Contents

Azure RTOS ThreadX 概述

2021/4/30 •

Azure RTOS ThreadX 是 Microsoft 的高级工业级实时操作系统 (RTOS)，专用于深度嵌入式应用程序、实时应用程序和物联网应用程序。Azure RTOS ThreadX 提供高级计划、通信、同步、计时器、内存管理和中断管理功能。此外，Azure RTOS ThreadX 具有许多高级功能，包括 picokernel™ 体系结构、preemption-threshold™ 计划、event-chaining™、执行分析、性能指标和系统事件跟踪。Azure RTOS ThreadX 非常易于使用，适用于要求极其苛刻的嵌入式应用程序。自 2017 年起，Azure RTOS ThreadX 在各种产品（包括消费者设备、医疗电子设备和工业控制设备）上的部署次数超过了 62 亿次。

API 协议

Azure RTOS ThreadX API

- API 直观且一致
- 名词-动词命名约定
- 所有 API 均具有前导 tx_，可轻松识别为 Azure RTOS ThreadX
- 阻塞 API 具有可选的线程超时
- 许多 API 可以直接从应用程序 ISR 调用

Azure RTOS ThreadX 服务

- 动态线程创建
- 线程数无限制
- 主要的线程 API 包括：
  - tx_thread_create
  - tx_thread_delete
  - tx_thread_preemption_change
  - tx_thread_priority_change
  - tx_thread_relinquish
  - tx_thread_reset
  - tx_thread_resume
  - tx_thread_sleep
  - tx_thread_suspend
  - tx_thread_terminate
  - tx_thread_wait_abort
- 其他信息和性能 API

消息队列

- 动态队列创建
- 队列数无限制
- 按值（或通过指针按引用）复制消息
- 消息大小介于 1 到 16 个 32 位字词之间
- 在线程为空和已满时暂停线程（可选）
- 在全部暂停时触发超时（可选）
- 主要的消息队列 API 包括：
  - tx_queue_create
  - tx_queue_delete

    - tx_queue_flush
    - tx_queue_front_send
    - tx_queue_receive
    - tx_queue_send_notify
- 其他信息和性能 API

计数信号灯

- 动态信号灯创建
- 信号灯数量无限制
- 32 位计数信号灯（0 到 4,294,967,295）
- 支持使用者-生成者保护或资源保护
- 在信号灯不可用时暂停线程（可选）
- 在全部暂停时触发超时（可选）
- 主要的信号灯 API 包括：
    - tx_semaphore_create
    - tx_semaphore_delete
    - tx_semaphore_get
    - tx_semaphore_put
    - tx_semaphore_put_notify
- 其他信息和性能 API

Mutexes

- 动态互斥锁创建
- 互斥锁个数无限制
- 支持嵌套资源保护
- 支持继承优先级（可选）
- 在互斥锁不可用时暂停线程（可选）
- 在全部暂停时触发超时（可选）
- 主要的互斥锁 API 包括：
    - tx_mutex_create
    - tx_mutex_delete
    - tx_mutex_get
    - tx_mutex_put
- 其他信息和性能 API

事件标志

- 动态事件标志组创建
- 事件标志组数无限制
- 同步一个或多个线程
- 支持 Atomic get 和 Atomic clear
- 在发生 AND/OR 事件集时暂停多个线程（可选）
- 在全部暂停时触发超时（可选）
- 主要的事件标志 API 包括：
    - tx_event_flags_create
    - tx_event_flags_delete
    - tx_event_flags_get
    - tx_event_flags_set
    - tx_event_flags_set_notify

- 其他信息和性能 API

块内存池

- 动态块池创建
- 块池数量无限制
- 固定大小块或池的大小无限制
- 最快的内存分配/解除分配
- 在池为空时暂停线程（可选）
- 在全部暂停时触发超时（可选）
- 主要的块池 API 包括：
  - tx_block_pool_create
  - tx_block_pool_delete
  - tx_block_allocate
  - tx_block_release
- 其他信息和性能 API

字节内存池

- 动态字节池创建
- 字节池数量无限制
- 字节池大小无限制
- 最灵活的变长内存分配/解除分配
- 支持分配大小位置
- 在池为空时暂停线程（可选）
- 在全部暂停时触发超时（可选）
- 主要的字节池 API 包括：
  - tx_byte_pool_create
  - tx_byte_pool_delete
  - tx_byte_allocate
  - tx_byte_release
- 其他信息和性能 API

应用程序计时器

- 动态计时器创建
- 计时器数量无限制
- 支持周期性计时器或单次计时器
- 周期性计时器可能具有不同的初始有效期值
- 在激活或停用计时器时不执行搜索
- 所有计时器均由一个硬件计时器中断驱动
- 主要的计时器 API 包括：
  - tx_timer_create
  - tx_timer_delete
  - tx_timer_activate
  - tx_timer_change
  - tx_timer_deactivate
- 其他信息和性能 API

Azure RTOS ThreadX 核心计划程序

- 最少占用 2KB 闪存和 1KB RAM
- 快速的亚毫秒级上下文切换

- 完全确定性，与线程数无关
- 基于优先级的完全抢占式计划
- 32 个默认优先级，最多可选择定义 1024 个优先级
- 优先级内的协作式计划 (FIFO)
- 抢占式阈值技术
- 可选的计时器服务，包括：
  - 每线程可选时间片
  - 在全部阻塞时触发超时（可选）
  - 在硬件计时器中断时要求调用 API
- 执行分析
- 系统级跟踪
- 安全性通过许多标准的认证

部署次数最多的 RTOS

根据领先的 M2M 市场情报公司 VDC Research 的调查，Azure RTOS ThreadX 在全球的部署次数超过了 62 亿次。Azure RTOS ThreadX 的普及证明了它具有卓越的可靠性、质量、大小、性能、高级功能、易用性和总体上市时间。

> “自成立以来，我们公司一直跟踪 ThreadX 在无线市场和物联网市场的增长轨迹，ThreadX 在各行各业的广泛应用给我们留下了越来越深刻的印象。” - VDC Research 执行副总裁 Chris Rommel

占用空间少

Azure RTOS ThreadX 的占用空间非常小，最少只需要一个 2KB 的指令区域和 1 KB 的 RAM。这在很大程度上归功于其非分层 picokernel™ 体系结构和自动缩放。自动缩放是指在链接时仅将应用程序使用的服务（和支持基础结构）包括到最终映像中。

下面是 Azure RTOS ThreadX 的一些典型大小特征：

AZURE RTOS THREADX **	**** (******)
核心服务（必需）	2,000
队列服务	900
事件标志服务	900
信号灯服务	450
互斥锁服务	1,200
块内存服务	550
字节内存服务	900

快速执行

Azure RTOS ThreadX 可在大多数主流处理器上实现亚毫秒级上下文切换，并且总体性能比其他商业 RTOS 大大提高了。除了快速，Azure RTOS ThreadX 还具有高度确定性。无论准备就绪的线程有 200 个还是只有一个，它都能实现同样的高性能。

下面是 Azure RTOS ThreadX 的一些典型性能特征：

- 快速启动：Azure RTOS ThreadX 可在 120 个周期内启动。
- 禁用基本错误检查（可选）：编译时可以跳过基本的 Azure RTOS ThreadX 错误检查。当应用程序代码已通过验证，且不再需要对每个参数进行错误检查时，这很有用。请注意，可在编译单元上禁用此检查，而不是在系统范围禁用。
- picokernel™ 设计：服务不彼此叠加，从而消除了不必要的函数调用开销。
- * 优化中断处理：除非需要抢占，否则只有在进入/退出 ISR 时才会保存/还原暂存寄存器。
- 优化 API 处理：

AZURE RTOS THREADX	*********
暂停线程	0.6
继续线程	0.6
发送队列	0.3
接收队列	0.3
获取信号灯	0.2
放置信号灯	0.2
上下文切换	0.4
中断响应	0.0-0.6

*性能指标基于运行频率为 200MHz 的典型处理器。

通过 TUV 和 UL 许多安全标准的预认证

Azure RTOS ThreadX 和 Azure RTOS ThreadX SMP 已通过 SGS-TUV Saar 的认证，符合 IEC-61508 SIL 4、IEC-62304 SW Safety Class C、ISO 26262 ASIL D 和 EN 50128 标准，可用于安全关键型系统。该认证证明：Azure RTOS ThreadX 和 Azure RTOS ThreadX SMP 可用于开发符合最高安全完整性级别 IEC-61508、IEC-62304、ISO 26262 和 EN 50128 标准的安全相关软件，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-TUV Saar 由德国的 SGS-Group 和 TUV 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、ISO 26262 和 EN 50128）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、汽车和铁路控制系统的功能安全。

Azure RTOS ThreadX 和 Azure RTOS ThreadX SMP 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。UL 是一家全球性、独立的安全科学公司。他们历史悠久并且拥有极其丰富的专业知识，从电力的公共应用到可持续发展的突破成就再到可再生能源和纳米技术，他们不断创新安全解决方案。

与 TUV 和 UL 认证相关的项目（证书、安全手册和测试报告等）可供销售。

EAL4+ 通用标准安全认证

Azure RTOS 已获得 EAL4+ 通用标准安全认证。评估对象（Target of Evalution，TOE）包括 Azure RTOS ThreadX、Azure RTOS NetX-Duo、Azure RTOS NetX Secure TLS 和 Azure RTOS NetX MQTT。其代表了深度嵌入式传感器、设备、边缘路由器和网关所需的最典型物联网协议。

用于 Azure RTOS 安全认证的 IT 安全评估机构是 Brightsight BV，认证机构是 SERTIT。

简单、易于使用

Azure RTOS ThreadX 非常易于使用。Azure RTOS ThreadX API 既直观又功能强大。API 名称由实词组成，而非由十分简略的缩写词组成，后者在其他 RTOS 产品中很常见。所有 Azure RTOS ThreadX API 均具有前导 `tx_`，并遵循名词-动词命名约定。此外，整个 API 具有功能一致性。例如，所有暂停的 API 均具有可选的超时期限，其功能对所有 API 都一致。

生成 Azure RTOS ThreadX 应用程序非常简单。应用程序需要包含 tx_api.h，从主线程调用 `tx_kernel_enter`，定义 `tx_application_define` 函数并创建一个线程，定义线程入口点函数并链接到 Azure RTOS ThreadX 库（通常是 tx.a）。

Azure RTOS ThreadX 还提供质量非常高的文档。

高级技术

Azure RTOS ThreadX 是高级技术，其最值得关注的功能是抢占式阈值计划。此功能是 Azure RTOS ThreadX 独有的，并已成为广泛的学术研究课题。例如，请参阅由康考迪亚大学的 Yun Wang 与匹兹堡大学的 Manas Saksena 联合发表的《Scheduling Fixed-Priority Tasks with Preemption Threshold》。

考虑 Azure RTOS ThreadX 的功能：

- 完整全面的多任务处理功能
  - 线程、应用程序计时器、消息队列、计数信号灯、互斥锁、事件标志、块内存池和字节内存池
- 基于优先级的抢占式计划
- 优先级灵活性 - 最多可定义 1024 个优先级
- 协作式计划
- Preemption-Threshold™ - Azure RTOS ThreadX 的独有功能，有助于减少上下文切换并帮助确保可计划性（根据学术研究）
- 通过 Azure RTOS ThreadX MODULES 进行内存保护
- 完全确定性
- 事件跟踪 - 捕获最后 n 个系统事件/应用程序事件

- Event-chaining™ - 为每个 Azure RTOS ThreadX 通信或同步对象注册特定于应用程序的"通知"回调函数
- 具有可选内存保护的 Azure RTOS ThreadX MODULES
- 运行时性能指标
  - 线程恢复数
  - 线程暂停数
  - 请求的线程抢占数
  - 异步线程中断抢占数
  - 线程优先级反转数
  - 线程放弃次数
- Execution Profile Kit (EPK)
- 单独中断堆栈
- 运行时堆栈分析
- 优化计时器中断处理

多核支持(AMP 和 SMP)

标准 Azure RTOS ThreadX 通常以非对称多重处理 (AMP) 方式使用，在这种情况下，Azure RTOS ThreadX 的单独副本和应用程序(或 Linux)在每个核心上执行，并通过共享内存或处理器间通信机制(例如 Azure RTOS ThreadX 所支持的 OpenAMP)相互通信。这是使用 Azure RTOS ThreadX 的最典型多核配置，如果应用程序能够有效地加载处理器，这将是最有效的配置。

对于高度动态加载处理器的环境，Azure RTOS ThreadX 对称多重处理 (SMP) 可用于以下处理器系列：

- ARM Cortex-Ax
- ARM Cortex-Rx
- ARM Cortex-A5x(64 位)
- MIPS 34K、1004K 和 interAptiv
- PowerPC
- Synopsys ARC HS
- x86

Azure RTOS ThreadX SMP 跨 n 个处理器执行动态负载均衡，并允许任何核心上的任何线程访问所有 Azure RTOS ThreadX 资源(队列、信号灯、事件标志和内存池等)。Azure RTOS ThreadX SMP 在所有核心上启用全部 Azure RTOS ThreadX API，并引入了以下适用于 SMP 操作的新 API：

- `UINT tx_thread_smp_core_exclude(TX_THREAD *thread_ptr, ULONG exclusion_map);`
- `UINT tx_thread_smp_core_exclude_get(TX_THREAD *thread_ptr, ULONG *exclusion_map_ptr);`
- `UINT tx_thread_smp_core_get(void);`
- `UINT tx_timer_smp_core_exclude(TX_TIMER *timer_ptr, ULONG exclusion_map);`
- `UINT tx_timer_smp_core_exclude_get(TX_TIMER *timer_ptr, ULONG *exclusion_map_ptr);`

通过 Azure RTOS ThreadX MODULES 进行内存保护

使用称为 Azure RTOS ThreadX MODULES 的附加产品可将一个或多个应用程序线程捆绑成一个"模块"，该模块可动态加载并在目标上运行(或就地执行)。

使用这些模块可进行现场升级、缺陷修复和程序分区，从而使大型应用程序仅占用活动线程所需的内存。

这些模块还具有完全独立于 Azure RTOS ThreadX 的地址空间。这样，Azure RTOS ThreadX 就可以通过 MPU 或 MMU 对模块进行内存保护，从而阻止模块外部的意外访问损坏任何其他软件组件。

最快上市时间

Azure RTOS ThreadX 易于安装、学习、使用、调试、验证、认证和维护。因此，根据 Embedded Market Forecasters (EMF) 的调查，Azure RTOS ThreadX 在过去七年里一直是上市时间更短的 RTOS。这些调查一致地表明，在使用 Azure RTOS ThreadX 设计的产品中，70% 的产品能够及时进入市场，这超越了所有其他 RTOS。

我们的上市时间能够一贯保持优势的原因如下：

- 高质量文档
- 提供完整的源代码
- 易于使用的 API
- 全面且高级的功能集
- 广泛的第三方工具集成 - 尤其是 IAR 的 Embedded Workbench™

免版税

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

最优质的完整源代码

从一开始，Azure RTOS ThreadX 就设计成随完整 C 源代码一起分发的工业级 RTOS。Azure RTOS ThreadX 源代码在质量和易于理解方面树立了标杆。此外，Azure RTOS NetX 约定每个文件具有一个功能，正因为此，你可以轻松导览至源代码。

Azure RTOS ThreadX 遵守严格的编码约定，并满足每行 C 代码都具有有意义的注释这一要求。此外，Azure RTOS ThreadX 源代码已经过最高标准的认证。

符合 MISRA

Azure RTOS ThreadX 和 Azure RTOS ThreadX SMP 的源代码符合 MISRA-C:2004 以及 MISRA C:2012。MISRA C 是一组编程准则，面向使用 C 编程语言的关键系统。最初的 MISRA C 准则主要面向汽车业应用；但是，现在人们广泛认可 MISRA C 适用于任何安全关键应用。Azure RTOS ThreadX 符合 MISRA-C:2004 和 MISRA C:2012 的所有必需规则和强制性规则。

支持最常用的体系结构

Azure RTOS ThreadX 开箱即用，可在大多数主流 32 位/64 位微处理器上运行，已经过充分测试且完全受支持，其中包括以下体系结构：

- Analog Devices：SHARC、Blackfin、CM4xx
- Andes Core：RISC-V
- Ambiqmicro：Apollo MCU
- ARM：ARM7、ARM9、ARM11、Cortex-M0/M3/M4/M7/A15/A5/A7/A8/A9/A5x（64 位）/A7x（64 位）/R4/R5、TrustZone ARMv8-M
- Cadence：Xtensa、Diamond
- CEVA：PSoC、PSoC 4、PSoC 5、PSoC 6、FM0+、FM3、MF4、WICED WiFi

- Cypress：RISC-V
- EnSilica：eSi-RISC
- Infineon：XMC1000、XMC4000、TriCore
- Intel；Intel FPGA：x36/Pentium、XScale、NIOS II、Cyclone、Arria 10
- Microchip：AVR32、ARM7、ARM9、Cortex-M3/M4/M7、SAM3/4/7/9/A/C/D/E/G/L/SV、PIC24/PIC32
- Microsemi：RISC-V
- NXP：LPC、ARM7、ARM9、PowerPC、68K、i.MX、ColdFire、Kinetis Cortex-M3/M4
- Renesas：SH、HS、V850、RX、RZ、Synergy
- Silicon Labs：EFM32
- Synopsys：ARC 600、700、ARC EM、ARC HS
- ST：STM32、ARM7、ARM9、Cortex-M3/M4/M7
- TI：C5xxx、C6xxx、Stellaris、Sitara、Tiva-C
- Wave Computing：MIPS32 4K、24K、34K、1004K、MIPS64 5K、microAptiv、interAptiv、proAptiv、M-Class
- Xilinx：MicroBlaze、PowerPC 405、ZYNQ、ZYNQ UltraSCALE

支持大部分常用工具

Azure RTOS ThreadX 支持大部分常用的嵌入式开发工具，包括 IAR 的 Embedded Workbench™，该工具还提供最全面的 Azure RTOS ThreadX 内核感知功能。其他工具集成包括 GNU (GCC)、ARM DS-5/uVision®、Green Hills MULTI®、Wind River Workbench™、Imagination Codescape、Renesas e2studio、Metaware SeeCode™、NXP CodeWarrior、Lauterbach TRACE32®、TI Code-Composer Studio、CrossCore 和所有模拟设备。

关于 Azure RTOS ThreadX 指南

2021/4/30 •

本指南旨在全面介绍 Azure RTOS ThreadX(Microsoft 的高性能实时内核)。

本指南适用对象为嵌入式实时软件的开发者。开发者应熟悉标准实时操作系统函数和 C 编程语言。

组织

第 1 章 - 简要概述 Azure RTOS ThreadX 及其与实时嵌入式开发的关系

第 2 章 - 介绍在应用程序中安装和使用 Azure RTOS ThreadX 的基本步骤(开箱即用)

第 3 章 - 详细介绍 Azure RTOS ThreadX(高性能实时内核)的功能操作

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

第 5 章 - 介绍如何编写 Azure RTOS ThreadX 应用程序的 I/O 驱动程序

第 6 章 - 介绍随每个 Azure RTOS ThreadX 处理器支持包一起提供的演示应用程序

附录 A - Azure RTOS ThreadX API

附录 B - Azure RTOS ThreadX 常数

附录 C - Azure RTOS ThreadX 数据类型

附录 D - ASCII 图表

指南约定

斜体 - 字样表示书名，强调重要词语以及指示参数。

粗体 - 字样表示关键字、常数、类型名称、用户界面元素、变量名称，并进一步强调重要词语。

斜体和粗体 - 字样表示文件名和函数名称。

> **IMPORTANT**
> 信息符号，旨在让用户注意会影响性能或功能的重要信息或附加信息。

> **WARNING**
> 警告符号，旨在让用户注意开发者应小心避免的情况，这类情况可导致灾难性错误。

Azure RTOS ThreadX 数据类型

除了自定义的 Azure RTOS ThreadX 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 ThreadX 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。这样做是为了确保不同 C 编译器之间的可移植性。有关准确实现的方法，请参阅源提供程序随附的 tx_port.h 文件。

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

UINT	基本无符号整数。此类型必须支持 8 位无符号数据，但是它将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	通常为标准的 8 位字符类型。

Azure RTOS ThreadX 源中还使用了其他数据类型。这些数据类型也列于 tx_port.h 文件中。

客户支持中心

如有疑问或需要帮助，请使用此处的步骤通过 Azure 门户提交支持票证。请在电子邮件中提供以下信息，以便我们可以更有效地解决您的支持请求：

1. 详细说明有关问题，包括发生的频率以及是否可以可靠地重现该问题。
2. 详细说明发生问题前对应用程序和/或 Azure RTOS ThreadX 做的任何更改。
3. 可在分发的 tx_port.h 文件中找到的 _tx_version_id 字符串的内容。此字符串将为我们提供有关运行时间环境的重要信息。
4. RAM 中 _Tx_build_options ULONG 变量的内容。此变量将为我们提供有关 Azure RTOS ThreadX 库生成方式的信息。

第 1 章 - Azure RTOS ThreadX 简介

2021/4/29 •

Azure RTOS ThreadX 是专门为嵌入式应用程序设计的高性能实时内核。本章提供了该产品的简介，并说明了其应用和优势。

ThreadX 的独特功能

与其他实时内核不同，ThreadX 功能多样。通过使用强大的 CISC、RISC 和 DSP 处理器的应用程序，可以轻松地在基于小型微控制器的应用程序之间扩展。

ThreadX 可基于其基础体系结构进行扩展。因为 ThreadX 服务是作为 C 库实现的，所以只有应用程序实际使用的那些服务被引入了运行时映像。因此，ThreadX 的实际大小完全由应用程序决定。对于大多数应用程序，ThreadX 的指令映像的大小在 2 KB 至 15 KB 之间。

picokernel™ 体系结构

ThreadX 服务没有像传统的微内核体系结构那样将内核函数相互层叠，而是直接将其插入核心。由此产生了最快的上下文切换和服务呼叫性能。我们将此非分层设计称为 picokernel 体系结构。

ANSI C 源代码

ThreadX 主要是在 ANSI C 中编写的。用户需要少量的汇编语言来定制内核，从而满足基础目标处理器的需求。此设计使用户可以在很短的时间内（通常在几周内）将 ThreadX 移植到新的处理器系列！

高级技术

下面是 ThreadX 高级技术的亮点。

- 简单的 picokernel 体系结构
- 自动扩展（占用空间少）
- 确定性处理
- 快速实时性能
- 抢先式和协作式计划
- 灵活的线程优先级支持
- 动态系统对象创建
- 无限量的系统对象
- 经过优化的中断处理
- 抢占阈值 (Preemption-threshold™)
- 优先级继承
- 事件链接 (Event-chaining™)
- 快速软件计时器
- 运行时内存管理
- 运行时性能监视
- 运行时堆栈分析
- 内置系统跟踪
- 广泛的处理器支持
- 广泛的开发工具支持
- 字节顺序完全中性

不是黑盒

ThreadX 的大多数发行版都包含完整的 C 源代码以及特定于处理器的汇编语言。这消除了许多商业内核所出现的“黑盒”问题。借助 ThreadX，应用程序开发人员可以看到内核正在进行的确切操作。没有任何秘密可言！

源代码还允许进行特定于应用程序的修改。尽管不建议这么做，但如果真的需要，具备修改内核的能力当然是有益的。

对于习惯使用自己的内部内核的开发人员而言，这些功能尤其令人欣慰。他们期望拥有源代码和修改内核的能力。ThreadX 是适用于这类开发人员的终极内核。

RTOS 标准

由于 ThreadX 的多功能性、高性能的 picokernel 体系结构、先进的技术以及经证实的可移植性，目前已部署 ThreadX 的设备超过了 20 亿。这实际上使 ThreadX 成为深度嵌入式应用程序的 RTOS 标准。

安全认证

TÜV 认证

ThreadX 已被 SGS-TÜV Saar 认证可用于安全关键型系统，并且符合 IEC61508 和 IEC-62304 标准。该认证证明：ThreadX 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 SGSGroup 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 ThreadX 进行了认证，确定其可用于安全关键型汽车系统。此外，ThreadX 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。

此外，SGS-TÜV Saar 已对 ThreadX 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

> **NOTE**
> 请联系我们，了解 ThreadX 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。

符合 MISRA C

MISRA C 是一组编程准则，面向使用 C 编程语言的关键系统。最初的 MISRA C 准则主要面向汽车业应用；但是，现在人们广泛认可 MISRA C 适用于任何安全关键应用。ThreadX 符合 MISRA-C:2004 和 MISRA C:2012 的所有“必需”规则和“强制性”规则。ThreadX 还符合除三个“公告”规则之外的所有规则。有关更多详细信息，请参阅 ThreadX_MISRA_Compliance.pdf 文档。

UL 认证

ThreadX 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

> **NOTE**
> 请联系 Microsoft，了解 ThreadX 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。

认证包

ThreadX Certification Pack™ 是100% 完整、统包式、行业特定的独立包，提供所有必要的 ThreadX 证据，以认证或成功申报基于 ThreadX 的产品，致力于达到安全关键型航空、医疗和工业系统所需的最高可靠性和关键性等级。支持的认证包括 DO-178B、ED-12B、DO-278、FDA510(k)、IEC62304、IEC-60601、ISO-14971、UL-1998、IEC-61508、CENELEC EN50128、BS50128 和 49CFR236。有关认证包的更多信息，请联系 Microsoft。

嵌入式应用程序

嵌入式应用程序在诸如无线通信设备、汽车引擎、激光打印机、医疗器械等产品内置的微处理器上执行。嵌入式应用程序的另一个区别在于它们的软件和硬件有着特定的用途。

实时软件

当对应用软件施加时间限制时，它被称为实时软件。由于嵌入式应用程序与外部事件的固有交互，因此几乎总是实时的。

多任务

如前所述，嵌入式应用程序有着特定的用途。为了实现此目的，软件必须执行各种任务。任务是执行特定职责的应用程序的半独立部分。同样，事实上某些任务比其他任务更重要。嵌入式应用程序的主要难题之一是在各种应用程序任务之间分配处理器。在竞争任务之间分配处理器是 ThreadX 的主要目的。

任务与线程

关于任务的另一个区别在于，“任务”这个术语是以多种方式使用的。有时它意味着可单独加载的程序。在其他情况下，它可以指内部程序段。因此，在现代操作系统中，有两个术语或多或少地替代了“任务”：“进程”和“线程”。“进程”是具有自己的地址空间的完全独立的程序，而“线程”是在进程内执行的半独立程序段。线程共享相同的进程地址空间。与线程管理相关的开销是最小的。

大多数嵌入式应用程序负担不起与面向进程的成熟操作系统相关的开销(内存和性能)。此外，较小的微处理器并没有支持真正面向进程的操作系统的硬件体系结构。出于这些原因，ThreadX 实现了一个线程模型，对于大多数实时嵌入式应用程序来说，该模型非常高效实用。

为避免混淆，ThreadX 不使用"任务"这个术语。取而代之的是，使用更具描述性和现代性的名称"线程"。

ThreadX 的好处

使用 ThreadX 为嵌入式应用程序带来了许多好处。当然，主要好处在于如何为嵌入式应用程序线程分配处理时间。

更高的响应能力

在出现像 ThreadX 这样的实时内核之前，大多数嵌入式应用程序都使用简单的控制循环(通常来自 C main 函数内部)来分配处理时间。在非常小或简单的应用程序中，这种方法仍在使用。但是，在大型或复杂的应用程序中，这是不切实际的，因为对任何事件的响应时间是一个函数，是一次通过控制循环的传递的最差处理时间的函数。

更糟糕的是，每当对控制循环进行修改时，应用程序的时序特性都会发生变化。这使应用程序本身就不稳定，难以维护和改进。

ThreadX 提供对重要外部事件的快速确定性响应时间。ThreadX 通过其基于优先级的抢先式计划算法来实现这一点，该算法允许优先级较高的线程抢先于正在执行的优先级较低的线程。因此，最差的响应时间接近执行上下文切换所需的时间。这不仅是确定性的，而且还是非常快速的。

软件维护

ThreadX 内核使应用程序开发人员能够专注于其应用程序线程的特定要求，而无需担心更改应用程序其他区域的计时。此功能还简化了使用 ThreadX 的应用程序的修复或增强。

更高的吞吐量

解决控制循环响应时间问题的一种方法可以是添加更多轮询。这样可以提高响应能力，但仍不能保证最差响应时间保持不变，而且也不利于推动应用程序的未来修改。而且，由于额外的轮询，处理器现在正在执行更多不必要的处理。所有这些不必要的处理都会降低系统的总体吞吐量。

关于开销，有趣的一点是，许多开发人员都认为多线程环境(如 ThreadX)会增加开销，并对总系统吞吐量产生负面影响。但是在某些情况下，多线程实际上通过消除在控制循环环境中发生的所有冗余轮询减少了开销。与多线程内核相关的开销通常是上下文切换所需时间的函数。如果上下文切换时间少于轮询过程，则 ThreadX 提供的解决方案可能会减少开销并提高吞吐量。无论应用程序的复杂性和大小如何，这都使 ThreadX 成为明智之选。

处理器隔离

ThreadX 在应用程序与基础处理器之间提供可靠的独立于处理器的接口。这使开发人员能够专注于应用程序，而不是花费大量时间研究硬件详细信息。

划分应用程序

在基于控件循环的应用程序中，每个开发人员都必须对整个应用程序的运行时行为和要求有着深入的了解。这是因为处理器分配逻辑分散在整个应用程序中。随着应用程序的大小或复杂性的增加，所有开发人员都无法记住整个应用程序的精确处理要求。

ThreadX 让每个开发人员摆脱了与处理器分配相关的烦恼，并允许他们专注于嵌入式应用程序的特定部分。此外，ThreadX 强制将应用程序划分为明确定义的线程。将应用程序划分为多个线程本身可使开发变得更简单。

易用性

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

缩短上市时间

ThreadX 的这一切优点都加速了软件开发过程。ThreadX 负责处理大多数处理器问题和最常见的安全认证，从而从开发计划中省去了这些工作。所有这些都使上市时间缩短！

保护软件投资

由于其体系结构，ThreadX 可轻松移植到新的处理器和/或开发工具环境。基于这一特性，再加上 ThreadX 可以将应用程序与基础处理器的详细信息隔离的事实，从而使 ThreadX 应用程序变得高度可移植。最终保证了应用程序的迁移路径，并保护了原始开发投资。

第 2 章 - 安装和使用 Azure RTOS ThreadX

2021/4/29 •

本章旨在介绍与安装、设置和使用高性能 Azure RTOS ThreadX 内核相关的各种问题。

主机注意事项

嵌入式软件通常是在 Windows 或 Linux (Unix) 主机计算机上开发的。在对应用程序进行编译和链接并将其放置在主机上之后，将应用程序下载到目标硬件，以执行它。

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

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

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

目标注意事项

ThreadX 要求目标具有 2 KB 和 20 KB 只读内存 (ROM)。对于 ThreadX 系统堆栈和其他全局数据结构，它还需要 1 到 2KB 的目标的随机访问内存 (RAM)。

对于与计时器相关的函数（如服务调用超时、时间切片和要运行的应用程序计时器），基础目标硬件必须提供定期中断源。如果处理器具有此功能，则 ThreadX 会使用它。否则，如果目标处理器无法生成定期中断，则用户的硬件必须提供此功能。计时器中断的设置和配置通常位于 ThreadX 分发的 tx_initialize_low_level 程序集文件中。

> **NOTE**
> 即使没有可用的定期计时器中断源，ThreadX 仍可正常运行。但是，与计时器相关的服务都无法正常工作。

产品分发

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

下面列出了该存储库中的几个重要文件。

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

> **NOTE**
> 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为 Linux (Unix) 开发平台。

ThreadX 安装

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

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

或者，也可以使用 GitHub 主页上的“下载”按钮来下载存储库的副本。

你还可以在联机存储库的首页上找到有关生成 ThreadX 库的说明。

> **NOTE**
> *应用程序软件需要访问 ThreadX 库文件（通常为 tx.a 或 tx.lib）和 C 包含文件 tx_api.h 和 tx_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 ThreadX

若要使用 ThreadX，应用程序代码必须在编译期间包含 tx_api.h，并与 ThreadX 运行时库 tx.a（或 tx.lib）链接。

生成 ThreadX 应用程序需要四个步骤。

1. 在所有使用 ThreadX 服务或数据结构的应用程序文件中包含 tx_api.h 文件。
2. 创建标准 C main 函数。此函数必须最终调用 tx_kernel_enter 才能启动 ThreadX。在输入内核之前，可能会添加不涉及 ThreadX 的应用程序特定的初始化。

   > **IMPORTANT**
   > *ThreadX entry 函数 tx_kernel_enter 不返回。因此，请确保在其之后不进行任何处理或函数调用。

3. 创建 tx_application_define 函数。这是创建初始系统资源的位置。系统资源的示例包括线程、队列、内存池、事件标志组、互斥体和信号灯。
4. 编译应用程序源并与 ThreadX 运行时库 tx.lib 链接。生成的图像可下载到目标并执行！

小型示例系统

图 1 中的小型示例系统显示了优先级为 3 的单个线程的创建。线程执行，递增计数器，然后休眠一个时钟周期。此过程会永远继续下去。

```
#include "tx_api.h"
unsigned long my_thread_counter = 0;
TX_THREAD my_thread;
main( )
{
    /* Enter the ThreadX kernel. */
    tx_kernel_enter( );
}
void tx_application_define(void *first_unused_memory)
{
    /* Create my_thread! */
    tx_thread_create(&my_thread, "My Thread",
    my_thread_entry, 0x1234, first_unused_memory, 1024,
    3, 3, TX_NO_TIME_SLICE, TX_AUTO_START);
}
void my_thread_entry(ULONG thread_input)
{
    /* Enter into a forever loop. */
    while(1)
    {
        /* Increment thread counter. */
        my_thread_counter++;
        /* Sleep for 1 tick. */
        tx_thread_sleep(1);
    }
}
```

图 1 应用程序开发的模板

尽管这是一个简单的示例，但它为真正的应用程序开发提供了一个很好的模板。

疑难解答

每个 ThreadX 端口都随演示应用程序一起提供。首先运行演示系统，无论是在实际的目标硬件上，还是在模拟环境中，这始终是个不错的主意。

如果演示系统未正确执行，以下是一些故障排除提示。

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. 重新生成 ThreadX 库，其中定义了 TX_ENABLE_STACK_CHECKING。这将启用内置的 ThreadX 堆栈检查。
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 ThreadX 生成 ThreadX 库和应用程序时，有几个配置选项。可以在应用程序源、命令行或 tx_user.h 包含文件中定义以下选项。

> **IMPORTANT**
> 只有当应用程序和 ThreadX 库在构建时定义了 TX_INCLUDE_USER_DEFINE_FILE，才会应用在 tx_user.h *中定义的选项*。

最小配置

对于最小的代码大小，应考虑以下 ThreadX 配置选项（缺少所有其他选项）。

```
TX_DISABLE_ERROR_CHECKING
TX_DISABLE_PREEMPTION_THRESHOLD
TX_DISABLE_NOTIFY_CALLBACKS
TX_DISABLE_REDUNDANT_CLEARING
TX_DISABLE_STACK_FILLING
TX_NOT_INTERRUPTABLE
TX_TIMER_PROCESS_IN_ISR
```

最快配置

为实现最快的执行，在以前的最小配置中使用相同的配置选项，但也将这些选项考虑在内。

```
TX_REACTIVATE_INLINE
TX_INLINE_THREAD_RESUME_SUSPEND
```

介绍了详细的配置选项。

全局时间源

对于其他 Microsoft Azure RTOS 产品（FileX、NetX、GUIX、USBX 等），ThreadX 定义表示一秒的 ThreadX 计时器时钟周期数。其他产品基于此常量派生其时间要求。默认情况下，该值为 100（假设 10ms 定期中断）。用户可以通过在 tx_port.h 或在 IDE 或命令行中以所需值定义 TX_TIMER_TICKS_PER_SECOND 来覆盖此值。

详细的配置选项

TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO

如果定义了此选项，则可以在字节池上收集性能信息。默认情况下，未定义此选项。

TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO

如果定义了此选项，则可以在字节池上收集性能信息。默认情况下，未定义此选项。

TX_DISABLE_ERROR_CHECKING

跳过基本服务调用错误检查。在应用程序源中定义时，将禁用所有基本参数错误检查。这可能会将性能提高多达 30%，还可能会减少图像大小。

> **NOTE**
> 仅当应用程序可以绝对保证所有输入参数（包括从外部输入派生的输入参数）在所有情况下始终有效，才可以安全地禁用错误检查。如果在禁用错误检查的情况下向 API 提供无效输入，则生成的行为是未定义的，并且可能会导致内存损坏或系统崩溃。

> **NOTE**
> 不受禁用错误检查影响的 ThreadX API 返回值在第 4 章的每个 API 说明的“返回值”部分中以粗体列出。如果通过使用 TX_DISABLE_ERROR_CHECKING 选项禁用了错误检查，则非加粗返回值无效。

TX_DISABLE_NOTIFY_CALLBACKS

定义后，将对各种 ThreadX 对象禁用通知回调。使用此选项可以略微减少代码大小并提高性能。默认情况下，未定义此选项。

TX_DISABLE_PREEMPTION_THRESHOLD

定义后，将禁用抢占阈值功能，并略微减少代码大小并提高性能。当然，抢占阈值功能不再可用。默认情况下，未定义此选项。

TX_DISABLE_REDUNDANT_CLEARING

定义后，删除用于将 ThreadX 全局 C 数据结构初始化为零的逻辑。仅当编译器的初始化代码将所有未初始化的 C 全局数据设置为零时，才应使用此选项。在初始化期间，使用此选项可以略微减少代码大小并提高性能。默认情况下，未定义此选项。

