF) 函数出错

示例

```
/* Start stream data reception. */
status = ux_device_class_audio_reception_start(stream);

if(status != UX_SUCCESS)
    return;
```

ux_device_class_audio_sample_read8

从音频流中读取 8 位样本

原型

```
UINT ux_device_class_audio_sample_read8(
    UX_DEVICE_CLASS_AUDIO_STREAM *stream,
    UCHAR *buffer);
```

说明

此函数从指定的流中读取 8 位音频样本数据。

具体而言，它将从 FIFO 中的当前音频帧缓冲区读取样本数据。读取音频帧中的最后一个样本后，将自动释放该帧，以便可以使用它接受来自主机的其他数据。

参数

- stream：指向音频流实例的指针。
- buffer：指向用于保存样本字节的缓冲区的指针。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_CONFIGURATION_HANDLE_UNKNOWN：(0x51) 接口已关闭。
- UX_BUFFER_OVERFLOW：(0x5d) FIFO 缓冲区为 null。
- UX_ERROR：(0xFF) 函数出错

示例

```
/* Read a byte in audio FIFO. */

status = ux_device_class_audio_sample_read8(stream, &sample_byte);

if(status != UX_SUCCESS)
    return;
```

ux_device_class_audio_sample_read16

从音频流中读取 16 位样本

原型

```
UINT ux_device_class_audio_sample_read16(
    UX_DEVICE_CLASS_AUDIO_STREAM *stream,
    USHORT *buffer);
```

说明

此函数从指定的流中读取 16 位音频样本数据。

具体而言，它将从 FIFO 中的当前音频帧缓冲区读取样本数据。读取音频帧中的最后一个样本后，将自动释放该帧，以便可以使用它接受来自主机的其他数据。

参数

- stream：指向音频流实例的指针。
- buffer：指向用于保存 16 位样本的缓冲区的指针。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_CONFIGURATION_HANDLE_UNKNOWN：(0x51) 接口已关闭。
- UX_BUFFER_OVERFLOW：(0x5d) FIFO 缓冲区为 null。
- UX_ERROR：(0xFF) 函数出错

示例

```
/* Read a 16-bit sample in audio FIFO. */

status = ux_device_class_audio_sample_read16(stream, &sample_word);

if(status != UX_SUCCESS)
    return;
```

ux_device_class_audio_sample_read24

从音频流中读取 24 位样本

原型

```
UINT ux_device_class_audio_sample_read24(
    UX_DEVICE_CLASS_AUDIO_STREAM *stream,
    ULONG *buffer);
```

说明

此函数从指定的流中读取 24 位音频样本数据。

具体而言，它将从 FIFO 中的当前音频帧缓冲区读取样本数据。读取音频帧中的最后一个样本后，将自动释放该帧，以便可以使用它接受来自主机的其他数据。

参数

- stream：指向音频流实例的指针。
- buffer：指向用于保存 3 字节样本的缓冲区的指针。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_CONFIGURATION_HANDLE_UNKNOWN：(0x51) 接口已关闭。
- UX_BUFFER_OVERFLOW：(0x5d) FIFO 缓冲区为 null。
- UX_ERROR：(0xFF) 函数出错

示例

```
/* Read 3 bytes to in audio FIFO. */

status = ux_device_class_audio_sample_read24(stream, &sample_bytes);

if(status != UX_SUCCESS)
    return;
```

ux_device_class_audio_sample_read32

从音频流中读取 32 位样本

原型

```
UINT ux_device_class_audio_sample_read32(
    UX_DEVICE_CLASS_AUDIO_STREAM *stream,
    ULONG *buffer);
```

说明

此函数从指定的流中读取 32 位音频样本数据。

具体而言，它将从 FIFO 中的当前音频帧缓冲区读取样本数据。读取音频帧中的最后一个样本后，将自动释放该帧，以便可以使用它接受来自主机的其他数据。

参数

- stream：指向音频流实例的指针。
- buffer：指向用于保存 4 字节数据的缓冲区的指针。

返回值

- UX_SUCCESS：(0x00) 此操作成功。

- UX_CONFIGURATION_HANDLE_UNKNOWN：(0x51) 接口已关闭。
- UX_BUFFER_OVERFLOW：(0x5d) FIFO 缓冲区为 null。
- UX_ERROR：(0xFF) 函数出错

示例

```
/* Read 4 bytes in audio FIFO. */

status = ux_device_class_audio_sample_read32(stream, &sample_bytes);

if(status != UX_SUCCESS)
    return;
```

ux_device_class_audio_read_frame_get

获取对音频流中音频帧的访问权限

原型

```
UINT ux_device_class_audio_read_frame_get(
    UX_DEVICE_CLASS_AUDIO_STREAM *stream,
    UCHAR **frame_data,
    ULONG *frame_length);
```

说明

此函数返回指定的流的 FIFO 中第一个音频帧缓冲区及其长度。当应用程序处理完数据时，必须使用 ux_device_class_audio_read_frame_free 释放 FIFO 中的帧缓冲区。

参数

- stream：指向音频流实例的指针。
- frame_data：指向数据指针的指针，将在该数据指针中返回数据指针。
- frame_length：指向用于保存帧长度（以字节数计）的缓冲区的指针。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_CONFIGURATION_HANDLE_UNKNOWN：(0x51) 接口已关闭。
- UX_BUFFER_OVERFLOW：(0x5d) FIFO 缓冲区为 null。
- UX_ERROR：(0xFF) 函数出错

示例

```
/* Get frame access. */

status = ux_device_class_audio_read_frame_get(stream, &frame, &frame_length);

if(status != UX_SUCCESS)
    return;
```

ux_device_class_audio_read_frame_free

释放音频流中的音频帧缓冲区

原型

```
UINT ux_device_class_audio_read_frame_free(UX_DEVICE_CLASS_AUDIO_STREAM *stream);
```

说明

此函数释放位于指定流的 FIFO 前面的音频帧缓冲区，使该缓冲区能够从主机接收数据。

参数

- stream：指向音频流实例的指针。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_CONFIGURATION_HANDLE_UNKNOWN：(0x51) 接口已关闭。
- UX_BUFFER_OVERFLOW：(0x5d) FIFO 缓冲区为 null。
- UX_ERROR：(0xFF) 函数出错

示例

```
/* Refree a frame buffer in FIFO. */

status = ux_device_class_audio_read_frame_free(stream);

if(status != UX_SUCCESS)
    return;
```

ux_device_class_audio_transmission_start

开始传输音频流的音频数据

原型

```
UINT ux_device_class_audio_transmission_start(UX_DEVICE_CLASS_AUDIO_STREAM *stream);
```

说明

此函数用于在音频类中开始发送写入到 FIFO 的音频数据。

参数

- stream：指向音频流实例的指针。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_CONFIGURATION_HANDLE_UNKNOWN：(0x51) 接口已关闭。
- UX_BUFFER_OVERFLOW：(0x5d) FIFO 缓冲区为 null。
- UX_ERROR：(0xFF) 函数出错

示例

```
/* Start stream data transmission. */

status = ux_device_class_audio_transmission_start(stream);

if(status != UX_SUCCESS)
    return;
```

ux_device_class_audio_frame_write

将音频帧写入音频流

原型

```
UINT ux_device_class_audio_frame_write(
    UX_DEVICE_CLASS_AUDIO_STREAM *stream,
    UCHAR *frame,
    ULONG frame_length);
```

说明

此函数将帧写入音频流的 FIFO。帧数据将复制到 FIFO 中的可用缓冲区，这样就可以发送到主机。

参数

- stream：指向音频流实例的指针。
- frame：指向帧数据的指针。
- frame_length：帧长度，以字节数计。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_CONFIGURATION_HANDLE_UNKNOWN：(0x51) 接口已关闭。
- UX_BUFFER_OVERFLOW：(0x5d) FIFO 缓冲区已满。
- UX_ERROR：(0xFF) 函数出错

示例

```
/* Get frame access. */

status = ux_device_class_audio_frame_write(stream, frame, frame_length);

if(status != UX_SUCCESS)
    return;
```

ux_device_class_audio_write_frame_get

获取对音频流中音频帧的访问权限

原型

```
UINT ux_device_class_audio_write_frame_get(
    UX_DEVICE_CLASS_AUDIO_STREAM *stream,
    UCHAR **frame_data,
    ULONG *frame_length);
```

说明

此函数检索 FIFO 的最后一个音频帧缓冲区的地址；它还检索音频帧缓冲区的长度。在应用程序将所需数据填充到音频帧缓冲区后，必须使用 ux_device_class_audio_write_frame_commit 向 FIFO 添加/提交帧缓冲区。

参数

- stream：指向音频流实例的指针。
- frame_data：指向帧数据指针的指针，将在该帧数据指针中返回帧数据指针。
- frame_length：指向用于保存帧长度（以字节数计）的缓冲区的指针。

返回值

- UX_SUCCESS：(0x00) 此操作成功。

- UX_CONFIGURATION_HANDLE_UNKNOWN：(0x51) 接口已关闭。
- UX_BUFFER_OVERFLOW：(0x5d) FIFO 缓冲区已满。
- UX_ERROR：(0xFF) 函数出错

示例

```
/* Get frame access. */

status = ux_device_class_audio_write_frame_get(stream, &frame, &frame_length);

if(status != UX_SUCCESS)
     return;
```

ux_device_class_audio_write_frame_commit

在音频流中提交音频帧缓冲区。

原型

```
UINT ux_device_class_audio_write_frame_commit(
    UX_DEVICE_CLASS_AUDIO_STREAM *stream,
    ULONG length);
```

说明

此函数将最后一个音频帧缓冲区添加/提交到 FIFO，使该缓冲区随时可传输到主机；请注意，应已通过 ux_device_class_write_frame_get 填充最后一个音频帧缓冲区。

参数

- stream：指向音频流实例的指针。
- length：缓冲区中准备就绪的字节数。

返回值

- UX_SUCCESS：(0x00) 此操作成功。
- UX_CONFIGURATION_HANDLE_UNKNOWN：(0x51) 接口已关闭。
- UX_BUFFER_OVERFLOW：(0x5d) FIFO 缓冲区已满。
- UX_ERROR (0xFF) 函数出错

示例

```
/* Commit a frame after fill values in buffer. */

status = ux_device_class_audio_write_frame_commit(stream, 192);

if(status != UX_SUCCESS)
    return;
```

ux_device_class_audio10_control_process

处理 USB 音频 1.0 控制请求

原型

```
UINT ux_device_class_audio10_control_process(
    UX_DEVICE_CLASS_AUDIO *audio,
    UX_SLAVE_TRANSFER *transfer_request,
    UX_DEVICE_CLASS_AUDIO10_CONTROL_GROUP *group);
```

说明

此函数在 USB 音频 1.0 特定类型的控制终结点上管理主机发送的基本请求。

音量和静音请求的音频 1.0 功能将在该函数中处理。处理请求时，将使用最后一个参数 (group) 传递的预定义数据来应答请求并存储控制更改。

参数

- audio：指向音频实例的指针。
- transfer：指向传输请求实例的指针。
- group：请求进程的数据组。

返回值

- **UX_SUCCESS** (0x00) 此操作成功。
- **UX_ERROR** (0xFF) 函数出错

示例

```
/* Initialize audio 1.0 control values. */

audio_control[0].ux_device_class_audio10_control_fu_id = 2;
audio_control[0].ux_device_class_audio10_control_mute[0] = 0;
audio_control[0].ux_device_class_audio10_control_volume[0] = 0;
audio_control[1].ux_device_class_audio10_control_fu_id = 5;
audio_control[1].ux_device_class_audio10_control_mute[0] = 0;
audio_control[1].ux_device_class_audio10_control_volume[0] = 0;

/* Handle request and update control values.
Note here only mute and volume for master channel is supported.
*/

status = ux_device_class_audio10_control_process(audio, transfer, &group);
if (status == UX_SUCCESS)
{
    /* Request handled, check changes */
    switch(audio_control[0].ux_device_class_audio10_control_changed)
    {
        case UX_DEVICE_CLASS_AUDIO10_CONTROL_MUTE_CHANGED:
        case UX_DEVICE_CLASS_AUDIO10_CONTROL_VOLUME_CHANGED:
        default: break;
    }
}
```

ux_device_class_audio20_control_process

处理 USB 音频 1.0 控制请求

原型

```
UINT ux_device_class_audio20_control_process(
    UX_DEVICE_CLASS_AUDIO *audio,
    UX_SLAVE_TRANSFER *transfer_request,
    UX_DEVICE_CLASS_AUDIO20_CONTROL_GROUP *group);
```

说明

此函数在 USB 音频 2.0 特定类型的控制终结点上管理主机发送的基本请求。

音频2.0 采样率(假设为单一固定频率)以及音量和静音请求的功能将在该函数中处理。处理请求时，将使用最后一个参数 (group) 传递的预定义数据来应答请求并存储控制更改。

参数

- audio：指向音频实例的指针。
- transfer：指向传输请求实例的指针。
- group：请求进程的数据组。

返回值

- **UX_SUCCESS** (0x00) 此操作成功。
- **UX_ERROR** (0xFF) 函数出错

示例

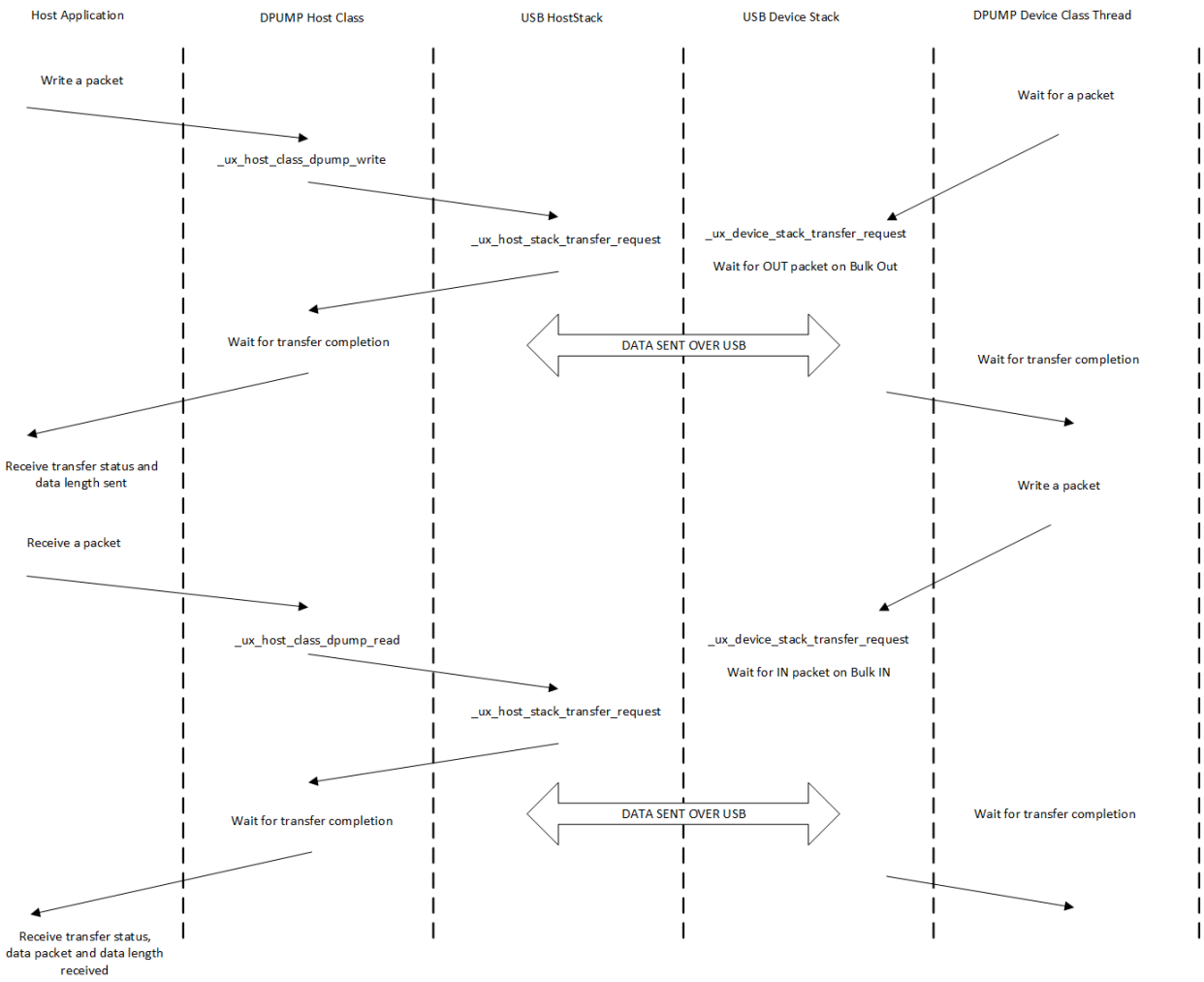
```
/* Initialize audio 2.0 control values. */

audio_control[0].ux_device_class_audio20_control_cs_id = 0x10;
audio_control[0].ux_device_class_audio20_control_sampling_frequency = 48000;
audio_control[0].ux_device_class_audio20_control_fu_id = 2;
audio_control[0].ux_device_class_audio20_control_mute[0] = 0;
audio_control[0].ux_device_class_audio20_control_volume_min[0] = 0;
audio_control[0].ux_device_class_audio20_control_volume_max[0] = 100;
audio_control[0].ux_device_class_audio20_control_volume[0] = 50;
audio_control[1].ux_device_class_audio20_control_cs_id = 0x10;
audio_control[1].ux_device_class_audio20_control_sampling_frequency = 48000;
audio_control[1].ux_device_class_audio20_control_fu_id = 5;
audio_control[1].ux_device_class_audio20_control_mute[0] = 0;
audio_control[1].ux_device_class_audio20_control_volume_min[0] = 0;
audio_control[1].ux_device_class_audio20_control_volume_max[0] = 100;
audio_control[1].ux_device_class_audio20_control_volume[0] = 50;

/* Handle request and update control values.
Note here only mute and volume for master channel is supported.
*/

status = ux_device_class_audio20_control_process(audio, transfer, &group);
if (status == UX_SUCCESS)
{
    /* Request handled, check changes */
    switch(audio_control[0].ux_device_class_audio20_control_changed)
    {
        case UX_DEVICE_CLASS_AUDIO20_CONTROL_MUTE_CHANGED:
        case UX_DEVICE_CLASS_AUDIO20_CONTROL_VOLUME_CHANGED:
        default: break;
    }
}
```

第 3 章 - USBX DPUMP 类注意事项

2021/4/29 •

USBX 包含一个用于主机和设备端的 DPUMP 类。此类本身不是一个标准类，而是一个示例，它展示了如何通过以下方式创建一个简单的设备：使用两个大容量管道并在这两个管道上来回发送数据。DPUMP 类可用于启动自定义类，也可用于旧式的 RS232 设备。

USB DPUMP 流程图：



USBX DPUMP 设备类

设备 DPUMP 类使用一个线程，该线程在连接到 USB 主机时启动。线程在大容量输出终结点上等待数据包的到来。收到数据包时，它将内容复制到大容量输入终结点缓冲区中，在此终结点上发布事务，并等待主机发出从此终结点进行读取的请求。这在大容量输出终结点与大容量输入终结点之间提供了一种环回机制。

第 4 章 - USBX Pictbridge 实现

2021/4/29 •

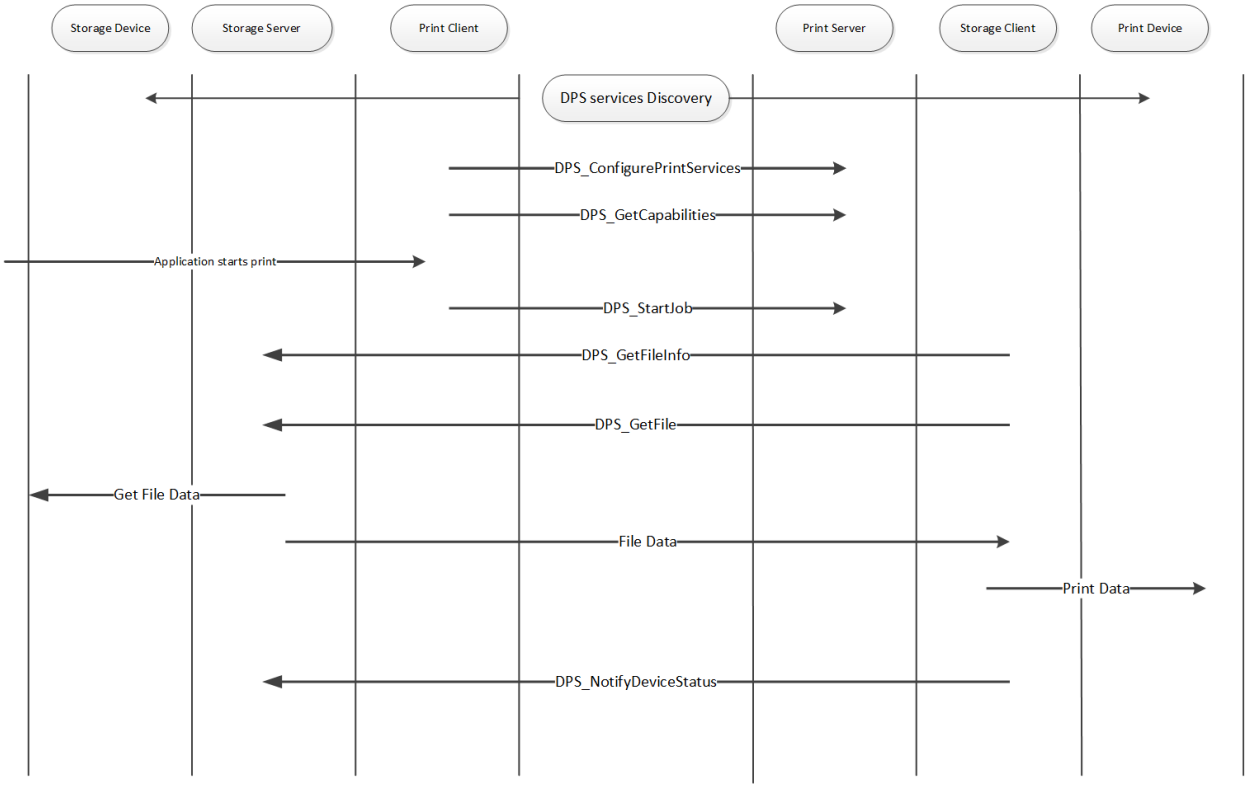
USBX 在主机和设备上都支持完全 Pictbridge 实现。在两端，Pictbridge 都位于 USBX PIMA 类之上。

PictBridge 标准允许将数码照相机或智能手机直接连接到打印机，而不使用 PC，从而可以直接使用特定的 Pictbridge 感知打印机进行打印。

当照相机或手机连接到打印机时，打印机即为 USB 主机，照相机即为 USB 设备。然而，在使用 Pictbridge 时，照相机显示为主机，而且命令是从照相机驱动的。照相机是存储服务器，打印机是存储客户端。照相机是打印客户端，打印机当然是打印服务器。

Pictbridge 使用 USB 作为传输层，但依赖于 PTP（图片传输协议）作为通信协议。

下图展示了在执行打印作业时 DPS 客户端与 DPS 服务器之间的命令/响应：



Pictbridge 客户端实现

客户端上的 Pictbridge 要求先运行 USBX 设备堆栈和 PIMA 类。

设备框架以如下方式描述 PIMA 类。

```
UCHAR device_framework_full_speed[] =
{
    /* Device descriptor */
    0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x20,
    0xA9, 0x04, 0xB6, 0x30, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x01,
    /* Configuration descriptor */
    0x09, 0x02, 0x27, 0x00, 0x01, 0x01, 0x00, 0xc0, 0x32,
    /* Interface descriptor */
    0x09, 0x04, 0x00, 0x00, 0x03, 0x06, 0x01, 0x01, 0x00,
    /* Endpoint descriptor (Bulk Out) */
    0x07, 0x05, 0x01, 0x02, 0x40, 0x00, 0x00,
    /* Endpoint descriptor (Bulk In) */
    0x07, 0x05, 0x82, 0x02, 0x40, 0x00, 0x00,
    /* Endpoint descriptor (Interrupt) */
    0x07, 0x05, 0x83, 0x03, 0x08, 0x00, 0x60
};
```

PIMA 类使用 ID 字段 0x06，对于静态图像，它的子类为 0x01，对于 PIMA 15740，它的协议为 0x01。

此类中定义了 3 个终结点，2 个用于发送/接收数据的批处理，1 个用于事件的中断。

与其他 USBX 设备实现不同，Pictbridge 应用程序本身不需要定义类，而是调用函数 ux_pictbridge_dpsclient_start。下面是一个示例。

```
/* Initialize the Pictbridge string components. */
ux_utility_memory_copy
    (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_vendor_name,
    "ExpressLogic",13);

ux_utility_memory_copy
    (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_product_name,
    "EL_Pictbridge_Camera",21);

ux_utility_memory_copy
    (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_serial_no, "ABC_123",7);

ux_utility_memory_copy
    (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_dpsversions,
    "1.0 1.1",7);

pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_vendor_specific_version = 0x0100;

/* Start the Pictbridge client. */
status = ux_pictbridge_dpsclient_start(&pictbridge);

if(status != UX_SUCCESS)
    return;
```

传递给 pictbridge 客户端的参数如下所示。

```
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_vendor_name
    : String of Vendor name
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_product_name
    : String of product name
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_serial_no
    : String of serial number
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_dpsversions
    : String of version
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_vendor_specific_version
    : Value set to 0x0100;
```

下一步是让设备和主机同步，并准备好进行信息交换。

这是通过等待事件标志来完成的，如下所示。

```
/* We should wait for the host and the client to discover one another. */
status = ux_utility_event_flags_get(&pictbridge.ux_pictbridge_event_flags_group,
    UX_PICTBRIDGE_EVENT_FLAG_DISCOVERY,TX_AND_CLEAR,
    &actual_flags, UX_PICTBRIDGE_EVENT_TIMEOUT);
```

如果状态机处于“DISCOVERY_COMPLETE”状态，则照相机端（DPS 客户端）将收集有关打印机及其功能的信息。

如果 DPS 客户端已准备好接受打印作业，则其状态设置为“UX_PICTBRIDGE_NEW_JOB_TRUE”。可以按如下进行检查。

```
/* Check if the printer is ready for a print job. */
if (pictbridge.ux_pictbridge_dpsclient.ux_pictbridge_devinfo_newjobok ==
    UX_PICTBRIDGE_NEW_JOB_TRUE)
/* We can print something … */
```

接下来，需要按如下方式填充一些打印工作描述符：

```
/* We can start a new job. Fill in the JobConfig and PrintInfo structures. */
jobinfo = &pictbridge.ux_pictbridge_jobinfo;

/* Attach a printinfo structure to the job. */
jobinfo -> ux_pictbridge_jobinfo_printinfo_start = &printinfo;

/* Set the default values for print job. */
jobinfo -> ux_pictbridge_jobinfo_quality =
    UX_PICTBRIDGE_QUALITIES_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_papersize =
    UX_PICTBRIDGE_PAPER_SIZES_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_papertype =
    UX_PICTBRIDGE_PAPER_TYPES_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_filetype =
    UX_PICTBRIDGE_FILE_TYPES_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_dateprint =
    UX_PICTBRIDGE_DATE_PRINTS_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_filenameprint =
    UX_PICTBRIDGE_FILE_NAME_PRINTS_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_imageoptimize =
    UX_PICTBRIDGE_IMAGE_OPTIMIZES_OFF;
jobinfo -> ux_pictbridge_jobinfo_layout =
    UX_PICTBRIDGE_LAYOUTS_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_fixedsize =
    UX_PICTBRIDGE_FIXED_SIZE_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_cropping =
    UX_PICTBRIDGE_CROPPINGS_DEFAULT;

/* Program the callback function for reading the object data. */
jobinfo -> ux_pictbridge_jobinfo_object_data_read =
    ux_demo_object_data_copy;

/* This is a demo, the fileID is hardwired (1 and 2 for scripts, 3 for photo to be printed. */
printinfo.ux_pictbridge_printinfo_fileid =
    UX_PICTBRIDGE_OBJECT_HANDLE_PRINT;
ux_utility_memory_copy(printinfo.ux_pictbridge_printinfo_filename,
    "Pictbridge demo file", 20);
ux_utility_memory_copy(printinfo.ux_pictbridge_printinfo_date, "01/01/2008",
    10);

/* Fill in the object info to be printed. First get the pointer to the object container in the job info
structure. */
object = (UX_SLAVE_CLASS_PIMA_OBJECT *) jobinfo ->
    ux_pictbridge_jobinfo_object;

/* Store the object format: JPEG picture. */
object -> ux_device_class_pima_object_format = UX_DEVICE_CLASS_PIMA_OFC_EXIF_JPEG;
object -> ux_device_class_pima_object_compressed_size = IMAGE_LEN;
object -> ux_device_class_pima_object_offset = 0;
object -> ux_device_class_pima_object_handle_id =
    UX_PICTBRIDGE_OBJECT_HANDLE_PRINT;
object -> ux_device_class_pima_object_length = IMAGE_LEN;

/* File name is in Unicode. */
ux_utility_string_to_unicode("JPEG Image", object ->
    ux_device_class_pima_object_filename);

/* And start the job. */
status =ux_pictbridge_dpsclient_api_start_job(&pictbridge);
```

