Contents

Azure RTOS FileX 文档
FileX 概述
FileX 用户指南
关于本指南
第 1 章 - FileX 简介
第 2 章 - FileX 的安装和使用
第 3 章 - FileX 的功能组件
第 4 章 - FileX 服务的说明
第 5 章 - FileX 的 I/O 驱动程序
第 6 章 - FileX 容错模块
附录 A - FileX 服务
附录 B - FileX 常量
附录 C - FileX 数据类型
附录 D - ASCII 字符代码
FileX 存储库
相关服务
Defender for IoT - RTOS（预览版）
Microsoft Azure RTOS 组件
Microsoft Azure RTOS
ThreadX
ThreadX Modules
NetX Duo
NetX
GUIX
FileX
LevelX
USBX
TraceX

Azure RTOS FileX 概述

2021/4/30 •

Azure RTOS FileX 嵌入式文件系统是 Azure RTOS 的先进工业级解决方案，适用于 Microsoft FAT 文件格式，专为深度嵌入式实时 IoT 应用程序而设计。Azure RTOS FileX 支持 Microsoft 的所有文件格式，包括 FAT12、FAT16、FAT32 和 exFAT。FileX 还通过名为 Azure RTOS LevelX 的附加产品提供可选的容错和闪存磨损均衡 (wear leveling) 技术。凭借所有这些以及占用内存少、执行速度快、易于使用的优势，Azure RTOS FileX 已成为要求最高的嵌入式 IoT 应用程序的理想选择。

API 协议

媒体服务

- FAT 12/16/32 和 exFAT 支持
- 占用空间极小，只需 6KB 闪存，2.5KB RAM
- 完整的媒体访问服务
- 无限数目的媒体实例
- 简单的读/写逻辑扇区驱动程序接口
- 多分区支持
- 逻辑扇区缓存
- FAT 条目缓存
- 可选容错支持
- 辅助 FAT 延缓更新
- 通过 Azure RTOS TraceX 进行系统级跟踪
- 直观的媒体访问 API，包括：
  - fx_media_open
  - fx_media_close
  - fx_media_format
  - fx_media_space_available

目录服务

- 最多 256 字节的路径
- 支持长目录名称和 8.3 目录名称
- 目录创建和删除
- 目录导航和遍历
- 目录特性管理
- 通过 Azure RTOS TraceX 进行系统级跟踪
- 直观的目录访问 API，包括：
  - fx_directory_create
  - fx_directory_delete
  - fx_directory_attributes_set
  - fx_directory_attributes_read
  - fx_directory_first_entry_find
  - fx_directory_next_entry_find

文件服务

- 占用空间极小，只需 3.3KB 闪存

- 打开的文件数无限制
- 可以多次打开只读文件
- 支持长目录名称和 8.3 目录名称
- 连续文件支持
- 快速寻道逻辑
- 簇预分配
- 文件创建、删除和重命名
- 文件读取、写入和查看
- 文件特性管理
- 通过 Azure RTOS TraceX 进行系统级跟踪
- 直观的文件访问 API，包括：
  - fx_file_create
  - fx_file_delete
  - fx_file_attributes_set
  - fx_file_attributes_read
  - fx_file_read
  - fx_file_seek
  - fx_file_write

先进技术

Azure RTOS FileX 是先进的技术，其中包括以下优势。

- FAT 12/16/32 和 exFAT 支持
- 多分区支持
- 自动缩放
- 支持各种字节序
- 长文件名和 8.3 文件名支持
- 可选容错支持
- 逻辑扇区缓存
- FAT 条目缓存
- 簇预分配
- 连续文件支持
- 可选性能指标
- Azure RTOS TraceX 系统分析支持

NOR/NAND 磨损均衡 (Azure RTOS LevelX)

Azure RTOS LevelX 是 Microsoft 的 NOR/NAND 闪存磨损均衡产品。Azure RTOS LevelX 可与 FileX 结合使用，也可以单独用作应用程序的直接读/写闪存扇区库。

关于本 FileX 用户指南

2021/4/30 •

本指南包含有关 Azure RTO FileX（Microsoft 提供的高性能实时嵌入式文件系统）的综合信息。为了充分利用本指南，你应该熟悉标准的实时操作系统函数、FAT 文件系统服务和 C 编程语言。

组织

第 1 章 - 介绍 Azure RTOS FileX

第 2 章 - 介绍安装 Azure RTOS FileX 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 FileX 系统的功能概述和有关 FAT 文件系统格式的基本信息

第 4 章 - 详细介绍应用程序的 Azure RTOS NetX 接口

第 5 章 - 介绍提供的 Azure RTOS FileX RAM 驱动程序以及如何编写你自己的自定义 Azure RTOS FileX 驱动程序

第 6 章 - 介绍 Azure RTOS FileX 容错模块

附录 A - Azure RTOS FileX 服务

附录 B - Azure RTOS FileX 常量

附录 C - Azure RTOS FileX 数据类型

附录 D - ASCII 图表

指南约定

斜体字样表示书名，强调重要字词并表示变量。

粗体字样表示文件名和关键字，并进一步强调重要字词和变量。

> **NOTE**
>
> 信息符号提示开发人员注意可能影响性能或功能的重要信息或附加信息。

> **IMPORTANT**
>
> 警告符号提示开发人员应当注意避免的情况，这些情况可能会导致灾难性错误。

FileX 数据类型

除了自定义的 Azure RTOS FileX 控制结构数据类型之外，我们还提供了一系列特殊的数据类型，用于 Azure RTOS FileX 服务调用接口。这些特殊数据类型直接映射到基础 C 编译器的数据类型。这样做是为了确保在不同 C 编译器之间的可移植性。具体实现继承自 Azure RTOS ThreadX，可在 Azure RTOS ThreadX 分发中包含的 tx_port.h 文件中找到。

下面列出了 Azure RTOS FileX 服务调用数据类型及其关联的含义。| 类型 | 说明 | |---|---| | UINT | 基本的无符号整数。此类型必须支持 8 位无符号数据；但是，它映射到最方便的无符号数据类型。| | ULONG | 无符号 long 类型。此类型必须支持 32 位无符号数据。| | VOID | 几乎始终等效于编译器的 VOID 类型。| | CHAR | 最常为标准的 8 位字符类型。| | ULONG64 | 64 位无符号整数数据类型。|

FileX 源中还使用了其他数据类型。这些数据类型位于 tx_port.h 或 fx_port.h 文件中。

客户支持中心

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

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

   _tx_build_options

   _fx_system_build_options1

   _fx_system_build_options2

   _fx_system_build_options3

第 1 章 - Azure RTOS FileX 简介

2021/4/29 •

Azure RTOS FileX 是一个完全 FAT 格式的介质和文件管理系统，适用于深层嵌入应用程序。本章介绍了 FileX 及其应用程序和优点。

FileX 的独特功能

Azure RTOS FileX 同时支持无限量的介质设备，包括 RAM 磁盘、FLASH 管理器和实际的物理设备。它支持 12、16 和 32 位文件分配表 (FAT) 格式，还支持扩展文件分配表 (exFAT)、连续文件分配，并针对大小和性能进行了高度优化。FileX 还包括容错支持、介质开启/关闭和文件写入回叫函数。

FileX 使用与 ThreadX 相同的设计和编码方法，旨在满足不断增长的 FLASH 设备需求。与所有 Microsoft 产品一样，FileX 使用完整的 ANSI C 源代码进行分发，并且没有运行时版税。

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
- 不受限制的 FileX 对象（介质、目录和文件）
- 动态 FileX 对象创建/删除
- 可变内存使用
- 自动缩放大小
- 小型内存占用情况（低至 6 Kb）指令区域大小：6-30000
- 完成与 ThreadX 的集成
- 支持各种字节序
- 易于实现的 FileX I/O 驱动程序
- 12、16 和 32 位 FAT 支持
- exFAT 支持
- 长文件名支持
- 内部 FAT 条目缓存
- Unicode 名称支持
- 连续文件分配
- 连续扇区和群集读取/写入
- 内部逻辑扇区缓存
- RAM 磁盘演示开箱即用
- 介质格式功能
- 错误检测和恢复
- 容错选项
- 内置性能统计信息
- 独立支持（无 Azure RTOS）

安全认证

TÜV 认证

FileX 已被 SGS-TÜV Saar 认证可用于安全关键型系统，并且符合 IEC-61508 和 IEC-62304 标准。该认证证明：FileX 可用于开发达到国际电工委员会 (IEC) 61508 和 IEC 62304 最高安全完整性等级的安全相关软件，这些安全完整性级别旨在确保“电气设备、电子设备和可编程的安全相关电子系统的功能安全”。SGS-TÜV Saar 由德国的 SGS-Group 和 TÜV Saarland 合并而成，现已成为领先的经过资格验证的独立公司，专门为全球的安全相关系统测试、审核、验证和认证嵌入式软件。工业安全标准 IEC 61508 以及从其派生的所有标准(包括 IEC 62304)用于确保电气设备、电子设备和可编程的安全相关电子医疗设备、流程控制系统、工业机械和铁路控制系统的功能安全。

SGS-TÜV Saar 已根据 ISO 26262 标准对 FileX 进行了认证，确定其可用于安全关键型汽车系统。此外，FileX 还获得了汽车安全完整性等级 (ASIL) D 的认证，该等级代表了 ISO 26262 认证的最高等级。

而且，SGS-TÜV Saar 已对 FileX 进行了认证，确定其可用于安全关键型铁路系统，符合 EN 50128 标准并达到 SW-SIL 4 等级。

- IEC 61508，达到 SIL 4 等级
- IEC 62304，SW 安全类别为 C 类
- ISO 26262 ASIL D
- EN 50128 SW-SIL 4

> **IMPORTANT**
>
> 请联系我们，了解 FileX 的哪些版本已通过 TÜV 认证，或者了解如何获取测试报表、证书和相关文档。*

UL 认证

FileX 已通过 UL 的认证，符合面向可编程软件组件的 UL 60730-1 Annex H、CSA E60730-1 Annex H、IEC 60730-1 Annex H、UL 60335-1 Annex R、IEC 60335-1 Annex R 和 UL 1998 安全标准。连同 IEC/UL 60730-1(其附件 H 中对“使用软件进行控制”的要求)一起，IEC 60335-1 标准在其附件 R 中描述了“可编程电子电路”的要求。IEC 60730 附件 H 和 IEC 60335 -1 附件 R 阐述了在洗衣机、洗碗机、烘干机、冰箱、冰柜和烤箱等电器中使用的 MCU 硬件和软件的安全性。

UL/IEC 60730、UL/IEC 60335、UL 1998

> **IMPORTANT**
>
> 请联系我们，了解 FileX 的哪些版本已通过 TÜV 认证，或者了解如何获取测试报表、证书和相关文档。

强大的 FileX 服务

多介质管理

FileX 可以支持无限量的物理介质。每个介质实例都有其自己的不同内存区域和 fx_media_open 调用上指定的相关驱动程序。FileX 的默认分布附带了一个简单的 RAM 介质驱动程序和一个使用此 RAM 磁盘的演示系统。

逻辑扇区缓存

通过减少与介质之间的整个扇区传输数，FileX 逻辑扇区缓存可显著提高性能。FileX 为每个打开的介质维护逻辑扇区缓存。逻辑扇区缓存的深度取决于使用 fx_media_open API 调用提供给 FileX 的内存量。

连续文件支持

FileX 通过 API 服务 fx_file_allocate 提供连续的文件支持，以改善文件访问时间并使其具有确定性。此例程获取所请求的内存量，并查找一系列相邻群集来满足请求。如果找到此类群集，则会使其成为已分配群集链的一部分来预分配这些群集。在移动物理介质时，FileX 连续文件支持会显著提高性能，并使访问时间具有确定性。

动态创建

FileX 允许动态创建系统资源。如果你的应用程序有多个或动态配置要求，则这一点尤其重要。此外，可以使用无预先确定限量的 FileX 资源（介质或文件）。而且，系统对象的数量不会对性能产生任何影响。

易于使用的 API

FileX 以一种易于理解且易于使用的方式提供了最佳的深度嵌入文件系统技术！ FileX 应用程序编程接口 (API) 使服务直观且一致。无需解密其他文件系统中过于常见的“字母汤”服务。

有关 FileX 版本 5 服务的完整列表，请参阅附录 A。

exFAT 支持

exFAT（扩展文件分配表）是 Microsoft 设计的一种文件系统，允许文件大小超过 2 GB，即 FAT32 文件系统的限制。它是容量超过 32GB 的 SD 卡的默认文件系统。用 FileX exFAT 格式进行格式设置的 SD 卡或 U 盘与 Windows 兼容。exFAT 支持的文件大小可高达 1 百亿亿字节 (EB)，约为 10 亿 GB。

希望使用 exFAT 的用户必须在定义了 FX_ENABLE_EXFAT 符号的情况下重新编译 FileX 库。打开介质时，FileX 将检测到介质类型。如果介质使用 exFAT 进行格式设置，则 FileX 会按照 exFAT 标准读写文件系统。若要使用 exFAT 设置新介质的格式，请使用服务 fx_media_exFAT_format。默认情况下，exFAT 未启用。

容错支持

FileX 容错模块旨在防止文件或目录更新期间由中断导致的文件系统损坏。例如，在将数据追加到文件时，FileX 需要更新文件内容、目录条目和 FAT 条目。如果此更新顺序中断（例如电源故障，或者介质在更新过程中弹出），则文件系统处于不一致状态，这可能会影响整个文件系统的完整性，从而导致其他文件的损坏。

FileX 容错模块的工作原理是记录整个过程中更新文件或目录所需的所有步骤。此日志条目存储在 FileX 可以查找和访问的专用扇区（块）上。即使没有适当的文件系统，也可以访问日志数据的位置。因此，在文件系统损坏的情况下，FileX 仍能找到日志条目，并将文件系统还原到良好状态。

FileX 更新文件或目录时，会创建日志条目。成功完成更新操作后，会删除日志条目。如果在成功更新文件后未正确删除日志条目，恢复过程确定日志条目中的内容与文件系统一致，则无需执行任何操作即可清理日志条目。

如果文件系统更新操作中断，则下一次 FileX 装载介质时，容错模块会分析日志条目。日志条目中的信息允许 FileX 返回已应用到文件系统的部分更改（以防在文件更新操作初期发生失败），或者如果日志条目包含恢复信息，FileX 可以应用完成先前操作所需的更改。

此容错功能适用于 FileX 支持的所有 FAT 文件系统，包括 FAT12、FAT16、FAT32 和 exFAT。默认情况下，FileX 中不启用容错。若要启用容错功能，必须在定义了 FX_ENABLE_FAULT_TOLERANT 和 FX_FAULT_TOLERANT 符号的情况下生成 FileX 。在运行时，应用程序调用 fx_fault_tolerant_enable 启动容错服务。服务启动后，所有文件和目录写入操作都将经历容错模块。

当容错服务启动时，它首先检测介质是否受容错模块保护。如果不是，FileX 会假定文件系统完整，并从文件系统分配可用于记录和缓存的自由块来启动保护。如果在文件系统上找到容错模块日志，它会分析日志条目。FileX 会还原先前的操作或恢复之前的操作，具体取决于日志条目的内容。在处理完所有以前的日志条目后，文件系统将变为可用状态。这可确保 FileX 从已知的良好状态启动。

在 FileX 容错模块下保护介质后，该介质不会由另一个文件系统进行更新。这样做会使文件系统上的日志条目与

FAT 表中的内容(目录条目)不一致。如果在将介质移回带有容错模块的 FileX 之前由另一个文件系统进行了更新，则结果可能不确定。

回调函数

以下三个回叫函数已添加到 FileX：

- 介质开启回叫
- 介质关闭回叫
- 文件写入回叫

注册后，这些函数将在发生此类事件时通知应用程序。

易于集成

FileX 可以轻松地与几乎所有 FLASH 或介质设备集成。移植 FileX 非常简单。本指南详细介绍了该过程，并且从演示系统的 RAM 驱动程序开始是一个不错的选择！

第 2 章 - 安装和使用 Azure RTOS FileX

2021/4/29 •

本章介绍 Azure RTOS FileX 及其安装条件、安装过程和用法。

主机注意事项

计算机类型

嵌入式开发通常在 Windows 或 Linux (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

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

调试工具

大多数开发工具调试器通过 JTAG (IEEE 1149.1) 和背景调试模式 (BDM) 等芯片调试 (OCD) 连接与目标硬件进行通信。调试器还通过线路内仿真 (ICE) 连接与目标硬件进行通信。OCD 和 ICE 连接提供可靠的解决方案，对目标常驻软件的干扰已降至最低限度。

所需的硬盘空间

FileX 的源代码以 ASCII 格式提供，并要求主计算机的硬盘上具有约 500 KB 的可用空间

目标注意事项

FileX 要求目标具有 6 KB 到 30 KB 的只读内存 (ROM)。FileX 全局数据结构要求目标具有额外的 100 字节随机存取内存 (RAM)。每个打开的媒体除了要求提供 RAM（通常为 512 字节）用于存储一个扇区的数据以外，还要求为控制块提供 1.5 KB RAM。

为了使日期/时间戳记正常运行，FileX 依赖于 ThreadX 计时器设施。这是通过在 FileX 初始化期间创建 FileX 特定的计时器实现的。FileX 还依赖于 ThreadX 信号灯来实现多线程保护和 I/O 挂起。

产品分发

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

下面列出了该存储库中的几个重要文件：

- fx_api.h：此 C 头文件包含所有系统等式、数据结构和服务原型。
- fx_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。
- demo_filex.c：此 C 文件包含小型演示应用程序。
- fx.a（或 fx.lib）：这是 FileX C 库的二进制版本。它随标准包一起分发。

> **IMPORTANT**
>
> 所有文件均采用小写文件名。通过此命名约定，你可以更轻松地将命令转换到 Linux (Unix) 开发平台。

FileX 安装

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

语法:

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

或者，也可以使用 GitHub 主页上的“下载”按钮来下载存储库的副本。

还可以在联机存储库的首页上找到有关生成 FileX 库的说明。

> **IMPORTANT**
> 应用程序软件需要访问 FileX 库文件(通常为 fx.a 或 fx.lib)和 C include 文件 fx_api.h 和 fx_port.h。为实现此目的，可以设置开发工具的相应路径，或者将这些文件复制到应用程序开发区域。

使用 FileX

FileX 很容易使用。简单而言，应用程序代码在编译期间必须包含 fx_api.h，并且必须与 FileX 运行时库 fx.a(或 fx.lib)相链接。当然，ThreadX 文件 - 即 tx_api.h 和 tx.a(或 tx.lib) - 也是必需的。

> **IMPORTANT**
> 在独立模式下使用 FileX 时(必须定义 FX_STANDALONE_ENABLE)，不需要 ThreadX 文件/库。

假设你已在使用 ThreadX，需要执行四个步骤来生成 FileX 应用程序:

1. 在所有使用 FileX 服务或数据结构的应用程序文件中包括 fx_api.h 文件。
2. 通过从 tx_application_define 函数或应用程序线程调用 fx_system_initialize 来初始化 FileX 系统。

   > **IMPORTANT**
   > 在独立模式下使用 FileX 时，应直接从应用程序代码调用 fx_system_initialize。

3. 添加对 fx_media_open 的一个或多个调用可设置 FileX 媒体。必须从应用程序线程的上下文发出此调用。

   > **IMPORTANT**
   > 请记住，fx_media_open 调用需要足够的 RAM 来存储一个扇区的数据。

4. 编译应用程序源，并与 FileX 和 ThreadX 运行库 fx.a(或 fx.lib)及 tx.a(或 tx.lib)相链接。生成的映像可下载到目标并执行!

疑难解答

每个 FileX 端口都随演示应用程序一起提供。最好先让演示系统在目标硬件或特定演示环境上运行。

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:

1. 确定演示的运行量。
2. 增加堆栈大小(这在实际的应用程序代码中比在演示中更为重要)。
3. 请确保为 32 KB 默认 RAM 磁盘大小提供足够的 RAM。基本系统运行时使用的 RAM 会少得多;但是，随着使用的 RAM 磁盘越来越多，在内存不足的情况下，问题就会出现。
4. 暂时跳过最近所做的任何更改，以查看问题是否消失或发生更改。此类信息对于 Microsoft 支持工程师非常有用。按照“客户支持中心”中概述的步骤，发送从故障排除步骤中收集的信息。

配置选项

使用 FileX 生成 FileX 库和应用程序时，有几个配置选项可用。可以在应用程序源、命令行或 fx_user.h 包含文件中定义以下选项。

> **IMPORTANT**
> 仅当在定义了 FX_INCLUDE_USER_DEFINE_FILE 的情况下生成应用程序和 ThreadX 库时，才会应用 fx_user.h 中定义的选项。在独立模式下使用 FileX 时(必须定义 FX_STANDALONE_ENABLE)，不需要 ThreadX 文件/库。**

以下列表详细描述了每个配置选项：

"	"
FX_MAX_LAST_NAME_LEN	此值定义最大文件名长度(包括完整路径名称)。默认情况下，此值为 256。
FX_DONT_UPDATE_OPEN_FILES	已定义，FileX 不会更新已打开的文件。
FX_MEDIA_DISABLE_SEARCH_CACHE	已定义，将禁用文件搜索缓存优化。
FX_MEDIA_DISABLE_SEARCH_CACHE	已定义，将禁用文件搜索缓存优化。
FX_DISABLE_DIRECT_DATA_READ_CACHE_FILL	已定义，将禁用缓存的直接读取扇区更新。
FX_MEDIA_STATISTICS_DISABLE	已定义，将禁用媒体统计信息收集。
FX_SINGLE_OPEN_LEGACY	已定义，将为同一文件启用传统单开逻辑。
FX_RENAME_PATH_INHERIT	已定义，将重命名继承路径信息。
FX_DISABLE_ERROR_CHECKING	删除基本 FileX 错误检查 API，可提高性能(最高 30%)并减少代码量。
FX_MAX_LONG_NAME_LEN	指定 FileX 的最大文件名大小。默认值为 256，但可以使用命令行定义重写此值。合法值的范围为 13 到 256。
FX_MAX_SECTOR_CACHE	指定可由 FileX 缓存的最大逻辑扇区数。可缓存的实际扇区数小于此常量，以及在 fx_media_open 提供的内存量中可装入的扇区数。默认值为 256。所有值必须是 2 的幂。
FX_FAT_MAP_SIZE	指定可在 FAT 更新映射中表示的扇区数。默认值为 256，但可以使用命令行定义重写此值。较大的值有助于减少不必要地更新辅助 FAT 扇区。
FX_MAX_FAT_CACHE	指定内部 FAT 缓存中的条目数。默认值为 16，但可以使用命令行定义重写此值。所有值必须是 2 的幂。
FX_FAULT_TOLERANT	如果已定义，FileX 会立即将所有系统扇区(启动、FAT 和目录扇区)的写入请求传递到媒体的驱动程序。这可能会降低性能，但有助于限制对已丢失的簇造成损坏。请注意，启用此功能不会自动启用 FileX 容错模块(可通过定义来启用)

"	"
FX_FAULT_TOLERANT_DATA	如果已定义，FileX 会立即将所有文件数据写入请求传递到媒体的驱动程序。这可能会降低性能，但有助于限制文件数据丢失。请注意，启用此功能不会自动启用 FileX 容错模块，该模块可通过定义 FX_ENABLE_FAULT_TOLERANT 启用
FX_NO_LOCAL_PATH	从 FileX 中删除本地路径逻辑，从而减少代码量。
FX_NO_TIMER	消除了用于更新 FileX 系统时间和日期的 ThreadX 计时器设置。这会导致在所有文件操作中添加默认时间和日期。
FX_UPDATE_RATE_IN_SECONDS	指定调整 FileX 中系统时间的频率。默认情况下值为 10，它指定每隔 10 秒更新 FileX 系统时间。
FX_ENABLE_EXFAT	如果已定义，则在 FileX 中启用用于处理 exFAT 文件系统的逻辑。默认未定义此符号。
FX_UPDATE_RATE_IN_TICKS	指定与 FX_UPDATE_RATE_IN_SECONDS（参阅上文）相同的频率，但基础 ThreadX 计时器频率方面除外。默认值为 1000（采用 10 毫秒 ThreadX 计时器频率和 10 秒间隔）。
FX_SINGLE_THREAD	消除 FileX 源中的 ThreadX 保护逻辑。如果仅从一个线程使用 FileX，或者使用了 FileX 但未使用 ThreadX，则应使用该选项。
FX_DRIVER_USE_64BIT_LBA	如果已定义后，将启用 I/O 驱动程序中使用的 64 位扇区地址。默认未定义此选项。
FX_ENABLE_FAULT_TOLERANT	如果已定义，将启用 FileX 容错模块。启用容错会自动定义符号 FX_FAULT_TOLERANT 和 FX_FAULT_TOLERANT_DATA。默认未定义此选项。
FX_FAULT_TOLERANT_BOOT_INDEX	定义容错日志簇所在的引导扇区中的字节偏移量。默认情况下，此值为 116。此字段采用 4 个字节。之所以选择第 116 到 119 字节，是因为它们已根据 FAT 12/16/32/exFAT 规范标记为保留字节。
FX_FAULT_TOLERANT_MINIMAL_CLUSTER	此符号已弃用。FileX 容错不再使用它。
FX_STANDALONE_ENABLE	已定义，将允许在独立模式下使用 FileX（不带 Azure RTOS）。默认未定义此符号。

IMPORTANT

如果已定义 FX_STANDALONE_ENABLE，则会禁用本地路径逻辑和 ThreadX 计时器设置。

FileX 版本 ID

当前版本的 FileX 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 fx_port.h 文件来获取 FileX 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 fx_version_id 来获取 FileX 版本。_

第 3 章 - Azure RTOS FileX 功能组件

2021/4/30 •

本章旨在从功能角度介绍高性能 Azure RTOS FileX 嵌入式文件系统。

介质说明

FileX 是符合 FAT 文件系统格式的高性能嵌入式文件系统。FileX 将物理介质视为逻辑扇区数组。如何将这些扇区与底层物理介质相映射取决于 fx_media_open 调用期间连接到 FileX 介质的 I/O 驱动程序。

FAT12/16/32 逻辑扇区

介质逻辑扇区的确切组织取决于物理介质启动记录的内容。介质逻辑扇区的一般布局如图 1 所示。

FileX 逻辑扇区始于逻辑扇区 1，指向介质的第一个保留扇区。虽然保留扇区为可选，但在使用时，这些扇区通常包含启动代码等系统信息。

FAT12/16/32 介质启动记录

介质"逻辑扇区"视图中其他区域的确切扇区偏移来自介质启动记录的内容。启动记录通常位于第 0 扇区。但是，如果介质具有隐藏扇区，则启动扇区的偏移必须考虑到这些隐藏扇区(位于启动扇区之前)。表 1 列出了介质的启动记录组件，这些组件将在下文段落中进行介绍。

- **跳转指令**"跳转指令"字段是一个三字节字段，表示处理器跳转的 Intel x86 机器指令。在大多数情况下，"跳转指令"属于遗留字段。

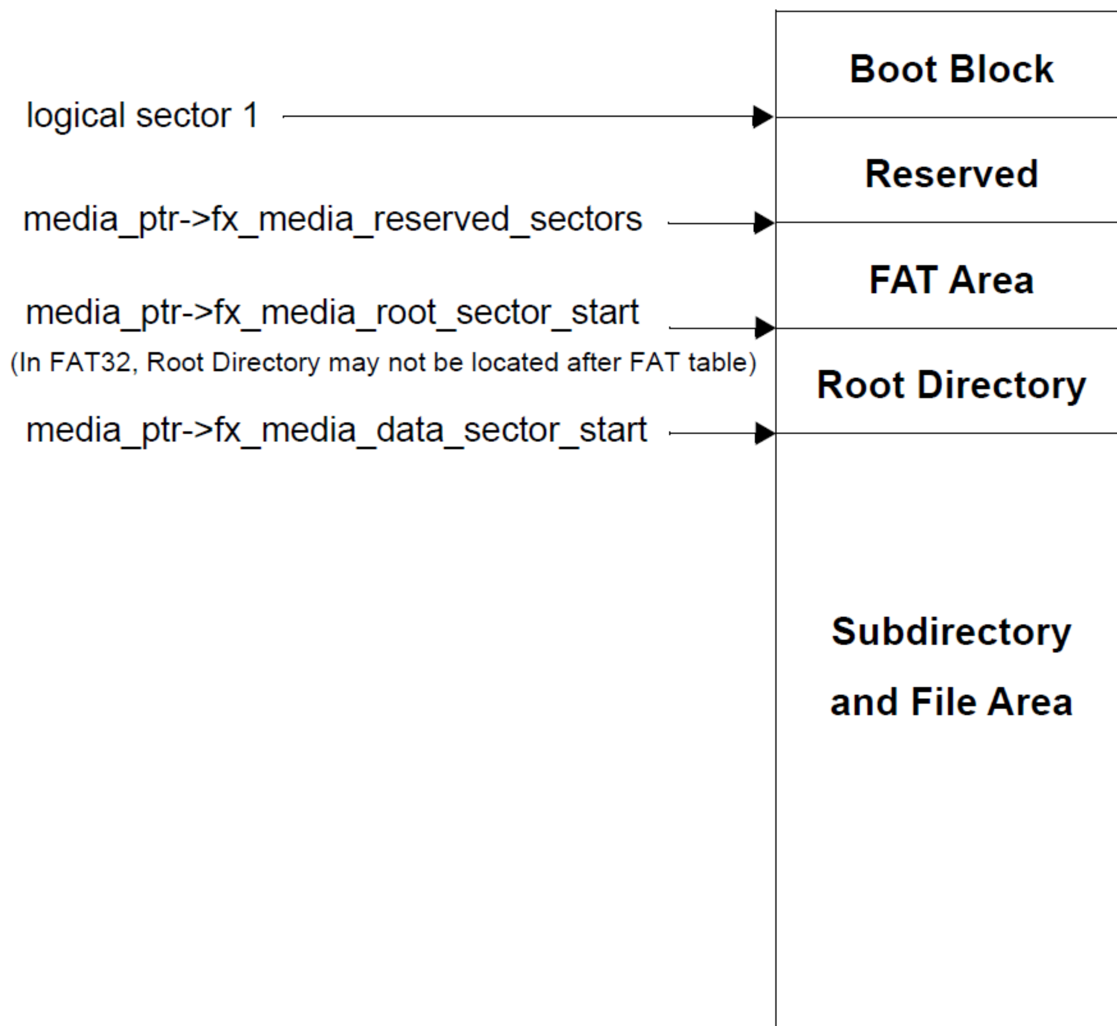

图 1. FileX 介质逻辑扇区视图

- **OEM 名称**“OEM 名称”字段保留供制造商放置他们的名称或设备名称。
- **每扇区字节数** 介质启动记录中的“每扇区字节数”字段定义每个扇区(介质的基本元素)的字节数。
- **每群集扇区数** 介质启动记录中的“每群集扇区数”字段定义分配给每个群集的扇区数。群集是 FAT 兼容文件系统中的基础分配元素。所有文件信息和子目录都是根据文件分配表 (FAT)，从介质的可用群集中进行分配。

表 1. FileX 介质启动记录

OFFSET	[illegible]	[illegible]
0x00	跳转指令(e9、xx、xx 或 eb、xx、90)	3
0x03	OEM 名称	8
0x0B	每扇区字节数	2
0x0D	每群集扇区数	1
0x0E	保留扇区数	2
0x10	FAT 数	1

OFFSET	[illegible]	[illegible]
0x11	根目录大小	2
0x13	FAT-12 和 FAT-16 扇区数	2
0x15	介质类型	1
0x16	每 FAT 扇区数	2
0x18	每磁道扇区数	2
0x1A	磁头数	2
0x1C	隐藏扇区数	4
0x20	扇区数 - FAT-32	4
0x24	每 FAT (FAT-32) 扇区数	4
0x2C	根目录群集	4
0x3E	系统启动代码	448
0x1FE	0x55AA	2

- **保留扇区数** 介质启动记录中的“保留扇区数”字段定义在启动记录和 FAT 区域第一个扇区之间保留的扇区数。在大多数情况下，此条目为零。
- **FAT 数** 介质启动记录中的“FAT 数”条目定义介质中的 FAT 数。介质中必须始终至少有一个 FAT。其他 FAT 只是主要(第一个)FAT 的副本，通常由诊断或恢复软件使用。
- **根目录大小** 介质启动记录中的“根目录大小”条目定义介质根目录中的固定条目数。此字段不适用于子目录和 FAT-32 根目录，因为它们从介质的群集中分配而来。
- **FAT-12 和 FAT-16 扇区数** 介质启动记录中的“扇区数”字段包含介质中的扇区总数。如果此字段为零，则扇区总数将包含在位于启动记录之后的“FAT-32 扇区数”字段中。
- **介质类型**“介质类型”字段用于标识设备驱动程序中的介质类型。这属于遗留字段。
- **每 FAT 扇区数** 介质启动记录中的“每 FAT 扇区数”字段包含与 FAT 区域中每个 FAT 关联的扇区数。FAT 扇区数必须足够大，以满足介质中可以分配的最大群集数。
- **每磁道扇区数** 介质启动记录中的“每磁道扇区数”字段定义每个磁道的扇区数。这通常仅适用于实际磁盘型介质。闪存设备不使用此映射。
- **磁头数** 介质启动记录中的“磁头数”字段定义介质中的磁头数。这通常仅适用于实际磁盘型介质。闪存设备不使用此映射。
- **隐藏扇区数** 介质启动记录中的“隐藏扇区数”字段定义启动记录之前的扇区数。此字段在 FX_MEDIA 控制块中进行维护，并且必须在 FileX 发出的所有读取和写入请求的 FileX I/O 驱动程序中进行说明。
- **FAT-32 扇区数** 仅当双字节“扇区数”字段为零时，介质启动记录中的“扇区数”字段才有效。在这种情况下，此四字节字段包含介质中的扇区数。
- **每 FAT (FAT-32) 扇区数**“每 FAT (FAT-32) 扇区数”字段仅适用于 FAT-32 格式，并包含为介质的每个 FAT 分配的扇区数。