TX_DISABLE_STACK_FILLING

定义后，禁止在创建时将 0xEF 值置于每个线程的堆栈的每个字节中。默认情况下，未定义此选项。

TX_ENABLE_EVENT_TRACE

定义后，ThreadX 启用事件收集代码，用于创建 TraceX 跟踪缓冲区。

TX_ENABLE_STACK_CHECKING

定义后，将启用 ThreadX 运行时堆栈检查，其中包括分析已使用的堆栈量，以及堆栈区域前后的数据模式"防护"检查。如果检测到堆栈错误，则会调用已注册应用程序堆栈错误处理程序。此选项会导致系统开销和代码大小略有增加。有关详细信息，请查看 tx_thread_stack_error_notify API 函数。默认情况下，未定义此选项。

TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO

定义后，可以收集有关事件标志组的性能信息。默认情况下，未定义此选项。

TX_INLINE_THREAD_RESUME_SUSPEND

定义后，ThreadX 通过行内代码改进 tx_thread_resume 和 tx_thread_suspend API 调用。这会增加代码大小，但会增强这两个 API 调用的性能。

TX_MAX_PRIORITIES

定义 ThreadX 的优先级别。合法值的范围为 32 到 1024（含），且必须能被 32 整除。增加支持的优先级别数量会使每组 32 个优先级别的 RAM 用量增加 128 字节。但是，对性能的影响可忽略不计。默认情况下，此值设置为 32 个优先级别。

TX_MINIMUM_STACK

定义最小堆栈大小（以字节为单位）。它用于在创建线程时进行错误检查。默认值是端口特定的，位于 tx_port.h 中。

TX_MISRA_ENABLE

定义后，ThreadX 将使用 MISRA C 兼容约定。有关详细信息，请参阅 ThreadX_MISRA_Compliance.pdf。

TX_MUTEX_ENABLE_PERFORMANCE_INFO

定义后，可以收集有关互斥体的性能信息。默认情况下，未定义此选项。

TX_NO_TIMER

定义后，ThreadX 计时器逻辑被完全禁用。这在无法使用 ThreadX 计时器功能（线程睡眠、API 超时、时间切片和应用程序计时器）的情况下十分有用。如果指定 TX_NO_TIMER，还必须定义选项 TX_TIMER_PROCESS_IN_ISR。

TX_NOT_INTERRUPTABLE

定义后，ThreadX 不会尝试最大程度地缩短中断锁定时间。这会提升执行速度，但会略微增加中断锁定时间。

TX_QUEUE_ENABLE_PERFORMANCE_INFO

定义后，可以收集有关队列的性能信息。默认情况下，未定义此选项。

TX_REACTIVATE_INLINE

定义后，执行 ThreadX 计时器内联的激活，而不是使用函数调用。这可以提升性能，但会略微增加代码大小。默

认情况下，未定义此选项。

TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO

定义后，可以收集有关信号灯的性能信息。默认情况下，未定义此选项。

TX_THREAD_ENABLE_PERFORMANCE_INFO

定义后，可以收集有关线程的性能信息。默认情况下，未定义此选项。

TX_TIMER_ENABLE_PERFORMANCE_INFO

定义后，可以收集有关计时器的性能信息。默认情况下，未定义此选项。

TX_TIMER_PROCESS_IN_ISR

定义后，将消除 ThreadX 的内部系统计时器线程。这会提升计时器事件和更小 RAM 要求的性能，因为不再需要计时器堆栈和控制块。但是，使用此选项会将所有计时器过期处理移动到计时器 ISR 级别。默认情况下，未定义此选项。

> **NOTE**
> 计时器允许的服务可能得不到 ISR 的允许，因此，在使用此选项时可能不允许这样做。

TX_TIMER_THREAD_PRIORITY

定义内部 ThreadX 系统计时器线程的优先级。默认值为优先级 0 - ThreadX 中的最高优先级。默认值定义在 tx_port.h中。

TX_TIMER_THREAD_STACK_SIZE

定义内部 ThreadX 系统计时器线程的堆栈大小（以字节为单位）。此线程处理所有线程睡眠请求以及所有服务调用超时。此外，从该上下文调用所有应用程序计时器回调例程。默认值是端口特定的，位于 tx_port.h 中。

ThreadX 版本 ID

程序员可以通过检查 tx_port.h 文件来获得 ThreadX 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 tx_version_id 来获取 ThreadX 版本。

第 3 章 - Azure RTOS ThreadX 的功能组件

2021/4/29 •

本章从功能角度介绍了高性能 Azure RTOS ThreadX 内核。每个功能组件都将采用易于理解的方式进行介绍。

执行概述

ThreadX 应用程序包含四种类型的程序执行：初始化、线程执行、中断服务例程 (ISR) 和应用程序计时器。

图 2 显示了各种不同类型的程序执行。本章的后续部分更详细地介绍了其中的每种类型。

初始化

顾名思义，这是 ThreadX 应用程序中的第一种程序执行。初始化包括处理器重置与线程计划循环入口点之间的所有程序执行。

线程执行

初始化完成后，ThreadX 会进入其线程计划循环。计划循环查找准备好执行的应用程序线程。找到准备就绪的线程后，ThreadX 将控制权转交给该线程。系统完成该线程（或另一个优先级较高的线程变为就绪）后，将执行权转交回线程计划循环，以查找下一个优先级最高的就绪线程。

这个持续执行和计划线程的过程是 ThreadX 应用程序中最常见的程序执行类型。

中断服务例程 (ISR)

中断是实时系统的基础。如果没有中断，则很难及时响应外部环境的变化。检测到中断时，处理器会保存当前程序执行的重要信息（通常在堆栈上），然后将控制权转交到预定义的程序区域。这个预定义的程序区域通常称为中断服务例程。在大多数情况下，中断在线程执行期间（或线程计划循环中）发生。但也可能在执行 ISR 或应用程序计时器时发生中断。

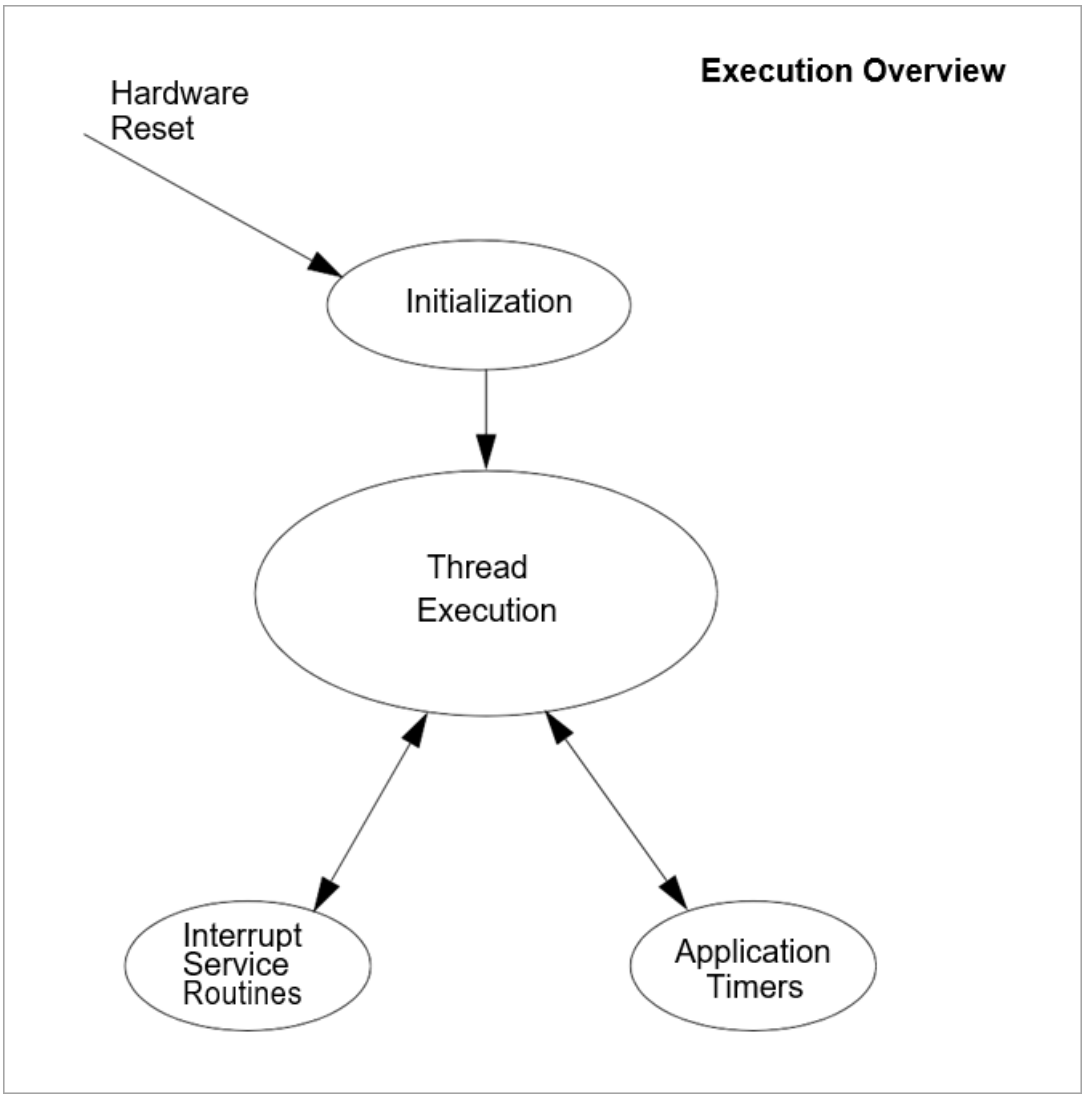


图 2. 程序执行类型

应用程序计时器

应用程序计时器与 ISR 类似，不同之处在于硬件实现（通常使用单个定期硬件中断）已对应用程序隐藏。应用程序使用此类计时器来执行超时、定期任务和/或监视器服务。与 ISR 一样，应用程序计时器最常中断线程执行。但与 ISR 不同的是，应用程序计时器无法相互中断。

内存用量

ThreadX 与应用程序一起驻留。因此，ThreadX 的静态内存（或固定内存）使用情况取决于开发工具，例如编译器、链接器和定位符。动态内存（或运行时内存）使用情况由应用程序直接控制。

静态内存使用情况

大多数开发工具将应用程序映像分为五个基本区域：指令、常数、已初始化的数据、未初始化的数据和系统堆栈。图 3 显示了这些内存区域的示例。

请务必了解，这只是一个示例。实际的静态内存布局特定于处理器、开发工具和基础硬件。

指令区域包含程序的所有处理器指令。此区域通常最大，一般位于 ROM 中。

常数区域包含各种已编译的常数，包括程序中定义或引用的字符串。此外，此区域包含已初始化的数据区域的“初始副本”。在内存使用情况编译器的初始化过程中，常数区域的这个部分用于设置 RAM 中已初始化的数据区域。常数区域通常在指令区域之后，一般位于 ROM 中。

已初始化的数据和未初始化的数据区域包含所有全局和静态变量。这些区域始终位于 RAM 中。

系统堆栈通常紧跟在初始化和未初始化的数据区域之后设置。

系统堆栈供编译器在初始化期间使用，然后供 ThreadX 在初始化期间使用，随后在 ISR 处理中使用。

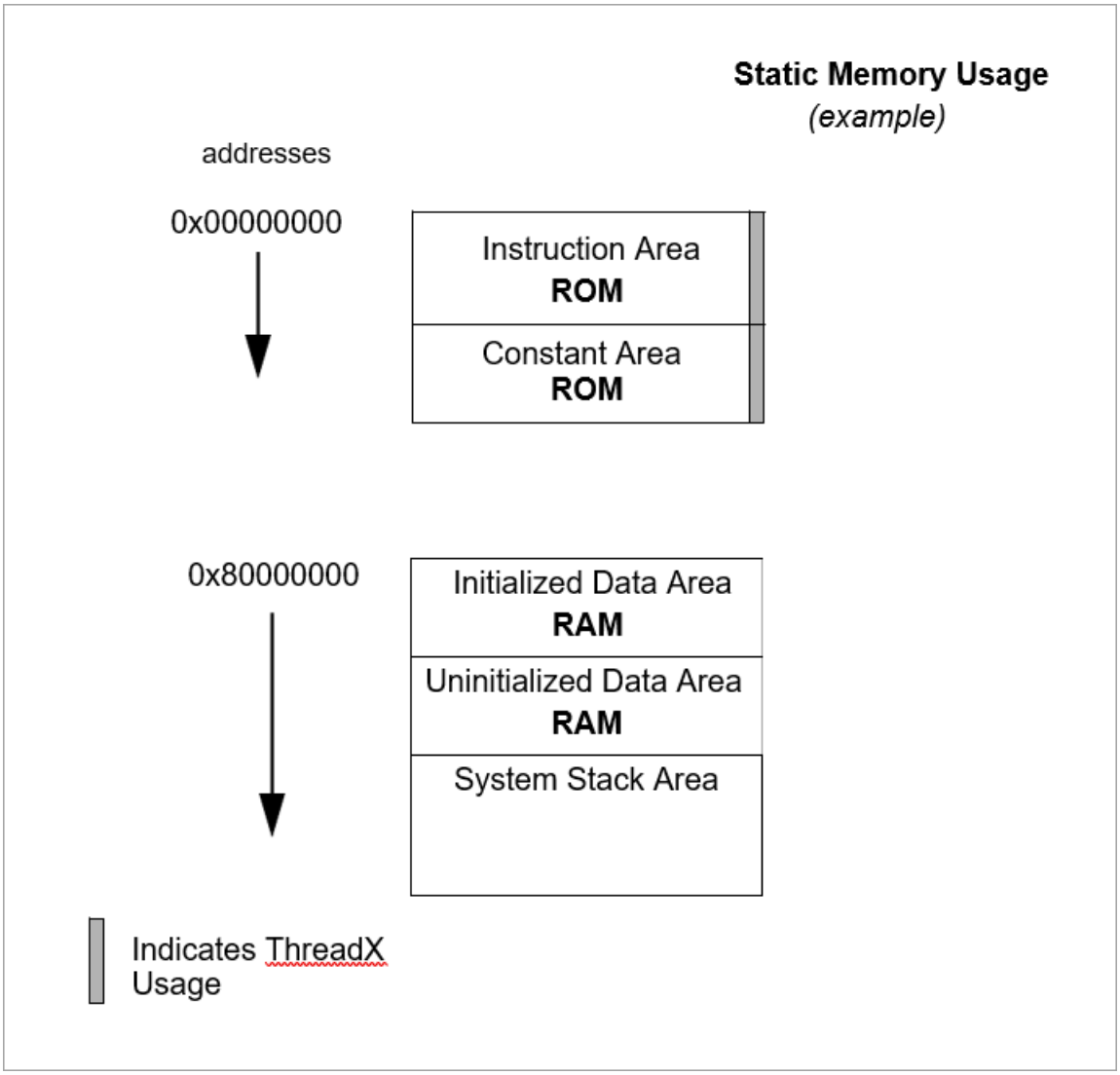


图 3. 内存区域示例

动态内存使用情况

如前所述，动态内存使用情况由应用程序直接控制。与堆栈、队列和内存池关联的控制块和内存区域可以放置在目标内存空间中的任何位置。这是一项重要的功能，因为该功能有助于轻松利用不同类型的物理内存。

例如，假设目标硬件环境同时具有快速内存和慢速内存。如果应用程序需要额外的性能来处理高优先级线程，则可将其控制块 (TX_THREAD) 和堆栈放置在快速内存区域中，这可能会显著提高其性能。

初始化

了解初始化过程非常重要。初始硬件环境在此处设置。此外，这里还为应用程序指定了初始个性化设置。

> **NOTE**
> ThreadX 尝试（尽可能）利用完整的开发工具初始化过程。这样，以后就可以更轻松地升级到开发工具的新版本。

系统重置向量

所有微处理器都具有重置逻辑。当（硬件或软件）发生重置时，将从特定内存位置检索应用程序入口点的地址。检索到入口点后，处理器会将控制权转交到该位置。应用程序入口点通常用本机程序集语言编写，并且通常由开发工具提供（至少采用模板形式）。在某些情况下，ThreadX 会附带入口程序的特殊版本。

开发工具初始化

低级初始化完成后，控制权将转交给开发工具的高级初始化。这个位置通常设置了已初始化的全局变量和静态 C 变量。请记住，其初始值将从常数区域检索。确切的初始化处理将特定于开发工具。

main 函数

开发工具初始化完成后，控制权将转交给用户提供的 main 函数。此时，应用程序会控制接下来执行的操作。对于大多数应用程序，main 函数只调用 tx_kernel_enter，这是 ThreadX 的入口。但是，应用程序可在进入 ThreadX 之前执行初步处理（通常用于硬件初始化）。

> **IMPORTANT**
> 对 tx_kernel_enter 的调用不会返回结果，因此请勿在此后执行任何处理。

tx_kernel_enter

entry 函数协调各种内部 ThreadX 数据结构的初始化，然后调用应用程序的定义函数 tx_application_define。

当 tx_application_define 返回时，控制权将转交给线程计划循环。这标志着初始化结束。

应用程序定义函数

tx_application_define 函数定义所有初始应用程序线程、队列、信号灯、互斥、事件标志、内存池和计时器。在应用程序的正常操作过程中，还可以在线程中创建和删除系统资源。但是，所有初始应用程序资源都在此处定义。

值得一提的是，tx_application_define 函数只有一个输入参数。第一个可用的 RAM 地址是该函数唯一的输入参数。该地址通常用作线程堆栈、队列和内存池的初始运行时内存分配起点。

> **NOTE**
> 初始化完成后，只有正在执行的线程才能创建和删除系统资源（包括其他线程）。因此，在初始化期间必须至少创建一个线程。

中断

在整个初始化过程中，中断处于禁用状态。如果应用程序以某种方式启用中断，则可能出现不可预知的行为。图 4 显示了从系统重置到特定于应用程序的初始化的整个初始化过程。

线程执行

计划和执行应用程序线程是 ThreadX 最重要的活动。线程通常定义为具有专用用途的半独立程序段。所有线程的组合处理构成了应用程序。

线程在初始化或线程执行期间通过调用 tx_thread_create 来动态创建。创建的线程处于“就绪”或“已挂起”状态。

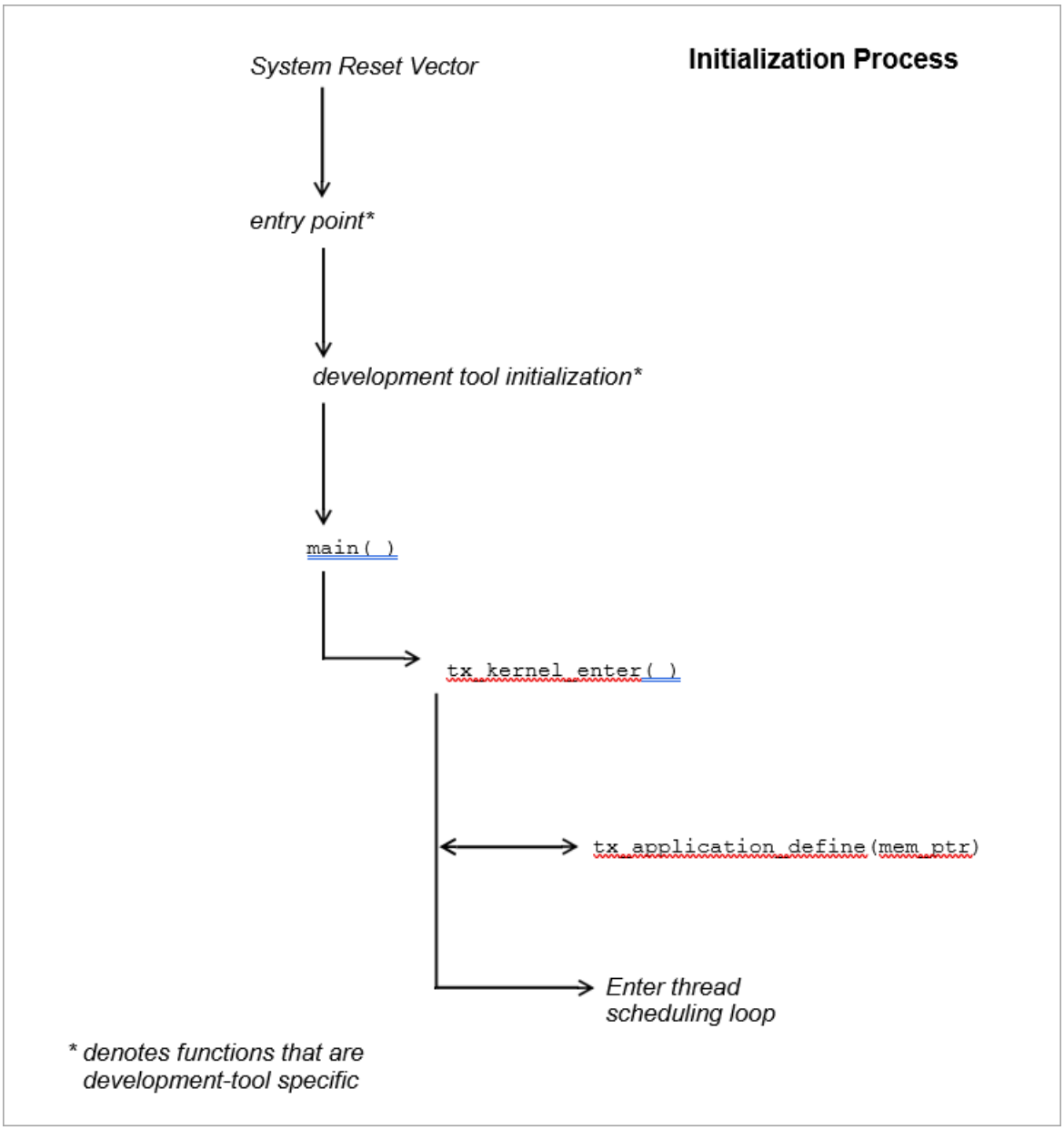


图 4. 初始化过程

线程执行状态

了解线程的不同处理状态是了解整个多线程环境的关键要素。ThreadX 有五个不同的线程状态：就绪、已挂起、正在执行、已终止和已完成 。图 5 显示了 ThreadX 的线程状态转换图。

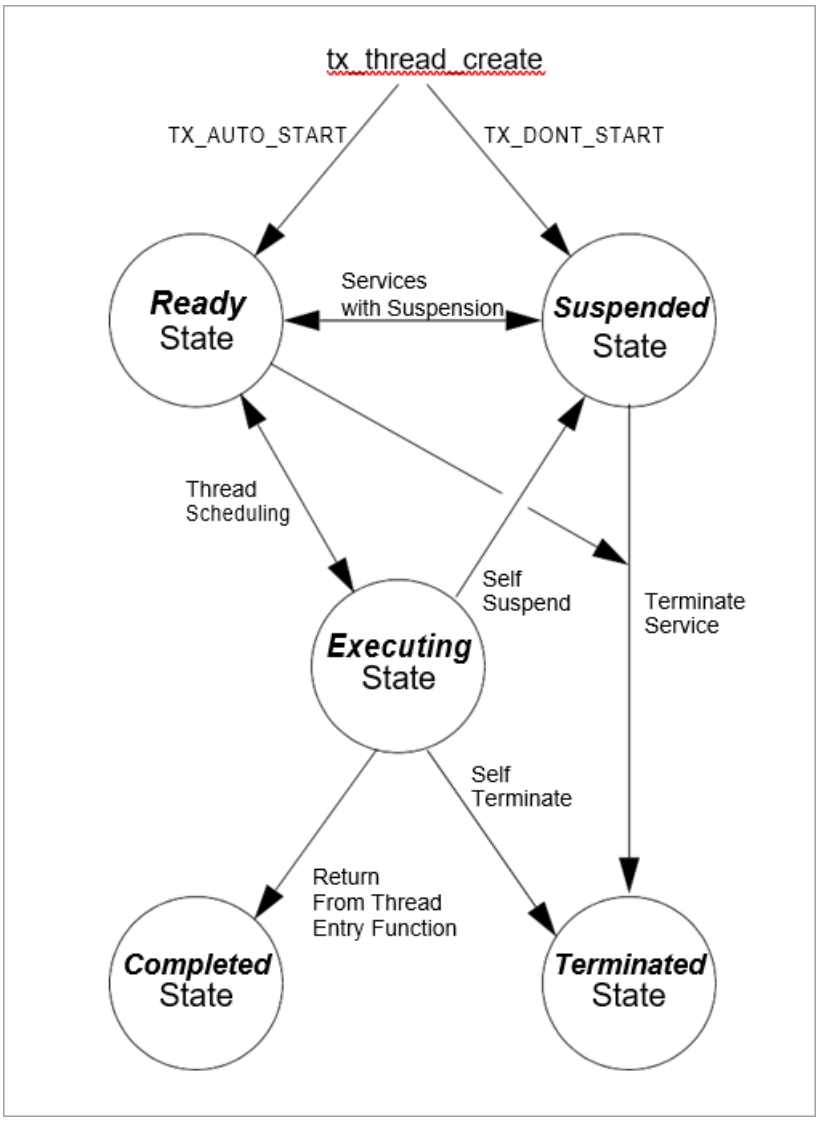


图 5. 线程状态转换

线程准备好执行时处于"就绪"状态。在处于就绪状态的线程中，只有优先级最高的就绪线程才会执行。发生这种情况时，ThreadX 会执行该线程，然后将其状态更改为"正在执行"。

如果更高优先级的线程已准备就绪，正在执行的线程会恢复为"就绪"状态。然后执行新的高优先级就绪线程，并将其逻辑状态更改为"正在执行"。每次发生线程抢占时，即会在"就绪"和"正在执行"之间转换 。

在任意指定时刻，只有一个线程处于"正在执行"状态。这是因为处于"正在执行"状态的线程可以控制基础处理器。

处于"已挂起"状态的线程不符合执行条件。导致处于"已挂起"状态的原因包括挂起时间、队列消息、信号灯、互斥、事件标志、内存和基本线程挂起。排除导致挂起的原因后，线程将恢复"就绪"状态。

处于"已完成"状态的线程是指已完成其处理任务并从其 entry 函数返回的线程。entry 函数在线程创建期间指定。处于"已完成"状态的线程无法再次执行。

线程处于"已终止"状态，因为另一个线程或该线程本身调用了 tx_thread_terminate 服务 。处于"已终止"状态的线程无法再次执行。

> **IMPORTANT**
>
> 如果需要重新启动已完成或已终止的线程，应用程序必须首先删除该线程。然后可以重新创建并重新启动该线程。

线程进入/退出通知

某些应用程序可能会发现，在特定线程首次进入、完成或终止时收到通知非常有利。ThreadX 通过

tx_thread_entry_exit_notify 服务提供此功能。此服务为特定线程注册应用程序通知函数，ThreadX 会在该线程开始运行、完成或终止时调用该函数。应用程序通知函数在调用后可以执行特定于应用程序的处理。这通常涉及通过 ThreadX 同步基元通知此事件的另一个应用程序线程。

线程优先级

如前所述，线程是具有专用用途的半独立程序段。但是，所有线程在创建时并非完全相同！ 某些线程具有比其他线程更重要的专用用途。这种异类的线程重要性是嵌入式实时应用程序的标志。

ThreadX 在创建线程时通过分配表示其"优先级"的数值来确定线程的重要性。ThreadX 的最大优先级数可在 32 到 1024（增量为 32）之间进行配置。实际的最大优先级数由 TX_MAX_PRIORITIES 常数在 ThreadX 库的编译过程中确定。设置更多优先级并不会显著增加处理开销。但是，每个包含 32 个优先级的组额外需要 128 字节的 RAM 进行管理。例如，32 个优先级需要 128 字节的 RAM，64 个优先级需要 256 字节的 RAM，而 96 个优先级需要 384 字节的 RAM。

默认情况下，ThreadX 具有 32 个优先级，范围从优先级 0 到优先级 31。数值越小，优先级越高。因此，优先级 0 表示最高优先级，而优先级 (TX_MAX_PRIORITIES-1) 表示最低优先级。

通过协作调度或时间切片，多个线程可以具有相同的优先级。此外，线程优先级还可以在运行时更改。

线程调度

ThreadX 根据线程的优先级来调度线程。优先级最高的就绪线程最先执行。如果有多个具有相同优先级的线程准备就绪，则按照先进先出 (FIFO) 的方式执行。

轮循调度

ThreadX 支持通过轮循调度处理具有相同优先级的多个线程。此过程通过以协作方式调用 tx_thread_relinquish 来实现。此服务为相同优先级的所有其他就绪线程提供了在 tx_thread_relinquish 调用方再次执行之前执行的机会。

时间切片

时间切片是轮循调度的另一种形式。时间片指定线程在不放弃处理器的情况下可以执行的最大计时器时钟周期数（计时器中断）。在 ThreadX 中，时间切片按每个线程提供。线程的时间片在创建时分配，可在运行时修改。当时间片过期时，具有相同优先级的所有其他就绪线程有机会在时间切片线程重新执行之前执行。

当线程挂起、放弃、执行导致抢占的 ThreadX 服务调用或自身经过时间切片后，该线程将获得一个新的线程时间片。

当时间切片的线程被抢占时，该线程将在其剩余的时间片内比具有相同优先级的其他就绪线程更早恢复执行。

> **NOTE**
>
> 使用时间切片会产生少量的系统开销。由于时间切片仅适用于多个线程具有相同优先级的情况，因此不应为具有唯一优先级的线程分配时间片。

优先

抢占是为了支持优先级更高的线程而暂时中断正在执行的线程的过程。此过程对正在执行的线程不可见。当更高优先级的线程完成时，控制权将转交回发生抢占的确切位置。这是实时系统中一项非常重要的功能，因为该功能有助于快速响应重要的应用程序事件。尽管抢占是一项非常重要的功能，但也可能导致各种问题，包括资源不足、开销过大和优先级反转。

抢占阈值 (Preemption Threshold™)

为了缓解抢占的一些固有问题，ThreadX 提供了一个独特的高级功能，名为抢占阈值。

抢占阈值允许线程指定禁用抢占的优先级上限。优先级高于上限的线程仍可以执行抢占，但不允许优先级低于上限的线程执行抢占。

例如，假设优先级为 20 的线程只与一组优先级介于 15 到 20 之间的线程进行交互。在其关键部分中，优先级为

20 的线程可将其抢占阈值设置为 15，从而防止该线程和与之交互的所有线程发生抢占。这仍允许（优先级介于 0和 14 之间）真正重要的线程在其关键部分处理中抢占此线程的资源，这会使处理的响应速度更快。

当然，仍有可能通过将其抢占阈值设置为 0 来为线程禁用所有抢占。此外，可以在运行时更改抢占阈值。

> **NOTE**
> 使用抢占阈值会禁用指定线程的时间切片。

优先级继承

ThreadX 还支持其互斥服务内的可选优先级继承，本章稍后将对此进行介绍。优先级继承允许低优先级线程暂时假设正在等待归较低优先级线程所有的互斥的高优先级线程的优先级。借助此功能，应用程序可以消除中间优先级线程的抢占，从而避免出现不确定的优先级反转。当然，也可以使用抢占阈值获得类似的结果。

线程创建

应用程序线程在初始化或执行其他应用程序线程的过程中创建。应用程序可以创建的线程数量没有限制。

线程控制块 TX_THREAD

每个线程的特征都包含在其控制块中。此结构在 tx_api.h 文件中定义。

线程的控制块可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

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。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，而不管另一个线程是否正将其用于控制块。

在大多数情况下，应用程序不知道线程控制块的内容。但在某些情况下，尤其是在调试过程中，观察特定成员会很有用。下面是一些更有用的控制块成员。

tx_thread_run_count 包含记录线程已调用次数的计数器。计数器增加表示正在调度和执行线程。

tx_thread_state 包含相关线程的状态。下面列出了可能的线程状态。

TX_READY	(0x00)
TX_COMPLETED	(0x01)
TX_TERMINATED	(0x02)
TX_SUSPENDED	(0x03)
TX_SLEEP	(0x04)
TX_QUEUE_SUSP	(0x05)
TX_SEMAPHORE_SUSP	(0x06)
TX_EVENT_FLAG	(0x07)
TX_BLOCK_MEMORY	(0x08)
TX_BYTE_MEMORY	(0x09)

TX_MUTEX_SUSP	(0x0D)

> **NOTE**
> 当然，线程控制块中还有很多有趣的字段，包括堆栈指针、时间片值、优先级等。欢迎用户查看控制块成员，但严格禁止修改！

> **IMPORTANT**
> 没有与本节前面提到的“正在执行”状态等同的状态。这不是必需的，因为在给定时间只有一个正在执行的线程。正在执行的线程的状态也是 TX_READY。

当前正在执行的线程

如前所述，在任何给定时间都只有一个正在执行的线程。有多种方法可以识别正在执行的线程，具体取决于发出请求的线程。通过调用 tx_thread_identify，程序段可以获取正在执行的线程的控制块地址。这在从多个线程执行的应用程序代码的共享部分中很有用。

在调试会话中，用户可以检查 ThreadX 内部指针 _tx_thread_current_ptr。该数组包含当前正在执行的线程的控制块地址。如果此指针为 NULL，则不执行任何应用程序线程；也就是说，ThreadX 在其调度循环中等待线程准备就绪。

线程堆栈区域

每个线程都必须有自己的堆栈，用于保存其上次执行和编译器使用的上下文。大多数 C 编译器使用堆栈来执行函数调用和临时分配局部变量。图 6 显示了典型的线程堆栈。

线程堆栈在内存中的位置取决于应用程序。堆栈区域在线程创建期间指定，可以位于目标地址空间的任意位置。这是一项重要的功能，因为应用程序可借助该功能，通过将重要线程堆栈置于高速 RAM 中来提高线程的性能。

堆栈内存区域（示例）

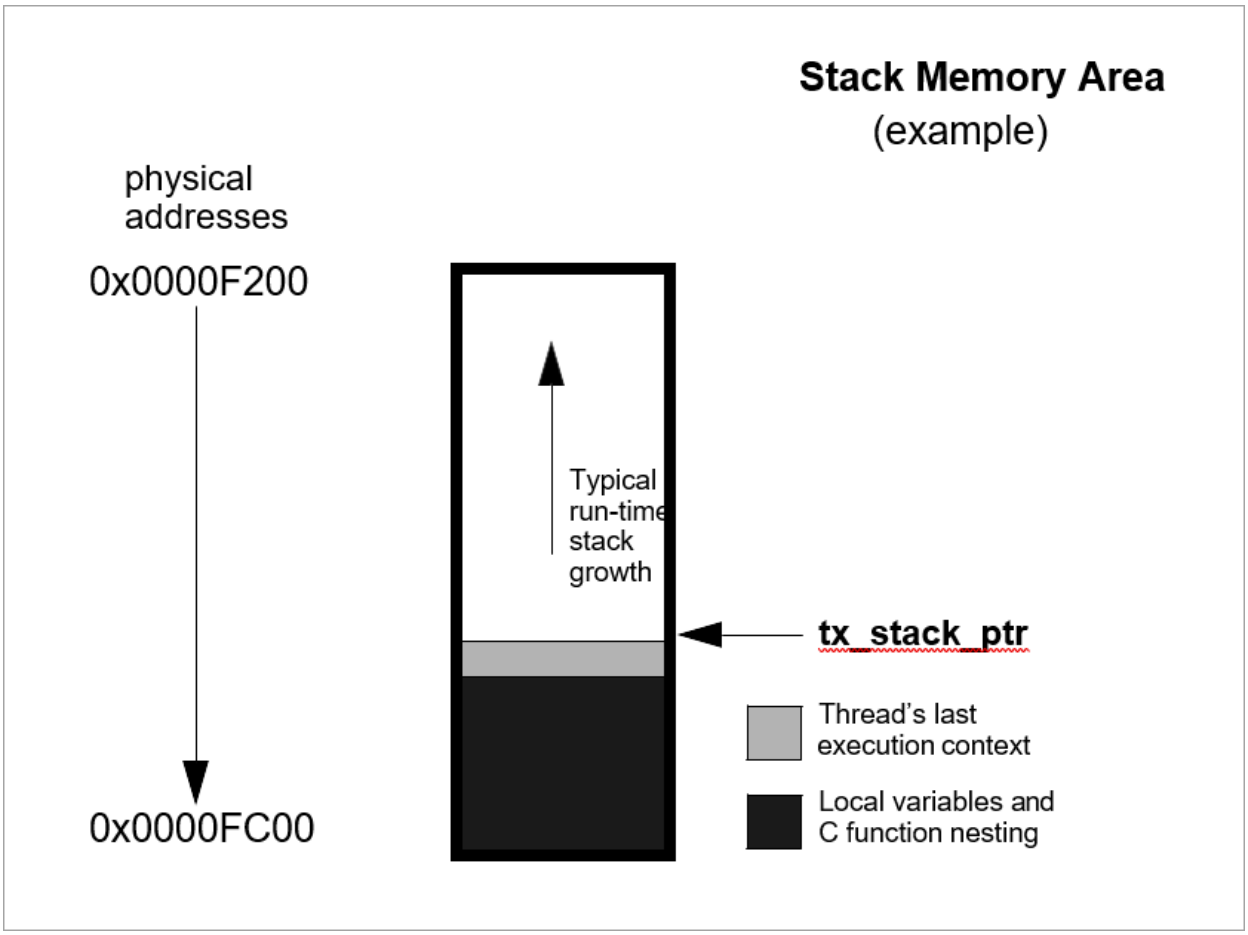


图 6. 典型的线程堆栈

应该设置多大的堆栈是有关线程的最常见问题之一。线程的堆栈区域必须大到足以容纳最坏情况下的函数调用嵌套、局部变量分配，并保存其最后一个执行上下文。

最小堆栈大小 TX_MINIMUM_STACK 由 ThreadX 定义。这种大小的堆栈支持保存线程的上下文、最少量的函数调用和局部变量分配。

但对大多数线程来说，最小的堆栈大小太小，用户必须通过检查函数调用嵌套和局部变量分配来确定最坏情况下的大小要求。当然，最好从较大的堆栈区域开始。

调试应用程序后，如果内存不足，可以调整线程堆栈大小。常用的技巧是在创建线程之前，使用诸如 (0xEFEF) 之类易于识别的数据模式预设所有堆栈区域。在应用程序经过全面测试后，可以对堆栈区域进行检查，具体方法是通过查找堆栈区域（其中的数据模式仍保持不变）来查看实际使用了多少堆栈。图 7 显示了堆栈在线程完全执行后预设为 0xEFEF。