现在，Pictbridge 客户端有一个打印作业要执行，它将通过字段中定义的回叫从应用程序中一次提取多个图像块

```
jobinfo -> ux_pictbridge_jobinfo_object_data_read
```

此函数的原型定义如下：

ux_pictbridge_jobinfo_object_data_read

从用户空间复制数据块以供打印

原型

```
UINT ux_pictbridge_jobinfo_object_data_read(
    UX_PICTBRIDGE *pictbridge,
    UCHAR *object_buffer,
    ULONG object_offset,
    ULONG object_length,
    ULONG *actual_length)
```

说明

当 DPS 客户端需要检索数据块以在目标 Pictbridge 打印机上打印时，就会调用此函数。

参数

- pictbridge：指向 pictbridge 类实例的指针。
- object_buffer：指向对象缓冲区的指针
- object_offset：从哪里开始读取数据块
- object_length：要返回的长度
- actual_length：返回的实际长度

返回值

- **UX_SUCCESS** (0x00) 此操作成功。
- **UX_ERROR** (0x01) 应用程序无法检索数据。

示例

```
/* Copy the object data. */
UINT ux_demo_object_data_copy(
    UX_PICTBRIDGE *pictbridge,
    UCHAR *object_buffer,
    ULONG object_offset,
    ULONG object_length,
    ULONG *actual_length)
{
    /* Copy the demanded object data portion. */
    ux_utility_memory_copy(object_buffer, image + object_offset,
        object_length);
    /* Update the actual length. */
    *actual_length = object_length;
    /* We have copied the requested data. Return OK. */
    return(UX_SUCCESS);
}
```

Pictbridge 主机实现

Pictbridge 的主机实现与客户端不同。

在 Pictbridge 主机环境中要做的第一件事是注册 PIMA 类，如下面的示例所示：

```
status = ux_host_stack_class_register(_ux_system_host_class_pima_name,
    ux_host_class_pima_entry);
if(status != UX_SUCCESS)
    return;
```

此类是位于 USB 堆栈与 Pictbridge 层之间的通用 PTP 层。

下一步是初始化打印服务的 Pictbridge 默认值，如下所示：

PICTBRIDGE 值	值
DpsVersion[0]	0x00010000
DpsVersion[1]	0x00010001
DpsVersion[2]	0x00000000
VendorSpecificVersion	0x00010000
PrintServiceAvailable	0x30010000
Qualities[0]	UX_PICTBRIDGE_QUALITIES_DEFAULT
Qualities[1]	UX_PICTBRIDGE_QUALITIES_NORMAL
Qualities[2]	UX_PICTBRIDGE_QUALITIES_DRAFT
Qualities[3]	UX_PICTBRIDGE_QUALITIES_FINE
PaperSizes[0]	UX_PICTBRIDGE_PAPER_SIZES_DEFAULT
PaperSizes[1]	UX_PICTBRIDGE_PAPER_SIZES_4IX6I
PaperSizes[2]	UX_PICTBRIDGE_PAPER_SIZES_L
PaperSizes[3]	UX_PICTBRIDGE_PAPER_SIZES_2L
PaperSizes[4]	UX_PICTBRIDGE_PAPER_SIZES_LETTER
PaperTypes[0]	UX_PICTBRIDGE_PAPER_TYPES_DEFAULT
PaperTypes[1]	UX_PICTBRIDGE_PAPER_TYPES_PLAIN
PaperTypes[2	UX_PICTBRIDGE_PAPER_TYPES_PHOTO
FileTypes[0]	UX_PICTBRIDGE_FILE_TYPES_DEFAULT
FileTypes[1]	UX_PICTBRIDGE_FILE_TYPES_EXIF_JPEG
FileTypes[2]	UX_PICTBRIDGE_FILE_TYPES_JFIF
FileTypes[3]	UX_PICTBRIDGE_FILE_TYPES_DPOF
DatePrints[0]	UX_PICTBRIDGE_DATE_PRINTS_DEFAULT
DatePrints[1]	UX_PICTBRIDGE_DATE_PRINTS_OFF
DatePrints[2]	UX_PICTBRIDGE_DATE_PRINTS_ON

PICTBRIDGE 能	能
FileNamePrints[0]	UX_PICTBRIDGE_FILE_NAME_PRINTS_DEFAULT
FileNamePrints[1]	UX_PICTBRIDGE_FILE_NAME_PRINTS_OFF
FileNamePrints[2]	UX_PICTBRIDGE_FILE_NAME_PRINTS_ON
ImageOptimizes[0]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_DEFAULT
ImageOptimizes[1]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_OFF
ImageOptimizes[2]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_ON
Layouts[0]	UX_PICTBRIDGE_LAYOUTS_DEFAULT
Layouts[1]	UX_PICTBRIDGE_LAYOUTS_1_UP_BORDER
Layouts[2]	UX_PICTBRIDGE_LAYOUTS_INDEX_PRINT
Layouts[3]	UX_PICTBRIDGE_LAYOUTS_1_UP_BORDERLESS
FixedSizes[0]	UX_PICTBRIDGE_FIXED_SIZE_DEFAULT
FixedSizes[1]	UX_PICTBRIDGE_FIXED_SIZE_35IX5I
FixedSizes[2]	UX_PICTBRIDGE_FIXED_SIZE_4IX6I
FixedSizes[3]	UX_PICTBRIDGE_FIXED_SIZE_5IX7I
FixedSizes[4]	UX_PICTBRIDGE_FIXED_SIZE_7CMX10CM
FixedSizes[5]	UX_PICTBRIDGE_FIXED_SIZE_LETTER
FixedSizes[6]	UX_PICTBRIDGE_FIXED_SIZE_A4
Croppings[0]	UX_PICTBRIDGE_CROPPINGS_DEFAULT
Croppings[1]	UX_PICTBRIDGE_CROPPINGS_OFF
Croppings[2]	UX_PICTBRIDGE_CROPPINGS_ON

DPS 主机的状态机将被设置为"空闲"，并准备好接受新的打印作业。

现在可以启动 Pictbridge 的主机部分，如下面的示例所示：

```
/* Activate the pictbridge dpshost. */
status = ux_pictbridge_dpshost_start(&pictbridge, pima);

if (status != UX_SUCCESS)
    return;
```

当数据可供打印时，Pictbridge 主机函数需要回叫。这是通过在 pictbridge 主机结构中传递函数指针来完成的，

如下所示。

```
/* Set a callback when an object is being received. */
pictbridge.ux_pictbridge_application_object_data_write =
    tx_demo_object_data_write;
```

此函数有以下属性。

ux_pictbridge_application_object_data_write

编写数据块以供打印

原型

```
UINT ux_pictbridge_application_object_data_write(
    UX_PICTBRIDGE *pictbridge,
    UCHAR *object_buffer,
    ULONG offset,
    ULONG total_length,
    ULONG length);
```

说明

当 DPS 服务器需要从 DPS 客户端中检索数据块以在本地打印机上打印时，就会调用此函数。

参数

- pictbridge：指向 pictbridge 类实例的指针。
- object_buffer：指向对象缓冲区的指针
- object_offset：从哪里开始读取数据块
- total_length：对象的整个长度
- length：此缓冲区的长度

返回值

- **UX_SUCCESS** (0x00) 此操作成功。
- **UX_ERROR** (0x01) 应用程序无法打印数据。

示例

```
/* Copy the object data. */
UINT tx_demo_object_data_write(UX_PICTBRIDGE *pictbridge,
    UCHAR *object_buffer, ULONG offset, ULONG total_length, ULONG length);
{
    UINT status;
    /* Send the data to the local printer. */
    status = local_printer_data_send(object_buffer, length);

    /* We have printed the requested data. Return status. */
    return(status);
}
```

第 5 章 - USBX OTG

2021/4/30 •

当在硬件设计中采用兼容 OTG 的 USB 控制器时，USBX 支持使用 USB 的 OTG 功能。

USBX 支持核心 USB 堆栈中的 OTG。但要使 OTG 正常工作，它需要一个特定的 USB 控制器。USBX OTG 控制器函数可在 usbx_otg 目录中找到。当前 USBX 版本仅支持具有完全 OTG 功能的 NXP LPC3131。

常规的控制器驱动程序函数（主机或设备）仍然可以在标准 USBX usbx_device_controllers 和 usbx_host_controllers 中找到，但是 usbx_otg 目录包含与 USB 控制器关联的特定 OTG 函数。

除了常见的主机/设备函数外，OTG 控制器还有四类函数。

- VBUS 特定函数
- 控制器的启动和停止
- USB 角色管理器
- 中断处理程序

VBUS 函数

每个控制器都需要配备一个 VBUS 管理器，以根据电源管理要求更改 VBUS 的状态。通常，此函数仅用于打开或关闭 VBUS。

启动和停止控制器

与常规的 USB 实现不同，OTG 要求在角色发生更改时激活和停用主机和/或设备堆栈。

USB 角色管理器

USB 角色管理器可接收用于更改 USB 状态的命令。需要在多个状态之间进行转换：

[illegible]	[illegible]	[illegible]
UX_OTG_IDLE	0	设备处于空闲状态。未连接到任何对象
UX_OTG_IDLE_TO_HOST	1	设备已连接 A 类型连接器
UX_OTG_IDLE_TO_SLAVE	2	设备已连接 B 类型连接器
UX_OTG_HOST_TO_IDLE	3	主机设备已断开连接
UX_OTG_HOST_TO_SLAVE	4	将角色从主机调换为从属设备
UX_OTG_SLAVE_TO_IDLE	5	从属设备已断开连接
UX_OTG_SLAVE_TO_HOST	6	将角色从从属设备调换为主机

中断处理程序

OTG 的主机驱动程序和设备控制器驱动程序都需要不同的中断处理程序来监视传统 USB 中断以外的信号，尤其是 SRP 和 VBUS 引起的信号。

如何初始化 USB OTG 控制器。此处以 NXP LPC3131 为例。

```
/* Initialize the LPC3131 OTG controller. */
status = ux_otg_lpc3131_initialize(0x19000000, lpc3131_vbus_function,
    tx_demo_change_mode_callback);
```

在此示例中，我们通过传递 VBUS 函数和模式更改回调（从主机到从属设备，或者相反的方向），在 OTG 模式下初始化 LPC3131。

回调函数应直接记录新模式并唤醒待处理的线程以处理新状态。

```
void tx_demo_change_mode_callback(ULONG mode) {
    /* Simply save the otg mode. */
    otg_mode = mode;

    /* Wake up the thread that is waiting. */
    ux_utility_semaphore_put(&mode_change_semaphore);
}
```

传递的模式值可以有以下值。

- UX_OTG_MODE_IDLE
- UX_OTG_MODE_SLAVE
- UX_OTG_MODE_HOST

应用程序始终可以通过查看变量来确定是哪个设备：

```
_ux_system_otg -> ux_system_otg_device_type
```

它的值可以是下列任一值。

- UX_OTG_DEVICE_A
- UX_OTG_DEVICE_B
- UX_OTG_DEVICE_IDLE

USB OTG 主机设备始终可以通过发出以下命令来请求角色交换。

```
/* Ask the stack to perform a HNP swap with the device. We relinquish the host role to A device. */
ux_host_stack_role_swap(storage -> ux_host_class_storage_device);
```

对于从属设备，没有要发出的命令，但从属设备可以设置状态以更改角色，主机在发出 GET_STATUS 时将选取该角色，然后发起交换。

```
/* We are a B device, ask for role swap.
   The next GET_STATUS from the host will get the status change and do the HNP. */
_ux_system_otg -> ux_system_otg_slave_role_swap_flag =
    UX_OTG_HOST_REQUEST_FLAG;
```

Azure RTOS USBX 主机堆栈用户指南

2021/4/29 •

本指南全面介绍 Azure RTOS USBX（Microsoft 提供的高性能 USB 基础软件）。

它适用于嵌入式实时软件开发人员。开发人员应熟悉标准实时操作系统函数、USB 规范和 C 编程语言。

如需与 USB 相关的技术信息，请参阅可从 https://www.USB.org/developers 下载的 USB 规范和 USB 类规范

组织

- 第 1 章 - 包含 Azure RTOS USBX 简介
- 第 2 章 - 介绍安装 Azure RTOS USBX 并将其与 Azure RTOS ThreadX 应用程序配合使用的基本步骤
- 第 3章 - 提供 Azure RTOS USBX 的功能概述以及 USB 的基本信息
- 第 4 章 - 详细介绍主机模式下应用程序的 Azure RTOS USBX 接口
- 第 5 章 - 介绍 Azure RTOS USBX 主机类的 API
- 第 6 章 - 介绍 Azure RTOS USBX CDC-ECM 类

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. 详细说明发生问题前对应用程序和/或 Azure RTOS ThreadX 所做的任何更改。
3. 可在分发的 tx_port.h 文件中找到的 _tx_version_id 字符串的内容。此字符串将为我们提供有关运行时环境的重要信息。
4. RAM 中 _tx_build_options ULONG 变量的内容。此变量将为我们提供有关 Azure RTOS ThreadX 库生成方式的信息。

第 1 章 - Azure RTOS USBX 主机堆栈简介

2021/4/30 •

USBX 是适用于深度嵌入应用程序的全功能 USB 堆栈。本章引入了 USBX，介绍其应用程序和优点。

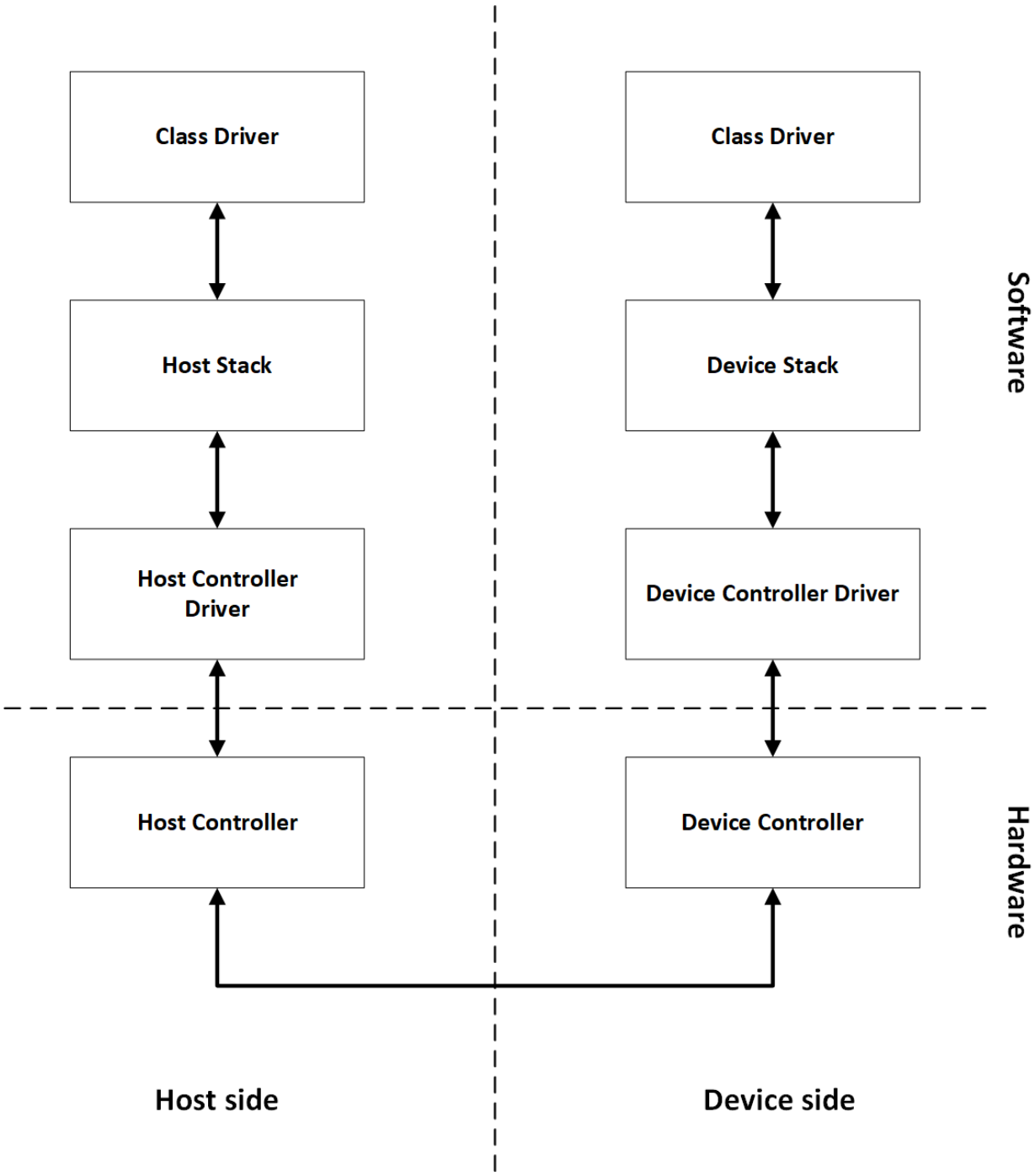
USBX 功能

USBX 支持三个现有的 USB 规范：1.1、2.0 和 OTG。它设计为能够缩放，适用于只有一个已连接设备的简单 USB 拓扑，以及具有多个设备和级联中心的复杂拓扑。USBX 支持 USB 协议的所有数据传输类型：控制、批量、中断和常时等量。

USBX 同时支持主机端和设备端。每一端都由三个层组成。

- 控制器层
- 堆栈层
- 类层

USB 层之间的关系如下所示。



产品亮点

- 完整的 ThreadX 处理器支持
- 无版税
- 完整的 ANSI C 源代码
- 实时性能
- 快速响应的技术支持
- 多主机控制器支持
- 多类支持
- 多类实例
- 各个类与 ThreadX、FileX 和 NetX 集成
- 支持具有多个配置的 USB 设备
- 支持 USB 复合设备
- 支持级联中心

- 支持 USB 电源管理
- 支持 USB OTG
- 导出 TraceX 的跟踪事件

强大的 USBX 服务

多主机控制器支持

USBX 可以支持同时运行的多个 USB 主机控制器。此功能允许 USBX 使用与当今市场上大多数 USB 2.0 主机控制器关联的后向兼容方案来支持 USB 2.0 标准。

USB 软件计划程序

USBX 包含一个 USB 软件计划程序，该计划程序是为没有硬件列表处理功能的 USB 控制器提供支持所必需的。USBX 软件计划程序将以正确的服务频率和优先级组织 USB 传输，并指示 USB 控制器执行每次传输。

完整的 USB 设备框架支持

USBX 可以支持要求最苛刻的 USB 设备，包括多个配置、多个接口和多个备用设置。

易于使用的 API

USBX 以一种易于理解和使用的方式提供了最佳的深度嵌入 USB 堆栈。USBX API 使服务直观且一致。通过使用提供的 USBX 类 API，用户应用程序无需了解 USB 协议的复杂性。

第 2 章 - Azure RTOS USBX 主机堆栈安装

2021/5/1 •

主机注意事项

计算机类型

嵌入式开发通常在 Windows PC 或 Unix 主机计算机上执行。在对应用程序进行编译和链接并将其放置在主机上之后，将应用程序下载到目标硬件进行执行。

下载接口

通常，虽然并行接口、USB 接口和以太网接口变得越来越普遍，但目标下载通过 RS-232 串行接口进行。有关可用选项，请参阅开发工具文档。

调试工具

通常通过与程序映像下载相同的链接进行调试。存在多种调试器，包括在目标上运行的小型监视器程序、后台调试监视器 (BDM) 和在线仿真器 (ICE) 工具等。ICE 工具可提供最可靠的实际目标硬件调试。

所需的硬盘空间

USBX 的源代码以 ASCII 格式提供，并要求主计算机的硬盘具有约 500 KB 可用空间。

目标注意事项

USBX 要求处于主机模式的目标具有 24 KB 到 64 KB 只读内存 (ROM)。所需的内存量取决于所使用的控制器类型和链接到 USBX 的 USB 类。USBX 全局数据结构和内存池要求目标具有额外的 32 KB 随机存取内存 (RAM)。还可以根据 USB 接口上预期连接的设备数和 USB 控制器的类型调整此内存池。USBX 设备端需要大约 10 K 到 12 K ROM，具体取决于设备控制器的类型。RAM 内存使用量取决于设备仿真的类的类型。

USBX 还依赖于 ThreadX 信号灯、互斥锁和线程进行多线程保护、I/O 暂停和定期处理，以监视 USB 总线拓扑。

产品分发

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

下面列出了该存储库中的几个重要文件：

- ux_api.h：此 C 头文件包含所有系统等式、数据结构和服务原型。
- ux_port.h：此 C 头文件包含所有特定于开发工具的数据定义和结构。
- ux.lib：这是 USBX C 库的二进制版本，它随标准包一起分发。
- demo_usbx.c：包含简单 USBX 演示的 C 文件

所有文件名均为小写。此命名约定使你可以更轻松地将命令转换为 Unix 开发平台命令。

USBX 安装

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

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

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。

你还可以在联机存储库的首页上找到有关生成 USBX 库的说明。

以下一般说明适用于几乎所有安装：

1. 使用之前在主机硬盘驱动器上安装 ThreadX 的同一目录。所有 USBX 名称都是唯一的，不会干扰以前的 USBX 安装。
2. 在 tx_application_define 的开头或附近添加对 ux_system_initialize 的调用。这是初始化 USBX 资源的位置。_ _
3. 添加对 ux_host_stack_initialize 的调用。* *****
4. 添加一个或多个调用，以初始化所需的 USBX。
5. 添加一个或多个调用，以初始化系统中可用的主机控制器。
6. 可能需要修改 tx_low_level_initialize.c 文件，以添加低级别硬件初始化并中断矢量路由。此说明特定于硬件平台，本文不做讨论。
7. 编译应用程序源代码并将其与 USBX 运行时库和 ThreadX 运行时库（如果要编译 USB 存储类和/或 USB 网络类，则可能还需要 FileX 和/或 Netx）、ux.a（或 ux.lib）以及 tx.a（或 tx.lib）链接起来。生成的库可下载到目标并执行。

配置选项

有多个配置选项用于生成 USBX 库。所有选项都位于 ux_user.h 中。

以下列表详细介绍了每个配置选项。

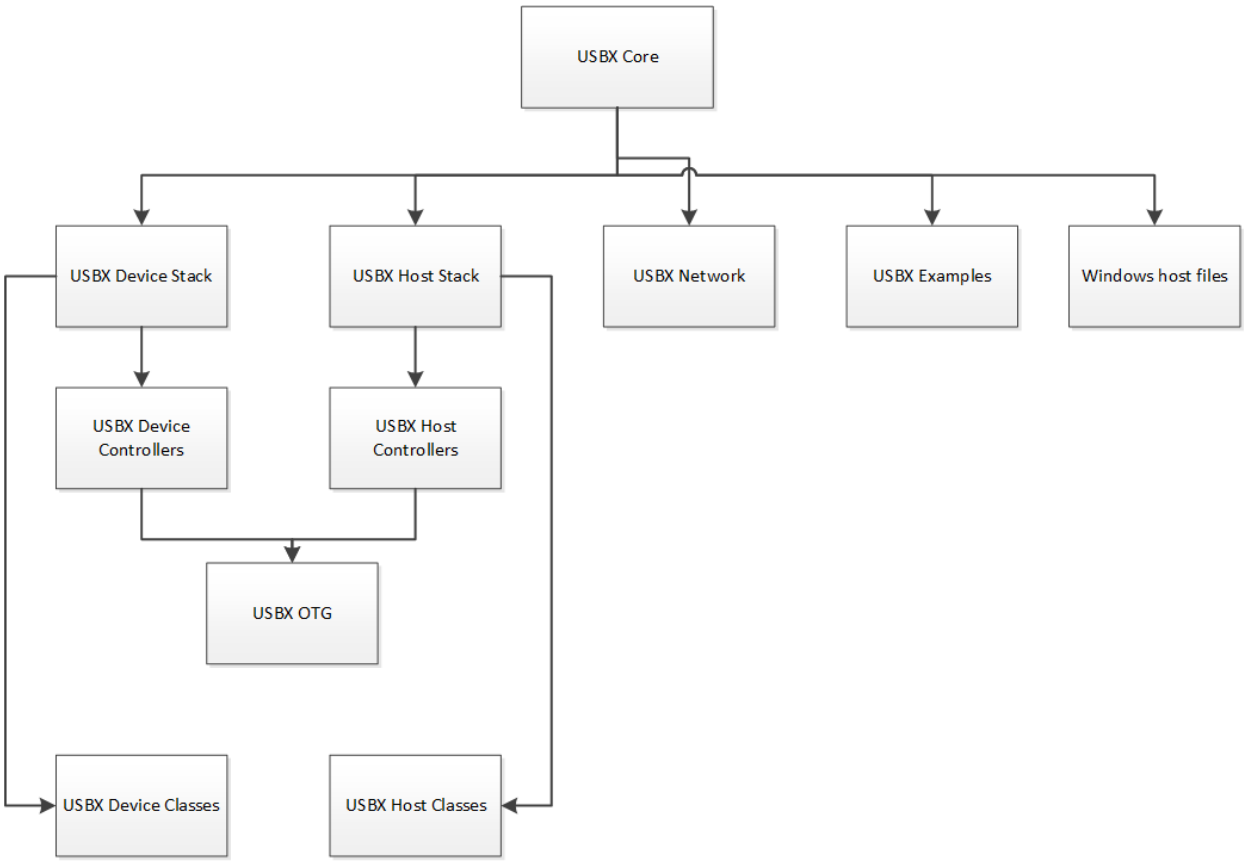
- UX_PERIODIC_RATE：此值表示特定硬件平台每秒的时钟周期数。默认值为 1000，表示每毫秒一个时钟周期。
- UX_MAX_CLASS_DRIVER：此值是 USBX 可加载的最大类数。此值表示类容器，而不是类的实例数。例如，如果特定的 USBX 实现需要集线器类、打印机类和存储类，则无论属于这些类的设备数量为多少，UX_MAX_CLASS_DRIVER 值都可以设置为 3。
- UX_MAX_HCD：此值表示系统中可用的不同主机控制器的数量。对于 USB 1.1 支持，此值主要设置为 1。对于 USB 2.0 支持，此值可以大于 1。此值表示同时运行的并发主机控制器的数量。例如，如果有两个 OHCI 实例正在运行，或者一个 EHCI 控制器和一个 OHCI 控制器正在运行，则 UX_MAX_HCD 应设置为 2。
- UX_MAX_DEVICES：此值表示可连接到 USB 接口的最大设备数。通常，理论上单个 USB 接口可连接的最大设备数为 127。可以缩减此值来节省内存。应注意的是，无论系统中的 USB 总线数量为多少，此值都表示设备总数。
- UX_MAX_ED：此值表示控制器池中的最大 ED 数目。此数目仅分配给一个控制器。如果存在多个控制器实例，则每个控制器都将使用此值。
- UX_MAX_TD and UX_MAX_ISO_TD：此值表示控制器池中常规 TD 和常时等量 TD 的最大数目。此数目仅分配给一个控制器。如果存在多个控制器实例，则每个控制器都将使用此值。
- UX_THREAD_STACK_SIZE：此值为 USBX 线程的堆栈大小（以字节为单位）。它通常可以是 1024 字节或 2048 字节，具体取决于所用的处理器和主机控制器。
- UX_HOST_ENUM_THREAD_STACK_SIZE：此值为 USB 主机枚举线程的堆栈大小。如果未设置此符号，则 USBX 主机枚举线程堆栈大小设置为 UX_THREAD_STACK_SIZE。
- UX_HOST_HCD_THREAD_STACK_SIZE：此值为 USB 主机 HCD 线程的堆栈大小。如果未设置此符号，则 USBX 主机 HCD 线程堆栈大小设置为 UX_THREAD_STACK_SIZE。
- UX_THREAD_PRIORITY_ENUM：这是用于监视总线拓扑的 USBX 枚举线程的 ThreadX 优先级值。
- UX_THREAD_PRIORITY_CLASS：这是标准 USBX 线程的 ThreadX 优先级值。
- UX_THREAD_PRIORITY_KEYBOARD：这是 USBX HID 键盘类的 ThreadX 优先级值。
- UX_THREAD_PRIORITY_HCD：这是主机控制器线程的 ThreadX 优先级值。
- UX_NO_TIME_SLICE：此值实际上定义将用于线程的时间片。例如，如果定义为 0，则 ThreadX 目标端口不使用时间片。
- UX_MAX_HOST_LUN：此值表示主机存储类驱动程序中表示的最大 SCSI 逻辑单元数。
- UX_HOST_CLASS_STORAGE_INCLUDE_LEGACY_PROTOCOL_SUPPORT：如果定义此选项，则此值包含用于处理使用 CB 或 CBI 协议的存储设备（例如软盘）的代码。默认关闭此支持，因为这些协议已过时并由仅批量传输 (Bulk Only Transport, BOT) 协议取代，几乎所有现代存储设备都使用该协议。
- UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE：如果定义此选项，则此值会导致

ux_host_class_hid_keyboard_key_get 仅报告按键更改，即按下按键和释放按键。默认情况下，它仅在某个按键已按住时进行报告。

- UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE_REPORT_KEY_DOWN_ONLY：仅在定义了 UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE 时使用。如果定义此选项，将导致 ux_host_class_hid_keyboard_key_get 仅报告与按下/按住按键相关的更改，而不报告与释放按键相关的更改。
- UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE_REPORT_LOCK_KEYS：仅在定义了 UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE 时使用。如果定义此选项，将导致 ux_host_class_hid_keyboard_key_get 报告与锁定键 (CapsLock/NumLock/ScrollLock) 相关的更改。
- UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE_REPORT_MODIFIER_KEYS：仅在定义了 UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE 时使用。如果定义此选项，将导致 ux_host_class_hid_keyboard_key_get 报告与修改键 (Ctrl/Alt/Shift/GUI) 相关的更改。
- UX_HOST_CLASS_CDC_ECM_NX_PKPOOL_ENTRIES：如果定义此选项，则此值表示 CDC-ECM 主机类中的数据包数。默认值为 16。