- **根目录群集**“根目录群集”字段仅适用于 FAT-32 格式，并包含根目录的起始群集号。
- **系统启动代码**“系统启动代码”字段是用于存储一小部分启动代码的区域。目前在大多数设备中，“系统启动代码”属于遗留字段。
- **签名 0x55AA**“签名”字段是用于标识启动记录的数据模式。如果此字段不存在，则启动记录无效。

exFAT

FAT32 中的最大文件大小为 4GB，这限制了高清晰多媒体文件的广泛采用。默认情况下，FAT32 支持最大为 32GB 的存储介质。随着闪存和 SD 卡容量的增加，FAT32 在管理大容量文件时效率较低。exFAT 旨在克服这些限制。exFAT 支持的文件大小可高达 1 艾字节 (EB)，约为 10 亿 GB。ExFAT 和 FAT32 的另一个重要区别在于，exFAT 使用位图来管理卷中的可用空间，使得其在向文件中写入数据时能够更有效地查找可用空间。对于存储在连续群集中的文件，exFAT 无需遍历 FAT 链以查找所有群集，从而在访问大型文件时更有效。闪存存储和 SD 卡大于 32GB 时，需要使用 exFAT。

exFAT 逻辑扇区

exFAT 中介质逻辑扇区的一般布局如图 2 所示。在 exFAT 中，启动块和 FAT 区域属于系统区域。其余群集属于用户区域。虽然没有严格限制，但 exFAT 标准确实建议“分配位图”位于“用户区域”的起始位置，其次是“大写转换表”和根目录。

exFAT 介质启动记录

ExFAT 与 FAT12/16/32 拥有不同的“介质启动记录”内容。ExFAT 介质启动记录内容如表 2 所示。为避免混淆，0x0B 和 0x40 之间的区域（其中包含 FAT12/16/32 中的各种介质参数）在 exFAT 中标记为“保留”。此保留区域必须设置为零，以避免任何对“介质启动记录”的误解。

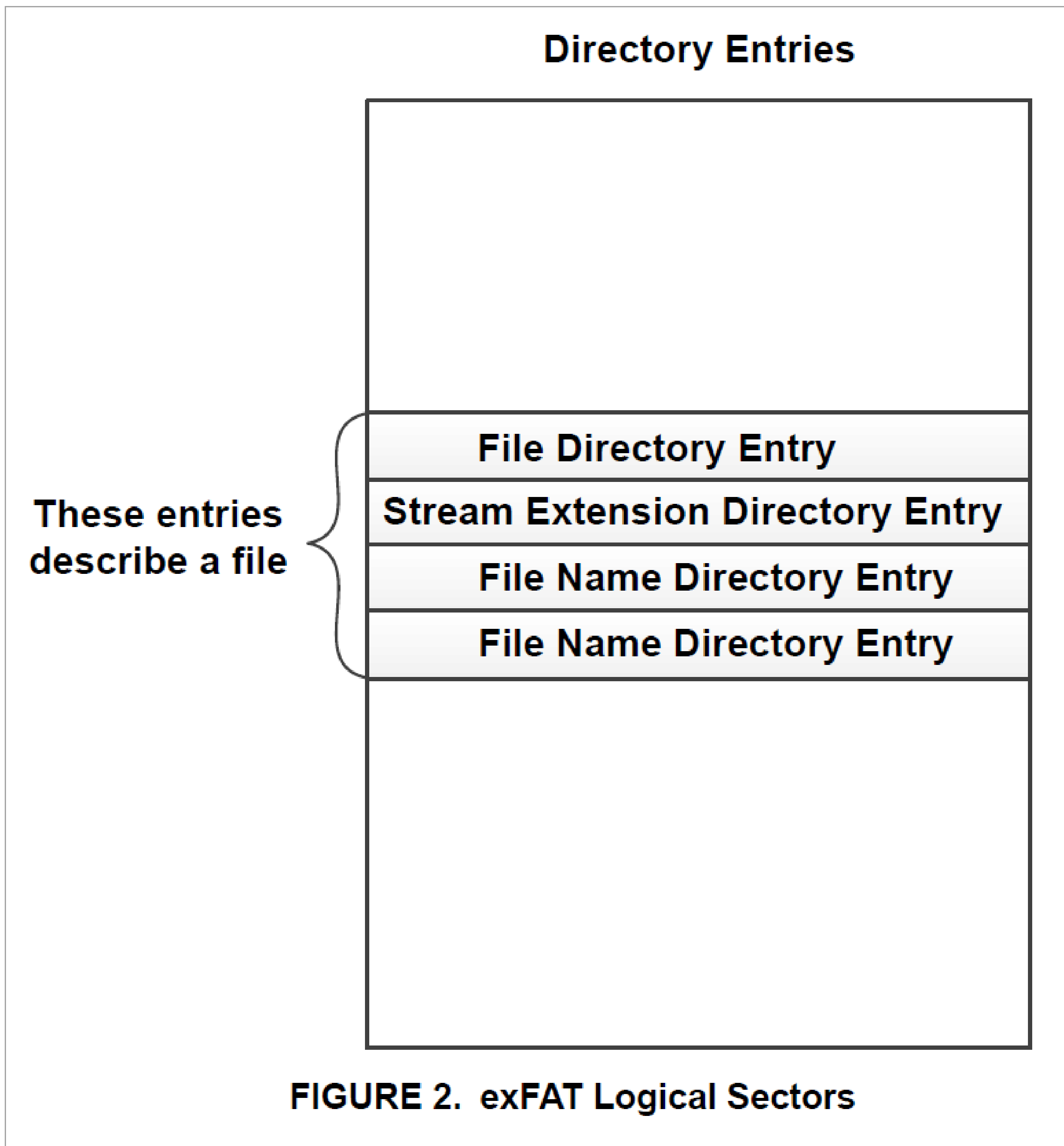

图 2. exFAT 逻辑**扇区**

- **跳转指令**"跳转指令"字段是一个三字节字段，表示处理器跳转的 Intel x86 机器指令。在大多数情况下，"跳转指令"属于遗留字段。

表 2. exFAT **介**质**启**动记录

OFFSET	[[	[[[
0x00	跳转指令	3
0x03	文件系统名称	8
0x0B	保留	53
0x40	分区偏移	8
0x48	卷长度	8

OFFSET	ɪɪ	ɪɪɪ
0x50	FAT 偏移	4
0x54	FAT 长度	4
0x58	群集堆偏移	4
0x5C	群集计数	4
0x60	根目录的第一个群集	4
0x64	卷序列号	4
0x68	文件系统修订	2
0x6A	卷标志	2
0x6C	每扇区字节数移位	1
0x6D	每群集扇区数移位	1
0x6E	FAT 数	1
0x6F	驱动器选择	1
0x70	使用百分比	7
0x71	保留	1
0x78	启动代码	390
0x1FE	启动签名	2

- **文件系统名称** 对于 exFAT，“文件系统名称”字段必须是“exFAT”，后跟三个尾随空格。
- **保留**“保留”字段的内容必须为零。此区域与 FAT12/16/32 中的启动记录重叠。将此区域设置为零可避免文件系统对该卷的错误解释。
- **分区偏移**“分区偏移”字段表示此分区的起始位置。
- **卷长度**“卷长度”字段定义此分区的大小，以扇区数表示。
- **FAT 偏移**“FAT 偏移”字段定义此分区 FAT 表的起始扇区号（相对于此分区的起始位置）。
- **FAT 长度**“FAT 长度”字段定义 FAT 表的大小，以扇区数表示。
- **群集堆偏移**“群集堆偏移”字段定义群集堆的起始扇区号（相对于此分区的起始位置）。群集堆是存储目录信息和文件数据的区域。
- **群集计数**“群集计数”字段定义此分区的群集数。
- **根目录的第一个群集**“根目录的第一个群集”字段定义根目录的起始位置，建议位于分配位图和大写转换表之后。
- **卷序列号**“卷序列号”字段定义此分区的序列号。

- **文件系统修订**“文件系统修订”字段定义 exFAT 的主版本和次版本。
- **卷标志**“卷标志”字段包含指示此卷状态的标志。
- **每扇区字节数移位**“每扇区字节数移位”字段定义每个扇区的字节数，以 log2(n) 表示，其中 n 为每个扇区的字节数。例如，在 SD 卡中，扇区大小为 512 字节。因此，此字段应为 9 (log2(512) = 9)。
- **每群集扇区数移位**“每群集扇区数移位”字段定义每个群集的扇区数，以 log2(n) 表示，其中 n 为每个群集的扇区数。
- **FAT 数**“FAT 数”字段定义此分区中 FAT 表的数量。对于 exFAT，建议此值为 1，即一个 FAT 表。
- **驱动器选择**“驱动器选择”字段定义扩展的 INT 13h 驱动器号。
- **使用百分比**“使用百分比”字段定义群集堆中正在分配的群集的百分比。有效值介于 0 和 100（含）之间。
- **保留** 此字段保留供将来使用。
- **启动代码**“启动代码”字段是用于存储一小部分启动代码的区域。目前在大多数设备中，“系统启动代码”属于遗留字段。
- **签名 0x55AA**“启动签名”字段是用于标识启动记录的数据模式。如果此字段不存在，则启动记录无效。

文件分配表 (FAT)

“文件分配表 (FAT)”在介质中的保留扇区之后启动。FAT 区域本质上是由 12 位、16 位或 32 位条目组成的数组，用于确定该群集是分配而来，还是属于构成子目录或文件的群集链。每个 FAT 条目的大小取决于需要表示的群集数。如果群集数（由扇区总数除以每群集扇区数得出）小于或等于 4,086，则使用 12 位 FAT 条目。如果群集总数大于 4,086 但小于 65,525，则使用 16 位 FAT 条目。否则，如果群集总数大于或等于 65,525，则使用 32 位 FAT 或 exFAT。

对于 FAT12/16/32，FAT 表不仅维护群集链，还提供有关群集分配的信息，即群集是否可用。在 exFAT 中，群集分配信息由“分配位图目录条目”进行维护。每个分区都有其自己的分配位图。位图的大小足以容纳所有可用的群集。如果某个群集可用于分配，则分配位图中的对应位将设置为 0。否则，该对应位将设置为 1。对于占用连续群集的文件，exFAT 无需使用 FAT 链跟踪所有群集。但是，对于不占用连续群集的文件，仍需要维护 FAT 链。

FAT 条目内容

FAT 表中的前两个条目并不使用，通常包含以下内容。

FAT [illegible]	12 [illegible] FAT	16 [illegible] FAT	32 [illegible] FAT	EXFAT
条目 0	0x0F0	0x00F0	0x000000F0	0xF8FFFFFF
条目 1	0xFFF	0xFFFF	0x0FFFFFFF	0xFFFFFFFF

FAT 条目编号 2 表示介质数据区域中的第一个群集。每个群集条目的内容可决定该群集是可用，还是属于分配给文件或子目录的群集链接列表。如果群集条目包含另一个有效的群集条目，则将分配该群集，且其值指向群集链中分配的下一个群集。

可能的群集条目定义如下。

[illegible]	12 [illegible] FAT	16 [illegible] FAT	32 [illegible] FAT	EXFAT
可用群集	0x000	0x0000	0x00000000	0x00000000
不使用	0x001	0x0001	0x00000001	0x00000001

[illegible]	12 [illegible] FAT	16 [illegible] FAT	32 [illegible] FAT	EXFAT
保留	0xFF0-FF6	0xFFF0-FFF6	0x0FFFFFF0-6	ClusterCounter + 2 至 0xFFFFFFF6
不良群集	0xFF7	0xFFF7	0x0FFFFFF7	0xFFFFFFF7
保留	-	-	-	0xFFFFFFF8-E
最后一个群集	0xFF8-FFF	0xFFF8-FFFF	0x0FFFFFF8-F	0xFFFFFFFF
群集链接	0x002-0xFEF	0x0002-FFEF	0x2-0x0FFFFFEF	0x2 - ClusterCount + 1

已分配的群集链中的最后一个群集包含“最后一个群集”值（如上定义）。在文件或子目录的目录条目中找到第一个群集号。

内部逻辑缓存

FileX 为每个打开的介质维护最新使用的逻辑扇区缓存。逻辑扇区缓存的最大大小由常量 FX_MAX_SECTOR_CACHE 定义，并位于 fx_api.h 中。这是决定内部逻辑扇区缓存大小的第一个因素。

决定逻辑扇区缓存大小的另一个因素是由应用程序提供给 fx_media_open*_ 调用的内存量。需要有足够的内存，供至少一个逻辑扇区使用。如果需要 FX_MAX_SECTOR_CACHE_* 个以上逻辑扇区，则必须在 fx_api.h 中更改常量，并且必须重新生成整个 FileX 库。

> **IMPORTANT**
>
> FileX 中每个打开的介质可能具有不同的缓存大小，具体取决于打开调用期间提供的内存。

写入保护

借助 FileX，应用程序驱动程序能够在介质上动态设置写入保护。如果需要写入保护，则驱动程序会将关联的 FX_MEDIA 结构中的 fx_media_driver_write_protect 字段设置为 FX_TRUE。当如是设置后，则应用程序修改介质以及打开文件进行写入的所有尝试均会遭到拒绝。驱动程序还可以通过清除此字段来禁用写入保护。

可用扇区更新

FileX 提供一种机制，用于在扇区不再使用时通知应用程序驱动程序。这对于管理 FileX 使用的所有逻辑扇区的闪存管理器特别有用。

如果需要可用扇区通知，则应用程序驱动程序会将关联的 FX_MEDIA 结构中的 fx_media_driver_free_sector_update 字段设置为 FX_TRUE。此分配通常在驱动程序初始化期间完成。

设置此字段后，FileX 会发出 FX_DRIVER_RELEASE_SECTORS 驱动程序调用，指示一个或多个连续扇区何时可用。

介质控制块 FX_MEDIA

介质控制块中包含 FileX 中每个打开的介质的特征。fx_api.h 文件中定义了此结构。

介质控制块可以位于内存中的任意位置，但最常见的做法是在任何函数的作用域外部定义控制块，以使其成为全局结构。

如同所有动态分配内存一样，将控制块放置于其他区域时需要多加小心。如果在 C 函数内分配控制块，则与之相关联的内存是调用线程堆栈的一部分。

WARNING

通常，请避免对控制块使用本地存储，因为在函数返回后，将释放其所有局部变量堆栈空间，而不管该空间是否仍在使用！

FAT12/16/32 □□□□

FileX 支持 8.3 和 Windows 长文件名 (LFN) 名称格式。除了名称之外，每个目录条目还包含条目的属性、上次修改时间和日期、起始群集索引和条目大小（以字节为单位）。FileX 8.3 目录条目的内容和大小如表 3 所示。

- **目录名称**

  FileX 支持长度为 1 到 255 个字符的文件名。介质上单个目录条目中文件名为标准八个字符。这些文件名在“目录名称”字段中保持左对齐，并以空白填充。此外，包含名称的 ASCII 字符始终大写。

  “长文件名”(LFN) 由连续的目录条目以相反顺序表示，后跟 8.3 标准文件名。创建的 8.3 名称包含与该名称关联的所有有意义的目录信息。用于存放“长文件名”信息的目录条目的内容如表 4 所示，表 5 则显示了一个 39 字符 LFN 的示例，该 LFN 总共需要四个目录条目。

IMPORTANT

fx_api.h 中定义的常量 FX_MAX_LONG_NAME_LEN 包含 FileX 支持的最大长度。

- **目录文件扩展名**

  对于标准 8.3 文件名，FileX 还支持可选的三字符目录文件扩展名。如同八个字符的文件名一样，文件扩展名在“目录文件扩展名”字段中保持左对齐，以空白填充，并且始终大写。

表 3. FileX 8.3 目录条目

OFFSET	□□	□□□
0x00	目录条目名称	8
0x08	目录扩展	3
0x0B	属性	1
0x0C	NT（由长文件名格式引入，并为 NT 保留 [始终为 0]）	1
0x0D	以毫秒为单位的创建时间（由长文件名格式引入，表示创建文件时的毫秒数。）	1
0x0E	以小时和分钟为单位的创建时间（由长文件名格式引入，表示创建文件的小时和分钟）	2
0x10	创建日期（由长文件名格式引入，表示创建文件的日期。）	2
0x12	上次访问日期（由长文件名格式引入，表示上次访问文件的日期。）	2
0x14	起始群集（仅限高 16 位 FAT-32）	2

OFFSET	ɪɪ	ɪɪɪ
0x16	修改时间	2
0x18	修改日期	2
0x1A	起始群集（低 16 位 FAT-32 或 FAT-12 或 FAT-16）	2
0x1C	文件大小	4

- **目录属性**

  单字节“目录属性”字段条目包含一系列位，用于指定目录条目的各种属性。目录属性定义如下：

ɪɪɪ	ɪɪ
0x01	此为只读条目。
0x02	此为隐藏条目。
0x04	此为系统条目。
0x08	此为卷标签条目
0x10	此为目录条目。
0x20	此条目已修改。

  由于所有属性位均互斥，因此一次可能会设置多个属性位。

- **目录时间**

  双字节“目录时间”字段包含对指定目录条目最后一次更改的小时、分钟和秒。第 15 位到第 11 位为小时，第 10 位到第 5 位为分钟，而第 4 位到第 0 位则为秒数的一半。实际秒数需除以 2 才能写入此字段。

- **目录日期**

  双字节“目录日期”字段包含对指定目录条目最后一次更改的年份（相对于 1980 年的偏移）、月份和日期。第 15 位到第 9 位为年份偏移，第 8 位到第 5 位为月份偏移，而第 4 位到第 0 位为日期。

- **目录起始群集**

  对于 FAT-12 和 FAT-16，此字段占用 2 个字节。对于 FAT-32，此字段占用 4 个字节。此字段包含分配给条目（子目录或文件）的第一个群集号。

> **NOTE**
>
> *请注意，FileX 创建新文件时没有初始群集（“起始群集”字段为零），以便用户有选择地为新创建的文件分配连续数量的群集。*

- **目录文件大小**

  四字节“目录文件大小” 字段包含文件中的字节数。如果该条目确实为子目录，则该“大小”字段为零。

长文件名目录

- **序号**

单字节"序号"字段指定 LFN 条目编号。由于 LFN 条目以相反顺序放置，因此，由单个 LFN 组成的 LFN 目录条目的序号值会减 1。此外，8.3 文件名之前的 LFN 序号值必须为 1。

表 4. 长文件名目录条目

OFFSET	[illegible]	[illegible]
0x00	序号字段	1
0x01	Unicode 字符 1	2
0x03	Unicode 字符 2	2
0x05	Unicode 字符 3	2
0x07	Unicode 字符 4	2
0x09	Unicode 字符 5	2
0x0B	LFN 属性	1
0x0C	LFN 类型（保留，始终为 0）	1
0x0D	LFN 校验和	1
0x0E	Unicode 字符 6	2
0x10	Unicode 字符 7	2
0x12	Unicode 字符 8	2
0x14	Unicode 字符 9	2
0x16	Unicode 字符 10	2
0x18	Unicode 字符 11	2
0x1A	LFN 群集（未使用，始终为 0）	2
0x1C	Unicode 字符 12	2
0x1E	Unicode 字符 13	2

- **Unicode 字符**

双字节"Unicode 字符"字段旨在支持多种不同语言的字符。标准 ASCII 字符的表达方式为存储在 Unicode 字符的第一个字节中的 ASCII 字符后跟一个空格字符。

- **LFN 属性**

单字节"LFN 属性"字段包含的属性将目录条目标识为 LFN 目录条目。这可通过设置"只读"、"系统"、"隐藏"和"卷"属性来实现。

- **LFN 类型**

  单字节“LFN 类型”字段是保留字段，始终为 0。

- **LFN 校验和**

  单字节“LFN 校验和”字段表示关联的 MSDOS 8.3 文件名的 11 个字符的校验和。此校验和存储在每个 LFN 条目中，以帮助确保 LFN 条目与适当的的 8.3 文件名相对应。

- **LFN 群集**

  双字节“LFN 群集”字段未使用，始终为 0。

表 5. 由 39 个字符 LFN 组成的目录条目

[illegible]	[illegible]
1	LFN 目录条目 3
2	LFN 目录条目 2
3	LFN 目录条目 1
4	8.3 目录条目 (ttttt~n.xx)

exFAT 目录说明

exFAT 文件系统以不同的方式存储目录条目和文件名。目录条目包含条目的属性以及创建、修改和访问条目时的各种时间戳。其他信息（如文件大小和起始群集）存储在紧随主目录条目之后的“流扩展目录条目”中。exFAT 仅支持“长文件名”(LFN) 名称格式。exFAT 存储在紧随“流扩展目录条目”之后的“文件名目录条目”中，如表 2 所示。

exFAT 文件目录条目

关于 exFAT 文件目录条目及其内容的说明，见下表和后文段落。

- **条目类型**

  “条目类型”字段表示此条目的类型。对于“文件目录条目”，此字段必须为 0x85。

- **辅助条目计数**

  “辅助条目计数”字段表示紧随主要条目之后的辅助条目数。与文件目录条目关联的辅助条目包括一个流扩展目录条目和一个或多个文件名目录条目。

表 6. exFAT 文件目录条目

OFFSET	[illegible]	[illegible]
0x00	条目类型	1
0x01	辅助条目	1
0x02	校验和	2
0x04	文件属性	2
0x06	保留 1	2
0x08	创建时间戳	4

OFFSET	[illegible]	[illegible]
0x0C	上次修改时间戳	4
0x10	上次访问时间戳	4
0x14	创建 10ms 增量	1
0x15	上次修改 10ms 增量	1
0x16	创建 UTC 偏移	1
0x17	上次修改 UTC 偏移	1
0x18	上次访问 UTC 偏移	1
0x19	保留 2	7

- **校验和**

  “校验和”字段包含目录条目集中所有条目（文件目录条目及其辅助条目）的校验和值。

- **文件属性**

  单字节“属性”字段条目包含一系列位，用于指定目录条目的各种属性。大多数属性位的定义与 FAT 12/16/32 相同。目录属性定义如下：

[illegible]	[illegible]
0x01	此为只读条目
0x02	此为隐藏条目
0x04	此为系统条目
0x08	此为保留条目
0x10	此为目录条目
0x20	此条目已修改
所有其他位	保留

- Reserved1

  此字段应为零。

- **创建时间戳**

  “创建时间戳”字段，结合“创建 10ms 增量”字段中的信息，描述创建文件或目录的本地日期和时间。

- **上次修改时间戳**

  “上次修改时间戳”字段，结合“上次修改 10ms 增量”字段中的信息，描述上次修改文件或目录的本地日期和时间。请参阅下文时间戳注意事项。

- **上次访问时间戳**

  “上次访问时间戳”字段描述上次访问文件或目录的本地日期和时间。请参阅下文时间戳注意事项。

- **创建 10ms 增量**

  “创建 10ms 增量”字段，结合“创建时间戳”字段中的信息，描述创建文件或目录的本地日期和时间。请参阅下文时间戳注意事项。

- **上次修改 10ms 增量**

  “上次修改 10ms 增量”字段，结合“上次修改时间戳”字段中的信息，描述上次修改文件或目录的本地日期和时间。请参阅下文时间戳注意事项。

- **创建 UTC 偏移**

  “创建 UTC 偏移”字段描述创建文件或目录时本地时间和 UTC 时间之间的差异。请参阅下文时间戳注意事项。

- **上次修改 UTC 偏移**

  “上次修改 UTC 偏移”字段描述上次修改文件或目录时本地时间和 UTC 时间之间的差异。请参阅下文时间戳注意事项。

- **上次访问 UTC 偏移**

  “上次访问 UTC 偏移”字段描述上次访问文件或目录时本地时间和 UTC 时间之间的差异。请参阅下文时间戳注意事项。

- **Reserved2**

  此字段应为零。

时间戳注意事项

- **时间戳条目**“时间戳”字段解释如下：
- **10ms 增量字段**“10ms 增量”字段中的值为时间戳值提供了更精细的粒度。有效值介于 0 (0ms) 和 199 (1990ms) 之间。

31	25	21	16	11	5	0
Year (Relative to year 1980)	Month: 1-12	Day: 1-31	Hour: 0-23	Minute: 0-59	Seconds divide by 2 (0-29)	

- **UTC 偏移字段**

7	6	0
Valid	Offset Value	

- **偏移值**

  7 位带符号整数表示相对于 UTC 时间的偏移（以 15 分钟为增量）。

- **有效**

  表示“偏移值”字段中的值是否有效。0 表示“偏移值”字段中的值无效。1 表示“偏移值”字段中的值有效。

流扩展目录条目

关于“流扩展目录条目”及其内容的说明如下表所示。

表 7. 流扩展目录条目

OFFSET	[illegible]	[illegible]
0x00	条目类型	1
0x01	Flags	1
0x02	保留 1	1
0x03	名称长度	1
0x04	名称哈希	2
0x06	保留 2	2
0x08	有效数据长度	8
0x10	保留 3	4
0x14	第一个群集	4
0x18	数据长度	8

- **条目类型**

  “条目类型”字段表示此条目的类型。对于流式处理扩展目录条目，此字段必须为 0xC0。

- 标记

  此字段包含一系列指定各种属性的位：

[illegible]	[illegible]
0x01	此字段表示是否可以分配群集。此字段应为 1。
0x02	此字段表示关联的群集是否连续。值 0 表示 FAT 条目有效，FileX 应遵循 FAT 链。值 1 表示 FAT 条目无效，群集具有连续性。
所有其他位	保留。

- **保留 1**

  此字段应为 0。

- **名称长度**

  “名称长度”字段包含文件名目录条目中共同包含的 Unicode 字符串的长度。文件名目录条目应紧跟在此流扩展目录条目之后。

- **名称哈希**

  “名称哈希”字段是双字节条目，包含大写文件名的哈希值。利用哈希值，可更快地查找文件/目录名称：如果哈希值不匹配，则与此条目关联的文件名亦不匹配。

- **保留 2**

此字段应为 0。

- **有效数据长度**

  “有效数据长度”字段表示文件中的有效数据量。

- **保留 3**

  此字段应为 0。

- **第一个群集**

  “第一个群集”字段包含数据流的第一个群集的索引。

- **数据长度**

  “数据长度”字段包含已分配群集中的字节总数。此值可能大于“有效数据长度”，因为 exFAT 允许预分配数据群集。

根目录

在 FAT 12 位和 16 位格式中，根目录位于介质中所有 FAT 扇区之后，可以通过检查打开的 FX_MEDIA ** 控制块中的 fx_media_root_sector_start** 进行查找。根目录的大小以目录条目数（每 32 个字节的大小）表示，取决于介质启动记录中的相应条目。

FAT-32 和 exFAT 中的根目录可位于可用群集中的任意位置。其位置和大小取决于打开介质时的启动记录。打开介质后，可以使用 fx_media_root_sector_start 字段查找 FAT-32 或 exFAT 根目录的起始群集。

子目录

FAT 系统中有任意数量的子目录。子目录的名称与文件名一样位于目录条目中。但是，目录属性规范 (0x10) 设置为表示该条目为子目录，且文件大小始终为零。全新单群集子目录（带有一个名为 FILE.TXT_*** 文件的 SAMPLE.DIR*_）的典型子目录结构如图 3 所示。在大多数情况下，子目录与文件条目非常相似。“第一个群集”字段指向群集链接列表的第一个群集。创建子目录时，前两个目录条目包含默认目录，即 "." 目录和 ".." 目录。"." 目录指向子目录本身，而 ".." 目录指向上一个或父目录。

全局默认路径

FileX 为介质提供全局默认路径。默认路径用于任何未明确指定完整路径的文件或目录服务。

最初，全局默认目录设置为介质的根目录。应用程序可以通过调用 fx_directory_default_set 对全局默认目录进行更改。

可以通过调用 fx_directory_default_get*_ 来检查介质的当前默认路径。此例程提供一个字符串指针，该指针指向在 FX_MEDIA* 控制块内进行维护的默认路径字符串。

本地默认路径

FileX 还提供了针对线程的默认路径，该路径允许不同的线程具有唯一路径，而不会发生冲突。应用程序在调用 fx_directory_local_path_set _ ***和 fx_directory_local_path_restore***_ 期间提供 FX_LOCAL_PATH 结构，用于修改调用线程的本地路径。

如果存在本地路径，则本地路径优先于全局默认介质路径。如果未设置本地路径，或者使用 fx_directory_local_path_clear 服务清除了本地路径，则会再次使用介质的全局默认路径。

文件描述

FileX 支持标准 8.3 字符和带有三个字符扩展名的长文件名。除了 ASCII 名称之外，每个文件条目还包含条目的属性、上次修改时间和日期、起始群集索引和条目大小（以字节为单位）。

文件分配

FileX 支持 FAT 格式的标准群集分配方案。此外，FileX 还支持对连续群集进行预分配。为此，创建每个 FileX 文件时均未分配群集。可应后续写入请求或 fx_file_allocate 请求分配群集，以预分配连续群集。

图 4“FileX FAT-16 文件示例”显示了一个名为 FILE.TXT 的文件，向该文件分配了两个连续群集，始于群集 101，大小为 26，且字母表为该文件第一个数据群集号 101 中的数据。

文件访问

可以同时打开多次 FileX 文件以进行读取访问。但是，写入时，文件只能打开一次。FX_FILE 文件控制块中包含用于支持文件访问的信息。

> **NOTE**
>
> 请注意，介质驱动程序可以动态设置写入保护。设置后，所有写入请求以及打开文件进行写入的尝试均会遭到拒绝。

ExFAT 中的文件布局

如果数据存储在连续群集中，则 exFAT 的设计无需为文件维护 FAT 链。“流扩展目录条目”中的 NoFATChain 位表示从文件中读取数据时是否应使用 FAT 链。如果设置了 NoFATChain，则 FileX 会从“流扩展目录条目”的“第一个群集”字段指示的群集中依次进行读取。

另一方面，如果清除 NoFATChain，则 FileX 将遵循 FAT 链以遍历整个文件，这与 FAT12/16/32 中的 FAT 链类似。

图 3 显示了两个示例文件，一个无需 FAT 链，另一个则需要 FAT 链。

系统信息

FileX 系统信息包含跟踪打开的介质实例并维护全局系统时间和日期。

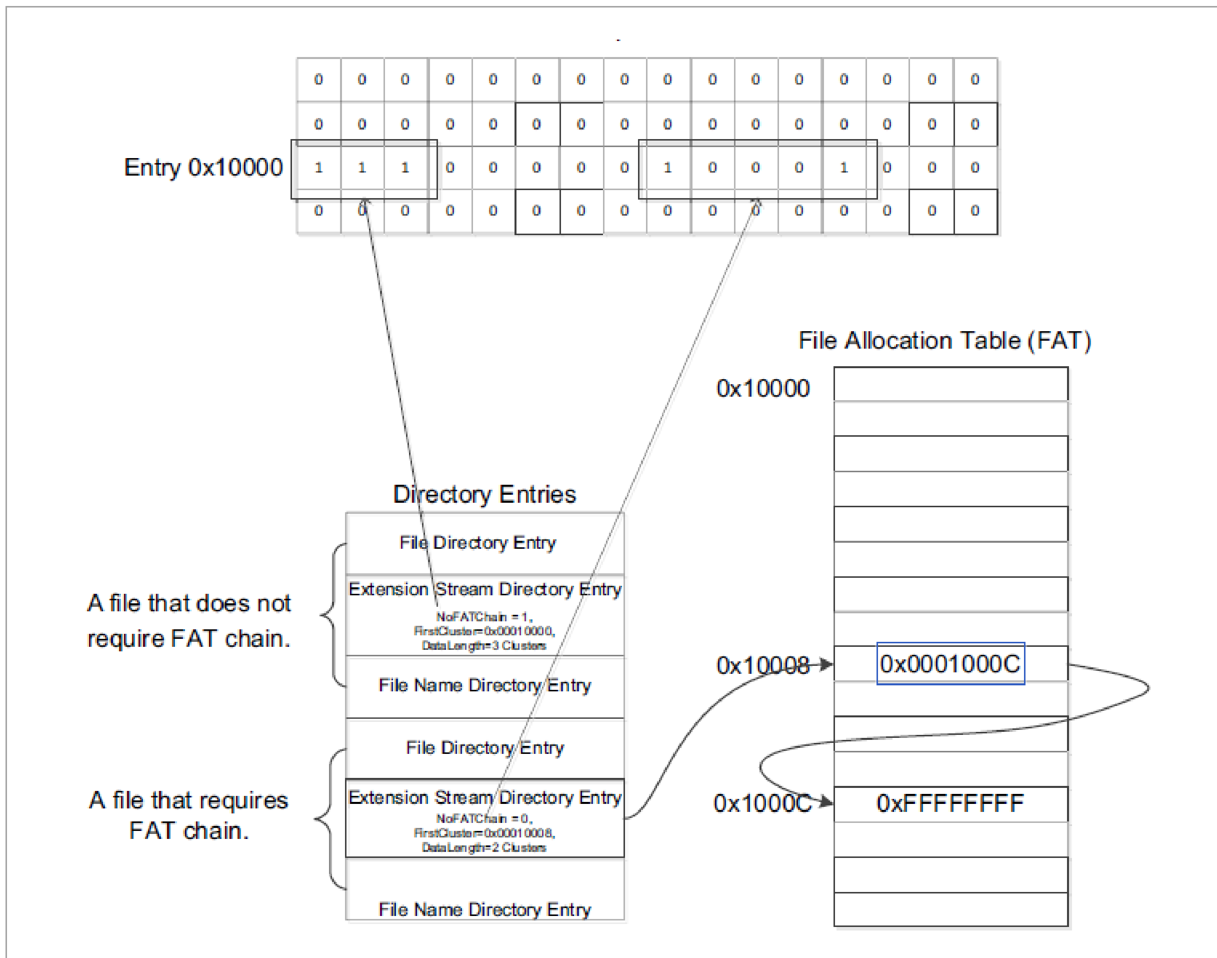

图 3. 具有连续群集的文件 vs. 需要 FAT 链接的文件

默认情况下，系统日期和时间设置为 FileX 的上次发布日期。若要获得准确的系统日期和时间，应用程序必须在初始化期间调用 fx_system_time_set*_ 和 fx_system_date_set _*。