堆栈内存区域（另一个示例）

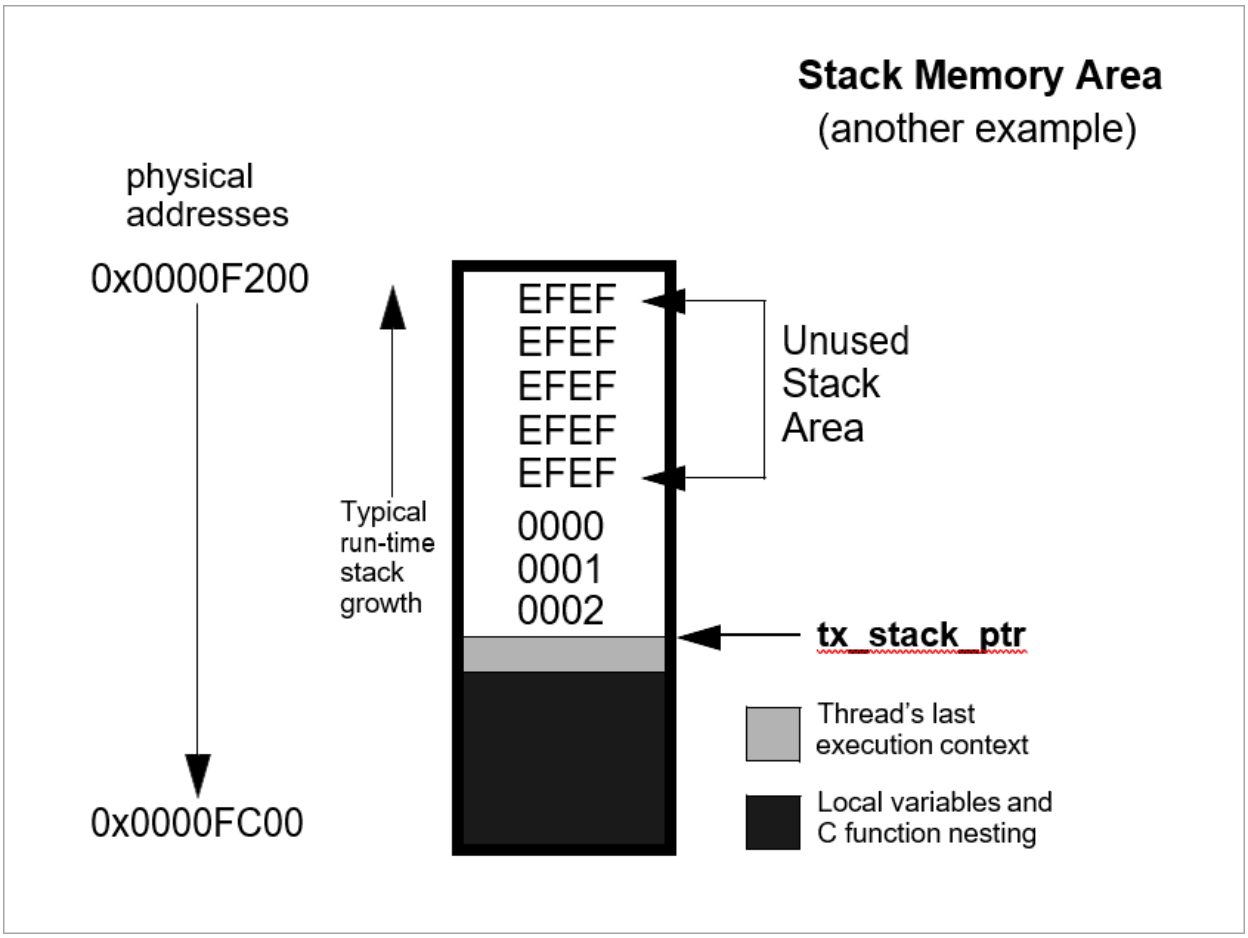


图 7. 堆栈预设为 0xEFEF

> **IMPORTANT**
> 默认情况下，ThreadX 使用值 0xEF 初始化每个线程堆栈的每个字节。

内存缺陷

线程的堆栈需求可能很大。因此，务必要将应用程序设计为可以容纳数量合理的线程。此外，必须注意避免在线程中过度使用堆栈。应避免使用递归算法和大型本地数据结构。

在大多数情况下，溢出的堆栈会导致线程执行损坏其堆栈区域相邻（通常在此位置之前）的内存。其结果不可预测，但大多会导致程序计数器出现反常的变化。这通常称为"深陷细枝末节"。当然，防止出现这种情况的唯一方法是确保所有线程堆栈足够大。

可选的运行时堆栈检查

ThreadX 提供了在运行时检查每个线程的堆栈是否损坏的功能。默认情况下，ThreadX 在创建过程中使用 0xEF 数据模式填充线程堆栈的每个字节。如果应用程序生成了已定义 TX_ENABLE_STACK_CHECKING 的 ThreadX 库，ThreadX 会检查每个线程的堆栈是否因为线程挂起或恢复而损坏。如果检测到堆栈损坏，ThreadX 将调用在调用 tx_thread_stack_error_notify 时指定的应用程序堆栈错误处理例程。否则，如果未指定堆栈错误处理程序，ThreadX 将调用内部 tx_thread_stack_error_handler 例程。

重新进入

多线程处理的真正优点之一是，可以从多个线程中调用相同的 C 函数。这提供了强大的功能，还有助于减少代码空间。但是，此功能要求从多个线程调用的 C 函数是可重入的函数。

基本上，可重入函数将调用方的返回地址存储在当前堆栈上，并且不依赖于先前设置的全局或静态 C 变量。大多数编译器将返回地址放在堆栈上。因此，应用程序开发人员必须只关心如何使用全局和静态变量 。

非重入函数的一个例子是在标准 C 库中找到的字符串标记函数 strtok。此函数在后续调用时"记住了"前面的字

符串指针。此函数通过静态字符串指针实现此功能。如果从多个线程调用此函数，则最有可能返回无效的指针。

线程优先级缺陷

选择线程优先级是多线程处理最重要的方面之一。有时很容易根据感知的线程重要性概念来分配优先级，而不是确定运行时到底需要什么。滥用线程优先级会导致其他线程资源枯竭、产生优先级反转、减少处理带宽，致使应用程序出现难以理解的运行时行为。

如前所述，ThreadX 提供基于优先级的抢占式调度算法。优先级较低的线程只能在没有更高优先级的线程可以执行时才会执行。如果优先级较高的线程始终准备就绪，则不会执行优先级较低的线程。这种情况称为线程资源不足。

大多数线程资源不足的问题都是在调试初期检测到的，可通过确保优先级较高的线程不会连续执行来解决。另外，还可以在应用程序中添加此逻辑，以便逐渐提高资源不足的线程的优先级，直到有机会执行这些线程。

与线程优先级相关的另一个缺陷是优先级反转。当优先级较高的线程由于优先级较低的线程占用其所需资源而挂起时，将会发生优先级反转。当然，在某些情况下，有必要让两个优先级不同的线程共享一个公用资源。如果这些线程是唯一处于活动状态的线程，优先级反转时间就与低优先级线程占用资源的时间息息相关。这种情况既具有确定性又非常正常。不过，如果在这种优先级反转的情况，中等优先级的线程变为活动状态，优先级反转时间就不再确定，并且可能导致应用程序失败。

主要有三种不同的方法可防止 ThreadX 中出现不确定的优先级反转。首先，在设计应用程序优先级选择和运行时行为时，可以采用能够防止出现优先级反转问题的方式。其次，优先级较低的线程可以利用抢占阈值来阻止中等优先级线程在其与优先级较高的线程共享资源时执行抢占。最后，使用 ThreadX 互斥对象保护系统资源的线程可以利用可选的互斥优先级继承来消除不确定的优先级反转。

优先级开销

要减少多线程处理开销，最容易被忽视的一种方法是减少上下文切换的次数。如前所述，当优先级较高的线程执行优先于正在执行的线程时，则会发生上下文切换。值得一提的是，在发生外部事件（例如中断）和执行线程发出服务调用时，优先级较高的线程可能变为就绪状态。

为了说明线程优先级对上下文切换开销的影响，假设有一个包含三个线程的环境，这些线程分别命名为 thread_1、thread_2 和 thread_3 。进一步假定所有线程都处于等待消息的挂起状态。当 thread_1 收到消息后，立即将其转发给 thread_2。随后，thread_2 将此消息转发给 thread_3。Thread_3 只是丢弃此消息。每个线程处理其消息后，则会返回并等待另一个消息。

执行这三个线程所需的处理存在很大的差异，具体取决于其优先级。如果所有线程都具有相同优先级，则会在执行每个线程之前发生一次上下文切换。当每个线程在空消息队列中挂起时，将会发生上下文切换。

但是，如果 thread_2 的优先级高于 thread_1，thread_3 的优先级高于 thread_2，上下文切换的次数将增加一倍。这是因为当其检测到优先级更高的线程现已准备就绪时，会在 tx_queue_send 服务中执行另一次上下文切换。

ThreadX 抢占阈值机制可以避免出现这些额外的上下文切换，并且仍支持前面提到的优先级选择。这是一项重要的功能，因为该功能允许在调度期间使用多个线程优先级，同时避免在线程执行期间出现一些不需要的上下文切换。

运行时线程性能信息

ThreadX 提供可选的运行时线程性能信息。如果 ThreadX 库和应用程序是在定义 TX_THREAD_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX 会累积以下信息。

整个系统的总数：

- 线程恢复数
- 线程挂起数
- 服务调用抢占次数
- 中断抢占次数

- 优先级反转数
- 时间片数
- 放弃次数
- 线程超时数
- 挂起中止数
- 空闲系统返回数
- 非空闲系统返回数

每个线程的总数：

- 恢复数
- 挂起数
- 服务调用抢占次数
- 中断抢占次数
- 优先级反转数
- 时间片数
- 线程放弃数
- 线程超时数
- 挂起中止数

此信息在运行时通过 tx_thread_performance_info_get 和 tx_thread_performance_system_info_get 服务提供。线程性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。例如，如果服务调用抢占数量相对较多，可能表明线程的优先级和/或抢占阈值过低。此外，如果空闲系统返回次数相对较少，可能表明优先级较低的线程没有完全挂起。

调试缺陷

调试多线程应用程序更为困难，因为同一个程序代码可以通过多个线程执行。在这种情况下，只使用断点可能不够。调试器还必须使用条件断点观察当前线程指针 _tx_thread_current_ptr，以查看调用线程是否为要调试的线程。

其中很多工作都是使用各种开发工具供应商提供的多线程支持包进行处理。因为这些支持并设计简单，因此将 ThreadX 与不同的开发工具集成相对容易。

堆栈大小始终是多线程处理的重要调试主题。如果观察到无法解释的行为，通常最好是增加所有线程的堆栈大小，尤其是要执行的最后一个线程的堆栈大小！

> **TIP**
> 使用定义的 TX_ENABLE_STACK_CHECKING 生成 ThreadX 库也是个好办法。这将有助于在处理过程中尽早排除堆栈损坏问题。

消息队列

消息队列是 ThreadX 中线程间通信的主要方式。消息队列中可以驻留一个或多个消息。保留单个消息的消息队列通常称为邮箱。

消息通过 tx_queue_send 复制到队列，然后通过 tx_queue_receive 从队列中复制。唯一的例外是在等待空队列中

的消息时线程会挂起。在这种情况下，发送到队列的下一条消息将直接放入该线程的目标区域。

每个消息队列都是一个公用资源。ThreadX 对如何使用消息队列没有任何限制。

创建消息队列

消息队列由应用程序线程在初始化期间或运行时创建。应用程序中的消息队列数没有限制。

消息大小

每个消息队列都支持许多固定大小的消息。可用的消息大小为 1 到 16 个 32 位的字（含）。消息大小在创建队列时指定。超过 16 个字的应用程序消息必须通过指针传递。为此，可以创建消息大小为 1 个字的队列（足以容纳一个指针），然后发送和接收消息指针，而不是整个消息。

消息队列容量

队列可以容纳的消息数是消息大小与创建期间提供的内存区域大小的函数。队列的总消息容量的计算方法是，将每条消息中的字节数除以所提供的内存区域的总字节数。

例如，如果支持消息大小为 1 个 32 位字（4 个字节）的消息队列是使用 100 字节内存区域创建的，则其容量为 25 条消息。

队列内存区域

如前所述，用于缓冲消息的内存区域在队列创建期间指定。与 ThreadX 中的其他内存区域一样，该区域可以位于目标地址空间的任何位置。

这是一项重要的功能，因为它为应用程序提供了相当大的灵活性。例如，应用程序可能会在高速 RAM 中定位重要队列的内存区域，从而提高性能。

线程挂起

尝试从队列发送或接收消息时，应用程序线程可能会挂起。线程挂起通常涉及等待来自空队列的消息。但是，线程也可能在尝试向已满队列发送消息时挂起。

解决挂起的条件后，请求的服务完成，等待的线程也相应恢复。如果同一队列中的多个线程挂起，这些线程将按照挂起的顺序 (FIFO) 恢复。

不过，如果应用程序在取消线程挂起的队列服务之前调用 tx_queue_prioritize，还可以恢复优先级。队列设置优先级服务将优先级最高的线程置于挂起列表的前面，让所有其他挂起的线程采用相同的 FIFO 顺序。

超时也可用于所有队列挂起。从根本上说，超时会指定线程保持挂起状态的最大计时器时钟周期数。如果发生超时，则会恢复线程，该服务会返回相应的错误代码。

队列发送通知

某些应用程序可能会发现，在将消息放入队列时收到通知十分有利。ThreadX 通过 tx_queue_send_notify 服务提供此功能。此服务将提供的应用程序通知函数注册到指定的队列。只要有消息发送到队列，ThreadX 就会调用此应用程序通知函数。应用程序通知函数内的确切处理由应用程序决定；但这通常包括恢复相应的线程以处理新消息。

队列事件链接™

ThreadX 中的通知功能可用于链接各种同步事件。当单个线程必须处理多个同步事件时，这通常很有用。

例如，假设单个线程负责处理来自五个不同队列的消息，还必须在没有可用消息时挂起。通过为每个队列注册应用程序通知函数，并引入额外的计数信号灯，即可轻松实现这一点。具体而言，每次调用应用程序通知函数时，该函数就会执行 tx_semaphore_put（信号灯计数表示所有五个队列中的消息总数）。处理线程通过 tx_semaphore_get 服务挂起此信号灯。当信号灯可用时（在这种情况下是消息可用时），即会恢复处理线程。随后，它会询问每个队列来获取一条消息，处理找到的消息，然后执行另一个 tx_semaphore_get，以等待下一条消息。在不使用事件链的情况下实现这一点非常困难，可能需要更多线程和/或附加的应用程序代码。

通常情况下，事件链会减少线程、降低开销和减少 RAM 要求。它还提供了一种非常灵活的机制来处理更复杂系统的同步要求。

运行时队列性能信息

ThreadX 提供可选的运行时队列性能信息。如果 ThreadX 库和应用程序是在定义 TX_QUEUE_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX 会累积以下信息。

整个系统的总数：

- 发送的消息数
- 收到的消息数
- 队列为空的挂起数
- 队列已满的挂起数
- 队列已满返回的错误数（未指定挂起）
- 队列超时数

每个队列的总数：

- 发送的消息数
- 收到的消息数
- 队列为空的挂起数
- 队列已满的挂起数
- 队列已满返回的错误数（未指定挂起）
- 队列超时数

此信息在运行时通过 tx_queue_performance_info_get 和 tx_queue_performance_system_info_get 服务提供。队列性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。例如，如果"队列已满的挂起数"相对较多，则表明增加队列大小可能有好处。

队列控制块 TX_QUEUE

每个消息队列的特征都可在其控制块中找到。控制块包含受关注的信息，例如队列中的消息数。此结构在 tx_api.h 文件中定义。

消息队列控制块也可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

消息目标缺陷

如前所述，消息将在队列区域和应用程序数据区域之间复制。请务必确保收到消息的目标大到足以容纳整个消息。否则，可能会损坏消息目标后面的内存。

> **NOTE**
> 如果堆栈上的消息目标太小，就特别致命，没有什么比损坏函数的返回地址更重要！

统计信号量

ThreadX 提供 32 位计数信号灯，其值范围在 0 到 4,294,967,295 之间。计数信号灯有两个操作：tx_semaphore_get 和 tx_semaphore_put 。执行获取操作会将信号灯数量减一。如果信号灯为 0，获取操作不会成功。获取操作的逆操作是放置操作。该操作会将信号灯数量加一。

每个计数信号灯都是一个公用资源。ThreadX 对如何使用计数信号灯没有任何限制。

计数信号灯通常用于互相排斥。不过，也可将计数信号灯用作事件通知的方法。

互相排斥

互相排斥用于控制线程对某些应用程序区域（也称为关键部分或应用程序资源）的访问 。将信号灯用于互相排斥时，信号灯的"当前计数"表示允许访问的线程总数。在大多数情况下，用于互相排斥的计数信号灯的初始值为 1，这意味着每次只有一个线程可以访问关联的资源。只有 0 或 1 值的计数信号灯通常称为二进制信号灯。

> **IMPORTANT**
> 如果使用二进制信号灯，用户必须阻止同一个线程对其已拥有的信号灯执行获取操作。第二个获取操作将失败，并且可能导致调用线程无限期挂起和资源永久不可用。

事件通知

还可以采用生成者-使用者的方式，将计数信号灯用作事件通知。使用者尝试获取计数信号灯，而生成者则在有可用的信息时增加信号灯。此类信号灯的初始值通常为 0，此值不会在生成者为使用者准备好信息之前增加。用于事件通知的信号灯也可能从使用 tx_semaphore_ceiling_put 服务调用中获益。此服务确保信号灯计数值永远不会超过调用中提供的值。

创建计数信号灯

计数信号灯由应用程序线程在初始化期间或运行时创建。信号灯的初始计数在创建过程中指定。应用程序中计数信号灯的数量没有限制。

线程挂起

尝试对当前计数为 0 的信号灯执行获取操作时，应用程序线程可能会挂起。

执行放置操作后，才会执行挂起线程的获取操作并恢复该线程。如果同一计数信号灯上挂起多个线程，这些线程将按照挂起的顺序 (FIFO) 恢复。

不过，如果应用程序在取消线程挂起的信号灯放置调用之前调用 tx_semaphore_prioritize，还可以恢复优先级。信号灯设置优先级服务将优先级最高的线程放于挂起列表的前面，同时让所有其他挂起的线程采用相同的 FIFO 顺序。

信号灯放置通知

某些应用程序可能会发现，在放置信号灯时收到通知十分有利。ThreadX 通过 tx_semaphore_put_notify 服务提供此功能。此服务将提供的应用程序通知函数注册到指定的信号灯。只要放置了信号灯，ThreadX 就会调用此应用程序通知函数。应用程序通知函数内的确切处理由应用程序决定；但这通常包括恢复相应的线程以处理新信号灯放置事件。

信号灯事件链接™

ThreadX 中的通知功能可用于链接各种同步事件。当单个线程必须处理多个同步事件时，这通常很有用。

例如，应用程序可以为每个对象注册一个通知例程，而不是为队列消息、事件标志和信号灯而挂起单独的线程。在调用后，应用程序通知例程会恢复单个线程，该线程可以询问每个对象以查找并处理新事件。

通常情况下，事件链会减少线程、降低开销和减少 RAM 要求。它还提供了一种非常灵活的机制来处理更复杂系统的同步要求。

运行时信号灯性能信息

ThreadX 提供可选的运行时信号灯性能信息。如果 ThreadX 库和应用程序是在定义 TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX 会累积以下信息。

整个系统的总数：

- 信号灯放置数
- 信号灯获取数
- 信号灯获取挂起数

- 信号灯获取超时数

每个信号灯的总数：

- 信号灯放置数
- 信号灯获取数
- 信号灯获取挂起数
- 信号灯获取超时数

此信息在运行时通过 tx_semaphore_performance_info_get 和 tx_semaphore_performance_system_info_get 服务提供。信号灯性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。例如，“信号灯获取超时数”相对较高可能表明其他线程占用资源的时间太长。

信号灯控制块 TX_SEMAPHORE

每个计数信号灯的特征都可在其控制块中找到。该控制块包含诸如当前的信号灯计数等信息。此结构在 tx_api.h 文件中定义。

信号灯控制块可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

抱死

与用于互相排斥的信号灯相关的最有趣且最危险的缺陷之一是抱死。抱死或死锁是指两个或多个线程在尝试获取归对方所有的信号灯时无限期挂起的情况。

这种情况最好用两个线程、两个信号灯的示例来说明。假设第一个线程拥有第一个信号灯，第二个线程拥有第二个信号灯。如果第一个线程尝试获取第二个信号灯，同时第二个线程尝试获取第一个信号灯，这两个线程就会进入死锁状态。此外，如果这些线程永远保持挂起状态，与之关联的资源也会永久锁定。图 8 说明了这一示例。

抱死（示例）

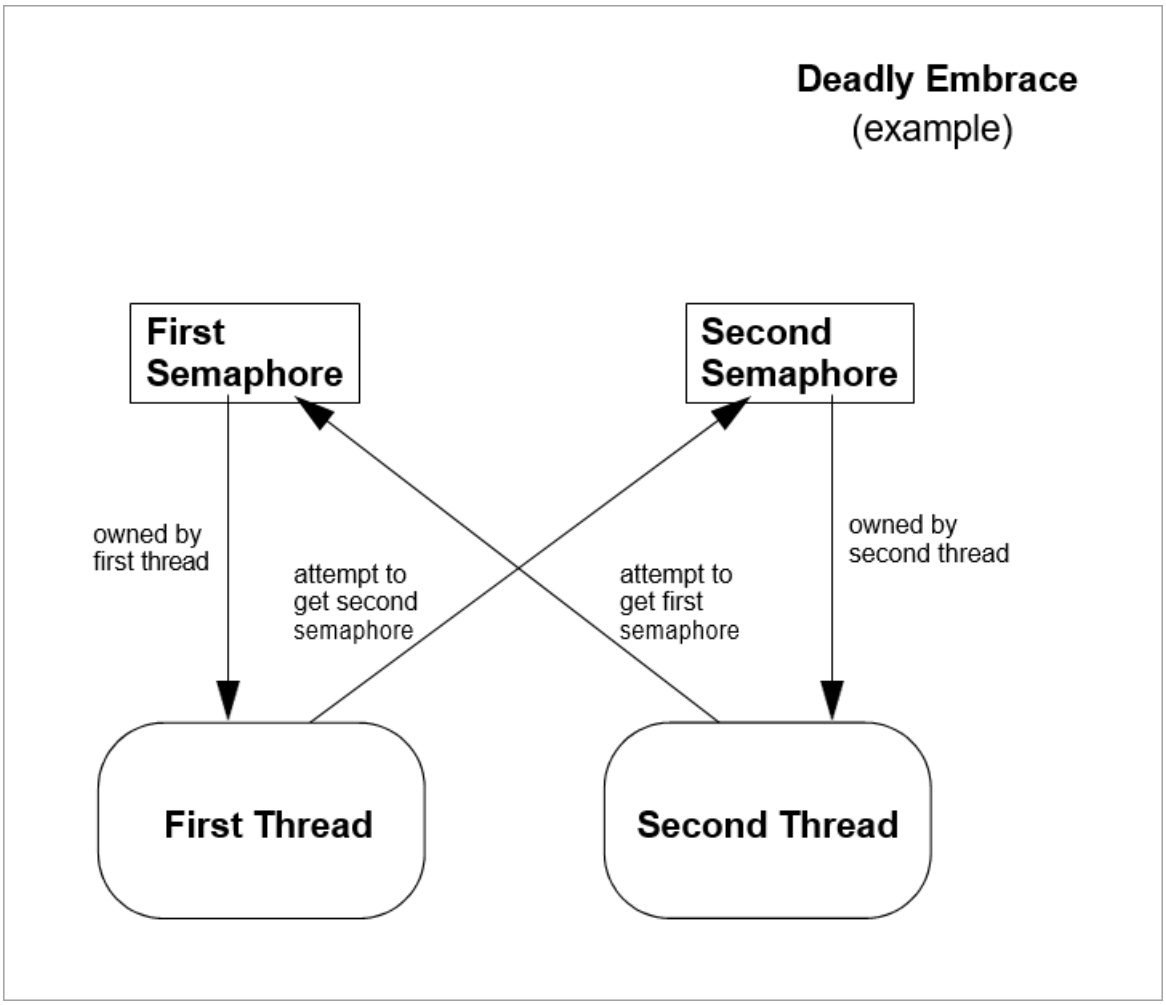


图 8. 挂起线程的示例

对于实时系统，可以通过对线程获取信号灯的方式设置一些限制来防止抱死。线程每次只能拥有一个信号灯。或者，如果线程按照相同的顺序收集多个信号灯，则可以拥有这些信号灯。在前面的示例中，如果第一个和第二个线程按顺序获取第一个和第二个信号灯，则可防止抱死。

> **TIP**
> 也可以使用与获取操作关联的挂起超时从抱死状态中恢复。

优先级反转

与互相排斥信号灯相关的另一个缺陷是优先级反转。"线程优先级缺陷"更全面地讨论了本主题。

根本问题源自这种情况，即低优先级线程拥有较高优先级线程所需的信号灯。这本身很正常。但是，它们之间具有优先级的线程可能会导致优先级反转持续不确定的时间。这种情况可通过以下方式处理：谨慎选择线程优先级，使用抢占阈值，并将拥有该资源的线程的优先级暂时提升到高优先级线程的级别。

Mutexes

除了信号灯，ThreadX 还提供互斥对象。互斥实质上是二进制信号灯，这意味着每次只有一个线程可以拥有一个互斥。此外，同一线程可能会对已拥有的互斥多次执行成功的互斥获取操作，准确来说是4,294,967,295 次。互斥对象有两个操作：tx_mutex_get 和 tx_mutex_put。获取操作获得不归另一个线程所有的互斥，而放置操作释放以前获得的互斥。对于要释放互斥的线程，放置操作数必须等于先前的获取操作数。

每个互斥都是一个公用资源。ThreadX 对如何使用互斥没有任何限制。

ThreadX 互斥仅用于互相排斥。与计数信号灯不同，互斥不能作为事件通知的方法。

互斥体互相排斥

与"计数信号灯"部分的讨论类似，互相排斥用于控制线程对特定应用程序区域（也称为关键部分或应用程序资源）的访问 。如果可用，ThreadX 互斥的所有权计数就为 0。线程获得互斥后，在互斥时每成功执行一次 Get 操作，所有权计数就会递增一次，每成功执行一次 Put 操作则会递减一次。

创建互斥

ThreadX 互斥由应用程序线程在初始化期间或运行时创建。互斥的初始条件始终是"可用"。还可以在选择优先级继承的情况下创建互斥。

线程挂起

当尝试对已归另一个线程所有的互斥执行获取操作时，应用程序线程可能会挂起。

当拥有线程执行了相同数量的获取操作后，将执行挂起线程的获取操作，为其提供互斥所有权，并恢复线程。如果多个线程在同一互斥上挂起，这些线程将按照挂起的相同顺序 (FIFO) 恢复。

但是，如果在创建期间选择了互斥优先级继承，则会自动执行优先级恢复。如果应用程序在取消线程挂起的互斥 put 调用之前调用 tx_mutex_prioritize，还可以恢复优先级。互斥设置优先级服务将优先级最高的线程置于挂起列表的前面，让所有其他挂起的线程采用相同的 FIFO 顺序。

运行时互斥性能信息

ThreadX 提供可选的运行时互斥性能信息。如果 ThreadX 库和应用程序是在定义 TX_MUTEX_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX 会累积以下信息。

整个系统的总数：

- 互斥放置数
- 互斥获取数
- 互斥获取挂起数
- 互斥获取超时数
- 互斥优先级反转数
- 互斥优先级继承数

每个互斥的总数：

- 互斥放置数
- 互斥获取数
- 互斥获取挂起数
- 互斥获取超时数
- 互斥优先级反转数
- 互斥优先级继承数

此信息在运行时通过 tx_mutex_performance_info_get 和 tx_mutex_performance_system_info_get 服务提供。互斥性能信息可用于确定应用程序是否正常运行。此信息对于优化应用程序也很有用。例如，"互斥获取超时数"相对较高可能表明其他线程占用资源的时间太长。

互斥控制块 TX_MUTEX

每个互斥的特征都可在其控制块中找到。该控制块包含诸如当前互斥所有权计数以及拥有互斥的线程的指针等信息。此结构在 tx_api.h 文件中定义。互斥控制块可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

抱死

与互斥所有权相关的最有趣且最危险的缺陷之一是抱死。抱死或死锁是指两个或多个线程在尝试获取归其他线程所有的互斥时无限期挂起的情况。有关抱死及其补救措施的讨论也完全适用于互斥对象。

优先级反转

如前所述，与互相排斥相关的主要缺陷是优先级反转。"线程优先级缺陷"更全面地讨论了本主题。

根本问题源自这种情况，即低优先级线程拥有较高优先级线程所需的信号灯。这本身很正常。但是，它们之间具有优先级的线程可能会导致优先级反转持续不确定的时间。与前面讨论的信号灯不同，ThreadX 互斥对象具有可选的优先级继承。优先级继承的基本思路是，优先级较低的线程将其优先级暂时提升为需要归低优先级线程所有的相同互斥的高优先级线程的优先级。当优先级较低的线程释放互斥时，即会恢复为其原始优先级，并为优先级较高的线程提供互斥所有权。此功能可将反转量限制为拥有互斥的较低优先级线程的时间，以此来消除不确定的优先级反转。当然，本章前面讨论的用于处理不确定的优先级反转的技巧也适用于互斥。

事件标志

事件标志为线程同步提供了强大的工具。每个事件标志由一个位表示。事件标志按照 32 个一组的形式排列。线程可以同时对组中的所有 32 个事件标记执行操作。事件由 tx_event_flags_set 设置，由 tx_event_flags_get 检索。

可通过在当前事件标志和新事件标志之间执行逻辑 AND/OR 运算来设置事件标志。逻辑运算的类型（AND 或 OR）在 tx_event_flags_set 调用中指定。

可使用类似的逻辑选项来检索事件标志。获取请求可以指定需要所有指定的事件标志（一个逻辑 AND）。

获取请求还可以指定任何指定的事件标志都满足该请求（一个逻辑 OR）。与事件标志检索相关的逻辑运算类型在 tx_event_flags_get 调用中指定。

> **IMPORTANT**
> 如果 TX_OR_CLEAR 或 TX_AND_CLEAR 由该请求指定，则使用满足获取请求的事件标志（例如，设置为零）。

每个事件标志组都是一个公用资源。ThreadX 对如何使用事件标志组没有任何限制。

创建事件标志组

事件标志组由应用程序线程在初始化期间或运行时创建。创建事件标志组时，组中的所有事件标志均设置为零。应用程序中事件标志组的数量没有限制。

线程挂起

尝试从组中获取事件标志的任意逻辑组合时，应用程序线程可能会挂起。设置事件标志后，将检查所有挂起线程的获取请求。所有现已包含所需事件标志的线程都会恢复。

> **NOTE**
> 设置事件标志组的事件标志时，将检查该事件标志组中所有已挂起的线程。当然，这会产生额外的开销。因此，最好将使用同一事件标志组的线程数限制为合理的数量。

事件标志设置通知

某些应用程序可能会发现，在设置事件标志时收到通知十分有利。ThreadX 通过 tx_event_flags_set_notify 服务提供此功能。此服务提供的应用程序通知函数注册到指定的事件标志组。只要在组中设置了事件标志，ThreadX 就会调用此应用程序通知函数。应用程序通知函数内的确切处理由应用程序决定，但这通常包括恢复相应的线程以处理新事件标志。

事件标志事件链接 (Event chaining™)

ThreadX 中的通知功能可用于"链接"各种同步事件。当单个线程必须处理多个同步事件时，这通常很有用。

例如，应用程序可以为每个对象注册一个通知例程，而不是为队列消息、事件标志和信号灯而挂起单独的线程。在调用后，应用程序通知例程会恢复单个线程，该线程可以询问每个对象以查找并处理新事件。

通常情况下，事件链会减少线程、降低开销和减少 RAM 要求。它还提供了一种非常灵活的机制来处理更复杂系统的同步要求。

运行时事件标志性能信息

ThreadX 提供可选的运行时事件标志性能信息。如果 ThreadX 库和应用程序是在定义 TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX 会累积以下信息。

整个系统的总数：

- 事件标志集数
- 事件标志获取数
- 事件标志获取挂起数
- 事件标志获取超时数

每个事件标志组的总数：

- 事件标志集数
- 事件标志获取数
- 事件标志获取挂起数
- 事件标志获取超时数

此信息在运行时通过 tx_event_flags_performance_info_get 和 tx_event_flags_performance_system_info_get 服务提供。事件标志的性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。例如，tx_event_flags_get 服务中的超时次数相对较多，可能表明事件标志挂起超时时间太短。

事件标志组控制块 TX_EVENT_FLAGS_GROUP

每个事件标志组的特征都可在其控制块中找到。该控制块包含诸如当前事件标志设置和事件中挂起的线程数等信息。此结构在 tx_api.h 文件中定义。

事件组控制块可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

内存块池

在实时应用程序中，采用快速且确定的方式分配内存始终是一项挑战。考虑到这一点，ThreadX 提供了创建和管理多个固定大小的内存块池的功能。

由于内存块池由固定大小的块组成，因此永远不会出现任何碎片问题。当然，碎片会导致出现本质上不确定的行为。此外，分配和释放固定大小内存块所需的时间与简单的链接列表操作所需的时间相当。另外，还可以在可用列表的开头完成内存块分配和取消分配。这可以提供最快的链接列表处理速度，并且有助于将实际的内存块保存在缓存中。

缺乏灵活性是固定大小内存池的主要缺点。池的块大小必须足够大，才能处理其用户最坏情况下的内存需求。当然，如果对同一个池发出了许多大小不同的内存请求，则可能会浪费内存。一种可能的解决方案是创建多个不同的内存块池，这些池包含不同大小的内存块。

每个内存块池都是一个公用资源。ThreadX 对如何使用池没有任何限制。

创建内存块池

内存块池由应用程序线程在初始化期间或运行时创建。应用程序中内存块池的数量没有限制。

内存块大小

如前所述，内存块池包含许多固定大小的块。块大小（以字节为单位）在创建池时指定。

> **NOTE**
>
> ThreadX 为池中的每个内存块增加了少量开销（C 指针的大小）。此外，ThreadX 可能需要填充块大小，从而确保每个内存块的开头能够正确对齐。

池容量

池中的内存块数是在创建过程中提供的内存区域的块大小和总字节数的函数。池容量的计算方法是将块大小（包括填充和指针开销字节）除以提供的内存区域的总字节数。

池的内存区域

如前所述，块池的内存区域在创建时指定。与 ThreadX 中的其他内存区域一样，该区域可以位于目标地址空间的任何位置。

这是一项重要的功能，因为它提供了相当大的灵活性。例如，假设某个通信产品有一个用于 I/O 的高速内存区域。将此内存区域设置为 ThreadX 内存块池，即可轻松对其进行管理。

线程挂起

在等待空池中的内存块时，应用程序线程可能会挂起。当块返回到池时，将为挂起的线程提供此块，并恢复线程。

如果同一内存块池中挂起多个线程，这些线程将按挂起的顺序 (FIFO) 恢复。

不过，如果应用程序在取消线程挂起的块释放调用之前调用 tx_block_pool_prioritize，还可以恢复优先级。块池设置优先级服务将优先级最高的线程置于挂起列表的前面，让所有其他挂起的线程采用相同的 FIFO 顺序。

运行时块池性能信息

ThreadX 提供可选的运行时块池性能信息。如果 ThreadX 库和应用程序是在定义 TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX 会累积以下信息。