源代码树

USBX 文件在多个目录中提供。



为了使文件可通过其名称识别，已采用以下约定：

ux_host_stack	usbx 主机堆栈核心文件
ux_host_class	usbx 主机堆栈类文件
ux_hcd	usbx 主机堆栈控制器驱动程序文件
ux_device_stack	usbx 设备堆栈核心文件

[illegible]	[illegible]
ux_device_class	usbx 设备堆栈类文件
ux_dcd	usbx 设备堆栈控制器驱动程序文件
ux_otg	usbx otg 控制器驱动程序相关文件
ux_pictbridge	usbx pictbridge 文件
ux_utility	usbx 实用工具函数
demo_usbx	USBX 的演示文件

USBX 资源的初始化

USBX 有自己的内存管理器。在初始化 USBX 的主机或设备端之前，需要将内存分配给 USBX。USBX 内存管理器可以容纳可缓存内存的系统。

以下函数可将 USBX 内存资源初始化为具有 128K 常规内存且没有用于缓存安全内存的单独池：

```
/* Initialize USBX Memory */

ux_system_initialize(memory_pointer,(128*1024),UX_NULL,0);
```

ux_system_initialize 的原型如下所示。

```
UINT ux_system_initialize(
    VOID *regular_memory_pool_start,
    ULONG regular_memory_size,
    VOID *cache_safe_memory_pool_start,
    ULONG cache_safe_memory_size);
```

输入参数：

- regular_memory_pool_start：常规内存池的开头。
- regular_memory_size：常规内存池的大小。
- cache_safe_memory_pool_start：缓存安全内存池的开头。
- cache_safe_memory_size：缓存安全内存池的大小。|

并非所有系统都需要定义缓存安全内存。在此类系统中，在初始化过程中为内存指针传递的值将设置为 UX_NULL，并且池的大小会设置为 0。然后，USBX 将使用常规内存池来代替缓存安全池。

在常规内存不是缓存安全内存的系统中，如果控制器（例如，OHCI 控制器、EHCI 控制器及其他控制器）需要执行 DMA 内存，则需要在缓存安全区域中定义一个内存池。

USBX 资源的取消初始化

可以通过释放 USBX 的资源来将其终止。在终止 USBX 之前，需要正确终止所有类和控制器资源。以下函数将取消初始化 USBX 内存资源：

```
/* Unitialize USBX Resources */

ux_system_uninitialize();
```

ux_system_initialize 的原型如下所示。

```
UINT ux_system_uninitialize(VOID);
```

USB 主机控制器的定义

需要定义至少一个 USB 主机控制器，USBX 才能在主机模式下运行。应用程序初始化文件应包含此定义。以下行将定义通用主机控制器。

```
ux_host_stack_hcd_register("ux_hcd_controller",
        ux_hcd_controller_initialize, 0xd0000, 0x0a);
```

ux_host_stack_hcd_register 具有以下原型。

```
UINT ux_host_stack_hcd_register(
    UCHAR *hcd_name,
    UINT (*hcd_initialize_function)(struct UX_HCD_STRUCT *),
    ULONG hcd_param1, ULONG hcd_param2);
```

ux_host_stack_hcd_register 函数具有以下参数。

- hcd_name：控制器名称的字符串
- hcd_initialize_function：控制器的初始化函数
- hcd_param1：通常是控制器使用的 IO 值或内存
- hcd_param2：通常是控制器使用的 IRQ

在上述示例中：

- “ux_hcd_controller”是控制器的名称
- “ux_hcd_controller_initialize”是主机控制器的初始化例程
- 0xd0000 是主机控制器寄存器在内存中多位于的地址
- 0x0a 是主机控制器使用的 IRQ。

下面是将处于主机模式的 USBX 初始化为具有一个主机控制器和多个类的示例。

```
UINT status;

/* Initialize USBX. */
ux_system_initialize(memory_ptr, (128*1024),0,0);

/* The code below is required for installing the USBX host stack. */
status = ux_host_stack_initialize(UX_NULL);

/* If status equals UX_SUCCESS, host stack has been initialized. */

/* Register all the host classes for this USBX implementation. */
status = ux_host_class_register("ux_host_class_hub", ux_host_class_hub_entry);

/* If status equals UX_SUCCESS, host class has been registered. */
status = ux_host_class_register("ux_host_class_storage", ux_host_class_storage_entry);

/* If status equals UX_SUCCESS, host class has been registered. */
status = ux_host_class_register("ux_host_class_printer", ux_host_class_printer_entry);

/* If status equals UX_SUCCESS, host class has been registered. */
status = ux_host_class_register("ux_host_class_audio", ux_host_class_audio_entry);

/* If status equals UX_SUCCESS, host class has been registered. */

/* Register all the USB host controllers available in this system. */
status = ux_host_stack_hcd_register("ux_hcd_controller", ux_hcd_controller_initialize, 0x300000, 0x0a);

/* If status equals UX_SUCCESS, USB host controllers have been registered. */
```

主机类的定义

需要随 USBX 一起定义一个或多个主机类。USB 堆栈配置了 USB 设备后，需要定义 USB 类来驱动 USB 设备。USB 类特定于设备。可能需要定义一个或多个类来驱动 USB 设备，具体取决于 USB 设备描述符中包含的接口数量。

下面是注册集线器类的示例。

```
status = ux_host_stack_class_register("ux_host_class_hub", ux_host_class_hub_entry);
```

函数 ux_host_class_register 具有以下原型。

```
UINT ux_host_stack_class_register(
    UCHAR *class_name,
    UINT (*class_entry_address) (struct UX_HOST_CLASS_COMMAND_STRUCT *));
```

- class_name 是类的名称
- class_entry_address 是类的入口点

在集线器类初始化的示例中：

- “ux_host_class_hub”是集线器类的名称
- ux_host_class_hub_entry 是集线器类的入口点。

疑难解答

USBX 附带演示文件和模拟环境。最好先让演示平台在目标硬件或特定演示平台上运行。

如果演示系统无法工作，请尝试通过以下方法缩小问题的范围。

USBX 版本 ID

当前版本的 USBX 在运行时可供用户和应用程序软件使用。程序员可以通过检查 ux_port.h 文件来获取 USBX 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检查全局字符串 *ux_version_id（在 ux_port.h 中定义）来获取 USBX 版本。***

第 3 章 - USBX 主机堆栈的功能组件

2021/4/30 •

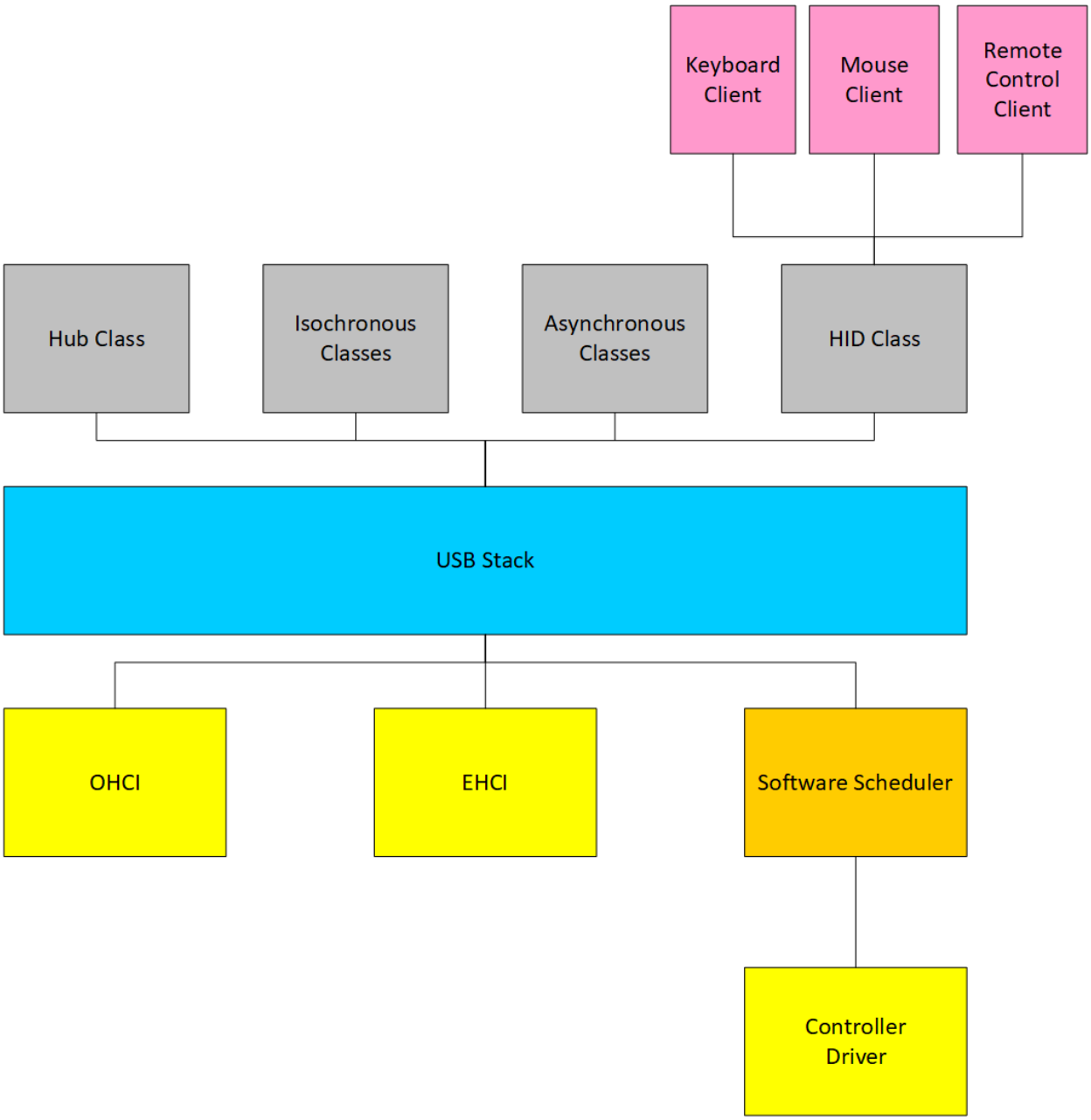
本章从功能角度介绍了高性能 USBX 嵌入式 USB 主机堆栈。

执行概述

USBX 由以下几个组件组成：

- 初始化
- 应用程序接口调用
- 根集线器
- 集线器类
- 主机类
- USB 主机堆栈
- 主机控制器

下图演示了 USBX 主机堆栈。



初始化

若要激活 USBX，必须调用函数 ux_system_initialize。此函数会初始化 USBX 的内存资源。

若要激活 USBX 主机设施，必须调用函数 ux_host_stack_initialize。此函数将对 USBX 主机堆栈使用的所有资源（如 ThreadX 线程、互斥和信号灯）进行初始化。

应该通过应用程序初始化来激活至少一个 USB 主机控制器和一个或多个 USB 类。如果已将类注册到堆栈，并且已调用主机控制器初始化函数，总线即会处于活动状态，并且可以启动设备发现。如果主机控制器的根集线器检测到连接的设备，则会唤醒 USB 枚举线程（负责管理 USB 拓扑），然后继续枚举设备。

由于根集线器和下游集线器的性质，可能无法在主机控制器初始化函数返回时完全配置所有已连接的 USB 设备。枚举所有 USB 设备可能需要几秒钟的时间，在根集线器和 USB 设备之间存在一个或多个集线器的情况下更是如此。

应用程序接口调用

USBX 中有两个级别的 API。

- USB 主机堆栈 API
- USB 主机类 API

通常，USBX 应用程序无需调用任何 USB 主机堆栈 API 函数。因为大多数应用程序将只访问 USB 类 API 函数。

USB 主机堆栈 API

主机堆栈 API 函数负责注册 USBX 组件（主机类和主机控制器）、配置设备以及传输可用设备终结点的请求。

USB 主机类 API

类 API 与每个 USB 类特别相关。USB 类的大多数常见 API 函数提供诸如打开/关闭设备以及读取和写入设备等服务。

根集线器

每个主机控制器实例包含一个或多个 USB 根集线器。根集线器的数量可以由控制器的性质确定，也可以通过从控制器读取特定寄存器来检索。

集线器类

集线器类负责驱动 USB 集线器。USB 集线器可以是独立的集线器，也可以是复合设备的一部分（如键盘或监视器）。集线器可以自供电或由总线供电。由总线供电的集线器最多包含四个下游端口，并且仅允许连接使用小于 100 毫安的自供电设备或由总线供电的设备。集线器可以级联。最多可以将五个集线器相互连接。

USB 主机堆栈

USB 主机堆栈是 USBX 的核心。它有三个主要功能。

- 管理 USB 的拓扑。
- 将 USB 设备绑定到一个或多个类。
- 向类提供一个 API，用于执行设备描述符询问和 USB 传输。

拓扑管理器

当新设备成功连接或某个设备断开连接时，即会唤醒 USB 堆栈拓扑线程。根集线器或常规集线器可以接受设备连接。将设备连接到 USB 后，拓扑管理器将会检索其设备描述符。此描述符包含适用于此设备的可能的配置的数量。大多数设备只有一个配置。某些设备的运行方式可能不同，具体取决于设备连接的端口上可用的电量。如果是这种情况，设备将具有多个可根据可用电量选择的配置。如果设备由拓扑管理器配置，则系统可根据其配置描述符中所指定的电量向其供电。

USB 类绑定

配置设备后，拓扑管理器会让类管理器通过查看设备接口描述符继续发现设备。一个设备可以有一个或多个接口描述符。

接口表示设备中的函数。例如，USB 扬声器有三个接口，一个用于音频流，一个用于音频控制，还有一个用于管理各种扬声器按钮。

类管理器采用两种机制将设备接口加入到一个或多个类中。它可以使用在接口描述符中找到的 PID/VID（产品 ID 和供应商 ID）组合或类/子类/协议的组合。

PID/VID 组合对无法通过泛型类驱动的接口有效。类/子类/协议组合供属于 USB IF 认证类（如打印机、集线器、存储器、音频或 HID）的接口使用。

类管理器包含在 USBX 初始化中已注册的类的列表。类管理器每次调用一个类，直到一个类接受管理该设备的接口。一个类只能管理一个接口。以 USB 音频扬声器为例，类管理器将调用每个接口的所有类。

类接受接口后，将创建该类的新实例。然后，类管理器将搜索该接口的默认备用设置。设备的每个接口可能有一个或多个备用设置。默认情况下将使用备用设置 0，直到类决定更改该设置。

对于默认的备用设置，类管理器将装载该备用设置中包含的所有终结点。如果每个终结点装载成功，类管理器将返回到完成接口初始化的类，从而完成其作业。

USBX API

USB 堆栈会为 USB 类导出一定数量的 API，以便在设备上执行询问，并在特定终结点上执行 USB 传输。本参考手册中详细介绍了这些 API 函数。

主机控制器

主机控制器驱动程序负责驱动特定类型的 USB 控制器。USB 主机控制器内部可能包含多个控制器。例如，某些 Intel PC 芯片包含两个 UHCI 控制器。某些 USB 2.0 控制器除了包含一个 EHCI 控制器实例外，还包含多个 OHCI 控制器实例。

主机控制器仅管理同一个控制器的多个实例。若要驱动大多数 USB 2.0 主机控制器，必须在 USBX 初始化期间初始化 OCHI 控制器和 EHCI 控制器。

主机控制器负责管理以下各项。

- 根集线器
- 电源管理
- 终结点
- 传输

根集线器

根集线器管理负责启动每个控制器端口并确定是否已插入设备。USBX 泛型根集线器使用此功能来询问控制器下游端口。

电源管理

电源管理处理可在联动模式下处理暂停/恢复信号，因此会同时影响所有控制器下游端口，或在控制器提供此功能时单独处理。

终结点

终结点管理用于创建或析构与控制器连接的物理终结点。物理终结点是由控制器分析的内存实体（如果控制器支持主 DMA）或写入控制器的内存实体。物理终结点包含将由控制器执行的事务信息。

传输

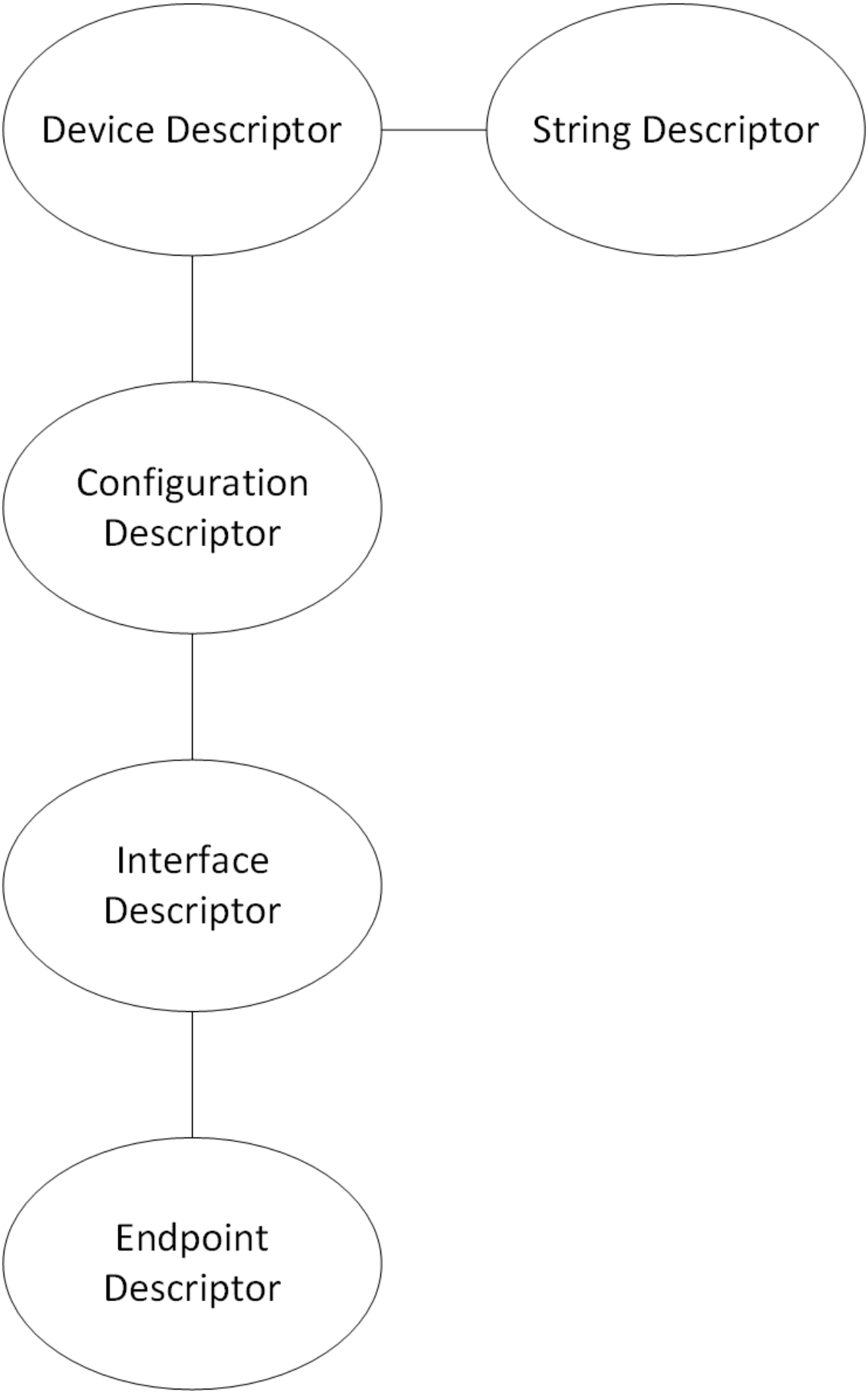
为类提供的转移管理用于对已创建的每个终结点执行事务。每个逻辑终结点都包含一个名为“TRANSFER REQUEST”（转移请求）的组件，用于 USB 转移请求。堆栈使用转移请求来描述事务。然后将此转移请求传递到堆栈和控制器，后者可能会根据控制器的功能将其划分为多个子事务。

USB 设备框架

USB 设备由描述符树表示。描述符有六种主要类型。

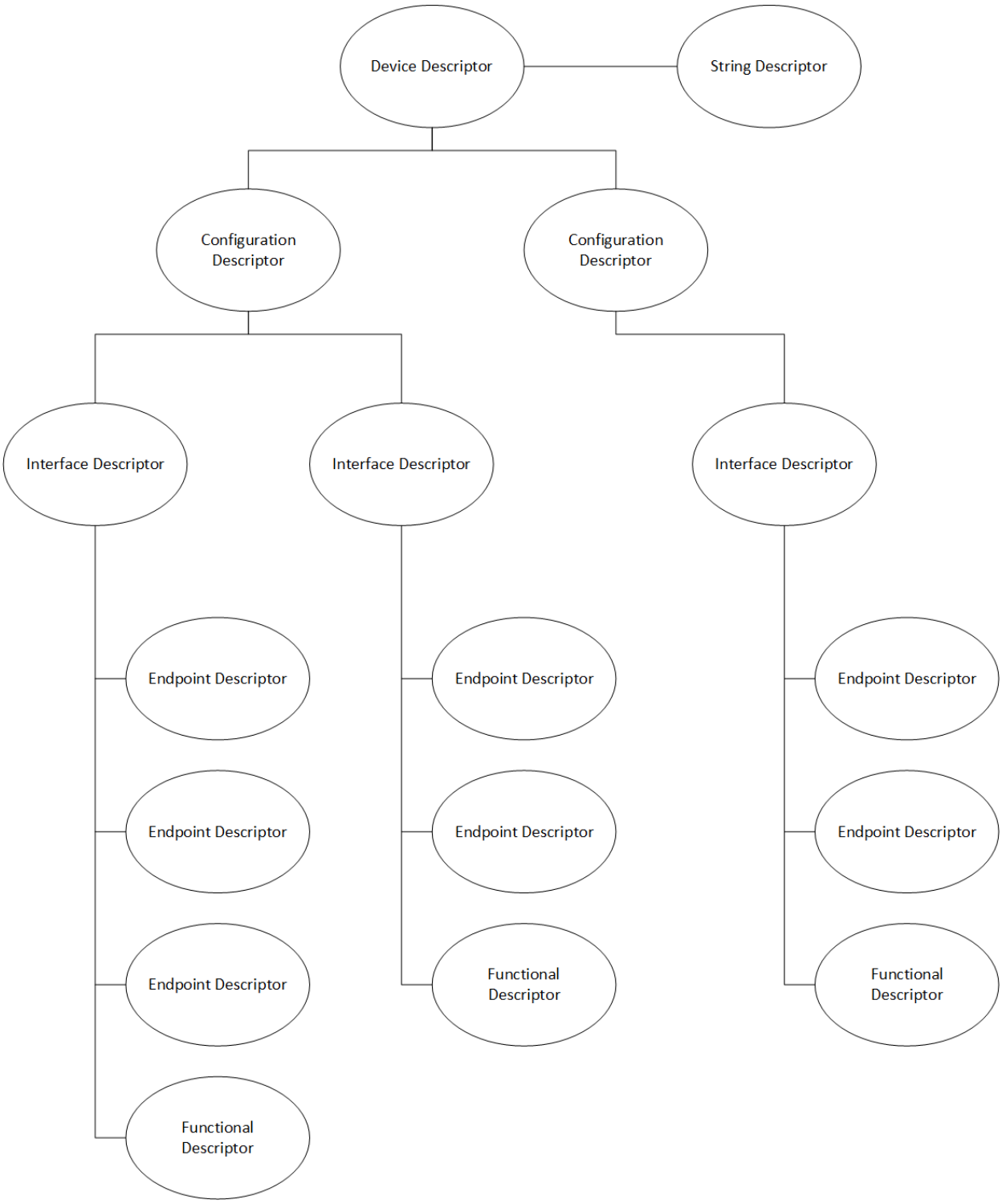
- 设备描述符
- 配置描述符
- 接口描述符
- 终结点描述符
- 字符串描述符
- 功能描述符

USB 设备的描述可能非常简单，如下所示。简单的 USB 设备



在上图中，设备只有一个配置。将单个接口附加到此配置，表示此设备只有一个函数，并且只有一个终结点。附加到设备描述符的是一个字符串描述符，用于提供设备的可见标识。

但是，设备可能更为复杂，可能如下所示。



在上图中，设备将两个配置描述符附加到设备描述符。此设备可能表示它具有两种电源模式，或者可由标准类或专用类驱动。

附加到第一个配置的是两个接口，表示此设备具有两个逻辑函数。第一个函数具有 3 个终结点描述符和一个函数描述符。负责驱动此接口的类可以使用功能说明符来获取通过泛型描述符通常找不到的有关此接口的更多信息。

设备描述符

每个 USB 设备都有一个单一的设备描述符。此描述符包含设备标识、支持的配置数，以及用于配置设备的默认控制终结点的特征。

OFFSET				

OFFSET	𝚝𝚝	𝚝𝚝	𝚝	𝚝𝚝
0	BLength	1	Number	此描述符的大小(以字节为单位)
1	bDescriptorType	1	返回的常量	设备描述符类型
2	bcdUSB	2	BCD	BinaryCoded Dec 中的 USB 规范发布号示例:2.10 相对于 0x210。此字段标识设备及其描述符符合的 USB 规范的版本。
4	bDeviceClass	1	类	类代码(由 USB-IF 分配)。 如果将此字段重置为 0,配置中的每个接口都会指定其自己的类信息,并且各个接口独立运行。 如果将此字段设置为 1 到 0xFE 之间的值,则设备在不同接口上支持不同的类规范,接口可能不会独立运行。此值标识用于聚合接口的类定义。 如果将此字段设置为 0xFF,则设备类特定于供应商。
5	bDeviceSubClass	1	子类	子类代码(由 USB-IF 分配)。 这些代码由 bDeviceClass 字段的值限定。如果将 bDeviceClass 字段重置为 0,则也必须将此字段重置为 0。如果未将 bDeviceClass 字段设置为 0xFF,则会保留所有值以供 USB 分配。

OFFSET	𝕀𝕀	𝕀𝕀	𝕀	𝕀𝕀
6	bDeviceProtocol	1	协议	协议代码(由 USB-IF 分配)。 这些代码由 bDeviceClass 和 bDeviceSubClass 字段的值限定。如果设备支持基于设备使用特定于类的协议，而不是基于接口，则此代码将标识设备按照设备类规范的定义使用的协议。如果将此字段重置为 0，则设备将不会基于设备使用特定于类的协议。但是，设备可能会基于接口使用特定于类的协议。 如果将此字段设置为 0xFF，设备将基于设备使用特定于供应商的协议。
7	bMaxPacketSize0	1	Number	终结点零的最大数据包大小(只有大小为 8、16、32 或 64 字节才有效)
8	idVendor	2	ID	供应商 ID(由 USB-IF 分配)
10	idProduct	2	ID	产品 ID(由制造商分配)
12	bcdDevice	2	BCD	设备版本号(采用二进制编码的十进制形式)
14	iManufacturer	1	索引	用于描述制造商的字符串描述符的索引
15	iProduct	1	索引	描述产品的字符串描述符的索引
16	iSerialNumbe	1	索引	用于描述设备序列号的字符串描述符的索引
17	bNumConfigurations	1	Number	可能的配置数

USBX 按照如下所示定义 USB 设备描述符：

```
typedef struct UX_DEVICE_DESCRIPTOR_STRUCT
{
    UINT      bLength;
    UINT      bDescriptorType;
    USHORT    bcdUSB;
    UINT      bDeviceClass;
    UINT      bDeviceSubClass;
    UINT      bDeviceProtocol;
    UINT      bMaxPacketSize0;
    USHORT    idVendor;
    USHORT    idProduct;
    USHORT    bcdDevice;
    UINT      iManufacturer;
    UINT      iProduct;
    UINT      iSerialNumber;
    UINT      bNumConfigurations;
} UX_DEVICE_DESCRIPTOR;
```