系统日期

FileX 系统日期在全局 _fx_system_date 变量中进行维护。第 15 位到第 9 位为相对于 1980 年的年份偏移，第 8 位到第 5 位为月份偏移，而第 4 位到第 0 位为日期。|

系统时间

FileX 系统时间在全局 _fx_system_time 变量中进行维护。第 15 位到第 11 位为小时，第 10 位到第 5 位为分钟，而第 4 位到第 0 位则为秒数的一半。

定期时间更新

在系统初始化过程中，FileX 会创建一个 ThreadX 应用程序计时器，以便定期更新系统日期和时间。系统日期和时间更新速度取决于 _fx_system_initialize 函数使用的两个常量。

常量 FX_UPDATE_RATE_IN_SECONDS 和 FX_UPDATE_RATE_IN_TICKS 表示相同时间段。常量 FX_UPDATE_RATE_IN_TICKS 是表示秒数（由常量 FX_UPDATE_RATE_IN_SECONDS 指定）的 ThreadX 计时器时钟周期数。FX_UPDATE_RATE_IN_SECONDS 常量指定每次 FileX 时间更新之间的秒数。因此，内部 FileX 时间以 FX_UPDATE_RATE_IN_SECONDS 的间隔递增。在 fx_system_initialize_ *编译过程中，可能会提供这些常量，或者开发人员可能修改在 FileX 版本的 fx_port.h_* 文件中找到的默认值。

定期 FileX 计时器仅用于更新全局系统日期和时间，而该日期和时间仅用于文件时间戳。如果无需时间戳，只需在编译 fx_system_initialize.c 时定义 FX_NO_TIMER，以避免创建 FileX 定期计时器。

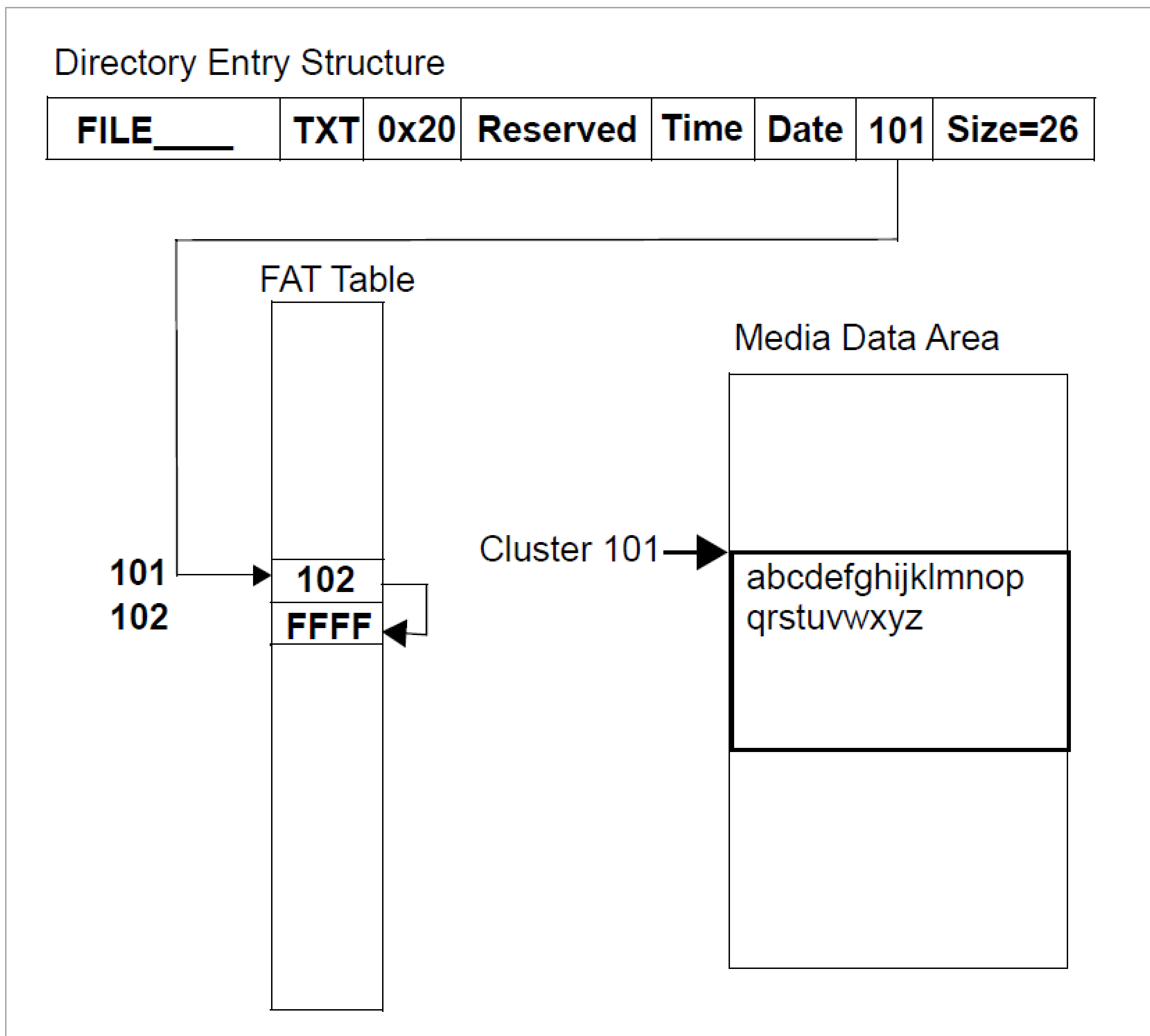

图 4. FileX FAT-16 文件示例

第 4 章 - Azure RTOS FileX 服务说明

2021/4/29 •

本章按字母顺序介绍所有 Azure RTOS FileX 服务。服务名称设计为将所有相似的服务组合在一起。

fx_directory_attributes_read

读取目录属性

原型

```
UINT fx_directory_attributes_read (
    FX_MEDIA *media_ptr,
    CHAR *directory_name,
    UINT *attributes_ptr);
```

说明

此服务从指定的媒体读取目录的属性。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **directory_name**：指向请求的目录名称的指针（目录路径是可选的）。
- **attributes** _ptr：指向目录属性要放置的目标的指针。目录属性以位图格式返回，具有以下可能的设置：
  - FX_READ_ONLY (0x01)
  - FX_HIDDEN (0x02)
  - FX_SYSTEM (0x04)
  - FX_VOLUME (0x08)
  - FX_DIRECTORY (0x10)
  - FX_ARCHIVE (0x20)

返回值

- **FX_SUCCESS** (0x00) 成功读取目录属性
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开
- **FX _NOT FOUND** (0x04) 在媒体中未找到指定的目录
- **FX_NOT_DIRECTORY** (0x0E) 条目不是目录
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误
- **FX_FILE_CORRUPT** (0x08) 文件已损坏
- **FX_SECTOR_INVALID** (0x89) 扇区无效
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_MEDIA_INVALID** (0x02) 媒体无效
- **FX_PTR_ERROR** (0x18) 媒体指针无效
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA     my_media;
UINT     status;
/* Retrieve the attributes of "mydir" from the specified media.*/
status = fx_directory_attributes_read(&my_media, "mydir", &attributes);
/* If status equals FX_SUCCESS, "attributes" contains the directory attributes of "mydir". */
```

另请参阅

- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_attributes_set

设置目录属性

原型

```
UINT fx_directory_attributes_set(
    FX_MEDIA *media_ptr,
    CHAR *directory_name,
    UINT *attributes);
```

说明

此服务将目录的属性设置为调用方指定的属性。

> **WARNING**
>
> 仅允许此应用程序使用此服务修改目录属性的子集。任何设置其他属性的尝试都将导致错误。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **directory_name**：指向请求的目录名称的指针（目录路径是可选的）。
- **attributes**：此目录的新属性。有效目录属性定义如下：

- FX_READ_ONLY (0x01)
    - FX_HIDDEN (0x02)
    - FX_SYSTEM (0x04)
    - FX_ARCHIVE (0x20)

返回值

- **FX_SUCCESS** (0x00) 成功设置目录属性
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开
- **FX_NOT_FOUND** (0x04) 在媒体中未找到指定的目录
- **FX_NOT_DIRECTORY** (0x0E) 条目不是目录
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护
- **FX_FILE_CORRUPT** (0x08) 文件已损坏
- **FX_SECTOR_INVALID** (0x89) 扇区无效
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_MEDIA_INVALID** (0x02) 媒体无效
- **FX_NO_MORE_ENTRIES** (0x0F) 此目录中不再有条目
- **FX_PTR_ERROR** (0x18) 媒体指针无效
- **FX_INVALID_ATTR** (0x19) 选择了无效属性。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA            my_media;
UINT                status;
status = fx_directory_attributes_set(&my_media, "mydir", FX_READ_ONLY);
/*Set the attributes of "mydir" to read-only. */
/* If status equals FX_SUCCESS, the directory "mydir" is read-only. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find

- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_create

创建子目录

原型

```
UINT fx_directory_create(
    FX_MEDIA *media_ptr,
    CHAR *directory_name);
```

说明

此服务在当前默认目录或目录名称中提供的路径中创建子目录。与根目录不同，子目录不会限制其可容纳的文件数。根目录只能容纳由启动记录确定的条目数。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **directory_name**：指向要创建的目录名称的指针（目录路径是可选的）。

返回值

- **FX_SUCCESS** (0x00) 成功创建目录。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开
- **FX_NOT_FOUND** (0x04) 在媒体中未找到指定的目录
- **FX_NOT_DIRECTORY** (0x0E) 条目不是目录
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误
- **FX_FILE _CORRUPT** (0x08) 文件已损坏
- **FX_SECTOR_INVALID** (0x89) 扇区无效
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_MEDIA_INVALID** (0x02) 媒体无效
- **FX_NO_MORE_ENTRIES** (0x0F) 此目录中不再有条目
- **FX_PTR_ERROR** (0x18) 媒体指针无效
- **FX_INVALID_ATTR** (0x19) 选择了无效属性。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA             my_media;
UINT                 status;
/* Create a subdirectory called "temp" in the current default directory. */

status = fx_directory_create(&my_media, "temp");

/* If status equals FX_SUCCESS, the new subdirectory "temp" has been created. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_default_get

获取上一个默认目录

原型

```
UINT fx_directory_default_get(
    FX_MEDIA *media_ptr,
    CHAR **return_path_name);
```

说明

此服务将指针返回到 fx_directory_default_set 最后设置的路径。如果尚未设置默认目录，或者当前默认目录为根目录，则返回值 FX_NULL。

> **IMPORTANT**
>
> 内部路径字符串的默认大小为 256 个字符；通过修改 fx_api.h 中的 FX_MAXIMUM_PATH 并重新生成整个 FileX 库可以更改大小。字符串路径会保留用于应用程序，不会由 FileX 在内部使用 。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **return_path_name**：指向最后一个默认目录字符串的目标的指针。如果默认目录的当前设置是根目录，则返回值 FX_NULL。媒体打开时，根目录为默认值。

返回值

- **FX_SUCCESS** (0x00) 成功获取默认目录
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开
- **FX_PTR_ERROR** (0x18) 媒体或目标指针无效。

- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA my_media;
CHAR *current_default_dir;
UINT status;
/* Retrieve the current default directory. */
status = fx_directory_default_get(&my_media, &current_default_dir);
/* If status equals FX_SUCCESS, "current_default_dir"
    contains a pointer to the current default directory).*/
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_default_set

设置默认目录

原型

```
UINT fx_directory_default_set(
    FX_MEDIA *media_ptr,
    CHAR *new_path_name);
```

说明

此服务设置媒体的默认目录。如果提供了值 FX_NULL，则默认目录将设置为媒体的根目录。所有未显式指定路径的后续文件操作都将默认为此目录。

> **IMPORTANT**
>
> 内部路径字符串的默认大小为 256 个字符；通过修改 fx_api.h 中的 FX_MAXIMUM_PATH 并重新生成整个 FileX 库可以更改大小。字符串路径会保留用于应用程序，不会由 FileX 在内部使用 。

> **IMPORTANT**
>
> 对于应用程序提供的名称，FileX 支持反斜杠 (\) 和正斜杠字符串来分隔目录、子目录和文件名。但是，FileX 仅在返回到应用程序的路径中使用反斜杠字符。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **new_path_name**：指向新默认目录名称的指针。如果提供了值 FX_NULL，则媒体的默认目录将设置为媒体的根目录。

返回值

- **FX_SUCCESS** (0x00) 成功设置默认目录
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开
- **FX_INVALID_PATH** (0x0D) 找不到新目录
- **FX_PTR_ERROR** (0x18) 媒体指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA     my_media;
UINT status;
/* Set the default directory to \abc\def\ghi. */
status = fx_directory_default_set(&my_media, "\\abc\\def\\ghi");
/* If status equals FX_SUCCESS, the default directory for this media is \abc\def\ghi. All subsequent file
operations that do not explicitly specify a path will default to this directory. Note that the character "\"
serves as an escape character in a string. To represent the character "\", use the construct "\\". This is
done because of the C language- only one "\" is really present in the string. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test

- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_delete

删除子目录

原型

```
UINT fx_directory_delete(
    FX_MEDIA *media_ptr,
    CHAR *directory_name);
```

说明

此服务删除指定的目录。请注意，目录必须为空才能将其删除。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **directory_name**：指向要删除的目录名称的指针（目录路径是可选的）。

返回值

- **FX_SUCCESS** (0x00) 成功创建目录
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开
- **FX_NOT_FOUND** (0x04) 未找到指定的目录
- **FX_DIR_NOT_EMPTY** (0x10) 指定的目录不为空
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护
- **FX_FILE_CORRUPT** (0x08) 文件已损坏
- **FX_SECTOR_INVALID** (0x89) 扇区无效
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_MEDIA_INVALID** (0x02) 媒体无效
- **FX_NO_MORE_ENTRIES** (0x0F) 此目录中不再有条目
- **FX_NOT_DIRECTORY** (0x0E) 不是目录条目
- **FX_PTR_ERROR** (0x18) 媒体指针无效
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA     my_media;
UINT status;
/* Set the default directory to \abc\def\ghi. */
status = fx_directory_delete(&my_media, "abc");
/* Delete the subdirectory "abc." */
/* If status equals FX_SUCCESS, the subdirectory "abc" was deleted. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_first_entry_find

获取第一个目录条目

原型

```
UINT fx_directory_first_entry_find(
    FX_MEDIA *media_ptr,
    CHAR *return_entry_name);
```

说明

此服务检索默认目录中的第一个条目名称，并将其复制到指定目标。

> **WARNING**
>
> 指定的目标必须足够大，才能容纳 FX_MAX_LONG_NAME_LEN 定义的最大大小的 FileX 名称。

> **WARNING**
>
> 如果使用非本地路径，则务必在目录遍历发生时（通过 ThreadX 信号灯、互斥或优先级更改）阻止其他应用程序线程更改此目录。否则，可能获得无效结果。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **return_entry_name**：指向默认目录中第一个条目名称的目标的指针。

返回值

- FX_SUCCESS (0x00) 成功查找第一个目录条目
- FX_MEDIA_NOT_OPEN (0x11) 指定的媒体未打开
- FX_NO_MORE_ENTRIES (0x0F) 此目录中不再有条目
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误
- FX_FILE_CORRUPT (0x08) 文件已损坏
- FX_SECTOR_INVALID (0x89) 扇区无效
- FX_FAT_READ_ERROR (0x03) 无法读取 FAT 条目
- FX_PTR_ERROR (0x18) 媒体或目标指针无效
- FX_CALLER_ERROR (0x20) 调用方不是线程

允许来自

线程数

示例

```
FX_MEDIA         my_media;
UINT             status;
CHAR             entry[FX_MAX_LONG_NAME_LEN];
/* Retrieve the first directory entry in the current directory. */
status = fx_directory_first_entry_find(&my_media, entry);
/* If status equals FX_SUCCESS, the entry in the directory is the "entry" string. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_first_full_entry_find

获取包含完整信息的第一个目录条目

原型

```
UINT fx_directory_first_full_entry_find(
    FX_MEDIA *media_ptr,
    CHAR *directory_name,
    UINT *attributes,
    ULONG *size,
    UINT *year, UINT *month, UINT *day,
    UINT *hour, UINT *minute, UINT *second);
```

输入参数

- media_ptr：指向媒体控制块的指针。
- directory_name：指向目录条目名称的目标的指针。大小必须至少为 FX_MAX_LONG_NAME_LEN。
- attributes：如果非 Null，则为指向条目属性要放置的目标的指针。属性以位图格式返回，具有以下可能的设置：
  - FX_READ_ONLY (0x01)
  - FX_HIDDEN (0x02)
  - FX_SYSTEM (0x04)
  - FX_VOLUME (0x08)
  - FX_DIRECTORY (0x10)
  - FX_ARCHIVE (0x20)
- size：如果非 Null，则为指向条目大小（字节）的目标的指针。
- year：如果非 Null，则为指向条目的修改年份的目标的指针。
- month：如果非 Null，则为指向条目的修改月份的目标的指针。
- day：如果非 Null，则为指向条目的修改日期的目标的指针。
- hour：如果非 Null，则为指向条目的修改时间（小时）的目标的指针。
- minute：如果非 Null，则为指向条目的修改时间（分钟）的目标的指针。
- second：如果非 Null，则为指向条目的修改时间（秒）的目标的指针。

返回值

- FX_SUCCESS (0x00) 成功查找第一个目录条目
- FX_MEDIA_NOT_OPEN (0x11) 指定的媒体未打开
- FX_NO_MORE_ENTRIES (0x0F) 此目录中不再有条目
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误
- FX_WRITE_PROTECT (0x23) 指定的媒体受到写入保护
- FX_FILE _CORRUPT (0x08) 文件已损坏
- FX_SECTOR_INVALID (0x89) 扇区无效
- FX_FAT_READ_ERROR (0x03) 无法读取 FAT 条目
- FX_NO_MORE_SPACE (0x0A) 不再有空间来完成该操作
- FX_MEDIA_INVALID (0x02) 媒体无效
- FX_PTR_ERROR (0x18) 媒体或目标指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA     my_media;
UINT         status;
CHAR         entry_name[FX_MAX_LONG_NAME_LEN];
UINT         attributes;
ULONG        size;
UINT         year;
UINT         month;
UINT         day;
UINT         hour;
UINT         minute;
UINT         second;
/* Get the first directory entry in the default directory with full information. */
status = fx_directory_first_full_entry_find(&my_media, entry_name,
    &attributes, &size, &year, &month, &day, &hour, &minute, &second);

/* If status equals FX_SUCCESS, the entry's information is in the local variables. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_information_get：

获取目录条目信息

原型

```
UINT fx_directory_first_full_entry_find(
    FX_MEDIA *media_ptr,
    CHAR *directory_name,
    UINT *attributes,
    ULONG *size,
    UINT *year, UINT *month, UINT *day,
    UINT *hour, UINT *minute, UINT *second);
```

输入参数

- media_ptr：指向媒体控制块的指针。
- directory_name：指向目录条目名称的指针。
- attributes：指向属性的目标的指针。
- size：指向大小的目标的指针。
- year：指向年份的目标的指针。
- month：指向月份的目标的指针。
- day：指向日期的目标的指针。
- hour：指向时间（小时）的目标的指针。
- minute：指向时间（分钟）的目标的指针。
- second：指向时间（秒）的目标的指针。

返回值

- FX_SUCCESS (0x00) 成功查找第一个目录条目
- FX_MEDIA_NOT_OPEN (0x11) 指定的媒体未打开
- FX_NOT_FOUND (0x04) 在媒体中未找到指定的目录
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误
- FX_MEDIA_INVALID (0x02) 媒体无效
- FX_FILE _CORRUPT (0x08) 文件已损坏
- FX_FAT_READ_ERROR (0x03) 无法读取 FAT 条目
- FX_NO_MORE_SPACE (0x0A) 不再有空间来完成该操作
- FX_SECTOR_INVALID (0x89) 扇区无效
- FX_PTR_ERROR (0x18) 媒体或目标指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA     my_media;
UINT         status; attributes; year; month; day;
CHAR         entry_name[FX_MAX_LONG_NAME_LEN];
ULONG        size;
UINT         hour; minute; second;
/* Retrieve information about the directory entry "myfile.txt".*/
status = fx_directory_information_get(&my_media, "myfile.txt", &attributes, &size,
                                      &year, &month, &day,
                                      &hour, &minute, &second);
/* If status equals FX_SUCCESS, the directory entry information is available in the local variables. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_local_path_clear
- fx_directory_local_path_get

- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_local_path_clear：

清除默认本地路径

原型

```
UINT fx_directory_local_path_clear(FX_MEDIA *media_ptr);
```

说明

此服务清除为调用线程设置的之前的本地路径。

输入参数

- **media_ptr**：指向先前打开的媒体的指针。

返回值

- **FX_SUCCESS** (0x00) 成功清除本地路径。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体当前未打开
- **FX_NOT_IMPLEMENTED** (0x22) 已定义 FX_NO_LCOAL_PATH
- **FX_PTR_ERROR** (0x18) 媒体指针无效

允许来自

线程数

示例

```
FX_MEDIA            my_media;
UINT                status;
/* Clear the previously setup local path for this media. */
status = fx_directory_local_path_clear(&my_media);
/* If status equals FX_SUCCESS the local path is cleared. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find

- fx_directory_information_get
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_local_path_get：

获取当前的本地路径字符串

原型

```
UINT fx_directory_local_path_clear(
    FX_MEDIA *media_ptr,
    CHAR **return_path_name);
```

说明

此服务返回指定媒体的本地路径指针。如果没有设置本地路径，则会向调用方返回 NULL。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **return_path_name**：指向本地路径字符串要存储的目标字符串指针的指针。

返回值

- **FX_SUCCESS** (0x00) 成功获取本地路径。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体当前未打开
- **FX_NOT_IMPLEMENTED** (0x22) NX_NO_LCOAL_PATH
- **FX_PTR_ERROR** (0x18) 媒体指针无效

允许来自

线程数

示例

```
FX_MEDIA          my_media;
CHAR              *my_path;
UINT              status;
/* Retrieve the current local path string. */
status = fx_directory_local_path_get(&my_media, &my_path);
/* If status equals FX_SUCCESS, "my_path" points to the local path string. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create

- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_local_path_restore：

还原以前的本地路径

原型

```
UINT fx_directory_local_path_restore(
    FX_MEDIA *media_ptr,
    FX_LOCAL_PATH *local_path_ptr);
```

说明

此服务将还原以前设置的本地路径。设置在此本地路径上的目录搜索位置也会被存储，这使得此例程在应用程序的递归目录遍历中非常有用。

> **IMPORTANT**
>
> 每个本地路径都包含 FX_MAXIMUM_PATH 大小（默认值为 256 个字符）的本地路径字符串。此内部路径字符串不用于 FileX，仅供应用程序使用。如果 FX_LOCAL_PATH 将声明为本地变量，则用户应注意增加了此结构大小的堆栈。用户可随意减小 FX_LOCAL_PATH 的大小，并重新生成 FileX 库源 。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **local_path_ptr**：指向先前设置的本地路径的指针。请务必确保此指针确实指向之前使用的并仍保持不变的本地路径。

返回值

- **FX_SUCCESS** (0x00) 成功还原本地路径。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体当前未打开。
- **FX_NOT_IMPLEMENTED** (0x22) 已定义 FX_NO_LCOAL_PATH。
- **FX_PTR_ERROR** (0x18) 媒体或本地路径指针无效。

允许来自

线程数

示例

```
FX_MEDIA                    my_media;
FX_LOCAL_PATH               my_previous_local_path;
UINT                        status;
/* Restore the previous local path. */
status = fx_directory_local_path_restore(&my_media, &my_previous_local_path);
/* If status equals FX_SUCCESS, the previous local path has been restored. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_local_path_set

设置特定于线程的本地路径

原型

```
UINT fx_directory_local_path_set(
    FX_MEDIA *media_ptr,
    FX_LOCAL_PATH *local_path_ptr,
    CHAR *new_path_name);
```

说明

此服务按 new_path_string 指定的方式设置特定于线程的路径。成功完成此例程后，local_path_ptr 中存储的本地路径信息将优先于此线程所执行的所有文件和目录操作的全局媒体路径。这不会影响系统中的任何其他线程

> **IMPORTANT**
>
> 本地路径字符串的默认大小为 256 个字符；通过修改 fx_api.h 中的 FX_MAXIMUM_PATH 并重新生成整个 FileX 库可以更改大小。字符串路径会保留用于应用程序，不会由 FileX 在内部使用 。

> **IMPORTANT**
>
> 对于应用程序提供的名称，FileX 支持反斜杠 (\) 和正斜杠字符串来分隔目录、子目录和文件名。但是，FileX 仅在返回到应用程序的路径中使用反斜杠字符。

输入参数

- **media_ptr**：指向先前打开的媒体的指针。
- **local_path_ptr**：保存特定于线程的本地路径信息的目标。将来可以向本地路径还原功能提供此结构的地址。
- **new_path_name**：指定安装程序的本地路径。

返回值

- **FX_SUCCESS** (0x00) 成功设置默认目录。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_NOT_IMPLEMENTED** (0x22) **FX_NO_LCOAL_PATH
- **FX_INVALID_PATH** (0x0D) 找不到新目录。
- **FX_NOT_IMPLEMENTED** (0x22)- **已定义 FX_NO_LOCAL_PATH。
- **FX_PTR_ERROR** (0x18) 媒体或本地路径指针无效。

允许来自

线程数

示例

```
FX_MEDIA         my_media;
UINT             status;
FX_LOCAL_PATH     my_previous_local_path;
/* Set the local path to \abc\def\ghi. */
status = fx_directory_local_path_set (&my_media,&local_path,"\\abc\\def\\ghi");

/* If status equals FX_SUCCESS, the default directory for this thread
is \abc\def\ghi. All subsequent file operations that do not explicitly
specify a path will default to this directory. Note that the character
"\" serves as an escape character in a string. To represent the
character "\", use the construct "\\".*/
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find

- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_long_name_get：

从短名称获取长名称

原型

```
UINT fx_directory_long_name_get(
    FX_MEDIA *media_ptr,
    CHAR *short_name,
    CHAR *long_name);
```

说明

此服务检索与提供的短（8.3 格式）名称关联的长名称（如果有）。短名称可以是文件名或目录名。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **short_name**：指向源短名称（8.3 格式）的指针。
- **long_name**：指向长名称的目标的指针。如果没有长名称，则返回短名称。请注意，长名称的目标必须足够大，才能容纳 FX_MAX_LONG_NAME_LEN 个字符。

返回值

- **FX_SUCCESS** (0x00) 成功获取长名称
- **FX_NOT_FOUND** (0x04) 未找到短名称
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误
- **FX_MEDIA_INVALID** (0x02) 媒体无效
- **FX_FILE_CORRUPT** (0x08) 文件已损坏
- **FX_SECTOR_INVALID** (0x89) 扇区无效
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_PTR_ERROR** (0x18) 媒体或名称指针无效
- **FX_CALLER_ERROR** (0x20) 调用方不是线程

允许来自

线程数

示例

```
FX_MEDIA            my_media;
UCHAR               my_long_name[FX_MAX_LONG_NAME_LEN];
/* Retrieve the long name associated with "TEXT~01.TXT". */
status = fx_directory_long_name_get(&my_media, "TEXT~01.TXT", my_long_name);
/* If status is FX_SUCCESS the long name was successfully retrieved. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set

- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_long_name_get_extended

从短名称获取长名称

原型

```
UINT fx_directory_long_name_get_extended(
    FX_MEDIA *media_ptr,
    CHAR *short_name,
    CHAR *long_name,
    UINT long_file_name_buffer_length);
```

说明

此服务检索与提供的短(8.3 格式)名称关联的长名称(如果有)。短名称可以是文件名或目录名。

输入参数

- **media_ptr**:指向媒体控制块的指针。
- **short_name**:指向源短名称(8.3 格式)的指针。
- **long_name**:指向长名称的目标的指针。如果没有长名称，则返回短名称。注意:长名称的目标必须足够大，才能容纳 FX_MAX_LONG_NAME_LEN 个字符。
- **long_file_name_buffer_length**:长名称缓冲区的长度。

返回值

- **FX_SUCCESS** (0x00) 成功获取长名称。
- **FX_NOT_FOUND** (0x04) 未找到短名称。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_MEDIA_INVALID** (0x02) 媒体无效。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作。

- **FX_PTR_ERROR** (0x18) 媒体或名称指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        my_media;
UCHAR            my_long_name[FX_MAX_LONG_NAME_LEN];
/* Retrieve the long name associated with "TEXT~01.TXT". */

status = fx_directory_long_name_get_extended(&my_media,
    "TEXT~01.TXT", my_long_name), sizeof(my_long_name));

/* If status is FX_SUCCESS the long name was successfully retrieved. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_name_test

目录的测试

原型

```
UINT fx_directory_name_test(
    FX_MEDIA *media_ptr,
    CHAR *directory_name);
```

说明

此服务将测试提供的名称是否为目录。如果是，则返回 FX_SUCCESS。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **directory_name**：指向目录条目名称的指针。

返回值

- **FX_SUCCESS** (0x00) 提供的名称是目录。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开
- **FX_NOT_FOUND** (0x04) 找不到目录条目。
- **FX_NOT_DIRECTORY** (0x0E) 条目不是目录
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_MEDIA_INVALID** (0x02) 媒体无效。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作。
- **FX_PTR_ERROR** (0x18) 媒体或名称指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        my_media;
UNIT             status;
/* Check to see if the name "abc" is directory */

status = fx_directory_name_test(&my_media, "abc");

/* If status equals FX_SUCCESS "abc" is a directory. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get

- fx_unicode_directory_create

fx_directory_next_entry_find

选取下一个目录条目

原型

```
UINT fx_directory_next_entry_find(
    FX_MEDIA *media_ptr,
    CHAR *return_entry_name);
```

说明

此服务返回当前默认目录中的下一个条目名称。

> **WARNING**
> 如果使用非本地路径，也务必在目录遍历发生时（通过 ThreadX 信号灯或线程优先级）阻止其他应用程序线程更改此目录。否则，可能获得无效结果。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **return_entry_name**：指向默认目录中下一个条目名称的目标的指针。此指针指向的缓冲区必须足够大，才能容纳由 FX_MAX_LONG_NAME_LEN 定义的最大大小的 FileX 名称。

返回值

- **FX_SUCCESS** (0x00) 成功找到下一个条目
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_NO_MORE_ENTRIES** (0x0F) 此目录中不再有条目。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_PTR_ERROR** (0x18) 媒体指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        my_media;
CHAR            next_name[FX_MAX_LONG_NAME_LEN];
UINT            status;

/* Retrieve the next entry in the default directory. */

status = fx_directory_next_entry_find(&my_media, next_name);

/* If status equals TX_SUCCESS, the name of the next directory entry is in "next_name". */
```

另请参阅

- fx_directory_attributes_read

- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_next_full_entry_find：

获取包含完整信息的下一个目录条目

原型

```
UINT fx_directory_next_full_entry_find(
    FX_MEDIA *media_ptr,
    CHAR *directory_name,
    UINT *attributes,
    ULONG *size,
    UINT *year,
    UINT *month,
    UINT *day,
    UINT *hour,
    UINT *minute,
    UINT *second);
```

说明

此服务检索默认目录中的下一个条目名称，并将其复制到指定目标。它还会返回有关其他输入参数指定的条目的完整信息。

> **WARNING**
>
> 指定的目标必须足够大，才能容纳 FX_MAX_LONG_NAME_LEN 定义的最大大小的 FileX 名称

> **WARNING**
>
> 如果使用非本地路径，则务必在目录遍历发生时（通过 ThreadX 信号灯、互斥或优先级更改）阻止其他应用程序线程更改此目录。否则，可能获得无效结果。

输入参数

- media_ptr：指向媒体控制块的指针。
- directory_name：指向目录条目名称的目标的指针。大小必须至少为 FX_MAX_LONG_NAME_LEN。
- attributes：如果非 Null，则为指向条目属性要放置的目标的指针。属性以位图格式返回，具有以下可能的设置：
  - FX_READ_ONLY (0x01)
  - FX_HIDDEN (0x02)
  - FX_SYSTEM (0x04)
  - FX_VOLUME (0x08)
  - FX_DIRECTORY (0x10)
  - FX_ARCHIVE (0x20)
- size：如果非 Null，则为指向条目大小（字节）的目标的指针。
- month：如果非 Null，则为指向条目的修改月份的目标的指针。
- year：如果非 Null，则为指向条目的修改年份的目标的指针。
- day：如果非 Null，则为指向条目的修改日期的目标的指针。
- hour：如果非 Null，则为指向条目的修改时间（小时）的目标的指针。
- minute：如果非 Null，则为指向条目的修改时间（分钟）的目标的指针。
- second：如果非 Null，则为指向条目的修改时间（秒）的目标的指针。

返回值

- FX_SUCCESS (0x00) 成功找到下一个目录条目。
- FX_MEDIA_NOT_OPEN (0x11) 指定的媒体未打开。
- FX_NO_MORE_ENTRIES (0x0F) 此目录中不再有条目。
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误。
- FX_FILE_CORRUPT (0x08) 文件已损坏。
- FX_SECTOR_INVALID (0x89) 扇区无效。
- FX_FAT_READ_ERROR (0x03) 无法读取 FAT 条目。
- FX_NO_MORE_SPACE (0x0A) 不再有空间来完成该操作。
- FX_MEDIA_INVALID (0x02) 媒体无效。
- FX_PTR_ERROR (0x18) 媒体指针无效或所有输入参数均为 NULL。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA    my_media;
UINT        status;
CHAR        entry_name[FX_MAX_LONG_NAME_LEN];
UINT        attributes;
ULONG       size;
UINT        year;
UINT        month;
UINT        day;
UINT        hour;
UINT        minute;
UINT        second;