整个系统的总数：

- 已分配的块数
- 已释放的块数
- 分配挂起数
- 分配超时数

每个块池的总数：

- 已分配的块数
- 已释放的块数
- 分配挂起数
- 分配超时数

此信息在运行时通过 tx_block_pool_performance_info_get 和 tx_block_pool_performance_system_info_get 服务提供。块池性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。例如，“分配挂起数”相对较高可能表明块池太小。

内存块池控制块 TX_BLOCK_POOL

每个内存块池的特征都可在其控制块中找到。该控制块包含诸如可用的内存块数和内存池块大小等信息。此结构在 tx_api.h 文件中定义。

池控制块也可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

覆盖内存块

务必确保已分配内存块的用户不会在其边界之外写入。如果发生这种情况，则会损坏其相邻的内存区域（通常是后续区域）。结果不可预测，且对于应用程序来说通常很严重。

内存字节池

ThreadX 内存字节池与标准 C 堆类似。与标准 C 堆的不同之处在于，该内存字节池可以包含多个内存字节池。此外，线程可在池中挂起，直到请求的内存可用为止。

内存字节池的分配与传统的 malloc 调用类似，其中包含所需的内存量（以字节为单位）。内存采用“首次适应”的方式从池中分配；例如，使用满足请求的第一个可用内存块。此块中多余的内存会转换为新块，并放回可用内存列表中。此过程称为碎片。

相邻的可用内存块在后续的分配搜索过程中合并为一个足够大的可用内存块。此过程称为碎片整理。

每个内存字节池都是一个公用资源。除了不能从 ISR 调用内存字节服务之外，ThreadX 对如何使用池没有任何限制。

创建内存字节池

内存字节池由应用程序线程在初始化期间或运行时创建。应用程序中内存字节池的数量没有限制。

池容量

内存字节池中可分配的字节数略小于创建期间指定的字节数。这是因为可用内存区域的管理带来了一些开销。池中的每个可用内存块都需要相当于两个 C 指针的开销。此外，创建的池包含两个块：一个较大的可用块和在内存区域末端永久分配的一个较小的块。这个分配块用于提高分配算法的性能。这样就无需在合并期间持续检查池区域末端。

在运行时，池中的开销通常会增加。如果分配奇数字节数，系统会加以填充，以确保正确对齐下一个内存块。此外，随着池变得更加零碎，开销也会增加。

池的内存区域

内存字节池的内存区域在创建过程中指定。与 ThreadX 中的其他内存区域一样，该区域可以位于目标地址空间的任何位置。这是一项重要的功能，因为它提供了相当大的灵活性。例如，如果目标硬件有高速内存区域和低速内存区域，用户可以通过在每个区域中创建池来管理这两个区域的内存分配。

线程挂起

在等待池中的内存字节时，应用程序线程可能会挂起。当有足够的连续内存可用时，将为已挂起的线程提供其请求的内存，并且恢复线程。

如果同一内存字节池中挂起多个线程，则按这些线程挂起的顺序 (FIFO) 为其提供内存（恢复）。

不过，如果应用程序在信号灯发出取消线程挂起的字节释放调用之前调用 tx_byte_pool_prioritize，还可以恢复优先级。字节池设置优先级服务将最高优先级的线程置于挂起列表的前面，让所有其他挂起的线程采用相同的 FIFO 顺序。

运行时字节池性能信息

ThreadX 提供可选的运行时字节池性能信息。如果 ThreadX 库和应用程序是在定义 TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX 会累积以下信息。

整个系统的总数：

- 分配数
- 版本

- 搜索的片段数
- 合并的片段数
- 创建的片段数
- 分配挂起数
- 分配超时数

每个字节池的总数：

- 分配数
- 版本
- 搜索的片段数
- 合并的片段数
- 创建的片段数
- 分配挂起数
- 分配超时数

此信息在运行时通过 tx_byte_pool_performance_info_get 和 tx_byte_pool_performance_system_info_get 服务提供。字节池性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。例如，"分配挂起数"相对较高可能表明字节池太小。

内存字节池控制块 TX_BYTE_POOL

每个内存字节池的特征都可在其控制块中找到。该控制块包含诸如池中可用的字节数等有用的信息。此结构在 tx_api.h 文件中定义。

池控制块也可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

非确定性行为

尽管内存字节池提供了最灵活的内存分配，但这些池也受一些非确定性行为的影响。例如，内存字节池可能有 2,000 字节的可用内存，但可能无法满足 1,000 字节的分配请求。这是因为无法保证有多少可用字节是连续的。即使存在 1,000 字节可用块，也不能保证找到此块需要多长时间。完全有可能需要搜索整个内存池来查找这个 1,000 字节块。

> **TIP**
> 由于内存字节池的不确定性行为，通常应避免在需要确定性实时行为的区域中使用内存字节服务。许多应用程序会在初始化或运行时配置期间预先分配其所需的内存。

覆盖内存块

务必确保已分配内存的用户不会在其边界之外写入。如果发生这种情况，则会损坏其相邻的内存区域（通常是后续区域）。结果不可预测，且对于程序执行来说通常是灾难性的。

应用程序计时器

快速响应异步外部事件是嵌入式实时应用程序中最重要的功能。但是，其中的许多应用程序还必须按预定的时间间隔执行某些活动。

借助 ThreadX 应用程序计时器，应用程序能够按特定的时间间隔执行应用程序 C 函数。应用程序计时器也可能只过期一次。这种类型的计时器称为单次计时器，而重复间隔计时器称为定期计时器 。

每个应用程序计时器都是一个公用资源。ThreadX 对如何使用应用程序计时器没有任何限制。

计时器间隔

在 ThreadX 中，时间间隔通过定期计时器中断来测量。每个计时器中断称为计时器时钟周期。计时器时钟周期之间的实际时间由应用程序指定，但 10 毫秒是大多数实现的标准时间。定期计时器设置通常位于 tx_initialize_low_level 程序集文件中。

值得一提的是，基础硬件必须能够生成定期中断，应用程序计时器才会正常运行。在某些情况下，处理器具有内置的定期中断功能。如果处理器没有此功能，用户的主板必须包含可生成定期中断的外围设备。

> **IMPORTANT**
> 即使没有定期中断源，ThreadX 仍可正常工作。但是，随后将禁用所有与计时器相关的处理。这包括时间切片、挂起超时和计时器服务。

计时器准确性

计时器过期时间根据时钟周期指定。达到每个计时器时钟周期时，指定到期值将减一。由于应用程序计时器可在计时器中断（或计时器时钟周期）之前启用，因此，实际过期时间可能会提前一个时钟周期。

如果计时器时钟周期速率为 10 毫秒，应用程序计时器可能会提前 10 毫秒过期。与 1 秒计时器相比，这对 10 毫秒计时器更重要。当然，增加计时器中断频率会减少此误差范围。

计时器执行

应用程序计时器按照其激活的顺序执行。例如，如果创建了三个具有相同过期值的计时器并已激活，这些计时器对应的过期函数将保证按它们激活的顺序执行。

创建应用程序计时器

应用程序计时器由应用程序线程在初始化期间或运行时创建。应用程序中应用程序计时器的数量没有限制。

运行时应用程序计时器性能信息

ThreadX 提供可选的运行时应用程序计时器性能信息。如果 ThreadX 库和应用程序是在定义 TX_TIMER_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX 会累积以下信息。

整个系统的总数：

- 激活数
- 停用数
- 重新激活数（定期计时器）
- expirations
- 过期调整数

每个应用程序计时器的总数：

- 激活数
- 停用数
- 重新激活数（定期计时器）
- expirations
- 过期调整数

此信息在运行时通过 tx_timer_performance_info_get 和 tx_timer_performance_system_info_get 服务提供。应用程序计时器性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。

应用程序计时器控制块 TX_TIMER

每个应用程序计时器的特征都可在其控制块中找到。该控制块包含诸如 32 位过期标识值等有用信息。此结构在 tx_api.h 文件中定义。

应用程序计时器控制块可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

计时器过多

默认情况下，应用程序计时器在优先级为 0 时运行的隐藏系统线程中执行，该线程的优先级通常比任何应用程序线程都高。因此，在应用程序计时器内进行处理应保持最小值。

如果可能，还应尽可能避免使用在每个时钟周期过期的计时器。这种情况可能导致应用程序的开销过大。

> **IMPORTANT**
> 如前所述，应用程序计时器在隐藏的系统线程中执行。因此，请不要在应用程序计时器的过期函数内执行任何 ThreadX 服务调用时选择挂起。

相对时间

除了前面所述的应用程序计时器，ThreadX 还提供单个连续递增的 32 位时钟周期计数器。每次发生计时器中断时，时钟周期计数器或时间就会加一。

应用程序可以通过分别调用 tx_time_get 和 tx_time_set 来读取或设置此 32 位计数器。此时钟周期计数器的使用完全由应用程序确定。ThreadX 不在内部使用此计时器。

中断

快速响应异步事件是嵌入式实时应用程序的主函数。应用程序知道此类事件是因为硬件中断而出现的。

中断是处理器执行中的异步更改。发生中断时，中断处理器通常会将当前执行的一小部分保存在堆栈上，并将控制权转交给相应的中断向量。中断向量基本上只是负责处理特定类型中断的例程的地址。确切的中断处理过程特定于处理器。

中断控制

使用 tx_interrupt_control 服务，应用程序可以启用和禁用中断。上一个中断启用/禁用状态由此服务返回。值得一提的是，中断控制只会影响当前正在执行的程序段。例如，如果某个线程禁用中断，这些中断仅在该线程执行期间保持禁用状态。

> **NOTE**
> 不可屏蔽的中断 (NMI) 是无法通过硬件禁用的中断。此类中断可供 ThreadX 应用程序使用。但是，不允许应用程序的 NMI 处理例程使用 ThreadX 上下文管理或任何 API 服务。

ThreadX 托管中断

ThreadX 提供具有完整中断管理的应用程序。此管理包括保存和还原中断执行的上下文。此外，ThreadX 允许在中断服务例程 (ISR) 内调用特定服务。下面是允许从应用程序 ISR 调用的 ThreadX 服务的列表。

```
tx_block_allocate
tx_block_pool_info_get tx_block_pool_prioritize
tx_block_pool_performance_info_get
tx_block_pool_performance_system_info_get tx_block_release
tx_byte_pool_info_get tx_byte_pool_performance_info_get
tx_byte_pool_performance_system_info_get
tx_byte_pool_prioritize tx_event_flags_info_get
tx_event_flags_get tx_event_flags_set
tx_event_flags_performance_info_get
tx_event_flags_performance_system_info_get
tx_event_flags_set_notify tx_interrupt_control
tx_mutex_performance_info_get
tx_mutex_performance_system_info_get tx_queue_front_send
tx_queue_info_get tx_queue_performance_info_get
tx_queue_performance_system_info_get tx_queue_prioritize
tx_queue_receive tx_queue_send tx_semaphore_get
tx_queue_send_notify tx_semaphore_ceiling_put
tx_semaphore_info_get tx_semaphore_performance_info_get
tx_semaphore_performance_system_info_get
tx_semaphore_prioritize tx_semaphore_put tx_thread_identify
tx_semaphore_put_notify tx_thread_entry_exit_notify
tx_thread_info_get tx_thread_resume
tx_thread_performance_info_get
tx_thread_performance_system_info_get
tx_thread_stack_error_notify tx_thread_wait_abort tx_time_get
tx_time_set tx_timer_activate tx_timer_change
tx_timer_deactivate tx_timer_info_get
tx_timer_performance_info_get
tx_timer_performance_system_info_get
```

IMPORTANT

不允许从 ISR 中挂起。因此，从 ISR 发出的所有 ThreadX 服务调用的 wait_option 参数必须设置为 TX_NO_WAIT **。

ISR 模板

若要管理应用程序中断，必须在应用程序 ISR 的开头和结尾调用多个 ThreadX 实用程序。中断处理的确切格式因端口而异。

以下小代码段是大多数 ThreadX 托管 ISR 的典型代码段。在大多数情况下，此处理采用汇编语言。

```
_application_ISR_vector_entry:

; Save context and prepare for

; ThreadX use by calling the ISR

; entry function.

CALL _tx_thread_context_save

; The ISR can now call ThreadX

; services and its own C functions

; When the ISR is finished, context

; is restored (or thread preemption)

; by calling the context restore ; function. Control does not return!

JUMP _tx_thread_context_restore
```

高频中断

某些中断发生的频率很高，导致在每次中断时保存和还原完整上下文消耗过多的处理带宽。在这种情况下，应用程序通常有一种小型汇编语言 ISR，用于对大多数此类高频中断执行一定量的处理。

在某个时间点之后，这种小型 ISR 可能需要与 ThreadX 交互。这种交互通过调用上述模板所述的入口和出口函数来实现。

中断延迟

ThreadX 在短时间内锁定中断。禁用中断的最大时间与保存或还原线程上下文所需的时间大致相同。

第 4 章 - Azure RTOS ThreadX 服务的说明

2021/4/29 •

本章按字母顺序介绍所有 Azure RTOS ThreadX 服务。它们的名称经过设计，以便将所有相似的服务组合在一起。在以下说明的"返回值"部分中，以粗体显示的值不受用于禁用 API 错误检查的 NX_DISABLE_ERROR_CHECKING 定义影响，而不以粗体显示的值则是完全禁用的。此外，"可以抢占"标题下列出的"是"表示调用该服务可能会恢复更高优先级的线程，从而抢占调用线程 。

tx_block_allocate

分配固定大小的内存块

原型

```
UINT tx_block_allocate(
    TX_BLOCK_POOL *pool_ptr,
    VOID **block_ptr,
    ULONG wait_option);
```

说明

此服务从指定的内存池中分配固定大小的内存块。内存块的实际大小是在创建内存池的过程中确定的。

> **IMPORTANT**
> 请务必确保应用程序代码不在已分配的内存块之外写入。如果发生这种情况，则会损坏其相邻的内存块（通常是后面的内存块）。其结果不可预测，通常很严重！

参数

- **pool_ptr**
  指向之前创建的内存块池的指针。
- **block_ptr**
  指向目标块指针的指针。成功分配时，已分配内存块的地址就位于此参数所指向的位置。
- **wait_option**
  定义此服务在没有可用的内存块时的行为方式。等待选项的定义如下：
  - TX_NO_WAIT (0x00000000) - 如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回 。如果从非线程（例如初始化、计时器或 ISR）调用服务，则这是唯一有效的选项。
  - TX_WAIT_FOREVER (0xFFFFFFFF) - 选择 TX_WAIT_FOREVER 会导致发出调用的线程无限期挂起，直到内存块可用为止 。
  - 超时值（0x00000001 至 0xFFFFFFFE）- 如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待内存块时发出调用的线程保持挂起的最大计时器时钟周期数。

返回值

- **TX_SUCCESS**：(0x00) 成功分配内存块。
- **TX_DELETED**：(0x01) 线程挂起时删除了内存块池。
- **TX_NO_MEMORY**：(0X10) 服务无法在指定的等待时间内分配内存块。
- **TX_WAIT_ABORTED**：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- **TX_POOL_ERROR**：(0x02) 内存块池指针无效。
- **TX_WAIT_ERROR**：(0x04) 从非线程调用时指定了除 TX_NO_WAIT 以外的等待选项。

- **TX_PTR_ERROR**：(0x03) 指向目标指针的指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_BLOCK_POOL my_pool;
unsigned char *memory_ptr;

UINT status;

/* Allocate a memory block from my_pool. Assume that the pool has
already been created with a call to tx_block_pool_create. */

status = tx_block_allocate(&my_pool, (VOID **) &memory_ptr,
  TX_NO_WAIT);

/* If status equals TX_SUCCESS, memory_ptr contains the address of
the allocated block of memory. */
```

另请参阅

- tx_block_pool_create
- tx_block_pool_delete
- tx_block_pool_info_get
- tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_pool_prioritize
- tx_block_release

tx_block_pool_create

创建固定大小内存块的池

原型

```
UINT tx_block_pool_create(
    TX_BLOCK_POOL pool_ptr,
    CHAR name_ptr,
    ULONG block_size,
    VOID pool_start,
    ULONG pool_size);
```

说明

此服务创建固定大小内存块的池。使用以下公式将指定的内存区域划分为尽可能多的固定大小内存块：

总块数 =（总字节数）/（块大小 + sizeof(void *)）

> **NOTE**
>
> *每个内存块包含一个开销指针，该指针对用户不可见，在前面的公式中用“sizeof(void)”表示。

参数

- **pool_ptr**：指向内存块池控制块的指针。
- **name_ptr**：指向内存块池名称的指针。
- **block_size**：每个内存块中的字节数。
- **pool_start**：内存块池的起始地址。起始地址必须与 ULONG 数据类型的大小一致。
- **pool_size**：内存块池可用的总字节数。

返回值

- **TX_SUCCESS**：(0X00) 成功创建内存块池。
- **TX_POOL_ERROR**：(0x02) 内存块池指针无效。指针为 NULL 或池已创建。
- **TX_PTR_ERROR**：(0x03) 池的起始地址无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。
- **TX_SIZE_ERROR**：(0x05) 池大小无效。

允许来自

初始化和线程

可以抢占

否

示例

```
TX_BLOCK_POOL my_pool;

UINT status;

/* Create a memory pool whose total size is 1000 bytes starting at
address 0x100000. Each block in this pool is defined to be 50 bytes
long. */
status = tx_block_pool_create(&my_pool, "my_pool_name",
  50, (VOID *) 0x100000, 1000);

/* If status equals TX_SUCCESS, my_pool contains 18 memory blocks
of 50 bytes each. The reason there are not 20 blocks in the pool is
because of the one overhead pointer associated with each block. */
```

另请参阅

- tx_block_allocate、tx_block_pool_delete
- tx_block_pool_info_get、tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_pool_prioritize、tx_block_release

tx_block_pool_delete

删除内存块池

原型

```
UINT tx_block_pool_delete(TX_BLOCK_POOL *pool_ptr);
```

说明

此服务删除指定的块内存池。所有挂起并等待来自此池的内存块的线程都将恢复，并获得 TX_DELETED 返回状态。

> **NOTE**
>
> 应用程序负责管理与池关联的内存区域，该内存区域在此服务完成后可用。此外，应用程序必须阻止使用已删除的池或其以前的内存块。

参数

- **pool_ptr**：指向之前创建的内存块池的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功删除内存块池。
- **TX_POOL_ERROR**：(0x02) 内存块池指针无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程数

可以抢占

是

示例

```
TX_BLOCK_POOL my_pool;

UINT          status;

/* Delete entire memory block
pool. Assume that the pool has already been created with a call to
tx_block_pool_create. */
status = tx_block_pool_delete(&my_pool);

/* If status equals TX_SUCCESS, the memory block pool is deleted.*/
```

另请参阅

- tx_block_allocate
- tx_block_pool_create
- tx_block_pool_info_get、tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_pool_prioritize、tx_block_release

tx_block_pool_info_get

检索有关块池的信息

原型

```
UINT tx_block_pool_info_get(
    TX_BLOCK_POOL *pool_ptr,
    CHAR **name,
    ULONG *available,
    ULONG *total_blocks,
    TX_THREAD **first_suspended,
    ULONG *suspended_count,
    TX_BLOCK_POOL **next_pool);
```

说明

此服务检索所指定块内存池的相关信息。

参数

- **pool_ptr**：指向之前创建的内存块池的指针。
- **name**：指向块池名称指针这一目标的指针。
- **available**：指向块池中可用块数这一目标的指针。
- **total_blocks**：指向块池中总块数这一目标的指针。
- **first_suspended**：指向此块池的挂起列表上第一个线程的指针这一目标的指针。
- **suspended_count**：指向此块池中当前挂起的线程数这一目标的指针。
- **next_pool**：指向下一个已创建的块池的指针这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0x00) 成功检索块池信息。
- **TX_POOL_ERROR**：(0x02) 内存块池指针无效。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_BLOCK_POOL my_pool;
CHAR *name;
ULONG available;
ULONG total_blocks;
TX_THREAD *first_suspended;
ULONG suspended_count;
TX_BLOCK_POOL *next_pool;
UINT status;

/* Retrieve information about the previously created
block pool "my_pool." */
status = tx_block_pool_info_get(&my_pool, &name,
  &available,&total_blocks,
  &first_suspended, &suspended_count,
  &next_pool);

/* If status equals TX_SUCCESS, the information requested is
valid. */
```

另请参阅

- tx_block_allocate
- tx_block_pool_create
- tx_block_pool_delete
- tx_block_pool_info_get、tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_pool_prioritize、tx_block_release

tx_block_pool_performance_info_get

获取块池性能信息

原型

```
UINT tx_block_pool_performance_info_get(
    TX_BLOCK_POOL *pool_ptr,
    ULONG *allocates,
    ULONG *releases,
    ULONG *suspensions,
    ULONG *timeouts));
```

说明

此服务检索所指定内存块池的相关性能信息。

> **IMPORTANT**
> 必须使用为此服务定义的 TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。

参数

- **pool_ptr**：指向之前创建的内存块池的指针。
- **allocates**：指向对此池执行的分配请求数这一目标的指针。
- **releases**：指向对此池执行的释放请求数这一目标的指针。
- **suspensions**：指向此池的线程分配挂起次数这一目标的指针。
- **timeouts**：指向此池的分配挂起超时次数这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0x00) 成功获取块池性能信息。
- **TX_PTR_ERROR**：(0x03) 块池指针无效。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_BLOCK_POOL my_pool;
ULONG allocates;
ULONG releases;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on the previously created block
pool. */
status = tx_block_pool_performance_info_get(&my_pool, &allocates,
  &releases,
  &suspensions,
  &timeouts);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

另请参阅

- tx_block_allocate

- tx_block_pool_create
- tx_block_pool_delete
- tx_block_pool_info_get
- tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_release

tx_block_pool_performance_system_info_get

获取块池系统性能信息

原型

```
UINT tx_block_pool_performance_system_info_get(
    ULONG *allocates,
    ULONG *releases,
    ULONG *suspensions,
    ULONG *timeouts);
```

说明

此服务检索应用程序中所有内存块池的相关性能信息。

> **IMPORTANT**
> 必须使用为此服务定义的 TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。

参数

- **allocates**：指向对所有块池执行的分配请求总数这一目标的指针。
- **releases**：指向对所有块池执行的释放请求总数这一目标的指针。
- **suspensions**：指向所有块池的线程分配挂起总次数这一目标的指针。
- **timeouts**：指向所有块池的分配挂起超时总次数这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0x00) 成功获取块池系统性能信息。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
ULONG       allocates;
ULONG       releases;
ULONG       suspensions;
ULONG       timeouts;

/* Retrieve performance information on all the block pools in
the system. */
status = tx_block_pool_performance_system_info_get(&allocates,
    &releases,&suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_block_allocate
- tx_block_pool_create
- tx_block_pool_delete
- tx_block_pool_info_get
- tx_block_pool_performance_info_get
- tx_block_pool_prioritize
- tx_block_release

tx_block_pool_prioritize

设置块池挂起列表的优先级

原型

```
UINT tx_block_pool_prioritize(TX_BLOCK_POOL *pool_ptr);
```

说明

此服务将因此池中某个内存块而挂起的最高优先级线程放在挂起列表前面。所有其他线程保持它们挂起时的相同 FIFO 顺序。

参数

- **pool_ptr**：指向内存块池控制块的指针。

返回值

- **TX_SUCCESS**：(0x00) 成功为块池设置优先级。
- **TX_POOL_ERROR**：(0x02) 内存块池指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_BLOCK_POOL my_pool;
UINT status;

/* Ensure that the highest priority thread will receive
the next free block in this pool. */
status = tx_block_pool_prioritize(&my_pool);

/* If status equals TX_SUCCESS, the highest priority
suspended thread is at the front of the list. The
next tx_block_release call will wake up this thread. */
```

另请参阅

- tx_block_allocate
- tx_block_pool_create
- tx_block_pool_delete
- tx_block_pool_info_get
- tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_release

tx_block_release

释放固定大小的内存块

原型

```
UINT tx_block_release(VOID *block_ptr);
```

说明

此服务将以前分配的块释放回其关联的内存池。如果有一个或多个线程挂起并等待此池中的内存块，则为第一个挂起的线程提供此内存块并恢复该线程。

> **IMPORTANT**
> 内存块区域已释放回池中后，应用程序应阻止使用该内存块区域。

参数

- **block_ptr**：指向之前分配的内存块的指针。

返回值

- **TX_SUCCESS**：(0x00) 成功释放内存块。
- **TX_PTR_ERROR**：(0x03) 指向内存块的指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_BLOCK_POOL my_pool;
unsigned char *memory_ptr;
UINT status;

/* Release a memory block back to my_pool. Assume that the
pool has been created and the memory block has been
allocated. */
status = tx_block_release((VOID *) memory_ptr);

/* If status equals TX_SUCCESS, the block of memory pointed
to by memory_ptr has been returned to the pool. */
```

另请参阅

- tx_block_allocate
- tx_block_pool_create
- tx_block_pool_delete
- tx_block_pool_info_get
- tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_pool_prioritize

tx_byte_allocate

分配内存的字节

原型

```
UINT tx_byte_allocate(
    TX_BYTE_POOL *pool_ptr,
    VOID **memory_ptr,
    ULONG memory_size,
    ULONG wait_option);
```

说明

此服务从指定的内存字节池中分配指定的字节数。

> **IMPORTANT**
> 请务必确保应用程序代码不在已分配的内存块之外写入。如果发生这种情况，则会损坏其相邻的内存块（通常是后面的内存块）。其结果不可预测，通常很严重！

> **NOTE**
> 此服务的性能是块大小和池中碎片量的函数。因此，在执行的时间关键型线程期间不应使用此服务。

参数

- pool_ptr
  指向之前创建的内存块池的指针。
- memory_ptr
  指向目标内存指针的指针。成功分配时，已分配内存区域的地址就位于在此参数所指向的位置。
- memory_size
  请求的字节数。

- **wait_option**
  定义此服务在没有足够内存可用时的行为方式。等待选项的定义如下：
  - TX_NO_WAIT (0x00000000) - 如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回 。如果从初始化调用服务，则这是唯一有效的选项。
  - TX_WAIT_FOREVER (0xFFFFFFFF) - 选择 TX_WAIT_FOREVER 会导致发出调用的线程无限期挂起，直到有足够内存可用为止 。
  - 超时值（0x00000001 至 0xFFFFFFFE）- 如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待内存时发出调用的线程保持挂起的最大计时器时钟周期数。

返回值

- **TX_SUCCESS**：(0x00) 成功分配内存。
- **TX_DELETED**：(0x01) 线程挂起时删除了内存池。
- **TX_NO_MEMORY**：(0X10) 服务无法在指定的等待时间内分配内存。
- **TX_WAIT_ABORTED** (0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- **TX_POOL_ERROR**：(0x02) 内存池指针无效。
- **TX_PTR_ERROR**：(0x03) 指向目标指针的指针无效。
- **TX_SIZE_ERROR**：(0X05) 所请求的大小为零或超过池大小。
- **TX_WAIT_ERROR**：(0x04) 从非线程调用时指定了除 TX_NO_WAIT 以外的等待选项。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

初始化和线程

可以抢占

是

示例

```
TX_BYTE_POOL my_pool;
unsigned char*memory_ptr;
UINT status;
/* Allocate a 112 byte memory area from my_pool. Assume
that the pool has already been created with a call to
tx_byte_pool_create. */
status = tx_byte_allocate(&my_pool, (VOID **) &memory_ptr,
    112, TX_NO_WAIT);

/* If status equals TX_SUCCESS, memory_ptr contains the
address of the allocated memory area. */
```

另请参阅

- tx_byte_pool_create
- tx_byte_pool_delete
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize
- tx_byte_release

tx_byte_pool_create

创建内存字节池

原型

```
UINT tx_byte_pool_create(
    TX_BYTE_POOL *pool_ptr,
    CHAR *name_ptr,
    VOID *pool_start,
    ULONG pool_size);
```

说明

此服务在指定的区域中创建内存字节池。最初，池基本上仅包含一个非常大的可用块。但是，随着不断进行分配，池将分成多个较小的块。

参数

- **pool_ptr**：指向内存池控制块的指针。
- **name_ptr**：指向内存池名称的指针。
- **pool_start**：内存池的起始地址。起始地址必须与 ULONG 数据类型的大小一致。
- **pool_size**：内存池可用的总字节数。

返回值

- **TX_SUCCESS**：(0X00) 成功创建内存池。
- **TX_POOL_ERROR**：(0x02) 内存池指针无效。指针为 NULL 或池已创建。
- **TX_PTR_ERROR**：(0x03) 池的起始地址无效。
- **TX_SIZE_ERROR**：(0x05) 池大小无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

初始化和线程

可以抢占

否

示例

```
TX_BYTE_POOL my_pool;
UINT status;
/* Create a memory pool whose total size is 2000 bytes
starting at address 0x500000. */
status = tx_byte_pool_create(&my_pool, "my_pool_name",
    (VOID *) 0x500000, 2000);

/* If status equals TX_SUCCESS, my_pool is available for
allocating memory. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_delete
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize
- tx_byte_release

tx_byte_pool_delete

删除内存字节池

原型

```
UINT tx_byte_pool_delete(TX_BYTE_POOL *pool_ptr);
```

说明

此服务删除指定的内存字节池。所有挂起并等待来自此池的内存的线程都将恢复，并获得 TX_DELETED 返回状态。

> **IMPORTANT**
> 应用程序负责管理与池关联的内存区域，该内存区域在此服务完成后可用。此外，应用程序必须阻止使用已删除的池或先前从其分配的内存。

参数

- **pool_ptr**：指向之前创建的内存池的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功删除内存池。
- **TX_POOL_ERROR**：(0x02) 内存池指针无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程数

可以抢占

是

示例

```
TX_BYTE_POOL my_pool;
UINT status;
/* Delete entire memory pool. Assume that the pool has already
been created with a call to tx_byte_pool_create. */
status = tx_byte_pool_delete(&my_pool);

/* If status equals TX_SUCCESS, memory pool is deleted. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_create
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize
- tx_byte_release

tx_byte_pool_info_get

检索有关字节池的信息

原型

```
UINT tx_byte_pool_info_get(
    TX_BYTE_POOL *pool_ptr,
    CHAR **name,
    ULONG *available,
    ULONG *fragments,
    TX_THREAD **first_suspended,
    ULONG *suspended_count,
    TX_BYTE_POOL **next_pool);
```

说明

此服务检索所指定内存字节池的相关信息。

参数

- **pool_ptr**：指向之前创建的内存池的指针。
- **name**：指向字节池名称指针这一目标的指针。
- **available**：指向池中可用字节数这一目标的指针。
- **fragments**：指向字节池中内存片段总数这一目标的指针。
- **first_suspended**：指向此字节池的挂起列表上第一个线程的指针这一目标的指针。
- **suspended_count**：指向此字节池中当前挂起的线程数这一目标的指针。
- **next_pool**：指向下一个已创建的字节池的指针这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0x00) 成功检索池信息。
- **TX_POOL_ERROR**：(0x02) 内存池指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_BYTE_POOL my_pool;
CHAR *name;
ULONG available;
ULONG fragments;
TX_THREAD *first_suspended;
ULONG suspended_count;
TX_BYTE_POOL *next_pool;
UINT status;

/* Retrieve information about the previously created
block pool "my_pool." */
status = tx_byte_pool_info_get(&my_pool, &name,
  &available, &fragments,
  &first_suspended, &suspended_count,
  &next_pool);

/* If status equals TX_SUCCESS, the information requested is
valid. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_create
- tx_byte_pool_delete
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize
- tx_byte_release

tx_byte_pool_performance_info_get

获取字节池性能信息

原型

```
UINT tx_byte_pool_performance_info_get(
    TX_BYTE_POOL *pool_ptr,
    ULONG *allocates,
    ULONG *releases,
    ULONG *fragments_searched,
    ULONG *merges,
    ULONG *splits,
    ULONG *suspensions,
    ULONG *timeouts);
```

说明

此服务检索所指定内存字节池的相关性能信息。

> **IMPORTANT**
> 必须使用为此服务定义的 TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。

参数

- **pool_ptr**：指向之前创建的内存字节池的指针。
- **allocates**：指向对此池执行的分配请求数这一目标的指针。
- **releases**：指向对此池执行的释放请求数这一目标的指针。
- **fragments_searched**：指向此池上分配请求期间搜索的内部内存片段数这一目标的指针。
- **merges**：指向此池上分配请求期间合并的内部内存块数这一目标的指针。
- **splits**：指向此池上分配请求期间创建的内部内存块拆分（片段）数这一目标的指针。
- **suspensions**：指向此池的线程分配挂起次数这一目标的指针。
- **timeouts**：指向此池的分配挂起超时次数这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0x00) 成功获取字节池性能信息。
- **TX_PTR_ERROR**：(0x03) 字节池指针无效。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_BYTE_POOL my_pool;
ULONG fragments_searched;
ULONG merges;
ULONG splits;
ULONG allocates;
ULONG releases;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on the previously created byte
pool. */
status = tx_byte_pool_performance_info_get(&my_pool,
  &fragments_searched,
  &merges, &splits,
  &allocates, &releases,
  &suspensions,&timeouts);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_create
- tx_byte_pool_delete
- tx_byte_pool_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize
- tx_byte_release

tx_byte_pool_performance_system_info_get

获取字节池系统性能信息

原型

```
UINT tx_byte_pool_performance_system_info_get(
    ULONG *allocates,
    ULONG *releases,
    ULONG *fragments_searched,
    ULONG *merges,
    ULONG *splits,
    ULONG *suspensions,
    ULONG *timeouts);
```

说明

此服务检索系统中所有内存字节池的相关性能信息。

> **IMPORTANT**
> 必须使用为此服务定义的 TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。

参数

- **allocates**：指向对此池执行的分配请求数这一目标的指针。

- **releases**：指向对此池执行的释放请求数这一目标的指针。
- **fragments_searched**：指向所有字节池上分配请求期间搜索的内部内存片段总数这一目标的指针。
- **merges**：指向所有字节池上分配请求期间合并的内部内存块总数这一目标的指针。
- **splits**：指向所有字节池上分配请求期间创建的内部内存块拆分（片段）总数这一目标的指针。
- **suspensions**：指向所有字节池的线程分配挂起总次数这一目标的指针。
- **timeouts**：指向所有字节池的分配挂起超时总次数这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0x00) 成功获取字节池性能信息。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
ULONG fragments_searched;
ULONG merges;
ULONG splits;
ULONG allocates;
ULONG releases;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on all byte pools in the
system. */
status =
tx_byte_pool_performance_system_info_get(&fragments_searched,
  &merges, &splits, &allocates, &releases,
  &suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_create
- tx_byte_pool_delete
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_prioritize
- tx_byte_release

tx_byte_pool_prioritize

设置字节池挂起列表的优先级

原型

```
UINT tx_byte_pool_prioritize(TX_BYTE_POOL *pool_ptr);
```

说明

此服务将因此池中的内存而挂起的最高优先级线程放在挂起列表前面。所有其他线程保持它们挂起时的相同 FIFO 顺序。

参数

- **pool_ptr**：指向内存池控制块的指针。

返回值

- **TX_SUCCESS**：(0x00) 成功为内存池设置优先级。
- **TX_POOL_ERROR**：(0x02) 内存池指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_BYTE_POOL my_pool;
UINT status;

/* Ensure that the highest priority thread will receive
the next free memory from this pool. */
status = tx_byte_pool_prioritize(&my_pool);

/* If status equals TX_SUCCESS, the highest priority
suspended thread is at the front of the list. The
next tx_byte_release call will wake up this thread,
if there is enough memory to satisfy its request. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_create
- tx_byte_pool_delete
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_release

tx_byte_release

将字节释放回内存池

原型

```
UINT tx_byte_release(VOID *memory_ptr);
```

说明

此服务将以前分配的内存区域释放回其关联的池。如果有一个或多个挂起的线程正在等待此池中的内存，则会为每个挂起的线程分配内存并恢复该线程，直到内存耗尽或不再有任何挂起的线程为止。向挂起的线程分配内存这一过程始终从第一个挂起的线程开始。

> **IMPORTANT**
> 应用程序必须阻止在释放某个内存区域后使用该区域。

参数

- **memory_ptr**：指向之前分配的内存区域的指针。

返回值

- **TX_SUCCESS**：(0x00) 成功释放内存。
- **TX_PTR_ERROR**：(0x03) 内存区域指针无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

初始化和线程

可以抢占

是

示例

```
unsigned char *memory_ptr;
UINT status;

/* Release a memory back to my_pool. Assume that the memory
area was previously allocated from my_pool. */
status = tx_byte_release((VOID *) memory_ptr);

/* If status equals TX_SUCCESS, the memory pointed to by
memory_ptr has been returned to the pool. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_create
- tx_byte_pool_delete
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize

tx_event_flags_create

创建事件标志组

原型

```
UINT tx_event_flags_create(
    TX_EVENT_FLAGS_GROUP *group_ptr,
    CHAR *name_ptr);
```

说明

此服务创建一组事件标志(32 个)。该组中的所有 32 个事件标志都被初始化为零。每个事件标志由一个位表示。

参数

- **group_ptr**：指向事件标志组控制块的指针。
- **name_ptr**：指向事件标志组名称的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功创建事件组。
- **TX_GROUP_ERROR**：(0x06) 事件组指针无效。指针为 NULL 或事件组已创建。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

初始化和线程

可以抢占

否

示例

```
TX_EVENT_FLAGS_GROUP my_event_group;
UINT status;