USB 设备描述符是如下所述的设备容器的一部分：

```
typedef struct UX_DEVICE_STRUCT
{
    ULONG ux_device_handle;
    ULONG ux_device_type;
    ULONG ux_device_state;
    ULONG ux_device_address;
    ULONG ux_device_speed;
    ULONG ux_device_port_location;
    ULONG ux_device_max_power;
    ULONG ux_device_power_source;
    UINT ux_device_current_configuration;

    TX_SEMAPHORE ux_device_protection_semaphore;
    struct UX_DEVICE_STRUCT *ux_device_parent;
    struct UX_HOST_CLASS_STRUCT *ux_device_class;
    VOID *ux_device_class_instance;
    struct UX_HCD_STRUCT *ux_device_hcd;
    struct UX_CONFIGURATION_STRUCT *ux_device_first_configuration;
    struct UX_DEVICE_STRUCT *ux_device_next_device;
    struct UX_DEVICE_DESCRIPTOR_STRUCT ux_device_descriptor;
    struct UX_ENDPOINT_STRUCT ux_device_control_endpoint;
    struct UX_HUB_TT_STRUCT ux_device_hub_tt[UX_MAX_TT];
} UX_DEVICE;
```

- ux_device_handle：设备的句柄。这通常是设备的此结构实例的地址。
- ux_device_type：过时值。未使用。
- ux_device_state：设备状态，可具有以下值之一：
  - UX_DEVICE_RESET 0
  - UX_DEVICE_ATTACHED 1
  - UX_DEVICE_ADDRESSED 2
  - UX_DEVICE_CONFIGURED 3
  - UX_DEVICE_SUSPENDED 4
  - UX_DEVICE_RESUMED 5
  - UX_DEVICE_SELF_POWERED_STATE 6
  - UX_DEVICE_SELF_POWERED_STATE 7
  - UX_DEVICE_REMOTE_WAKEUP 8
  - UX_DEVICE_BUS_RESET_COMPLETED 9
  - UX_DEVICE_REMOVED 10

    - UX_DEVICE_FORCE_DISCONNECT 11
- ux_device_address：设备接受 SET_ADDRESS 命令后的地址（1 到 127）。
- ux_device_speed：设备的速度：
    - UX_LOW_SPEED_DEVICE 0
    - UX_FULL_SPEED_DEVICE 1
    - UX_HIGH_SPEED_DEVICE 2
- ux_device_port_location：父设备（根集线器或集线器）端口的索引。
- ux_device_max_power：设备在所选配置中可能采用的最大功率（以毫安为单位）。
- ux_device_power_source：可以是以下两个值之一：
    - UX_DEVICE_BUS_POWERED 1
    - UX_DEVICE_SELF_POWERED 2
- ux_device_current_configuration：该设备当前使用的配置的索引。
- ux_device_parent：此设备的父级的设备容器指针。如果指针为 Null，则父级为控制器的根集线器。
- ux_device_class：指向拥有此设备的类类型的指针。
- ux_device_class_instance：指向拥有此设备的类实例的指针。
- ux_device_hcd：连接此设备的 USB 主机控制器实例。
- ux_device_first_configuration：指向此设备的第一个配置容器的指针。
- ux_device_next_device：指向 USBX 检测到的任何总线上设备列表中的下一个设备的指针。
- ux_device_descriptor：USB 设备描述符。
- ux_device_control_endpoint：此设备使用的默认控制终结点的描述符。
- ux_device_hub_tt：设备的集线器 TT 数组

配置描述符

配置描述符描述有关特定设备配置的信息。USB 设备可能包含一个或多个配置描述符。设备描述符中的 bNumConfigurations 字段表示配置描述符的数量。描述符包含 bConfigurationValue 字段，其中包含一个可用作集配置请求参数的值，这会导致设备采用所述的配置。

此描述符描述了配置提供的接口数量。每个接口都表示设备中的逻辑函数，并且可以独立运行。例如，USB 音频扬声器可能有三个接口，一个用于音频流，一个用于音频控制，还有一个 HID 接口用于管理扬声器的按钮。

当主机对配置描述符发出 GET_DESCRIPTOR 请求时，将返回所有相关的接口和终结点描述符。

OFFSET	ㄩ	ㄩ	ㄧ	ㄩ
0	bLength	1	Number	此描述符的大小（以字节为单位）。
1	bDescriptorType	1	返回的常量	CONFIGURATION
2	wTotalLength	2	Number	为此配置返回的数据的总长度。包括为此配置返回的所有描述符（特定于配置、接口、终结点和类或供应商）的组合长度。
4	bNumInterfaces	1	Number	此配置支持的接口数量。
5	bConfigurationValue	1	Number	用作集参数的值用于选择此配置的配置。

OFFSET	字段	大小	值	说明
6	iConfiguration	1	索引	描述此配置的字符串描述符的索引。
7	bMAttributes	1	Bitmap	配置特征 D7 总线供电 D6 自供电 D5 远程唤醒 D4..0 已保留(重置为 0) 采用总线供电的设备配置和本地源设置了 D7 和 D6。运行时的实际电源可使用获取状态设备请求来确定。 如果设备配置支持远程唤醒，则将 D5 设置为 1。
8	MaxPower	1	毫安	当设备完全正常运行时，此特定配置中 USB 设备从总线获得的最大功率消耗。 以 2 毫安为单位表示(例如 50 = 100 毫安)。 注意：设备配置会报告配置由总线供电还是自供电。 设备状态会报告设备当前是否为自供电。 如果设备与其外部电源断开连接，则会更新设备状态，用于表示该设备不再采用自供电。

USBX 按如下所示定义 USB 配置描述符。

```
typedef struct UX_CONFIGURATION_DESCRIPTOR_STRUCT
{
    UINT bLength;
    UINT bDescriptorType;
    USHORT wTotalLength;
    UINT bNumInterfaces;
    UINT bConfigurationValue;
    UINT iConfiguration;
    UINT bmAttributes;
    UINT MaxPower;
} UX_CONFIGURATION_DESCRIPTOR;
```

USB 配置描述符是如下所示的配置容器的一部分。

```
typedef struct UX_CONFIGURATION_STRUCT
{
    ULONG ux_configuration_handle;
    ULONG ux_configuration_state;
    struct UX_CONFIGURATION_DESCRIPTOR_STRUCT ux_configuration_descriptor;
    struct UX_INTERFACE_STRUCT *ux_configuration_first_interface;
    struct UX_CONFIGURATION_STRUCT *ux_configuration_next_configuration;
    struct UX_DEVICE_STRUCT *ux_configuration_device;
} UX_CONFIGURATION;
```

- ux_configuration_handle：配置的句柄。这通常是配置的此结构实例的地址。
- ux_configuration_state：配置的状态。
- ux_configuration_descriptor：USB 设备描述符。
- ux_configuration_first_interface：指向此配置的第一个接口的指针。
- ux_configuration_next_configuration：指向同一设备的下一个配置的指针。
- ux_configuration_device：指向此配置的设备所有者的指针。

接口描述符

接口描述符描述了配置中的特定接口。接口是 USB 设备中的逻辑函数。配置提供一个或多个接口，每个接口都有零个或多个终结点描述符，用于描述配置中唯一的一组终结点。如果配置支持多个接口，特定接口的终结点描述符将遵循指定配置的 GET_DESCRIPTOR 请求返回的数据中的接口描述符。

接口描述符始终作为配置描述符的一部分返回。接口描述符不能通过 GET_DESCRIPTOR 请求直接访问。

接口可能包含备用设置，可通过这些设置在配置设备后改变终结点和/或其特征。接口的默认设置始终为备用设置零。可以选择类来更改当前的备用设置，以更改接口行为和关联终结点的特征。SET_INTERFACE 请求用于选择备用设置或返回到默认设置。

备用设置允许更改部分设备配置，而其他接口仍保持运行状态。如果某个配置的一个或多个接口有备用设置，则在每个设置中包含单独的接口描述符及其关联的终结点。

如果设备配置包含具有两个备用设置的单个接口，此配置的 GET_DESCRIPTOR 请求将返回配置描述符，然后返回 bInterfaceNumber 和 bAlternateSetting 字段设置为零的接口描述符，接下来返回该设置的终结点描述符，后跟另一个接口描述符及其关联的终结点描述符。第二个接口描述符的 bInterfaceNumber 字段也将设置为零，但第二个接口描述符的 bAlternateSetting 字段将设置为 1，用于表示此备用设置属于第一个接口。

接口可能不含与之关联的任何终结点，在这种情况下，只有默认的控制终结点对该接口有效。

备用设置主要用于更改与接口关联的定期终结点所请求的带宽。例如，USB 扬声器流接口应具有第一个备用设置，其常时等量终结点上的带宽需求为 0。其他备用设置可能会根据音频流频率选择不同的带宽要求。

接口的 USB 描述符如下：

OFFSET				
0	bLength	1	Number	此描述符的大小（以字节为单位）。
1	bDescriptorType	1	返回的常量	接口描述符类型
2	bInterfaceNumber	1	Number	接口数量。从零开始的值，用于标识此配置支持的并发接口数组中的索引。

OFFSET	[illegible]	[illegible]	[illegible]	[illegible]
3	bAltenateSetting	1	Number	用于为先前字段中标识的接口选择替代设置的值。
4	bNumEndpoints	1	Number	此接口使用的终结点数量(排除终结点零)。如果此值为 0，则此接口只使用终结点零。
5	bInterfaceClass	1	类	类代码(由 USB 分配) 如果将此字段重置为 0，该接口不属于任何 USB 指定的设备类。 如果将此字段设置为 0xFF，则接口类特定于供应商。 所有其他值保留供 USB 分配。
6	bInterfaceSubClass	1	子类	子类代码(由 USB 分配)。 这些代码由 bInterfaceClass 字段的值限定。如果将 bInterfaceClass 字段重置为 0，则也必须将此字段重置为 0。 如果未将 bInterfaceClass 字段设置为 0xFF，则会保留所有值以供 USB 分配。
7	bInterfaceProtocol	1	协议	协议代码(由 USB 分配)。这些代码由 bInterfaceClass 和 bInterfaceSubClass 字段的值限定。如果接口支持特定于类的请求，此代码将标识设备按照设备类规范的定义使用的协议。 如果将此字段重置为 0，则设备不在此接口上使用特定于类的协议。如果将此字段设置为 0xFF，设备将对此接口使用特定于供应商的协议。
8	iInterface	1	索引	描述此接口的字符串描述符的索引。

USBX 按如下所示定义 USB 接口描述符。

```
typedef struct UX_INTERFACE_DESCRIPTOR_STRUCT
{
    UINT bLength;
    UINT bDescriptorType;
    UINT bInterfaceNumber;
    UINT bAlternateSetting;
    UINT bNumEndpoints;
    UINT bInterfaceClass
    UINT bInterfaceSubClass;
    UINT bInterfaceProtocol;
    UINT iInterface;
} UX_INTERFACE_DESCRIPTOR;
```

USB 接口描述符是如下所述的接口容器的一部分。

```
typedef struct UX_INTERFACE_STRUCT
{
    ULONG ux_interface_handle;
    ULONG ux_interface_state;
    ULONG ux_interface_current_alternate_setting;
    struct UX_INTERFACE_DESCRIPTOR_STRUCT ux_interface_descriptor;
    struct UX_HOST_CLASS_STRUCT    *ux_interface_class;
    VOID    *ux_interface_class_instance;
    struct UX_ENDPOINT_STRUCT    *ux_interface_first_endpoint;
    struct UX_INTERFACE_STRUCT    *ux_interface_next_interface;
    struct UX_CONFIGURATION_STRUCT    *ux_interface_configuration;
} UX_INTERFACE;
```

- ux_interface_handle：接口的句柄。这通常是接口的此结构实例的地址。
- ux_interface_state：接口的状态。
- ux_interface_descriptor：USB 接口描述符。
- ux_interface_class：指向拥有此接口的类类型的指针。
- ux_interface_class_instance：指向拥有此设备的类实例的指针。
- ux_interface_first_endpoint：指向注册到此接口的第一个终结点的指针。
- ux_interface_next_interface：指向与配置关联的下一个接口的指针。
- ux_interface_configuration：指向此接口的配置所有者的指针。

终结点描述符

与接口关联的每个终结点都有自己的终结点描述符。此描述符包含主机堆栈确定以下内容所需的信息：每个终结点的带宽要求、与终结点关联的最大有效负载、主机堆栈的周期和方向。终结点描述符始终由配置的 GET_DESCRIPTOR 命令返回。

与该接口关联的默认控制终结点不会算作与该接口关联的终结点，因此不会在此描述符中返回。

当主机软件请求更改接口的备用设置时，将根据新的备用设置来修改所有关联的终结点及其 USB 资源。

不能在接口之间共享除默认控制终结点之外的终结点。

OFFSET				
0	bLength	1	Number	此描述符的大小（以字节为单位）。
1	bDescriptorType	1	返回的常量	终结点描述符类型。

OFFSET	口	口	口	口
2	bEndpointAddress	1	端点	此描述符所描述的 USB 设备上终结点的地址。该地址按如下方式进行编码： 位 3...0：终结点编号 位 6...4：已保留，重置为零 位 7：方向，已为控制终结点忽略 0 = 输出终结点 1 = 输入终结点
3	bmAttributes	1	Bitmap	当使用 bConfigurationValue 字段配置终结点时，此字段将描述其属性。位 1..0：传送类型 00 = 控制 01 = 常时等量 10 = 批量 11 = 中断 如果不是常时等量终结点，则保留位 5..2，并且必须设置为零。如果是常时等量，则按如下方式定义： 位 3..2：同步类型 00 = 不同步 01 = 异步 10 = 自适应 11 = 同步 位 5..4：使用类型 00 = 数据终结点 01 = 反馈终结点 10 = 隐式反馈数据终结点 11 = 已保留

OFFSET	[illegible]	[illegible]	[illegible]	[illegible]
4	wMaxPacketSize	2	Number	选择此配置时，此终结点可以发送或接收的最大数据包大小。 对于常时等量终结点，此值用于在计划中保留每个(微)帧数据负载所需的总线时间。通常，管道实际使用的带宽可能比保留的带宽更少。如有必要，设备会通过其非 USB 定义的正常机制报告已使用的实际带宽。 对于所有终结点，位 10..0 指定最大数据包大小(以字节为单位)。 对于高速常时等量和中断端点： 位 12..11 指定每个微帧额外的事务机会数：00 = 无(每个微帧 1 个事务) 01 = 1 个额外(每个微帧 2 个) 10 = 2 个额外(每个微帧 3 个) 11 = 已保留 位 15..13 表示已保留，并且必须设置为零。

OFFSET	𝕀𝕀	𝕀𝕀	𝕀	𝕀𝕀
6	bInterval	1	Number	轮询终结点用于数据传输的数区间。用帧或微帧表示，具体取决于设备运行速度（例如，1 毫秒或 125 微妙单位）。对于全速/高速常时等量终结点，此值必须在 1 到 16 的范围内。bInterval 用作 2bInterval-1 值的指数；例如，bInterval 4 表示 8 的一个周期 (24-1)。对于全速/低速中断终结点，此字段的值可以是 1 到 255。对于高速中断终结点，bInterval 将用作 2bInterval-1 值的指数；例如，bInterval 4 表示 8 的一个周期 (24-1)。此值必须介于 1 到 16 之间。对于高速大容量/控制终结点，bInterval 指定终结点的最大 NAK 率。值 0 表示终结点永不使用 NAK。其他值表示微帧的每个 bInterval 最多一个 NAK。此值必须介于 0 到 255 之间。

USBX 按照如下所示定义 USB 终结点描述符：

```
typedef struct UX_ENDPOINT_DESCRIPTOR_STRUCT
{
    UINT bLength;
    UINT bDescriptorType;
    UINT bEndpointAddress;
    UINT bmAttributes;
    USHORT wMaxPacketSize;
    UINT bInterval;
} UX_ENDPOINT_DESCRIPTOR;
```

USB 终结点描述符是如下所述的终结点容器的一部分。

```
typedef struct UX_ENDPOINT_STRUCT {
    ULONG    ux_endpoint_handle;
    ULONG    ux_endpoint_state;
    VOID    *ux_endpoint_ed;
    struct UX_ENDPOINT_DESCRIPTOR_STRUCT    ux_endpoint_descriptor;
    struct UX_ENDPOINT_STRUCT    *ux_endpoint_next_endpoint;
    struct UX_INTERFACE_STRUCT    *ux_endpoint_interface;
    struct UX_DEVICE_STRUCT    *ux_endpoint_device;
    struct UX_TRANSFER REQUEST_STRUCT    ux_endpoint_transfer request;
} UX_ENDPOINT;
```

- ux_endpoint_handle：终结点的句柄。这通常是终结点的此结构实例的地址。
- ux_endpoint_state：终结点的状态。
- ux_endpoint_ed：指向主机控制器层上物理终结点的指针。
- ux_endpoint_descriptor：USB 终结点描述符。
- ux_endpoint_next_endpoint：指向属于同一接口的下一个终结点的指针。
- ux_endpoint_interface：指向拥有此终结点接口的接口的指针。
- ux_endpoint_device：指向父设备容器的指针。
- ux_endpoint_transfer request：USB 转移请求，用于向/从设备发送/接收数据。

字符串描述符

字符串描述符是可选的。如果设备不支持字符串描述符，则必须将对设备、配置和接口描述符中的字符串描述符的所有引用重置为零。

字符串描述符使用 UNICODE 编码，因此可以支持多个字符集。USB 设备中的字符串可能支持多种语言。请求字符串描述符时，请求方使用由 USB-IF 定义的语言 ID 指定所需的语言。当前定义的 USB LANGID 列表可在 USBX 附录中找到。所有语言的字符串索引零返回一个字符串描述符，其中包含设备支持的双字节 LANGID 代码的数组。应注意的是，UNICODE 字符串不是以 0 终止。相反，字符串数组的大小是通过从描述符第一个字节中包含的数组大小中减去二来计算的。

USB 字符串描述符 0 按如下方式进行编码。

OFFSET	[illegible]	[illegible]	[illegible]	[illegible]
0	bLength	1	N+2	此描述符的大小（以字节为单位）
1	bDescriptorType	1	返回的常量	字符串描述符类型
2	wLANGID[0]	2	Number	LANGID 代码 0
...	...]	...	...	...
N	wLANGID[n]	2	Number	LANGID 代码 n

其他 USB 字符串描述符按如下方式进行编码。

OFFSET	[illegible]	[illegible]	[illegible]	[illegible]
0	bLength	1	Number	此描述符的大小（以字节为单位）
1	bDescriptorType	1	返回的常量	字符串描述符类型

OFFSET	[illegible]	[illegible]	[illegible]	[illegible]
2	bString	n	Number	UNICODE 编码字符串

USBX 定义了一个非零长度的 USB 字符串描述符，如下所示：

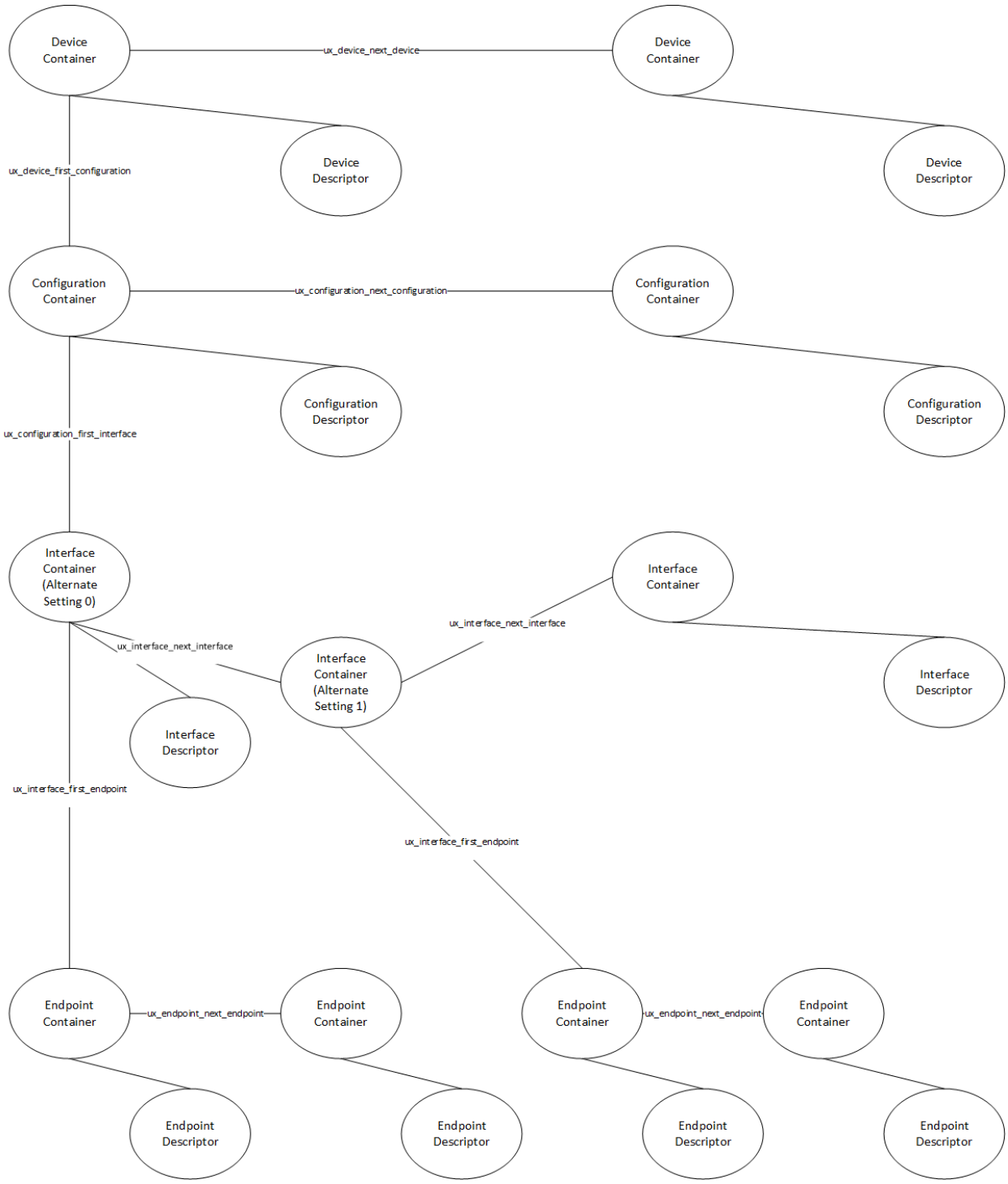
```
typedef struct UX_STRING_DESCRIPTOR_STRUCT
{
    UINT bLength;
    UINT bDescriptorType;
    USHORT bString[1];
} UX_STRING_DESCRIPTOR;
```

功能描述符

功能描述符也称为特定于类的描述符。它们通常使用与泛型描述符相同的基本结构，并且可以向类提供额外的信息。例如，对于 USB 音频扬声器，特定于类的描述符允许音频类为每个备用设置检索支持的音频类型。

内存中的 USBX 设备描述符框架

USBX 在内存中维护大多数设备描述符，即除字符串和函数描述符之外的所有描述符。下图显示了这些描述符如何存储和关联。


Device Container
ux_device_next_device
Device Container
ux_device_first_configuration
Device Descriptor
Device Descriptor
Configuration Container
ux_configuration_next_configuration
Configuration Container
Configuration Descriptor
Configuration Descriptor
ux_configuration_first_interface
Interface Container (Alternate Setting 0)
Interface Container
ux_interface_next_interface
ux_interface_next_interface
Interface Container (Alternate Setting 1)
Interface Descriptor
Interface Descriptor
ux_interface_first_endpoint
ux_interface_first_endpoint
Endpoint Container
ux_endpoint_next_endpoint
Endpoint Container
Endpoint Container
ux_endpoint_next_endpoint
Endpoint Container
Endpoint Descriptor
Endpoint Descriptor
Endpoint Descriptor
Endpoint Descriptor

第 4 章 - USBX 主机服务的说明

2021/4/29 •

ux_host_stack_initialize

为主机操作初始化 USBX。

原型

```
UINT ux_host_stack_initialize(
    UINT (*system_change_function)
    (ULONG, UX_HOST_CLASS *));
```

说明

此函数会初始化 USB 主机堆栈。提供的内存区域会进行设置以供 USBX 内部使用。如果返回 UX_SUCCESS，则 USBX 已准备好用于主机控制器和类注册。

输入参数

- system_change_function 指向用于向应用程序通知设备更改的可选回调例程的指针。

返回值

- UX_SUCCESS (0x00) 初始化成功。
- UX_MEMORY_INSUFFICIENT (0x12) 内存分配失败。

示例

```
UINT status;

/* Initialize USBX for host operation, without notification. */
status = ux_host_stack_initialize(UX_NULL);

/* If status equals UX_SUCCESS, USBX has been successfully initialized for host operation. */
```

ux_host_stack_endpoint_transfer_abort

为终结点中止附加到传输请求的所有事务。

原型

```
UINT ux_host_stack_endpoint_transfer_abort(UX_ENDPOINT *endpoint);
```

说明

此函数会为附加到终结点的特定传输请求，取消所有活动事务或挂起事务。如果传输请求附加了回调函数，则会使用 UX_TRANSACTION_ABORTED 状态调用回调函数。

输入参数

- endpoint 指向终结点的指针。

返回值

- UX_SUCCESS (0x00) 无错误。
- UX_ENDPOINT_HANDLE_UNKNOWN (0x53) 终结点句柄无效。

示例

```
UX_HOST_CLASS_PRINTER *printer;
UINT status;

/* Get the instance for this class. */
printer = (UX_HOST_CLASS_PRINTER *) command ->
    ux_host_class_command_instance;

/* The printer is being shut down. */

printer -> printer_state = UX_HOST_CLASS_INSTANCE_SHUTDOWN;

/* We need to abort transactions on the bulk out pipe. */
status = ux_host_stack_endpoint_transfer_abort
    (printer -> printer_bulk_out_endpoint);

/* If status equals UX_SUCCESS, the operation was successful */
```

ux_host_stack_class_get

获取指向类容器的指针。

原型

```
UINT ux_host_stack_class_get(
    UCHAR *class_name,
    UX_HOST_CLASS **class);
```

说明

此函数返回指向类容器的指针。类需要从 USB 堆栈获取容器，以便在类或应用程序需要打开设备时搜索实例。

> **NOTE**
> class_name 的 C 字符串必须以 NULL 结尾，并且其长度(除开 NULL 终止符本身)不能长过 UX_MAX_CLASS_NAME_LENGTH。

参数

- class_name 指向类名称的指针。
- class 由函数调用更新的指针，其中包含类名称的类容器。

返回值

- UX_SUCCESS (0x00) 无错误，返回类字段时会随指向类容器的指针一起归档。
- UX_HOST_CLASS_UNKNOWN (0x59) 类对于堆栈是未知的。

示例

```
UX_HOST_CLASS *printer_container;
UINT status;

/* Get the container for this class. */
status = ux_host_stack_class_get("ux_host_class_printer", &printer_container);

/* If status equals UX_SUCCESS, the operation was successful */
```

ux_host_stack_class_register

将 USB 类注册到 USB 堆栈。

原型

```
UINT ux_host_stack_class_register(
    UCHAR *class_name,
    UINT (*class_entry_address) (struct UX_HOST_CLASS_COMMAND_STRUCT *));
```

说明

此函数会将 USB 类注册到 USB 堆栈。类必须为 USB 堆栈指定入口点，以便发送命令（如以下内容）。

- UX_HOST_CLASS_COMMAND_QUERY
- UX_HOST_CLASS_COMMAND_ACTIVATE
- UX_HOST_CLASS_COMMAND_DESTROY

> **NOTE**
> class_name 的 C 字符串必须以 NULL 结尾，并且其长度（除开 NULL 终止符本身）不能长过 UX_MAX_CLASS_NAME_LENGTH。

参数

- class_name 指向类名称的指针，在 USBX 的 USB 类下的文件 ux_system_initialize.c 中找到有效入口。
- class_entry_address 类的入口函数的地址。

返回值

- UX_SUCCESS (0x00) 类已成功安装。
- UX_MEMORY_ARRAY_FULL (0x1a) 没有更多内存来存储此类。
- UX_HOST_CLASS_ALREADY_INSTALLED (0x58) 主机类已安装。

示例：

```
UINT status;

/* Register all the classes for this implementation. */
status = ux_host_stack_class_register("ux_host_class_hub", ux_host_class_hub_entry);

/* If status equals UX_SUCCESS, class was successfully installed. */
```

ux_host_stack_class_instance_create

为类容器创建新的类实例。

原型

```
UINT ux_host_stack_class_instance_create(
    UX_HOST_CLASS *class,
    VOID *class_instance);
```