/* Get the next directory entry in the default directory with full information. */
status = fx_directory_next_full_entry_find(&my_media, entry_name, &attributes, &size,
                                           &year, &month, &day,
                                           &hour, &minute, &second);

/* If status equals FX_SUCCESS, the entry's information is in the local variables. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_rename

重命名目录

原型

```
UINT fx_directory_rename(
    FX_MEDIA *media_ptr,
    CHAR *old_directory_name,
    CHAR *new_directory_name);
```

说明

此服务将目录名称更改为指定的新目录名称。在指定路径的相对路径或默认路径中还可以完成重命名。如果在新目录名称中指定路径，则重命名的目录将被有效地移动到指定的路径。如果未指定路径，则重命名的目录将被放置在当前默认路径中。

输入参数

- media_ptr：指向媒体控制块的指针。
- old_directory_name：指向当前目录名称的指针。
- ：指向新目录名称的指针。

返回值

- FX_SUCCESS (0x00) 成功重命名目录。
- FX_MEDIA_NOT_OPEN (0x11) 指定的媒体未打开。
- FX_NOT_FOUND (0x04) 找不到目录条目。
- FX_NOT_DIRECTORY (0x0E) 条目不是目录。
- FX_INVALID_NAME (0x0C) 新目录名称无效。
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误。
- FX_WRITE_PROTECT (0x23) 指定的媒体受到写入保护。
- FX_FILE_CORRUPT (0x08) 文件已损坏。
- FX_SECTOR_INVALID (0x89) 扇区无效。
- FX_FAT_READ_ERROR (0x03) 无法读取 FAT 条目。
- FX_NO_MORE_SPACE (0x0A) 不再有空间来完成该操作。
- FX_MEDIA_INVALID (0x02) 媒体无效。
- FX_NO_MORE_ENTRIES (0x0F) 此目录中不再有条目。
- FX_INVALID_PATH (0x0D) 目录名称中提供的路径无效。
- FX_ALREADY_CREATED (0x0B) 已创建指定的目录。
- FX_PTR_ERROR (0x18) 媒体指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        my_media;
UINT            status;

/* Change the directory "abc" to "def". */
status = fx_directory_rename(&my_media, "abc", "def");

/* If status equals FX_SUCCESS, the directory was changed to "def". */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get

- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_short_name_get
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_short_name_get：

从长名称获取短名称

原型

```
UINT fx_directory_short_name_get(
    FX_MEDIA *media_ptr,
    CHAR *long_name,
    CHAR *short_name);
```

说明

此服务检测与提供的长名称关联的短（8.3 格式）名称。长名称可以是文件名或目录名。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **long_name**：指向源长名称的指针。
- **short_name**：指向目标短名称（8.3 格式）的指针。请注意，短名称的目标必须足够大，才能容纳 14 个字符。

返回值

- **FX_SUCCESS** (0x00) 成功获取短名称。
- **FX_NOT_FOUND** (0x04) 未找到长名称。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_MEDIA_INVALID** (0x02) 媒体无效。
- **FX_PTR_ERROR** (0x18) 媒体或名称指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA         my_media;
UCHAR             my_short_name[14];

/* Retrieve the short name associated with "my_really_long_name". */

status = fx_directory_short_name_get(&my_media,
    "my_really_long_name", my_short_name);

/* If status is FX_SUCCESS the short name was successfully retrieved. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_directory_short_name_get_extended

从长名称获取短名称

原型

```
UINT fx_directory_short_name_get_extended(
    FX_MEDIA *media_ptr,
    CHAR *long_name,
    CHAR *short_name,
    UINT short_file_name_length);
```

说明

此服务检测与提供的长名称关联的短(8.3 格式)名称。长名称可以是文件名或目录名。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **long_name**：指向源长名称的指针。
- **short_name**：指向目标短名称(8.3 格式)的指针。注意：短名称的目标必须足够大，才能容纳 14 个字符。

- short_file_name_length：短名称缓冲区的长度。

返回值

- **FX_SUCCESS** (0x00) 成功获取短名称。
- **FX_NOT_FOUND** (0x04) 未找到长名称。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_MEDIA_INVALID** (0x02) 媒体无效。
- **FX_PTR_ERROR** (0x18) 媒体或名称指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        my_media;
UCHAR            my_short_name[14];

/* Retrieve the short name associated with "my_really_long_name". */

status = fx_directory_short_name_get_extended(&my_media,
    "my_really_long_name", my_short_name, sizeof(my_short_name));

/* If status is FX_SUCCESS the short name was successfully retrieved. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_unicode_directory_create
- fx_unicode_directory_rename

fx_fault_tolerant_enable

启用容错服务

原型

```
UINT fx_fault_tolerant_enable(
    FX_MEDIA *media_ptr,
    VOID *memory_buffer,
    UINT memory_size);
```

说明

此服务启用容错模块。启动后，容错模块会检测文件系统是否受 FileX 容错保护。如果没有，则服务会在文件系统上找到可用扇区，以存储文件系统事务的相关日志。如果文件系统受 FileX 容错保护，则会将日志应用到文件系统以保持其完整性。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **memory_ptr**：指向容错模块用作暂存内存的内存块的指针。
- **memory_size**：暂存内存的大小。为了使容错正常工作，暂存内存大小至少应为 3072 个字节，并且必须是扇区大小的倍数。

返回值

- **FX_SUCCESS** (0x00) 已成功启用容错。
- **FX_NOT_ENOUGH_MEMORY** (0x91) 内存大小太小。
- **FX_BOOT_ERROR** (0x01) 启动扇区错误。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有可用的免费群集。
- **FX_NO_MORE_SPACE** (0x0A) 与此文件关联的媒体没有足够的可用群集。
- **FX_SECTOR_INVALID** (0x89) 扇区无效
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_PTR_ERROR** (0x18) 媒体指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

初始化、线程

示例

```
/* Declare memory space used for fault tolerant. */

ULONG fault tolerant_memory[3072 / sizeof(ULONG)];

/* Enable fault tolerant. */

fx_fault_tolerant_enable(media_ptr, fault_tolerant_memory, sizeof(fault_tolerant_memory));
```

另请参阅

- fx_system_initialize
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check

- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write

fx_file_allocate

为文件分配空间

原型

```
UINT fx_file_allocate(
    FX_FILE *file_ptr,
    ULONG size);
```

说明

此服务分配一个或多个连续群集并将其链接到指定文件的末尾。FileX 通过将请求的大小除以每个群集的字节数来确定所需的群集数。然后，将结果向上舍入到下一个整群集。

若要分配超过 4GB 的空间，应用程序应使用服务 fx_file_extended_allocate。

输入参数

- **file_ptr**：指向先前打开的文件的指针。
- **size**：要为文件分配的字节数。

返回值

- **FX_SUCCESS** (0x00) 文件分配成功。
- **FX_ACCESS_ERROR** (0x06) 指定的文件未打开以进行写入。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_NOT_OPEN** (0x07) 指定的文件当前未打开。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有可用的免费群集。
- **FX_NO_MORE_SPACE** (0x0A) 与此文件关联的媒体没有足够的可用群集。
- **FX_SECTOR_INVALID** (0x89) 扇区无效
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_PTR_ERROR** (0x18) 文件指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE            my_file;
UINT            status;

/* Allocate 1024 bytes to the end of my_file. */

status = fx_file_allocate(&my_file, 1024);

/* If status equals FX_SUCCESS the file now has one or more
    contiguous cluster(s) that can accommodate at least 1024 bytes of user data. */
```

另请参阅

- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open- fx_file_read
- fx_file_relative_seek
- fx_file_rename- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_attributes_read

读取文件属性

原型

```
    UINT fx_file_attributes_read(
    FX_MEDIA *media_ptr,
    CHAR *file_name,
    UINT *attributes_ptr);
```

说明

此服务从指定的媒体读取文件的属性。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **file_name**：指向请求的文件的名称的指针（目录路径是可选的）。
- **attributes_ptr**：指向文件属性要放置的目标的指针。文件属性以位图格式返回，具有以下可能的设置：
  - FX_READ_ONLY (0x01)
  - FX_HIDDEN (0x02)
  - FX_SYSTEM (0x04)
  - FX_VOLUME (0x08)
  - FX_DIRECTORY (0x10)
  - FX_ARCHIVE (0x20)

返回值

- **FX_SUCCESS** (0x00) 成功读取属性。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_NOT_FOUND** (0x04) 在媒体中未找到指定的文件。
- **FX_NOT_A_FILE** (0x05) 指定的文件是一个目录。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有 FAT 条目。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_PTR_ERROR** (0x18) 媒体或属性指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        my_media;
UINT            status;
UINT            attributes;

/* Retrieve the attributes of "myfile.txt" from the specified media. */

status = fx_file_attributes_read(&my_media, "myfile.txt", &attributes);

/* If status equals FX_SUCCESS, "attributes"
    contains the file attributes for "myfile.txt". */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek

- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_attributes_set

设置文件属性

原型

```
UINT fx_file_attributes_set(
    FX_MEDIA *media_ptr,
    CHAR *file_name,
    UINT attributes);
```

说明

此服务将文件的属性设置为调用方指定的属性。

> **WARNING**
>
> 此应用程序只能使用此服务修改文件的部分属性。尝试设置其他属性将导致错误。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **file_name**：指向请求的文件的名称的指针**（目录路径是可选的）。
- **attributes**：文件的新属性。有效文件属性定义如下：
  - FX_READ_ONLY (0x01)
  - FX_HIDDEN (0x02)
  - FX_SYSTEM (0x04)
  - FX_ARCHIVE (0x20)

返回值

- **FX_SUCCESS** (0x00) 成功设置属性。
- **FX_ACCESS_ERROR** (0x06) 文件处于打开状态，并且无法设置其属性。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。

- **FX_NO_MORE_ENTRIES** (0x0F) FAT 表或 exFAT 群集图中不再有条目。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作。
- **FX_NOT_FOUND** (0x04) 在媒体中未找到指定的文件。
- **FX_NOT_A_FILE** (0x05) 指定的文件是一个目录。
- **FX_SECTOR_INVALID** (0x89) 扇区无效
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_MEDIA_INVALID** (0x02) 媒体无效。
- **FX_PTR_ERROR** (0x18) 媒体指针无效。
- **FX_INVALID_ATTR** (0x19) 选择了无效属性。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        my_media;
UINT            status;

/* Set the attributes of "myfile.txt" to read-only. */

status = fx_file_attributes_set(&my_media, "myfile.txt", FX_READ_ONLY);

/* If status equals FX_SUCCESS, the file is now read-only. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set

- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_best_effort_allocate

为文件分配空间的最大工作量

原型

```
UINT fx_file_best_effort_allocate(
    FX_FILE *file_ptr,
    ULONG size,
    ULONG *actual_size_allocated);
```

说明

此服务分配一个或多个连续群集并将其链接到指定文件的末尾。FileX 通过将请求的大小除以每个群集的字节数来确定所需的群集数。然后，将结果向上舍入到下一个整群集。如果媒体中没有足够的连续可用群集，此服务会将连续群集的最大可用块链接到该文件。实际分配给该文件的空间量将返回给调用方。

若要分配超过 4GB 的空间，应用程序应使用服务 fx_file_extended_best_effort_allocate。

输入参数

- **file_ptr**：指向先前打开的文件的指针。
- **size**：要为文件分配的字节数。

返回值

- **FX_SUCCESS** (0x00) 成功完成最大工作量的文件分配。
- **FX_ACCESS_ERROR** (0x06) 指定的文件未打开以进行写入。
- **FX_NOT_OPEN** (0x07) 指定的文件当前未打开。
- **FX_NO_MORE_SPACE** (0x0A) 与此文件关联的媒体没有足够的可用群集。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有 FAT 条目。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_PTR_ERROR** (0x18) 文件指针或目标无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE         my_file;
UINT             status;
ULONG             actual_allocation;

/* Attempt to allocate 1024 bytes to the end of my_file. */

status = fx_file_best_effort_allocate(&my_file, 1024, &actual_allocation);

/* If status equals FX_SUCCESS, the number of bytes
    allocated to the file is found in actual_allocation. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_close

关闭文件

原型

```
UINT fx_file_close(FX_FILE *file_ptr);
```

说明

此服务关闭指定的文件。如果文件已打开以进行写入，并且该文件已修改，则此服务通过使用新大小和当前系统

的时间和日期更新其目录条目来完成文件修改过程。

输入参数

- file_ptr：指向先前打开的文件的指针。

返回值

- FX_SUCCESS (0x00) 成功关闭文件。
- FX_NOT_OPEN (0x07) 指定的文件未打开。
- FX_FILE_CORRUPT (0x08) 文件已损坏。
- FX_SECTOR_INVALID (0x89) 扇区无效。
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误。
- FX_PTR_ERROR (0x18) 媒体或属性指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE        my_file;
UINT        status;

/* Close the previously opened file "my_file". */
status = fx_file_close(&my_file);

/* If status equals FX_SUCCESS, the file was closed successfully. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set

- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_create

创建文件

原型

```
UINT fx_file_create(
    FX_MEDIA *media_ptr,
    CHAR *file_name);
```

说明

此服务在默认目录或文件名提供的目录路径中创建指定的文件。

> **WARNING**
>
> 此服务创建一个长度为零的文件，即未分配任何群集。然后在后续文件写入时自动进行分配，也可以使用 fx_file_allocate 服务或用于超过 4GB) 空间的 fx_file_extended_allocate 服务提前完成分配。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **file_name**：指向要创建的文件的名称的指针（目录路径是可选的）。

返回值

- **FX_SUCCESS** (0x00) 成功创建文件。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_ALREADY_CREATED** (0x0B) 指定的文件已创建。
- **FX_NO_MORE_SPACE** (0x0A) 根目录中不再有条目，或者不再有可用群集。
- **FX_INVALID_PATH** (0x0D) 文件名中提供的路径无效。
- **FX_INVALID_NAME** (0x0C) 文件名无效。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有 FAT 条目。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_MEDIA_INVALID** (0x02) 媒体无效。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 基础媒体受到写入保护。
- **FX_PTR_ERROR** (0x18) 媒体或文件名指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA         my_media;
UINT             status;

/* Create a file called "myfile.txt" in the
    root or the default directory of the media. */

status = fx_file_create(&my_media, "myfile.txt");

/* If status equals FX_SUCCESS, a zero sized file named "myfile.txt". */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_date_time_set

设置文件日期和时间

设置文件日期和时间

原型

```
UINT fx_file_date_time_set(
    FX_MEDIA *media_ptr,
    CHAR *file_name,
    UINT year,
    UINT month,
    UINT day,
    UINT hour,
    UINT minute,
    UINT second);
```

说明

此服务设置指定文件的日期和时间。

```
FX_MEDIA         my_media;
/* Set the date/time of "my_file". */
status = fx_file_date_time_set(&my_media, "my_file", 1999, 12, 31, 23, 59, 59);

/* If status is FX_SUCCESS the file's date/time was successfully set. /*
```

输入参数

- media_ptr：指向媒体控制块的指针。
- file_name：指向文件名称的指针。
- year：年份的值（1980-2107，包含端值）。
- month：月份的值（1-12，包含端值）。
- day：日期的值（1-31，包含端值）。
- hour：小时的值（0-23，包含端值）。
- minute：分钟的值（0-59，包含端值）。
- second：秒的值（0-59，包含端值）。

返回值

- FX_SUCCESS (0x00) 成功设置日期/时间。
- FX_MEDIA_NOT_OPEN (0x11) 指定的媒体未打开。
- FX_NOT_FOUND (0x04) 未找到文件。
- FX_FILE_CORRUPT (0x08) 文件已损坏。
- FX_SECTOR_INVALID (0x89) 扇区无效。
- FX_FAT_READ_ERROR (0x03) 无法读取 FAT 条目。
- FX_NO_MORE_ENTRIES (0x0F) 不再有 FAT 条目。
- FX_NO_MORE_SPACE (0x0A) 不再有空间来完成该操作。
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误。
- FX_WRITE_PROTECT (0x23) 指定的媒体受到写入保护。
- FX_PTR_ERROR (0x18) 媒体或名称指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。
- FX_INVALID_YEAR (0x12) 年份无效。
- FX_INVALID_MONTH (0x13) 月份无效。
- FX_INVALID_DAY (0x14) 日期无效。
- FX_INVALID_HOUR (0x15) 小时无效。
- FX_INVALID_MINUTE (0x16) 分钟无效。
- FX_INVALID_SECOND (0x17) 秒无效。

允许来自

线程数

示例

```
FX_MEDIA         my_media;
/* Set the date/time of "my_file". */
status = fx_file_date_time_set(&my_media, "my_file", 1999, 12, 31, 23, 59, 59);

/* If status is FX_SUCCESS the file's date/time was successfully set. /*
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_delete

删除文件

文件删除

原型

```
UINT fx_file_delete(
    FX_MEDIA *media_ptr,
    CHAR *file_name);
```

说明

此服务删除指定的文件。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **file_name**：指向要删除的文件的名称的指针（目录路径是可选的）。

返回值

- **FX_SUCCESS** (0x00) 成功删除文件。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_NOT_FOUND** (0x04) 未找到指定的文件。
- **FX_NOT_A_FILE** (0x05) 指定的文件名是目录或卷。
- **FX_ACCESS_ERROR** (0x06) 指定的文件当前处于打开状态。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有 FAT 条目。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_MEDIA_INVALID** (0x02) 媒体无效。
- **FX_PTR_ERROR** (0x18) 媒体指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA            my_media;
UINT                 status;
/* Delete the file "myfile.txt". */

status = fx_file_delete(&my_media, "myfile.txt");

/* If status equals FX_SUCCESS, "myfile.txt" has been deleted. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release

- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_extended_allocate

为文件分配空间

原型

```
UINT fx_file_extended_allocate(
    FX_FILE *file_ptr,
    ULONG64 size);
```

说明

此服务分配一个或多个连续群集并将其链接到指定文件的末尾。FileX 通过将请求的大小除以每个群集的字节数来确定所需的群集数。然后，将结果向上舍入到下一个整群集。

此服务是为 exFAT 设计的。size 参数采用 64 位整数值，使调用方能够预分配超过 4GB 范围的空间。

输入参数

- **file_ptr**：指向先前打开的文件的指针。
- **size**：要为文件分配的字节数。

返回值

- **FX_SUCCESS** (0x00) 文件分配成功。
- **FX_ACCESS_ERROR** (0x06) 指定的文件未打开以进行写入。
- **FX_NOT_OPEN** (0x07) 指定的文件当前未打开。
- **FX_NO_MORE_SPACE** (0x0A) 与此文件关联的媒体没有足够的可用群集。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有 FAT 条目。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_PTR_ERROR** (0x18) 文件指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE            my_file;
UINT            status;
/* Allocate 0x100000000 bytes to the end of my_file. */

status = fx_file_extended_allocate(&my_file, 0x100000000);

/* If status equals FX_SUCCESS the file now has
    one or more contiguous cluster(s) that can accommodate at least
    1024 bytes of user data. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_extended_best_effort_allocate

为文件分配空间的最大工作量

原型

```
UINT fx_file_extended best_effort_allocate(
    FX_FILE *file_ptr,
    ULONG64 size,
    ULONG64 *actual_size_allocated);
```

说明

此服务分配一个或多个连续群集并将其链接到指定文件的末尾。FileX 通过将请求的大小除以每个群集的字节数来确定所需的群集数。然后，将结果向上舍入到下一个整群集。如果媒体中没有足够的连续可用群集，此服务会将连续群集的最大可用块链接到该文件。实际分配给该文件的空间量将返回给调用方。

此服务是为 exFAT 设计的。size 参数采用 64 位整数值，使调用方能够预分配超过 4GB 范围的空间。

输入参数

- **file_ptr**：指向先前打开的文件的指针。
- **size**：要为文件分配的字节数。

返回值

- **FX_SUCCESS** (0x00) 文件分配成功。
- **FX_ACCESS_ERROR** (0x06) 指定的文件未打开以进行写入。
- **FX_NOT_OPEN** (0x07) 指定的文件当前未打开。
- **FX_NO_MORE_SPACE** (0x0A) 与此文件关联的媒体没有足够的可用群集。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有 FAT 条目。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_PTR_ERROR** (0x18) 文件指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE my_file;
UINT             status;
ULONG64         actual_allocation;

/* Attempt to allocate 0x100000000 bytes to the end of my_file. */

status = fx_file_extended_best_effort_allocate(&my_file,
    0x100000000, &actual_allocation);

/* If status equals FX_SUCCESS, the number of bytes
    allocated to the file is found in actual_allocation. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate

- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_extended_relative_seek

相对字节偏移量的位置

原型

```
UINT fx_file_extended_relative_seek(
    FX_FILE *file_ptr,
    ULONG64 byte_offset,
    UINT seek_from);
```

说明

此服务将内部文件读取/写入指针定位到指定的相对字节偏移量。任何后续的文件读取或写入请求将在文件中的此位置开始。

此服务是为 exFAT 设计的。byte_offset 参数采用 64 位整数值，使调用方能够重新定位超过 4GB 范围的读取/写入指针。

> **IMPORTANT**
> 如果搜寻操作尝试在文件末尾进行搜寻，则文件的读取/写入指针将定位到文件末尾。相反，如果搜寻操作尝试定位到文件开头，则文件的读取/写入指针将定位到文件的开头。

输入参数

- **file_ptr**：指向先前打开的文件的指针。
- **byte_offset**：文件中所需的相对字节偏移量。
- **seek_from**：从何处执行相对搜寻的方向和位置。有效搜寻选项的定义如下：
  - FX_SEEK_BEGIN (0x00)
  - FX_SEEK_END (0x01)
  - FX_SEEK_FORWARD (0x02)
  - FX_SEEK_BACK (0x03) 如果指定了 FX_SEEK_BEGIN，则将从文件的开头执行搜寻操作。如果指定了 FX_SEEK_END，则将从文件末尾向上执行搜寻操作。如果指定了 FX_SEEK_FORWARD，则将从当前文

件位置向下执行搜寻操作。如果指定了 FX_SEEK_BACK，则将从当前文件位置向上执行搜寻操作。

返回值

- **FX_SUCCESS** (0x00) 成功进行文件相对搜寻。
- **FX_NOT_OPEN** (0x07) 指定的文件当前未打开。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_PTR_ERROR** (0x18) 文件指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE     my_file;
UINT         status;

/* Attempt to seek forward 0x100000000 bytes in "my_file". */

status = fx_file_extended_relative_seek(&my_file, 0x100000000, FX_SEEK_FORWARD);

/* If status equals FX_SUCCESS, the file read/write
    pointers are positioned 0x100000000 bytes forward. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create

- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_extended_seek

定位到字节偏移量

原型

```
UINT fx_file_extended_seek(
    FX_FILE *file_ptr,
    ULONG64 byte_offset);
```

说明

此服务将内部文件读取/写入指针定位到指定的字节偏移量。任何后续的文件读取或写入请求将在文件中的此位置开始。

此服务是为 exFAT 设计的。byte_offset 参数采用 64 位整数值，使调用方能够重新定位超过 4GB 范围的读取/写入指针。

输入参数

- **file_ptr**：指向文件控制块的指针。
- **byte_offset**：文件中所需的字节偏移量。如果值为零，则会将读取/写入指针定位在文件的开头，而值大于文件大小，则会将读取/写入指针定位到文件末尾。

返回值

- **FX_SUCCESS** (0x00) 成功进行文件搜寻。
- **FX_NOT_OPEN** (0x07) 指定的文件未打开。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_PTR_ERROR** (0x18) 文件指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE         my_file;
UINT             status;
/* Seek to position 0x100000000 of "my_file." */

status = fx_file_extended_seek(&my_file, 0x100000000);

/* If status equals FX_SUCCESS, the file read/write pointer
    is now positioned 0x100000000 bytes from the beginning of the file. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set

- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_extended_truncate

截断文件

原型

```
UINT fx_file_truncate(
    FX_FILE *file_ptr,
    ULONG64 size);
```

说明

此服务将文件截断为指定大小。如果提供的大小大于实际文件大小，则此服务不会执行任何操作。不会释放与该文件关联的任何媒体群集。

> **WARNING**
>
> 在截断还可以同时打开以进行读取的文件时请务必小心。截断打开以进行读取的文件可能会导致读取无效数据。

此服务是为 exFAT 设计的。size 参数采用 64 位整数值，使调用方的操作范围超过 4GB。

输入参数

- **file_ptr**：指向文件控制块的指针。
- **size**：新的文件大小。超过此新文件大小的字节将被丢弃。

返回值

- **FX_SUCCESS** (0x00) 成功截断文件。
- **FX_NOT_OPEN** (0x07) 指定的文件未打开。

- **FX_ACCESS_ERROR** (0x06) 指定的文件未打开以进行写入。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有 FAT 条目。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 基础媒体受到写入保护。
- **FX_PTR_ERROR** (0x18) 文件指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE             my_file;
UINT             status;
/* Truncate "my_file" to 0x100000000 bytes. */

status = fx_file_extended_truncate(&my_file, 0x100000000);

/* If status equals FX_SUCCESS, "my_file" contains 0x100000000 or fewer bytes. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_extended_truncate_release

截断文件并释放群集

原型

```
UINT fx_file_extended_truncate_release(
    FX_FILE *file_ptr,
    ULONG64 size);
```

说明

此服务将文件截断为指定大小。如果提供的大小大于实际文件大小，则此服务不会执行任何操作。与 fx_file_extended_truncate 服务不同，此服务不释放任何未使用的群集。

> **WARNING**
>
> 在截断还可以同时打开以进行读取的文件时请务必小心。截断打开以进行读取的文件可能会导致读取无效数据。

此服务是为 exFAT 设计的。size 参数采用 64 位整数值，使调用方的操作范围超过 4GB。

输入参数

- **file_ptr**：指向先前打开的文件的指针。
- **size**：新的文件大小。超过此新文件大小的字节将被丢弃。

返回值

- **FX_SUCCESS** (0x00) 成功截断文件。
- **FX_ACCESS_ERROR** (0x06) 指定的文件未打开以进行写入。
- **FX_NOT_OPEN** (0x07) 指定的文件当前未打开。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有 FAT 条目。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_PTR_ERROR** (0x18) 文件指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE            my_file;
UINT           status;
/* Attempt to truncate everything after the first 0x100000000 bytes of "my_file". */

status = fx_file_extended_truncate_release(&my_file, 0x100000000);

/* If status equals FX_SUCCESS, the file is now 0x100000000
    bytes or fewer and all unused clusters have been released. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_open

打开文件

原型

```
UINT fx_file_open(
    FX_MEDIA *media_ptr,
    FX_FILE *file_ptr,
    CHAR *file_name,
    UINT open_type);
```

说明

此服务将打开指定的文件以进行读取或写入操作中的一种。一个文件在打开后可以进行多次读取，而在写入器关闭文件前只能进行一次写入。

> **IMPORTANT**
>
> 在文件打开后同时进行读取和写入时请务必小心。当文件打开后同时进行读取操作时，读取器可能不会看到执行的文件写入操作，除非读取器关闭并重新打开以进行读取。同样，在使用文件截断服务时，应小心使用文件写入器。如果某个文件由写入器截断，则同一个文件的读取器可能会返回无效数据。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **file_ptr**：指向文件控制块的指针。
- **file_name**：指向要打开的文件的名称的指针（目录路径是可选的）。
- **open_type**：文件打开的类型。有效的打开类型选项有：
  - FX_OPEN_FOR_READ (0x00)
  - FX_OPEN_FOR_WRITE (0x01)
  - FX_OPEN_FOR_READ_FAST (0x02)

FX_OPEN_FOR_READ 和 FX_OPEN_FOR_READ_FAST 的打开文件相似：

- FX_OPEN_FOR_READ 包括验证包含文件的群集的链接列表是否保持不变，而 FX_OPEN_FOR_READ_FAST 不会执行此验证，因此速度更快。

返回值

- **FX_SUCCESS** (0x00) 成功打开文件。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_NOT_FOUND** (0x04) 未找到指定的文件。
- **FX_NOT_A_FILE** (0x05) 指定的文件名是目录或卷。
- **FX_FILE_CORRUPT** (0x08) 指定的文件已损坏，并且打开失败。
- **FX_ACCESS_ERROR** (0x06) 指定的文件已打开，或打开的类型无效。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_MEDIA_INVALID** (0x02) 媒体无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 基础媒体受到写入保护。
- **FX_PTR_ERROR** (0x18) 媒体或文件指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA     my_media;
FX_FILE     my_file;
UINT         status;

/* Open the file "myfile.txt" for reading. */

status = fx_file_open(&my_media, &my_file, "myfile.txt", FX_OPEN_FOR_READ);

/* If status equals FX_SUCCESS, file "myfile.txt" is now
    open and may be accessed now with the my_file pointer. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close- fx_file_create
- fx_file_date_time_set

- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_read

读取文件的字节

原型

```
UINT fx_file_read(
    FX_FILE *file_ptr,
    VOID *buffer_ptr,
    ULONG request_size,
    ULONG *actual_size);
```

说明

此服务从文件中读取字节，并将它们存储在所提供的缓冲区中。读取完成后，文件的内部读取指针将调整为指向文件中的下一个字节。如果请求中剩余的字节更少，则仅剩余的字节存储在缓冲区中。在任何情况下，缓冲区中放置的字节总数将返回给调用方。

> **WARNING**
>
> 应用程序必须确保所提供的缓冲区能够存储指定数量的请求字节。

> **WARNING**
>
> 如果目标缓冲区在长字边界上，并且请求的大小被 sizeof (ULONG) 整除，则可实现更快的性能。

输入参数

- **file_ptr**：指向文件控制块的指针。
- **buffer_ptr**：指向读取的目标缓冲区的指针。
- **request_size**：要读取的最多的字节数。
- **actual_size**：指向用于保存读取到提供的缓冲区中的实际字节数的变量的指针。

返回值

- **FX_SUCCESS** (0x00) 成功读取文件。
- **FX_NOT_OPEN** (0x07) 指定的文件未打开。
- **FX_FILE_CORRUPT** (0x08) 指定的文件已损坏，并且读取失败。
- **FX_END_OF_FILE** (0x09) 已到达文件的末尾。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_PTR_ERROR** (0x18) 文件或缓冲区指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE                my_file;
unsigned char          my_buffer[1024];
ULONG                  actual_bytes;
UINT                   status;

/* Read up to 1024 bytes into "my_buffer." */
status = fx_file_read(&my_file, my_buffer, 1024, &actual_bytes);

/* If status equals FX_SUCCESS, "my_buffer" contains the bytes
    read from the file. The total number of bytes read is in "actual_bytes." */
```

另请参阅

- fx_file_allocate,
- fx_file_attributes_read,
- fx_file_attributes_set,
- fx_file_best_effort_allocate,
- fx_file_close,
- fx_file_create,
- fx_file_date_time_set,
- fx_file_delete,
- fx_file_extended_allocate,
- fx_file_extended_best_effort_allocate,
- fx_file_extended_relative_seek,
- fx_file_extended_seek,
- fx_file_extended_truncate,
- fx_file_extended_truncate_release,
- fx_file_open,
- fx_file_relative_seek,
- fx_file_rename,
- fx_file_seek,
- fx_file_truncate,
- fx_file_truncate_release,
- fx_file_write,
- fx_file_write_notify_set,
- fx_unicode_file_create,

- fx_unicode_file_rename,
- fx_unicode_name_get,
- fx_unicode_short_name_get

fx_file_relative_seek

相对字节偏移量的位置

原型

```
UINT fx_file_relative_seek(
    FX_FILE *file_ptr,
    ULONG byte_offset,
    UINT seek_from);
```

说明

此服务将内部文件读取/写入指针定位到指定的相对字节偏移量。任何后续的文件读取或写入请求将在文件中的此位置开始。

> **IMPORTANT**
> 如果搜寻操作尝试在文件末尾进行搜寻，则文件的读取/写入指针将定位到文件末尾。相反，如果搜寻操作尝试定位到文件开头，则文件的读取/写入指针将定位到文件的开头。

若要使用超过 4GB 的偏移值进行搜寻，应用程序应使用服务 fx_file_extended_relative_seek。

输入参数

- **file_ptr**：指向先前打开的文件的指针。
- **byte_offset**：文件中所需的相对字节偏移量。
- **seek_from**：从何处执行相对搜寻的方向和位置。有效搜寻选项的定义如下：
  - FX_SEEK_BEGIN (0x00)
  - FX_SEEK_END (0x01)
  - FX_SEEK_FORWARD (0x02)
  - FX_SEEK_BACK (0x03)

如果指定了 FX_SEEK_BEGIN，则将从文件的开头执行搜寻操作。如果指定了 FX_SEEK_END，则将从文件末尾向上执行搜寻操作。如果指定了 FX_SEEK_FORWARD，则将从当前文件位置向下执行搜寻操作。如果指定了 FX_SEEK_BACK，则将从当前文件位置向上执行搜寻操作。

返回值

- **FX_SUCCESS** (0x00) 成功进行文件相对搜寻。
- **FX_NOT_OPEN** (0x07) 指定的文件当前未打开。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有 FAT 条目。
- **FX_PTR_ERROR** (0x18) 文件指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE     my_file;
UINT         status;

/* Attempt to move 10 bytes forward in "my_file". */

status = fx_file_relative_seek(&my_file, 10, FX_SEEK_FORWARD);

/* If status equals FX_SUCCESS, the file read/write pointers
    are positioned 10 bytes forward. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_rename

重命名文件

原型