/* Create an event flags group. */
status = tx_event_flags_create(&my_event_group,
  "my_event_group_name");

/* If status equals TX_SUCCESS, my_event_group is ready
for get and set services. */
```

另请参阅

- tx_event_flags_delete
- tx_event_flags_get
- tx_event_flags_info_get
- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set
- tx_event_flags_set_notify

tx_event_flags_delete

删除事件标志组

原型

```
UINT tx_event_flags_delete(TX_EVENT_FLAGS_GROUP *group_ptr);
```

说明

此服务删除指定的事件标志组。所有挂起并等待此组中的事件的线程都将恢复，并获得 TX_DELETED 返回状态。

> **IMPORTANT**
> 在删除此事件标志组之前，应用程序必须确保完成（或禁用）此事件标志组的设置通知回调。此外，应用程序必须阻止将来再使用已删除的事件标志组。

参数

- **group_ptr**：指向以前创建的事件标志组的指针。

返回值

- **TX_SUCCESS**:(0X00) 成功删除事件标志组。
- **TX_GROUP_ERROR**:(0x06) 事件标志组指针无效。
- **NX_CALLER_ERROR**:(0x13) 此服务的调用方无效。

允许来自

线程数

可以抢占

是

示例

```
TX_EVENT_FLAGS_GROUP my_event_flags_group;
UINT status;

/* Delete event flags group. Assume that the group has
already been created with a call to
tx_event_flags_create. */
status = tx_event_flags_delete(&my_event_flags_group);

/* If status equals TX_SUCCESS, the event flags group is
deleted. */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_get
- tx_event_flags_info_get
- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set
- tx_event_flags_set_notify

tx_event_flags_get

从事件标志组获取事件标志

原型

```
UINT tx_event_flags_get(
    TX_EVENT_FLAGS_GROUP *group_ptr,
    ULONG requested_flags,
    UINT get_option,
    ULONG *actual_flags_ptr,
    ULONG wait_option);
```

说明

此服务从指定事件标志组检索事件标志。每个事件标志组都包含 32 个事件标志。每个标志由一个位表示。此服务可以检索由输入参数选择的各种事件标志组合。

参数

- **group_ptr**
  指向以前创建的事件标志组的指针。
- **requested_flags**

32 位无符号变量，表示请求的事件标志。

- **get_option**
  指定是否需要所有或任何请求的事件标志。以下是有效的选择：
  - **TX_AND** (0x02)
  - **TX_AND_CLEAR** (0x03)
  - **TX_OR** (0x00)
  - **TX_OR_CLEAR** (0x01)

    如果选择 TX_AND 或 TX_AND_CLEAR，则会指定所有事件标志必须出现在组中。如果选择 TX_OR 或 TX_OR_CLEAR，则会指定任何事件标志都符合要求。如果指定 TX_AND_CLEAR 或 TX_OR_CLEAR，则会清除满足请求的事件标志（设置为零）。
- **actual_flags_ptr**
  指向放置检索到的事件标志的位置这一目标的指针。注意，实际获得的标志可能包含没有请求的标志。
- **wait_option**
  定义未设置所选事件标志时服务的行为方式。等待选项的定义如下：
  - TX_NO_WAIT (0x00000000) - 如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从非线程（例如初始化、计时器或 ISR）调用服务，则这是唯一有效的选项。
  - TX_WAIT_FOREVER 超时值 (0xFFFFFFFF) - 选择 TX_WAIT_FOREVER 会导致发出调用的线程无限期挂起，直到事件标志可用为止。
  - 超时值（0x00000001 至 0xFFFFFFFE）- 如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待事件标志时发出调用的线程保持挂起的最大计时器时钟周期数。

返回值

- **TX_SUCCESS**：(0X00) 成功获取事件标志。
- **TX_DELETED**：(0x01) 线程挂起时删除了事件标志组。
- **TX_NO_EVENTS**：(0X07) 服务无法在指定的等待时间内获取指定的事件。
- **TX_WAIT_ABORTED**：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- **TX_GROUP_ERROR**：(0x06) 事件标志组指针无效。
- **TX_PTR_ERROR**：(0x03) 指向实际事件标志的指针无效。
- **TX_WAIT_ERROR**：(0x04) 从非线程调用时指定了除 TX_NO_WAIT 以外的等待选项。
- **TX_OPTION_ERROR**：(0x08) 指定的 get-option 无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_EVENT_FLAGS_GROUP my_event_flags_group;
ULONG actual_events;
UINT status;

/* Request that event flags 0, 4, and 8 are all set. Also,
if they are set they should be cleared. If the event
flags are not set, this service suspends for a maximum of
20 timer-ticks. */
status = tx_event_flags_get(&my_event_flags_group, 0x111,
    TX_AND_CLEAR, &actual_events, 20);

/* If status equals TX_SUCCESS, actual_events contains the
actual events obtained. */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_info_get
- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set
- tx_event_flags_set_notify

tx_event_flags_info_get

检索有关事件标志组的信息

原型

```
UINT tx_event_flags_info_get(
    TX_EVENT_FLAGS_GROUP *group_ptr,
    CHAR **name, ULONG *current_flags,
    TX_THREAD **first_suspended,
    ULONG *suspended_count,
    TX_EVENT_FLAGS_GROUP **next_group);
```

说明

此服务检索所指定事件标志组的相关信息。

参数

- **group_ptr**：指向事件标志组控制块的指针。
- **name**：指向事件标志组名称指针这一目标的指针。
- **current_flags**：指向事件标志组中当前设置标志这一目标的指针。
- **first_suspended**：指向此事件标志组的挂起列表上第一个线程的指针这一目标的指针。
- **suspended_count**：指向此事件标志组上当前挂起的线程数这一目标的指针。
- **next_group**：指向下一个创建的事件标志组的指针这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功检索事件组信息。

- **TX_GROUP_ERROR**：(0x06) 事件组指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_EVENT_FLAGS_GROUP my_event_group;
CHAR *name;
ULONG current_flags;
TX_THREAD *first_suspended;
ULONG suspended_count;
TX_EVENT_FLAGS_GROUP *next_group;
UINT status;

/* Retrieve information about the previously created
event flags group "my_event_group." */
status = tx_event_flags_info_get(&my_event_group, &name,
    &current_flags,
    &first_suspended, &suspended_count,
    &next_group);
/* If status equals TX_SUCCESS, the information requested is
valid. */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_get
- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set
- tx_event_flags_set_notify

tx_event_flags_performance_info_get

获取事件标志组性能信息

原型

```
UINT tx_event_flags_performance_info_get(
    TX_EVENT_FLAGS_GROUP *group_ptr,
    ULONG *sets, ULONG *gets,
    ULONG *suspensions,
    ULONG *timeouts);
```

说明

此服务检索所指定事件标志组的相关性能信息。

> **IMPORTANT**
>
> 必须使用为此服务定义的 TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。

参数

- **group_ptr**：指向以前创建的事件标志组的指针。
- **sets**：指向对此组执行的事件标志设置请求数这一目标的指针。
- **gets**：指向对此组执行的事件标志获取请求数这一目标的指针。
- **suspensions**：指向此组上的线程事件标志获取挂起数这一目标的指针。
- **timeouts**：指向此组上事件标志获取挂起超时数这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功获取事件标志组性能信息。
- **TX_PTR_ERROR**：(0x03) 事件标志组指针无效。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_EVENT_FLAGS_GROUP my_event_flag_group;
ULONG sets;
ULONG gets;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on the previously created event
flag group. */
status = tx_event_flags_performance_info_get(&my_event_flag_group,
    &sets, &gets, &suspensions,
    &timeouts);

/* If status is TX_SUCCESS the performance information was successfully
retrieved. */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_get
- tx_event_flags_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set
- tx_event_flags_set_notify

tx_event_flags_performance_system_info_get

检索系统性能信息

原型

```
UINT tx_event_flags_performance_system_info_get(
    ULONG *sets,
    ULONG *gets,
    ULONG *suspensions,
    ULONG *timeouts);
```

说明

此服务检索系统中所有事件标志组的相关性能信息。

> **IMPORTANT**
> 必须使用为此服务定义的 TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。

参数

- **sets**：指向对所有组执行的事件标志设置请求总数这一目标的指针。
- **gets**：指向对所有组执行的事件标志获取请求总数这一目标的指针。
- **suspensions**：指向所有组上的线程事件标志获取挂起总数这一目标的指针。
- **timeouts**：指向所有组上事件标志获取挂起超时总数这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功获取事件标志系统性能信息。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

示例

```
ULONG sets;
ULONG gets;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on all previously created event
flag groups. */
status = tx_event_flags_performance_system_info_get(&sets, &gets,
    &suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_get
- tx_event_flags_info_get
- tx_event_flags_performance_info_get
- tx_event_flags_set
- tx_event_flags_set_notify

tx_event_flags_set

在事件标志组中设置事件标志

原型

```
UINT tx_event_flags_set(
    TX_EVENT_FLAGS_GROUP *group_ptr,
    ULONG flags_to_set,
    UINT set_option);
```

说明

此服务根据指定的 set-option 设置或清除事件标志组中的事件标志。所有现已满足其事件标志请求的挂起线程都将恢复。

参数

- **group_ptr**
  指向以前创建的事件标志组控制块的指针。
- **flags_to_set**
  根据所选的 set-option，指定要设置或清除的事件标志。
- **set_option**
  指定将所指定的事件标志与该组的当前事件标志进行"AND"或"OR"运算。以下是有效的选择：
  - **TX_AND** (0x02)
  - **TX_OR** (0x00)

  如果选择 TX_AND，则会指定将所指定的事件标志与该组的当前事件标志进行"AND"运算。此选项通常用于清除组中的事件标志。否则，如果指定了 TX_OR，则所指定的事件标志与该组的当前事件标志进行"OR"运算。

返回值

- **TX_SUCCESS**：(0X00) 成功设置事件标志。
- **TX_GROUP_ERROR**：(0x06) 指向事件标志组的指针无效。
- **TX_OPTION_ERROR**：(0x08) 指定的 set-option 无效。

可以抢占

是

示例

```
TX_EVENT_FLAGS_GROUP my_event_flags_group;
UINT status;

/* Set event flags 0, 4, and 8. */
status = tx_event_flags_set(&my_event_flags_group,
    0x111, TX_OR);

/* If status equals TX_SUCCESS, the event flags have been
set and any suspended thread whose request was satisfied
has been resumed. */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_get

- tx_event_flags_info_get
- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set_notify

tx_event_flags_set_notify

设置事件标志时通知应用程序

原型

```
UINT tx_event_flags_set_notify(
    TX_EVENT_FLAGS_GROUP *group_ptr,
    VOID (*events_set_notify)(TX_EVENT_FLAGS_GROUP *));
```

说明

此服务注册一个通知回调函数，只要在指定的事件标志组中设置了一个或多个事件标志，就会调用该函数。通知回调的处理由应用程序定义。

参数

- **group_ptr**：指向以前创建的事件标志组的指针。
- **events_set_notify**：指向应用程序的事件标志设置通知函数的指针。如果此值为 TX_NULL，则禁用通知。

返回值

- **TX_SUCCESS**：(0X00) 成功注册事件标志设置通知。
- **TX_GROUP_ERROR**：(0x06) 事件标志组指针无效。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时禁用了通知功能。

示例

```
TX_EVENT_FLAGS_GROUP my_group;

/* Register the "my_event_flags_set_notify" function for monitoring
event flags set in the event flags group "my_group." */
status = tx_event_flags_set_notify(&my_group, my_event_flags_set_notify);

/* If status is TX_SUCCESS the event flags set notification function
was successfully registered. */
void my_event_flags_set_notify(TX_EVENT_FLAGS_GROUP *group_ptr)

/* One or more event flags was set in this group! */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_get
- tx_event_flags_info_get
- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set

tx_interrupt_control

启用和禁用中断

原型

```
UINT tx_interrupt_control(UINT new_posture);
```

说明

此服务启用或禁用由输入参数 new_posture 指定的中断。

> **NOTE**
>
> 如果从应用程序线程调用此服务，则中断状态仍然是该线程上下文的一部分。例如，如果线程调用此例程来禁用中断，然后挂起，则当该线程恢复时，将再次禁用中断。

> **WARNING**
>
> 在初始化期间，不应使用此服务来启用中断！这样做可能会导致不可预知的结果。

参数

- **new_posture**：此参数指定是禁用还是启用中断。合法值包括 TX_INT_DISABLE 和 TX_INT_ENABLE 。这些参数的实际值特定于端口。此外，某些处理体系结构可能还支持其他中断禁用状态。

返回值

- **previous posture**：此服务将以前的中断状态返回给调用方。这使该服务的用户可以在禁用中断后恢复先前的状态。

允许来自

线程、计时器和 ISR

可以抢占

否

示例

```
UINT my_old_posture;

/* Lockout interrupts */
my_old_posture = tx_interrupt_control(TX_INT_DISABLE);

/* Perform critical operations that need interrupts
locked-out.... */

/* Restore previous interrupt lockout posture. */
tx_interrupt_control(my_old_posture);
```

另请参阅

无

tx_mutex_create

创建互斥锁

原型

```
UINT tx_mutex_create(
    TX_MUTEX *mutex_ptr,
    CHAR *name_ptr,
    UINT priority_inherit);
```

说明

此服务为线程间互斥创建一个互斥锁，用于保护资源。

参数

- **mutex_ptr**：指向互斥锁控制块的指针。
- **name_ptr**：指向互斥锁名称的指针。
- **priority_inherit**：指定此互斥锁是否支持优先级继承。如果此值为 TX_INHERIT，则支持优先级继承。但是，如果指定了 TX_NO_INHERIT，则此互斥锁不支持优先级继承。

返回值

- **TX_SUCCESS**：(0X00) 成功创建互斥锁。
- **TX_MUTEX_ERROR**：(0X1C) 互斥锁指针无效。指针为 NULL 或已创建互斥锁。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。
- **TX_INHERIT_ERROR**：(0x1F) 优先级继承参数无效。

允许来自

初始化和线程

可以抢占

否

示例

```
TX_MUTEX my_mutex;
UINT status;

/* Create a mutex to provide protection over a
common resource. */
status = tx_mutex_create(&my_mutex,"my_mutex_name",
    TX_NO_INHERIT);

/* If status equals TX_SUCCESS, my_mutex is ready for
use. */
```

另请参阅

- tx_mutex_delete
- tx_mutex_get
- tx_mutex_info_get
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_prioritize
- tx_mutex_put

tx_mutex_delete

删除互斥锁

原型

```
UINT tx_mutex_delete(TX_MUTEX *mutex_ptr);
```

说明

此服务删除指定的互斥锁。所有挂起并等待互斥锁的线程都将恢复，并获得 TX_DELETED 返回状态。

> **NOTE**
> 应用程序负责阻止使用已删除的互斥锁。

参数

- **mutex_ptr**：指向之前创建的互斥锁的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功删除互斥锁。
- **TX_MUTEX_ERROR**：(0X1C) 互斥锁指针无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程数

可以抢占

是

示例

```
TX_MUTEX my_mutex;
UINT status;

/* Delete a mutex. Assume that the mutex
has already been created. */
status = tx_mutex_delete(&my_mutex);

/* If status equals TX_SUCCESS, the mutex is
deleted. */
```

另请参阅

- tx_mutex_create
- tx_mutex_get
- tx_mutex_info_get
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_prioritize
- tx_mutex_put

tx_mutex_get

获取互斥锁的所有权

原型

```
UINT tx_mutex_get(
    TX_MUTEX *mutex_ptr,
    ULONG wait_option);
```

说明

此服务尝试获取指定互斥锁的独占所有权。如果调用线程已拥有互斥锁，则内部计数器会递增，并返回成功状态。

如果互斥锁由另一个线程拥有，并且该线程具有更高的优先级，并且在创建互斥锁时指定了优先级继承，则较低优先级线程的优先级将临时提升到调用线程的优先级。

> **NOTE**
> 对于拥有互斥锁并指定了优先级继承的较低优先级线程，在拥有互斥锁期间，其优先级绝不能由外部线程修改。

参数

- **mutex_ptr**
  指向之前创建的互斥锁的指针。
- **wait_option**
  定义该互斥锁已由另一个线程拥有时此服务的行为方式。等待选项的定义如下：
  - TX_NO_WAIT (0x00000000) - 如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从初始化调用服务，则这是唯一有效的选项。
  - TX_WAIT_FOREVER 超时值 (0xFFFFFFFF) - 选择 TX_WAIT_FOREVER 会导致发出调用的线程无限期挂起，直到互斥锁可用为止 。
  - 超时值（0x00000001 至 0xFFFFFFFE）- 如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待互斥锁时发出调用的线程保持挂起的最大计时器时钟周期数。

返回值

- **TX_SUCCESS**：(0X00) 成功执行互斥锁获取操作。
- **TX_DELETED**：(0x01) 线程挂起时删除了互斥锁。
- **TX_NOT_AVAILABLE**：(0X1D) 服务无法在指定的等待时间内获得互斥锁的所有权。
- **TX_WAIT_ABORTED**：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- **TX_MUTEX_ERROR**：(0X1C) 互斥锁指针无效。
- **TX_WAIT_ERROR**：(0x04) 从非线程调用时指定了除 TX_NO_WAIT 以外的等待选项。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

初始化、线程和计时器

可以抢占

是

示例

```
TX_MUTEX my_mutex;
UINT status;

/* Obtain exclusive ownership of the mutex "my_mutex".
If the mutex "my_mutex" is not available, suspend until it
becomes available. */
status = tx_mutex_get(&my_mutex, TX_WAIT_FOREVER);
```

另请参阅

- tx_mutex_create
- tx_mutex_delete
- tx_mutex_info_get
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_prioritize
- tx_mutex_put

tx_mutex_info_get

检索有关互斥锁的信息

原型

```
UINT tx_mutex_info_get(
    TX_MUTEX *mutex_ptr,
    CHAR **name,
    ULONG *count,
    TX_THREAD **owner,
    TX_THREAD **first_suspended,
    ULONG *suspended_count,
    TX_MUTEX **next_mutex);
```

说明

此服务从指定的互斥锁中检索信息。

参数

- **mutex_ptr**：指向互斥锁控制块的指针。
- **name**：指向互斥锁名称指针这一目标的指针。
- **count**：指向互斥锁所有权计数这一目标的指针。
- **owner**：指向拥有线程的指针这一目标的指针。
- **first_suspended**：指向此互斥锁的挂起列表上第一个线程的指针这一目标的指针。
- **suspended_count**：指向此互斥锁中当前挂起的线程数这一目标的指针。
- **next_mutex**：指向下一个已创建的互斥锁的指针这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功检索互斥锁信息。
- **TX_MUTEX_ERROR**：(0X1C) 互斥锁指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_MUTEX my_mutex;
CHAR *name;
ULONG count;
TX_THREAD *owner;
TX_THREAD *first_suspended;
ULONG suspended_count;
TX_MUTEX *next_mutex;
UINT status;

/* Retrieve information about the previously created
mutex "my_mutex." */
status = tx_mutex_info_get(&my_mutex, &name,
    &count, &owner,
    &first_suspended, &suspended_count,
    &next_mutex);

/* If status equals TX_SUCCESS, the information requested is
valid. */
```

另请参阅

- tx_mutex_create
- tx_mutex_delete
- tx_mutex_get
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_prioritize
- tx_mutex_put

tx_mutex_performance_info_get

获取互斥锁性能信息

原型

```
UINT tx_mutex_performance_info_get(
    TX_MUTEX *mutex_ptr,
    ULONG *puts,
    ULONG *gets,
    ULONG *suspensions,
    ULONG *timeouts,
    ULONG *inversions,
    ULONG *inheritances);
```

说明

此服务检索所指定互斥锁的相关性能信息。

> **IMPORTANT**
> 必须使用为此服务定义的 TX_MUTEX_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。*

参数

- **mutex_ptr**：指向之前创建的互斥锁的指针。
- **puts**：指向对此互斥锁执行的放置请求数这一目标的指针。
- **gets**：指向对此互斥锁执行的获取请求数这一目标的指针。
- **suspensions**：指向此互斥锁上的线程互斥锁获取挂起数这一目标的指针。

- **timeouts**：指向此互斥锁的互斥锁获取挂起超时次数这一目标的指针。
- **inversions**：指向此互斥锁上线程优先级倒置数这一目标的指针。
- **inheritances**：指向此互斥锁上线程优先级继承操作数这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功获取互斥锁性能信息。
- **TX_PTR_ERROR**：(0x03) 互斥锁指针无效。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_MUTEX my_mutex;
ULONG puts;
ULONG gets;
ULONG suspensions;
ULONG timeouts;
ULONG inversions;
ULONG inheritances;

/* Retrieve performance information on the previously created
mutex. */
status = tx_mutex_performance_info_get(&my_mutex_ptr, &puts, &gets,
    &suspensions, &timeouts, &inversions, &inheritances);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_mutex_create
- tx_mutex_delete
- tx_mutex_get
- tx_mutex_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_prioritize
- tx_mutex_put

tx_mutex_performance_system_info_get

获取互斥锁系统性能信息

原型

```
UINT tx_mutex_performance_system_info_get(
    ULONG *puts,
    ULONG *gets,
    ULONG *suspensions,
    ULONG *timeouts,
    ULONG *inversions,
    ULONG *inheritances);
```

说明

此服务检索系统中所有互斥锁的相关性能信息。

> **IMPORTANT**
>
> 必须使用为此服务定义的 TX_MUTEX_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。

参数

- **puts**：指向对所有互斥锁执行的放置请求总数这一目标的指针。
- **gets**：指向对所有互斥锁执行的获取请求总数这一目标的指针。
- **suspensions**：指向所有互斥锁上的线程互斥锁获取挂起总数这一目标的指针。
- **timeouts**：指向所有互斥锁的互斥锁获取挂起超时总数这一目标的指针。
- **inversions**：指向所有互斥锁上线程优先级倒置总数这一目标的指针。
- **inheritances**：指向所有互斥锁上线程优先级继承操作总数这一目标的指针。

> **NOTE**
>
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功获取互斥锁系统性能信息。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
ULONG puts;
ULONG gets;
ULONG suspensions;
ULONG timeouts;
ULONG inversions;
ULONG inheritances;

/* Retrieve performance information on all previously created
mutexes. */
status = tx_mutex_performance_system_info_get(&puts, &gets,
    &suspensions, &timeouts,
    &inversions, &inheritances);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_mutex_create

- tx_mutex_delete
- tx_mutex_get
- tx_mutex_info_get
- tx_mutex_performance_info_get
- tx_mutex_prioritize
- tx_mutex_put

tx_mutex_prioritize

设置互斥锁挂起列表的优先级

原型

```
UINT tx_mutex_prioritize(TX_MUTEX *mutex_ptr);
```

说明

此服务将因互斥锁所有权而挂起的最高优先级线程放在挂起列表前面。所有其他线程保持它们挂起时的相同 FIFO 顺序。

参数

- **mutex_ptr**：指向之前创建的互斥锁的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功设置互斥锁优先级。
- **TX_MUTEX_ERROR**：(0X1C) 互斥锁指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_MUTEX my_mutex;
UINT status;

/* Ensure that the highest priority thread will receive
ownership of the mutex when it becomes available. */
status = tx_mutex_prioritize(&my_mutex);

/* If status equals TX_SUCCESS, the highest priority
suspended thread is at the front of the list. The
next tx_mutex_put call that releases ownership of the
mutex will give ownership to this thread and wake it
up. */
```

另请参阅

- tx_mutex_create
- tx_mutex_delete
- tx_mutex_get
- tx_mutex_info_get
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get

- tx_mutex_put

tx_mutex_put

释放互斥锁的所有权

原型

```
UINT tx_mutex_put(TX_MUTEX *mutex_ptr);
```

说明

此服务使指定互斥锁的所有权计数递减。如果所有权计数为零，则互斥锁可用。

> **NOTE**
>
> 如果在创建互斥锁期间选择了优先级继承，则释放线程的优先级将恢复为最初获得互斥锁所有权时的优先级。在拥有互斥锁期间对释放线程所做的任何其他优先级更改都可以撤消。

参数

- mutex_ptr：指向之前创建的互斥锁的指针。

返回值

- TX_SUCCESS：(0X00) 成功释放互斥锁。
- TX_NOT_OWNED：(0X1E) 互斥锁不归调用方所有。
- TX_MUTEX_ERROR：(0X1C) 互斥锁指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

允许来自

初始化、线程和计时器

可以抢占

是

示例

```
TX_MUTEX my_mutex;
UINT status;

/* Release ownership of "my_mutex." */
status = tx_mutex_put(&my_mutex);

/* If status equals TX_SUCCESS, the mutex ownership
count has been decremented and if zero, released. */
```

另请参阅

- tx_mutex_create
- tx_mutex_delete
- tx_mutex_get
- tx_mutex_info_get
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_prioritize

tx_queue_create

创建消息队列

原型

```
UINT tx_queue_create(
    TX_QUEUE *queue_ptr,
    CHAR *name_ptr,
    UINT message_size,
    VOID *queue_start,
    ULONG queue_size);
```

说明

该服务创建通常用于线程间通信的消息队列。消息总数是根据指定的消息大小和队列中的字节总数来计算的。

> **NOTE**
> 如果队列内存区域中指定的总字节数不能被指定的消息大小整除，则不会使用内存区域中剩余的字节。

参数

- **queue_ptr**：指向消息队列控制块的指针。
- **name_ptr**：指向消息队列名称的指针。
- **message_size**：指定队列中每条消息的大小。消息大小从 1 个 32 位字到 16 个 32 位字不等。有效的消息大小选项是从 1 到 16（含）的数值。
- **queue_start**：消息队列的起始地址。起始地址必须与 ULONG 数据类型的大小一致。
- **queue_size**：可用于消息队列的总字节数。

返回值

- **TX_SUCCESS**：(0X00) 成功创建消息队列。
- **TX_QUEUE_ERROR**：(0X09) 消息队列指针无效。指针为 NULL 或已创建队列。
- **TX_PTR_ERROR**：(0x03) 消息队列的起始地址无效。
- **TX_SIZE_ERROR**：(0x05) 消息队列大小无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

初始化和线程

可以抢占

否

示例

```
TX_QUEUE my_queue;
UINT status;

/* Create a message queue whose total size is 2000 bytes
starting at address 0x300000. Each message in this
queue is defined to be 4 32-bit words long. */
status = tx_queue_create(&my_queue, "my_queue_name",
    4, (VOID *) 0x300000, 2000);

/* If status equals TX_SUCCESS, my_queue contains room
for storing 125 messages (2000 bytes/ 16 bytes per
message). */
```

另请参阅

- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_delete

删除消息队列

原型

```
UINT tx_queue_delete(TX_QUEUE *queue_ptr);
```

说明

此服务删除指定的消息队列。所有挂起并等待来自此队列的消息的线程都将恢复，并获得 TX_DELETED 返回状态。

> **IMPORTANT**
>
> 在删除队列之前，应用程序必须确保完成（或禁用）此队列的任何发送通知回调。此外，应用程序必须阻止将来使用已删除的队列。
>
> 应用程序还负责管理与队列关联的内存区域，该内存区域在此服务完成后可用。

参数

- **queue_ptr**：指向以前创建的消息队列的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功删除消息队列。
- **TX_QUEUE_ERROR**：(0X09) 消息队列指针无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程数

可以抢占

是

示例

```
TX_QUEUE my_queue;
UINT status;

/* Delete entire message queue. Assume that the queue
has already been created with a call to
tx_queue_create. */
status = tx_queue_delete(&my_queue);

/* If status equals TX_SUCCESS, the message queue is
deleted. */
```

另请参阅

- tx_queue_create
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_flush

清空消息队列中的消息

原型

```
UINT tx_queue_flush(TX_QUEUE *queue_ptr);
```

说明

此服务删除存储在指定消息队列中的所有消息。

如果队列已满，则丢弃所有挂起线程的消息。然后，每个挂起的线程都将恢复，并具有一个返回状态，指示消息发送成功。如果队列为空，则此服务不执行任何操作。

参数

- **queue_ptr**：指向以前创建的消息队列的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功刷新消息队列。
- **TX_QUEUE_ERROR**：(0X09) 消息队列指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_QUEUE my_queue;
UINT status;

/* Flush out all pending messages in the specified message
queue. Assume that the queue has already been created
with a call to tx_queue_create. */
status = tx_queue_flush(&my_queue);

/* If status equals TX_SUCCESS, the message queue is
empty. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_front_send

将消息发送到队列的前面

原型

```
UINT tx_queue_front_send(
    TX_QUEUE *queue_ptr,
    VOID *source_ptr,
    ULONG wait_option);
```

说明

此服务将消息发送到指定消息队列的前端位置。将消息从源指针指定的内存区域复制到队列的前面。

参数

- **queue_ptr**
  指向消息队列控制块的指针。
- **source_ptr**
  指向消息的指针。
- **wait_option**
  定义在消息队列已满时服务的行为方式。等待选项的定义如下：
  - TX_NO_WAIT (0x00000000) - 如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从非线程（例如初始化、计时器或 ISR）调用服务，则这是唯一有效的选项。
  - TX_WAIT_FOREVER (0xFFFFFFFF) - 选择 TX_WAIT_FOREVER 会导致发出调用的线程无限期挂起，直到队列中有空间为止。
  - 超时值（0x00000001 至 0xFFFFFFFE）- 如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待队列空间时发出调用的线程保持挂起的最大计时器时钟周期数。

返回值

- **TX_SUCCESS**：(0X00) 成功发送消息。

- **TX_DELETED**：(0x01) 线程挂起时删除了消息队列。
- **TX_QUEUE_FULL**：(0x0B) 服务无法发送消息，因为在指定的等待时间内队列已满。
- **TX_WAIT_ABORTED**：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- **TX_QUEUE_ERROR**：(0X09) 消息队列指针无效。
- **TX_PTR_ERROR**：(0x03) 消息的源指针无效。
- **TX_WAIT_ERROR**：(0x04) 从非线程调用时指定了除 TX_NO_WAIT 以外的等待选项。

允许来自

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_QUEUE my_queue;
UINT status;
ULONG my_message[4];

/* Send a message to the front of "my_queue." Return
immediately, regardless of success. This wait
option is used for calls from initialization, timers,
and ISRs. */
status = tx_queue_front_send(&my_queue, my_message,
    TX_NO_WAIT);

/* If status equals TX_SUCCESS, the message is at the front
of the specified queue. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_info_get

检索有关队列的信息

原型

```
UINT tx_queue_info_get(
    TX_QUEUE *queue_ptr,
    CHAR **name,
    ULONG *enqueued,
    ULONG *available_storage
    TX_THREAD **first_suspended,
    ULONG *suspended_count,
    TX_QUEUE **next_queue);
```

说明

此服务检索所指定消息队列的相关信息。

参数

- **queue_ptr**：指向以前创建的消息队列的指针。
- **name**：指向队列名称指针这一目标的指针。
- **enqueued**：指向队列中当前的消息数这一目标的指针。
- **available_storage**：指向队列当前可容纳的消息数这一目标的指针。
- **first_suspended**：指向此队列的挂起列表上第一个线程的指针这一目标的指针。
- **suspended_count**：指向此队列中当前挂起的线程数这一目标的指针。
- **next_queue**：指向下一个创建的队列的指针这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功获取队列信息。
- **TX_QUEUE_ERROR**：(0X09) 消息队列指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_QUEUE my_queue;
CHAR *name;
ULONG enqueued;
ULONG available_storage;
TX_THREAD *first_suspended;
ULONG suspended_count;
TX_QUEUE *next_queue;
UINT status;

/* Retrieve information about the previously created
message queue "my_queue." */
status = tx_queue_info_get(&my_queue, &name,
    &enqueued, &available_storage,
    &first_suspended, &suspended_count,
    &next_queue);

/* If status equals TX_SUCCESS, the information requested is
valid. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize

- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_performance_info_get

获取队列性能信息

原型

```
UINT tx_queue_performance_info_get(
    TX_QUEUE *queue_ptr,
    ULONG *messages_sent,
    ULONG *messages_received,
    ULONG *empty_suspensions,
    ULONG *full_suspensions,
    ULONG *full_errors,
    ULONG *timeouts);
```

说明

此服务检索所指定消息队列的相关性能信息。

> **IMPORTANT**
> 必须使用为此服务定义的 TX_QUEUE_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。*

参数

- **queue_ptr**：指向之前创建的队列的指针。
- **messages_sent**：指向对此队列执行的发送请求数这一目标的指针。
- **messages_received**：指向对此队列执行的接收请求数这一目标的指针。
- **empty_suspensions**：指向此队列上队列为空的挂起数这一目标的指针。
- **full_suspensions**：指向此队列上队列已满的挂起数这一目标的指针。
- **full_errors**：指向此队列上队列已满的错误数这一目标的指针。
- **timeouts**：指向此队列的线程挂起超时次数这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功获取队列性能信息。
- **TX_PTR_ERROR**：(0x03) 队列指针无效。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_QUEUE my_queue;
ULONG messages_sent;
ULONG messages_received;
ULONG empty_suspensions;
ULONG full_suspensions;
ULONG full_errors;
ULONG timeouts;

/* Retrieve performance information on the previously created
queue. */
status = tx_queue_performance_info_get(&my_queue, &messages_sent,
    &messages_received, &empty_suspensions,
    &full_suspensions, &full_errors, &timeouts);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_performance_system_info_get

获取队列系统性能信息

原型

```
UINT tx_queue_performance_system_info_get(
    ULONG *messages_sent,
    ULONG *messages_received,
    ULONG *empty_suspensions,
    ULONG *full_suspensions,
    ULONG *full_errors,
    ULONG *timeouts);
```

说明

此服务检索系统中所有队列的相关性能信息。

> **IMPORTANT**
> 必须使用为此服务定义的 TX_QUEUE_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。*

参数

- **messages_sent**：指向对所有队列执行的发送请求总数这一目标的指针。
- **messages_received**：指向对所有队列执行的接收请求总数这一目标的指针。
- **empty_suspensions**：指向所有队列上队列为空的挂起总数这一目标的指针。

- **full_suspensions**：指向所有队列上队列已满的挂起总数这一目标的指针。
- **full_errors**：指向所有队列上队列已满的错误总数这一目标的指针。
- **timeouts**：指向所有队列的线程挂起超时总数这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功获取队列系统性能信息。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
ULONG messages_sent;
ULONG messages_received;
ULONG empty_suspensions;
ULONG full_suspensions;
ULONG full_errors;
ULONG timeouts;

/* Retrieve performance information on all previously created
queues. */
status = tx_queue_performance_system_info_get(&messages_sent,
    &messages_received, &empty_suspensions,
    &full_suspensions, &full_errors, &timeouts);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_prioritize

设置队列挂起列表的优先级

原型

```
UINT tx_queue_prioritize(TX_QUEUE *queue_ptr);
```

说明

此服务将因此队列中的消息而挂起的最高优先级线程(或将某个消息)放在挂起列表前面。

所有其他线程保持它们挂起时的相同 FIFO 顺序。

参数

- **queue_ptr**：指向以前创建的消息队列的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功设置队列优先级。
- **TX_QUEUE_ERROR**：(0X09) 消息队列指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_QUEUE my_queue;
UINT status;

/* Ensure that the highest priority thread will receive
the next message placed on this queue. */
status = tx_queue_prioritize(&my_queue);

/* If status equals TX_SUCCESS, the highest priority
suspended thread is at the front of the list. The
next tx_queue_send or tx_queue_front_send call made
to this queue will wake up this thread. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_receive

从消息队列获取消息

原型

```
UINT tx_queue_receive(
    TX_QUEUE *queue_ptr,
    VOID *destination_ptr,
    ULONG wait_option);
```

说明

此服务从指定的消息队列检索消息。将检索到的消息从队列复制到目标指针指定的内存区域中。然后从队列中删除该消息。

> **IMPORTANT**
> 指定的目标内存区域必须足够大以容纳消息；也就是说，由 destination_ptr 指向的消息目标内存区域必须至少与此队列的消息大小一样大。* 否则，如果目标内存区域不够大，则会在下一内存区域中发生内存损坏。

参数

- **queue_ptr**
  指向以前创建的消息队列的指针。
- **destination_ptr**
  复制消息的位置。
- **wait_option**
  定义在消息队列为空时服务的行为方式。等待选项的定义如下：
  - TX_NO_WAIT (0x00000000) - 如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从非线程（例如初始化、计时器或 ISR）调用服务，则这是唯一有效的选项。
  - TX_WAIT_FOREVER (0xFFFFFFFF) - 选择 TX_WAIT_FOREVER 会导致发出调用的线程无限期挂起，直到消息可用为止。
  - 超时值（0x00000001 至 0xFFFFFFFE）- 如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待消息时发出调用的线程保持挂起的最大计时器时钟周期数。

返回值

- **TX_SUCCESS**：(0X00) 成功检索消息。
- **TX_DELETED**：(0x01) 线程挂起时删除了消息队列。
- **TX_QUEUE_EMPTY**：(0x0A) 服务无法检索消息，因为队列在指定的等待时间内为空。
- **TX_WAIT_ABORTED**：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- **TX_QUEUE_ERROR**：(0X09) 消息队列指针无效。
- **TX_PTR_ERROR**：(0x03) 消息的目标指针无效。
- **TX_WAIT_ERROR**：(0x04) 从非线程调用时指定了除 TX_NO_WAIT 以外的等待选项。

允许来自

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_QUEUE my_queue;
UINT status;
ULONG my_message[4];

/* Retrieve a message from "my_queue." If the queue is
empty, suspend until a message is present. Note that
this suspension is only possible from application
threads. */
status = tx_queue_receive(&my_queue, my_message,
    TX_WAIT_FOREVER);

/* If status equals TX_SUCCESS, the message is in
"my_message." */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_send
- tx_queue_send_notify

tx_queue_send

将消息发送到消息队列

原型