说明

此函数会为类容器创建新的类实例。类的实例不包含在类代码中，以降低类复杂性。相反，每个类实例都附加到位于主堆栈中的类容器。

参数

- class 指向类容器的指针。

- class_instance 指向要创建的类实例的指针。

返回值

- UX_SUCCESS (0x00) 类实例已附加到类容器。

示例

```
UINT status;

UX_HOST_CLASS_PRINTER *printer;

/* Obtain memory for this class instance. */

printer = ux_memory_allocate(UX_NO_ALIGN, sizeof(UX_HOST_CLASS_PRINTER));

if (printer == UX_NULL)
    return(UX_MEMORY_INSUFFICIENT);

/* Store the class container into this instance. */
printer -> printer_class = command -> ux_host_class;

/* Create this class instance. */
status = ux_host_stack_class_instance_create(printer -> printer_class, (VOID *)printer);

/* If status equals UX_SUCCESS, the class instance was successfully created and attached to the class
container. */
```

ux_host_stack_class_instance_destroy

为类容器销毁类实例。

原型

```
UINT ux_host_stack_class_instance_destroy(
    UX_HOST_CLASS *class,
    VOID *class_instance);
```

说明

此函数会为类容器销毁类实例。

参数

- class 指向类容器的指针。
- class_instance 指向要销毁的实例的指针。

返回值

- UX_SUCCESS (0x00) 类实例已销毁。
- UX_HOST_CLASS_INSTANCE_UNKNOWN (0x5b) 类实例未附加到类容器。

示例

```
UINT status;
UX_HOST_CLASS_PRINTER *printer;

/* Get the instance for this class. */
printer = (UX_HOST_CLASS_PRINTER *) command -> ux_host_class_command_instance;

/* The printer is being shut down. */
printer -> printer_state = UX_HOST_CLASS_INSTANCE_SHUTDOWN;

/* Destroy the instance. */
status = ux_host_stack_class_instance_destroy(printer -> printer_class, (VOID *) printer);

/* If status equals UX_SUCCESS, the class instance was successfully destroyed. */
```

ux_host_stack_class_instance_get

获取特定类的类实例指针。

原型

```
UINT ux_host_stack_class_instance_get(
    UX_HOST_CLASS *class,
    UINT class_index,
    VOID **class_instance);
```

说明

此函数会获取特定类的类实例指针。类的实例不包含在类代码中，以降低类复杂性。相反，每个类实例都附加到类容器。此函数用于在类容器中搜索类实例。

参数

- class 指向类容器的指针。
- class_index 要由函数调用在容器的附加类列表中使用的索引。
- class_instance 指向由函数调用返回的实例的指针。

返回值

- UX_SUCCESS (0x00) 找到类实例。
- UX_HOST_CLASS_INSTANCE_UNKNOWN (0x5b) 没有更多类实例附加到类容器。

示例

```
UINT status;

UX_HOST_CLASS_PRINTER *printer;

/* Obtain memory for this class instance. */
printer = ux_memory_allocate(UX_NO_ALIGN, sizeof(UX_HOST_CLASS_PRINTER));

if (printer == UX_NULL) return(UX_MEMORY_INSUFFICIENT);

/* Search for instance index 2. */
status = ux_host_stack_class_instance_get(class, 2, (VOID *) printer);

/* If status equals UX_SUCCESS, the class instance was found. */
```

ux_host_stack_device_configuration_get

获取指向配置容器的指针。

原型

```
UINT ux_host_stack_device_configuration_get(
    UX_DEVICE *device,
    UINT configuration_index,
    UX_CONFIGURATION *configuration);
```

说明

此函数会基于设备句柄和配置索引返回配置容器。

参数

- device 指向拥有所请求配置的设备容器的指针。
- configuration_index 要搜索的配置的索引。
- configuration 指向要返回的配置容器的指针地址。

返回值

- UX_SUCCESS (0x00) 找到配置。
- UX_DEVICE_HANDLE_UNKNOWN (0x50) 设备容器不存在。
- UX_CONFIGURATION_HANDLE_UNKNOWN (0x51) 索引的配置句柄不存在。

示例

```
UINT status;

UX_HOST_CLASS_PRINTER *printer;

/* If the device has been configured already, we don't need to do it
again. */

if (printer -> printer_device -> ux_device_state == UX_DEVICE_CONFIGURED)
    return(UX_SUCCESS);

/* A printer normally has one configuration, retrieve 1st configuration only. */

status = ux_host_stack_device_configuration_get(printer -> printer_device,
    0, configuration);

/* If status equals UX_SUCCESS, the configuration was found. */
```

ux_host_stack_device_configuration_select

为设备选择特定配置。

原型

```
UINT ux_host_stack_device_configuration_select (UX_CONFIGURATION *configuration);
```

说明

此函数会为设备选择特定配置。当此配置设置为设备，默认情况下会在设备上激活每个设备接口及其关联备用设置 0。如果设备/接口类希望更改特定接口的设置，则需要发出 ux_host_stack_interface_setting_select 服务调用。

参数

- configuration 指向要为此设备启用的配置容器的指针。

返回值

- UX_SUCCESS (0x00) 配置选择成功。
- UX_CONFIGURATION_HANDLE_UNKNOWN (0x51) 配置句柄不存在。
- UX_OVER_CURRENT_CONDITION (0x43) 此配置的总线上存在过电流状况。

示例

```
UINT status;

UX_HOST_CLASS_PRINTER *printer;

/* If the device has been configured already, we don't need to do it again. */
if (printer -> printer_device -> ux_device_state == UX_DEVICE_CONFIGURED)
    return(UX_SUCCESS);

/* A printer normally has one configuration - retrieve 1st configuration only. */
status = ux_host_stack_device_configuration_get(printer -> printer_device, 0,configuration);

/* If status equals UX_SUCCESS, the configuration selection was successful. */

/* If valid configuration, ask USBX to set this configuration. */
status = ux_host_stack_device_configuration_select(configuration);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_stack_device_get

获取指向设备容器的指针。

原型

```
UINT ux_host_stack_device_get(
    ULONG device_index,
    UX_DEVICE *device);
```

说明

此函数会基于其索引返回设备容器。设备索引从 0 开始。请注意，索引为 ULONG，因为可以有多个控制器并且字节索引可能不够。设备索引不应与特定于总线的设备地址混淆。

参数

- device_index 设备的索引。
- device 要返回的设备容器的指针地址。

返回值

- UX_SUCCESS (0x00) 设备容器存在并返回
- UX_DEVICE_HANDLE_UNKNOWN (0x50) 设备未知

示例

```
UINT status;

/* Locate the first device in USBX. */
status = ux_host_stack_device_get(0, device);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_stack_interface_endpoint_get

获取终结点容器。

原型

```
UINT ux_host_stack_interface_endpoint_get(
    UX_INTERFACE *interface,
    UINT endpoint_index,
    UX_ENDPOINT *endpoint);
```

说明

此函数会基于接口句柄和终结点索引返回终结点容器。假设在搜索终结点之前选择了接口的备用设置或是在使用默认设置。

参数

- interface 指向包含所请求终结点的接口容器的指针。
- endpoint_index 此接口中的终结点的索引。
- endpoint 要返回的终结点容器的地址。

返回值

- UX_SUCCESS (0x00) 终结点容器存在并返回。
- UX_INTERFACE_HANDLE_UNKNOWN (0x52) 指定接口不存在。
- UX_ENDPOINT_HANDLE_UNKNOWN (0x53) 终结点索引不存在。

示例

```
UINT status;
UX_HOST_CLASS_PRINTER *printer;

for(endpoint_index = 0;
    endpoint_index < printer -> printer_interface ->
        ux_interface_descriptor.bNumEndpoints;
    endpoint_index++)
{
    status = ux_host_stack_interface_endpoint_get (printer -> printer_interface,
        endpoint_index, &endpoint);

    if (status == UX_SUCCESS)
    {
        /* Check if endpoint is bulk and OUT. */
        if (((endpoint -> ux_endpoint_descriptor.bEndpointAddress &
            UX_ENDPOINT_DIRECTION) == UX_ENDPOINT_OUT) &&
            ((endpoint -> ux_endpoint_descriptor.bmAttributes &
            UX_MASK_ENDPOINT_TYPE) == UX_BULK_ENDPOINT))
        return(UX_SUCCESS);
    }
}
```

ux_host_stack_hcd_register

将 USB 控制器注册到 USB 堆栈。

原型

```
UINT ux_host_stack_hcd_register(
    UCHAR *hcd_name,
    UINT (*hcd_function)(struct UX_HCD_STRUCT *),
    ULONG hcd_param1, ULONG hcd_param2);
```

说明

此函数会将 USB 控制器注册到 USB 堆栈。它主要分配此控制器使用的内存，并将初始化命令传递到控制器。

参数

- hcd_name 主机控制器的名称
- hcd_function 主机控制器中负责进行初始化的函数。
- hcd_param1 hcd 使用的 IO 或内存资源。
- hcd_param2 主机控制器使用的 IRQ。

返回值

- UX_SUCCESS (0x00) 控制器已正确初始化。
- UX_MEMORY_INSUFFICIENT (0x12) 此控制器内存不足。
- UX_PORT_RESET_FAILED (0x31) 控制器重置失败。
- UX_CONTROLLER_INIT_FAILED (0x32) 控制器未能正确初始化。

示例

```
UINT status;

/* Initialize a host controller mapped at address 0xd0000 and using IRQ 10. */

status = ux_host_stack_hcd_register("ux_hcd_controller",
    ux_hcd_controller_initialize, 0xd0000, 0x0a);

/* If status equals UX_SUCCESS, the controller was initialized properly. */

/* Note that the application must also setup a call to the
    interrupt handler for the controller.
    The function for the controller is called _ux_hch_controller_interrupt_handler. */
```

ux_host_stack_configuration_interface_get

获取接口容器指针。

原型

```
UINT ux_host_stack_configuration_interface_get (
    UX_CONFIGURATION *configuration,
    UINT interface_index,
    UINT alternate_setting_index,
    UX_INTERFACE **interface);
```

说明

此函数会基于配置句柄、接口索引和备用设置索引返回接口容器。

参数

- configuration 指向拥有接口的配置容器的指针。
- interface_index 要搜索的接口索引。
- alternate_setting_index 要搜索的接口中的备用设置。
- interface 要返回的接口容器指针的地址。

返回值

- UX_SUCCESS (0x00) 接口索引的接口容器和备用设置已找到并返回。
- UX_CONFIGURATION_HANDLE_UNKNOWN (0x51) 配置不存在。
- UX_INTERFACE_HANDLE_UNKNOWN (0x52) 接口不存在。

示例

```
UINT status;

/* Search for the default alternate setting on the first interface for the printer. */
status = ux_host_stack_configuration_interface_get(configuration, 0, 0,
    &printer -> printer_interface);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_stack_interface_setting_select

为接口选择备用设置。

原型

```
UINT ux_host_stack_interface_setting_select(UX_INTERFACE *interface);
```

说明

此函数会为属于所选配置的给定接口选择特定备用设置。此函数用于从默认备用设置更改为新设置或恢复为默认备用设置。选择新的备用设置后，以前的终结点特征无效，应重新加载。

输入参数

- interface 指向要选择其备用设置的接口容器的指针。

返回值

- UX_SUCCESS (0x00) 已成功选择此接口的备用设置。
- UX_INTERFACE_HANDLE_UNKNOWN (0x52) 接口不存在。

示例

```
UINT status;

/* Select a new alternate setting for this interface. */
status = ux_host_stack_interface_setting_select(interface);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_stack_transfer_request_abort

中止挂起的传输请求。

原型

```
UINT ux_host_stack_transfer_request_abort(UX_TRANSFER REQUEST *transfer request);
```

说明

此函数会中止以前已提交的挂起传输请求。此函数仅取消特定传输请求。此函数的回调会具有 UX_TRANSFER REQUEST_STATUS_ABORT 状态。

参数

- transfer request 指向要中止的传输请求的指针。

返回值

- UX_SUCCESS (0x00) 已取消此传输请求的 USB 传输。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_stack_transfer_request_abort(transfer request);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_stack_transfer_request

请求 USB 传输。

原型

```
UINT ux_host_stack_transfer_request(UX_TRANSFER REQUEST *transfer request);
```

说明

此函数会执行 USB 事务。进入时，传输请求会提供为此事务选择的终结点管道，以及与传输关联的参数（数据有效负载、事务长度）。对于控制管道，事务会被阻止，仅当控制传输的三个阶段已完成或存在以前的错误时才会返回。对于其他管道，USB 堆栈会在 USB 上计划事务，但不会等待其完成。非阻止管道的每个传输请求都必须指定完成例程处理程序。

当函数调用返回时，应检查传输请求的状态，因为它包含事务的结果。

输入参数

- transfer request 指向传输请求的指针。传输请求包含传输所需的所有必要信息。

返回值

- UX_SUCCESS (0x00) 已正确计划此传输请求的 USB 传输。传输请求完成后，应检查传输请求的状态代码。
- UX_MEMORY_INSUFFICIENT (0x12) 内存不足，无法分配所需控制器资源。
- UX_TRANSFER_NOT_READY (0x25) 设备处于无效状态 - 状态必须为 ATTACHED、ADDRESSED 或 CONFIGURED。

示例：

```
UINT status;

/* Create a transfer request for the SET_CONFIGURATION request. No data for this request. */
transfer_request -> ux_transfer_request_requested_length = 0;
transfer_request -> ux_transfer_request_function = UX_SET_CONFIGURATION;
transfer_request -> ux_transfer_request_type =
    UX_REQUEST_OUT |
    UX_REQUEST_TYPE_STANDARD |
    UX_REQUEST_TARGET_DEVICE;

transfer_request -> ux_transfer_request_value = (USHORT)
    configuration -> ux_configuration_descriptor.bConfigurationValue;
transfer_request -> ux_transfer_request_index = 0;

/* Send request to HCD layer. */
status = ux_host_stack_transfer_request(transfer_request);

/* If status equals UX_SUCCESS, the operation was successful. */
```

第 5 章 - USBX 主机类 API

2021/4/29 •

本章介绍 USBX 主机类的所有公开的 API。详细介绍了以下每个类的 API。

- HID 类
- CDC-ACM 类
- CDC-ECM 类
- 存储类

ux_host_class_hid_client_register

向 HID 类注册一个 HID 客户端。

原型

```
UINT ux_host_class_hid_client_register(
    UCHAR *hid_client_name,
    UINT (*hid_client_handler)
    (struct UX_HOST_CLASS_HID_CLIENT_COMMAND_STRUCT *));
```

说明

此函数用于向 HID 类注册 HID 客户端。在从该设备请求数据之前，HID 类需要在 HID 设备与 HID 客户端之间找到匹配项。

> **NOTE**
> hid_client_name 的 C 字符串必须以 NULL 终止，并且其长度（除去 NULL 终止符本身）不能超过 UX_HOST_CLASS_HID_MAX_CLIENT_NAME_LENGTH。

参数

- **hid_client_name** 指向 HID 客户端名称的指针。
- **hid_client_handler** 指向 HID 客户端处理程序的指针。

返回值

- **UX_SUCCESS** (0x00) 已完成数据传输
- **UX_MEMORY_INSUFFICIENT** (0x12) 客户端内存分配失败。
- **UX_MEMORY_ARRAY_FULL** (0x1a) 已注册最大客户端数。
- **UX_HOST_CLASS_ALREADY_INSTALLED** (0x58) 此类已存在

示例

```
UINT status;

/* The following example illustrates how to register a HID client, in
this case a USB mouse, to the HID class. */

status = ux_host_class_hid_client_register("ux_host_class_hid_client_mouse",
    ux_host_class_hid_mouse_entry);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_report_callback_register

从 HID 类注册回调。

原型

```
UINT ux_host_class_hid_report_callback_register(
    UX_HOST_CLASS_HID *hid,
    UX_HOST_CLASS_HID_REPORT_CALLBACK *call_back);
```

说明

此函数用于在接收到报表时，将对 HID 类的回调注册到 HID 客户端。

参数

- **hid** 指向 HID 类实例的指针
- **call_back** 指向 call_back 结构的指针

返回值

- **UX_SUCCESS** (0x00) 已完成数据传输
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) 无效 HID 实例。
- **UX_HOST_CLASS_HID_REPORT_ERROR** (0x79) 报表回调注册中有错。

示例

```
UINT status;

/* This example illustrates how to register a HID client, in this case a USB mouse, to the HID class. In
this case, the HID client is asking the HID class to call the client for each usage received in the HID
report. */

call_back.ux_host_class_hid_report_callback_id = 0;
call_back.ux_host_class_hid_report_callback_function = ux_host_class_hid_mouse_callback;

call_back.ux_host_class_hid_report_callback_buffer = UX_NULL;
call_back.ux_host_class_hid_report_callback_flags = UX_HOST_CLASS_HID_REPORT_INDIVIDUAL_USAGE;

call_back.ux_host_class_hid_report_callback_length = 0;

status = ux_host_class_hid_report_callback_register(hid, &call_back);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_periodic_report_start

启动 HID 类实例的定期终结点。

原型

```
UINT ux_host_class_hid_periodic_report_start(UX_HOST_CLASS_HID *hid);
```

说明

此函数用于为绑定到此 HID 客户端的 HID 类的实例启动定期（中断）终结点。HID 类无法启动定期终结点，直到 HID 客户端被激活，因此它将留在 HID 客户端上以启动此终结点来接收报告。

输入参数

- **hid** 指向 HID 类实例的指针。

返回值

- **UX_SUCCESS** (0x00) 定期报告已成功启动。
- **ux_host_class_hid_PERIODIC_REPORT_ERROR** (0x7A) 定期报表中有错误。
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID 类实例不存在。

示例

```
UINT status;

/* The following example illustrates how to start the periodic
endpoint. */

status = ux_host_class_hid_periodic_report_start(hid);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_periodic_report_stop

停止 HID 类实例的定期终结点。

原型

```
UINT ux_host_class_hid_periodic_report_stop(UX_HOST_CLASS_HID *hid);
```

说明

此函数用于为绑定到此 HID 客户端的 HID 类的实例停止定期（中断）终结点。HID 类无法停止定期终结点，直到 HID 客户端被停用，所有资源被释放，因此它将留在 HID 客户端上以停止此终结点。

输入参数

- **hid** 指向 HID 类实例的指针。

返回值

- **UX_SUCCESS** (0x00) 定期报告已成功停止。
- **ux_host_class_hid_PERIODIC_REPORT_ERROR** (0x7A) 定期报表中有错误。
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID 类实例不存在

示例

```
UINT status;

/* The following example illustrates how to stop the periodic endpoint. */

status = ux_host_class_hid_periodic_report_stop(hid);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_report_get

从 HID 类实例获取报表。

原型

```
UINT ux_host_class_hid_report_get(
    UX_HOST_CLASS_HID *hid,
    UX_HOST_CLASS_HID_CLIENT_REPORT *client_report);
```

说明

此函数用于直接从设备接收报表，而无需依赖于定期终结点。此报表来自控制终结点，但其处理方式与在定期终结点上的处理方式相同。

参数

- **hid** 指向 HID 类实例的指针。
- **client_report** 指向 HID 客户端报表的指针。

返回值

- **UX_SUCCESS** (0x00) 已成功接收报表。
- **UX_HOST_CLASS_HID_REPORT_ERROR** (0x70) 客户端报表在传输过程中无效或出错。
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID 类实例不存在。
- **UX_BUFFER_OVERFLOW** (0x5d) 提供的缓冲区不够大，无法容纳未压缩的报表。

示例

```
UX_HOST_CLASS_HID_CLIENT_REPORT input_report;

UINT status;

/* The following example illustrates how to get a report. */

input_report.ux_host_class_hid_client_report = hid_report;
input_report.ux_host_class_hid_client_report_buffer = buffer;
input_report.ux_host_class_hid_client_report_length = length;
input_report.ux_host_class_hid_client_flags = UX_HOST_CLASS_HID_REPORT_INDIVIDUAL_USAGE;

status = ux_host_class_hid_report_get(hid, &input_report);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_report_set

发送报表

原型

```
UINT ux_host_class_hid_report_set(
    UX_HOST_CLASS_HID *hid,
    UX_HOST_CLASS_HID_CLIENT_REPORT *client_report);
```

说明

此函数用于将报表直接发送到设备。

参数

- **hid** 指向 HID 类实例的指针。

- client_report 指向 HID 客户端报表的指针。

返回值

- UX_SUCCESS (0x00) 已成功发送报表。
- UX_HOST_CLASS_HID_REPORT_ERROR (0x70) 客户端报表在传输过程中无效或出错。
- UX_HOST_CLASS_INSTANCE_UNKNOWN (0x5b) HID 类实例不存在。
- UX_HOST_CLASS_HID_REPORT_OVERFLOW (0x5d) 提供的缓冲区不够大，无法容纳未压缩的报表。

示例

```
/* The following example illustrates how to send a report. */

UX_HOST_CLASS_HID_CLIENT_REPORT input_report;

input_report.ux_host_class_hid_client_report = hid_report;
input_report.ux_host_class_hid_client_report_buffer = buffer;
input_report.ux_host_class_hid_client_report_length = length;
input_report.ux_host_class_hid_client_report_flags = UX_HOST_CLASS_HID_REPORT_INDIVIDUAL_USAGE;

status = ux_host_class_hid_report_set(hid, &input_report);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_mouse_buttons_get

获取鼠标按钮。

原型

```
UINT ux_host_class_hid_mouse_buttons_get(
    UX_HOST_CLASS_HID_MOUSE *mouse_instance,
    ULONG *mouse_buttons);
```

说明

此函数用于获取鼠标按钮

参数

- mouse_instance 指向 HID 鼠标实例的指针。
- mouse_buttons 指向返回按钮的指针。

返回值

- UX_SUCCESS (0x00) 已成功检索鼠标按钮。
- UX_HOST_CLASS_INSTANCE_UNKNOWN (0x5b) HID 类实例不存在。

示例

```
/* The following example illustrates how to obtain mouse buttons. */

UX_HOST_CLASS_HID_MOUSE *mouse_instance;

ULONG mouse_buttons;

status = ux_host_class_hid_mouse_button_get(mouse_instance, &mouse_buttons);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_mouse_position_get

获取鼠标位置。

原型

```
UINT ux_host_class_hid_mouse_position_get(
    UX_HOST_CLASS_HID_MOUSE *mouse_instance,
    SLONG *mouse_x_position,
    SLONG *mouse_y_position);
```

说明

此函数用于获取 x 和 y 坐标中的鼠标位置。

参数

- **mouse_instance** 指向 HID 鼠标实例的指针。
- **mouse_x_position** 指向 x 坐标的指针。
- **mouse_y_position** 指向 y 坐标的指针。

返回值

- **UX_SUCCESS** (0x00) 已成功检索 X & Y 坐标。
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID 类实例不存在。

示例

```
/* The following example illustrates how to obtain mouse coordinates. */

UX_HOST_CLASS_HID_MOUSE *mouse_instance;

SLONG mouse_x_position;
SLONG mouse_y_position;

status = ux_host_class_hid_mouse_position_get(mouse_instance,
    &mouse_x_position, &mouse_y_position);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_keyboard_key_get

获取键盘键和状态。

原型

```
UINT ux_host_class_hid_keyboard_key_get(
    UX_HOST_CLASS_HID_KEYBOARD *keyboard_instance,
    ULONG *keyboard_key,
    ULONG *keyboard_state);
```

说明

此函数用于获取键盘键和状态。

参数

- **keyboard_instance** 指向 HID 键盘实例的指针。
- **keyboard_key** 指向键盘键容器的指针。
- **keyboard_state** 指向键盘状态容器的指针。

返回值

- **UX_SUCCESS** (0x00) **已成功检索键和状态。**
- **UX_ERROR** (0xff) **没有要报告的内容。**
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID **类实例不存在。**

键盘**状态可能有下列值。**

- **UX_HID_KEYBOARD_STATE_KEY_UP** 0x10000
- **UX_HID_KEYBOARD_STATE_NUM_LOCK** 0x0001
- **UX_HID_KEYBOARD_STATE_CAPS_LOCK** 0x0002
- **UX_HID_KEYBOARD_STATE_SCROLL_LOCK** 0x0004
- **UX_HID_KEYBOARD_STATE_MASK_LOCK** 0x0007
- **UX_HID_KEYBOARD_STATE_LEFT_SHIFT** 0x0100
- **UX_HID_KEYBOARD_STATE_RIGHT_SHIFT** 0x0200
- **UX_HID_KEYBOARD_STATE_SHIFT** 0x0300
- **UX_HID_KEYBOARD_STATE_LEFT_ALT** 0x0400
- **UX_HID_KEYBOARD_STATE_RIGHT_ALT** 0x0800
- **UX_HID_KEYBOARD_STATE_ALT** 0x0a00
- **UX_HID_KEYBOARD_STATE_LEFT_CTRL** 0x1000
- **UX_HID_KEYBOARD_STATE_RIGHT_CTRL** 0x2000
- **UX_HID_KEYBOARD_STATE_CTRL** 0x3000
- **UX_HID_KEYBOARD_STATE_LEFT_GUI** 0x4000
- **UX_HID_KEYBOARD_STATE_RIGHT_GUI** 0x8000
- **UX_HID_KEYBOARD_STATE_GUI** 0xa000

示例

```
while (1)
{

    /* Get a key/state from the keyboard. */
    status = ux_host_class_hid_keyboard_key_get(keyboard,
        &keyboard_char, &keyboard_state);

    /* Check if there is something. */
    if (status == UX_SUCCESS)
    {

        #ifdef UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE
            if (keyboard_state & UX_HID_KEYBOARD_STATE_KEY_UP)
            {
                /* The key was released. */
            } else
            {
                /* The key was pressed. */
            }
        #endif

        /* We have a character in the queue. */
        keyboard_queue[keyboard_queue_index] = (UCHAR) keyboard_char;

        /* Can we accept more ? */
        if(keyboard_queue_index < 1024)
            keyboard_queue_index++;
    }

    tx_thread_sleep(10);

}
```

ux_host_class_hid_keyboard_ioctl

对 HID 键盘执行 IOCTL 函数。

原型

```
UINT ux_host_class_hid_keyboard_ioctl(
    UX_HOST_CLASS_HID_KEYBOARD *keyboard_instance,
    ULONG ioctl_function, VOID *parameter);
```

说明

此函数对 HID 键盘执行特定的 ioctl 函数。调用将被阻止，并且仅在出现错误或命令完成时返回。

参数

- **keyboard_instance** 指向 HID 键盘实例的指针。
- **ioctl_function** 要执行的 ioctl 函数。查看以下表获取允许的 ioctl 函数之一。
- **parameter** 指向特定于 ioctl 的参数的指针。

返回值

- **UX_SUCCESS** (0x00) ioctl 函数已成功完成。
- **UX_FUNCTION_NOT_SUPPORTED** (0x54) 未知 IOCTL 函数

IOCTL 函数

- UX_HID_KEYBOARD_IOCTL_SET_LAYOUT
- UX_HID_KEYBOARD_IOCTL_KEY_DECODING_ENABLE
- UX_HID_KEYBOARD_IOCTL_KEY_DECODING_DISABLE