```
UINT fx_file_rename(
    FX_MEDIA *media_ptr,
    CHAR *old_file_name,
    CHAR *new_file_name);
```

说明

此服务更改 old_file_name 指定的文件的名称。在指定路径的相对路径或默认路径中还可以完成重命名。如果在新文件名中指定路径，则重命名的文件将被有效地移动到指定的路径。如果未指定路径，则重命名的文件将被放置在当前默认路径中。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **old_file_name**：指向要重命名的文件名称的指针（目录路径是可选的）。
- **new_file_name**：指向新文件名的指针。不允许目录路径。

返回值

- FX_SUCCESS (0x00) 成功重命名文件。
- FX_MEDIA_NOT_OPEN (0x11) 指定的媒体未打开。
- FX_NOT_FOUND (0x04) 未找到指定的文件。
- FX_NOT_A_FILE (0x05) 指定的文件是一个目录。
- FX_ACCESS_ERROR (0x06) 指定的文件已处于打开状态。
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误。
- FX_WRITE_PROTECT (0x23) 指定的媒体受到写入保护。
- FX_INVALID_NAME (0x0C) 指定的新文件名不是有效的文件名。
- FX_INVALID_PATH (0x0D) 路径无效。
- FX_ALREADY_CREATED (0x0B) 已使用新文件名称。
- FX_MEDIA_INVALID (0x02) 媒体无效。
- FX_FILE_CORRUPT (0x08) 文件已损坏。
- FX_SECTOR_INVALID (0x89) 扇区无效。
- FX_NO_MORE_ENTRIES (0x0F) 不再有 FAT 条目。
- FX_NO_MORE_SPACE (0x0A) 不再有空间来完成该操作
- FX_FAT_READ_ERROR (0x03) 无法读取 FAT 表。
- FX_PTR_ERROR (0x18) 媒体指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        my_media;
UINT            status;

/* Rename "myfile1.txt" to "myfile2.txt" in the default directory of the media. */

status = fx_file_rename(&my_media, "myfile1.txt", "myfile2.txt");

/* If status equals FX_SUCCESS, the file was successfully renamed. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set

- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_seek

定位到字节偏移量

原型

```
UINT fx_file_seek(
    FX_FILE *file_ptr,
    ULONG byte_offset);
```

说明

此服务将内部文件读取/写入指针定位到指定的字节偏移量。任何后续的文件读取或写入请求将在文件中的此位置开始。

若要使用超过 4GB 的偏移值进行搜寻，应用程序应使用服务 fx_file_extended_seek。

输入参数

- **file_ptr**：指向文件控制块的指针。
- **byte_offset**：文件中所需的字节偏移量。如果值为零，则会将读取/写入指针定位在文件的开头，而值大于文件大小，则会将读取/写入指针定位到文件末尾。

返回值

- **FX_SUCCESS** (0x00) 成功进行文件搜寻。
- **FX_NOT_OPEN** (0x07) 指定的文件未打开。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_NO_MORE_SPACE** (0x0A) 不再有空间来完成该操作
- **FX_PTR_ERROR** (0x18) 文件指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE            my_file;
UINT            status;
/* Seek to the beginning of "my_file." */
status = fx_file_seek(&my_file, 0);
/* If status equals FX_SUCCESS, the file read/write pointer
    is now positioned to the beginning of the file. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_truncate

截断文件

原型

```
UINT fx_file_truncate(
    FX_FILE *file_ptr,
    ULONG size);
```

说明

此服务将文件截断为指定大小。如果提供的大小大于实际文件大小，则此服务不会执行任何操作。不会释放与该文件关联的任何媒体群集。

> **WARNING**
>
> 在截断还可以同时打开以进行读取的文件时请务必小心。截断打开以进行读取的文件可能会导致读取无效数据。

若要超过 4GB 范围进行操作，应用程序应使用服务 fx_file_extended_truncate。

输入参数

- file_ptr：指向文件控制块的指针。
- size：新的文件大小。超过此新文件大小的字节将被丢弃。

返回值

- FX_SUCCESS (0x00) 成功截断文件。
- FX_NOT_OPEN (0x07) 指定的文件未打开。
- FX_ACCESS_ERROR (0x06) 指定的文件未打开以进行写入。
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误。
- FX_WRITE_PROTECT (0x23) 指定的媒体受到写入保护。
- FX_FILE_CORRUPT (0x08) 文件已损坏。
- FX_SECTOR_INVALID (0x89) 扇区无效。
- FX_NO_MORE_ENTRIES (0x0F) 不再有 FAT 条目。
- FX_NO_MORE_SPACE (0x0A) 不再有空间来完成该操作
- FX_PTR_ERROR (0x18) 文件指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE                 my_file;
UINT                status;
/* Truncate "my_file" to 100 bytes. */

status = fx_file_truncate(&my_file, 100);

/* If status equals FX_SUCCESS, "my_file" contains 100 or fewer bytes. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek

- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_truncate_release

截断文件并释放群集

原型

```
UINT fx_file_truncate(
    FX_FILE *file_ptr,
    ULONG size);
```

说明

此服务将文件截断为指定大小。如果提供的大小大于实际文件大小，则此服务不会执行任何操作。与 fx_file_truncate 服务不同，此服务不释放任何未使用的群集。

> **WARNING**
>
> 在截断还可以同时打开以进行读取的文件时请务必小心。截断打开以进行读取的文件可能会导致读取无效数据。

要在超过 4GB 范围进行操作，应用程序应使用服务 fx_file_extended_truncate_release。

输入参数

- **file_ptr**：指向先前打开的文件的指针。
- **size**：新的文件大小。超过此新文件大小的字节将被丢弃。

返回值

- **FX_SUCCESS** (0x00) 成功截断文件。
- **FX_ACCESS_ERROR** (0x06) 指定的文件未打开以进行写入。
- **FX_NOT_OPEN** (0x07) 指定的文件当前未打开。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 基础媒体受到写入保护。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有 FAT 条目。

- FX_NO_MORE_SPACE (0x0A) 不再有空间来完成该操作。
- FX_PTR_ERROR (0x18) 文件指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE         my_file;
UINT             status;

/* Attempt to truncate everything after the first 100 bytes of "my_file". */

status = fx_file_truncate_release(&my_file, 100);

/* If status equals FX_SUCCESS, the file is now 100 bytes
    or fewer and all unused clusters have been released. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_file_write

写入文件的字节

原型

```
UINT fx_file_write(
    FX_FILE *file_ptr,
    VOID *buffer_ptr,
    ULONG size);
```

说明

此服务从文件的当前位置开始，写入指定缓冲区中的字节。写入完成后，文件的内部读取指针将调整为指向文件中的下一个字节。

> **WARNING**
>
> 如果源缓冲区在长字边界上，并且请求的大小被 sizeof (ULONG) 整除，则可实现更快的性能。

输入参数

- **file_ptr**：指向文件控制块的指针。
- **buffer_ptr**：指向写入的源缓冲区的指针。
- **size**：要写入的字节数。

返回值

- **FX_SUCCESS** (0x00) 成功写入文件。
- **FX_NOT_OPEN** (0x07) 指定的文件未打开。
- **FX_ACCESS_ERROR** (0x06) 指定的文件未打开以进行写入。
- **FX_NO_MORE_SPACE** (0x0A) 媒体中不再有可用空间来执行此写入。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 条目。
- **FX_NO_MORE_ENTRIES** (0x0F) 不再有 FAT 条目。
- **FX_PTR_ERROR** (0x18) 文件或缓冲区指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_FILE            my_file;
UINT             status;
/* Write a 10 character buffer to "my_file." */

status = fx_file_write(&my_file, "1234567890", 10);

/* If status equals FX_SUCCESS, the small text string was written out to the file. */
```

另请参阅

- fx_file_allocate,
- fx_file_attributes_read,
- fx_file_attributes_set,
- fx_file_best_effort_allocate,

- fx_file_close,
- fx_file_create,
- fx_file_date_time_set,
- fx_file_delete,
- fx_file_extended_allocate,
- fx_file_extended_best_effort_allocate,
- fx_file_extended_relative_seek,
- fx_file_extended_seek,
- fx_file_extended_truncate,
- fx_file_extended_truncate_release,
- fx_file_open,
- fx_file_read,
- fx_file_relative_seek,
- fx_file_rename,
- fx_file_seek,
- fx_file_truncate,
- fx_file_truncate_release,
- fx_file_write_notify_set,
- fx_unicode_file_create,
- fx_unicode_file_rename,
- fx_unicode_name_get,
- fx_unicode_short_name_get

fx_file_write_notify_set

设置文件写入通知函数

原型

```
UINT fx_file_write_notify_set(
    FX_FILE *file_ptr,
    VOID (*file_write_notify)(FX_FILE*));
```

说明

此服务安装在文件写入操作成功后调用的回调函数。

输入参数

- **file_ptr**：指向文件控制块的指针。
- **file_write_notify**：要安装的文件写入回调函数。将回调函数设置为 NULL 可禁用回调函数。

返回值

- **FX_SUCCESS** (0x00) 已成功安装回调函数。
- **FX_PTR_ERROR** (0x18) file_ptr 为 NULL。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
fx_file_write_notify_set(file_ptr, my_file_close_callback);
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_media_abort

中止媒体活动

原型

```
UINT fx_media_abort(FX_MEDIA *media_ptr);
```

说明

此服务将中止与媒体关联的所有当前活动，包括关闭所有打开的文件，将中止请求发送到关联的驱动程序，并将媒体置于中止状态。如果检测到 I/O 错误，通常会调用此服务。

> **WARNING**
> 执行中止操作后，必须重新打开媒体以再次使用。

输入参数

- **media_ptr**：指向媒体控制块的指针。

返回值

- **FX_SUCCESS** (0x00) 成功中止媒体。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_PTR_ERROR** (0x18) 媒体指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA    my_media;
UINT        status;
/* Abort all activity associated with "my_media". */

status = fx_media_abort(&my_media);

/* If status equals FX_SUCCESS, all activity
    associated with the media has been aborted. */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_cache_invalidate

使逻辑扇区缓存失效

原型

```
UINT fx_media_cache_invalidate(FX_MEDIA *media_ptr);
```

说明

此服务刷新缓存中的所有脏扇区，使整个逻辑扇区缓存失效。

输入参数

- media_ptr：指向媒体控制块的指针

返回值

- FX_SUCCESS (0x00) 成功使媒体缓存失效。
- FX_MEDIA_NOT_OPEN (0x11) 指定的媒体未打开。
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误。
- FX_PTR_ERROR (0x18) 媒体或暂存指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA     my_media;

/* Invalidate the cache of the media. */
status = fx_media_cache_invalidate(&my_media);

/* If status is FX_SUCCESS the cache in the media
    was successfully flushed and invalidated. */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_check

检查媒体错误

原型

```
UINT fx_media_check(
    FX_MEDIA *media_ptr,
    UCHAR *scratch_memory_ptr,
    ULONG scratch_memory_size,
    ULONG error_correction_option,
    ULONG *errors_detected_ptr);
```

说明

此服务检查指定的媒体是否存在基本结构错误，包括文件/目录交叉链接、FAT 链无效以及群集丢失。此服务还提供了更正检测到的错误的功能。

fx_media_check 服务需要暂存内存，以便对媒体中的目录和文件进行深度优先分析。具体而言，提供给媒体检查服务的暂存内存必须足够大，才能保存多个目录条目，这是一种在进入子目录之前"堆叠"当前目录条目位置，最后是逻辑 FAT 位图的数据结构。对于逻辑 FAT 位图，暂存内存应至少有 512-1024 字节的额外内存，这需要位数与媒体中的群集数量相同。例如，具有 8000 群集的设备需要 1000 个字节来表示，因此需要一个大约 2048 字节的总暂存区。

> **WARNING**
> 只能在 fx_media_open 之后且没有任何其他文件系统活动的情况下立即调用此服务。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **scratch_memory_ptr**：指向暂存内存起点的指针。
- **scratch_memory_size**：暂存内存的大小（字节）。
- **error_correction_option**：错误更正选项位，设置位后，将执行错误更正。错误更正选项位定义如下：
  - FX_FAT_CHAIN_ERROR (0x01)
  - FX_DIRECTORY_ERROR (0x02)
  - FX_LOST_CLUSTER_ERROR (0x04) 只是所需的错误更正选项。如果不需要错误更正，则应提供值 0。
- **errors_detected_ptr**：错误检测位的目标，如下所示：
  - FX_FAT_CHAIN_ERROR (0x01)
  - FX_DIRECTORY_ERROR (0x02) FX_LOST_CLUSTER_ERROR (0x04)
  - FX_FILE_SIZE_ERROR (0x08)

返回值

- **FX_SUCCESS** (0x00) 成功检查媒体，请查看检测到错误的目标以获取详细信息。
- **FX_ACCESS_ERROR** (0x06) 无法对打开的文件执行检查。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_NO_MORE_SPACE** (0x0A) 媒体上不再有空间。
- **FX_NOT_ENOUGH_MEMORY** (0x91) 提供的暂存内存不够大。
- **FX_ERROR_NOT_FIXED** (0x93) FAT32 根目录损坏不能修复。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_PTR_ERROR** (0x18) 媒体或暂存指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA         my_media;
ULONG             detected_errors;
UCHAR             sratch_memory[4096];

/* Check the media and correct all errors. */

status = fx_media_check(&my_media, sratch_memory, 4096,
                        FX_FAT_CHAIN_ERROR |
                        FX_DIRECTORY_ERROR |
                        FX_LOST_CLUSTER_ERROR, &detected_errors);

/* If status is FX_SUCCESS and detected_errors is 0,
    the media was successfully checked and found to be error free. */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_close

关闭媒体

原型

```
UINT fx_media_close(FX_MEDIA *media_ptr);
```

说明

此服务关闭指定的媒体。在关闭媒体的过程中，将关闭所有打开的文件，并将任何剩余的缓冲区刷新到物理媒体。

输入参数

- **media_ptr**：指向媒体控制块的指针。

返回值

- **FX_SUCCESS** (0x00) 成功关闭媒体。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。

- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_PTR_ERROR** (0x18) 媒体指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA    my_media;
UINT        status;
/* Close "my_media". */

status = fx_media_close(&my_media);

/* If status equals FX_SUCCESS, "my_media" is closed. */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_close_notify_set

设置媒体关闭通知功能

原型

```
UINT fx_media_close_notify_set(
    FX_MEDIA *media_ptr,
    VOID (*media_close_notify)(FX_MEDIA*));
```

说明

此服务设置一个通知回调函数，该函数将在成功关闭媒体后被调用。

输入参数

- **media_ptr**：指向媒体控制块的指针。

- **media_close_notify**：要安装的媒体关闭通知回调函数。传递 NULL 作为回调函数将禁用媒体关闭回调。

返回值

- **FX_SUCCESS** (0x00) 已成功安装回调函数。
- **FX_PTR_ERROR** (0x18) media_ptr 为 NULL。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
fx_media_close_notify_set(media_ptr, my_media_close_callback);
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_exFAT_format

格式化媒体

原型

```
UINT fx_media_exFAT_format(
    FX_MEDIA *media_ptr,
    VOID (*driver)(FX_MEDIA *media),
    VOID *driver_info_ptr,
    UCHAR *memory_ptr,
    UINT memory_size,
    CHAR *volume_name,
    UINT number_of_fats,
    ULONG64 hidden_sectors,
    ULONG64 total_sectors,
    UINT bytes_per_sector,
    UINT sectors_per_cluster,
    UINT volume_serial_number,
    UINT boundary_unit);
```

说明

此服务基于提供的参数以 exFAT 兼容的方式格式化提供的媒体。在打开媒体之前，必须调用此服务。

> **WARNING**
> 格式化已有格式的媒体可有效地清除媒体上的所有文件和目录。

输入参数

- media_ptr：指向媒体控制块的指针。该参数仅用于提供驱动程序运行所需的一些基本信息。
- driver：指向此媒体的 I/O 驱动程序的指针。这通常是提供给后续 fx_media_open 调用的同一个驱动程序。
- driver_info_ptr：指向 I/O 驱动程序可以使用的可选信息的指针。
- memory_ptr：指向媒体的工作内存的指针。memory_size 指定工作媒体内存的大小。大小必须至少与媒体扇区的大小相同。
- volume_name：指向卷名称字符串的指针，最多为 11 个字符。
- number_of_fats：媒体上的 FAT 数。当前实现支持媒体上有一个 FAT。
- hidden_sectors：在此媒体启动扇区前隐藏的扇区数。如果存在多个分区，则通常会出现这种情况。
- total_sectors：媒体中的扇区总数。
- bytes_per_sector：每个扇区的字节数，通常为 512。FileX 要求此数为 32 的倍数。
- sectors_per_cluster：每个群集中的扇区数。群集是 FAT 文件系统中的最小分配单元。
- volumne_serial_number：要用于此卷的序列号。
- boundary_unit：物理数据区域对齐大小（扇区数）。

返回值

- FX_SUCCESS (0x00) 成功格式化媒体。
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误。
- FX_PTR_ERROR (0x18) 媒体、驱动程序或内存指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA             sd_card;
UCHAR                 media_memory[512];

/* Format a 64GB SD card with exFAT file system. The media has
    been properly partitioned, with the partition starts from sector 32768.
    For 64GB, there are total of 120913920 sectors, each sector 512 bytes. */

status = fx_media_exFAT_format(&sd_card, _fx_sd_driver,
                                driver_information, media_memory,
                                sizeof(media_memory),
                                "exFAT_DISK" /* Volume Name */,
                                1 /* Number of FATs */,
                                32768 /* Hidden sectors */,
                                120913920 /* Total sectors */,
                                512 /* Sector size */,
                                256 /* Sectors per cluster */,
                                12345 /* Volume ID */,
                                8192 /* Boundary unit */);

/* If status is FX_SUCCESS, the media was successfully formatted. */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_extended_space_available

返回可用媒体空间

原型

```
UINT fx_media_extended_space_available(
    FX_MEDIA *media_ptr,
    ULONG64 *available_bytes_ptr);
```

说明

此服务返回媒体中可用的字节数。

此服务是为 exFAT 设计的。指向 available_bytes 参数的指针采用 64 位整数值，使调用方能够使用超过 4GB 范

围的媒体。

输入参数

- **media_ptr**：指向先前打开的媒体的指针。
- **available_bytes_ptr**：媒体中剩余的可用字节数。

返回值

- **FX_SUCCESS** (0x00) 成功检索到媒体上的可用空间。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_PTR_ERROR** (0x18) 媒体指针无效或可用字节指针为 NULL。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        my_media;
ULONG64            available_bytes;
UINT            status;
/* Retrieve the available bytes in the media. */

status = fx_media_extended_space_available(&my_media, &available_bytes);

/* If status equals FX_SUCCESS, the number of available bytes is in "available_bytes." */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_flush

将数据刷新到物理媒体

原型

```
UINT fx_media_flush(FX_MEDIA *media_ptr);
```

说明

此服务将任何已修改文件的所有已缓存扇区和目录条目刷新到物理媒体。

> **WARNING**
>
> 应用程序会定期调用此例程，以降低目标突然断电时文件损坏和/或数据丢失的风险。

输入参数

- **media_ptr**：指向媒体控制块的指针。

返回值

- **FX_SUCCESS** (0x00) 成功刷新媒体。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_PTR_ERROR** (0x18) 媒体指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        my_media;
UINT            status;

/* Flush all cached sectors and modified file entries to the physical media. */

status = fx_media_flush(&my_media);

/* If status equals FX_SUCCESS, the physical media is completely up-to-date. */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available

- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_format

格式化媒体

原型

```
UINT fx_media_format(
    FX_MEDIA *media_ptr,
    VOID (*driver)(FX_MEDIA *media),
    VOID *driver_info_ptr,
    UCHAR *memory_ptr,
    UINT memory_size,
    CHAR *volume_name,
    UINT number_of_fats,
    UINT directory_entries,
    UINT hidden_sectors,
    ULONG total_sectors,
    UINT bytes_per_sector,
    UINT sectors_per_cluster,
    UINT heads,
    UINT sectors_per_track);
```

说明

此服务基于提供的参数以 FAT 12/16/32 兼容的方式格式化提供的媒体。在打开媒体之前，必须调用此服务。

> **WARNING**
> 格式化已有格式的媒体可有效地清除媒体上的所有文件和目录。

输入参数

- **media_ptr**：指向媒体控制块的指针。该参数仅用于提供驱动程序运行所需的一些基本信息。
- **driver**：指向此媒体的 I/O 驱动程序的指针。这通常是提供给后续 fx_media_open 调用的同一个驱动程序。
- **driver_info_ptr**：指向 I/O 驱动程序可以使用的可选信息的指针。
- **memory_ptr**：指向媒体的工作内存的指针。
- **memory_size**：指定工作媒体内存的大小。大小必须至少与媒体扇区的大小相同。
- **volume_name**：指向卷名称字符串的指针，最多为 11 个字符。
- **number_of_fats**：媒体中的 FAT 数。对于主 FAT，最小值为 1。大于 1 的值会导致在运行时保留其他 FAT 副本。
- **directory_entries**：根目录中目录条目的数目。
- **hidden_sectors**：在此媒体启动扇区前隐藏的扇区数。如果存在多个分区，则通常会出现这种情况。
- **total_sectors**：媒体中的扇区总数。
- **bytes_per_sector**：每个扇区的字节数，通常为 512。FileX 要求此数为 32 的倍数。
- **sectors_per_cluster**：每个群集中的扇区数。群集是 FAT 文件系统中的最小分配单元。
- **heads**：物理磁头的数目。
- **sectors_per_track**：每个磁道的扇区数。

返回值

- **FX_SUCCESS** (0x00) 成功格式化媒体。

- FX_IO_ERROR (0x90) 驱动程序 I/O 错误。
- FX_PTR_ERROR (0x18) 媒体、驱动程序或内存指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA         ram_disk;
UCHAR             media_memory[512];
UCHAR             ram_disk_memory[32768];

/* Format a RAM disk with 32768 bytes and 512 bytes per sector. */

status = fx_media_format(&ram_disk, _fx_ram_driver,
                         ram_disk_memory, media_memory,
                         sizeof(media_memory),
                         "MY_RAM_DISK" /* Volume Name */,
                         1 /* Number of FATs */,
                         32 /* Directory Entries */,
                         0 /* Hidden sectors */,
                         64 /* Total sectors */,
                         512 /* Sector size */,
                         1 /* Sectors per cluster */,
                         1 /* Heads */,
                         1 /* Sectors per track */);

/* If status is FX_SUCCESS, the media was successfully formatted
    and can now be opened with the following call: */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_open

打开媒体以进行文件访问

原型

```
UINT fx_media_open(
    FX_MEDIA *media_ptr,
    CHAR *media_name,
    VOID(*media_driver)(FX_MEDIA *),
    VOID *driver_info_ptr,
    VOID *memory_ptr,
    ULONG memory_size);
```

说明

此服务使用提供的 I/O 驱动程序打开媒体以进行文件访问。

> **WARNING**
>
> 提供给此服务的内存用于实现内部逻辑扇区缓存，因此，提供的内存越多，减少的物理 I/O 就越多。FileX 需要至少一个逻辑扇区（每个媒体扇区的字节数）的缓存。

输入参数

- media_ptr：指向媒体控制块的指针。
- media_name：指向媒体名称的指针。
- media_driver：指向此媒体的 I/O 驱动程序的指针。I/O 驱动程序必须符合第 5 章中定义的 FileX 驱动程序要求。
- driver_info_ptr：指向提供的 I/O 驱动程序可以使用的可选信息的指针。
- memory_ptr：指向媒体的工作内存的指针。
- memory_size：指定工作媒体内存的大小。大小必须和媒体的扇区大小（通常为 512 个字节）一样。

返回值

- FX_SUCCESS (0x00) 成功打开媒体。
- FX_BOOT_ERROR (0x01) 读取媒体的启动扇区时出错。
- FX_MEDIA_INVALID (0x02) 指定媒体的启动扇区已损坏或无效。此外，此返回代码用于指示逻辑扇区缓存大小或 FAT 条目大小不是 2 的幂。
- FX_FAT_READ_ERROR (0x03) 读取媒体 FAT 时出错。
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误。
- FX_PTR_ERROR (0x18) 一个或多个指针为 NULL。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA         ram_disk,
UINT             status;
UCHAR             buffer[128];
/* Open a 32KByte RAM disk starting at the fixed address of 0x800000.
    Note that the total 32KByte media size and 128-byte sector size is defined inside of the driver. */

status = fx_media_open(&ram_disk, "RAM DISK", fx_ram_driver, 0, &buffer[0], sizeof(buffer));

/* If status equals FX_SUCCESS, the RAM disk has been successfully setup and is ready for file access! */
```

另请参阅

- fx_fault_tolerant_enable

- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_open_notify_set

设置媒体打开通知函数

原型

```
UINT fx_media_open_notify_set(
    FX_MEDIA *media_ptr,
    VOID (*media_open_notify)(FX_MEDIA*));
```

说明

此服务设置一个通知回调函数，该函数将在成功打开媒体后被调用。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **media_open_notify**：要安装的媒体打开通知回调函数。传递 NULL 作为回调函数将禁用媒体打开回调。

返回值

- **FX_SUCCESS** (0x00) 已成功安装回调函数。
- **FX_PTR_ERROR** (0x18) media_ptr 为 NULL。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
fx_media_open_notify_set(media_ptr, my_media_open_callback);
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate

- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_read

读取媒体的逻辑扇区

原型

```
UINT fx_media_read(
    FX_MEDIA *media_ptr,
    ULONG logical_sector,
    VOID *buffer_ptr);
```

说明

此服务读取媒体的逻辑扇区，并将其放入所提供的缓冲区。

输入参数

- **media_ptr**：指向先前打开的媒体的指针。
- **logical_sector**：要读取的逻辑扇区。
- **buffer_ptr**：指向逻辑扇区读取的目标的指针。

返回值

- **FX_SUCCESS** (0x00) 成功读取媒体。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_PTR_ERROR** (0x18) 媒体或缓冲区指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA         my_media;
UCHAR             my_buffer[128];
UINT            STATUS;
/* Read logical sector 22 into "my_buffer" assuming the
    physical media has a sector size of 128. */
status = fx_media_read(&my_media, 22, my_buffer);
/* If status equals FX_SUCCESS, the contents of logical sector 22 are in "my_buffer". */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_space_available

返回可用媒体空间

原型

```
UINT fx_media_space_available(
    FX_MEDIA *media_ptr,
    ULONG *available_bytes_ptr);
```

说明

此服务返回媒体中可用的字节数。

若要处理大于 4GB 的媒体，应用程序应使用服务 fx_media_extended_space_available。

输入参数

- **media_ptr**：指向先前打开的媒体的指针。
- **available_bytes_ptr**：媒体中剩余的可用字节数。

返回值

- **FX_SUCCESS** (0x00) 成功返回媒体上的可用空间。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_PTR_ERROR** (0x18) 媒体指针无效或可用字节指针为 NULL。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        my_media;
ULONG            available_bytes;
UINT            status;
/* Retrieve the available bytes in the media. */

status = fx_media_space_available(&my_media, &available_bytes);

/* If status equals FX_SUCCESS, the number of available bytes is in "available_bytes." */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_volume_get
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_volume_get

获取媒体卷名称

原型

```
UINT fx_media_volume_get(
    FX_MEDIA *media_ptr,
    CHAR *volume_name,
    UINT volume_source);
```

说明

此服务检索之前打开的媒体的卷名称。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **volume_name**：指向卷名称的目标的指针。请注意，目标的大小必须至少足以容纳 12 个字符。
- **volume_source**：指定从启动扇区或根目录检索名称的位置。此参数的有效值是：
  - FX_BOOT_SECTOR

- FX_DIRECTORY_SECTOR

返回值

- **FX_SUCCESS** (0x00) 成功获取媒体卷。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_NOT_FOUND** (0x04) 未找到卷。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_PTR_ERROR** (0x18) 媒体或卷目标指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        ram_disk;
UCHAR             volume_name[12];
/* Retrieve the volume name of the RAM disk, from the boot sector. */

status = fx_media_volume_get_extended(&ram_disk, volume_name,
                                      sizeof(volume_name), FX_BOOT_SECTOR);

/* If status is FX_SUCCESS, the volume name was successfully retrieved. */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_volume_get_extended

获取之前打开的媒体的媒体卷名称

原型

```
UINT fx_media_volume_get_extended(
    FX_MEDIA *media_ptr,
    CHAR *volume_name,
    UINT volume_name_buffer_length,
    UINT volume_source);
```

说明

此服务检索之前打开的媒体的卷名称。

> **IMPORTANT**
> 除了调用方传入 volume_name 缓冲区的大小之外，此服务与 fx_media_volume_get() 相同。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **volume_name**：指向卷名称的目标的指针。请注意，目标的大小必须至少足以容纳 12 个字符。
- **volume_name_buffer_length**：volume_name 缓冲区的大小。
- **volume_source**：指定从启动扇区或根目录检索名称的位置。此参数的有效值是：
  - FX_BOOT_SECTOR
  - FX_DIRECTORY_SECTOR

返回值

- **FX_SUCCESS** (0x00) 成功获取媒体卷。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_NOT_FOUND** (0x04) 未找到卷。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_PTR_ERROR** (0x18) 媒体或卷目标指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA         ram_disk;
UCHAR             volume_name[12];

/* Retrieve the volume name of the RAM disk, from the boot sector. */

status = fx_media_volume_get_extended(&ram_disk, volume_name,
                                      sizeof(volume_name),
                                      FX_BOOT_SECTOR);

/* If status is FX_SUCCESS, the volume name was successfully retrieved. */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_close_notify_set

- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_set
- fx_media_write
- fx_system_initialize

fx_media_volume_set

设置媒体卷名称

原型

```
UINT fx_media_volume_set(
    FX_MEDIA *media_ptr,
    CHAR *volume_name);
```

说明

此服务设置之前打开的媒体的卷名称。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **volume_name**：指向卷名称的指针。

返回值

- **FX_SUCCESS** (0x00) 成功设置媒体卷。
- **FX_INVALID_NAME** (0x0C) Volume_name 无效。
- **FX_MEDIA_INVALID** (0x02) 无法设置卷名。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_PTR_ERROR** (0x18) 媒体或卷名称指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA ram_disk;

/* Set the volume name to "MY_VOLUME". */

status = fx_media_volume_set(&ram_disk, "MY_VOLUME");

/* If status is FX_SUCCESS, the volume name was successfully set. */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_get
- fx_media_write
- fx_system_initialize

fx_media_write

写入逻辑扇区

原型

```
UINT fx_media_write(
    FX_MEDIA *media_ptr,
    ULONG logical_sector,
    VOID *buffer_ptr);
```

说明

此服务将所提供的缓冲区写入指定的逻辑扇区。

输入参数

- **media_ptr**：指向先前打开的媒体的指针。
- **logical_sector**：要写入的逻辑扇区。
- **buffer_ptr**：指向逻辑扇区写入的源的指针。

返回值

- **FX_SUCCESS** (0x00) 成功写入媒体。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_PTR_ERROR** (0x18) 媒体指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA            my_media;
UCHAR                my_buffer[128];
UINT                status;

/* Write logical sector 22 from "my_buffer" assuming
    the physical media has a sector size of 128. */

status = fx_media_write(&my_media, 22, my_buffer);

/* If status equals FX_SUCCESS, the contents of logical
    sector 22 are now the same as "my_buffer." */
```

另请参阅

- fx_fault_tolerant_enable
- fx_media_abort
- fx_media_cache_invalidate
- fx_media_check
- fx_media_close
- fx_media_close_notify_set
- fx_media_exFAT_format
- fx_media_extended_space_available
- fx_media_flush
- fx_media_format
- fx_media_open
- fx_media_open_notify_set
- fx_media_read
- fx_media_space_available
- fx_media_volume_get
- fx_media_volume_set
- fx_system_initialize

fx_system_date_get

获取文件系统日期

原型

```
UINT fx_system_date_get(
    UINT *year,
    UINT *month,
    UINT *day);
```

说明

此服务返回当前系统日期。

输入参数

- **year**：指向年份的目标的指针。
- **month**：指向月份的目标的指针。
- **day**：指向日期的目标的指针。

返回值

- **FX_SUCCESS** (0x00) 成功检索日期。

- FX_PTR_ERROR (0x18) 一个或多个输入参数为 NULL。

允许来自

线程数

示例

```
UINT            status;
UINT            year;
UINT            month;
UINT            day;
/* Retrieve the current system date. */

status = fx_system_date_get(&year, &month, &day);

/* If status equals FX_SUCCESS, the year, month,
    and day parameters now have meaningful information. */
```

另请参阅

- fx_system_date_set
- fx_system_initialize
- fx_system_time_get
- fx_system_time_set

fx_system_date_set

设置系统日期

原型

```
UINT fx_system_date_set(
    UINT year,
    UINT month,
    UINT day);
```

说明

此服务根据指定设置系统日期。

> **WARNING**
> 应在 fx_system_initialize 后不久调用此服务以设置初始系统日期。默认情况下，系统日期是最后一个通用 FileX 版本的日期。

输入参数

- **year**：新年份。有效范围是从 1980 到 2107 年。
- **month**：新月份。有效范围是从 1 到 12。
- **day**：新日期。有效范围是从 1 到 31，具体取决于月份和闰年条件。

返回值

- FX_SUCCESS (0x00) 成功设置日期。
- FX_INVALID_YEAR (0x12) 指定的年份无效。
- FX_INVALID_MONTH (0x13) 指定的月份无效。
- FX_INVALID_DAY (0x14) 指定的日期无效。

允许来自

初始化、线程

示例

```
 UINT            status;

/* Set the system date to December 12, 2005. */

status = fx_system_date_set(2005, 12, 12);

/* If status equals FX_SUCCESS, the file system date is now
    12-12-2005. */
```

另请参阅

- fx_system_date_get
- fx_system_initialize
- fx_system_time_get
- fx_system_time_set

fx_system_initialize

初始化整个系统

原型

```
VOID fx_system_initialize(void);
```

说明

此服务初始化所有主要的 FileX 数据结构。应在 tx_application_define 中或从初始化线程中调用该方法，并且必须在使用任何其他 FileX 服务之前调用它。

> **WARNING**
>
> 应用程序一旦被此调用初始化，其应调用 fx_system_date_set 和 fx_system_time_set 从准确的系统日期和时间开始。