```
UINT tx_queue_send(
    TX_QUEUE *queue_ptr,
    VOID *source_ptr,
    ULONG wait_option);
```

说明

此服务将消息发送到指定消息队列。将发送的消息从源指针指定的内存区域复制到队列。

参数

- **queue_ptr**
  指向以前创建的消息队列的指针。
- **source_ptr**
  指向消息的指针。
- **wait_option**
  定义在消息队列已满时服务的行为方式。等待选项的定义如下：
  - TX_NO_WAIT (0x00000000) - 如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从非线程（例如初始化、计时器或 ISR）调用服务，则这是唯一有效的选项。
  - TX_WAIT_FOREVER (0xFFFFFFFF) - 选择 TX_WAIT_FOREVER 会导致发出调用的线程无限期挂起，直到队列中有空间为止。
  - 超时值（0x00000001 至 0xFFFFFFFE）- 如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待队列空间时发出调用的线程保持挂起的最大计时器时钟周期数。

返回值

- **TX_SUCCESS**：(0X00) 成功发送消息。
- **TX_DELETED**：(0x01) 线程挂起时删除了消息队列。
- **TX_QUEUE_FULL**：(0x0B) 服务无法发送消息，因为在指定的等待时间内队列已满。
- **TX_WAIT_ABORTED**：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- **TX_QUEUE_ERROR**：(0X09) 消息队列指针无效。
- **TX_PTR_ERROR**：(0x03) 消息的源指针无效。
- **TX_WAIT_ERROR**：(0x04) 从非线程调用时指定了除 TX_NO_WAIT 以外的等待选项。

允许来自

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_QUEUE my_queue;
UINT status;
ULONG my_message[4];

/* Send a message to "my_queue." Return immediately,
regardless of success. This wait option is used for
calls from initialization, timers, and ISRs. */
status = tx_queue_send(&my_queue, my_message, TX_NO_WAIT);

/* If status equals TX_SUCCESS, the message is in the
queue. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send_notify

tx_queue_send_notify

在将消息发送到队列时通知应用程序

原型

```
UINT tx_queue_send_notify(
    TX_QUEUE *queue_ptr,
    VOID (*queue_send_notify)(TX_QUEUE *));
```

说明

此服务注册一个通知回调函数，每当消息发送到指定队列时，就会调用该函数。通知回调的处理由应用程序定义。

> **NOTE**
> 不允许应用程序的队列发送通知回调调用任何带有挂起选项的 ThreadX API。

参数

- **queue_ptr**：指向之前创建的队列的指针。
- **queue_send_notify**：指向应用程序的队列发送通知函数的指针。如果此值为 TX_NULL，则禁用通知。

返回值

- **TX_SUCCESS**：(0X00) 成功注册队列发送通知。
- **TX_QUEUE_ERROR**：(0X09) 队列指针无效。

- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时禁用了通知功能。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_QUEUE my_queue;
/* Register the "my_queue_send_notify" function for monitoring
messages sent to the queue "my_queue." */
status = tx_queue_send_notify(&my_queue, my_queue_send_notify);

/* If status is TX_SUCCESS the queue send notification function was
successfully registered. */
void my_queue_send_notify(TX_QUEUE *queue_ptr)
{
    /* A message was just sent to this queue! */
}
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send

tx_semaphore_ceiling_put

将实例放入具有上限的计数信号灯

原型

```
UINT tx_semaphore_ceiling_put(
    TX_SEMAPHORE *semaphore_ptr,
    ULONG ceiling);
```

说明

此服务将一个实例放入指定的计数信号灯，实际上会将计数信号灯增加 1。如果计数信号灯的当前值大于或等于指定的上限，则不会放入该实例，并将返回 TX_CEILING_EXCEEDED 错误。

参数

- **semaphore_ptr**：指向之前创建的信号灯的指针。
- **ceiling**：允许的信号灯上限（有效值范围为 1 至 0xFFFFFFFF）。

返回值

- **TX_SUCCESS**：(0X00) 成功设置信号灯上限。
- **TX_CEILING_EXCEEDED**：(0x21) 放置请求超过上限。
- **TX_INVALID_CEILING**：(0x22) 为上限提供的值 (0) 无效。
- **TX_SEMAPHORE_ERROR**：(0x0C) 信号灯指针无效。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_SEMAPHORE my_semaphore;

/* Increment the counting semaphore "my_semaphore" but make sure
that it never exceeds 7 as specified in the call. */
status = tx_semaphore_ceiling_put(&my_semaphore, 7);

/* If status is TX_SUCCESS the semaphore count has been
incremented. */
```

另请参阅

- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_get
- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_create

创建计数信号灯

原型

```
UINT tx_semaphore_create(
    TX_SEMAPHORE *semaphore_ptr,
    CHAR *name_ptr,
    ULONG initial_count);
```

说明

此服务为线程间同步创建计数信号灯。初始信号灯计数指定为输入参数。

参数

- **semaphore_ptr**：指向信号灯控制块的指针。
- **name_ptr**：指向信号灯名称的指针。
- **initial_count**：指定此信号灯的初始计数。合法值的范围为 0x00000000 至 0xFFFFFFFF。

返回值

- **TX_SUCCESS**：(0X00) 成功创建信号灯。
- **TX_SEMAPHORE_ERROR**：(0x0C) 信号灯指针无效。指针为 NULL 或已创建信号灯。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

初始化和线程

可以抢占

否

示例

```
TX_SEMAPHORE my_semaphore;
UINT status;

/* Create a counting semaphore whose initial value is 1.
This is typically the technique used to make a binary
semaphore. Binary semaphores are used to provide
protection over a common resource. */
status = tx_semaphore_create(&my_semaphore,
    "my_semaphore_name", 1);

/* If status equals TX_SUCCESS, my_semaphore is ready for
use. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_delete
- tx_semaphore_get
- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_delete

删除计数信号灯

原型

```
UINT tx_semaphore_delete(TX_SEMAPHORE *semaphore_ptr);
```

说明

此服务删除指定的计数信号灯。所有挂起并等待信号灯实例的线程都将恢复，并获得 TX_DELETED 返回状态。

> **IMPORTANT**
> 在删除此信号灯之前，应用程序必须确保完成（或禁用）此信号灯的放置通知回调。此外，应用程序必须阻止将来使用已删除的信号灯。

参数

- **semaphore_ptr**：指向之前创建的信号灯的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功删除计数信号灯。
- **TX_SEMAPHORE_ERROR**：(0x0C) 计数信号灯指针无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程数

可以抢占

是

示例

```
TX_SEMAPHORE my_semaphore;
UINT status;

/* Delete counting semaphore. Assume that the counting
semaphore has already been created. */
status = tx_semaphore_delete(&my_semaphore);

/* If status equals TX_SUCCESS, the counting semaphore is
deleted. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semaphore_get
- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_get

从计数信号灯获取实例

原型

```
UINT tx_semaphore_get(
    TX_SEMAPHORE *semaphore_ptr,
    ULONG wait_option);
```

说明

此服务从指定的计数信号灯检索实例(单个计数)。因此，指定信号灯的计数将减少 1。

参数

- **semaphore_ptr**
  指向之前创建的计数信号灯的指针。
- **wait_option**
  定义在没有可用信号灯实例(即信号灯计数为零)的情况下服务的行为方式。等待选项的定义如下：
  - TX_NO_WAIT (0x00000000) - 如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从非线程(例如初始化、计时器或 ISR)调用服务，则这是唯一有效的选项。
  - TX_WAIT_FOREVER (0xFFFFFFFF) - 选择 TX_WAIT_FOREVER 会导致发出调用的线程无限期挂起，直到信号灯实例可用为止。
  - 超时值(0x00000001 至 0xFFFFFFFE)- 如果选择一个数值(1 到 0xFFFFFFFE)，则会指定在等待信号灯实例时发出调用的线程保持挂起的最大计时器时钟周期数。

返回值

- **TX_SUCCESS**：(0X00) 成功检索信号灯实例。
- **TX_DELETED**：(0x01) 线程挂起时删除了计数信号灯。

- **TX_NO_INSTANCE**：(0X0D) 服务无法检索计数信号灯的实例（信号灯计数在指定的等待时间内为零）。
- **TX_WAIT_ABORTED**：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- **TX_SEMAPHORE_ERROR**：(0x0C) 计数信号灯指针无效。
- **TX_WAIT_ERROR**：(0x04) 从非线程调用时指定了除 TX_NO_WAIT 以外的等待选项。

允许来自

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_SEMAPHORE my_semaphore;
UINT status;

/* Get a semaphore instance from the semaphore
"my_semaphore." If the semaphore count is zero,
suspend until an instance becomes available.
Note that this suspension is only possible from
application threads. */
status = tx_semaphore_get(&my_semaphore, TX_WAIT_FOREVER);

/* If status equals TX_SUCCESS, the thread has obtained
an instance of the semaphore. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semahore_delete
- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_info_get

检索有关信号灯的信息

原型

```
UINT tx_semaphore_info_get(
    TX_SEMAPHORE *semaphore_ptr,
    CHAR **name, ULONG *current_value,
    TX_THREAD **first_suspended,
    ULONG *suspended_count,
    TX_SEMAPHORE **next_semaphore);
```

说明

此服务检索所指定信号灯的相关信息。

参数

- **semaphore_ptr**：指向信号灯控制块的指针。
- **name**：指向信号灯名称指针这一目标的指针。

- current_value：指向当前信号灯的计数这一目标的指针。
- first_suspended：指向此信号灯的挂起列表上第一个线程的指针这一目标的指针。
- suspended_count：指向此信号灯中当前挂起的线程数这一目标的指针。
- next_semaphore：指向下一个创建的信号灯的指针这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0X00) 检索信息。
- TX_SEMAPHORE_ERROR：(0x0C) 信号灯指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_SEMAPHORE my_semaphore;
CHAR *name;
ULONG current_value;
TX_THREAD *first_suspended;
ULONG suspended_count;
TX_SEMAPHORE *next_semaphore;
UINT status;

/* Retrieve information about the previously created
semaphore "my_semaphore." */
status = tx_semaphore_info_get(&my_semaphore, &name,
    &current_value,
    &first_suspended, &suspended_count,
    &next_semaphore);

/* If status equals TX_SUCCESS, the information requested is
valid. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_performance_info_get

获取信号灯性能信息

原型

```
UINT tx_semaphore_performance_info_get(
    TX_SEMAPHORE *semaphore_ptr,
    ULONG *puts,
    ULONG *gets,
    ULONG *suspensions,
    ULONG *timeouts);
```

说明

此服务检索所指定信号灯的相关性能信息。

> **IMPORTANT**
> 必须使用为此服务定义的 TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。*

参数

- **semaphore_ptr**：指向之前创建的信号灯的指针。
- **puts**：指向对此信号灯执行的放置请求数这一目标的指针。
- **gets**：指向对此信号灯执行的获取请求数这一目标的指针。
- **suspensions**：指向此信号灯上的线程挂起数这一目标的指针。
- **timeouts**：指向此信号灯的线程挂起超时次数这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功获取信号灯性能信息。
- **TX_PTR_ERROR**：(0X03) 信号灯指针无效。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_SEMAPHORE my_semaphore;
ULONG puts;
ULONG gets;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on the previously created
semaphore. */
status = tx_semaphore_performance_info_get(&my_semaphore, &puts,
    &gets, &suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create

- tx_semaphore_delete
- tx_semaphore_get
- tx_semaphore_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_performance_system_info_get

获取信号灯系统性能信息

原型

```
UINT tx_semaphore_performance_system_info_get(
    ULONG *puts,
    ULONG *gets,
    ULONG *suspensions,
    ULONG *timeouts);
```

说明

此服务检索系统中所有信号灯的相关性能信息。

> **IMPORTANT**
> 必须使用为此服务定义的 TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。*

参数

- **puts**：指向对所有信号灯执行的放置请求总数这一目标的指针。
- **gets**：指向对所有信号灯执行的获取请求总数这一目标的指针。
- **suspensions**：指向所有信号灯上的线程挂起总数这一目标的指针。
- **timeouts**：指向所有信号灯的线程挂起超时总数这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 获取系统性能信息。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
ULONG puts;
ULONG gets;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on all previously created
semaphores. */
status = tx_semaphore_performance_system_info_get(&puts, &gets,
    &suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_get
- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_prioritize

设置信号灯挂起列表的优先级

原型

```
UINT tx_semaphore_prioritize(TX_SEMAPHORE *semaphore_ptr);
```

说明

此服务将因信号灯实例而挂起的最高优先级线程放在挂起列表前面。所有其他线程保持它们挂起时的相同 FIFO 顺序。

参数

- **semaphore_ptr**：指向之前创建的信号灯的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功设置信号灯优先级。
- **TX_SEMAPHORE_ERROR**：(0x0C) 计数信号灯指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_SEMAPHORE my_semaphore;
UINT status;

/* Ensure that the highest priority thread will receive
the next instance of this semaphore. */
status = tx_semaphore_prioritize(&my_semaphore);

/* If status equals TX_SUCCESS, the highest priority
suspended thread is at the front of the list. The
next tx_semaphore_put call made to this semaphore will
wake up this thread. */
```

另请参阅

- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_get
- tx_semaphore_info_get
- tx_semaphore_put

tx_semaphore_put

将实例放入计数信号灯

原型

```
UINT tx_semaphore_put(TX_SEMAPHORE *semaphore_ptr);
```

说明

此服务将一个实例放入指定的计数信号灯，实际上会将计数信号灯增加 1。

> **NOTE**
> 如果在信号灯全部为一 (OxFFFFFFFF) 时调用此服务，则新的放置操作将导致信号灯重置为零。

参数

- **semaphore_ptr**：指向之前创建的计数信号灯控制块的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功放置信号灯。
- **TX_SEMAPHORE_ERROR**：(0x0C) 计数信号灯指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_SEMAPHORE my_semaphore;
UINT status;

/* Increment the counting semaphore "my_semaphore." */
status = tx_semaphore_put(&my_semaphore);

/* If status equals TX_SUCCESS, the semaphore count has
been incremented. Of course, if a thread was waiting,
it was given the semaphore instance and resumed. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_get
- tx_semaphore_put_notify

tx_semaphore_put_notify

放置信号灯时通知应用程序

原型

```
UINT tx_semaphore_put_notify(
    TX_SEMAPHORE *semaphore_ptr,
    VOID (*semaphore_put_notify)(TX_SEMAPHORE *));
```

说明

此服务注册一个通知回调函数，每当放置指定信号灯时，就会调用该函数。通知回调的处理由应用程序定义。

> **NOTE**
> 不允许应用程序的信号灯通知回调调用任何带有挂起选项的 ThreadX API。

参数

- **semaphore_ptr**：指向之前创建的信号灯的指针。
- **semaphore_put_notify**：指向应用程序的信号灯放置通知函数的指针。如果此值为 TX_NULL，则禁用通知。

返回值

- **TX_SUCCESS**：(0X00) 成功注册信号灯放置通知。
- **TX_SEMAPHORE_ERROR**：(0x0C) 信号灯指针无效。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时禁用了通知功能。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_SEMAPHORE my_semaphore;

/* Register the "my_semaphore_put_notify" function for monitoring
the put operations on the semaphore "my_semaphore." */
status = tx_semaphore_put_notify(&my_semaphore, my_semaphore_put_notify);

/* If status is TX_SUCCESS the semaphore put notification function
was successfully registered. */
void my_semaphore_put_notify(TX_SEMAPHORE *semaphore_ptr)
{
    /* The semaphore was just put! */
}
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_get
- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put

tx_thread_create

创建应用程序线程

原型

```
UINT tx_thread_create(
    TX_THREAD *thread_ptr,
    CHAR *name_ptr,
    VOID (*entry_function)(ULONG),
    ULONG entry_input,
    VOID *stack_start,
    ULONG stack_size,
    UINT priority,
    UINT preempt_threshold,
    ULONG time_slice,
    UINT auto_start);
```

说明

此服务创建一个应用程序线程，该线程在指定的任务入口函数处开始执行。堆栈、优先级、抢占阈值和时间片都是由输入参数指定的属性。此外，还指定了线程的初始执行状态。

参数

- **thread_ptr**：指向线程控制块的指针。
- **name_ptr**：指向线程名称的指针。
- **entry_function**：指定线程执行的初始 C 函数。当线程从此入口函数返回时，它将处于完成状态并无限期挂起。
- **entry_input**：首次执行时传递给线程的入口函数的 32 位值。该输入的用途完全由应用程序决定。
- **stack_start**：堆栈的内存区域的起始地址。

- **stack_size**：堆栈内存区域中的字节数。线程的堆栈区域必须足够大，以应对最坏情况下的函数调用嵌套和局部变量使用情况。
- **priority**：线程的优先级数值。合法值的范围为 0 至 (TX_MAX_PRIORITES-1)，其中 0 表示最高优先级。
- **preempt_threshold**：禁用抢占的最高优先级（0 至 (TX_MAX_PRIORITIES-1)）。只有高于此级别的优先级才允许抢占此线程。该值必须小于或等于指定的优先级。如果某值等于线程优先级，则将禁用抢占阈值。
- **time_slice**：在优先级相同的其他就绪线程有机会运行之前，允许该线程运行的计时器时钟周期数。请注意，使用抢占阈值将禁用时间片。合法时间片值的范围为 1 至 0xFFFFFFFF（含）。值为 TX_NO_TIME_SLICE（值为 0）将禁用此线程的时间片。

  > **NOTE**
  > 使用时间片会产生少量的系统开销。由于时间片仅适用于多个线程具有相同优先级的情况，因此不应为具有唯一优先级的线程分配时间片。

- **auto_start**：指定线程是立即启动还是置于挂起状态。合法选项为 TX_AUTO_START (0x01) 和 TX_DONT_START (0x00) 。如果指定了 TX_DONT_START，应用程序随后必须调用 tx_thread_resume 以便线程运行。

返回值

- **TX_SUCCESS**：(0X00) 成功创建线程。
- **TX_THREAD_ERROR**：(0x0E) 线程控制指针无效。指针为 NULL 或已创建线程。
- **TX_PTR_ERROR**：(0x03) 入口点的起始地址无效或堆栈区域无效（通常为 NULL）。
- **TX_SIZE_ERROR**：(0x05) 堆栈区域大小无效。线程必须至少有 TX_MINIMUM_STACK 个字节才能执行。
- **TX_PRIORITY_ERROR**：(0x0F) 线程优先级无效，该值超出范围（0 至 (TX_MAX_PRIORITIES-1)）。
- **TX_THRESH_ERROR**：(0x18) 指定的抢占阈值无效。此值必须是小于或等于线程初始优先级的有效优先级。
- **TX_START_ERROR**：(0x10) 自动启动选择无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

初始化和线程

可以抢占

是

示例

```
TX_THREAD my_thread;
UINT status;

/* Create a thread of priority 15 whose entry point is
"my_thread_entry". This thread’s stack area is 1000
bytes in size, starting at address 0x400000. The
preemption-threshold is setup to allow preemption of threads
with priorities ranging from 0 through 14. Time-slicing is
disabled. This thread is automatically put into a ready
condition. */
status = tx_thread_create(&my_thread, "my_thread_name",
    my_thread_entry, 0x1234,
    (VOID *) 0x400000, 1000,
    15, 15, TX_NO_TIME_SLICE,
    TX_AUTO_START);

/* If status equals TX_SUCCESS, my_thread is ready
for execution! */

...

/* Thread’s entry function. When "my_thread" actually
begins execution, control is transferred to this
function. */

VOID my_thread_entry (ULONG initial_input)
{
    /* When we get here, the value of initial_input is
    0x1234. See how this was specified during
    creation. */
    /* The real work of the thread, including calls to
    other function should be called from here! */
    /* When this function returns, the corresponding
    thread is placed into a "completed" state. */
}
```

另请参阅

- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_delete

删除应用程序线程

原型

```
UINT tx_thread_delete(TX_THREAD *thread_ptr);
```

说明

此服务删除指定的应用程序线程。由于指定的线程必须处于已终止或已完成状态，因此不能从尝试删除自身的线程中调用此服务。

> **NOTE**
> 应用程序负责管理与线程堆栈关联的内存区域，该内存区域在此服务完成后可用。此外，应用程序必须阻止使用已删除的线程。

参数

- **thread_ptr**：指向以前创建的应用程序线程的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功删除线程。
- **TX_THREAD_ERROR**：(0X0E) 应用程序线程指针无效。
- **TX_DELETE_ERROR**：(0X11) 指定线程未处于已终止或已完成状态。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程和计时器

可以抢占

否

示例

```
TX_THREAD my_thread;
UINT status;

/* Delete an application thread whose control block is
"my_thread". Assume that the thread has already been
created with a call to tx_thread_create. */
status = tx_thread_delete(&my_thread);

/* If status equals TX_SUCCESS, the application thread is
deleted. */
```

另请参阅

- tx_thread_create
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish

- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_entry_exit_notify

在线程进入和退出时通知应用程序

原型

```
UINT tx_thread_entry_exit_notify(
    TX_THREAD *thread_ptr,
    VOID (*entry_exit_notify)(TX_THREAD *, UINT));
```

说明

此服务注册一个通知回调函数，每当指定的线程进入或退出时，就会调用该函数。通知回调的处理由应用程序定义。

> **NOTE**
> 不允许应用程序的线程进入/退出通知回调调用任何带有挂起选项的 ThreadX API。

参数

- thread_ptr：指向之前创建的线程的指针。
- **entry_exit_notify**：指向应用程序的线程进入/退出通知函数的指针。进入/退出通知函数的第二个参数指定是否存在进入或退出情况。值 TX_THREAD_ENTRY (0x00) 表示线程已进入，而值 TX_THREAD_EXIT (0x01) 表示线程已退出 。如果此值为 TX_NULL，则禁用通知。

返回值

- **TX_SUCCESS**：(0X00) 成功注册线程进入/退出通知函数。
- **TX_THREAD_ERROR** (0X0E) 线程指针无效。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时禁用了通知功能。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_THREAD my_thread;
/* Register the "my_entry_exit_notify" function for monitoring
the entry/exit of the thread "my_thread." */
status = tx_thread_entry_exit_notify(&my_thread,
    my_entry_exit_notify);

/* If status is TX_SUCCESS the entry/exit notification function was
successfully registered. */
void my_entry_exit_notify(TX_THREAD *thread_ptr, UINT condition)
{
    /* Determine if the thread was entered or exited. */
    if (condition == TX_THREAD_ENTRY)
        /* Thread entry! */
    else if (condition == TX_THREAD_EXIT)
        /* Thread exit! */
}
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_identify

检索指向当前正在执行的线程的指针

原型

```
TX_THREAD* tx_thread_identify(VOID);
```

说明

此服务将返回指向当前正在执行的线程的指针。如果没有正在执行的线程，则此服务返回空指针。

> **NOTE**
>
> 如果从 ISR 调用此服务，则返回值表示在执行中断处理程序之前运行的线程。

参数

无

返回值

- **thread pointer**：指向当前正在执行的线程的指针。如果没有正在执行的线程，则返回值为 TX_NULL。

允许来自

线程和 ISR

可以抢占

否

示例

TX_THREAD *my_thread_ptr;

```
TX_THREAD *my_thread_ptr;

/* Find out who we are! */
my_thread_ptr = tx_thread_identify();

/* If my_thread_ptr is non-null, we are currently executing
from that thread or an ISR that interrupted that thread.
Otherwise, this service was called
from an ISR when no thread was running when the
interrupt occurred. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_info_get

检索有关线程的信息

原型

```
UINT tx_thread_info_get(
    TX_THREAD *thread_ptr,
    CHAR **name,
    UINT *state,
    ULONG *run_count,
    UINT *priority,
    UINT *preemption_threshold,
    ULONG *time_slice,
    TX_THREAD **next_thread,
    TX_THREAD **suspended_thread);
```

说明

此服务检索有关指定线程的信息。

参数

- thread_ptr：指向线程控制块的指针。
- name：指向线程名称指针这一目标的指针。
- state：指向线程的当前执行状态这一目标的指针。可能的值如下所示。
  - TX_READY (0x00)
  - TX_COMPLETED (0x01)
  - TX_TERMINATED (0x02)
  - TX_SUSPENDED (0x03)
  - TX_SLEEP (0x04)
  - TX_QUEUE_SUSP (0x05)
  - TX_SEMAPHORE_SUSP (0x06)
  - TX_EVENT_FLAG (0x07)
  - TX_BLOCK_MEMORY (0x08)
  - TX_BYTE_MEMORY (0x09)
  - TX_MUTEX_SUSP (0x0D)
- run_count：指向线程的运行计数这一目标的指针。
- priority：指向线程优先级这一目标的指针。
- preemption_threshold：指向线程的抢占阈值这一目标的指针。time_slice：指向线程的时间片这一目标的指针。next_thread：指向下一个创建的线程指针这一目标的指针。

suspended_thread：指向挂起列表中的下一个线程的指针这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0X00) 成功检索线程信息。
- TX_THREAD_ERROR：(0x0E) 线程控制指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_THREAD my_thread;
CHAR *name;
UINT state;
ULONG run_count;
UINT priority;
UINT preemption_threshold;
UINT time_slice;
TX_THREAD *next_thread;
TX_THREAD *suspended_thread;
UINT status;

/* Retrieve information about the previously created
thread "my_thread." */

status = tx_thread_info_get(&my_thread, &name,
    &state, &run_count,
    &priority, &preemption_threshold,
    &time_slice, &next_thread,&suspended_thread);

/* If status equals TX_SUCCESS, the information requested is
valid. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_performance_info_get

获取线程性能信息

原型

```
UINT tx_thread_performance_info_get(
    TX_THREAD *thread_ptr,
    LONG *resumptions,
    ULONG *suspensions,
    ULONG *solicited_preemptions,
    ULONG *interrupt_preemptions,
    ULONG *priority_inversions,
    ULONG *time_slices,
    ULONG *relinquishes,
    ULONG *timeouts,
    ULONG *wait_aborts,
    TX_THREAD **last_preempted_by);
```

说明

此服务检索有关指定线程的性能信息。

> **IMPORTANT**
> 必须使用为此服务定义的 TX_THREAD_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。*

参数

- thread_ptr：指向之前创建的线程的指针。
- **resumptions**：指向此线程的恢复数这一目标的指针。
- **suspensions**：指向此线程的挂起数这一目标的指针。
- **solicited_preemptions**：指向由于此线程进行的 ThreadX API 服务调用而导致的抢占数这一目标的指针。
- **interrupt_preemptions**：指向此线程由于中断处理而导致的抢占数这一目标的指针。
- **priority_inversions**：指向此线程的优先级倒置数这一目标的指针。
- **time_slices**：指向此线程的时间片数这一目标的指针。
- **relinquishes**：指向此线程执行的线程放弃次数这一目标的指针。
- **timeouts**：指向此线程的挂起超时次数这一目标的指针。
- **wait_aborts**：指向对此线程执行的等待中止数这一目标的指针。
- **last_preempted_by**：指向最后抢占此线程的线程指针这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功获取线程性能信息。
- **TX_PTR_ERROR**：(0x03) 线程指针无效。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_THREAD my_thread;
ULONG resumptions;
ULONG suspensions;
ULONG solicited_preemptions;
ULONG interrupt_preemptions;
ULONG priority_inversions;
ULONG time_slices;
ULONG relinquishes;
ULONG timeouts;
ULONG wait_aborts;
TX_THREAD *last_preempted_by;

/* Retrieve performance information on the previously created
thread. */

status = tx_thread_performance_info_get(&my_thread, &resumptions,
    &suspensions,
    &solicited_preemptions, &interrupt_preemptions,
    &priority_inversions, &time_slices,
    &relinquishes, &timeouts,
    &wait_aborts, &last_preempted_by);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_performance_system_info_get

获取线程系统性能信息

原型

```
UINT tx_thread_performance_system_info_get(
    ULONG *resumptions,
    ULONG *suspensions,
    ULONG *solicited_preemptions,
    ULONG *interrupt_preemptions,
    ULONG *priority_inversions,
    ULONG *time_slices,
    ULONG *relinquishes,
    ULONG *timeouts,
    ULONG *wait_aborts,
    ULONG *non_idle_returns,
    ULONG *idle_returns);
```

说明

此服务检索系统中所有线程的性能信息。

必须使用为此服务定义的 TX_THREAD_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序

才能返回性能信息。*

参数

- **resumptions**：指向线程恢复总数这一目标的指针。
- **suspensions**：指向线程挂起总数这一目标的指针。
- **solicited_preemptions**：指向由于线程调用 ThreadX API 服务而导致的线程抢占总数这一目标的指针。
- **interrupt_preemptions**：指向由于中断处理而导致的线程抢占总数这一目标的指针。
- **priority_inversions**：指向线程优先级倒置总数这一目标的指针。
- **resumptions**：指向线程时间片总数这一目标的指针。
- **relinquishes**：指向线程放弃总次数这一目标的指针。
- **timeouts**：指向线程挂起超时总数这一目标的指针。
- **wait_aborts**：指向线程等待中止总数这一目标的指针。
- **non_idle_returns**：指向线程在另一个线程准备好执行时返回到系统的次数这一目标的指针。
- **idle_returns**：指向线程在没有任何其他线程准备好执行（系统空闲）时返回到系统的次数这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功获取线程系统性能信息。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
ULONG resumptions;
ULONG suspensions;
ULONG solicited_preemptions;
ULONG interrupt_preemptions;
ULONG priority_inversions;
ULONG time_slices;
ULONG relinquishes;
ULONG timeouts;
ULONG wait_aborts;
ULONG non_idle_returns;
ULONG idle_returns;

/* Retrieve performance information on all previously created
thread. */

status = tx_thread_performance_system_info_get(&resumptions,
    &suspensions,
    &solicited_preemptions, &interrupt_preemptions,
    &priority_inversions, &time_slices, &relinquishes,
    &timeouts, &wait_aborts, &non_idle_returns,
    &idle_returns);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_preemption_change

更改应用程序线程的抢占阈值

原型

```
UINT tx_thread_preemption_change(
    TX_THREAD *thread_ptr,
    UINT new_threshold,
    UINT *old_threshold);
```

说明

此服务更改指定线程的抢占阈值。抢占阈值阻止指定线程被等于或小于抢占阈值的线程抢占。

> **NOTE**
> 使用抢占阈值会禁用指定线程的时间片。

参数

- **thread_ptr**：指向以前创建的应用程序线程的指针。
- **new_threshold**：新的抢占阈值优先级（0 至 (TX_MAX_PRIORITIES-1)）。
- **old_threshold**：指向返回上一个抢占阈值的位置的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功更改抢占阈值。
- **TX_THREAD_ERROR**：(0X0E) 应用程序线程指针无效。
- **TX_THRESH_ERROR**：(0x18) 指定的新抢占阈值是无效的线程优先级（值不介于 0 到 TX_MAX_PRIORITIES-1 之间）或大于（优先级更低）当前线程优先级。
- **TX_PTR_ERROR**：(0x03) 指向之前的抢占阈值存储位置的指针无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程和计时器

可以抢占

是

示例

```
TX_THREAD my_thread;
UINT my_old_threshold;
UINT status;

/* Disable all preemption of the specified thread. The
current preemption-threshold is returned in
"my_old_threshold". Assume that "my_thread" has
already been created. */

status = tx_thread_preemption_change(&my_thread,
    0, &my_old_threshold);

/* If status equals TX_SUCCESS, the application thread is
non-preemptable by another thread. Note that ISRs are
not prevented by preemption disabling. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_priority_change
- tx_thread_relinquish

- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_priority_change

更改应用程序线程的优先级

原型

```
UINT tx_thread_priority_change(
    TX_THREAD *thread_ptr,
    UINT new_priority,
    UINT *old_priority);
```

说明

此服务更改指定线程的优先级。有效优先级的范围为 0 至 (TX_MAX_PRIORITES-1)，其中 0 表示最高优先级。

> **IMPORTANT**
> 指定线程的抢占阈值自动设置为新的优先级。如果需要新的阈值，则必须在此调用之后使用 tx_thread_preemption_change 服务。

参数

- **thread_ptr**：指向以前创建的应用程序线程的指针。
- **new_priority**：新线程优先级（0 至 (TX_MAX_PRIORITIES-1)）。
- **old_priority**：指向返回线程上一个优先级的位置的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功更改优先级。
- **TX_THREAD_ERROR**：(0X0E) 应用程序线程指针无效。
- **TX_PRIORITY_ERROR**：(0x0F) 指定的新优先级无效（值不介于 0 到 TX_MAX_PRIORITIES-1 之间）。
- **TX_PTR_ERROR**：(0x03) 指向之前的优先级存储位置的指针无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程和计时器

可以抢占

是

示例

```
TX_THREAD my_thread;
UINT my_old_priority;
UINT status;

/* Change the thread represented by "my_thread" to priority
0. */

status = tx_thread_priority_change(&my_thread,
    0, &my_old_priority);

/* If status equals TX_SUCCESS, the application thread is
now at the highest priority level in the system. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_relinquish

将控制权让给其他应用程序线程

原型

```
VOID tx_thread_relinquish(VOID);
```

说明

此服务将处理器控件让给其他具有相同或更高优先级的就绪线程。

> **NOTE**
> 除了将控制权让给具有相同优先级的线程以外，此服务还将控制权让给由于当前线程的抢占阈值设置而阻止执行的最高优先级线程。

参数

无

返回值

无

允许来自

线程数

可以抢占

是

示例

```
ULONG run_counter_1 = 0;
ULONG run_counter_2 = 0;

/* Example of two threads relinquishing control to
each other in an infinite loop. Assume that
both of these threads are ready and have the same
priority. The run counters will always stay within one
of each other. */

VOID my_first_thread(ULONG thread_input)
{
    /* Endless loop of relinquish. */
    while(1)
    {
        /* Increment the run counter. */
        run_counter_1++;

        /* Relinquish control to other thread. */
        tx_thread_relinquish();
    }
}

VOID my_second_thread(ULONG thread_input)
{

    /* Endless loop of relinquish. */
    while(1)
    {
        /* Increment the run counter. */
        run_counter_2++;

        /* Relinquish control to other thread. */
        tx_thread_relinquish();
    }
}
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep

- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_reset

重置线程

原型

```
UINT tx_thread_reset(TX_THREAD *thread_ptr);
```

说明

此服务重置指定线程，使其在创建线程时定义的入口点执行。线程必须处于 TX_COMPLETED 或 TX_TERMINATED 状态才能重置

> **IMPORTANT**
> 线程必须恢复后才能再次执行。

参数

- **mutex_ptr**：指向之前创建的线程的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功重置线程。
- **TX_NOT_DONE**：(0X20) 指定的线程未处于 TX_COMPLETED 或 TX_TERMINATED 状态 。
- **TX_THREAD_ERROR** (0X0E) 线程指针无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程数

示例

TX_THREAD my_thread;

```
TX_THREAD my_thread;

/* Reset the previously created thread "my_thread." */

status = tx_thread_reset(&my_thread);

/* If status is TX_SUCCESS the thread is reset. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get

- tx_thread_preformance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_resume

恢复挂起的应用程序线程

原型

```
UINT tx_thread_resume(TX_THREAD *thread_ptr);
```

说明

此服务恢复或准备执行之前由 tx_thread_suspend 调用挂起的线程。此外，此服务将恢复在没有自动启动的情况下创建的线程。

参数

- **thread_ptr**：指向挂起的应用程序线程的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功恢复线程。
- **TX_SUSPEND_LIFTED**：(0X19) 之前设置的延迟挂起被解除。
- **TX_THREAD_ERROR**：(0X0E) 应用程序线程指针无效。
- **TX_RESUME_ERROR**：(0X12) 指定的线程未挂起，或者以前被 tx_thread_suspend 以外的服务挂起。

允许来自

初始化、线程、计时器和 ISR

可以抢占

是

TX_THREAD my_thread;

示例

```
TX_THREAD my_thread;
UINT status;

/* Resume the thread represented by "my_thread". */
status = tx_thread_resume(&my_thread);

/* If status equals TX_SUCCESS, the application thread is
now ready to execute. */
```

另请参阅

- tx_thread_create

- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_sleep

在指定的时间内挂起当前线程

原型

```
UINT tx_thread_sleep(ULONG timer_ticks);
```

说明

此服务使发出调用的线程在指定的计时器时钟周期数内挂起。与计时器时钟周期相关的物理时间量是特定于应用程序的。此服务只能从应用程序线程调用。

参数

- **timer_ticks**：挂起发出调用的应用程序线程的计时器时钟周期数，范围为 0 至 0xFFFFFFFF。如果指定 0，服务将立即返回。

返回值

- **TX_SUCCESS**：(0X00) 线程成功进入睡眠状态。
- **TX_WAIT_ABORTED**：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- **TX_CALLER_ERROR**：(0X13) 从非线程调用了服务。

允许来自

线程数

可以抢占

是

示例

```
UINT status;

/* Make the calling thread sleep for 100
timer-ticks. */
status = tx_thread_sleep(100);

/* If status equals TX_SUCCESS, the currently running
application thread slept for the specified number of
timer-ticks. */
```

另请参阅

- tx_thread_create、tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_stack_error_notify