示例:**更改键盘布局**

```
UINT status;

/* This example shows usage of the SET_LAYOUT IOCTL function.
    USBX receives raw key values from the device (these raw values
    are defined in the HID usage table specification) and optionally
    decodes them for application usage. The decoding is performed
    based on a set of arrays that act as maps – which array is used
    depends on the raw key value (i.e. keypad and non-keypad) and
    the current state of the keyboard (i.e. shift, caps lock, etc.). */

/* When the shift condition is not present and the raw key value
    is not within the keypad value range, this array will be used to decode the raw key value. */

static UCHAR keyboard_layout_raw_to_unshifted_map[] =
{
    0,0,0,0,
    'a','b','c','d','e','f','g',
    'h','i','j','k','l','m','n',
    'o','p','q','r','s','t',
    'u','v','w','x','y','z',
    '1','2','3','4','5','6','7','8','9','0',
    0x0d,0x1b,0x08,0x07,0x20,'-','=','[',']',
    '\\','#',';',0x27,'`',',','.','/',0xf0,
    0xbb,0xbc,0xbd,0xbe,0xbf,0xc0,0xc1,0xc2,0xc3,0xc4,0xc5,0xc6,
    0x00,0xf1,0x00,0xd2,0xc7,0xc9,0xd3,0xcf,0xd1,0xcd,0xcd,0xd0,0xc8,0xf2,
    '/','*','-','+',
    0x0d,'1','2','3','4','5','6','7','8','9','0','.','\\',0x00,0x00,'=',
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
};

/* When the shift condition is present and the raw key value
    is not within the keypad value range, this array will be used to decode the raw key value. */

static UCHAR keyboard_layout_raw_to_shifted_map[] =
{
    0,0,0,0,
    'A','B','C','D','E','F','G',
    'H','I','J','K','L','M','N',
    'O','P','Q','R','S','T',
    'U','V','W','X','Y','Z',
    '!','@','#','$','%','^','&','*','(',')',
    0x0d,0x1b,0x08,0x07,0x20,'_','+','{','}',
    '|','~',':','"','~','<','>','?',0xf0,
    0xbb,0xbc,0xbd,0xbe,0xbf,0xc0,0xc1,0xc2,0xc3,0xc4,0xc5,0xc6,
    0x00,0xf1,0x00,0xd2,0xc7,0xc9,0xd3,0xcf,0xd1,0xcd,0xcd,0xd0,0xc8,0xf2,
    '/','*','-','+',
    0x0d,'1','2','3','4','5','6','7','8','9','0','.','\\',0x00,0x00,'=',
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
};

/* When numlock is on and the raw key value is within the keypad
    value range, this array will be used to decode the raw key value. */

static UCHAR keyboard_layout_raw_to_numlock_on_map[] =
{
    '/','*','-','+',
    0x0d,'1','2','3','4','5','6','7','8','9','0','.','\\',0x00,0x00,'=',
};

/* When numlock is off and the raw key value is within the keypad value
    range, this array will be used to decode the raw key value. */
static UCHAR keyboard_layout_raw_to_numlock_off_map[] =
{
    '/','*','-','+',
    0x0d,0xcf,0xd0,0xd1,0xcb,'5',0xcd,0xc7,0xc8,0xc9,0xd2,0xd3,'\\',0x00,0x
    00,'=',
```

```
};

/* Specify the keyboard layout for USBX usage. */
static UX_HOST_CLASS_HID_KEYBOARD_LAYOUT keyboard_layout =
{
    keyboard_layout_raw_to_shifted_map,
    keyboard_layout_raw_to_unshifted_map,
    keyboard_layout_raw_to_numlock_on_map,
    keyboard_layout_raw_to_numlock_off_map,
    /* The maximum raw key value. Values larger than this are discarded. */
    UX_HID_KEYBOARD_KEYS_UPPER_RANGE,
    /* The raw key value for the letter 'a'. */
    UX_HID_KEYBOARD_KEY_LETTER_A,
    /* The raw key value for the letter 'z'. */
    UX_HID_KEYBOARD_KEY_LETTER_Z,
    /* The lower range raw key value for keypad keys - inclusive. */
    UX_HID_KEYBOARD_KEYS_KEYPAD_LOWER_RANGE,
    /* The upper range raw key value for keypad keys. */
    UX_HID_KEYBOARD_KEYS_KEYPAD_UPPER_RANGE
};

/* Call the IOCTL function to change the keyboard layout. */
status = ux_host_class_hid_keyboard_ioctl(keyboard,
    UX_HID_KEYBOARD_IOCTL_SET_LAYOUT, (VOID *)&keyboard_layout);

/* If status equals UX_SUCCESS, the operation was successful. */
```

示例:**禁用**键盘键**解**码

```
UINT status;

/* The following example illustrates IOCTL function of Disable key decode from keyboard layout. */
status = ux_host_class_hid_keyboard_ioctl(keyboard,
    UX_HID_KEYBOARD_IOCTL_DISABLE_KEYS_DECODE, UX_NULL);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_remote_control_usage_get

获取远程控制使用情况

原型

```
UINT ux_host_class_hid_remote_control_usage_get(
    UX_HOST_CLASS_HID_REMOTE_CONTROL *remote_control_instance,
    LONG *usage,
    ULONG *value);
```

说明

此函数用于获取远程控制使用情况。

参数

- **remote_control_instance** 指向 HID 远程控制实例的指针。
- **usage** 指向使用情况的指针。
- **value** 指向使用情况的值的指针。

返回值

- **UX_SUCCESS** (0x00) 已完成数据传输。
- **UX_ERROR** (0xff) 没有要报告的内容。

- UX_HOST_CLASS_INSTANCE_UNKNOWN (0x5b) HID 类实例不存在。

所有可能的使用情况的列表太长，无法纳入此用户指南。有关获取完整说明，请查看 ux_host_class_hid.h，它提供一组完整的值。

示例

```
/* Read usages and values as the user changes the vol/bass/treble buttons on the speaker */

while (remote_control != UX_NULL)
{
    status = ux_host_class_hid_remote_control_usage_get(remote_control, &usage, &value);
    if (status == UX_SUCCESS)
    {
        /* We have something coming from the HID remote control,
        we filter the usage here and only allow the
        volume usage which can be VOLUME, VOLUME_INCREMENT or VOLUME_DECREMENT */
        switch(usage)
        {
            case UX_HOST_CLASS_HID_CONSUMER_VOLUME :
            case UX_HOST_CLASS_HID_CONSUMER_VOLUME_INCREMENT :
            case UX_HOST_CLASS_HID_CONSUMER_VOLUME_DECREMENT :

            if (value<0x80)
            {
                if (current_volume + audio_control.ux_host_class_audio_control_res < 0xffff)
                    current_volume = current_volume + audio_control.ux_host_class_audio_control_res;
                } else {
                if (current_volume > audio_control.ux_host_class_audio_control_res)
                    current_volume = current_volumeaudio_control.ux_host_class_audio_control_res;
            }

            audio_control.ux_host_class_audio_control_channel = 1;
            audio_control.ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;
            audio_control.ux_host_class_audio_control_cur = current_volume;
            status = ux_host_class_audio_control_value_set(audio, &audio_control);
            audio_control.ux_host_class_audio_control_channel = 2;
            audio_control.ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;
            audio_control.ux_host_class_audio_control_cur = current_volume;
            status = ux_host_class_audio_control_value_set(audio, &audio_control);
            break;

        }
    }
    tx_thread_sleep(10);

}
```

ux_host_class_cdc_acm_read

从 cdc_acm 接口读取。

原型

```
UINT ux_host_class_cdc_acm_read(
    UX_HOST_CLASS_CDC_ACM *cdc_acm,
    UCHAR *data_pointer,
    ULONG requested_length,
    ULONG *actual_length);
```

说明

此函数从 cdc_acm 接口读取。调用将被阻止，并且仅在出现错误或传输完成时返回。

参数

- **cdc_acm** 指向 cdc_acm 类实例的指针。
- **data_pointer** 指向数据有效负载的缓冲区地址的指针。
- **requested_length** 要接收的长度。
- **actual_length** 实际接收的长度。

返回值

- **UX_SUCCESS** (0x00) 已完成数据传输。
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) cdc_acm 实例无效。
- **UX_TRANSFER_TIMEOUT** (0x5c) 传输超时，读取未完成。

示例

```
UINT status;

/* The following example illustrates this service. */

status = ux_host_class_cdc_acm_read(cdc_acm, data_pointer,
    requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_cdc_acm_write

写入 cdc_acm 接口

原型

```
UINT ux_host_class_cdc_acm_write(
    UX_HOST_CLASS_CDC_ACM *cdc_acm,
    UCHAR *data_pointer,
    ULONG requested_length,
    ULONG *actual_length);
```

说明

此函数写入 cdc_acm 接口。调用将被阻止，并且仅在出现错误或传输完成时返回。

参数

- **cdc_acm** 指向 cdc_acm 类实例的指针。
- **data_pointer** 指向数据有效负载的缓冲区地址的指针。
- **requested_length** 要发送的长度。
- **actual_length** 实际发送的长度。

返回值

- **UX_SUCCESS** (0x00) 已完成数据传输。
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) cdc_acm 实例无效。
- **UX_TRANSFER_TIMEOUT** (0x5c) 传输超时，写入未完成。

示例

```
UINT status;

/* The following example illustrates this service. */

status = ux_host_class_cdc_acm_write(cdc_acm, data_pointer,
    requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_cdc_acm_ioctl

向 cdc_acm 接口执行 IOCTL 函数。

原型

```
UINT ux_host_class_cdc_acm_ioctl(
    UX_HOST_CLASS_CDC_ACM *cdc_acm,
    ULONG ioctl_function,
    VOID *parameter);
```

说明

此函数对 cdc_acm 接口执行特定的 ioctl 函数。调用将被阻止，并且仅在出现错误或命令完成时返回。

参数

- **cdc_acm** 指向 cdc_acm 类实例的指针。
- **ioctl_function** 要执行的 ioctl 函数。查看以下表获取允许的 ioctl 函数之一。
- **parameter** 指向特定于 ioctl 的参数的指针

返回值

- **UX_SUCCESS** (0x00) 已完成数据传输。
- **UX_MEMORY_INSUFFICIENT** (0x12) 内存不足。
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) CDC-ACM 实例处于无效状态。
- **UX_FUNCTION_NOT_SUPPORTED** (0x54) 未知 IOCTL 函数。

IOCTL 函数：

- UX_HOST_CLASS_CDC_ACM_IOCTL_SET_LINE_CODING
- UX_HOST_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING
- UX_HOST_CLASS_CDC_ACM_IOCTL_SET_LINE_STATE
- UX_HOST_CLASS_CDC_ACM_IOCTL_SEND_BREAK
- UX_HOST_CLASS_CDC_ACM_IOCTL_ABORT_IN_PIPE
- UX_HOST_CLASS_CDC_ACM_IOCTL_ABORT_OUT_PIPE
- UX_HOST_CLASS_CDC_ACM_IOCTL_NOTIFICATION_CALLBACK
- UX_HOST_CLASS_CDC_ACM_IOCTL_GET_DEVICE_STATUS

```
UINT status;

/* The following example illustrates this service. */

status = ux_host_class_cdc_acm_ioctl(cdc_acm,
    UX_HOST_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING, (VOID *)&line_coding);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_cdc_acm_reception_start

开始从设备对数据进行后台接收。

原型

```
UINT ux_host_class_cdc_acm_reception_start(
    UX_HOST_CLASS_CDC_ACM *cdc_acm,
    UX_HOST_CLASS_CDC_ACM_RECEPTION *cdc_acm_reception);
```

说明

此函数会导致 USBX 在后台持续读取设备数据。完成每个事务后，将调用 cdc_acm_reception 中指定的回调，以便应用程序可以执行对事务数据的进一步处理。

> **NOTE**
> 如果正在使用后台接收，则不得使用 ux_host_class_cdc_acm_read。

参数

- **cdc_acm** 指向 cdc_acm 类实例的指针。
- **cdc_acm_reception** 指向包含定义背景接收行为的值的参数的指针。此参数的布局如下所示：

```
typedef struct UX_HOST_CLASS_CDC_ACM_RECEPTION_STRUCT
{
    ULONG ux_host_class_cdc_acm_reception_state;
    ULONG ux_host_class_cdc_acm_reception_block_size;
    UCHAR *ux_host_class_cdc_acm_reception_data_buffer;
    ULONG ux_host_class_cdc_acm_reception_data_buffer_size;
    UCHAR *ux_host_class_cdc_acm_reception_data_head;
    UCHAR *ux_host_class_cdc_acm_reception_data_tail;
    VOID (*ux_host_class_cdc_acm_reception_callback)(struct UX_HOST_CLASS_CDC_ACM_STRUCT *cdc_acm,
        UINT status, UCHAR *reception_buffer, ULONG reception_size);
} UX_HOST_CLASS_CDC_ACM_RECEPTION;
```

返回值

- **UX_SUCCESS** (0x00) 后台接收已成功启动。
- **UX_HOST_CLASS_INSTANCE _UNKNOWN** (0x5b) 错误类实例。

```
UINT status;
UX_HOST_CLASS_CDC_ACM_RECEPTION cdc_acm_reception;

/* Setup the background reception parameter. */

/* Set the desired max read size for each transaction.
    For example, if this value is 64, then the maximum amount of
    data received from the device in a single transaction is 64.
    If the amount of data received from the device is less than this value,
    the callback will still be invoked with the actual amount of data received. */
cdc_acm_reception.ux_host_class_cdc_acm_reception_block_size = block_size;

/* Set the buffer where the data from the device is read to. */
cdc_acm_reception.ux_host_class_cdc_acm_reception_data_buffer = cdc_acm_reception_buffer;

/* Set the size of the data reception buffer.
    Note that this should be at least as large as ux_host_class_cdc_acm_reception_block_size. */
cdc_acm_reception.ux_host_class_cdc_acm_reception_data_buffer_size = cdc_acm_reception_buffer_size;

/* Set the callback that is to be invoked upon each reception transfer completion. */
cdc_acm_reception.ux_host_class_cdc_acm_reception_callback = reception_callback;

/* Start background reception using the values we defined in the reception parameter. */
status = ux_host_class_cdc_acm_reception_start(cdc_acm_host_data, &cdc_acm_reception);

/* If status equals UX_SUCCESS, background reception has successfully started. */
```

ux_host_class_cdc_acm_reception_stop

停止数据包的后台接收。

原型

```
UINT ux_host_class_cdc_acm_reception_stop(
    UX_HOST_CLASS_CDC_ACM *cdc_acm,
    UX_HOST_CLASS_CDC_ACM_RECEPTION *cdc_acm_reception);
```

说明

此函数会导致 USBX 停止之前由 ux_host_class_cdc_acm_reception_start 启动的后台接收。

参数

- **cdc_acm** 指向 cdc_acm 类实例的指针。
- **cdc_acm_reception** 指向用于启动后台接收的同一参数的指针。此参数的布局如下所示：

```
typedef struct UX_HOST_CLASS_CDC_ACM_RECEPTION_STRUCT
{
    ULONG ux_host_class_cdc_acm_reception_state;
    ULONG ux_host_class_cdc_acm_reception_block_size;
    UCHAR *ux_host_class_cdc_acm_reception_data_buffer;
    ULONG ux_host_class_cdc_acm_reception_data_buffer_size;
    UCHAR *ux_host_class_cdc_acm_reception_data_head;
    UCHAR *ux_host_class_cdc_acm_reception_data_tail;
    VOID (*ux_host_class_cdc_acm_reception_callback)(
            struct UX_HOST_CLASS_CDC_ACM_STRUCT *cdc_acm, UINT status,
            UCHAR *reception_buffer, ULONG reception_size);
} UX_HOST_CLASS_CDC_ACM_RECEPTION;
```

返回值

- **UX_SUCCESS** (0x00) 后台接收已成功停止。

- UX_HOST_CLASS_INSTANCE _UNKNOWN (0x5b) 错误类实例。

```
UINT status;
UX_HOST_CLASS_CDC_ACM_RECEPTION cdc_acm_reception;

/* Stop background reception. The reception parameter should be the same
    that was passed to ux_host_class_cdc_acm_reception_start. */
status = ux_host_class_cdc_acm_reception_stop(cdc_acm, &cdc_acm_reception);

/* If status equals UX_SUCCESS, background reception has successfully stopped. */
```

第 6 章 - USBX CDC-ECM 类用法

2021/4/29 •

USBX 包含一个用于主机和设备端的 CDC-ECM 类。此类旨在用于 NetX，具体而言，USBX CDC-ECM 类充当 NetX 的驱动程序。这就是第 5 章未列出 CDC-ECM API 的原因。

NetX 和 USBX 完成初始化且 USBX 找到 CDC-ECM 设备的实例后，应用程序会以独占方式使用 NetX 与设备进行通信。初始化遵循以下示例所示的模式。

```
UINT status;

/* The USB controller should be the last component initialized so that
everything is ready when data starts being received. */

/* Initialize USBX. */

ux_system_initialize(memory_pointer, UX_USBX_MEMORY_SIZE, UX_NULL, 0);

/* The code below is required for installing the host portion of USBX */
status = ux_host_stack_initialize(UX_NULL);

/* Register cdc_ecm class. */

status = ux_host_stack_class_register(_ux_system_host_class_cdc_ecm_name,
    ux_host_class_cdc_ecm_entry);

/* Perform the initialization of the network driver. */

_ux_network_driver_init();

/* Initialize NetX. Refer to NetX user guide for details to add initialization code. */

/* Register the platform-specific USB controller. */

status = ux_host_stack_hcd_register("controller_name", controller_entry, param1, param2);

/* Find the CDC-ECM class. */
class_cdc_ecm_get();

/* Now wait for the link to be up. */

while (cdc_ecm -> ux_host_class_cdc_ecm_link_state != UX_HOST_CLASS_CDC_ECM_LINK_STATE_UP)
    tx_thread_sleep(10);

/* At this point, everything has been initialized, and we've found a CDC-ECM device.
    Now NetX can be used to communicate with the device. */
```

第 1 章 - USBX 主机堆栈用户指南补充简介

2021/4/29 •

本文档是对 USBX 主机堆栈用户指南的补充。它包含主要用户指南中未包括的未认证 USBX 主机类的文档。

组织

- 第 1 章包含 USBX 简介
- 第 2 章：USBX 主机类 API
- 第 3 章：USBX DPUMP 类注意事项
- 第 4 章：USBX Pictbridge 实现
- 第 5 章：USBX OTG

第 2 章：USBX 主机类 API

2021/4/29 •

本章介绍 USBX 主机类的所有公开的 API。详细介绍了以下每个类的 API。

- 打印机类
- 音频类
- Asix 类
- Pima/PTP 类
- Prolific 类
- 通用序列类

ux_host_class_printer_read

从打印机接口读取。

原型

```
UINT ux_host_class_printer_read(
    UX_HOST_CLASS_PRINTER *printer,
    UCHAR *data_pointer,
    ULONG requested_length,
    ULONG *actual_length)
```

说明

此函数从打印机接口读取。调用阻塞，仅在出现错误或传输完成时返回。只允许在双向打印机上进行读取。

参数

- **printer**：指向打印机类实例的指针。
- **data_pointer**：指向数据有效负载的缓冲区地址的指针。
- **requested_length**：要接收的长度。
- **actual_length**：实际接收的长度。

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_FUNCTION_NOT_SUPPORTED**：(0x54) 函数不受支持，因为打印机不是双向的。
- **UX_TRANSFER_TIMEOUT**：(0x5c) 传输超时，读取未完成。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_printer_read(printer, data_pointer, requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_printer_write

写入打印机接口。

原型

```
UINT ux_host_class_printer_write(
    UX_HOST_CLASS_PRINTER *printer,
    UCHAR *data_pointer,
    ULONG requested_length,
    ULONG *actual_length)
```

说明

此函数写入打印机接口。调用阻塞，仅在出现错误或传输完成时返回。

参数

- **printer**：指向打印机类实例的指针。
- **data_pointer**：指向数据有效负载的缓冲区地址的指针。
- **requested_length**：要发送的长度。
- **actual_length**：实际发送的长度。

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_TRANSFER_TIMEOUT**：(0x5c) 传输超时，写入未完成。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_printer_write(printer, data_pointer, requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_printer_soft_reset

对打印机执行软重置。

原型

```
UINT ux_host_class_printer_soft_reset(UX_HOST_CLASS_PRINTER *printer)
```

说明

此函数对打印机执行软重置。

输入参数

- **printer**：指向打印机类实例的指针。

返回值

- **UX_SUCCESS**：(0x00) 已完成重置。
- **UX_TRANSFER_TIMEOUT**：(0x5c) 传输超时，重置未完成。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_printer_soft_reset(printer);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_printer_status_get

获取打印机状态

原型

```
UINT ux_host_class_printer_status_get(
    UX_HOST_CLASS_PRINTER *printer,
    ULONG *printer_status)
```

说明

此函数获取打印机状态。打印机状态类似于 LPT 状态（1284 标准）。

参数

- **printer**：指向打印机类实例的指针。
- **printer_status**：要返回的状态的地址。

返回值

- **UX_SUCCESS** (0x00)：已完成重置。
- **UX_MEMORY_INSUFFICIENT**：(0x12) 内存不足，无法执行该操作。
- **UX_TRANSFER_TIMEOUT**：(0x5c) 传输超时，重置未完成

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_printer_status_get(printer, printer_status);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_printer_device_id_get

获取打印机设备 ID。

原型

```
UINT ux_host_class_printer_device_id_get(
    UX_HOST_CLASS_PRINTER *printer,
    UCHAR *descriptor_buffer,
    ULONG length)
```

说明

此函数获取打印机 IEEE 1284 设备 ID 字符串（包括 big endian 格式前两个字节的长度）。

参数

- **printer**：指向打印机类实例的指针。

- **descriptor_buffer**：指向用于填充 IEEE 1284 设备 ID 字符串（包括 BE 格式前两个字节的长度）的缓冲区的指针
- **length**：缓冲区的长度（以字节为单位）。

返回值

- **UX_SUCCESS** (0x00)：操作成功。
- **UX_MEMORY_INSUFFICIENT**：(0x12) 内存不足，无法执行该操作。
- **UX_TRANSFER_TIMEOUT**：(0x5c) 传输超时，请求未完成
- **UX_TRANSFER_NOT_READY**：(0x25) 设备处于无效状态 - 状态必须为 ATTACHED、ADDRESSED 或 CONFIGURED。
- **UX_TRANSFER_STALL**：(0x21) 传输停滞。
- **TX_WAIT_ABORTED**：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- **TX_SEMAPHORE_ERROR**：(0x0C) 无效计数信号灯指针。
- **TX_WAIT_ERROR**：(0x04) 在非线程的调用上指定了除 TX_NO_WAIT 以外的等待选项。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_printer_device_id_get(printer, descriptor_buffer, length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_read

从音频接口读取。

原型

```
UINT ux_host_class_audio_read(
    UX_HOST_CLASS_AUDIO *audio,
    UX_HOST_CLASS_AUDIO_TRANSFER_REQUEST
    *audio_transfer_request)
```

说明

此函数从音频接口读取。调用为非阻塞。应用程序必须确保为音频流式处理接口选择了适当的备用设置。

参数

- **audio**：指向音频类实例的指针。
- **audio_transfer_request**：指向音频传输结构的指针。

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输
- **UX_FUNCTION_NOT_SUPPORTED**：(0x54) 函数不受支持

示例

```
/* The following example reads from the audio interface. */

audio_transfer_request.ux_host_class_audio_transfer_request_completion_function =
tx_audio_transfer_completion_function;
audio_transfer_request.ux_host_class_audio_transfer_request_class_instance = audio;
audio_transfer_request.ux_host_class_audio_transfer_request_next_audio_audio_transfer_request = UX_NULL;
audio_transfer_request.ux_host_class_audio_transfer_request_data_pointer = audio_buffer;
audio_transfer_request.ux_host_class_audio_transfer_request_requested_length = requested_length;
audio_transfer_request.ux_host_class_audio_transfer_request_packet_length = AUDIO_FRAME_LENGTH;

status = ux_host_class_audio_read(audio, audio_transfer_request);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_write

写入音频接口。

原型

```
UINT ux_host_class_audio_write(
    UX_HOST_CLASS_AUDIO *audio,
    UX_HOST_CLASS_AUDIO_TRANSFER_REQUEST *audio_transfer_request)
```

说明

此函数写入音频接口。调用为非阻塞。应用程序必须确保为音频流式处理接口选择了适当的备用设置。

参数

- **audio**：指向音频类实例的指针
- **audio_transfer_request**：指向音频传输结构的指针

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_FUNCTION_NOT_SUPPORTED**：(0x54) 函数不受支持。
- **ux_host_CLASS_AUDIO_WRONG_INTERFACE**：(0x81) 接口不正确。

示例

```
UINT status;

/* The following example writes to the audio interface */

audio_transfer_request.ux_host_class_audio_transfer_request_completion_function =
tx_audio_transfer_completion_function;
audio_transfer_request.ux_host_class_audio_transfer_request_class_instance = audio;
audio_transfer_request.ux_host_class_audio_transfer_request_next_audio_audio_transfer_request = UX_NULL;
audio_transfer_request.ux_host_class_audio_transfer_request_data_pointer = audio_buffer;
audio_transfer_request.ux_host_class_audio_transfer_request_requested_length = requested_length;
audio_transfer_request.ux_host_class_audio_transfer_request_packet_length = AUDIO_FRAME_LENGTH;
status = ux_host_class_audio_write(audio, audio_transfer_request);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_control_get

从音频控件接口获取特定控件。

原型

```
UINT ux_host_class_audio_control_get(
    UX_HOST_CLASS_AUDIO *audio,
    UX_HOST_CLASS_AUDIO_CONTROL *audio_control)
```

说明

此函数从音频控件接口读取特定控件。

参数

- **audio**：指向音频类实例的指针
- **audio_control**：指向音频控件结构的指针

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输
- **UX_FUNCTION_NOT_SUPPORTED**：(0x54) 函数不受支持
- **UX_HOST_CLASS_AUDIO_WRONG_INTERFACE**：(0x81) 接口不正确

示例

```
UINT status;

/* The following example reads the volume control from a stereo USB speaker. */

UX_HOST_CLASS_AUDIO_CONTROL audio_control;

audio_control.ux_host_class_audio_control_channel = 1;
audio_control.ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;

status = ux_host_class_audio_control_get(audio, &audio_control);

/* If status equals UX_SUCCESS, the operation was successful. */

audio_control.ux_host_class_audio_control_channel = 2;
audio_control.ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;

status = ux_host_class_audio_control_get(audio, &audio_control);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_control_value_set

为音频控件接口设置特定的控件。

原型

```
UINT ux_host_class_audio_control_value_set(
    UX_HOST_CLASS_AUDIO *audio,
    UX_HOST_CLASS_AUDIO_CONTROL *audio_control)
```

**说明 **

此函数为音频控件接口设置特定的控件。

参数

- **audio**：指向音频类实例的指针
- **audio_control**：指向音频控件结构的指针

返回值

- UX_SUCCESS : (0x00) 已完成数据传输
- UX_FUNCTION_NOT_SUPPORTED : (0x54) 函数不受支持
- UX_HOST_CLASS_AUDIO_WRONG_INTERFACE : (0x81) 接口不正确

示例

```
/* The following example sets the volume control of a stereo USB speaker. */

UX_HOST_CLASS_AUDIO_CONTROL audio_control;

UINT status;

audio_control.ux_host_class_audio_control_channel = 1;
audio_control.ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;
audio_control.ux_host_class_audio_control_cur = 0xf000;

status = ux_host_class_audio_control_value_set(audio, &audio_control);
/* If status equals UX_SUCCESS, the operation was successful. */

current_volume = audio_control.audio_control_cur;
audio_control.ux_host_class_audio_control_channel = 2;
audio_control.ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;
audio_control.ux_host_class_audio_control_cur = 0xf000;

status = ux_host_class_audio_control_value_set(audio, &audio_control);
/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_streaming_sampling_set

设置音频流式处理接口的备用设置接口。

原型

```
UINT ux_host_class_audio_streaming_sampling_set
    (UX_HOST_CLASS_AUDIO *audio,
    UX_HOST_CLASS_AUDIO_SAMPLING *audio_sampling)
```

说明

此函数根据特定采样结构设置音频流式处理接口的适当备用设置接口。

参数

- audio : 指向音频类实例的指针。
- audio_sampling : 指向音频采样结构的指针。

返回值

- UX_SUCCESS : (0x00) 已完成数据传输
- UX_FUNCTION_NOT_SUPPORTED : (0x54) 函数不受支持
- UX_HOST_CLASS_AUDIO_WRONG_INTERFACE : (0x81) 接口不正确
- UX_NO_ALTERNATE_SETTING : (0x5e) 采样值没有备用设置

示例

```
/* The following example sets the alternate setting interface of a stereo USB speaker. */

UX_HOST_CLASS_AUDIO_SAMPLING audio_sampling;

UINT status;

sampling.ux_host_class_audio_sampling_channels = 2;
sampling.ux_host_class_audio_sampling_frequency = AUDIO_FREQUENCY;
sampling. ux_host_class_audio_sampling_resolution = 16;

status = ux_host_class_audio_streaming_sampling_set(audio, &sampling);
/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_streaming_sampling_get

获取音频流式处理接口的可能采样设置。

原型

```
UINT ux_host_class_audio_streaming_sampling_get(
    UX_HOST_CLASS_AUDIO *audio,
    UX_HOST_CLASS_AUDIO_SAMPLING_CHARACTERISTICS *audio_sampling)
```

说明

此函数逐个获取可在音频流式处理接口的每个备用设置中使用的所有可能的采样设置。第一次使用函数时，必须重置调用结构指针中的所有字段。此函数将在返回时返回一组特定的流式处理值，除非已达到替代设置的结尾。重用此函数时，将使用之前的采样值来查找下一采样值。

参数

- **audio**：指向音频类实例的指针。
- **audio_sampling**：指向音频采样结构的指针。

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输
- **UX_FUNCTION_NOT_SUPPORTED**：(0x54) 函数不受支持
- **UX_HOST_CLASS_AUDIO_WRONG_INTERFACE**：(0x81) 接口不正确
- **UX_NO_ALTERNATE_SETTING**：(0x5e) 采样值没有备用设置

示例

```
/* The following example gets the sampling values for the first alternate setting interface of a stereo USB
speaker. */

UX_HOST_CLASS_AUDIO_SAMPLING_CHARACTERISTICS audio_sampling;

UINT status;

sampling.ux_host_class_audio_sampling_channels=0;
sampling.ux_host_class_audio_sampling_frequency_low=0;
sampling.ux_host_class_audio_sampling_frequency_high=0;
sampling.ux_host_class_audio_sampling_resolution=0;

status = ux_host_class_audio_streaming_sampling_get(audio, &sampling);