输入参数

无

返回值

无。

允许来自

初始化、线程

示例

```
void tx_application_define(VOID *free_memory)
{
    UINT status;
    /* Initialize the FileX system. */
    fx_system_initialize();
    /* Set the file system date. */
    fx_system_date_set(my_year, my_month, my_day);

    /* Set the file system time. */
    fx_system_time_set(my_hour, my_minute, my_second);

    /* Now perform all other initialization and possibly FileX media
        open calls if the corresponding driver does not block on the boot sector read. */

    ...

}
```

另请参阅

- fx_system_date_get
- fx_system_date_set
- fx_system_time_get
- fx_system_time_set

fx_system_time_get

获取当前系统时间

原型

```
UINT fx_system_time_get(
    UINT *hour,
    UINT *minute,
    UINT *second);
```

说明

此服务检索当前系统时间。

输入参数

- **hour**：指向小时的目标的指针。
- **minute**：指向分钟的目标的指针。
- **second**：指向秒的目标的指针。

返回值

- FX_SUCCESS (0x00) 成功检索系统时间。
- FX_PTR_ERROR (0x18) 一个或多个输入参数

允许来自

线程数

示例

```
UINT            status;
UINT            hour;
UINT            minute;
UINT            second;

/* Retrieve the current system time. */

status = fx_system_time_get(&hour, &minute, &second);

/* If status equals FX_SUCCESS, the current system time
    is in the hour, minute, and second variables. */
```

另请参阅

- fx_system_date_get
- fx_system_date_set
- fx_system_initialize
- fx_system_time_set

fx_system_time_set

设置当前系统时间

原型

```
UINT fx_system_time_set(UINT hour, UINT minute, UINT second);
```

说明

此服务将当前系统时间设置为输入参数指定的时间。

> **WARNING**
>
> 应在 fx_system_initialize 后不久调用此服务以设置初始系统时间。默认情况下，系统时间为 0:0:0。

输入参数

- **hour**：新小时 (0-23)。
- **minute**：新分钟 (0-59)。
- **second**：新秒 (0-59)。

返回值

- **FX_SUCCESS** (0x00) 成功检索系统时间。
- **FX_INVALID_HOUR** (0x15) 新小时无效。
- **FX_INVALID_MINUTE** (0x16) 新分钟无效。
- **FX_INVALID_SECOND** (0x17) 新秒无效。

允许来自

线程数

示例

```
 UINT status;

/* Set the current system time to hour 23, minute 21, and second 20. */

status = fx_system_time_set(23, 21, 20);

/* If status is FX_SUCCESS, the current system time has been set. */
```

另请参阅

- fx_system_date_get
- fx_system_date_set
- fx_system_initialize
- fx_system_time_get

fx_unicode_directory_create

创建 Unicode 目录

原型

```
UINT fx_unicode_directory_create(
    FX_MEDIA *media_ptr,
    UCHAR *source_unicode_name,
    ULONG source_unicode_length,
    CHAR *short_name);
```

说明

此服务在当前默认目录中创建一个 Unicode 命名的子目录，这是因为 Unicode 源名称参数中不允许有路径信息。如果成功，该服务将返回新创建的 Unicode 子目录的短名称(8.3 格式)。

> **WARNING**
>
> Unicode 子目录上的所有操作(使其成为默认路径、删除等)应通过将返回的短名称(8.3 格式)提供给标准 FileX 目录服务来完成。

> **IMPORTANT**
>
> exFAT 媒体不支持此服务。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **source_unicode_name**：指向新子目录的 Unicode 名称的指针。
- **source_unicode_length**：Unicode 名称的长度。
- **short_name**：指向新 Unicode 子目录的短名称(8.3 格式)的目标的指针。

返回值

- FX_SUCCESS (0x00) 成功创建 Unicode 目录。
- FX_MEDIA_NOT_OPEN (0x11) 指定的媒体未打开。
- FX_ALREADY_CREATED (0x0B) 已存在指定的目录。
- FX_NO_MORE_SPACE (0x0A) 新目录条目的媒体中不再有可用的群集。
- FX_NOT_IMPLEMENTED (0x22) 未为 exFAT 文件系统实现服务。

- FX_WRITE_PROTECT (0x23) 指定的媒体受到写入保护。
- FX_PTR_ERROR (0x18) 媒体或名称指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误。

允许来自

线程数

示例

```
FX_MEDIA             ram_disk;
UCHAR                 my_short_name[13];
UCHAR                 my_unicode_name[] = {0x38,0xC1, 0x88,0xBC, 0xF8,0xC9, 0x20,0x00,
                                           0x54,0xD6, 0x7C,0xC7, 0x20,0x00, 0x74,0xC7,
                                           0x84,0xB9, 0x20,0x00, 0x85,0xC7, 0xC8,0xB2,
                                           0xE4,0xB2, 0x2E,0x00, 0x64,0x00, 0x6F,0x00,
                                           0x63,0x00, 0x00,0x00};

/* Create a Unicode subdirectory with the name contained in "my_unicode_name". */

length = fx_unicode_directory_create(&ram_disk, my_unicode_name, 17, my_short_name);

/* If successful, the Unicode subdirectory is created and "my_short_name"
    contains the 8.3 format name that can be used with other FileX services. */
```

另请参阅

- fx_directory_attributes_read
- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_rename

fx_unicode_directory_rename

使用 Unicode 字符串重命名目录

原型

```
UINT fx_unicode_directory_rename(
    FX_MEDIA *media_ptr,
    UCHAR *old_unicode_name,
    ULONG old_unicode_length,
    UCHAR *new_unicode_name,
    ULONG new_unicode_length,
    CHAR *new_short_name);
```

说明

此服务会将 Unicode 命名的子目录更改为当前工作目录中的指定新 Unicode 名称。Unicode 名称参数不能包含路径信息。

> **IMPORTANT**
>
> exFAT 媒体不支持此服务。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **old_unicode_name**：指向当前文件的 Unicode 名称的指针。
- **old_unicode_name_length**：当前 Unicode 名称的长度。
- **new_unicode_name**：指向新 Unicode 文件名的指针。
- **old_unicode_name_length**：新 Unicode 名称的长度。
- **new_short_name**：指向重命名 Unicode 文件的短名称（8.3 格式）的目标的指针。使用 Unicode 重命名目录

返回值

- **FX_SUCCESS** (0x00) 成功打开媒体。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_ALREADY_CREATED** (0x0B) 已存在指定的目录名称。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_PTR_ERROR** (0x18) 一个或多个指针为 NULL。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。

允许来自

线程数

示例

```
FX_MEDIA             my_media;
UINT                 status;
UCHAR                 my_short_name[13];
UCHAR                 my_old_unicode_name[] = {'a', '\0', 'b', '\0', 'c', '\0', '\0', '\0'};
UCHAR                 my_new_unicode_name[] = {'d', '\0', 'e', '\0', 'f', '\0', '\0', '\0'};

/* Change the Unicode-named file "abc" to "def". */

status = fx_unicode_directory_rename(&my_media, my_old_unicode_name,
    3, my_new_unicode_name, 3, my_short_name);

/* If status equals FX_SUCCESS, the directory was changed to "def" and
    "my_short_name" contains the 8.3 format name that can be used with other FileX services. */
```

另请参阅

- fx_directory_attributes_read

- fx_directory_attributes_set
- fx_directory_create
- fx_directory_default_get
- fx_directory_default_set
- fx_directory_delete
- fx_directory_first_entry_find
- fx_directory_first_full_entry_find
- fx_directory_information_get
- fx_directory_local_path_clear
- fx_directory_local_path_get
- fx_directory_local_path_restore
- fx_directory_local_path_set
- fx_directory_long_name_get
- fx_directory_name_test
- fx_directory_next_entry_find
- fx_directory_next_full_entry_find
- fx_directory_rename
- fx_directory_short_name_get
- fx_unicode_directory_create

fx_unicode_file_create

创建 Unicode 文件

原型

```
UINT fx_unicode_file_create(
    FX_MEDIA *media_ptr,
    UCHAR *source_unicode_name,
    ULONG source_unicode_length,
    CHAR *short_name);
```

说明

此服务在当前默认目录中创建一个 Unicode 命名的文件，Unicode 源名称参数中不允许有路径信息。如果成功，该服务将返回新创建的 Unicode 文件的短名称（8.3 格式）。

> **WARNING**
>
> Unicode 文件上的所有操作（打开、写入、读取、关闭等）应通过将返回的短名称（8.3 格式）提供给标准 FileX 文件服务来完成。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **source_unicode_name**：指向新文件的 Unicode 名称的指针。
- **source_unicode_length**：Unicode 名称的长度。
- **short_name**：指向新 Unicode 文件的短名称（8.3 格式）的目标的指针。

返回值

- **FX_SUCCESS** (0x00) 成功创建文件。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。

- **FX_ALREADY_CREATED** (0x0B) 已存在指定的文件。
- **FX_NO_MORE_SPACE** (0x0A) 新文件条目的媒体中不再有可用的群集。
- **FX_NOT_IMPLEMENTED** (0x22) 未为 exFAT 文件系统实现服务。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。
- **FX_PTR_ERROR** (0x18) 媒体或名称指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA         ram_disk;
UCHAR             my_short_name[13];
UCHAR             my_unicode_name[] = {0x38,0xC1, 0x88,0xBC, 0xF8,0xC9, 0x20,0x00,
                                     0x54,0xD6, 0x7C,0xC7, 0x20,0x00, 0x74,0xC7,
                                     0x84,0xB9, 0x20,0x00, 0x85,0xC7, 0xC8,0xB2,
                                     0xE4,0xB2, 0x2E,0x00, 0x64,0x00, 0x6F,0x00,
                                     0x63,0x00, 0x00,0x00};

/* Create a Unicode file with the name contained in "my_unicode_name". */

length = fx_unicode_file_create(&ram_disk, my_unicode_name, 17, my_short_name);

/* If successful, the Unicode file is created and "my_short_name"
    contains the 8.3 format name that can be used with other FileX services. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set

- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_unicode_file_rename

使用 unicode 字符串重命名文件

原型

```
UINT fx_unicode_file_rename(
    FX_MEDIA *media_ptr,
    UCHAR *old_unicode_name,
    ULONG old_unicode_length,
    UCHAR *new_unicode_name,
    ULONG new_unicode_length,
    CHAR *new_short_name);
```

说明

此服务会将 Unicode 命名的文件名称更改为当前默认目录中的指定新 Unicode 名称。Unicode 名称参数不能包含路径信息。

> **IMPORTANT**
> exFAT 媒体不支持此服务。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **old_unicode_name**：指向当前文件的 Unicode 名称的指针。
- **old_unicode_name_length**：当前 Unicode 名称的长度。
- **new_unicode_name**：指向新 Unicode 文件名的指针。
- **new_unicode_name_length**：新 Unicode 名称的长度。
- **new_short_name**：指向重命名 Unicode 文件的短名称（8.3 格式）的目标的指针。

返回值

- **FX_SUCCESS** (0x00) 成功打开媒体。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_ALREADY_CREATED** (0x0B) 已存在指定的文件名称。
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_PTR_ERROR** (0x18) 一个或多个指针为 NULL。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。
- **FX_WRITE_PROTECT** (0x23) 指定的媒体受到写入保护。

允许来自

线程数

示例

```
FX_MEDIA            my_media;
UINT                status;
UCHAR                my_short_name[13];
UCHAR                my_old_unicode_name[] = {'a', '\0', 'b', '\0', 'c', '\0', '\0', '\0'};
UCHAR                my_new_unicode_name[] = {'d', '\0', 'e', '\0', 'f', '\0', '\0', '\0'};

/* Change the Unicode-named file "abc" to "def". */

status = fx_unicode_file_rename(&my_media, my_old_unicode_name,
    3, my_new_unicode_name, 3, my_short_name);

/* If status equals FX_SUCCESS, the file name was changed to "def" and
    "my_short_name" contains the 8.3 format name that can be used with other FileX services. */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_unicode_length_get

获取 Unicode 名称的长度

原型

```
ULONG fx_unicode_length_get(UCHAR *unicode_name);
```

说明

此服务确定所提供的 Unicode 名称的长度。Unicode 字符由两个字节表示。Unicode 名称是一系列由两个 NULL 字节（两个 0 值字节）终止的两字节的 Unicode 字符。

输入参数

unicode_name：指向 Unicode 名称的指针。

返回值

length：Unicode 名称的长度（名称中的 Unicode 字符数）。

允许来自

线程数

示例

```
UCHAR     my_unicode_name[] = {0x38,0xC1, 0x88,0xBC, 0xF8,0xC9, 0x20,0x00,
                                0x54,0xD6, 0x7C,0xC7, 0x20,0x00, 0x74,0xC7,
                                0x84,0xB9, 0x20,0x00, 0x85,0xC7, 0xC8,0xB2,
                                0xE4,0xB2, 0x2E,0x00, 0x64,0x00, 0x6F,0x00,
                                0x63,0x00, 0x00,0x00};

UINT     length;

/* Get the length of "my_unicode_name". */

length = fx_unicode_length_get(my_unicode_name);

/* A value of 17 will be returned for the length of the "my_unicode_name". */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create

- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_unicode_length_get_extended

获取 Unicode 名称的长度

原型

```
UINT fx_unicode_length_get_extended(
    UCHAR *unicode_name,
    UINT buffer_length);
```

说明

此服务获取所提供的 Unicode 名称的长度。Unicode 字符由两个字节表示。Unicode 名称是一系列由两个 NULL 字节（两个 0 值字节）终止的两字节的 Unicode 字符。

> **IMPORTANT**
> 除了调用方传入 unicode_name 缓冲区的大小（包括两个 NULL 字符）之外，此服务与 fx_unicode_length_get() 相同 。

输入参数

- **unicode_name**：指向 Unicode 名称的指针。
- **buffer_length**：Unicode 名称缓冲区的大小。

返回值

length：Unicode 名称的长度（名称中的 Unicode 字符数）。

允许来自

线程数

示例

```
UCHAR        my_unicode_name[] = {0x38,0xC1, 0x88,0xBC, 0xF8,0xC9, 0x20,0x00,
                                  0x54,0xD6, 0x7C,0xC7, 0x20,0x00, 0x74,0xC7,
                                  0x84,0xB9, 0x20,0x00, 0x85,0xC7, 0xC8,0xB2,
                                  0xE4,0xB2, 0x2E,0x00, 0x64,0x00, 0x6F,0x00,
                                  0x63,0x00, 0x00,0x00};

UINT         length;

/* Get the length of "my_unicode_name". */

length = fx_unicode_length_get_extended(my_unicode_name, sizeof(my_unicode_name));

/* A value of 17 will be returned for the length of the "my_unicode_name". */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create

- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

fx_unicode_name_get

从短名称获取 Unicode 名称

原型

```
UINT fx_unicode_name_get(
    FX_MEDIA *media_ptr,
    CHAR *source_short_name,
    UCHAR *destination_unicode_name,
    ULONG *destination_unicode_length);
```

说明

此服务检索与当前默认目录中的提供的短名称(8.3 格式)相关联的 Unicode 名称，短名称参数中不允许存在路径信息。如果成功，该服务将返回与短名称关联的 Unicode 名称。

> **IMPORTANT**
>
> 此服务可用于获取文件和子目录的 Unicode 名称。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **short_name** 指向短名称(8.3 格式)的指针。
- **destination_unicode_name**：指向与所提供的短名称关联的 Unicode 名称的目标的指针。
- **destination_unicode_length**：指向返回的 Unicode 名称长度的指针。

返回值

- **FX_SUCCESS** (0x00) 成功检索 Unicode 名称。

- FX_FAT_READ_ERROR (0x03) 无法读取 FAT 表。
- FX_FILE_CORRUPT (0x08) 文件已损坏
- FX_IO_ERROR (0x90) 驱动程序 I/O 错误。
- FX_MEDIA_NOT_OPEN (0x11) 指定的媒体未打开。
- FX_NOT_FOUND (0x04) 未找到短名称，或 Unicode 目标大小太小。
- FX_SECTOR_INVALID (0x89) 扇区无效。
- FX_PTR_ERROR (0x18) 媒体或名称指针无效。
- FX_CALLER_ERROR (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA            ram_disk;
UCHAR                my_unicode_name[256*2];
ULONG                unicode_name_length;

/* Get the Unicode name associated with the short name "ABC0~111.TXT".
    Note that the Unicode destination must have 2 times the
    number of maximum characters in the name. */

length = fx_unicode_name_get(&ram_disk, "ABC0~111.TXT", my_unicode_name, unicode_name_length);

/* If successful, the Unicode name is returned in "my_unicode_name". */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create

- fx_unicode_file_rename
- fx_unicode_short_name_get

fx_unicode_name_get_extended

从短名称获取 Unicode 名称

原型

```
UINT fx_unicode_name_get_extended(
    FX_MEDIA *media_ptr,
    CHAR *source_short_name,
    UCHAR *destination_unicode_name,
    ULONG *destination_unicode_length,
    ULONG unicode_name_buffer_length);
```

说明

此服务检索与当前默认目录中的提供的短名称(8.3 格式)相关联的 Unicode 名称，短名称参数中不允许存在路径信息。如果成功，该服务将返回与短名称关联的 Unicode 名称。

> **IMPORTANT**
>
> 除了调用方提供目标 Unicode 缓冲区的大小作为输出参数之外，此服务与 fx_unicode_name_get 相同。这使服务可以保证它不会覆盖目标 Unicode 缓冲区

> **IMPORTANT**
>
> 此服务可用于获取文件和子目录的 Unicode 名称。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **short_name**：指向短名称(8.3 格式)的指针。
- **destination_unicode_name**：指向与所提供的短名称关联的 Unicode 名称的目标的指针。
- **destination_unicode_length**：指向返回的 Unicode 名称长度的指针。
- **unicode_name_buffer_length**：Unicode 名称缓冲区的大小。注意：需要 NULL 终止符，这会产生额外的字节。

返回值

- **FX_SUCCESS** (0x00) 成功检索 Unicode 名称。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 表。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_NOT_FOUND** (0x04) 未找到短名称，或 Unicode 目标大小太小。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_PTR_ERROR** (0x18) 媒体或名称指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA        ram_disk;
UCHAR            my_unicode_name[256*2];
ULONG            unicode_name_length;

/* Get the Unicode name associated with the short name "ABC0~111.TXT".
    Note that the Unicode destination must have 2 times the number of maximum characters in the name. */

length = fx_unicode_name_get_extended(&ram_disk, "ABC0~111.TXT",
    my_unicode_name,&unicode_name_length, sizeof(ny_unicode_name));

/* If successful, the Unicode name is returned in "my_unicode_name". */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_short_name_get

fx_unicode_short_name_get

从 Unicode 名称获取短名称

原型

```
UINT fx_unicode_short_name_get(
    FX_MEDIA *media_ptr,
    UCHAR *source_unicode_name,
    ULONG source_unicode_length,
    CHAR *destination_short_name);
```

此服务检索与当前默认目录中的 Unicode 名称关联的短名称（8.3 格式），Unicode 名称参数中不允许存在路径信息。如果成功，该服务将返回与 Unicode 名称关联的短名称。

> **IMPORTANT**
>
> 此服务可用于获取文件和子目录的短名称。

输入参数

- **media_ptr**：指向媒体控制块的指针。
- **source_unicode_name**：指向 Unicode 名称的指针。
- **source_unicode_length**：Unicode 名称的长度。
- **destination_short_name**：指向短名称（8.3 格式）的目标的指针。它的大小必须至少为 13 个字节。

返回值

- **FX_SUCCESS** (0x00) 成功检索短名称。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 表。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_NOT_FOUND** (0x04) 未找到 Unicode 名称。
- **FX_NOT_IMPLEMENTED** (0x22) 未为 exFAT 文件系统实现服务。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_PTR_ERROR** (0x18) 媒体或名称指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
FX_MEDIA         ram_disk;
UCHAR             my_short_name[13];
UCHAR             my_unicode_name[] = {0x38,0xC1, 0x88,0xBC, 0xF8,0xC9, 0x20,0x00,
                                      0x54,0xD6, 0x7C,0xC7, 0x20,0x00, 0x74,0xC7,
                                      0x84,0xB9, 0x20,0x00, 0x85,0xC7, 0xC8,0xB2,
                                      0xE4,0xB2, 0x2E,0x00, 0x64,0x00, 0x6F,0x00,
                                      0x63,0x00, 0x00,0x00};

/* Get the short name associated with the Unicode name contained in the array "my_unicode_name". */

length = fx_unicode_short_name_get(&ram_disk, my_unicode_name, 17, my_short_name);

/* If successful, the short name is returned in "my_short_name". */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete

- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek
- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get

fx_unicode_short_name_get_extended

从 Unicode 名称获取短名称

原型

```
UINT fx_unicode_short_name_get_extended(
    FX_MEDIA *media_ptr,
    UCHAR *source_unicode_name,
    ULONG source_unicode_length,
    CHAR *destination_short_name,
    ULONG short_name_buffer_length);
```

说明

此服务检索与当前默认目录中的 Unicode 名称关联的短名称(8.3 格式),Unicode 名称参数中不允许存在路径信息。如果成功,该服务将返回与 Unicode 名称关联的短名称。

> **IMPORTANT**
>
> 除了调用方提供目标缓冲区的大小作为输出参数之外,此服务与 fx_unicode_short_name_get() 相同。这使服务可以保证短名称不会超出目标缓冲区。

此服务可用于获取文件和子目录的短名称

输入参数

- **media_ptr**:指向媒体控制块的指针。
- **source_unicode_name**:指向 Unicode 名称的指针。
- **source_unicode_length**:Unicode 名称的长度。
- **destination_short_name**:指向短名称(8.3 格式)的目标的指针。它的大小必须至少为 13 个字节。
- **short_name_buffer_length**:目标缓冲区的大小。缓冲区大小必须至少为 14 字节。

返回值

- **FX_SUCCESS** (0x00) 成功检索短名称。
- **FX_FAT_READ_ERROR** (0x03) 无法读取 FAT 表。
- **FX_FILE_CORRUPT** (0x08) 文件已损坏
- **FX_IO_ERROR** (0x90) 驱动程序 I/O 错误。
- **FX_MEDIA_NOT_OPEN** (0x11) 指定的媒体未打开。
- **FX_NOT_FOUND** (0x04) 未找到 Unicode 名称。
- **FX_NOT_IMPLEMENTED** (0x22) 未为 exFAT 文件系统实现服务。
- **FX_SECTOR_INVALID** (0x89) 扇区无效。
- **FX_PTR_ERROR** (0x18) 媒体或名称指针无效。
- **FX_CALLER_ERROR** (0x20) 调用方不是线程。

允许来自

线程数

示例

```
#define         SHORT_NAME_BUFFER_SIZE 13
FX_MEDIA         ram_disk;
UCHAR             my_short_name[SHORT_NAME_BUFFER_SIZE];
UCHAR             my_unicode_name[] = {0x38,0xC1, 0x88,0xBC, 0xF8,0xC9, 0x20,0x00,
                                       0x54,0xD6, 0x7C,0xC7, 0x20,0x00, 0x74,0xC7,
                                       0x84,0xB9, 0x20,0x00, 0x85,0xC7, 0xC8,0xB2,
                                       0xE4,0xB2, 0x2E,0x00, 0x64,0x00, 0x6F,0x00,
                                       0x63,0x00, 0x00,0x00};

/* Get the short name associated with the Unicode name contained in the array "my_unicode_name". */

length = fx_unicode_short_name_get_extended(&ram_disk,
    my_unicode_name, 17, my_short_name,SHORT_NAME_BUFFER_SIZE);

/* If successful, the short name is returned in "my_short_name". */
```

另请参阅

- fx_file_allocate
- fx_file_attributes_read
- fx_file_attributes_set
- fx_file_best_effort_allocate
- fx_file_close
- fx_file_create
- fx_file_date_time_set
- fx_file_delete
- fx_file_extended_allocate
- fx_file_extended_best_effort_allocate
- fx_file_extended_relative_seek
- fx_file_extended_seek
- fx_file_extended_truncate
- fx_file_extended_truncate_release
- fx_file_open
- fx_file_read
- fx_file_relative_seek
- fx_file_rename
- fx_file_seek

- fx_file_truncate
- fx_file_truncate_release
- fx_file_write
- fx_file_write_notify_set
- fx_unicode_file_create
- fx_unicode_file_rename
- fx_unicode_name_get
- fx_unicode_short_name_get

第 5 章 - Azure RTOS FileX 的 I/O 驱动程序

2021/4/30 •

本章介绍 Azure RTOS FileX 的 I/O 驱动程序，旨在帮助开发人员编写应用程序特定的驱动程序。

I/O 驱动程序简介

FileX 支持多个媒体设备。FX_MEDIA 结构定义管理媒体设备所需的一切。此结构包含所有媒体信息，其中包括用于在驱动程序与 FileX 之间传递信息和状态的媒体特定 I/O 驱动程序及关联参数。在大多数系统中，每个 FileX 媒体实例都有唯一的 I/O 驱动程序。

I/O 驱动程序入口

每个 FileX I/O 驱动程序具有单个入口函数，该函数由 fx_media_open 服务调用定义。驱动程序入口函数的格式如下：

```
void my_driver_entry(FX_MEDIA *media_ptr);
```

FileX 调用 I/O 驱动程序入口函数来请求所有物理媒体访问权限，包括初始化和启动扇区读取权限。向驱动程序发出的请求是按顺序进行的；即，FileX 会先等待当前请求完成，然后再发送另一个请求。

I/O 驱动程序请求

由于每个 I/O 驱动程序都具有单个入口函数，因此 FileX 会通过媒体控制块发出特定的请求。具体而言，FX_MEDIA 的 fx_media_driver_request 成员用于指定确切的驱动程序请求 。I/O 驱动程序通过 FX_MEDIA 的 fx_media_driver_status 成员来传达请求的成功或失败结果 。如果驱动程序请求成功，则在驱动程序返回之前，会将 FX_SUCCESS 放入此字段。否则，如果检测到错误，则将 FX_IO_ERROR 放入此字段。

驱动程序初始化

尽管实际的驱动程序初始化处理特定于应用程序，但这种处理通常包括数据结构初始化，并可能包括某种预备性的硬件初始化。此请求首先由 FileX 发出，在 fx_media_open 服务内部完成。

如果检测到媒体写入保护，则应将 FX_MEDIA 的驱动程序 fx_media_driver_write_protect 成员设置为 FX_TRUE。

以下 FX_MEDIA 成员用于 I/O 驱动程序初始化请求：

FX_MEDIA [illegible]	[illegible]
fx_media_driver_request	FX_DRIVER_INIT

FileX 提供一种机制，用于在扇区不再使用时通知应用程序驱动程序。对于必须管理已映射到闪存的所有使用中逻辑扇区的闪存管理器，此机制特别有用。

如果需要此类可用扇区通知，应用程序驱动程序只需将关联的 FX_MEDIA 结构中的 fx_media_driver_free_sector_update 字段设置为 FX_TRUE 即可 。设置后，FileX 会发出 FX_DRIVER_RELEASE_SECTORS I/O 驱动程序调用，指示一个或多个连续扇区何时可用。

启动扇区读取

FileX 发出特定的请求来读取媒体的启动扇区，而不是使用标准读取请求。以下 FX_MEDIA 成员用于 I/O 驱动程序启动扇区读取请求：

FX_MEDIA [illegible]	[illegible]
fx_media_driver_request	FX_DRIVER_BOOT_READ
fx_media_driver_buffer	启动扇区目标的地址。

启动扇区写入

FileX 发出特定的请求来写入媒体的启动扇区，而不是使用标准写入请求。以下 FX_MEDIA 成员用于 I/O 驱动程序启动扇区写入请求：

FX_MEDIA [illegible]	[illegible]
fx_media_driver_request	FX_DRIVER_BOOT_WRITE
fx_media_driver_buffer	启动扇区源的地址。

扇区读取

FileX 通过向 I/O 驱动程序发出读取请求，将一个或多个扇区读入内存。以下 FX_MEDIA 成员用于 I/O 驱动程序读取请求：

FX_MEDIA [illegible]	[illegible]
fx_media_driver_request	FX_DRIVER_READ
fx_media_driver_logical_sector	要读取的逻辑扇区
fx_media_driver_sectors	要读取的扇区数
fx_media_driver_buffer	要读取的扇区的目标缓冲区
fx_media_driver_data_sector_read	如果请求了文件数据扇区，则设置为 FX_TRUE。否则，如果请求了系统扇区（FAT 或目录扇区），则设置为 FX_FALSE。
fx_media_driver_sector_type	定义所请求扇区的显式类型，如下所示： FX_FAT_SECTOR (2) FX_DIRECTORY_SECTOR (3) FX_DATA_SECTOR (4)

扇区写入

FileX 通过向 I/O 驱动程序发出写入请求，将一个或多个扇区写入物理媒体。以下 FX_MEDIA 成员用于 I/O 驱动程序写入请求：

FX_MEDIA [illegible]	[illegible]
fx_media_driver_request	FX_DRIVER_WRITE
fx_media_driver_logical_sector	要写入的逻辑扇区
fx_media_driver_sectors	要写入的扇区数
fx_media_driver_buffer	要写入的扇区的源缓冲区

FX_MEDIA [illegible]	[illegible]
fx_media_driver_system_write	如果请求了系统扇区(FAT 或目录扇区)，则设置为 FX_TRUE。否则，如果请求了文件数据扇区，则设置为 FX_FALSE。
fx_media_driver_sector_type	定义所请求扇区的显式类型，如下所示： FX_FAT_SECTOR (2) FX_DIRECTORY_SECTOR (3) FX_DATA_SECTOR (4)。

驱动程序刷新

FileX 通过向 I/O 驱动程序发出刷新请求，将当前位于驱动程序扇区缓存中的所有扇区刷新到物理媒体。当然，如果驱动程序不缓存扇区，则此请求就不需要进行驱动程序处理。以下 FX_MEDIA 成员用于 I/O 驱动程序刷新请求：

FX_MEDIA [illegible]	[illegible]
fx_media_driver_request	FX_DRIVER_FLUSH

驱动程序中止

FileX 通过向 I/O 驱动程序发出中止请求，通知驱动程序中止物理媒体中所有进一步的物理 I/O 活动。在驱动程序重新初始化之前，它不应再次执行任何 I/O。以下 FX_MEDIA 成员用于 I/O 驱动程序中止请求：

FX_MEDIA [illegible]	[illegible]
fx_media_driver_request	FX_DRIVER_ABORT

释放扇区

如果驱动程序以前在初始化期间已选择此设置，则每当有一个或多个连续扇区可用时，FileX 就会通知驱动程序。如果驱动程序实际上是闪存管理器，则可以使用此信息告知闪存管理器不再需要这些扇区。以下 FX_MEDIA 成员用于 I/O 释放扇区请求：

FX_MEDIA [illegible]	[illegible]
fx_media_driver_request	FX_DRIVER_RELEASE_SECTORS
fx_media_driver_logical_sector	可用扇区的起始位置
fx_media_driver_sectors	可用扇区数

驱动程序挂起

由于物理媒体中的 I/O 可能需要一段时间，因此经常需要挂起调用线程。当然，此操作假设基础 I/O 操作的完成是由中断驱动的。如果是这样，则使用 ThreadX 信号灯即可轻松实现线程挂起。开始输入或输出操作后，I/O 驱动程序会在收到其自己的内部 I/O 信号灯(在驱动程序初始化期间创建，初始计数为零)时挂起。在驱动程序 I/O 完成中断处理过程中，将设置相同的 I/O 信号灯，从而唤醒已挂起的线程。

扇区转换

由于 FileX 将媒体视为线性逻辑扇区，因此向 I/O 驱动程序发出的 I/O 请求是对逻辑扇区发出的。驱动程序负责在媒体的逻辑扇区与物理几何结构(这可能包括磁头、磁轨和物理扇区)之间进行转换。对于闪存和 RAM 磁盘媒体，逻辑扇区通常将目录映射到物理扇区。对于任何情况，都可以使用以下典型公式在 I/O 驱动程序中执行逻辑扇区到物理扇区的映射：

```
media_ptr -> fx_media_driver_physical_sector =
    (media_ptr -> fx_media_driver_logical_sector % media_ptr -> fx_media_sectors_per_track) + 1;

media_ptr -> fx_media_driver_physical_head =
    (media_ptr -> fx_media_driver_logical_sector/ media_ptr ->
    fx_media_sectors_per_track) % media_ptr -> fx_media_heads;

media_ptr -> fx_media_driver_physical_track =(media_ptr ->
    fx_media_driver_logical_sector/ (media_ptr -> fx_media_sectors_per_track *
    media_ptr -> fx_media_heads)));
```

请注意，物理扇区从 1 开始，而逻辑扇区从 0 开始。

隐藏的扇区

隐藏的扇区驻留在媒体上的启动记录之前。由于它们实际上超出了 FAT 文件系统布局的范围，因此必须在驱动程序执行的每个逻辑扇区操作中考虑到它们。

媒体写入保护

FileX 驱动程序可以通过在媒体控制块中设置 fx_media_driver_write_protect 字段来启用写入保护。如果发出任何 FileX 调用来试图写入该媒体，则会导致返回错误。

示例 RAM 驱动程序

FileX 演示系统随附了一个小型 RAM 磁盘驱动程序，该驱动程序在文件 fx_ram_driver.c 中定义。该驱动程序假设有 32K 内存空间，并且会为 256 个 128 字节的扇区创建启动记录。此文件提供了一个很好的示例用于演示如何实现应用程序特定的 FileX I/O 驱动程序。

```
/*FUNCTION: _fx_ram_driver
RELEASE: PORTABLE C 5.7
AUTHOR: William E. Lamie, Microsoft, Inc.
DESCRIPTION: This function is the entry point to the generic
    RAM disk driver that is delivered with all versions of FileX.
    The format of the RAM disk is easily modified by calling fx_media_format prior to opening the media.

    This driver also serves as a template for developing FileX drivers
    for actual devices. Simply replace the read/write sector logic
    with calls to read/write from the appropriate physical device.