注册线程堆栈错误通知回调

原型

```
UINT tx_thread_stack_error_notify(VOID (*error_handler)(TX_THREAD *));
```

说明

此服务注册用于处理线程堆栈错误的通知回调函数。当 ThreadX 在执行过程中检测到线程堆栈错误时，它将调用此通知函数来处理该错误。对错误的处理完全由应用程序定义。可以完成从挂起冲突线程到重置整个系统的一切操作。

> **IMPORTANT**
> 必须使用为此服务定义的 TX_ENABLE_STACK_CHECKING 生成 ThreadX 库才能返回性能信息。

参数

- **error_handler**：指向应用程序的堆栈错误处理函数的指针。如果此值为 TX_NULL，则禁用通知。

返回值

- **TX_SUCCESS**：(0X00) 成功重置线程。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
void my_stack_error_handler(TX_THREAD *thread_ptr);

/* Register the "my_stack_error_handler" function with ThreadX
so that thread stack errors can be handled by the application. */
status = tx_thread_stack_error_notify(my_stack_error_handler);

/* If status is TX_SUCCESS the stack error handler is registered.*/
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_preformance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_suspend

挂起应用程序线程

原型

```
UINT tx_thread_suspend(TX_THREAD *thread_ptr);
```

说明

此服务挂起指定的应用程序线程。线程可以调用此服务来挂起自身。

> **NOTE**
>
> 如果指定的线程由于其他原因已挂起，则会在内部保持此挂起，直到之前的挂起被解除为止。发生这种情况时，将执行对指定线程的无条件挂起。其他无条件挂起请求将不起作用。

挂起后，该线程必须由 tx_thread_resume 恢复后才能再次执行。

参数

- **thread_ptr**：指向应用程序线程的指针。

返回值

- **TX_SUCCESS**: (0X00) 线程成功挂起。
- **TX_THREAD_ERROR**: (0X0E) 应用程序线程指针无效。
- **TX_SUSPEND_ERROR**: (0X14) 指定的线程处于已终止或已完成状态。
- **NX_CALLER_ERROR**: (0x13) 此服务的调用方无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_THREAD my_thread;
UINT status;

/* Suspend the thread represented by "my_thread". */
status = tx_thread_suspend(&my_thread);

/* If status equals TX_SUCCESS, the application thread is
unconditionally suspended. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_terminate

终止应用程序线程

原型

```
UINT tx_thread_terminate(TX_THREAD *thread_ptr);
```

说明

不管指定的应用程序线程是否挂起，此服务都将终止该线程。线程可以调用此服务来终止自身。

> **NOTE**
>
> 应用程序负责确保线程处于适合终止的状态。例如，在关键应用程序处理期间，或者在此类处理可能会处于未知状态的其他中间件组件中，不应终止线程。

> **IMPORTANT**
>
> 终止后，必须重置该线程才能再次执行。

参数

- **thread_ptr**：指向应用程序线程的指针。

返回值

- **TX_SUCCESS**：(0X00) 线程成功终止。
- **TX_THREAD_ERROR**：(0X0E) 应用程序线程指针无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程和计时器

可以抢占

是

示例

```
TX_THREAD my_thread;
UINT status;

/* Terminate the thread represented by "my_thread". */
status = tx_thread_terminate(&my_thread);

/* If status equals TX_SUCCESS, the thread is terminated
and cannot execute again until it is reset. */
```

另请参阅

- tx_thread_create tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_time_slice_change

更改应用程序线程的时间片

原型

```
UINT tx_thread_time_slice_change(
    TX_THREAD *thread_ptr,
    ULONG new_time_slice,
    ULONG *old_time_slice);
```

说明

此服务更改指定应用程序线程的时间片。为线程选择时间片可确保在具有相同或更高优先级的其他线程有机会执行之前，该线程的执行时间不会超过指定的计时器时钟周期数。

> **NOTE**
> 使用抢占阈值会禁用指定线程的时间片。

参数

- **thread_ptr**：指向应用程序线程的指针。
- **new_time_slice**：新的时间片值。合法值包括 TX_NO_TIME_SLICE 和从 1 到 0xFFFFFFFF 的数值。
- **old_time_slice**：指向用于存储指定线程先前时间片值的位置的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功设置时间片。
- **TX_THREAD_ERROR**：(0X0E) 应用程序线程指针无效。
- **TX_PTR_ERROR**：(0x03) 指向之前的时间片存储位置的指针无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程和计时器

可以抢占

否

示例

```
TX_THREAD my_thread;
ULONG my_old_time_slice;
UINT status;

/* Change the time-slice of the thread associated with
"my_thread" to 20. This will mean that "my_thread"
can only run for 20 timer-ticks consecutively before
other threads of equal or higher priority get a chance
to run. */
status = tx_thread_time_slice_change(&my_thread, 20,
    &my_old_time_slice);

/* If status equals TX_SUCCESS, the thread's time-slice
has been changed to 20 and the previous time-slice is
in "my_old_time_slice." */
```

另请参阅

- tx_thread_create

- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_wait_abort

tx_thread_wait_abort

中止指定线程的挂起

原型

```
UINT tx_thread_wait_abort(TX_THREAD *thread_ptr);
```

说明

此服务将中止指定线程的睡眠状态或任何其他对象挂起。如果中止等待，则从线程正在等待的服务返回 TX_WAIT_ABORTED 值。

> **NOTE**
> 此服务不会释放由 tx_thread_suspend 服务执行的显式挂起。

参数

- **thread_ptr**：指向以前创建的应用程序线程的指针。

返回值

- **TX_SUCCESS**：(0X00) 线程等待成功中止。
- **TX_THREAD_ERROR**：(0X0E) 应用程序线程指针无效。
- **TX_WAIT_ABORT_ERROR**：(0X1b) 指定的线程未处于等待状态。

允许来自

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_THREAD my_thread;
UINT status;

/* Abort the suspension condition of "my_thread." */
status = tx_thread_wait_abort(&my_thread);

/* If status equals TX_SUCCESS, the thread is now ready
again, with a return value showing its suspension
was aborted (TX_WAIT_ABORTED). */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change

tx_time_get

检索当前时间

应用程序计时器

原型

```
ULONG tx_time_get(VOID);
```

说明

此服务返回内部系统时钟的内容。每个计时器时钟周期将内部系统时钟增加 1。系统时钟在初始化期间设置为零，并且可以通过服务 tx_time_set 更改为特定值。

> **NOTE**
> 每个计时器时钟周期代表的实际时间特定于应用程序。

参数

无

返回值

- **系统时钟计时周期**：内部自由运行系统时钟的值。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
ULONG current_time;

/* Pickup the current system time, in timer-ticks. */
current_time = tx_time_get();

/* Current time now contains a copy of the internal system
clock. */
```

另请参阅

- tx_time_set

tx_time_set

设置当前时间

原型

```
VOID tx_time_set(ULONG new_time);
```

说明

此服务将内部系统时钟设置为指定的值。每个计时器时钟周期将内部系统时钟增加 1。

> **NOTE**
> 每个计时器时钟周期代表的实际时间特定于应用程序。

参数

- **new_time**：要添加到系统时钟的新时间，合法值的范围为 0 至 0xFFFFFFFF。

返回值

无

允许来自

线程、计时器和 ISR

可以抢占

否

示例

```
/* Set the internal system time to 0x1234. */
tx_time_set(0x1234);

/* Current time now contains 0x1234 until the next timer
interrupt. */
```

另请参阅

- tx_time_get

tx_timer_activate

激活应用程序计时器

原型

```
UINT tx_timer_activate(TX_TIMER *timer_ptr);
```

说明

此服务激活指定的应用程序计时器。同时过期的计时器的过期例程将按其激活的顺序执行。

> **NOTE**
> 过期的一次性计时器必须通过 tx_timer_change 重置，然后才能再次激活 *。

参数

- **timer_ptr**：指向以前创建的应用程序计时器的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功激活应用程序计时器。
- **TX_TIMER_ERROR**：(0X15) 应用程序计时器指针无效。
- **TX_ACTIVATE_ERROR**：(0X17) 计时器已处于活动状态或是已过期的一次性计时器。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_TIMER my_timer;
UINT status;

/* Activate an application timer. Assume that the
application timer has already been created. */
status = tx_timer_activate(&my_timer);

/* If status equals TX_SUCCESS, the application timer is
now active. */
```

另请参阅

- tx_timer_change
- tx_timer_create
- tx_timer_deactivate
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

tx_timer_change

更改应用程序计时器

原型

```
UINT tx_timer_change(
    TX_TIMER *timer_ptr,
    ULONG initial_ticks,
    ULONG reschedule_ticks);
```

说明

此服务更改指定应用程序计时器的过期特性。必须先停用该计时器，然后才能调用此服务。

> **NOTE**
> 在此服务之后，需要调用 tx_timer_activate 服务，以便再次启动计时器 *。

参数

- **timer_ptr**：指向计时器控制块的指针。
- **initial_ticks**：指定计时器过期的初始时钟周期数。合法值的范围为 1 至 0xFFFFFFFF。
- **reschedule_ticks**：指定第一个计时器过期后所有计时器过期的时钟周期数。如果此参数为 0，则计时器是一次性的。否则，对于周期性计时器，合法值的范围为 1 至 0xFFFFFFFF。

> **NOTE**
> 过期的一次性计时器必须通过 tx_timer_change 重置，然后才能再次激活 *。

返回值

- **TX_SUCCESS**：(0X00) 成功更改应用程序计时器。
- **TX_TIMER_ERROR**：(0X15) 应用程序计时器指针无效。
- **TX_TICK_ERROR**：(0x16) 为初始时钟周期提供的值无效（零）。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程、计时器和 ISR

可以抢占

否

示例

```
TX_TIMER my_timer;
UINT status;

/* Change a previously created and now deactivated timer
to expire every 50 timer ticks, including the initial
expiration. */
status = tx_timer_change(&my_timer,50, 50);

/* If status equals TX_SUCCESS, the specified timer is
changed to expire every 50 ticks. */

/* Activate the specified timer to get it started again. */
status = tx_timer_activate(&my_timer);
```

另请参阅

- tx_timer_activate
- tx_timer_create
- tx_timer_deactivate
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

tx_timer_create

创建应用程序计时器

原型

```
UINT tx_timer_create(
    TX_TIMER *timer_ptr,
    CHAR *name_ptr,
    VOID (*expiration_function)(ULONG),
    ULONG expiration_input,
    ULONG initial_ticks,
    ULONG reschedule_ticks,
    UINT auto_activate);
```

说明

此服务创建具有指定过期函数和周期的应用程序计时器。

参数

- **timer_ptr**：指向计时器控制块的指针
- **name_ptr**：指向计时器名称的指针。
- **expiration_function**：在计时器过期时要调用的应用程序函数。
- **expiration_input**：在计时器过期时要传递到过期函数的输入。
- **initial_ticks**：指定计时器过期的初始时钟周期数。合法值的范围为 1 至 0xFFFFFFFF。
- **reschedule_ticks**：指定第一个计时器过期后所有计时器过期的时钟周期数。如果此参数为 0，则计时器是一次性的。否则，对于周期性计时器，合法值的范围为 1 至 0xFFFFFFFF。

  > **NOTE**
  > 一次性计时器过期后，必须通过 tx_timer_change 将其重置，然后才能再次激活。

- **auto_activate**：确定创建期间是否自动激活计时器。如果此值为 TX_AUTO_ACTIVATE (0x01)，则激活计时器。否则，如果选择了值 TX_NO_ACTIVATE (0x00)，则在该计时器在非活动状态下创建。在这种情况下，随后需要调用 tx_timer_activate 服务来实际启动该计时器。

返回值

- **TX_SUCCESS**：(0X00) 成功创建应用程序计时器。
- **TX_TIMER_ERROR**：(0X15) 应用程序计时器指针无效。指针为 NULL 或已创建计时器。
- **TX_TICK_ERROR**：(0x16) 为初始时钟周期提供的值无效（零）。
- **TX_ACTIVATE_ERROR**：(0x17) 选择的激活无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

初始化和线程

可以抢占

否

示例

```
TX_TIMER my_timer;
UINT status;

/* Create an application timer that executes
"my_timer_function" after 100 ticks initially and then
after every 25 ticks. This timer is specified to start
immediately! */
status = tx_timer_create(&my_timer,"my_timer_name",
    my_timer_function, 0x1234, 100, 25,
    TX_AUTO_ACTIVATE);

/* If status equals TX_SUCCESS, my_timer_function will
be called 100 timer ticks later and then called every
25 timer ticks. Note that the value 0x1234 is passed to
my_timer_function every time it is called. */
```

另请参阅

- tx_timer_activate
- tx_timer_change
- tx_timer_deactivate
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

tx_timer_deactivate

停用应用程序计时器

原型

```
UINT tx_timer_deactivate(TX_TIMER *timer_ptr);
```

说明

此服务停用指定的应用程序计时器。如果计时器已停用，则此服务不起作用。

参数

- **timer_ptr**：指向以前创建的应用程序计时器的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功停用应用程序计时器。
- **TX_TIMER_ERROR**：(0X15) 应用程序计时器指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_TIMER my_timer;
UINT status;

/* Deactivate an application timer. Assume that the
application timer has already been created. */
status = tx_timer_deactivate(&my_timer);

/* If status equals TX_SUCCESS, the application timer is
now deactivated. */
```

另请参阅

- tx_timer_activate
- tx_timer_change
- tx_timer_create
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

tx_timer_delete

删除应用程序计时器

原型

```
UINT tx_timer_delete(TX_TIMER *timer_ptr);
```

说明

此服务删除指定的应用程序计时器。

> **NOTE**
> 应用程序负责阻止使用已删除的计时器。

参数

- **timer_ptr**：指向以前创建的应用程序计时器的指针。

返回值

- **TX_SUCCESS**：(0X00) 成功删除应用程序计时器。
- **TX_TIMER_ERROR**：(0X15) 应用程序计时器指针无效。
- **NX_CALLER_ERROR**：(0x13) 此服务的调用方无效。

允许来自

线程数

可以抢占

否

示例

```
TX_TIMER my_timer;
UINT status;

/* Delete application timer. Assume that the application
timer has already been created. */
status = tx_timer_delete(&my_timer);

/* If status equals TX_SUCCESS, the application timer is
deleted. */
```

另请参阅

- tx_timer_activate
- tx_timer_change
- tx_timer_create
- tx_timer_deactivate
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

tx_timer_info_get

检索有关应用程序计时器的信息

原型

```
UINT tx_timer_info_get(
    TX_TIMER *timer_ptr,
    CHAR **name,
    UINT *active,
    ULONG *remaining_ticks,
    ULONG *reschedule_ticks,
    TX_TIMER **next_timer);
```

说明

此服务检索有关指定应用程序计时器的信息。

参数

- **timer_ptr**：指向以前创建的应用程序计时器的指针。
- **name**：指向计时器名称指针这一目标的指针。
- **active**：指向计时器活动指示这一目标的指针。如果计时器处于非活动状态或从计时器本身调用此服务，则返回 TX_FALSE 值。否则，如果计时器处于活动状态，则返回 TX_TRUE 值。
- **remaining_ticks**：指向在计时器过期前剩余的计时器时钟周期数这一目标的指针。
- **reschedule_ticks**：指向将用于自动重新计划此计时器的计时器时钟周期数这一目标的指针。如果该值为零，则计时器是一次性的，不会重新计划。
- **next_timer**：指向下一个创建的应用程序计时器的指针这一目标的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功检索计时器信息。
- **TX_TIMER_ERROR**：(0X15) 应用程序计时器指针无效。

允许来自

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_TIMER my_timer;
CHAR *name;
UINT active;
ULONG remaining_ticks;
ULONG reschedule_ticks;
TX_TIMER *next_timer;
UINT status;

/* Retrieve information about the previously created
application timer "my_timer." */
status = tx_timer_info_get(&my_timer, &name,
    &active,&remaining_ticks,
    &reschedule_ticks,
    &next_timer);

/* If status equals TX_SUCCESS, the information requested is
valid. */
```

另请参阅

- tx_timer_activate
- tx_timer_change
- tx_timer_create
- tx_timer_deactivate
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

tx_timer_performance_info_get

获取计时器性能信息

原型

```
UINT tx_timer_performance_info_get(
    TX_TIMER *timer_ptr,
    ULONG *activates,
    ULONG *reactivates,
    ULONG *deactivates,
    ULONG *expirations,
    ULONG *expiration_adjusts);
```

说明

此服务检索有关指定应用程序计时器的性能信息。

> **IMPORTANT**
>
> 必须使用为此服务定义的 TX_TIMER_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。*

参数

- **timer_ptr**：指向之前创建的计时器的指针。
- **activates**：指向对此计时器执行的激活请求数这一目标的指针。
- **reactivates**：指向对此周期性计时器执行的自动重新激活次数这一目标的指针。
- **deactivates**：指向对此计时器执行的停用请求数这一目标的指针。
- **expirations**：指向此计时器过期次数这一目标的指针。
- **expiration_adjusts**：指向对此计时器执行的内部过期调整次数这一目标的指针。这些调整是在计时器中断处理中完成的，用于大于默认计时器列表大小的计时器（默认情况下即为过期时间大于 32 个时钟周期的计时器）。

> [注意] 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功获取计时器性能信息。
- **TX_PTR_ERROR**：(0x03) 计时器指针无效。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
TX_TIMER my_timer;
ULONG activates;
ULONG reactivates;
ULONG deactivates;
ULONG expirations;
ULONG expiration_adjusts;

/* Retrieve performance information on the previously created
timer. */
status = tx_timer_performance_info_get(&my_timer, &activates,
    &reactivates,&deactivates, &expirations,
    &expiration_adjusts);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_timer_activate
- tx_timer_change
- tx_timer_create
- tx_timer_deactivate
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_system_info_get

tx_timer_performance_system_info_get

获取计时器系统性能信息

原型

```
UINT tx_timer_performance_system_info_get(
    ULONG *activates,
    ULONG *reactivates,
    ULONG *deactivates,
    ULONG *expirations,
    ULONG *expiration_adjusts);
```

说明

此服务检索有关系统中所有应用程序计时器的性能信息。

> **IMPORTANT**
>
> 必须使用为此服务定义的 TX_TIMER_ENABLE_PERFORMANCE_INFO 生成 ThreadX 库和应用程序才能返回性能信息。

参数

- **activates**：指向对所有计时器执行的激活请求总数这一目标的指针。
- **reactivates**：指向对所有周期性计时器执行的自动重新激活总数这一目标的指针。
- **deactivates**：指向对所有计时器执行的停用请求总数这一目标的指针。
- **expirations**：指向所有计时器过期总次数这一目标的指针。
- **expiration_adjusts**：指向对所有计时器执行的内部过期调整总次数这一目标的指针。这些调整是在计时器中断处理中完成的，用于大于默认计时器列表大小的计时器（默认情况下即为过期时间大于 32 个时钟周期的计时器）。

> **NOTE**
>
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- **TX_SUCCESS**：(0X00) 成功获取计时器系统性能信息。
- **TX_FEATURE_NOT_ENABLED**：(0xFF) 在编译系统时未启用性能信息。

允许来自

初始化、线程、计时器和 ISR

示例

```
ULONG activates;
ULONG reactivates;
ULONG deactivates;
ULONG expirations;
ULONG expiration_adjusts;

/* Retrieve performance information on all previously created
timers. */
status = tx_timer_performance_system_info_get(&activates,
    &reactivates, &deactivates, &expirations,
    &expiration_adjusts);

/* If status is TX_SUCCESS the performance information was
successfully retrieved. */
```

另请参阅

- tx_timer_activate
- tx_timer_change

- tx_timer_create
- tx_timer_deactivate
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_info_get

第 5 章 - 适用于 Azure RTOS ThreadX 的设备驱动程序

2021/4/30 •

本章介绍适用于 Azure RTOS ThreadX 的设备驱动程序。本章介绍的信息旨在帮助开发人员编写特定于应用程序的驱动程序。

设备驱动程序简介

与外部环境的通信是大多数嵌入式应用程序的重要组成部分。此通信通过嵌入式应用程序软件可访问的硬件设备实现。负责管理此类设备的软件组件通常称为设备驱动程序。

嵌入式实时系统中的设备驱动程序本质上依赖于应用程序。这是因为以下两个主要原因：目标硬件非常多样化，对各个实时应用程序施加的性能要求同样巨大。因此，实际上不可能提供一组可满足每个应用程序的要求的通用驱动程序。由于这些原因，本章中的信息旨在帮助用户自定义现成的 ThreadX 设备驱动程序以及编写自己的特定驱动程序。

驱动程序函数

ThreadX 设备驱动程序由八个基本函数区域组成，如下所示。

- **驱动程序初始化**
- 驱动程序控制
- 驱动程序访问
- 驱动程序输入
- 驱动程序输出
- 驱动程序中断
- 驱动程序状态
- 驱动程序终止

每个驱动程序函数区域都为可选，但初始化除外。此外，每个区域中的确切处理特定于设备驱动程序。

驱动程序初始化

此函数区域负责初始化实际的硬件设备和驱动程序的内部数据结构。在初始化完成之前，不允许调用其他驱动程序服务。

> **NOTE**
> *通常可通过 *tx_application_define 函数或初始化线程调用驱动程序的初始化函数组件。_ *

驱动程序控制

驱动程序初始化并准备好运行后，此函数区域负责运行时控制。通常，运行时控制包括对基础硬件设备进行更改。例如，更改串行设备的波特率或查找磁盘上的新扇区。

驱动程序访问

某些设备驱动程序只能通过单个应用程序线程调用。在这种情况下，不需要此函数区域。但是，在多个线程需要同时访问驱动程序的应用程序中，必须通过在设备驱动程序中添加分配/释放功能来控制这些线程的交互。或者，应用程序也可以使用信号灯来控制驱动程序访问，并避免驱动程序内部出现额外的开销和复杂情况。

驱动程序输入

此函数区域负责所有设备输入。与驱动程序输入相关的主要问题通常涉及如何缓冲输入以及线程如何等待此类输入。

驱动程序输出

此函数区域负责所有设备输出。与驱动程序输出相关的主要问题通常涉及如何缓冲输出以及线程如何等待执行输出。

驱动程序中断

大多数实时系统都依赖硬件中断向驱动程序通知设备输入、输出、控制和错误事件。中断可为此类外部事件提供有保证的响应时间。驱动程序软件可能会定期检查外部硬件以查找此类事件，而不依赖中断。这种技术称为轮询。它的实时性低于中断，但轮询对于一些实时性较低的应用程序可能有意义。

驱动程序状态

此函数区域负责提供与驱动程序操作关联的运行时状态和统计信息。此函数区域管理的信息通常包括以下内容。

- 当前设备状态
- 输入字节数
- 输出字节数
- 设备错误计数

驱动程序终止

此函数区域为可选。仅当驱动程序和/或物理硬件设备需要关闭时，才需要此函数区域。终止后，在重新初始化之前不能再次调用驱动程序。

简单驱动程序示例

举例是介绍设备驱动程序的最佳方式。在此示例中，驱动程序假定简单的串行硬件设备具有配置寄存器、输入寄存器和输出寄存器。此简单驱动程序示例阐释了初始化、输入、输出和中断函数区域。

简单驱动程序初始化

简单驱动程序的 tx_sdriver_initialize 函数可创建两个计数信号灯，用于管理驱动程序的输入和输出操作。当串行硬件设备收到字符时，输入信号灯由输入 ISR 设置。因此，所创建的输入信号灯的初始计数为零。

而输出信号灯用于指示串行硬件传输寄存器的可用性。创建输出信号灯时，将其值设为 1，表示传输寄存器在初始时可用。

初始化函数还负责为输入和输出通知安装低级别中断向量处理程序。与其他 ThreadX 中断服务例程一样，低级别处理程序在调用简单驱动程序 ISR 之前必须调用 tx_thread_context_save。* _ 驱动程序 ISR 返回后，低级别处理程序必须调用 tx_thread_context_restore。*_**

> **IMPORTANT**
> *请务必在调用任何其他驱动程序函数之前调用初始化。通常可通过 tx_application_define 调用驱动程序初始化。

```
VOID tx_sdriver_initialize(VOID)
{
    /* Initialize the two counting semaphores used to control
    the simple driver I/O. */
    tx_semaphore_create(&tx_sdriver_input_semaphore,
                        "simple driver input semaphore", 0);
    tx_semaphore_create(&tx_sdriver_output_semaphore,
                        "simple driver output semaphore", 1);

    /* Setup interrupt vectors for input and output ISRs.
    The initial vector handling should call the ISRs
    defined in this file. */

    /* Configure serial device hardware for RX/TX interrupt
    generation, baud rate, stop bits, etc. */
}
```

图 9. 简单驱动程序初始化

简单驱动程序输入

简单驱动程序的输入以输入信号灯为中心。系统收到串行设备输入中断时，会设置输入信号灯。如果一个或多个线程正在等待从驱动程序接收字符，则会恢复等待时间最长的线程。如果没有任何线程正在等待，则信号灯会保持不变，直到线程调用驱动程序输入函数为止。

系统在处理简单驱动程序输入时有几个限制。其中最重要的一点是，系统可能会删除输入字符。之所以有这种可能性，是因为系统在处理前一个字符之前无法缓冲到达的输入字符。可以通过添加输入字符缓冲区来轻松解决此限制。

> **NOTE**
>
> *只有线程可以调用 tx_sdriver_input 函数。* *_ _*

图 10 显示了与简单驱动程序输入关联的源代码。

```
UCHAR tx_sdriver_input(VOID)
{
  /* Determine if there is a character waiting. If not,
  suspend. */
  tx_semaphore_get(&tx_sdriver_input_semaphore,
  TX_WAIT_FOREVER;

  /* Return character from serial RX hardware register. */
  return(*serial_hardware_input_ptr);
}
  VOID tx_sdriver_input_ISR(VOID)
{
  /* See if an input character notification is pending. */
  if (!tx_sdriver_input_semaphore.tx_semaphore_count)
  {
    /* If not, notify thread of an input character. */
    tx_semaphore_put(&tx_sdriver_input_semaphore);
  }
}
```

图 10. 简单驱动程序输入

简单驱动程序输出

当串行设备的传输寄存器空闲时，输出处理将使用输出信号灯发出信号。在将输出字符实际写入到设备之前，系统会获取输出信号灯的相关信息。如果该信号灯不可用，则表示之前的传输尚未完成。

输出 ISR 负责处理传输完成中断。处理输出 ISR 等同于设置输出信号灯，从而允许输出另一个字符。

> **NOTE**
>
> *只有线程可以调用 tx_sdriver_output函数。* *_ _

图 11 显示了与简单驱动程序输出关联的源代码。

```
VOID tx_sdriver_output(UCHAR alpha)
{
  /* Determine if the hardware is ready to transmit a
  character. If not, suspend until the previous output
  completes. */
  tx_semaphore_get(&tx_sdriver_output_semaphore,
                                        TX_WAIT_FOREVER);

  /* Send the character through the hardware. */
  *serial_hardware_output_ptr = alpha;
}

VOID tx_sdriver_output_ISR(VOID)
{
  /* Notify thread last character transmit is
  complete. */
  tx_semaphore_put(&tx_sdriver_output_semaphore);
}
```

图 11. 简单驱动程序输出

简单驱动程序的缺点

此简单设备驱动程序示例阐释了 ThreadX 设备驱动程序的基本概念。但是，由于简单设备驱动程序无法解决数据缓冲问题或任何开销问题，因此它不能完全代表实际 ThreadX 驱动程序。下一节介绍了与设备驱动程序相关的一些更高级的问题。

驱动程序高级问题

如前所述，设备驱动程序具有与应用程序不同的要求。某些应用程序可能需要缓冲大量数据，而其他应用程序可能由于设备中断频率较高而需要优化的驱动程序 ISR。

I/O 缓冲

实时嵌入式应用程序中的数据缓冲需要进行大量规划。有些设计由基础硬件设备决定。如果设备提供基本的字节 I/O，则简单的循环缓冲区可能符合要求。但是，如果设备提供块 I/O、DMA I/O 或数据包 I/O，则可能需要缓冲区管理方案。

循环字节缓冲区

循环字节缓冲区通常在管理简单串行硬件设备（例如 UART）的驱动程序中使用。这种情况下最常使用两个循环缓冲区：一个用于输入，另一个用于输出。

每个循环字节缓冲区都包含一个字节内存区域（通常是一个 UCHAR 数组）、一个读指针和一个写指针。如果读指针和写指针引用缓冲区中的同一个内存位置，则将缓冲区视为空的。驱动程序初始化会将缓冲区读指针和缓冲区写指针设置为缓冲区的起始地址。

循环缓冲区输入

输入缓冲区用于在应用程序准备就绪之前保存到达的字符。收到输入字符（通常是在中断服务例程中）时，将从硬件设备检索新字符，并将其放入输入缓冲区中写指针所指向的位置。然后，写指针将前进到缓冲区中的下一个位置。如果下一个位置超出了缓冲区的末尾，则写指针将设置为缓冲区的开头。如果新的写指针与读指针相同，则通过取消写指针前进来解决队列已满状况。

向驱动程序发送的应用程序输入字节请求首先检查输入缓冲区的读指针和写指针。如果读指针与写指针相同，则缓冲区为空。否则，如果读指针不相同，则会从输入缓冲区中复制读指针所指向的字节，并且读指针会前进到下一个缓冲区位置。如果新的读指针超出了缓冲区的末尾，则将其重置为开头。图 12 显示了循环输入缓冲区的逻辑。

```
UCHAR   tx_input_buffer[MAX_SIZE];
UCHAR   tx_input_write_ptr;
UCHAR   tx_input_read_ptr;

/* Initialization. */
tx_input_write_ptr =    &tx_input_buffer[0];
tx_input_read_ptr =     &tx_input_buffer[0];

/* Input byte ISR... UCHAR alpha has character from device. */
save_ptr = tx_input_write_ptr;
*tx_input_write_ptr++ = alpha;
if (tx_input_write_ptr > &tx_input_buffer[MAX_SIZE-1])
    tx_input_write_ptr = &tx_input_buffer[0]; /* Wrap */
if (tx_input_write_ptr == tx_input_read_ptr)
    tx_input_write_ptr = save_ptr; /* Buffer full */

/* Retrieve input byte from buffer... */
if (tx_input_read_ptr != tx_input_write_ptr)
{
  alpha = *tx_input_read_ptr++;
  if (tx_input_read_ptr > &tx_input_buffer[MAX_SIZE-1])
      tx_input_read_ptr = &tx_input_buffer[0];
}
```

图 12. 循环输入缓冲区的逻辑

> **NOTE**
> *为了确保操作的可靠性，在操作循环输入缓冲区以及循环输出缓冲区的读指针和写指针时可能需要锁定中断。*

循环输出缓冲区

输出缓冲区用于在硬件设备完成发送前一个字节之前保存到达用于输出的字符。输出缓冲区处理类似于输入缓冲区处理，不同之处在于传输完成中断处理操作输出读指针，而应用程序输出请求使用输出写指针。在其他方面，输出缓冲区处理与输入缓冲区处理相同。图 13 显示了循环输出缓冲区的逻辑。

```
UCHAR   tx_output_buffer[MAX_SIZE];
UCHAR   tx_output_write_ptr;
UCHAR   tx_output_read_ptr;

/* Initialization. */
tx_output_write_ptr = &tx_output_buffer[0];
tx_output_read_ptr = &tx_output_buffer[0];

/* Transmit complete ISR... Device ready to send. */
if (tx_output_read_ptr != tx_output_write_ptr)
{
  *device_reg = *tx_output_read_ptr++;
  if (tx_output_read_reg > &tx_output_buffer[MAX_SIZE-1])
      tx_output_read_ptr = &tx_output_buffer[0];
}

/* Output byte driver service. If device busy, buffer! */
save_ptr = tx_output_write_ptr;
*tx_output_write_ptr++ = alpha;
if (tx_output_write_ptr > &tx_output_buffer[MAX_SIZE-1])
    tx_output_write_ptr = &tx_output_buffer[0]; /* Wrap */
if (tx_output_write_ptr == tx_output_read_ptr)
    tx_output_write_ptr = save_ptr; /* Buffer full! */
```

图 13. 循环输出缓冲区的逻辑

缓冲区 I/O 管理

为了提高嵌入式微处理器的性能，许多外围设备使用软件提供的缓冲区来传输和接收数据。在某些实现中，可能会使用多个缓冲区来传输或接收单个数据包。

I/O 缓冲区的大小和位置由应用程序和/或驱动程序软件确定。通常，缓冲区的大小是固定的，并在 ThreadX 块内存池中进行管理。图 14 介绍了典型 I/O 缓冲区以及管理缓冲区分配的 ThreadX 块内存池。

```
typedef struct TX_IO_BUFFER_STRUCT
{
      struct TX_IO_BUFFER_STRUCT *tx_next_packet;
      struct TX_IO_BUFFER_STRUCT *tx_next_buffer;
      UCHAR tx_buffer_area[TX_MAX_BUFFER_SIZE];
} TX_IO_BUFFER;

TX_BLOCK_POOL tx_io_block_pool;

/* Create a pool of I/O buffers. Assume that the pointer
"free_memory_ptr"points to an available memory area that
is 64 KBytes in size. */
tx_block_pool_create(&tx_io_block_pool,
                  "Sample IO Driver Buffer Pool",
                  free_memory_ptr, 0x10000,
                  sizeof(TX_IO_BUFFER));
```

图 14. I/O 缓冲区

TX_IO_BUFFER

typedef TX_IO_BUFFER 包含两个指针。tx_next_packet 指针用于在输入列表或输出列表中链接多个数据包。tx_next_buffer 指针用于将构成来自设备的单个数据包的缓冲区链接到一起。从池中分配缓冲区时，这两个指针都设置为 NULL。此外，某些设备可能需要另一个字段来指示实际包含数据的缓冲区大小。

缓冲 I/O 的优点

缓冲区 I/O 方案的优点是什么？ 最大的优点是不会在设备寄存器和应用程序的内存之间复制数据。相反，驱动程序会为设备提供一系列缓冲区指针。物理设备 I/O 直接利用提供的缓冲区内存。

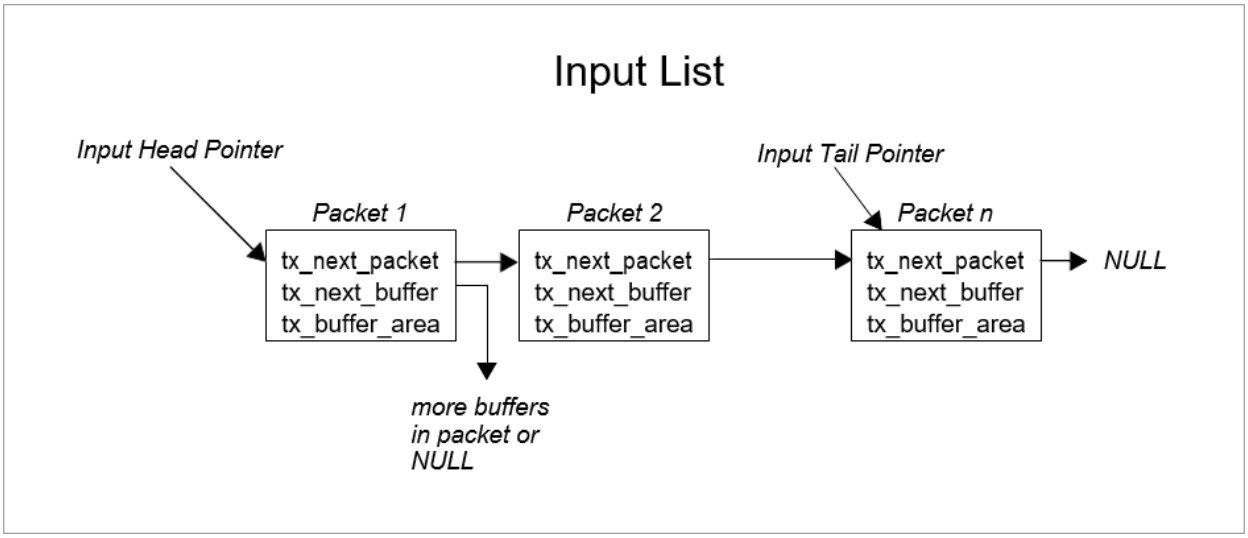
使用处理器复制信息的输入数据包或输出数据包会产生非常高的开销，在任何高吞吐量 I/O 情况下应避免使用这种方法。

缓冲 I/O 方法的另一个优点是输入列表和输出列表不会出现已满状况。所有可用的缓冲区在任何时候都可以位于任一列表中。这与本章前面介绍的简单字节循环缓冲区相反。每个缓冲区都具有固定大小，该大小在编译时确定。

缓冲驱动程序的职责

缓冲设备驱动程序只负责管理 I/O 缓冲区的链接列表。对于在应用程序软件准备就绪之前接收的数据包，会维护一个输入缓冲区列表。相反，对于发送速率比硬件设备的处理速度更快的数据包，会维护一个输出缓冲区列表。图 15 显示了包含数据包以及构成每个数据包的缓冲区的输入列表和输出列表的简单链接。