/* If status equals UX_SUCCESS, the operation was successful and information could be displayed as follows:

printf("Number of channels %d, Resolution %d bits, frequency range %d-%d\n",
    sampling.audio_channels, sampling.audio_resolution,
    sampling.audio_frequency_low, sampling.audio_frequency_high);

*/
```

ux_host_class_asix_read

从 asix 接口读取。

原型

```
UINT ux_host_class_asix_read(
    UX_HOST_CLASS_ASIX *asix,
    UCHAR *data_pointer,
    ULONG requested_length,
    ULONG *actual_length)
```

说明

此函数从 asix 接口读取。调用阻塞，仅在出现错误或传输完成时返回。

参数

- **asix**：指向 asix 类实例的指针。
- **data_pointer**：指向数据有效负载的缓冲区地址的指针。
- **requested_length**：要接收的长度。
- **actual_length**：实际接收的长度。

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_TRANSFER_TIMEOUT**：(0x5c) 传输超时，读取未完成。

示例

```
UINT status;

/* The following example illustrates this service. */

status = ux_host_class_asix_read(asix, data_pointer, requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_asix_write

写入 asix 接口。

原型

```
UINT ux_host_class_asix_write(
    VOID *asix_class,
    NX_PACKET *packet)
```

说明

此函数写入 asix 接口。调用为非阻塞。

参数

- **asix**：指向 asix 类实例的指针。
- **packet**：Netx 数据包

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_ERROR**：(0xFF) 无法请求传输。

示例

```
UINT status;

/* The following example illustrates this service. */

status = ux_host_class_asix_write(asix, packet);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_pima_session_open

在发起方和响应方之间打开会话。

原型

```
UINT ux_host_class_pima_session_open(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session)
```

说明

此函数在 PIMA 发起方和 PIMA 响应方之间打开一个会话。成功打开会话后，可以执行大多数 PIMA 命令。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针。

返回值

- **UX_SUCCESS**：(0x00) 会话已成功打开
- **UX_HOST_CLASS_PIMA_RC_SESSION_ALREADY_OPENED**：(0x201E) 会话已打开

示例

```
/* Open a pima session. */

status = ux_host_class_pima_session_open(pima, pima_session);

if (status != UX_SUCCESS)
    return(UX_PICTBRIDGE_ERROR_SESSION_NOT_OPEN);
```

ux_host_class_pima_session_close

在发起方和响应方之间关闭会话。

原型

```
UINT ux_host_class_pima_session_close(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session)
```

说明

此函数关闭之前在 PIMA 发起方和 PIMA 响应方之间打开的会话。会话关闭后，将无法再执行大多数 PIMA 命令。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针。

返回值

- **UX_SUCCESS**：(0x00) 会话已关闭。
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**：(0x2003) 会话未打开。

示例

```
/* Close the pima session. */

status = ux_host_class_pima_session_close(pima, pima_session);
```

ux_host_class_pima_storage_ids_get

从响应方获取存储 ID 数组。

原型

```
UINT ux_host_class_pima_storage_ids_get(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG *storage_ids_array,
    ULONG storage_id_length)
```

说明

此函数从响应方获取存储 ID 数组。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针
- **storage_ids_array**：将在其中返回存储 ID 的数组

- **storage_id_length**：存储阵列的长度

返回值

- **UX_SUCCESS**：(0x00) 已填充存储 ID 数组
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**：(0x2003) 会话未打开
- **UX_MEMORY_INSUFFICIENT**：(0x12) 内存不足，无法创建 PIMA 命令。

示例

```
/* Get the number of storage IDs. */
status = ux_host_class_pima_storage_ids_get(pima, pima_session,
    pictbridge ->ux_pictbridge_storage_ids, 64);

if (status != UX_SUCCESS)
{
    /* Close the pima session. */
    status = ux_host_class_pima_session_close(pima, pima_session);

    return(UX_PICTBRIDGE_ERROR_STORE_NOT_AVAILABLE);
}
```

ux_host_class_pima_storage_info_get

从响应方获取存储信息。

原型

```
UINT ux_host_class_pima_storage_info_get(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG storage_id,
    UX_HOST_CLASS_PIMA_STORAGE *storage)
```

说明

此函数获取值为 storage_id 的存储容器的存储信息。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针
- **storage_id**：存储容器的 ID
- **存储**：指向存储信息容器的指针

返回值

- **UX_SUCCESS**：(0x00) 已检索存储信息
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**：(0x2003) 会话未打开
- **UX_MEMORY_INSUFFICIENT** (0x12) 内存不足，无法创建 PIMA 命令。

示例

```
/* Get the first storage ID info container. */
status = ux_host_class_pima_storage_info_get(pima, pima_session,
    pictbridge ->ux_pictbridge_storage_ids[0],
    (UX_HOST_CLASS_PIMA_STORAGE *)pictbridge ->ux_pictbridge_storage);

if (status != UX_SUCCESS)
{
    /* Close the pima session. */
    status = ux_host_class_pima_session_close(pictbridge ->
        ux_pictbridge_pima, pima_session);
    return(UX_PICTBRIDGE_ERROR_STORE_NOT_AVAILABLE);
}
```

ux_host_class_pima_num_objects_get

从响应方获取存储容器上的对象数。

原型

```
UINT ux_host_class_pima_num_objects_get(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG storage_id,
    ULONG object_format_code)
```

说明

此函数获取与特定格式代码匹配且值为 storage_id 的特定存储容器中存储的对象数。对象数在 ux_host_class_pima_session_nb_objects of the pima_session structure 字段中返回。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针
- **storage_id**：存储容器的 ID
- **object_format_code**：对象格式代码筛选器。

对象格式代码可以有以下值之一。

□□□□□	□□	USBX □□
0x3000	未定义的非图像对象	UX_HOST_CLASS_PIMA_OFC_UNDEFINED
0x3001	关联（如文件夹）	UX_HOST_CLASS_PIMA_OFC_ASSOCIATION
0x3002	特定于脚本设备模型的脚本	UX_HOST_CLASS_PIMA_OFC_SCRIPT
0x3003	特定于可执行设备模型的二进制可执行文件	UX_HOST_CLASS_PIMA_OFC_EXECUTABLE
0x3004	文本文件	UX_HOST_CLASS_PIMA_OFC_TEXT
0x3005	HTML 超文本标记语言文件（文本）	UX_HOST_CLASS_PIMA_OFC_HTML
0x3006	DPOF 数码打印命令格式文件（文本）	UX_HOST_CLASS_PIMA_OFC_DPOF

[illegible]	[illegible]	USBX [illegible]
0x3007	AIFF 音频剪辑	UX_HOST_CLASS_PIMA_OFC_AIFF
0x3008	WAV 音频剪辑	UX_HOST_CLASS_PIMA_OFC_WAV
0x3009	MP3 音频剪辑	UX_HOST_CLASS_PIMA_OFC_MP3
0x300A	AVI 视频剪辑	UX_HOST_CLASS_PIMA_OFC_AVI
0x300B	MPEG 视频剪辑	UX_HOST_CLASS_PIMA_OFC_MPEG
0x300C	ASF Microsoft 高级流式处理格式(视频)	UX_HOST_CLASS_PIMA_OFC_ASF
0x3800	未定义的未知图像对象	UX_HOST_CLASS_PIMA_OFC_QT
0x3801	EXIF/JPEG 可交换文件格式, JEIDA 标准	UX_HOST_CLASS_PIMA_OFC_EXIF_JPEG
0x3802	TIFF/EP 适用于电子摄影的标记图像文件格式	UX_HOST_CLASS_PIMA_OFC_TIFF_EP
0x3803	FlashPix 结构化存储图像格式	UX_HOST_CLASS_PIMA_OFC_FLASHPIX
0x3804	BMP Microsoft Windows 位图文件	UX_HOST_CLASS_PIMA_OFC_BMP
0x3805	CIFF Canon 相机图像文件格式	UX_HOST_CLASS_PIMA_OFC_CIFF
0x3806	未定义的保留	
0x3807	GIF 图形交换格式	UX_HOST_CLASS_PIMA_OFC_GIF
0x3808	JFIF JPEG 文件交换格式	UX_HOST_CLASS_PIMA_OFC_JFIF
0x3809	PCD PhotoCD 图像 Pac	UX_HOST_CLASS_PIMA_OFC_PCD
0x380A	PICT Quickdraw 图像格式	UX_HOST_CLASS_PIMA_OFC_PICT
0x380B	PNG 可移植网络图形格式	UX_HOST_CLASS_PIMA_OFC_PNG
0x380C	未定义的保留	
0x380D	TIFF 标记图像文件格式	UX_HOST_CLASS_PIMA_OFC_TIFF
0x380E	TIFF/IT 适用于信息技术(图形艺术)的标记图像文件格式	UX_HOST_CLASS_PIMA_OFC_TIFF_IT
0x380F	JP2 JPEG2000 基线文件格式	UX_HOST_CLASS_PIMA_OFC_JP2
0x3810	JPX JPEG2000 扩展文件格式	UX_HOST_CLASS_PIMA_OFC_JPX

¢¢¢¢¢¢	¢¢	USBX ¢¢
MSN 0011 的所有其他代码	任何未定义的保留以供将来使用	
MSN 1011 的所有其他代码	供应商定义的任何类型：图像	

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**：(0x2003) 会话未打开
- **UX_MEMORY_INSUFFICIENT**：(0x12) 内存不足，无法创建 PIMA 命令。

示例

```
/* Get the number of objects on all containers matching a SCRIPT object. */
status = ux_host_class_pima_num_objects_get(pima, pima_session,
    UX_PICTBRIDGE_ALL_CONTAINERS, UX_PICTBRIDGE_OBJECT_SCRIPT);

if (status != UX_SUCCESS)
{
    /* Close the pima session. */
    status = ux_**host_class_pima_session_close(pima, pima_session);

    return(UX_PICTBRIDGE_ERROR_STORE_NOT_AVAILABLE);
} else
    /* The number of objects is returned in the field: pima_session -> ux_host_class_pima_session_nb_objects
*/
```

ux_host_class_pima_object_handles_get

从响应方获取对象句柄。

原型

```
UINT ux_host_class_pima_object_handles_get(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG *object_handles_array,
    ULONG object_handles_length,
    ULONG storage_id,
    ULONG object_format_code,
    ULONG object_handle_association)
```

说明

返回由 storage_id 参数指示的存储容器中存在的对象句柄的数组。如果需要跨所有存储的聚合列表，则此值应设置为 0xFFFFFFFF。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针
- **object_handes_array**：在其中返回句柄的数组
- **object_handles_length**：数组的长度
- **storage_id**：存储容器的 ID
- **object_format_code**：对象的格式代码（请参阅函数 ux_host_class_pima_num_objects_get 的表）
- **object_handle_association**：可选对象关联值

对象句柄关联可以是下表中的值之一：

[illegible]	[illegible]	[illegible]
0x0000	Undefined	Undefined
0x0001	GenericFolder	未使用
0x0002	相册:	保留
0x0003	TimeSequence	DefaultPlaybackDelta
0x0004	HorizontalPanoramic	未使用
0x0005	VerticalPanoramic	未使用
0x0006	2DPanoramic	ImagesPerRow
0x0007	AncillaryData	Undefined
位 15 设置为 0 的所有其他值	保留	Undefined
位 15 设置为 1 的所有值	供应商定义	供应商定义

返回值

- **UX_SUCCESS**:(0x00) 已完成数据传输。
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**:(0x2003) 会话未打开
- **UX_MEMORY_INSUFFICIENT**:(0x12) 内存不足，无法创建 PIMA 命令。

示例

```
/* Get the array of objects handles on the container. */
status = ux_**host_class_pima_object_handles_get(pima, pima_session,
    pictbridge ->ux_pictbridge_object_handles_array,
    4 * pima_session ->ux_host_class_pima_session_nb_objects,
    UX_PICTBRIDGE_ALL_CONTAINERS,
    UX_PICTBRIDGE_OBJECT_SCRIPT, 0);

if (status != UX_SUCCESS)
{
    /* Close the pima session. */
    status = ux_host_class_pima_session_close(pima, pima_session);
    return(UX_PICTBRIDGE_ERROR_STORE_NOT_AVAILABLE);
}
```

ux_host_class_pima_object_info_get

从响应方获取对象信息。

原型

```
UINT ux_host_class_pima_object_info_get(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG object_handle,
    UX_HOST_CLASS_PIMA_OBJECT *object)
```

说明

此函数获取对象句柄的对象信息。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针
- **object_handle**：对象的句柄
- **object**：指向对象信息容器的指针

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**：(0x2003) 会话未打开
- **UX_MEMORY_INSUFFICIENT**：(0x12) 内存不足，无法创建 PIMA 命令。

示例

```
/* We search for an object that is a picture or a script. */
object_index = 0;

while (object_index < pima_session -> ux_host_class_pima_session_nb_objects)
{
    /* Get the object info structure. */
    status = ux_**host_class_pima_object_info_get(pima, pima_session,
        pictbridge -> ux_pictbridge_object_handles_array[object_index],
        pima_object);

    if (status != UX_SUCCESS)
    {
        /* Close the pima session. */
        status = ux_host_class_pima_session_close(pima, pima_session);

        return(UX_PICTBRIDGE_ERROR_INVALID_OBJECT_HANDLE );
    }
}
```

ux_host_class_pima_object_info_send

将对象信息发送到响应方。

原型

```
UINT ux_host_class_pima_object_info_send(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG storage_id,
    ULONG parent_object_id,
    UX_HOST_CLASS_PIMA_OBJECT *object)
```

说明

此函数发送值为 storage_id 的存储容器的存储信息。将对象发送到响应方前，发起方应使用此命令。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针。
- **storage_id**：目标存储 ID。
- **parent_object_id**：应放置对象的响应方上的父 ObjectHandle。
- **object**：指向对象信息容器的指针。

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**：(0x2003) 会话未打开
- **UX_MEMORY_INSUFFICIENT**：(0x12) 内存不足，无法创建 PIMA 命令。

示例

```
/* Send a script info. */
status = ux_host_class_pima_object_info_send(pima, pima_session,
    0, 0, pima_object);

if (status != UX_SUCCESS)
{
    /* Close the pima session. */
    status = ux_host_class_pima_session_close(pima, pima_session);

    return(UX_ERROR);
}
```

ux_host_class_pima_object_open

打开存储在响应方中的对象。

原型

```
UINT ux_host_class_pima_object_open(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG object_handle,
    UX_HOST_CLASS_PIMA_OBJECT *object)
```

说明

在读取或写入之前，此函数将在响应方上打开一个对象。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针。
- **object_handle**：对象的句柄。
- **object**：指向对象信息容器的指针。

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**：(0x2003) 会话未打开
- **UX_HOST_CLASS_PIMA_RC_OBJECT_ALREADY_OPENED**：(0x2021) 对象已打开。
- **UX_MEMORY_INSUFFICIENT**：(0x12) 内存不足，无法创建 PIMA 命令。

示例

```
/* Open the object. */

status = ux_host_class_pima_object_open(pima, pima_session,
    object_handle, pima_object);

/* Check status. */
if (status != UX_SUCCESS)
    return(status);
```

ux_host_class_pima_object_get

获取存储在响应方中的对象。

原型

```
UINT ux_host_class_pima_object_get(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG object_handle,
    UX_HOST_CLASS_PIMA_OBJECT *object,
    UCHAR *object_buffer,
    ULONG object_buffer_length,
    ULONG *object_actual_length)
```

说明

此函数获取响应方上的对象。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针
- **object_handle**：对象的句柄
- **object**：指向对象信息容器的指针
- **object_buffer**：对象数据的地址
- **object_buffer_length**：请求的对象长度
- **object_actual_length**：已返回的对象长度

返回值

- **UX_SUCCESS**：(0x00) 已传输对象
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**：(0x2003) 会话未打开
- **UX_HOST_CLASS_PIMA_RC_OBJECT_NOT_OPENED**：(0x2023) 对象未打开。
- **UX_HOST_CLASS_PIMA_RC_ACCESS_DENIED**：(0x200f) 拒绝访问对象
- **UX_HOST_CLASS_PIMA_RC_INCOMPLETE_TRANSFER**：(0x2007) 传输未完成
- **UX_MEMORY_INSUFFICIENT**：(0x12) 内存不足，无法创建 PIMA 命令。
- **UX_TRANSFER_ERROR**：(0x23) 读取对象时出现传输错误

示例

```
/* Open the object. */

status = ux_host_class_pima_object_open(pima, pima_session,
    object_handle, pima_object);

/* Check status. */
if (status != UX_SUCCESS)
    return(status);

/* Set the object buffer pointer. */
object_buffer = pima_object ->ux_host_class_pima_object_buffer;

/* Obtain all the object data. */
while(object_length != 0)
{
    /* Calculate what length to request. */
    if (object_length > UX_PICTBRIDGE_MAX_PIMA_OBJECT_BUFFER)
        /* Request maximum length. */
        requested_length = UX_PICTBRIDGE_MAX_PIMA_OBJECT_BUFFER;
    else
        /* Request remaining length. */
        requested_length = object_length;

    /* Get the object data. */
    status = ux_host_class_pima_object_get(pima, pima_session,
        object_handle, pima_object, object_buffer,
        requested_length, &actual_length);

    if (status != UX_SUCCESS)
    {
        /* We had a problem, abort the transfer. */
        ux_host_class_pima_object_transfer_abort(pima, pima_session,
            object_handle, pima_object);

        /* And close the object. */
        ux_host_class_pima_object_close(pima, pima_session,
            object_handle, pima_object, object);

        return(status);

    }

    /* We have received some data, update the length remaining. */
    object_length -= actual_length;

    /* Update the buffer address. */
    object_buffer += actual_length;

}

/* Close the object. */
status = ux_host_class_pima_object_close(pima, pima_session,
    object_handle, pima_object, object);
```

ux_host_class_pima_object_send

发送存储在响应方中的对象。

原型

```
UINT ux_host_class_pima_object_send(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    UX_HOST_CLASS_PIMA_OBJECT *object,
    UCHAR *object_buffer, ULONG object_buffer_length)
```

说明

此函数将对象发送到响应方。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针
- **object_handle**：对象的句柄
- **object**：指向对象信息容器的指针
- **object_buffer**：对象数据的地址
- **object_buffer_length**：请求的对象长度

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**：(0x2003) 会话未打开
- **UX_HOST_CLASS_PIMA_RC_OBJECT_NOT_OPENED**：(0x2023) 对象未打开。
- **UX_HOST_CLASS_PIMA_RC_ACCESS_DENIED**：(0x200f) 拒绝访问对象
- **UX_HOST_CLASS_PIMA_RC_INCOMPLETE_TRANSFER**：(0x2007) 传输未完成
- **UX_MEMORY_INSUFFICIENT**：(0x12) 内存不足，无法创建 PIMA 命令。
- **UX_TRANSFER_ERROR**：(0x23) 写入对象时出现传输错误

示例

```
/* Open the object. */
status = ux_host_class_pima_object_open(pima, pima_session,
    object_handle, pima_object);

/* Get the object length. */
object_length = pima_object ->ux_host_class_pima_object_compressed_size;

/* Recall the object buffer address. */
pima_object_buffer = pima_object ->ux_host_class_pima_object_buffer;

/* Send all the object data. */
while(object_length != 0)
{
    /* Calculate what length to request. */
    if (object_length > UX_PICTBRIDGE_MAX_PIMA_OBJECT_BUFFER)
        /* Request maximum length. */
        requested_length = UX_PICTBRIDGE_MAX_PIMA_OBJECT_BUFFER;
    else
        /* Request remaining length. */
        requested_length = object_length;

    /* Send the object data. */
    status = ux_host_class_pima_object_send(pima,
        pima_session, pima_object,
        pima_object_buffer, requested_length);

    if (status != UX_SUCCESS)
    {
        /* Abort the transfer. */
        ux_host_class_pima_object_transfer_abort(pima, pima_session,
            object_handle, pima_object);

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

    /* We have sent some data, update the length remaining. */
    object_length -= requested_length;
}

/* Close the object. */
status = ux_host_class_pima_object_close(pima, pima_session, object_handle,
    pima_object, object);
```

ux_host_class_pima_thumb_get

获取存储在响应方中的 Thumb 对象。

原型

```
UINT ux_host_class_pima_thumb_get(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG object_handle,
    UX_HOST_CLASS_PIMA_OBJECT *object,
    UCHAR *thumb_buffer, ULONG thumb_buffer_length,
    ULONG *thumb_actual_length)
```

说明

此函数获取响应方上的 Thumb 对象。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针。
- **object_handle**：对象的句柄。
- **object**：指向对象信息容器的指针。
- **thumb_buffer**：Thumb 对象数据的地址。
- **thumb_buffer_length**：请求的 Thumb 对象的长度。
- **thumb_actual_length**：返回的 Thumb 对象的长度。

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**：(0x2003) 会话未打开。
- **UX_HOST_CLASS_PIMA_RC_OBJECT_NOT_OPENED**：(0x2023) 对象未打开。
- **UX_HOST_CLASS_PIMA_RC_ACCESS_DENIED**：(0x200f) 拒绝访问对象。
- **UX_HOST_CLASS_PIMA_RC_INCOMPLETE_TRANSFER**：(0x2007) 传输未完成。
- **UX_MEMORY_INSUFFICIENT**：(0x12) 内存不足，无法创建 PIMA 命令。
- **UX_TRANSFER_ERROR**：(0x23) 读取对象时出现传输错误。

示例

```
/* Get the thumb object data. */

status = ux_host_class_pima_thumb_get(pima, pima_session,
    object_handle, pima_object, object_buffer,
    requested_length, &actual_length);

if (status != UX_SUCCESS)
{
    /* And close the object. */
    ux_host_class_pima_object_close(pima, pima_session, object_handle, pima_object, object);

    return(status);
}
```

ux_host_class_pima_object_delete

删除存储在响应方中的对象。

原型

```
UINT ux_host_class_pima_object_delete(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG object_handle)
```

说明

此函数删除响应方上的对象

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针
- **object_handle**：对象的句柄

返回值

- **UX_SUCCESS**：(0x00) 已删除对象。

- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**：(0x2003) 会话未打开。
- **UX_HOST_CLASS_PIMA_RC_ACCESS_DENIED**：(0x200f) 无法删除对象。
- **UX_MEMORY_INSUFFICIENT**：(0x12) 内存不足，无法创建 PIMA 命令。

示例

```
/* Delete the object. */
status = ux_host_class_pima_object_delete(pima, pima_session, object_handle, pima_object);

/* Check status. */
if (status != UX_SUCCESS)
    return(status);
```

ux_host_class_pima_object_close

关闭存储在响应方中的对象

原型

```
UINT ux_host_class_pima_object_close(
    UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG object_handle, UX_HOST_CLASS_PIMA_OBJECT *object)
```

说明

此函数关闭响应方上的对象。

参数

- **pima**：指向 pima 类实例的指针。
- **pima_session**：指向 PIMA 会话的指针。
- **object_handle**：对象的句柄。
- **object**：指向对象的指针。

返回值

- **UX_SUCCESS**：(0x00) 已关闭对象。
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**：(0x2003) 会话未打开。
- **UX_HOST_CLASS_PIMA_RC_OBJECT_NOT_OPENED**：(0x2023) 对象未打开。
- **UX_MEMORY_INSUFFICIENT**：(0x12) 内存不足，无法创建 PIMA 命令。

示例

```
/* Close the object. */
status = ux_host_class_pima_object_close(pima, pima_session, object_handle, object);
```

ux_host_class_gser_read

从通用序列接口读取。

原型

```
UINT ux_host_class_gser_read(
    UX_HOST_CLASS_GSER *gser,
    ULONG interface_index,
    UCHAR *data_pointer,
    ULONG requested_length,
    ULONG *actual_length)
```

说明

此函数从通用序列接口读取。调用阻塞，仅在出现错误或传输完成时返回。

参数

- **gser**：指向 gser 类实例的指针。
- **interface_index**：要从其中读取的接口索引。
- **data_pointer**：指向数据有效负载的缓冲区地址的指针。
- **requested_length**：要接收的长度。
- **actual_length**：实际接收的长度。

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_TRANSFER_TIMEOUT**：(0x5c) 传输超时，读取未完成。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_gser_read(cdc_acm, interface_index,data_pointer, requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_gser_write

写入通用序列接口。

原型

```
UINT ux_host_class_gser_write(
    UX_HOST_CLASS_GSER *gser,
    ULONG interface_index,
    UCHAR *data_pointer,
    ULONG requested_length,
    ULONG *actual_length)
```

说明

此函数写入通用序列接口。调用阻塞，仅在出现错误或传输完成时返回。

参数

- **gser**：指向 gser 类实例的指针。
- **interface_index**：要写入的接口。
- **data_pointer**：指向数据有效负载的缓冲区地址的指针。
- **requested_length**：要发送的长度。
- **actual_length**：实际发送的长度。

返回值

- **UX_SUCCESS**: (0x00) 已完成数据传输。
- **UX_TRANSFER_TIMEOUT**: (0x5c) 传输超时，写入未完成。

示例

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_cdc_acm_write(gser, data_pointer, requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_gser_ioctl

对通用序列接口执行 IOCTL 函数。

原型

```
UINT ux_host_class_gser_ioctl(
    UX_HOST_CLASS_GSER *gser,
    ULONG ioctl_function,
    VOID *parameter)
```

说明

此函数对 gser 接口执行特定的 ioctl 函数。调用阻塞，仅在出现错误或命令完成时返回。

参数

- **gser**: 指向 gser 类实例的指针。
- **ioctl_function**: 要执行的 ioctl 函数。查看以下表获取一个允许的 ioctl 函数。
- **parameter**: 指向特定于 ioctl 的参数的指针

返回值

- **UX_SUCCESS**: (0x00) 已完成数据传输。
- **UX_MEMORY_INSUFFICIENT**: (0x12) 内存不足。
- **UX_HOST_CLASS_UNKNOWN**: (0x59) 错误的类实例
- **UX_FUNCTION_NOT_SUPPORTED**: (0x54) 未知 IOCTL 函数。

IOCTL 函数

- UX_HOST_CLASS_GSER_IOCTL_SET_LINE_CODING
- UX_HOST_CLASS_GSER_IOCTL_GET_LINE_CODING
- UX_HOST_CLASS_GSER_IOCTL_SET_LINE_STATE
- UX_HOST_CLASS_GSER_IOCTL_SEND_BREAK
- UX_HOST_CLASS_GSER_IOCTL_ABORT_IN_PIPE
- UX_HOST_CLASS_GSER_IOCTL_ABORT_OUT_PIPE
- UX_HOST_CLASS_GSER_IOCTL_NOTIFICATION_CALLBACK
- UX_HOST_CLASS_GSER_IOCTL_GET_DEVICE_STATUS

示例

```
UINT status;

/* The following example illustrates this service. */

status = ux_host_class_gser_ioctl(gser,
    UX_HOST_CLASS_GSER_IOCTL_GET_LINE_CODING,
    (VOID *)&line_coding);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_gser_reception_start

在通用序列接口上启动接收

原型

```
UINT ux_host_class_gser_reception_start(
    UX_HOST_CLASS_GSER *gser,
    UX_HOST_CLASS_GSER_RECEPTION *gser_reception)
```

说明

此函数在通用序列类接口上启动接收。此函数允许非阻塞接收。接收到缓冲区时，将在应用程序中调用回调。

参数

- **gser**：指向 gser 类实例的指针。
- **gser_reception**：包含接收参数的结构

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_HOST_CLASS_UNKNOWN**：(0x59) 错误的类实例
- **UX_ERROR**：(0x01) 错误

示例

```
/* Start the reception for gser. AT commands are on interface 2. */
gser_reception.ux_host_class_gser_reception_interface_index =
    UX_DEMO_GSER_AT_INTERFACE;
gser_reception.ux_host_class_gser_reception_block_size =
    UX_DEMO_RECEPTION_BLOCK_SIZE;
gser_reception.ux_host_class_gser_reception_data_buffer =
    gser_reception_buffer;
gser_reception.ux_host_class_gser_reception_data_buffer_size =
    UX_DEMO_RECEPTION_BUFFER_SIZE;
gser_reception.ux_host_class_gser_reception_callback =
    tx_demo_thread_callback;

ux_host_class_gser_reception_start(gser, &gser_reception);
```

ux_host_class_gser_reception_stop

在通用序列接口上停止接收

原型

```
UINT ux_host_class_gser_reception_stop(
    UX_HOST_CLASS_GSER *gser,
    UX_HOST_CLASS_GSER_RECEPTION *gser_reception)
```

说明

此函数在通用序列类接口上停止接收。

参数

- **gser**：指向 gser 类实例的指针。
- **gser_reception**：包含接收参数的结构

返回值

- **UX_SUCCESS**：(0x00) 已完成数据传输。
- **UX_HOST_CLASS_UNKNOWN**：(0x59) 错误的类实例
- **UX_ERROR**：(0x01) 错误