FileX RAM/FLASH structures look like the following:
Physical Sector             Contents
    0                         Boot record
    1                         FAT Area Start
    +FAT Sectors             Root Directory Start
    +Directory Sectors         Data Sector Start

INPUT: media_ptr Media control block pointer
OUTPUT: None
CALLS:     _fx_utility_memory_copy Copy sector memory
        _fx_utility_16_unsigned_read Read 16-bit unsigned
CALLED BY: FileX System Functions
RELEASE HISTORY:

    DATE         NAME         DESCRIPTION
    12-12-2005     William E.     Lamie Initial Version 5.0
    07-18-2007     William E.     Lamie Modified comment(s),
                                resulting in version 5.1
    03-01-2009     William E.     Lamie Modified comment(s),
                                resulting in version 5.2
    11-01-2015     William E.     Lamie Modified comment(s),
                                resulting in version 5.3
    04-15-2016     William E.     Lamie Modified comment(s),
                                resulting in version 5.4
    04-03-2017     William E.     Lamie Modified comment(s),
```

```
    04-03-2017     William E.     Lamie Modified comment(s),
                                fixed compiler warnings,
                                resulting in version 5.5
    12-01-2018     William E.     Lamie Modified comment(s),
                                checked buffer overflow,
                                resulting in version 5.6
    08-15-2019     William E.     Lamie Modified comment(s),
                                resulting in version 5.7
*/

VOID _fx_ram_driver(FX_MEDIA *media_ptr) { UCHAR *source_buffer;
                                           UCHAR *destination_buffer;
                                           UINT bytes_per_sector;

    /* There are several useful/important pieces of information contained
        in the media structure, some of which are supplied by FileX and
        others are for the driver to setup. The following
        is a summary of the necessary FX_MEDIA structure members:
    FX_MEDIA Member                     Meaning

    fx_media_driver_request             FileX request type.
        Valid requests from FileX are as follows:
        FX_DRIVER_READ
        FX_DRIVER_WRITE
        FX_DRIVER_FLUSH
        FX_DRIVER_ABORT
        FX_DRIVER_INIT
        FX_DRIVER_BOOT_READ
        FX_DRIVER_RELEASE_SECTORS
        FX_DRIVER_BOOT_WRITE FX_DRIVER_UNINIT

    fx_media_driver_status              This value is RETURNED by the driver.
        If the operation is successful, this field should be set to FX_SUCCESS
        for before returning. Otherwise, if an error occurred, this field should be set to FX_IO_ERROR.

    fx_media_driver_buffer              Pointer to buffer to read or write sector data. This is supplied by
FileX.

    fx_media_driver_logical_sector      Logical sector FileX is requesting.
    fx_media_driver_sectors             Number of sectors FileX is requesting.

    The following is a summary of the optional FX_MEDIA structure members: FX_MEDIA Member Meaning

    fx_media_driver_info                Pointer to any additional information or memory.
        This is optional for the driver use and is setup from the fx_media_open call.
        The RAM disk uses this pointer for the RAM disk memory itself.

    fx_media_driver_write_protect       The DRIVER sets this to FX_TRUE when media is write protected.
        This is typically done in initialization, but can be done anytime.

    fx_media_driver_free_sector_update  The DRIVER sets this to FX_TRUE when it needs
        to know when clusters are released. This is important for FLASH wear-leveling drivers.

    fx_media_driver_system_write        FileX sets this flag to FX_TRUE if the sector
        being written is a system sector, e.g., a boot, FAT, or directory sector.
        The driver may choose to use this to initiate error recovery logic for greater fault
        tolerance. fx_media_driver_data_sector_read FileX sets this flag to FX_TRUE
        if the sector(s) being read are file data sectors, i.e., NOT system sectors.

    fx_media_driver_sector_type         FileX sets this variable to the specific type of
        sector being read or written. The following sector types are identified:
            FX_UNKNOWN_SECTOR
            FX_BOOT_SECTOR
            FX_FAT_SECTOR
            FX_DIRECTORY_SECTOR
            FX_DATA_SECTOR */

    /* Process the driver request specified in the media control block. */
```

```
    switch (media_ptr -> fx_media_driver_request){

        case FX_DRIVER_READ: {

            /* Calculate the RAM disk sector offset. Note the RAM disk memory
            is pointed to by the fx_media_driver_info pointer, which is supplied
            by the application in the call to fx_media_open. */

            source_buffer = ((UCHAR *)media_ptr -> fx_media_driver_info) +
                ((media_ptr -> fx_media_driver_logical_sector +
                media_ptr -> fx_media_hidden_sectors) * media_ptr -> fx_media_bytes_per_sector);

            /* Copy the RAM sector into the destination. */

             _fx_utility_memory_copy(source_buffer,
                media_ptr -> fx_media_driver_buffer, media_ptr -> fx_media_driver_sectors *
                media_ptr -> fx_media_bytes_per_sector);

            /* Successful driver request. */

            media_ptr -> fx_media_driver_status = FX_SUCCESS; break; }

        case FX_DRIVER_WRITE: {

            /* Calculate the RAM disk sector offset. Note the RAM disk memory
                is pointed to by the fx_media_driver_info pointer, which is supplied
                by the application in the call to fx_media_open. */

            destination_buffer = ((UCHAR *)media_ptr -> fx_media_driver_info) +
                ((media_ptr -> fx_media_driver_logical_sector +
                media_ptr -> fx_media_hidden_sectors) * media_ptr -> fx_media_bytes_per_sector);

            /* Copy the source to the RAM sector. */

            _fx_utility_memory_copy(media_ptr -> fx_media_driver_buffer,
                destination_buffer, media_ptr -> fx_media_driver_sectors *
                media_ptr -> fx_media_bytes_per_sector);

            /* Successful driver request. */

            media_ptr -> fx_media_driver_status = FX_SUCCESS; break; }

        case FX_DRIVER_FLUSH: {
            /* Return driver success. */
            media_ptr -> fx_media_driver_status = FX_SUCCESS; break; }

        case FX_DRIVER_ABORT: {
            /* Return driver success. */
            media_ptr -> fx_media_driver_status = FX_SUCCESS; break; }

        case FX_DRIVER_INIT: {

            /* FLASH drivers are responsible for setting several fields
                in the media structure, as follows:
                media_ptr -> fx_media_driver_free_sector_update media_ptr ->
                fx_media_driver_write_protect
                The fx_media_driver_free_sector_update flag is used to instruct
                FileX to inform the driver whenever sectors are not being used.
                This is especially useful for FLASH managers so they don't have
                maintain mapping for sectors no longer in use.
                The fx_media_driver_write_protect flag can be set anytime by
                the driver to indicate the media is not writable. Write attempts
                made when this flag is set are returned as errors. */
            /* Perform basic initialization here... since the boot record is going
                to be read subsequently and again for volume name requests. */
            /* Successful driver request. */

            media_ptr -> fx_media_driver_status = FX_SUCCESS; break; }
```

```
 case FX_DRIVER_UNINIT: {

    /* There is nothing to do in this case for the RAM driver.
        For actual devices some shutdown processing may be necessary. */

    /* Successful driver request. */
    media_ptr -> fx_media_driver_status = FX_SUCCESS; break; }

case FX_DRIVER_BOOT_READ: {
    /* Read the boot record and return to the caller. */

    /* Calculate the RAM disk boot sector offset, which is at the
        very beginning of the RAM disk. Note the RAM disk memory is pointed
        to by the fx_media_driver_info pointer, which is supplied by the
        application in the call to fx_media_open. */

    source_buffer = (UCHAR *)media_ptr -> fx_media_driver_info;

    /* For RAM driver, determine if the boot record is valid. */

    if ((source_buffer[0] != (UCHAR)0xEB) ||

    ((source_buffer[1] != (UCHAR)0x34) &&

    (source_buffer[1] != (UCHAR)0x76)) || /* exFAT jump code. */

    (source_buffer[2] != (UCHAR)0x90)) {

    /* Invalid boot record, return an error! */
    media_ptr -> fx_media_driver_status = FX_MEDIA_INVALID; return; }

    /* For RAM disk only, pickup the bytes per sector. */

    bytes_per_sector =
        _fx_utility_16_unsigned_read(&source_buffer[FX_BYTES_SECTOR]); #ifdef FX_ENABLE_EXFAT

    /* if byte per sector is zero, then treat it as exFAT volume. */

    if (bytes_per_sector == 0 && (source_buffer[1] == (UCHAR)0x76)) {

    /* Pickup the byte per sector shift, and calculate byte per sector. */
    bytes_per_sector = (UINT)(1 << source_buffer[FX_EF_BYTE_PER_SECTOR_SHIFT]);

    }

    #endif /* FX_ENABLE_EXFAT */

    /* Ensure this is less than the media memory size. */
    if (bytes_per_sector \media_ptr -> fx_media_memory_size) {

    media_ptr -> fx_media_driver_status = FX_BUFFER_ERROR; break; }

    /* Copy the RAM boot sector into the destination. */

    _fx_utility_memory_copy(source_buffer, media_ptr -> fx_media_driver_buffer, bytes_per_sector);

    /* Successful driver request. */

    media_ptr -> fx_media_driver_status = FX_SUCCESS; break; }

case FX_DRIVER_BOOT_WRITE: {

    /* Write the boot record and return to the caller. */

    /* Calculate the RAM disk boot sector offset, which is at the
        very beginning of the RAM disk. Note the RAM disk memory is
        pointed to by the fx_media_driver_info pointer, which is supplied
        by the application in the call to fx_media_open. */
    destination_buffer = (UCHAR *)media_ptr -> fx_media_driver_info;
```

```
            /* Copy the RAM boot sector into the destination. */

            _fx_utility_memory_copy(media_ptr -> fx_media_driver_buffer,
                destination_buffer, media_ptr -> fx_media_bytes_per_sector);

            /* Successful driver request. */

            media_ptr -> fx_media_driver_status = FX_SUCCESS; break; }

        default: {
            /* Invalid driver request. */
            media_ptr -> fx_media_driver_status = FX_IO_ERROR; break;}
    }
}
```

第 6 章 - Azure RTOS FileX 容错模块

2021/4/29 •

本章包含 Azure RTOS FileX 容错模块的说明，该模块设计用于在文件写入操作过程中介质断电或弹出时保持文件系统完整性。

FileX 容错模块概述

当应用程序将数据写入文件时，FileX 会同时更新数据群集和系统信息。这些更新必须以原子操作的形式完成，以使文件系统中的信息保持一致。例如，在将数据追加到文件时，FileX 需要在介质中查找可用群集，更新 FAT 链，更新目录条目中存档的长度，并可能会更新目录条目中的起始群集号。电源故障或介质弹出可能会中断更新序列，这会使文件系统处于不一致状态。如果未更正不一致状态，则更新的数据可能会丢失，并且由于系统信息损坏，后续文件系统操作可能会损坏介质上的其他文件或目录。

FileX 容错模块的工作方式是在将更新文件所需的步骤应用于文件系统之前，记录这些步骤。如果文件更新成功，则会删除这些日志条目。但是，如果文件更新遭到中断，日志条目会存储在介质上。下次装载介质时，FileX 会从以前(未完成)的写入操作检测这些日志条目。在这种情况下，FileX 可以回滚已对文件系统进行的更改，或是重新应用所需更改以完成以前的操作，从而从失败中恢复。这样，如果介质在更新操作过程中断电，FileX 容错模块可保持文件系统完整性。

> **IMPORTANT**
>
> FileX 容错模块并不是旨在防止由于包含有效数据的物理介质损坏而导致的文件系统损坏。

> **IMPORTANT**
>
> FileX 容错模块保护介质之后，除了启用容错功能的 FileX，不得使用任何其他工具来装载介质。这样做可能会导致文件系统中的日志条目与介质上的系统信息不一致。如果 FileX 容错模块尝试在其他文件系统更新介质之后处理日志条目，则恢复过程可能会失败，从而使整个文件系统处于不可预测的状态。

容错模块的使用

FileX 容错功能可用于 FileX 支持的所有 FAT 文件系统，包括 FAT12、FAT16、FAT32 和 exFAT。若要启用容错功能，必须在定义了 FX_ENABLE_FAULT_TOLERANT 符号的情况下生成 FileX。在运行时，应用程序会在调用 fx_media_open 之后立即调用 fx_fault_tolerant_enable，从而启动容错服务。启用容错功能后，对指定介质进行的所有文件写入操作都会受到保护。默认情况下，容错模块未启用。

> **IMPORTANT**
>
> 应用程序需要确保在调用 fx_fault_tolerant_enable 之前未访问文件系统。如果在容错功能启用之前，应用程序将数据写入文件系统，则在以前的写入操作未完成时，写入操作可能会损坏介质，并且不会使用容错日志条目还原文件系统。

FileX 容错模块日志

FileX 容错日志在闪存中占用一个逻辑群集。该群集的起始群集号索引记录在介质的引导扇区中，偏移通过符号 FX_FAULT_TOLERANT_BOOT_INDEX 进行指定。默认情况下，此符号定义为 116。此位置已选择，因为它在 FAT12/16/32 和 exFAT 规范中标记为保留。

图 5“日志结构布局”显示了日志结构的常规布局。日志结构包含三个部分：日志标头、FAT 链和日志条目。

IMPORTANT

日志条目中存储的所有多字节值都为 Little Endian 格式。

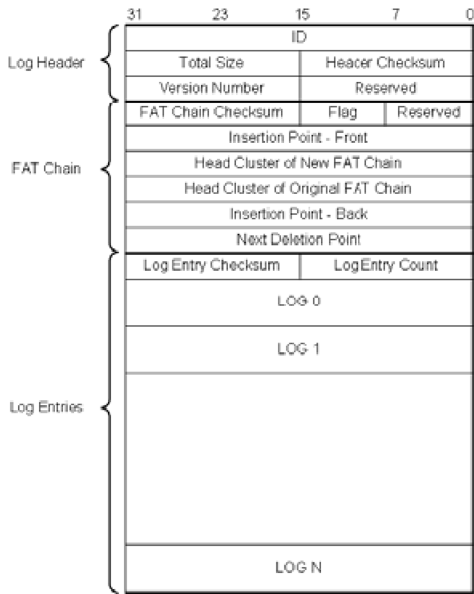

图 5. 日志结构布局

日志标头区域包含描述整个日志结构并详细说明每个字段的信息。

表 8. 日志标头区域

[illegible]	[illegible]([illegible])	[illegible]
ID	4	标识 FileX 容错日志结构。如果 ID 值不是 0x46544C52，则日志结构被视为无效。
总大小	2	指示整个日志结构的总大小(以字节为单位)。
标头校验和	2	用于转换日志标头区域的校验和。如果标头字段未通过校验和验证，则日志结构被视为无效。
版本号	2	FileX 容错主要版本号和次要版本号。
保留	2	供将来扩展。

日志标头区域后跟 FAT 链日志区域。图 9 包含描述应如何修改 FAT 链的信息。此日志区域包含有关分配给文件的群集、从文件中删除的群集以及插入/删除位置的信息，并描述了 FAT 链日志区域中的每个字段。

表 9. FAT 链日志区域

[illegible]	[illegible]([illegible])	[illegible]
FAT 链日志校验和	2	整个 FAT 链日志区域的校验和。如果未通过校验和验证，则 FAT 链日志区域被视为无效。

[illegible]	[illegible]([illegible])	[illegible]
标志	1	有效标志值包括: 0x01 FAT 链有效 0x02 正在使用位图
保留	1	保留以供将来使用
插入点 – 前	4	新创建的链将附加到的群集(属于原始 FAT 链)。
新 FAT 链的头群集	4	新创建的 FAT 链的第一个群集
原始 FAT 链的头群集	4	要删除的原始 FAT 链部分的第一个群集。
插入点 – 后	4	新创建的 FAT 链联接的原始群集。
下一个删除点	4	此字段可帮助进行 FAT 链清理过程。

日志条目区域包含的日志条目描述从失败中恢复所需的更改。FileX 容错模块中支持三种类型的日志条目:FAT 日志条目;目录日志条目;以及位图日志条目。

以下三个图和三个表详细描述了这些日志条目。

图 6. FAT 日志条目

表 10. FAT 日志条目

[illegible]	[illegible]([illegible])	[illegible]
类型	2	条目的类型，必须为 FX_FAULT_TOLERANT_FAT_LOG_TYPE
大小	2	此条目的大小
群集号	4	群集号
值	4	要写入 FAT 条目的值

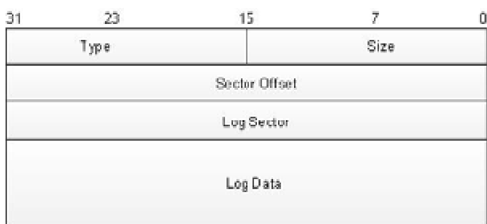

图 7. 目录日志条目

表 11. 目录日志条目

[illegible]	[illegible]([illegible])	[illegible]
类型	2	条目的类型，必须为 FX_FAULT_TOLERANT_DIRECTORY_LOG_TYPE
大小	2	此条目的大小
扇区偏移	4	到此目录所在的扇区的偏移（以字节为单位）。
日志扇区	4	目录条目所在的扇区
日志数据	变量	目录条目的内容

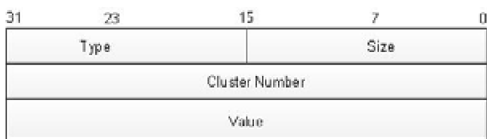

图 8. 位图日志条目

表 12. 位图日志条目

[illegible]	[illegible]([illegible])	[illegible]
类型	2	条目的类型，必须为 FX_FAULT_TOLERANT_BITMAP_LOG_TYPE
大小	2	此条目的大小
群集号	4	群集号
值	4	要写入 FAT 条目的值

容错保护

FileX 容错模块启动后，它首先会在介质中搜索现有容错日志文件。如果找不到有效日志文件，则 FileX 会将介质视为未受保护。在这种情况下，FileX 会在介质上创建容错日志文件。

> **IMPORTANT**
> *如果文件系统在 FileX 容错模块启动之前已损坏，则 FileX 无法保护该系统。*

如果找到容错日志文件，则 FileX 会检查现有日志条目。没有日志条目的日志文件指示以前的文件操作成功，所有日志条目都已删除。在这种情况下，应用程序可以开始使用具有容错保护的文件系统。

不过，如果找到日志条目，则 FileX 需要完成以前的文件操作，或还原已应用于文件系统的更改，从而有效地撤消更改。在任一情况下，将日志条目应用于文件系统后，文件系统会还原为一致状态，应用程序可以再次开始使用文件系统。

对于受 FileX 保护的介质，在文件更新操作过程中，数据部分会直接写入介质。在 FileX 写入数据时，它还会记录需要应用于目录条目 FAT 表的所有更改。此信息记录在文件容错日志条目中。此方法可保证将数据写入介质后，对文件系统进行更新。如果在数据写入阶段弹出介质，则至关重要的文件系统信息尚未更改。因此，文件系

统不受中断影响。

将所有数据成功写入介质后，FileX 随后会按照日志条目中的信息，将更改应用于系统信息（一次一个条目）。所有系统信息都提交到介质后，日志条目会从容错日志中删除。此时，FileX 便完成了文件更新操作。

在文件更新操作过程中，不会就地更新文件。容错模块会为数据分配扇区以供写入新数据，然后删除包含要覆盖的数据的扇区，从而更新相关 FAT 条目以将新扇区链接到链中。在需要修改群集中部分数据的情况下，FileX 会始终分配新群集，将包含更新的数据的旧群集中的所有数据写入新群集，然后释放旧群集。这可保证如果文件更新中断，原始文件保持不变。应用程序需要注意，在 FileX 容错保护下，更新文件中的数据需要介质具有足够的可用空间，以便在释放包含旧数据的扇区之前容纳新数据。如果介质没有足够空间来容纳新数据，则更新操作会失败。

附录 A - Azure RTOS FileX 服务

2021/4/29 •

系统服务

```
UINT    fx_system_date_get(UINT *year, UINT *month, UINT *day);
UINT    fx_system_date_set(UINT year, UINT month, UINT day);
UINT    fx_system_time_get(UINT *hour, UINT *minute, UINT *second);
UINT    fx_system_time_set(UINT hour, UINT minute, UINT second);
VOID    fx_system_initialize(VOID);
```

媒体服务

```
UINT    fx_media_abort(FX_MEDIA *media_ptr);
UINT    fx_media_cache_invalidate(FX_MEDIA *media_ptr);
UINT    fx_media_check(FX_MEDIA *media_ptr, UCHAR *scratch_memory_ptr,
                       ULONG scratch_memory_size, ULONG error_correction_option,
                       ULONG *errors_detected_ptr);

UINT    fx_media_close(FX_MEDIA *media_ptr);
UINT    fx_media_close_notify_set(FX_MEDIA *media_ptr, VOID (*media_close_notify)(FX_MEDIA *media));
UINT    fx_media_exFAT_format(FX_MEDIA *media_ptr, VOID (*driver)(FX_MEDIA *media),
                              VOID *driver_info_ptr, UCHAR *memory_ptr,
                              UINT memory_size, CHAR *volume_name,
                              UINT number_of_fats, ULONG64 hidden_sectors,
                              ULONG64 total_sectors, UINT bytes_per_sector,
                              UINT sectors_per_cluster, UINT volume_serial_number,
                              UINT boundary_unit);
UINT    fx_media_extended_space_available(FX_MEDIA *media_ptr, ULONG64 *available_bytes_ptr);
UINT    fx_media_flush(FX_MEDIA *media_ptr);
UINT    fx_media_format(FX_MEDIA *media_ptr, VOID (*driver)(FX_MEDIA *media),
                        VOID *driver_info_ptr, UCHAR *memory_ptr,
                        UINT memory_size, CHAR *volume_name,
                        UINT number_of_fats, UINT directory_entries,
                        UINT hidden_sectors, ULONG total_sectors,
                        UINT bytes_per_sector, UINT sectors_per_cluster,
                        UINT heads, UINT sectors_per_track);

UINT    fx_media_open(FX_MEDIA *media_ptr, CHAR *media_name,
                      VOID (*media_driver)(FX_MEDIA *),
                      VOID *driver_info_ptr, VOID *memory_ptr,
                      ULONG memory_size);

UINT    fx_media_open_notify_set(FX_MEDIA *media_ptr, VOID (*media_open_notify)(FX_MEDIA *media));

UINT    fx_media_read(FX_MEDIA *media_ptr, ULONG logical_sector, VOID *buffer_ptr);

UINT    fx_media_space_available(FX_MEDIA *media_ptr, ULONG *available_bytes_ptr);
UINT    fx_media_volume_get(FX_MEDIA *media_ptr, CHAR *volume_name, UINT volume_source);

UINT    fx_media_volume_get_extended(FX_MEDIA *media_ptr, CHAR *volume_name,
                                     UINT volume_name_buffer_length, UINT volume_source);

UINT    fx_media_volume_set(FX_MEDIA *media_ptr, CHAR *volume_name);
UINT    fx_media_write(FX_MEDIA *media_ptr, ULONG logical_sector, VOID *buffer_ptr);
```

目录服务

```
UINT    fx_directory_attributes_read(FX_MEDIA *media_ptr, CHAR *directory_name,
                                     UINT *attributes_ptr);

UINT    fx_directory_attributes_set(FX_MEDIA *media_ptr, CHAR *directory_name,
                                    UINT attributes);

UINT    fx_directory_create(FX_MEDIA *media_ptr, CHAR *directory_name);
UINT    fx_directory_default_get(FX_MEDIA *media_ptr, CHAR **return_path_name);
UINT    fx_directory_default_set(FX_MEDIA *media_ptr, CHAR *new_path_name);
UINT    fx_directory_delete(FX_MEDIA *media_ptr, CHAR *directory_name);
UINT    fx_directory_first_entry_find(FX_MEDIA *media_ptr, CHAR *directory_name);
UINT    fx_directory_first_full_entry_find(FX_MEDIA *media_ptr, CHAR *directory_name,
                                           UINT *attributes, ULONG *size,
                                           UINT *year, UINT *month, UINT *day,
                                           UINT *hour, UINT *minute, UINT *second);

UINT    fx_directory_information_get(FX_MEDIA *media_ptr, CHAR *directory_name,
                                      UINT *attributes, ULONG *size,
                                      UINT *year, UINT *month, UINT *day,
                                      UINT *hour, UINT *minute, UINT *second);
UINT    fx_directory_local_path_clear(FX_MEDIA *media_ptr);
UINT    fx_directory_local_path_get(FX_MEDIA *media_ptr, CHAR **return_path_name);
UINT    fx_directory_local_path_restore(FX_MEDIA *media_ptr, FX_LOCAL_PATH *local_path_ptr);
UINT    fx_directory_local_path_set(FX_MEDIA *media_ptr, FX_LOCAL_PATH *local_path_ptr,
                                    CHAR *new_path_name);

UINT    fx_directory_long_name_get(FX_MEDIA *media_ptr, CHAR *short_name, CHAR *long_name);

UINT    fx_directory_long_name_get_extended( FX_MEDIA *media_ptr, CHAR *short_name,
                                             CHAR *long_name, UINT long_file_name_buffer_length);

UINT    fx_directory_name_test(FX_MEDIA *media_ptr, CHAR *directory_name);
UINT    fx_directory_next_entry_find(FX_MEDIA *media_ptr, CHAR *directory_name);
UINT    fx_directory_next_full_entry_find(FX_MEDIA *media_ptr, CHAR *directory_name,
                                          UINT *attributes, ULONG *size,
                                          UINT *year,UINT *month, UINT *day,
                                          UINT *hour, UINT *minute, UINT *second);

UINT    fx_directory_rename(FX_MEDIA *media_ptr, CHAR *old_directory_name,
                            CHAR *new_directory_name);

UINT    fx_directory_short_name_get(FX_MEDIA *media_ptr, CHAR *long_name,
                                    CHAR *short_name);

UINT    fx_directory_short_name_get_extended(FX_MEDIA *media_ptr, CHAR *long_name,
                                             CHAR *short_name, UINT short_file_name_length);
```

文件服务

```
UINT    fx_fault_tolerant_enable(FX_MEDIA *media_ptr, VOID *memory_buffer, UINT memory_size);
UINT    fx_file_allocate(FX_FILE *file_ptr, ULONG size);
UINT    fx_file_attributes_read(FX_MEDIA *media_ptr, CHAR *file_name, UINT *attributes_ptr);
UINT    fx_file_attributes_set(FX_MEDIA *media_ptr, CHAR *file_name, UINT attributes);
UINT    fx_file_best_effort_allocate(FX_FILE *file_ptr, ULONG size, ULONG *actual_size_allocated);
UINT    fx_file_close(FX_FILE *file_ptr);
UINT    fx_file_create(FX_MEDIA *media_ptr, CHAR *file_name);
UINT    fx_file_date_time_set(FX_MEDIA *media_ptr, CHAR *file_name,
                              UINT year, UINT month, UINT day,
                              UINT hour, UINT minute, UINT second);
UINT    fx_file_delete(FX_MEDIA *media_ptr, CHAR *file_name);
UINT    fx_file_extended_allocate(FX_FILE *file_ptr, ULONG64 size);
UINT    fx_file_extended_best_effort_allocate(FX_FILE *file_ptr, ULONG64 size,
                                              ULONG64 *actual_size_allocated);
UINT    fx_file_extended_relative_seek(FX_FILE *file_ptr, ULONG64 byte_offset,
                                       UINT seek_from);
UINT    fx_file_extended_seek(FX_FILE *file_ptr, ULONG64 byte_offset);
UINT    fx_file_extended_truncate(FX_FILE *file_ptr, ULONG64 size);
UINT    fx_file_extended_truncate_release(FX_FILE *file_ptr, ULONG64 size);
UINT    fx_file_open(FX_MEDIA *media_ptr, FX_FILE *file_ptr,
                     CHAR *file_name, UINT open_type);
UINT    fx_file_read(FX_FILE *file_ptr, VOID *buffer_ptr,
                     ULONG request_size, ULONG *actual_size);
UINT    fx_file_relative_seek(FX_FILE *file_ptr, ULONG byte_offset, UINT seek_from);
UINT    fx_file_rename(FX_MEDIA *media_ptr, CHAR *old_file_name,
                       CHAR *new_file_name);
UINT    fx_file_seek(FX_FILE *file_ptr, ULONG byte_offset);
UINT    fx_file_truncate(FX_FILE *file_ptr, ULONG size);
UINT    fx_file_truncate_release(FX_FILE *file_ptr, ULONG size);
UINT    fx_file_write(FX_FILE *file_ptr, VOID *buffer_ptr, ULONG size);

UINT    fx_file_write_notify_set(FX_FILE *file_ptr, VOID (*file_write_notify)(FX_FILE *file));
```

Unicode 服务

```
UINT    fx_unicode_directory_create(FX_MEDIA *media_ptr, UCHAR *source_unicode_name,
                                    ULONG source_unicode_length, CHAR *short_name);

UINT    fx_unicode_directory_rename(FX_MEDIA *media_ptr, UCHAR *old_unicode_name,
                                    ULONG old_unicode_length, UCHAR *new_unicode_name,
                                    ULONG new_unicode_length, CHAR *new_short_name);

UINT    fx_unicode_file_create(FX_MEDIA *media_ptr, UCHAR *source_unicode_name,
                               ULONG source_unicode_length, CHAR *short_name);

UINT    fx_unicode_file_rename(FX_MEDIA *media_ptr, UCHAR *old_unicode_name,
                               ULONG old_unicode_length, UCHAR *new_unicode_name,
                               ULONG new_unicode_length, CHAR *new_short_name);

ULONG    fx_unicode_length_get(UCHAR *unicode_name);


UINT    fx_unicode_length_get_extended(UCHAR *unicode_name, UINT buffer_length);
UINT    fx_unicode_name_get(FX_MEDIA *media_ptr, CHAR *source_short_name,
                            UCHAR *destination_unicode_name, ULONG *destination_unicode_length);

UINT    fx_unicode_name_get_extended(FX_MEDIA *media_ptr, CHAR *source_short_name,
                                     UCHAR *destination_unicode_name, ULONG *destination_unicode_length,
                                     ULONG unicode_name_buffer_length);

UINT    fx_unicode_short_name_get(FX_MEDIA *media_ptr, UCHAR *source_unicode_name,
                                  ULONG source_unicode_length, CHAR *destination_short_name);