示例

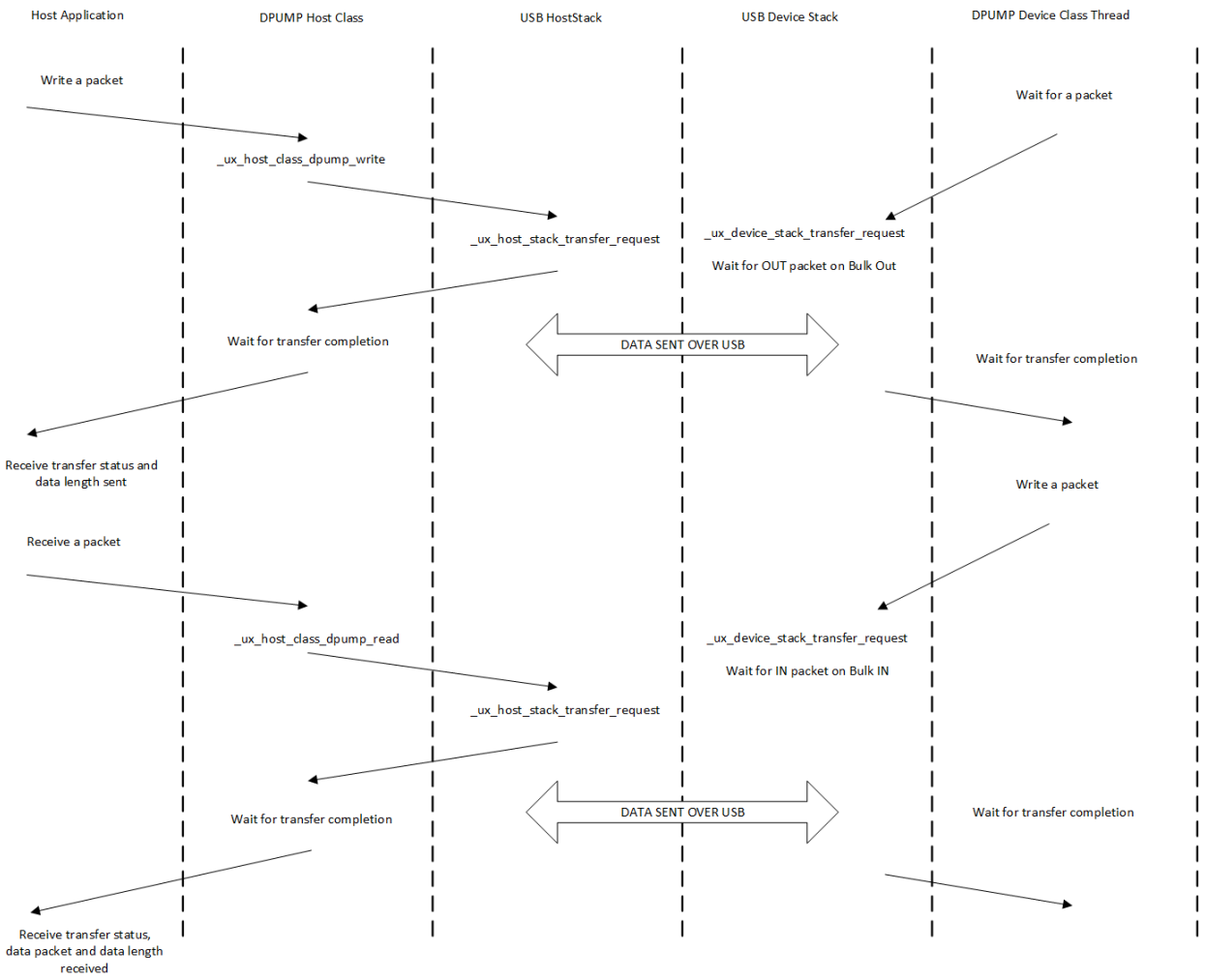
```
/* Stops the reception for gser. */
ux_host_class_gser_reception_stop(gser, &gser_reception);
```

第 3 章：USBX DPUMP 类注意事项

2021/4/29 •

USBX 包含一个用于主机和设备端的 DPUMP 类。此类本身不是一个标准类，而是一个示例，它展示了如何通过以下方式创建一个简单的设备：使用两个大容量管道并在这两个管道上来回发送数据。DPUMP 类可用于启动自定义类，也可用于旧式的 RS232 设备。

USB DPUMP 流程图：



USBX DPUMP 主机类

DPUMP 类的主机端有两个函数，一个用于发送数据，一个用于接收数据：

- `ux_host_class_dpump_write`
- `ux_host_class_dpump_read`

这两个函数都处于阻塞状态，使 DPUMP 应用程序运行起来更容易。如果需要同时运行两个管道（输入和输出），则应用程序需要创建一个传输线程和一个接收线程。

写入函数的原型如下所示。

```
UINT ux_host_class_dpump_write(UX_HOST_CLASS_DPUMP *dpump,
    UCHAR *data_pointer,
    ULONG requested_length,
    ULONG *actual_length)
```

其中：

- dpump 是类的实例
- data_pointer 是指向要发送的缓冲区的指针
- requested_length 是要发送的长度
- actual_length 是在传输完成（成功或部分成功）后发送的长度。

接收函数的原型类似。

```
UINT ux_host_class_dpump_read(
    UX_HOST_CLASS_DPUMP *dpump,
    UCHAR *data_pointer,
    ULONG requested_length,
    ULONG *actual_length)
```

下面是主机 DPUMP 类的一个示例，其中，应用程序将数据包写入到设备端，并在接收端接收相同的数据包：

```
/* We start with a 'A' in buffer. */
current_char = 'A';

while(1)
{
    /* Initialize the write buffer. */
    ux_utility_memory_set(out_buffer, current_char,
        UX_HOST_CLASS_DPUMP_PACKET_SIZE);

    /* Increment the character in buffer. */
    current_char++;

    /* Check for upper alphabet limit. */
    if (current_char > 'Z')
        current_char = 'A';

    /* Write to the Data Pump Bulk out endpoint. */
    status = ux_host_class_dpump_write (dpump, out_buffer,
        UX_HOST_CLASS_DPUMP_PACKET_SIZE,
        &actual_length);

    /* Verify that the status and the amount of data is correct. */
    if ((status == UX_SUCCESS) && actual_length == UX_HOST_CLASS_DPUMP_PACKET_SIZE)
    {
        /* Read to the Data Pump Bulk out endpoint. */
        status = ux_host_class_dpump_read (dpump, in_buffer,
            UX_HOST_CLASS_DPUMP_PACKET_SIZE, &actual_length);
    }
}
```

USBX DPUMP 设备类

设备 DPUMP 类使用一个线程，该线程在连接到 USB 主机时启动。线程在大容量输出终结点上等待数据包的到来。收到数据包时，它将内容复制到大容量输入终结点缓冲区中，在此终结点上发布事务，并等待主机发出从此终结点进行读取的请求。这在大容量输出终结点与大容量输入终结点之间提供了一种环回机制。

第 4 章：USBX Pictbridge 实现

2021/4/29 •

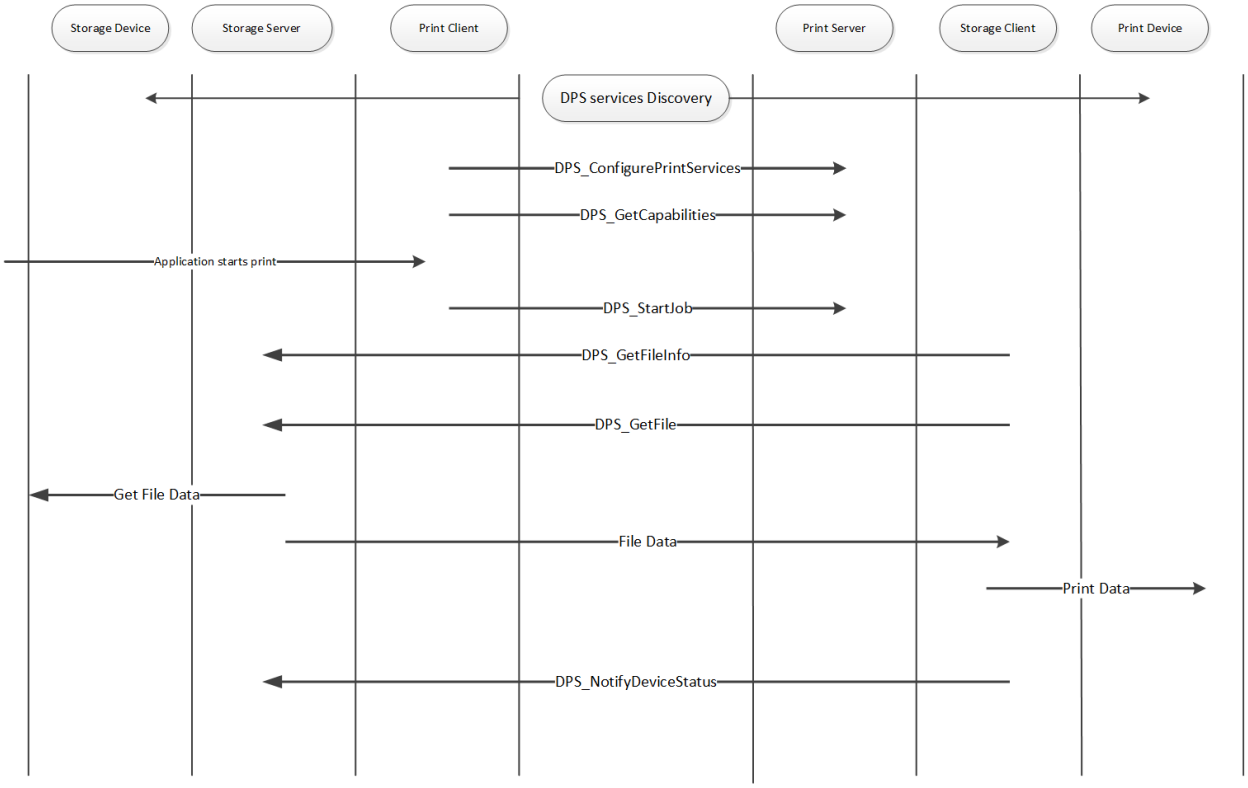
USBX 在主机和设备上都支持完全 Pictbridge 实现。在两端，Pictbridge 都位于 USBX PIMA 类之上。

PictBridge 标准允许将数码照相机或智能手机直接连接到打印机，而不使用 PC，从而可以直接使用特定的 Pictbridge 感知打印机进行打印。

当照相机或手机连接到打印机时，打印机即为 USB 主机，照相机即为 USB 设备。然而，在使用 Pictbridge 时，照相机显示为主机，而且命令是从照相机驱动的。照相机是存储服务器，打印机是存储客户端。照相机是打印客户端，打印机当然是打印服务器。

Pictbridge 使用 USB 作为传输层，但依赖于 PTP（图片传输协议）作为通信协议。

下图展示了在执行打印作业时 DPS 客户端与 DPS 服务器之间的命令/响应。



Pictbridge 客户端实现

客户端上的 Pictbridge 要求先运行 USBX 设备堆栈和 PIMA 类。

设备框架以如下方式描述 PIMA 类。

```
UCHAR device_framework_full_speed[] =
{
    /* Device descriptor */
       0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x20,
       0xA9, 0x04, 0xB6, 0x30, 0x00, 0x00, 0x00, 0x00,
       0x00, 0x01,
    /* Configuration descriptor */
       0x09, 0x02, 0x27, 0x00, 0x01, 0x01, 0x00, 0xc0, 0x32,
    /* Interface descriptor */
       0x09, 0x04, 0x00, 0x00, 0x03, 0x06, 0x01, 0x01, 0x00,
    /* Endpoint descriptor (Bulk Out) */
       0x07, 0x05, 0x01, 0x02, 0x40, 0x00, 0x00,
    /* Endpoint descriptor (Bulk In) */
       0x07, 0x05, 0x82, 0x02, 0x40, 0x00, 0x00,
    /* Endpoint descriptor (Interrupt) */
       0x07, 0x05, 0x83, 0x03, 0x08, 0x00, 0x60
};
```

PIMA 类使用 ID 字段 0x06，对于静态图像，它的子类为 0x01，对于 PIMA 15740，它的协议为 0x01。

此类中定义了 3 个终结点，2 个用于发送/接收数据的批处理，以及 1 个用于事件的中断。

与其他 USBX 设备实现不同，Pictbridge 应用程序本身不需要定义类，而是调用函数 ux_pictbridge_dpsclient_start。示例如下：

```
/* Initialize the Pictbridge string components. */
ux_utility_memory_copy
    (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_vendor_name,
    "Azure RTOS",10);
ux_utility_memory_copy
    (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_product_name,
    "EL_Pictbridge_Camera",21);
ux_utility_memory_copy
    (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_serial_no, "ABC_123",7);
ux_utility_memory_copy
    (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_dpsversions,
    "1.0 1.1",7);

pictbridge.ux_pictbridge_dpslocal.
    ux_pictbridge_devinfo_vendor_specific_version = 0x0100;

/* Start the Pictbridge client. */
status = ux_pictbridge_dpsclient_start(&pictbridge);

if(status != UX_SUCCESS)
    return;
```

传递给 pictbridge 客户端的参数如下所示：

```
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_vendor_name
    : String of Vendor name pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_product_name
    : String of product name pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_serial_no,
    : String of serial number pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_dpsversions
    : String of version pictbridge.ux_pictbridge_dpslocal. ux_pictbridge_devinfo_vendor_specific_version
    : Value set to 0x0100;
```

下一步是让设备和主机同步，并准备好进行信息交换。

这是通过等待事件标志来完成的，如下所示：

```
/* We should wait for the host and the client to discover one another. */
status = ux_utility_event_flags_get
    (&pictbridge.ux_pictbridge_event_flags_group,
    UX_PICTBRIDGE_EVENT_FLAG_DISCOVERY,TX_AND_CLEAR, &actual_flags,
    UX_PICTBRIDGE_EVENT_TIMEOUT);
```

如果状态机处于“DISCOVERY_COMPLETE”状态，则照相机端（DPS 客户端）将收集有关打印机及其功能的信息。

如果 DPS 客户端已准备好接受打印作业，则其状态设置为“UX_PICTBRIDGE_NEW_JOB_TRUE”。可以按如下进行检查：

```
/* Check if the printer is ready for a print job. */
if (pictbridge.ux_pictbridge_dpsclient.ux_pictbridge_devinfo_newjobok ==
    UX_PICTBRIDGE_NEW_JOB_TRUE)

    /* We can print something … */
```

接下来，需要按如下方式填充一些打印工作描述符：

```
/* We can start a new job. Fill in the JobConfig and PrintInfo structures. */
jobinfo = &pictbridge.ux_pictbridge_jobinfo;

/* Attach a printinfo structure to the job. */
jobinfo ->ux_pictbridge_jobinfo_printinfo_start = &printinfo;

/* Set the default values for print job. */
jobinfo ->ux_pictbridge_jobinfo_quality =
    UX_PICTBRIDGE_QUALITIES_DEFAULT;
jobinfo ->ux_pictbridge_jobinfo_papersize =
    UX_PICTBRIDGE_PAPER_SIZES_DEFAULT;
jobinfo ->ux_pictbridge_jobinfo_papertype =
    UX_PICTBRIDGE_PAPER_TYPES_DEFAULT;
jobinfo ->ux_pictbridge_jobinfo_filetype =
    UX_PICTBRIDGE_FILE_TYPES_DEFAULT;
jobinfo ->ux_pictbridge_jobinfo_dateprint =
    UX_PICTBRIDGE_DATE_PRINTS_DEFAULT;
jobinfo ->ux_pictbridge_jobinfo_filenameprint =
    UX_PICTBRIDGE_FILE_NAME_PRINTS_DEFAULT;
jobinfo ->ux_pictbridge_jobinfo_imageoptimize =
    UX_PICTBRIDGE_IMAGE_OPTIMIZES_OFF;
jobinfo ->ux_pictbridge_jobinfo_layout =
    UX_PICTBRIDGE_LAYOUTS_DEFAULT;
jobinfo ->ux_pictbridge_jobinfo_fixedsize =
    UX_PICTBRIDGE_FIXED_SIZE_DEFAULT;
jobinfo ->ux_pictbridge_jobinfo_cropping =
    UX_PICTBRIDGE_CROPPINGS_DEFAULT;

/* Program the callback function for reading the object data. */
jobinfo ->ux_pictbridge_jobinfo_object_data_read =
    ux_demo_object_data_copy;

/* This is a demo, the fileID is hardwired (1 and 2 for scripts, 3 for photo to be printed. */

printinfo.ux_pictbridge_printinfo_fileid =
    UX_PICTBRIDGE_OBJECT_HANDLE_PRINT;

ux_utility_memory_copy(printinfo.ux_pictbridge_printinfo_filename,
    "Pictbridge demo file", 20);
ux_utility_memory_copy(printinfo.ux_pictbridge_printinfo_date, "01/01/2008",
    10);

/* Fill in the object info to be printed. First get the pointer to the object container in the job info
structure. */
object = (UX_SLAVE_CLASS_PIMA_OBJECT *) jobinfo ->
    ux_pictbridge_jobinfo_object;

/* Store the object format: JPEG picture. */
object ->ux_device_class_pima_object_format = UX_DEVICE_CLASS_PIMA_OFC_EXIF_JPEG;
object ->ux_device_class_pima_object_compressed_size = IMAGE_LEN; object -
>ux_device_class_pima_object_offset = 0;
object ->ux_device_class_pima_object_handle_id =
    UX_PICTBRIDGE_OBJECT_HANDLE_PRINT;
object ->ux_device_class_pima_object_length = IMAGE_LEN;

/* File name is in Unicode. */
ux_utility_string_to_unicode("JPEG Image", object ->
    ux_device_class_pima_object_filename);

/* And start the job. */
status =ux_pictbridge_dpsclient_api_start_job(&pictbridge);
```

现在，Pictbridge 客户端有一个打印作业要执行，它将通过字段中定义的回叫从应用程序中一次提取多个图像块。

```
jobinfo ->ux_pictbridge_jobinfo_object_data_read
```

此函数的原型定义如下。

ux_pictbridge_jobinfo_object_data_read

从用户空间复制数据块以供打印。

原型

```
UINT **ux_pictbridge_jobinfo_object_data_read(
    UX_PICTBRIDGE *pictbridge,
    UCHAR *object_buffer,
    ULONG object_offset,
    ULONG object_length,
    ULONG *actual_length)
```

说明

当 DPS 客户端需要检索数据块以在目标 Pictbridge 打印机上打印时，就会调用此函数。

参数

- pictbridge：指向 pictbridge 类实例的指针。
- object_buffer：指向对象缓冲区的指针
- object_offset：从哪里开始读取数据块
- object_length：要返回的长度
- actual_length：返回的实际长度

返回值

- **UX_SUCCESS** (0x00)：此操作成功。
- **UX_ERROR** (0x01)：应用程序无法检索数据。

示例

```
/* Copy the object data. */

UINT ux_demo_object_data_copy(UX_PICTBRIDGE *pictbridge,UCHAR *object_buffer,
    ULONG object_offset, ULONG object_length, ULONG *actual_length)
{
    /* Copy the demanded object data portion. */
    ux_utility_memory_copy(object_buffer, image + object_offset,
        object_length);

    /* Update the actual length. */
    *actual_length = object_length;

    /* We have copied the requested data. Return OK. */
    return(UX_SUCCESS);
}
```

Pictbridge 主机实现

Pictbridge 的主机实现与客户端不同。

在 Pictbridge 主机环境中要做的第一件事是注册 PIMA 类，如下面的示例所示：

```
status = ux_host_stack_class_register(ux_system_host_class_pima_name,
    ux_host_class_pima_entry);
if(status != UX_SUCCESS)
    return;
```

此类是位于 USB 主机堆栈与 Pictbridge 层之间的通用 PTP 层。

下一步是初始化打印服务的 Pictbridge 默认值，如下所示。

PICTBRIDGE ▯▯	▯
DpsVersion[0]	0x00010000
DpsVersion[1]	0x00010001
DpsVersion[2]	0x00000000
VendorSpecificVersion	0x00010000
PrintServiceAvailable	0x30010000
Qualities[0]	UX_PICTBRIDGE_QUALITIES_DEFAULT
Qualities[1]	UX_PICTBRIDGE_QUALITIES_NORMAL
Qualities[2]	UX_PICTBRIDGE_QUALITIES_DRAFT
Qualities[3]	UX_PICTBRIDGE_QUALITIES_FINE
PaperSizes[0]	UX_PICTBRIDGE_PAPER_SIZES_DEFAULT
PaperSizes[1]	UX_PICTBRIDGE_PAPER_SIZES_4IX6I
PaperSizes[2]	UX_PICTBRIDGE_PAPER_SIZES_L
PaperSizes[3]	UX_PICTBRIDGE_PAPER_SIZES_2L
PaperSizes[4]	UX_PICTBRIDGE_PAPER_SIZES_LETTER
PaperTypes[0]	UX_PICTBRIDGE_PAPER_TYPES_DEFAULT
PaperTypes[1]	UX_PICTBRIDGE_PAPER_TYPES_PLAIN
PaperTypes[2]	UX_PICTBRIDGE_PAPER_TYPES_PHOTO
FileTypes[0]	UX_PICTBRIDGE_FILE_TYPES_DEFAULT
FileTypes[1]	UX_PICTBRIDGE_FILE_TYPES_EXIF_JPEG
FileTypes[2]	UX_PICTBRIDGE_FILE_TYPES_JFIF
FileTypes[3]	UX_PICTBRIDGE_FILE_TYPES_DPOF

PICTBRIDGE 功能	值
DatePrints[0]	UX_PICTBRIDGE_DATE_PRINTS_DEFAULT
DatePrints[1]	UX_PICTBRIDGE_DATE_PRINTS_OFF
DatePrints[2]	UX_PICTBRIDGE_DATE_PRINTS_ON
FileNamePrints[0]	UX_PICTBRIDGE_FILE_NAME_PRINTS_DEFAULT
FileNamePrints[1]	UX_PICTBRIDGE_FILE_NAME_PRINTS_OFF
FileNamePrints[2]	UX_PICTBRIDGE_FILE_NAME_PRINTS_ON
ImageOptimizes[0]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_DEFAULT
ImageOptimizes[1]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_OFF
ImageOptimizes[2]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_ON
Layouts[0]	UX_PICTBRIDGE_LAYOUTS_DEFAULT
Layouts[1]	UX_PICTBRIDGE_LAYOUTS_1_UP_BORDER
Layouts[2]	UX_PICTBRIDGE_LAYOUTS_INDEX_PRINT
Layouts[3]	UX_PICTBRIDGE_LAYOUTS_1_UP_BORDERLESS
FixedSizes[0]	UX_PICTBRIDGE_FIXED_SIZE_DEFAULT
FixedSizes[1]	UX_PICTBRIDGE_FIXED_SIZE_35IX5I
FixedSizes[2]	UX_PICTBRIDGE_FIXED_SIZE_4IX6I
FixedSizes[3]	UX_PICTBRIDGE_FIXED_SIZE_5IX7I
FixedSizes[4]	UX_PICTBRIDGE_FIXED_SIZE_7CMX10CM
FixedSizes[5]	UX_PICTBRIDGE_FIXED_SIZE_LETTER
FixedSizes[6]	UX_PICTBRIDGE_FIXED_SIZE_A4
Croppings[0]	UX_PICTBRIDGE_CROPPINGS_DEFAULT
Croppings[1]	UX_PICTBRIDGE_CROPPINGS_OFF
Croppings[2]	UX_PICTBRIDGE_CROPPINGS_ON

DPS 主机的状态机将被设置为“空闲”，并准备好接受新的打印作业。

现在可以启动 Pictbridge 的主机部分，如下面的示例所示。

```
/* Activate the pictbridge dpshost. */

status = ux_pictbridge_dpshost_start(&pictbridge, pima);
if (status != UX_SUCCESS)
    return;
```

当数据可供打印时，Pictbridge 主机函数需要回叫。这是通过在 pictbridge 主机结构中传递函数指针来完成的，如下所示。

```
/* Set a callback when an object is being received. */

pictbridge.ux_pictbridge_application_object_data_write =
    tx_demo_object_data_write;
```

此函数有以下属性：

ux_pictbridge_application_object_data_write

编写数据块以供打印。

原型

```
UINT **ux_pictbridge_application_object_data_write(
    UX_PICTBRIDGE *pictbridge,
    UCHAR *object_buffer,
    ULONG offset,
    ULONG total_length,
    ULONG length);
```

说明

当 DPS 服务器需要从 DPS 客户端中检索数据块以在本地打印机上打印时，就会调用此函数。

参数

- pictbridge：指向 pictbridge 类实例的指针。
- object_buffer：指向对象缓冲区的指针
- object_offset：从哪里开始读取数据块
- total_length：对象的整个长度
- length：此缓冲区的长度

返回值

- **UX_SUCCESS** (0x00)：此操作成功。
- **UX_ERROR** (0x01)：应用程序无法打印数据。

示例

```
/* Copy the object data. */

UINT tx_demo_object_data_write(UX_PICTBRIDGE *pictbridge,
    UCHAR *object_buffer, ULONG offset, ULONG total_length, ULONG length);
{
    UINT status;

    /* Send the data to the local printer. */
    status = local_printer_data_send(object_buffer, length);

    /* We have printed the requested data. Return status. */
    return(status);
}
```

第 5 章：USBX OTG

2021/4/29 •

当在硬件设计中采用兼容 OTG 的 USB 控制器时，USBX 支持使用 USB 的 OTG 功能。

USBX 支持核心 USB 堆栈中的 OTG。但要使 OTG 正常工作，它需要一个特定的 USB 控制器。USBX OTG 控制器函数可在 usbx_otg 目录中找到。当前 USBX 版本仅支持具有完全 OTG 功能的 NXP LPC3131。

常规的控制器驱动程序函数（主机或设备）仍然可以在标准 USBX usbx_device_controllers 和 usbx_host_controllers 中找到，但是 usbx_otg 目录包含与 USB 控制器关联的特定 OTG 函数。

除了常见的主机/设备函数外，OTG 控制器还有四类函数。

- VBUS 特定函数
- 控制器的启动和停止
- USB 角色管理器
- 中断处理程序

VBUS 函数

每个控制器都需要配备一个 VBUS 管理器，以根据电源管理要求更改 VBUS 的状态。通常，此函数仅用于打开或关闭 VBUS

启动和停止控制器

与常规的 USB 实现不同，OTG 要求在角色发生更改时激活和停用主机和/或设备堆栈。

USB 角色管理器

USB 角色管理器可接收用于更改 USB 状态的命令。有几种状态需要与下表中给出的状态相互转换。

“	“[”	“
UX_OTG_IDLE	0	设备处于空闲状态。未连接
UX_OTG_IDLE_TO_HOST	1	设备已连接 A 类型连接器
UX_OTG_IDLE_TO_SLAVE	2	设备已连接 B 类型连接器
UX_OTG_HOST_TO_IDLE	3	主机设备已断开连接
UX_OTG_HOST_TO_SLAVE	4	将角色从主机调换为从属设备
UX_OTG_SLAVE_TO_IDLE	5	从属设备已断开连接
UX_OTG_SLAVE_TO_HOST	6	将角色从从属设备调换为主机

中断处理程序

OTG 的主机驱动程序驱动程序和设备控制器驱动程序都需要不同的中断处理程序来监视传统 USB 中断以外的信号，尤其是 SRP 和 VBUS 引起的信号。

如何初始化 USB OTG 控制器。此处以 NXP LPC3131 为例。

```
/* Initialize the LPC3131 OTG controller. */
status = ux_otg_lpc3131_initialize(0x19000000, lpc3131_vbus_function,
    tx_demo_change_mode_callback);
```

在此示例中，我们通过传递 VBUS 函数和回调模式更改（从主机到从属设备，反之亦然）来在 OTG 模式下初始化 LPC3131。

回调函数应仅记录新模式并唤醒待处理的线程以执行新状态。

```
void tx_demo_change_mode_callback(ULONG mode)
{
    /* Simply save the otg mode. */
    otg_mode = mode;

    /* Wake up the thread that is waiting. */
    ux_utility_semaphore_put(&mode_change_semaphore);
}
```

传递的模式值可以具有以下值。

- UX_OTG_MODE_IDLE
- UX_OTG_MODE_SLAVE
- UX_OTG_MODE_HOST

应用程序始终可以通过查看变量来确定是哪个设备。

```
ux_system_otg ->ux_system_otg_device_type
```

其值可以是下列值之一。

- UX_OTG_DEVICE_A
- UX_OTG_DEVICE_B
- UX_OTG_DEVICE_IDLE

USB OTG 主机设备始终可以通过发出命令来请求角色交换。

```
/* Ask the stack to perform a HNP swap with the device. We relinquish the host role to A device. */

ux_host_stack_role_swap(storage ->ux_host_class_storage_device);
```

对于从属设备，没有要发出的命令，但从属设备可以设置状态以更改角色，主机在发出 GET_STATUS 时将选取该角色，然后发起交换。

```
/* We are a B device, ask for role swap. The next GET_STATUS from the host will get the status change and do
the HNP. */

ux_system_otg ->ux_system_otg_slave_role_swap_flag =
    UX_OTG_HOST_REQUEST_FLAG;
```