UINT    fx_unicode_short_name_get_extended(FX_MEDIA *media_ptr, UCHAR *source_unicode_name,
                                           ULONG source_unicode_length, CHAR *destination_short_name,
                                           ULONG short_name_buffer_length);
```

附录 B - Azure RTOS FileX 常量

2021/4/29 •

字母顺序|列表

[illegible]([illegible])	“[illegible]”	
EXFAT_BIT_MAP_FIRST_TABLE	0	
EXFAT_BOOT_REGION_SIZE	24	
EXFAT_DEFAULT_BOUNDARY_UNIT	28	
EXFAT_FAT_BITS	2	
EXFAT_FAT_BYTES_PER_SECTOR_SHIFT	0x009	
EXFAT_FAT_DRIVE_SELECT	0x080	
EXFAT_FAT_FILE_SYS_REVISION	0x100	
EXFAT_FAT_NUM_OF_FATS	0x001	
EXFAT_FAT_VOLUME_FLAG	0x000	
EXFAT_FAT_VOLUME_NAME_FIELD_SIZE	11	
EXFAT_LAST_CLUSTER_MASK	0xFFFFFFFF	
EXFAT_MIN_NUM_OF_RESERVED_SECTORS	1	
EXFAT_NUM_OF_DIR_ENTRIES	2	
FX_12_BIT_FAT_SIZE	4086	
FX_12BIT_SIZE	3	
FX_16_BIT_FAT_SIZE	65525	
FX_ACCESS_ERROR	0x06	
FX_ALREADY_CREATED	0x0B	
FX_ARCHIVE	0x20	
FX_BAD_CLUSTER	0xFFF7	

[illegible] ([illegible])	“[illegible]”	
FX_BAD_CLUSTER_32	0x0FFFFFF7	
FX_BAD_CLUSTER_EXFAT	0x0FFFFFF7	
FX_BASE_YEAR	1980	
FX_BIGDOS	0x06	
FX_BOOT_ERROR	0x01	
FX_BOOT_SECTOR	1	
FX_BOOT_SECTOR_SIZE	512	
FX_BOOT_SIG	0x026	
FX_BUFFER_ERROR	0x21	
FX_BYTES_SECTOR	0x00B	
FX_CALLER_ERROR	0x20	
FX_DATA_SECTOR	4	
FX_DAY_MASK	0x1F	
FX_DIR_ENTRY_DONE	0x00	
FX_DIR_ENTRY_FREE	0xE5	
FX_DIR_ENTRY_SIZE	32	
FX_DIR_EXT_SIZE	3	
FX_DIR_NAME_SIZE	8	
FX_DIR_NOT_EMPTY	0x10	
FX_DIR_RESERVED	8	
FX_DIRECTORY	0x10	
FX_DIRECTORY_ERROR	0x02	
FX_DIRECTORY_SECTOR	3	
FX_DRIVE_NUMBER	0x024	
FX_DRIVER_ABORT	3	

[illegible]	“[illegible]”	
FX_DRIVER_BOOT_READ	5	
FX_DRIVER_BOOT_WRITE	7	
FX_DRIVER_FLUSH	2	
FX_DRIVER_INIT	4	
FX_DRIVER_READ	0	
FX_DRIVER_RELEASE_SECTORS	6	
FX_DRIVER_UNINIT	8	
FX_DRIVER_WRITE	1	
FX_EF_BOOT_CODE	120	
FX_EF_BYTE_PER_SECTOR_SHIFT	108	
FX_EF_CLUSTER_COUNT	92	
FX_EF_CLUSTER_HEAP_OFFSET	88	
FX_EF_DRIVE_SELECT	111	
FX_EF_FAT_LENGTH	84	
FX_EF_FAT_OFFSET	80	
FX_EF_FILE_SYSTEM_REVISION	104	
FX_EF_FIRST_CLUSTER_OF_ROOT_DIR	96	
FX_EF_MUST_BE_ZERO	11	
FX_EF_NUMBER_OF_FATS	110	
FX_EF_PARTITION_OFFSET	64	
FX_EF_PERCENT_IN_USE	112	
FX_EF_RESERVED	113	
FX_EF_SECTOR_PER_CLUSTER_SHIFT	109	
FX_EF_VOLUME_FLAGS	106	
FX_EF_VOLUME_LENGTH	72	

[illegible]([illegible])	“[illegible]”	
FX_EF_VOLUME_SERIAL_NUMBER	100	
FX_END_OF_FILE	0x09	
FX_ERROR_FIXED	0x92	
FX_ERROR_NOT_FIXED	0x93	
FX_EXFAT	0x07	
FX_EXFAT_BIT_MAP_NUM_OF_CACHED_SECTORS	1	
FX_EXFAT_BITMAP_CLUSTER_FREE	0	
FX_EXFAT_BITMAP_CLUSTER_OCCUPIED	1	
FX_EXFAT_FAT_CHECK_SUM_OFFSET	11	
FX_EXFAT_FAT_MAIN_BOOT_SECTOR_OFFSET	1	
FX_EXFAT_FAT_MAIN_SYSTEM_AREA_SIZE	12	
FX_EXFAT_FAT_NUM_OF_SYSTEM_AREAS	2	
FX_EXFAT_FAT_OEM_PARAM_OFFSET	9	
FX_EXFAT_MAX_DIRECTORY_SIZE	0x10000000	
FX_EXFAT_SIZE_OF_FAT_ELEMENT_SHIFT	2	
FX_FALSE	0	
FX_FAT_CACHE_DEPTH	4	
FX_FAT_CACHE_HASH_MASK	0x3	
FX_FAT_CHAIN_ERROR	0x01	
FX_FAT_ENTRY_START	2	
FX_FAT_MAP_SIZE	128	
FX_FAT_READ_ERROR	0x03	
FX_FAT_SECTOR	2	

[illegible]（[illegible]）	“[illegible]”	
FX_FAT12	0x01	
FX_FAT16	0x04	
FX_FAT32	0x0B	
FX_FAULT_TOLERANT_CACHE_SIZE	1024	
FX_FILE_ABORTED_ID	0x46494C41UL	
FX_FILE_CLOSED_ID	0x46494C43UL	
FX_FILE_CORRUPT	0x08	
FX_FILE_ID	0x46494C45UL	
FX_FILE_SIZE_ERROR	0x08	
FX_FILE_SYSTEM_TYPE	0x036	
FX_FREE_CLUSTER	0x0000	
FX_HEADS	0x01A	
FX_HIDDEN	0x02	
FX_HIDDEN_SECTORS	0x01C	
FX_HOUR_MASK	0x1F	
FX_HOUR_SHIFT	11	
FX_HUGE_SECTORS	0x020	
FX_INITIAL_DATE	0x4761	
FX_INITIAL_TIME	0x0000	
FX_INVALID_ATTR	0x19	
FX_INVALID_CHECKSUM	0x95	
FX_INVALID_DAY	0x14	
FX_INVALID_HOUR	0x15	
FX_INVALID_MINUTE	0x16	
FX_INVALID_MONTH	0x13	

[illegible]([illegible])	“[illegible]”	
FX_INVALID_NAME	0x0C	
FX_INVALID_OPTION	0x24	
FX_INVALID_PATH	0x0D	
FX_INVALID_SECOND	0x17	
FX_INVALID_STATE	0x97	
FX_INVALID_YEAR	0x12	
FX_IO_ERROR	0x90	
FX_JUMP_INSTR	0x000	
FX_LAST_CLUSTER_1	0xFFF8	
FX_LAST_CLUSTER_1_32	0x0FFFFFF8	
FX_LAST_CLUSTER_2	0xFFFF	
FX_LAST_CLUSTER_2_32	0x0FFFFFFF	
FX_LAST_CLUSTER_EXFAT	0x0FFFFFFF	
FX_LONG_NAME	0xF	
FX_LONG_NAME_ENTRY_LEN	13	
FX_LOST_CLUSTER_ERROR	0x04	
FX_MAX_12BIT_CLUST	0x0FF0	
FX_MAX_EX_FAT_NAME_LEN	255	
FX_MAX_FAT_CACHE	256	
FX_MAX_LAST_NAME_LEN	256	
FX_MAX_LONG_NAME_LEN	256	
FX_MAX_SECTOR_CACHE	256	
FX_MAX_SHORT_NAME_LEN	13	
FX_MAXIMUM_HOUR	23	
FX_MAXIMUM_MINUTE	59	

ɪɪ（ɪɪɪɪɪ）	“ɪ”	
FX_MAXIMUM_MONTH	12	
FX_MAXIMUM_PATH	256	
FX_MAXIMUM_SECOND	59	
FX_MAXIMUM_YEAR	2107	
FX_MEDIA_ABORTED_ID	0x4D454441UL	
FX_MEDIA_CLOSED_ID	0x4D454443UL	
FX_MEDIA_ID	0x4D454449UL	
FX_MEDIA_INVALID	0x02	
FX_MEDIA_NOT_OPEN	0x11	
FX_MEDIA_TYPE	0x015	
FX_MINUTE_MASK	0x3F	
FX_MINUTE_SHIFT	5	
FX_MONTH_MASK	0x0F	
FX_MONTH_SHIFT	5	
FX_NO_FAT	0xFF	
FX_NO_MORE_ENTRIES	0x0F	
FX_NO_MORE_SPACE	0x0A	
FX_NOT_A_FILE	0x05	
FX_NOT_AVAILABLE	0x94	
FX_NOT_DIRECTORY	0x0E	
FX_NOT_ENOUGH_MEMORY	0x91	
FX_NOT_FOUND	0x04	
FX_NOT_IMPLEMENTED	0x22	
FX_NOT_OPEN	0x07	
FX_NOT_USED	0x0001	

[illegible]([illegible])	“[illegible]”	
FX_NULL	0	
FX_NUMBER_OF_FATS	0x010	
FX_OEM_NAME	0x003	
FX_OPEN_FOR_READ	0	
FX_OPEN_FOR_READ_FAST	2	
FX_OPEN_FOR_WRITE	1	
FX_PTR_ERROR	0x18	
FX_READ_CONTINU	0x96	
FX_READ_ONLY	0x01	
FX_RESERVED	0x025	
FX_RESERVED_1	0xFFF0	
FX_RESERVED_1_32	0x0FFFFFF0	
FX_RESERVED_1_EXFAT	0xFFFFFFF8	
FX_RESERVED_2	0xFFF6	
FX_RESERVED_2_32	0x0FFFFFF6	
FX_RESERVED_2_EXFAT	0xFFFFFFFE	
FX_RESERVED_SECTOR	0x00E	
FX_ROOT_CLUSTER_32	0x02C	
FX_ROOT_DIR_ENTRIES	0x011	
FX_SECOND_MASK	0x1F	
FX_SECTOR_CACHE_DEPTH	4	
FX_SECTOR_CACHE_HASH_ENABLE	16	
FX_SECTOR_CACHE_HASH_MASK	0x3	
FX_SECTOR_INVALID	0x89	
FX_SECTORS	0x013	

ɛɛ（ɛɛɛɛɛ）	“ɛ”	
FX_SECTORS_CLUSTER	0x00D	
FX_SECTORS_PER_FAT	0x016	
FX_SECTORS_PER_FAT_32	0x024	
FX_SECTORS_PER_TRK	0x018	
FX_SEEK_BACK	3	
FX_SEEK_BEGIN	0	
FX_SEEK_END	1	
FX_SEEK_FORWARD	2	
FX_SIG_BYTE_1	0x55	
FX_SIG_BYTE_2	0xAA	
FX_SIG_OFFSET	0x1FE	
FX_SIGN_EXTEND	0xF000	
FX_SUCCESS	0x00	
FX_SYSTEM	0x04	
FX_TRUE	1	
FX_UNKNOWN_SECTOR	0	
FX_VOLUME	0x08	
FX_VOLUME_ID	0x027	
FX_VOLUME_LABEL	0x02B	
FX_WRITE_PROTECT	0x23	
FX_YEAR_MASK	0x7F	
FX_YEAR_SHIFT	9	

按值列出的列表

ɛɛ（ɛɛ）	“ɛ”
FX_DIR_ENTRY_DONE	0x00

[illegible]([illegible])	“[illegible]”
FX_DRIVER_READ	0
FX_FALSE	0
EXFAT_BIT_MAP_FIRST_TABLE	0
FX_FREE_CLUSTER	0x0000
FX_INITIAL_TIME	0x0000
FX_JUMP_INSTR	0x000
FX_NULL	0
FX_OPEN_FOR_READ	0
FX_SEEK_BEGIN	0
FX_SUCCESS	0x00
FX_UNKNOWN_SECTOR	0
FX_EXFAT_FAT_MAIN_BOOT_SECTOR_OFFSET	0
FX_EXFAT_BITMAP_CLUSTER_FREE	0
EXFAT_FAT_VOLUME_FLAG	0x000
FX_BOOT_ERROR	0x01
FX_BOOT_SECTOR	1
FX_DRIVER_WRITE	1
FX_FAT_CHAIN_ERROR	0x01
FX_NOT_USED	0x0001
FX_OPEN_FOR_WRITE	1
FX_READ_ONLY	0x01
FX_FAT12	0x01
EXFAT_FAT_NUM_OF_FATS	0x001
FX_SEEK_END	1
FX_TRUE	1

[illegible]([illegible])	“[illegible]”
FX_EXFAT_BIT_MAP_NUM_OF_CACHED_SECTORS	1
FX_EXFAT_BITMAP_CLUSTER_OCCUPIED	1
FX_EXFAT_FAT_EXT_BOOT_SECTOR_OFFSET	1
EXFAT_MIN_NUM_OF_RESERVED_SECTORS	1
FX_DIRECTORY_ERROR	0x02
FX_HIDDEN	0x02
FX_MEDIA_INVALID	0x02
FX_DRIVER_FLUSH	2
FX_FAT_ENTRY_START	2
FX_FAT_SECTOR	2
FX_OPEN_FOR_READ_FAST	2
FX_SEEK_FORWARD	2
FX_EXFAT_SIZE_OF_FAT_ELEMENT_SHIFT	2
FX_EXFAT_FAT_NUM_OF_SYSTEM_AREAS	2
EXFAT_NUM_OF_DIR_ENTRIES	2
FX_12BIT_SIZE	3
FX_DIR_EXT_SIZE	3
FX_DIRECTORY_SECTOR	3
FX_DRIVER_ABORT	3
FX_FAT_CACHE_HASH_MASK	0x3
FX_FAT_READ_ERROR	0x03
FX_OEM_NAME	0x003
FX_SECTOR_CACHE_HASH_MASK	0x3
FX_SEEK_BACK	3
FX_DATA_SECTOR	4

[illegible]([illegible])	“[illegible]”
FX_DRIVER_INIT	4
FX_FAT_CACHE_DEPTH	4
FX_FAT16	0x04
FX_LOST_CLUSTER_ERROR	0x04
FX_NOT_FOUND	0x04
FX_SECTOR_CACHE_DEPTH	4
FX_SYSTEM	0x04
FX_DRIVER_BOOT_READ	5
FX_MINUTE_SHIFT	5
FX_MONTH_SHIFT	5
FX_NOT_A_FILE	0x05
FX_ACCESS_ERROR	0x06
FX_BIGDOS	0x06
FX_DRIVER_RELEASE_SECTORS	6
FX_DRIVER_BOOT_WRITE	7
FX_NOT_OPEN	0x07
FX_EXFAT	0x07
FX_DIR_NAME_SIZE	8
FX_DIR_RESERVED	8
FX_DRIVER_UNINIT	8
FX_FILE_CORRUPT	0x08
FX_FILE_SIZE_ERROR	0x08
FX_VOLUME	0x08
FX_END_OF_FILE	0x09
EXFAT_FAT_BYTES_PER_SECTOR_SHIFT	0x009

[illegible]([illegible])	“[illegible]”
FX_YEAR_SHIFT	9
FX_EXFAT_FAT_OEM_PARAM_OFFSET	9
FX_NO_MORE_SPACE	0x0A
FX_EF_MUST_BE_ZERO	11
EXFAT_FAT_VOLUME_NAME_FIELD_SIZE	11
FX_ALREADY_CREATED	0x0B
FX_FAT32	0x0B
FX_BYTES_SECTOR	0x00B
FX_HOUR_SHIFT	11
FX_EXFAT_FAT_CHECK_SUM_OFFSET	11
FX_INVALID_NAME	0x0C
FX_MAXIMUM_MONTH	12
FX_EXFAT_FAT_MAIN_SYSTEM_AREA_SIZE	12
FX_INVALID_PATH	0x0D
FX_SECTORS_CLUSTER	0x00D
FX_LONG_NAME_ENTRY_LEN	13
FX_MAX_SHORT_NAME_LEN	13
FX_NOT_DIRECTORY	0x0E
FX_RESERVED_SECTORS	0x00E
FX_LONG_NAME	0xF
FX_MONTH_MASK	0x0F
FX_NO_MORE_ENTRIES	0x0F
FX_DIR_NOT_EMPTY	0x10
FX_DIRECTORY	0x10
FX_MAX_FAT_CACHE	16

[illegible]([illegible])	“[illegible]”
FX_MAX_SECTOR_CACHE	16
FX_NUMBER_OF_FATS	0x010
FX_SECTOR_CACHE_HASH_ENABLE	16
FX_MEDIA_NOT_OPEN	0x11
FX_ROOT_DIR_ENTRIES	0x011
FX_INVALID_YEAR	0x12
FX_INVALID_MONTH	0x13
FX_SECTORS	0x013
FX_INVALID_DAY	0x14
FX_INVALID_HOUR	0x15
FX_MEDIA_TYPE	0x015
FX_INVALID_MINUTE	0x16
FX_SECTORS_PER_FAT	0x016
FX_INVALID_SECOND	0x17
FX_MAXIMUM_HOUR	23
FX_PTR_ERROR	0x18
EXFAT_BOOT_REGION_SIZE	24
FX_SECTORS_PER_TRK	0x018
FX_INVALID_ATTR	0x19
FX_HEADS	0x01A
FX_HIDDEN_SECTORS	0x01C
FX_DAY_MASK	0x1F
FX_HOUR_MASK	0x1F
FX_SECOND_MASK	0x1F
FX_ARCHIVE	0x20

[illegible]([illegible])	“[illegible]”
FX_CALLER_ERROR	0x20
FX_DIR_ENTRY_SIZE	32
EXFAT_FAT_BITS	32
FX_HUGE_SECTORS	0x020
FX_BUFFER_ERROR	0x21
FX_MAX_LONG_NAME_LEN	33
FX_NOT_IMPLEMENTED	0x22
FX_WRITE_PROTECT	0x23
FX_DRIVE_NUMBER	0x024
FX_INVALID_OPTION	0x24
FX_SECTORS_PER_FAT_32	0x024
FX_RESERVED	0x025
FX_BOOT_SIG	0x026
FX_VOLUME_ID	0x027
FX_VOLUME_LABEL	0x02B
FX_ROOT_CLUSTER_32	0x02C
FX_FILE_SYSTEM_TYPE	0x036
FX_MAXIMUM_MINUTE	59
FX_MAXIMUM_SECOND	59
FX_MINUTE_MASK	0x3F
FX_EF_PARTITION_OFFSET	64
FX_EF_VOLUME_LENGTH	72
FX_EF_FAT_OFFSET	80
FX_EF_FAT_LENGTH	84
FX_SIG_BYTE_1	0x55

[illegible]([illegible])	“[illegible]”
FX_EF_CLUSTER_HEAP_OFFSET	88
FX_EF_CLUSTER_COUNT	92
FX_EF_FIRST_CLUSTER_OF_ROOT_DIR	96
FX_EF_VOLUME_SERIAL_NUMBER	100
FX_EF_FILE_SYSTEM_REVISION	104
FX_EF_VOLUME_FLAGS	106
FX_EF_BYTE_PER_SECTOR_SHIFT	108
FX_EF_SECTOR_PER_CLUSTER_SHIFT	109
FX_EF_NUMBER_OF_FATS	110
FX_EF_DRIVE_SELECT	11
FX_EF_PERCENT_IN_USE	112
FX_EF_RESERVED	113
FX_EF_BOOT_CODE	120
FX_YEAR_MASK	0x7F
EXFAT_FAT_DRIVE_SELECT	0x80
FX_FAT_MAP_SIZE	128
EXFAT_DEFAULT_BOUNDARY_UNIT	128
FX_SECTOR_INVALID	0x89
FX_IO_ERROR	0x90
FX_NOT_ENOUGH_MEMORY	0x91
FX_ERROR_FIXED	0x92
FX_ERROR_NOT_FIXED	0x93
FX_NOT_AVAILABLE	0x94
FX_INVALID_CHECKSUM	0x95
FX_READ_CONTINUE	0x96

[illegible]（[illegible]）	“[illegible]”
FX_INVALID_STATE	0x97
FX_SIG_BYTE_2	0xAA
FX_DIR_ENTRY_FREE	0xE5
FX_NO_FAT	0xFF
EXFAT_FAT_FILE_SYS_REVISION	0x100
FX_MAX_EX_FAT_NAME_LEN	255
FX_MAXIMUM_PATH	256
FX_SIG_OFFSET	0x1FE
FX_BOOT_SECTOR_SIZE	512
FX_FAULT_TOLERANT_CACHE_SIZE	1024
FX_BASE_YEAR	1980
FX_MAXIMUM_YEAR	2107
FX_MAX_12BIT_CLUST	0x0FF0
FX_12_BIT_FAT_SIZE	4086
FX_INITIAL_DATE	0x4761
FX_SIGN_EXTEND	0xF000
FX_RESERVED_1	0xFFF0
FX_16_BIT_FAT_SIZE	65525
FX_RESERVED_2	0xFFF6
FX_BAD_CLUSTER	0xFFF7
FX_LAST_CLUSTER_1	0xFFF8
FX_LAST_CLUSTER_2	0xFFFF
FX_RESERVED_1_32	0x0FFFFFF0
FX_RESERVED_2_32	0x0FFFFFF6
FX_BAD_CLUSTER_32	0x0FFFFFF7

[illegible]（[illegible]）	“[illegible]”
FX_LAST_CLUSTER_1_32	0x0FFFFFF8
FX_LAST_CLUSTER_2_32	0x0FFFFFFF
FX_EXFAT_MAX_DIRECTORY_SIZE	0x10000000
FX_FILE_ABORTED_ID	0x46494C41UL
FX_FILE_CLOSED_ID	0x46494C43UL
FX_FILE_ID	0x46494C45UL
FX_MEDIA_ABORTED_ID	0x4D454441UL
FX_MEDIA_CLOSED_ID	0x4D454443UL
FX_MEDIA_ID	0x4D454449UL
FX_RESERVED_1_EXFAT	0xFFFFFFF8
FX_RESERVED_2_EXFAT	0xFFFFFFFE
FX_BAD_CLUSTER_EXFAT	0xFFFFFFF7
FX_LAST_CLUSTER_EXFAT	0xFFFFFFFF
EXFAT_LAST_CLUSTER_MASK	0xFFFFFFFF

附录 C - Azure RTOS FileX 数据类型

2021/4/29 •

FX_DIR_ENTRY

```
typedef struct FX_DIR_ENTRY_STRUCT
{
    CHAR        *fx_dir_entry_name;
    CHAR        fx_dir_entry_short_name[FX_MAX_SHORT_NAME_LEN];
    UINT        fx_dir_entry_long_name_present;
    UINT        fx_dir_entry_long_name_shorted;
    UCHAR       fx_dir_entry_attributes;
    UCHAR       fx_dir_entry_reserved;
    UCHAR       fx_dir_entry_created_time_ms;
    UINT        fx_dir_entry_created_time;
    UINT        fx_dir_entry_created_date;
    UINT        fx_dir_entry_last_accessed_date;
    UINT        fx_dir_entry_time;
    UINT        fx_dir_entry_date;
    ULONG       fx_dir_entry_cluster;
    ULONG64     fx_dir_entry_file_size;
    ULONG64     fx_dir_entry_log_sector;
    ULONG       fx_dir_entry_byte_offset;
    ULONG       fx_dir_entry_number;
    ULONG       fx_dir_entry_last_search_cluster;
    ULONG       fx_dir_entry_last_search_relative_cluster;
    ULONG64     fx_dir_entry_last_search_log_sector;
    ULONG       fx_dir_entry_last_search_byte_offset;
    ULONG64     fx_dir_entry_next_log_sector;

    #ifdef FX_ENABLE_EXFAT
        CHAR        fx_dir_entry_dont_use_fat;
        UCHAR       fx_dir_entry_type;
        ULONG64     fx_dir_entry_available_file_size;
        ULONG       fx_dir_entry_secondary_count;
    #endif

} FX_DIR_ENTRY;
```

FX_PATH

```
typedef struct FX_PATH_STRUCT
{
    FX_DIR_ENTRY    fx_path_directory;
    CHAR            fx_path_string[FX_MAXIMUM_PATH];
    CHAR            fx_path_name_buffer[FX_MAX_LONG_NAME_LEN];
    ULONG           fx_path_current_entry;
} FX_PATH;
```

FX_CACHED_SECTOR

```
typedef FX_PATH FX_LOCAL_PATH;
typedef struct FX_CACHED_SECTOR_STRUCT
{
    UCHAR            *fx_cached_sector_memory_buffer;
    ULONG            fx_cached_sector;
    UCHAR            fx_cached_sector_buffer_dirty;
    UCHAR            fx_cached_sector_valid;
    UCHAR            fx_cached_sector_type;
    UCHAR            fx_cached_sector_reserved;
    struct           FX_CACHED_SECTOR_STRUCT
                         *fx_cached_sector_next_used;
} FX_CACHED_SECTOR;
```

FX_MEDIA

```
typedef struct        FX_MEDIA_STRUCT
{
    ULONG        fx_media_id;
    CHAR        *fx_media_name;
    UCHAR       *fx_media_memory_buffer;
    ULONG        fx_media_memory_size;
    UINT        fx_media_sector_cache_hashed;
    ULONG        fx_media_sector_cache_size;
    UCHAR       *fx_media_sector_cache_end;
    struct        FX_CACHED_SECTOR_STRUCT
                    *fx_media_sector_cache_list_ptr;
    ULONG        fx_media_sector_cache_hashed_sector_valid;
    ULONG        fx_media_sector_cache_dirty_count;
    UINT        fx_media_bytes_per_sector;
    UINT        fx_media_sectors_per_track;
    UINT        fx_media_heads;
    ULONG64        fx_media_total_sectors;
    ULONG        fx_media_total_clusters;

    #ifdef FX_ENABLE_EXFAT
        ULONG        fx_media_exfat_volume_serial_number;
        UINT        fx_media_exfat_file_system_revision;
        UINT        fx_media_exfat_volume_flag;
        USHORT        fx_media_exfat_drive_select;
        USHORT        fx_media_exfat_percent_in_use;
        UINT        fx_media_exfat_bytes_per_sector_shift;
        UINT        fx_media_exfat_sector_per_clusters_shift;
        UCHAR        fx_media_exfat_bitmap_cache[512];
        ULONG        fx_media_exfat_bitmap_start_sector;
        ULONG        fx_media_exfat_bitmap_cache_size_in_sectors;
        ULONG        fx_media_exfat_bitmap_cache_start_cluster;
        ULONG        fx_media_exfat_bitmap_cache_end_cluster;
        UINT        fx_media_exfat_bitmap_clusters_per_sector_shift;
        UINT        fx_media_exfat_bitmap_cache_dirty;
    #endif

    UINT        fx_media_reserved_sectors;
    UINT        fx_media_root_sector_start;
    UINT        fx_media_root_sectors;
    UINT        fx_media_data_sector_start;
    UINT        fx_media_sectors_per_cluster;
    UINT        fx_media_sectors_per_FAT;
    UINT        fx_media_number_of_FATs;
    UINT        fx_media_12_bit_FAT;
    UINT        fx_media_32_bit_FAT;
    ULONG        fx_media_FAT32_additional_info_sector;
    UINT        fx_media_FAT32_additional_info_last_available;

    #ifdef FX_DRIVER_USE_64BIT_LBA
        ULONG64        fx_media_hidden_sectors;
```

```
    #else
        ULONG        fx_media_hidden_sectors;
    #endif

    ULONG        fx_media_root_cluster_32;
    UINT        fx_media_root_directory_entries;
    ULONG        fx_media_available_clusters;
    ULONG        fx_media_cluster_search_start;
    VOID        *fx_media_driver_info;
    UINT        fx_media_driver_request;
    UINT        fx_media_driver_status;
    UCHAR       *fx_media_driver_buffer;

    #ifdef FX_DRIVER_USE_64BIT_LBA
        ULONG64        fx_media_driver_logical_sector;
    #else
        ULONG        fx_media_driver_logical_sector;
    #endif

    ULONG        fx_media_driver_sectors;
    ULONG        fx_media_driver_physical_sector;
    UINT        fx_media_driver_physical_track;
    UINT        fx_media_driver_physical_head;
    UINT        fx_media_driver_write_protect;
    UINT        fx_media_driver_free_sector_update;
    UINT        fx_media_driver_system_write;
    UINT        fx_media_driver_data_sector_read;
    UINT        fx_media_driver_sector_type;

    VOID        (*fx_media_driver_entry)(struct        fx_MEDIA_STRUCT *);
    VOID        (*fx_media_open_notify)(struct        fx_MEDIA_STRUCT *);
    VOID        (*fx_media_close_notify)(struct        fx_MEDIA_STRUCT *);
    struct        FX_FILE_STRUCT
                    *fx_media_opened_file_list;
    ULONG        fx_media_opened_file_count;
    struct        FX_MEDIA_STRUCT
                    *fx_media_opened_next,
                    *fx_media_opened_previous;

    #ifndef FX_MEDIA_STATISTICS_DISABLE
        ULONG        fx_media_directory_attributes_reads;
        ULONG        fx_media_directory_attributes_sets;
        ULONG        fx_media_directory_creates;
        ULONG        fx_media_directory_default_gets;
        ULONG        fx_media_directory_default_sets;
        ULONG        fx_media_directory_deletes;
        ULONG        fx_media_directory_first_entry_finds;
        ULONG        fx_media_directory_first_full_entry_finds;
        ULONG        fx_media_directory_information_gets;
        ULONG        fx_media_directory_local_path_clears;
        ULONG        fx_media_directory_local_path_gets;
        ULONG        fx_media_directory_local_path_restores;
        ULONG        fx_media_directory_local_path_sets;
        ULONG        fx_media_directory_name_tests;
        ULONG        fx_media_directory_next_entry_finds;
        ULONG        fx_media_directory_next_full_entry_finds;
        ULONG        fx_media_directory_renames;
        ULONG        fx_media_file_allocates;
        ULONG        fx_media_file_attributes_reads;
        ULONG        fx_media_file_attributes_sets;
        ULONG        fx_media_file_best_effort_allocates;
        ULONG        fx_media_file_closes;
        ULONG        fx_media_file_creates;
        ULONG        fx_media_file_deletes;
        ULONG        fx_media_file_opens;
        ULONG        fx_media_file_reads;
        ULONG        fx_media_file_relative_seeks;
        ULONG        fx_media_file_renames;
        ULONG        fx_media_file_seeks;
```

```
    ULONG        fx_media_file_truncates;
    ULONG        fx_media_file_truncate_releases;
    ULONG        fx_media_file_writes;
    ULONG        fx_media_aborts;
    ULONG        fx_media_flushes;
    ULONG        fx_media_reads;
    ULONG        fx_media_writes;
    ULONG        fx_media_directory_entry_reads;
    ULONG        fx_media_directory_entry_writes;
    ULONG        fx_media_directory_searches;
    ULONG        fx_media_directory_free_searches;
    ULONG        fx_media_fat_entry_reads;
    ULONG        fx_media_fat_entry_writes;
    ULONG        fx_media_fat_entry_cache_read_hits;
    ULONG        fx_media_fat_entry_cache_read_misses;
    ULONG        fx_media_fat_entry_cache_write_hits;
    ULONG        fx_media_fat_entry_cache_write_misses;
    ULONG        fx_media_fat_cache_flushes;
    ULONG        fx_media_fat_sector_reads;
    ULONG        fx_media_fat_sector_writes;
    ULONG        fx_media_logical_sector_reads;
    ULONG        fx_media_logical_sector_writes;
    ULONG        fx_media_logical_sector_cache_read_hits;
    ULONG        fx_media_logical_sector_cache_read_misses;
    ULONG        fx_media_driver_read_requests;
    ULONG        fx_media_driver_write_requests;
    ULONG        fx_media_driver_boot_read_requests;
    ULONG        fx_media_driver_boot_write_requests;
    ULONG        fx_media_driver_release_sectors_requests;
    ULONG        fx_media_driver_flush_requests;
#endif

#ifndef FX_MEDIA_DISABLE_SEARCH_CACHE
    ULONG        fx_media_directory_search_cache_hits;
#endif

#ifndef FX_SINGLE_THREAD
    TX_MUTEX        fx_media_protect;
#endif

#ifndef FX_MEDIA_DISABLE_SEARCH_CACHE
    UINT        fx_media_last_found_directory_valid;
    FX_DIR_ENTRY        fx_media_last_found_directory;
    FX_DIR_ENTRY        fx_media_last_found_entry;
    CHAR        fx_media_last_found_file_name[FX_MAX_LONG_NAME_LEN];
    CHAR        fx_media_last_found_name[FX_MAX_LAST_NAME_LEN];
#endif

FX_PATH        fx_media_default_path;
FX_FAT_CACHE_ENTRY    fx_media_fat_cache[FX_MAX_FAT_CACHE];
UCHAR        fx_media_fat_secondary_update_map[FX_FAT_MAP_SIZE];
ULONG        fx_media_reserved_for_user;
CHAR        fx_media_name_buffer[4*FX_MAX_LONG_NAME_LEN];

#ifdef FX_RENAME_PATH_INHERIT
    CHAR        fx_media_rename_buffer[FX_MAXIMUM_PATH];
    struct        FX_CACHED_SECTOR_STRUCT
    fx_media_sector_cache[FX_MAX_SECTOR_CACHE];
    ULONG        fx_media_sector_cache_hash_mask;
    ULONG        fx_media_disable_burst_cache;
#ifdef FX_ENABLE_FAULT_TOLERANT
    UCHAR        fx_media_fault_tolerant_enabled;
    UCHAR        fx_media_fault_tolerant_state;
    USHORT        fx_media_fault_tolerant_transaction_count;
    ULONG        fx_media_fault_tolerant_start_cluster;
    ULONG        fx_media_fault_tolerant_clusters;
    ULONG        fx_media_fault_tolerant_total_logs;
    UCHAR       *fx_media_fault_tolerant_memory_buffer;
    ULONG        fx_media_fault_tolerant_memory_buffer_size;
```

```
        ULONG        fx_media_fault_tolerant_memory_buffer_size;
        ULONG        fx_media_fault_tolerant_file_size;
        ULONG        fx_media_fault_tolerant_cached_FAT_sector;
        fx_media_fault_tolerant_cache[FX_FAULT_TOLERANT_CACHE_SIZE >> 2];
        ULONG        fx_media_fault_tolerant_cached_FAT_sector;
    #endif

    ULONG        fx_media_fat_reserved;
    ULONG        fx_media_fat_last;
    UCHAR        fx_media_FAT_type;
} FX_MEDIA;
```

FX_FILE

```
typedef struct FX_FILE_STRUCT
{
    ULONG         fx_file_id;
    CHAR *        fx_file_name;
    ULONG         fx_file_open_mode;
    UCHAR         fx_file_modified;
    ULONG         fx_file_total_clusters;
    ULONG         fx_file_first_physical_cluster;
    ULONG         fx_file_consecutive_cluster;
    ULONG         fx_file_last_physical_cluster;
    ULONG         fx_file_current_physical_cluster;
    ULONG64     fx_file_current_logical_sector;
    ULONG         fx_file_current_logical_offset;
    ULONG         fx_file_current_relative_cluster;
    ULONG         fx_file_current_relative_sector;
    ULONG64     fx_file_current_file_offset;
    ULONG64     fx_file_current_file_size;
    ULONG64     fx_file_current_available_size;

    #ifdef FX_ENABLE_FAULT_TOLERANT
        ULONG64     fx_file_maximum_size_used;
    #endif /*FX_ENABLE_FAULT_TOLERANT */

    FX_MEDIA    *fx_file_media_ptr;
    struct         FX_FILE_STRUCT
                    *fx_file_opened_next,
                    *fx_file_opened_previous;
    FX_DIR_ENTRY fx_file_dir_entry;
    CHAR        fx_file_name_buffer[FX_MAX_LONG_NAME_LEN];
    ULONG         fx_file_disable_burst_cache;
    VOID        (*fx_file_write_notify)(struct FX_FILE_STRUCT *);
} FX_FILE;
```

附录 D - Azure RTOS FileX ASCII 字符代码

2021/5/1 •

□□□□ ASCII □□□□

more significant nibble

less significant nibble	0_	1_	2_	3_	4_	5_	6_	7_
_0	NUL	DLE	SP	0	@	P	'	p
_1	SOH	DC1	!	1	A	Q	a	q
_2	STX	DC2	"	2	B	R	b	r
_3	ETX	DC3	#	3	C	S	c	s
_4	EOT	DC4	$	4	D	T	d	t
_5	ENQ	NAK	%	5	E	U	e	u
_6	ACK	SYN	&	6	F	V	f	v
_7	BEL	ETB	'	7	G	W	g	w
_8	BS	CAN	(	8	H	X	h	x
_9	HT	EM	)	9	I	Y	i	y
_A	LF	SUB	*	:	J	Z	j	z
_B	VT	ESC	+	;	K	[	K	}
_C	FF	FS	,	<	L	\	l	|
_D	CR	GS	-	=	M	]	m	}
_E	SO	RS	.	>	N	^	n	~
_F	SI	US	/	?	O	_	o	DEL