输入列表



输出列表

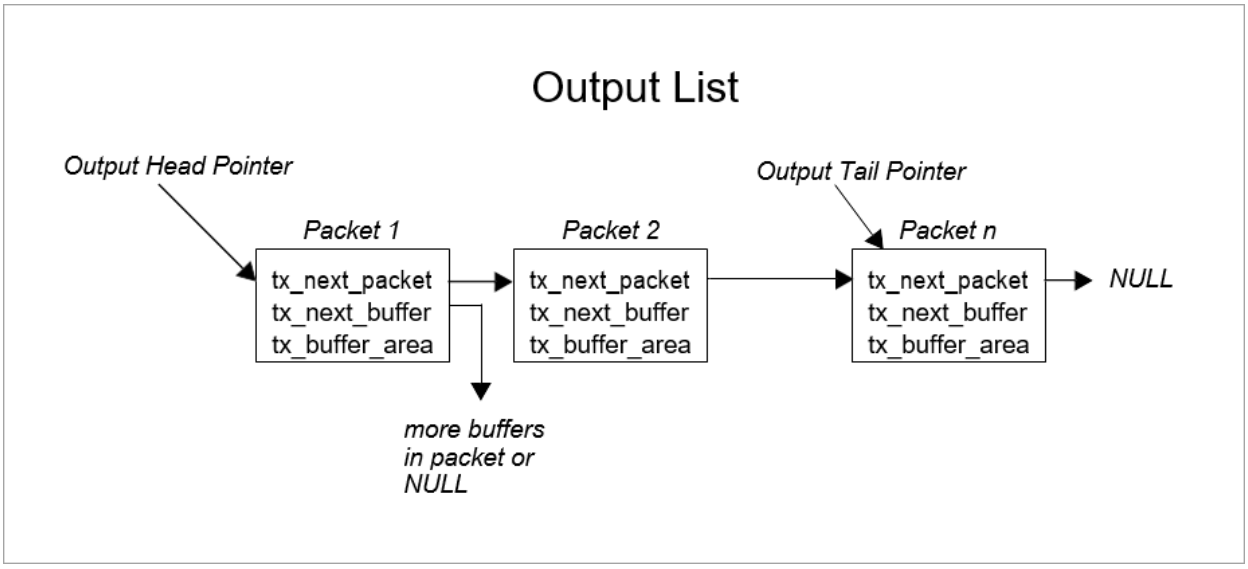


图 15. 输入列表-输出列表

应用程序使用相同的 I/O 缓冲区与缓冲驱动程序交互。传输时，应用程序软件会为驱动程序提供一个或多个缓冲区用于传输数据。当应用程序软件请求输入时，驱动程序将在 I/O 缓冲区中返回输入数据。

> **NOTE**
>
> *在某些应用程序中，构建要求应用程序为驱动程序的输入缓冲区交换可用缓冲区的驱动程序输入接口可能会很有用。这可以减少驱动程序内部的一些缓冲区分配处理。*

中断管理

在某些应用程序中，设备中断频率太高可能会禁止以 C 源代码格式编写 ISR，或者在每次中断时禁止与 ThreadX 交互。例如，如果需要 25us 来保存并还原中断的上下文，则当中断频率为 50us 时，建议不要执行完整上下文保存。在这种情况下，可以使用小型汇编语言 ISR 来处理大多数设备中断。此低开销 ISR 仅在必要时才与 ThreadX 交互。

在第 3 章末尾的中断管理讨论中可以找到类似的讨论。

线程暂停

在本章前面介绍的简单驱动程序示例中，输入服务的调用方在某个字符不可用时会暂停。在某些应用程序中，这可能是无法接受的。

例如，如果负责处理驱动程序输入的线程还承担其他职责，则只在驱动程序输入时暂停可能无法起作用。相反，需要对驱动程序进行自定义，使其以向线程发送其他处理请求的相似方式请求处理。

在大多数情况下，输入缓冲区放置在链接列表中，而输入事件消息会发送到线程的输入队列。

第 6 章 - Azure RTOS ThreadX 演示系统

2021/4/29 •

本章旨在介绍所有 Azure RTOS ThreadX 处理器支持包随附的演示系统。

概述

每个 ThreadX 产品分发均包含演示系统，可在所有受支持的微处理器上运行。

此示例系统可在分发文件 demo_threadx.c 中定义，旨在说明如何在嵌入式多线程环境中使用 ThreadX。演示包括初始化、八个线程、一个字节池、一个块池、一个队列、一个信号量、一个互斥体和一个事件标志组。

> **NOTE**
> 除线程的堆栈大小外，该演示应用程序在所有 ThreadX 支持的处理器上均相同。

demo_threadx.c 的完整列表，包括在本章其余部分引用的行号。

应用程序定义

完成基本的 ThreadX 初始化后，将执行 tx_application_define 函数。该函数负责设置所有初始系统资源，包括线程、队列、信号量、互斥体、事件标志和内存池。

演示系统的 tx_application_define*（*第 60-164 行*）按以下顺序创建演示对象：

- byte_pool_0
- thread_0
- thread_1
- thread_2
- thread_3
- thread_4
- thread_5
- thread_6
- thread_7
- queue_0
- semaphore_0
- event_flags_0
- mutex_0
- block_pool_0

该演示系统不会创建任何其他附加 ThreadX 对象。但是，实际应用程序可能会在运行时在执行线程内部创建系统对象。

初始执行

所有线程均使用 TX_AUTO_START 选项创建。因此，这些线程初步准备就绪，可以执行。tx_application_define 完成后，控制权将转移到线程计划程序，并从该处转移到各个线程。

线程执行的顺序取决于它们的优先级和创建顺序。在演示系统中，首先执行 thread_0，因为其具有最高优先级（创建时优先级为 1）。thread_0 挂起后，将执行 thread_5，然后执行 thread_3*、*thread_4*、*thread_6*、*thread_7 *

、thread_1，最后执行thread_2 **。

> **NOTE**
> 即使 thread_3 和 thread_4 具有相同优先级（创建时优先级均为 8），仍会首先执行 thread_3，因为 thread_3 在 thread_4 之前已创建并准备就绪。具有相同优先级的线程以 FIFO 方式执行。

线程 0

函数 thread_0_entry 标记线程的入口点（第 167-190 行）。Thread_0 是演示系统中要执行的第一个线程。其处理方式很简单：递增其计数器，休眠 10 个计时器时钟周期，设置一个事件标志以唤醒 thread_5，然后重复该序列。

Thread_0 是系统中优先级最高的线程。当其请求的休眠过期时，将抢占演示中任何其他正在执行的线程。

线程 1

函数 thread_1_entry*_ 标记线程的入口点（第 193-216 行 *）。Thread_1 是演示系统中要执行的倒数第二个线程。其处理方式包括递增其计数器，（通过 queue_0*）向 thread_2 发送消息，并重复该序列。请注意，只要 queue_0 已满，thread_1* 就会挂起（第 207 行_*）。

线程 2

函数 thread_2_entry 标记线程的入口点（第 219-243 行）。Thread_2 是演示系统中要执行的最后一个线程。其处理方式包括递增其计数器，（通过 queue_0）从 thread_1 获取消息，并重复该序列。请注意，只要 queue_0 为空，thread_2*_ 就会挂起（第 233 行_）。

尽管 thread_1 _ 和 thread_2_ 在演示系统中共享最低优先级（优先级为 16**），但它们是线程 3 和 4，

也是在大多数时候都可以执行的唯一线程。它们也是使用时间切片创建的唯一线程（第 87 行和第 93 行）。在执行另一个线程之前，每个线程最多可执行 4 个计时器时钟周期。

线程 3 和 4

函数 thread_3_and_4_entry 标记 thread_3*_ 和 thread_4_*的入口点（第 246-280 行）。这两个线程的优先级均为 8，这使得它们成为演示系统中要执行的第三个和第四个线程。每个线程的处理方式均相同：递增其计数器，获取 semaphore_0，休眠 2 个计时器时钟周期，释放 semaphore_0，并重复该序列。请注意，只要 semaphore_0 不可用，所有线程均会挂起（第 264 行）。

同时，两个线程在主处理中使用相同函数。这不会带来任何问题，因为它们都有自己唯一的堆栈，并且 C 天生是可重入函数。每个线程通过检查线程输入参数（第 258 行）来确定自己是哪一个线程，该参数可在创建线程时设置（第 102 行和第 109 行）。

> **NOTE**
> 在线程执行过程中获取当前线程点，并将其与控制块的地址进行比较，以确定线程标识，这一操作也很合理。

线程 5

函数 thread_5_entry 标记线程的入口点（第 283-305 行）。Thread_5 是演示系统中要执行的第二个线程。其处理方式包括递增其计数器，（通过 event_flags_0）从 thread_0 获取事件标志，并重复该序列。请注意，只要 event_flags_0 中的事件标志不可用，thread_5 就会挂起（第 298 行）。

线程 6 和 7

函数 thread_6_and_7_entry 标记 thread_6*_ 和 thread_7_* 的入口点（第 307-358 行）。这两个线程的优先级均为 8，这使得它们成为演示系统中要执行的第五个和第六个线程。每个线程的处理方式均相同：递增其计数器，获取 mutex_0 两次，休眠 2 个计时器时钟周期，释放 mutex_0 两次，并重复该序列。请注意，只要 mutex_0 不可用，所有线程均会挂起（第 325 行）。

同时，两个线程在主处理中使用相同函数。这不会带来任何问题，因为它们都有自己唯一的堆栈，并且 C 天生是可重入函数。每个线程通过检查线程输入参数（第 319 行）来确定自己是哪一个线程，该参数可在创建线程时设置（第 126 行和第 133 行）。

观看演示

每个演示线程均会递增自己的唯一计数器。可以检查以下计数器以核实演示的操作：

- thread_0_counter
- thread_1_counter
- thread_2_counter
- thread_3_counter
- thread_4_counter
- thread_5_counter
- thread_6_counter
- thread_7_counter

在演示执行过程中，每个计数器都应继续增加，其中，thread_1_counter*_ 和 thread_2_counter_* 的增加速度最快。

分发文件：demo_threadx.c

此部分显示 demo_threadx.c 的完整列表，包括本章中引用的行号。

```
/* This is a small demo of the high-performance ThreadX kernel. It includes examples of eight
threads of different priorities, using a message queue, semaphore, mutex, event flags group,
byte pool, and block pool. */

#include "tx_api.h"

#define DEMO_STACK_SIZE 1024
#define DEMO_BYTE_POOL_SIZE 9120
#define DEMO_BLOCK_POOL_SIZE 100
#define DEMO_QUEUE_SIZE 100

/* Define the ThreadX object control blocks... */

TX_THREAD thread_0;
TX_THREAD thread_1;
TX_THREAD thread_2;
TX_THREAD thread_3;
TX_THREAD thread_4;
TX_THREAD thread_5;
TX_THREAD thread_6;
TX_THREAD thread_7;
TX_QUEUE queue_0;
TX_SEMAPHORE semaphore_0;
TX_MUTEX mutex_0;
TX_EVENT_FLAGS_GROUP event_flags_0;
TX_BYTE_POOL byte_pool_0;
TX_BLOCK_POOL block_pool_0;

/* Define the counters used in the demo application... */

ULONG thread_0_counter;
```

```
ULONG thread_1_counter;
ULONG thread_1_messages_sent;
ULONG thread_2_counter;
ULONG thread_2_messages_received;
ULONG thread_3_counter;
ULONG thread_4_counter;
ULONG thread_5_counter;
ULONG thread_6_counter;
ULONG thread_7_counter;

/* Define thread prototypes. */

void thread_0_entry(ULONG thread_input);
void thread_1_entry(ULONG thread_input);
void thread_2_entry(ULONG thread_input);
void thread_3_and_4_entry(ULONG thread_input);
void thread_5_entry(ULONG thread_input);
void thread_6_and_7_entry(ULONG thread_input);


/* Define main entry point. */

int main()
{
    /* Enter the ThreadX kernel. */
    tx_kernel_enter();
}

/* Define what the initial system looks like. */
void tx_application_define(void *first_unused_memory)
{

    CHAR *pointer;

    /* Create a byte memory pool from which to allocate the thread stacks. */
    tx_byte_pool_create(&byte_pool_0, "byte pool 0", first_unused_memory,
        DEMO_BYTE_POOL_SIZE);

    /* Put system definition stuff in here, e.g., thread creates and other assorted
        create information. */

    /* Allocate the stack for thread 0. */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    /* Create the main thread. */
    tx_thread_create(&thread_0, "thread 0", thread_0_entry, 0,
        pointer, DEMO_STACK_SIZE,
        1, 1, TX_NO_TIME_SLICE, TX_AUTO_START);

    /* Allocate the stack for thread 1. */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    /* Create threads 1 and 2. These threads pass information through a ThreadX
        message queue. It is also interesting to note that these threads have a time
        slice. */
    tx_thread_create(&thread_1, "thread 1", thread_1_entry, 1,
        pointer, DEMO_STACK_SIZE,
        16, 16, 4, TX_AUTO_START);

    /* Allocate the stack for thread 2. */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
        tx_thread_create(&thread_2, "thread 2", thread_2_entry, 2,
        pointer, DEMO_STACK_SIZE,
        16, 16, 4, TX_AUTO_START);

    /* Allocate the stack for thread 3. */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    /* Create threads 3 and 4. These threads compete for a ThreadX counting semaphore.
```

```
        An interesting thing here is that both threads share the same instruction area. */
    tx_thread_create(&thread_3, "thread 3", thread_3_and_4_entry, 3,
        pointer, DEMO_STACK_SIZE,
        8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

    /* Allocate the stack for thread 4. */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    tx_thread_create(&thread_4, "thread 4", thread_3_and_4_entry, 4,
        pointer, DEMO_STACK_SIZE,
        8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

    /* Allocate the stack for thread 5. */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    /* Create thread 5. This thread simply pends on an event flag, which will be set
        by thread_0. */
    tx_thread_create(&thread_5, "thread 5", thread_5_entry, 5,
        pointer, DEMO_STACK_SIZE,
        4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);

    /* Allocate the stack for thread 6. */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    /* Create threads 6 and 7. These threads compete for a ThreadX mutex. */
    tx_thread_create(&thread_6, "thread 6", thread_6_and_7_entry, 6,
        pointer, DEMO_STACK_SIZE,
        8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

    /* Allocate the stack for thread 7. */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    tx_thread_create(&thread_7, "thread 7", thread_6_and_7_entry, 7,
        pointer, DEMO_STACK_SIZE,
        8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

    /* Allocate the message queue. */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_QUEUE_SIZE*sizeof(ULONG), TX_NO_WAIT);

    /* Create the message queue shared by threads 1 and 2. */
    tx_queue_create(&queue_0, "queue 0", TX_1_ULONG, pointer, DEMO_QUEUE_SIZE*sizeof(ULONG));

    /* Create the semaphore used by threads 3 and 4. */
    tx_semaphore_create(&semaphore_0, "semaphore 0", 1);

    /* Create the event flags group used by threads 1 and 5. */
    tx_event_flags_create(&event_flags_0, "event flags 0");

    /* Create the mutex used by thread 6 and 7 without priority inheritance. */
    tx_mutex_create(&mutex_0, "mutex 0", TX_NO_INHERIT);

    /* Allocate the memory for a small block pool. */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_BLOCK_POOL_SIZE, TX_NO_WAIT);

    /* Create a block memory pool to allocate a message buffer from. */
    tx_block_pool_create(&block_pool_0, "block pool 0", sizeof(ULONG), pointer,
        DEMO_BLOCK_POOL_SIZE);

    /* Allocate a block and release the block memory. */
    tx_block_allocate(&block_pool_0, &pointer, TX_NO_WAIT);

    /* Release the block back to the pool. */
    tx_block_release(pointer);
}

/* Define the test threads. */
void thread_0_entry(ULONG thread_input)
{
    UINT status;
```

```
    /* This thread simply sits in while-forever-sleep loop. */
    while(1)
    {

        /* Increment the thread counter. */
        thread_0_counter++;

        /* Sleep for 10 ticks. */
        tx_thread_sleep(10);

        /* Set event flag 0 to wakeup thread 5. */
        status = tx_event_flags_set(&event_flags_0, 0x1, TX_OR);

        /* Check status. */
        if (status != TX_SUCCESS)
            break;
    }
}


void thread_1_entry(ULONG thread_input)
{
    UINT status;


    /* This thread simply sends messages to a queue shared by thread 2. */
    while(1)
    {
        /* Increment the thread counter. */
        thread_1_counter++;

        /* Send message to queue 0. */
        status = tx_queue_send(&queue_0, &thread_1_messages_sent, TX_WAIT_FOREVER);

        /* Check completion status. */
        if (status != TX_SUCCESS)
            break;

        /* Increment the message sent. */
        thread_1_messages_sent++;
    }
}


void thread_2_entry(ULONG thread_input)
{
    ULONG received_message;
    UINT status;

    /* This thread retrieves messages placed on the queue by thread 1. */
    while(1)
    {
        /* Increment the thread counter. */
        thread_2_counter++;

        /* Retrieve a message from the queue. */
        status = tx_queue_receive(&queue_0, &received_message, TX_WAIT_FOREVER);

        /* Check completion status and make sure the message is what we
        expected. */
        if ((status != TX_SUCCESS) || (received_message != thread_2_messages_received))
            break;

        /* Otherwise, all is okay. Increment the received message count. */
        thread_2_messages_received++;
    }
}
```

```
}


void thread_3_and_4_entry(ULONG thread_input)
{
    UINT status;


    /* This function is executed from thread 3 and thread 4. As the loop
    below shows, these function compete for ownership of semaphore_0. */
    while(1)
    {
        /* Increment the thread counter. */
        if (thread_input == 3)
            thread_3_counter++;
        else
            thread_4_counter++;

        /* Get the semaphore with suspension. */
        status = tx_semaphore_get(&semaphore_0, TX_WAIT_FOREVER);

        /* Check status. */
        if (status != TX_SUCCESS)
            break;

        /* Sleep for 2 ticks to hold the semaphore. */
        tx_thread_sleep(2);

        /* Release the semaphore. */
        status = tx_semaphore_put(&semaphore_0);

        /* Check status. */
        if (status != TX_SUCCESS)
            break;
    }
}


void thread_5_entry(ULONG thread_input)
{
    UINT status;
    ULONG actual_flags;


    /* This thread simply waits for an event in a forever loop. */
    while(1)
    {
        /* Increment the thread counter. */
        thread_5_counter++;

        /* Wait for event flag 0. */
        status = tx_event_flags_get(&event_flags_0, 0x1, TX_OR_CLEAR,
            &actual_flags, TX_WAIT_FOREVER);

        /* Check status. */
        if ((status != TX_SUCCESS) || (actual_flags != 0x1))
            break;
    }
}

void thread_6_and_7_entry(ULONG thread_input)
{
    UINT status;

    /* This function is executed from thread 6 and thread 7. As the loop
        below shows, these function compete for ownership of mutex_0. */
    while(1)
    {
        /* Increment the thread counter. */
        if (thread_input == 6)
```

```
        if (thread_input == 0)
            thread_6_counter++;
        else
            thread_7_counter++;

        /* Get the mutex with suspension. */
        status = tx_mutex_get(&mutex_0, TX_WAIT_FOREVER);

        /* Check status. */
        if (status != TX_SUCCESS)
            break;

        /* Get the mutex again with suspension. This shows
            that an owning thread may retrieve the mutex it
            owns multiple times. */
        status = tx_mutex_get(&mutex_0, TX_WAIT_FOREVER);

        /* Check status. */
        if (status != TX_SUCCESS)
            break;

        /* Sleep for 2 ticks to hold the mutex. */
        tx_thread_sleep(2);

        /* Release the mutex. */
        status = tx_mutex_put(&mutex_0);

        /* Check status. */
        if (status != TX_SUCCESS)
            break;

        /* Release the mutex again. This will actually
            release ownership since it was obtained twice. */
        status = tx_mutex_put(&mutex_0);

        /* Check status. */
        if (status != TX_SUCCESS)
            break;
    }
}
```

附录 A - Azure RTOS ThreadX API 服务

2021/4/29 •

Entry 函数

```
VOID      tx_kernel_enter(VOID);
```

块内存服务

```
UINT      tx_block_allocate(TX_BLOCK_POOL *pool_ptr,
            VOID **block_ptr, ULONG wait_option);
```

```
UINT      tx_block_pool_create(TX_BLOCK_POOL *pool_ptr,
            CHAR *name_ptr, ULONG block_size,
            VOID *pool_start, ULONG pool_size);
```

```
UINT      tx_block_pool_delete(TX_BLOCK_POOL *pool_ptr);
```

```
UINT      tx_block_pool_info_get(TX_BLOCK_POOL *pool_ptr,
            CHAR **name,
            ULONG *available_blocks, ULONG *total_blocks,
            TX_THREAD **first_suspended,
            ULONG *suspended_count,
            TX_BLOCK_POOL **next_pool);
```

```
UINT      tx_block_pool_performance_info_get(TX_BLOCK_POOL *pool_ptr,
            ULONG *allocates, ULONG *releases, ULONG *suspensions,
            ULONG *timeouts);
```

```
UINT      tx_block_pool_performance_system_info_get(ULONG *allocates,
            ULONG *releases, ULONG *suspensions, ULONG *timeouts);
```

```
UINT      tx_block_pool_prioritize(TX_BLOCK_POOL *pool_ptr);
```

```
UINT      tx_block_release(VOID *block_ptr);
```

字节内存服务

```
UINT      tx_byte_allocate(TX_BYTE_POOL *pool_ptr,
              VOID **memory_ptr,
              ULONG memory_size, ULONG wait_option);
```

```
UINT      tx_byte_pool_create(TX_BYTE_POOL *pool_ptr,
              CHAR *name_ptr,
              VOID *pool_start, ULONG pool_size);
```

```
UINT      tx_byte_pool_delete(TX_BYTE_POOL *pool_ptr);
```

```
UINT      tx_byte_pool_info_get(TX_BYTE_POOL *pool_ptr,
              CHAR **name, ULONG *available_bytes,
              ULONG *fragments, TX_THREAD **first_suspended,
              ULONG *suspended_count,
              TX_BYTE_POOL **next_pool);
```

```
UINT      tx_byte_pool_performance_info_get(TX_BYTE_POOL *pool_ptr,
              ULONG *allocates,
              ULONG *releases, ULONG *fragments_searched, ULONG *merges,
              ULONG *splits, ULONG *suspensions, ULONG *timeouts);
```

```
UINT      tx_byte_pool_performance_system_info_get(ULONG *allocates,
              ULONG *releases, ULONG *fragments_searched, ULONG *merges,
              ULONG *splits, ULONG *suspensions, ULONG *timeouts);
```

```
UINT      tx_byte_pool_prioritize(TX_BYTE_POOL *pool_ptr);
```

```
UINT      tx_byte_release(VOID *memory_ptr);
```

事件标志服务

```
UINT      tx_event_flags_create(TX_EVENT_FLAGS_GROUP *group_ptr,
            CHAR *name_ptr);
```

```
UINT      tx_event_flags_delete(TX_EVENT_FLAGS_GROUP *group_ptr);
```

```
UINT      tx_event_flags_get(TX_EVENT_FLAGS_GROUP *group_ptr,
            ULONG requested_flags, UINT get_option,
            ULONG *actual_flags_ptr, ULONG wait_option);
```

```
UINT      tx_event_flags_info_get(TX_EVENT_FLAGS_GROUP *group_ptr,
            CHAR **name, ULONG *current_flags,
            TX_THREAD **first_suspended,
            ULONG *suspended_count,
            TX_EVENT_FLAGS_GROUP **next_group);
```

```
UINT      tx_event_flags_performance_info_get(TX_EVENT_FLAGS_GROUP
            *group_ptr, ULONG *sets, ULONG *gets, ULONG *suspensions,
            ULONG *timeouts);
```

```
UINT      tx_event_flags_performance_system_info_get(ULONG *sets,
            ULONG *gets,
            ULONG *suspensions, ULONG *timeouts);
```

```
UINT      tx_event_flags_set(TX_EVENT_FLAGS_GROUP *group_ptr,
            ULONG flags_to_set, UINT set_option);
```

```
UINT      tx_event_flags_set_notify(TX_EVENT_FLAGS_GROUP *group_ptr,
            VOID (*events_set_notify)(TX_EVENT_FLAGS_GROUP *));
```

中断控制

```
UINT      tx_interrupt_control(UINT new_posture);
```

互斥服务

```
UINT       tx_mutex_create(TX_MUTEX *mutex_ptr, CHAR *name_ptr,
              UINT inherit);
```

```
UINT       tx_mutex_delete(TX_MUTEX *mutex_ptr);
```

```
UINT       tx_mutex_get(TX_MUTEX *mutex_ptr, ULONG wait_option);
```

```
UINT       tx_mutex_info_get(TX_MUTEX *mutex_ptr, CHAR **name,
              ULONG *count, TX_THREAD **owner,
              TX_THREAD **first_suspended,
              ULONG *suspended_count,
              TX_MUTEX **next_mutex);
```

```
UINT       tx_mutex_performance_info_get(TX_MUTEX *mutex_ptr, ULONG
              *puts, ULONG *gets, ULONG *suspensions, ULONG *timeouts,
              ULONG *inversions, ULONG *inheritances);
```

```
UINT       tx_mutex_performance_system_info_get(ULONG *puts, ULONG
              *gets,
              ULONG *suspensions, ULONG *timeouts, ULONG *inversions,
              ULONG *inheritances);
```

```
UINT       tx_mutex_prioritize(TX_MUTEX *mutex_ptr);
```

```
UINT       tx_mutex_put(TX_MUTEX *mutex_ptr);
```

队列服务

```
UINT       tx_queue_create(TX_QUEUE *queue_ptr, CHAR *name_ptr,
              UINT message_size, VOID *queue_start,
              ULONG queue_size);
```

```
UINT       tx_queue_delete(TX_QUEUE *queue_ptr);
```

```
UINT       tx_queue_flush(TX_QUEUE *queue_ptr);
```

```
UINT       tx_queue_front_send(TX_QUEUE *queue_ptr, VOID *source_ptr,
              ULONG wait_option);
```

```
UINT       tx_queue_info_get(TX_QUEUE *queue_ptr, CHAR **name,
              ULONG *enqueued, ULONG *available_storage,
              TX_THREAD **first_suspended,
              ULONG *suspended_count, TX_QUEUE **next_queue);
```

```
UINT       tx_queue_performance_info_get(TX_QUEUE *queue_ptr,
              ULONG *messages_sent, ULONG *messages_received,
              ULONG *empty_suspensions, ULONG *full_suspensions,
              ULONG *full_errors, ULONG *timeouts);
```

```
UINT       tx_queue_performance_system_info_get(ULONG *messages_sent,
              ULONG *messages_received, ULONG *empty_suspensions,
              ULONG *full_suspensions, ULONG *full_errors,
              ULONG *timeouts);
```

```
UINT       tx_queue_prioritize(TX_QUEUE *queue_ptr);
```

```
UINT       tx_queue_receive(TX_QUEUE *queue_ptr,
              VOID *destination_ptr, ULONG wait_option);
```

```
UINT       tx_queue_send(TX_QUEUE *queue_ptr, VOID *source_ptr,
              ULONG wait_option);
```

```
UINT       tx_queue_send_notify(TX_QUEUE *queue_ptr, VOID
              (*queue_send_notify)(TX_QUEUE *));
```

信号灯服务

```
UINT       tx_semaphore_ceiling_put(TX_SEMAPHORE *semaphore_ptr,
              ULONG ceiling);
```

```
UINT       tx_semaphore_create(TX_SEMAPHORE *semaphore_ptr,
              CHAR *name_ptr, ULONG initial_count);
```

```
UINT       tx_semaphore_delete(TX_SEMAPHORE *semaphore_ptr);
```

```
UINT       tx_semaphore_get(TX_SEMAPHORE *semaphore_ptr,
              ULONG wait_option);
```

```
UINT       tx_semaphore_info_get(TX_SEMAPHORE *semaphore_ptr, CHAR **name,
              ULONG *current_value,
              TX_THREAD **first_suspended,
              ULONG *suspended_count,
              TX_SEMAPHORE **next_semaphore);
```

```
UINT       tx_semaphore_performance_info_get(TX_SEMAPHORE *semaphore_ptr,
              ULONG *puts, ULONG *gets, ULONG *suspensions,
              ULONG *timeouts);
```

```
UINT       tx_semaphore_performance_system_info_get(ULONG *puts,
              ULONG *gets, ULONG *suspensions, ULONG *timeouts);
```

```
UINT       tx_semaphore_prioritize(TX_SEMAPHORE *semaphore_ptr);
```

```
UINT       tx_semaphore_put(TX_SEMAPHORE *semaphore_ptr);
```

```
UINT       tx_semaphore_put_notify(TX_SEMAPHORE *semaphore_ptr,
              VOID (*semaphore_put_notify)(TX_SEMAPHORE *));
```

线程控制服务

```
UINT       tx_thread_create(TX_THREAD *thread_ptr,
              CHAR *name_ptr,
              VOID (*entry_function)(ULONG), ULONG entry_input,
              VOID *stack_start, ULONG stack_size,
              UINT priority, UINT preempt_threshold,
              ULONG time_slice, UINT auto_start);
```

```
UINT       tx_thread_delete(TX_THREAD *thread_ptr);
```

```
UINT       tx_thread_entry_exit_notify(TX_THREAD *thread_ptr,
              VOID (*thread_entry_exit_notify)(TX_THREAD *, UINT));
```

```
TX_THREAD      *tx_thread_identify(VOID);
```

```
UINT       tx_thread_info_get(TX_THREAD *thread_ptr, CHAR **name,
              UINT *state, ULONG *run_count, UINT *priority,
              UINT *preemption_threshold, ULONG *time_slice,
              TX_THREAD **next_thread,
              TX_THREAD **next_suspended_thread);
```

```
UINT       tx_thread_performance_info_get(TX_THREAD *thread_ptr,
              ULONG *resumptions, ULONG *suspensions,
              ULONG *solicited_preemptions,
              ULONG *interrupt_preemptions,
              ULONG *priority_inversions,ULONG *time_slices, ULONG
              *relinquishes, ULONG *timeouts,
              ULONG *wait_aborts, TX_THREAD **last_preempted_by);
```

```
UINT       tx_thread_performance_system_info_get(ULONG *resumptions,
              ULONG *suspensions,
              ULONG *solicited_preemptions,
              ULONG *interrupt_preemptions,
              ULONG *priority_inversions,ULONG *time_slices, ULONG
              *relinquishes, ULONG *timeouts,
              ULONG *wait_aborts, ULONG *non_idle_returns,
              ULONG *idle_returns);
```

```
UINT       tx_thread_preemption_change(TX_THREAD *thread_ptr,
              UINT new_threshold, UINT *old_threshold);
```

```
UINT       tx_thread_priority_change(TX_THREAD *thread_ptr,
              UINT new_priority, UINT *old_priority);
```

```
VOID       tx_thread_relinquish(VOID);
```

```
UINT       tx_thread_reset(TX_THREAD *thread_ptr);
```

```
UINT       tx_thread_resume(TX_THREAD *thread_ptr);
```

```
UINT       tx_thread_sleep(ULONG timer_ticks);
```

```
UINT       tx_thread_stack_error_notify
              VOID(*stack_error_handler)(TX_THREAD *));
```

```
UINT       tx_thread_suspend(TX_THREAD *thread_ptr);
```

```
UINT       tx_thread_terminate(TX_THREAD *thread_ptr);
```

```
UINT       tx_thread_time_slice_change(TX_THREAD *thread_ptr,
              ULONG new_time_slice, ULONG *old_time_slice);
```

```
UINT       tx_thread_wait_abort(TX_THREAD *thread_ptr);
```

时间服务

```
ULONG      tx_time_get(VOID);
VOID       tx_time_set(ULONG new_time);
```

计时器服务

```
UINT       tx_timer_activate(TX_TIMER *timer_ptr);
```

```
UINT       tx_timer_change(TX_TIMER *timer_ptr,
              ULONG initial_ticks,
              ULONG reschedule_ticks);
```

```
UINT       tx_timer_create(TX_TIMER *timer_ptr,
              CHAR *name_ptr,
              VOID (*expiration_function)(ULONG),
              ULONG expiration_input, ULONG initial_ticks,
              ULONG reschedule_ticks, UINT auto_activate);
```

```
UINT       tx_timer_deactivate(TX_TIMER *timer_ptr);
```

```
UINT       tx_timer_delete(TX_TIMER *timer_ptr);
```

```
UINT       tx_timer_info_get(TX_TIMER *timer_ptr, CHAR **name,
              UINT *active, ULONG *remaining_ticks,
              ULONG *reschedule_ticks,
              TX_TIMER **next_timer);
```

```
UINT       tx_timer_performance_info_get(TX_TIMER *timer_ptr,
              ULONG *activates,
              ULONG *reactivates, ULONG *deactivates,
              ULONG *expirations,
              ULONG *expiration_adjusts);
```

```
UINT       tx_timer_performance_system_info_get
              ULONG *activates, ULONG *reactivates,
              ULONG *deactivates, ULONG *expirations,
              ULONG *expiration_adjusts);
```

附录 B - Azure RTOS ThreadX 常数

2021/4/30 •

字母顺序列表

名称	值
TX_1_ULONG	1
TX_2_ULONG	2
TX_4_ULONG	4
TX_8_ULONG	8
TX_16_ULONG	16
TX_ACTIVATE_ERROR	0x17
TX_AND	2
TX_AND_CLEAR	3
TX_AUTO_ACTIVATE	1
TX_AUTO_START	1
TX_BLOCK_MEMORY	8
TX_BYTE_MEMORY	9
TX_CALLER_ERROR	0x13
TX_CEILING_EXCEEDED	0x21
TX_COMPLETED	1
TX_DELETE_ERROR	0x11
TX_DELETED	0x01
TX_DONT_START	0
TX_EVENT_FLAG	7

tt(ttttt)	"t"
TX_FALSE	0
TX_FEATURE_NOT_ENABLED	0xFF
TX_FILE	11
TX_GROUP_ERROR	0x06
TX_INHERIT	1
TX_INHERIT_ERROR	0x1F
TX_INVALID_CEILING	0x22
TX_IO_DRIVER	10
TX_LOOP_FOREVER	1
TX_MUTEX_ERROR	0x1C
TX_MUTEX_SUSP	13
TX_NO_ACTIVATE	0
TX_NO_EVENTS	0x07
TX_NO_INHERIT	0
TX_NO_INSTANCE	0x0D
TX_NO_MEMORY	0x10
TX_NO_TIME_SLICE	0
TX_NO_WAIT	0
TX_NOT_AVAILABLE	0x1D
TX_NOT_DONE	0x20
TX_NOT_OWNED	0x1E
TX_NULL	0
TX_OPTION_ERROR	0x08
TX_OR	0
TX_OR_CLEAR	1

tt(ttttt)	"t"
TX_POOL_ERROR	0x02
TX_PRIORITY_ERROR	0x0F
TX_PTR_ERROR	0x03
TX_QUEUE_EMPTY	0x0A
TX_QUEUE_ERROR	0x09
TX_QUEUE_FULL	0x0B
TX_QUEUE_SUSP	5
TX_READY	0
TX_RESUME_ERROR	0x12
TX_SEMAPHORE_ERROR	0x0C
TX_SEMAPHORE_SUSP	6
TX_SIZE_ERROR	0x05
TX_SLEEP	4
TX_STACK_FILL	0xEFEFEFEFUL
TX_START_ERROR	0x10
TX_SUCCESS	0x00
TX_SUSPEND_ERROR	0x14
TX_SUSPEND_LIFTED	0x19
TX_SUSPENDED	3
TX_TCP_IP	12
TX_TERMINATED	2
TX_THREAD_ENTRY	0
TX_THREAD_ERROR	0x0E
TX_THREAD_EXIT	1
TX_THRESH_ERROR	0x18

tt(ttttt)	"t"
TX_TICK_ERROR	0x16
TX_TIMER_ERROR	0x15
TX_TRUE	1
TX_WAIT_ABORT_ERROR	0x1B
TX_WAIT_ABORTED	0x1A
TX_WAIT_ERROR	0x04
TX_WAIT_FOREVER	0xFFFFFFFFUL

按值列出

tt(tt)	"t"
TX_DONT_START	0
TX_FALSE	0
TX_NO_ACTIVATE	0
TX_NO_INHERIT	0
TX_NO_TIME_SLICE	0
TX_NO_WAIT	0
TX_NULL	0
TX_OR	0
TX_READY	0
TX_SUCCESS	0x00
TX_THREAD_ENTRY	0
TX_1_ULONG	1
TX_AUTO_ACTIVATE	1
TX_AUTO_START	1
TX_COMPLETED	1

ıı(ıı)	“ı”
TX_INHERIT	1
TX_LOOP_FOREVER	1
TX_DELETED	0x01
TX_OR_CLEAR	1
TX_THREAD_EXIT	1
TX_TRUE	1
TX_2_ULONG	2
TX_AND	2
TX_POOL_ERROR	0x02
TX_TERMINATED	2
TX_AND_CLEAR	3
TX_PTR_ERROR	0x03
TX_SUSPENDED	3
TX_4_ULONG	4
TX_SLEEP	4
TX_WAIT_ERROR	0x04
TX_QUEUE_SUSP	5
TX_SIZE_ERROR	0x05
TX_GROUP_ERROR	0x06
TX_SEMAPHORE_SUSP	6
TX_EVENT_FLAG	7
TX_NO_EVENTS	0x07
TX_8_ULONG	8
TX_BLOCK_MEMORY	8
TX_OPTION_ERROR	0x08

ıı(ıı)	“ı”
TX_BYTE_MEMORY	9
TX_QUEUE_ERROR	0x09
TX_IO_DRIVER	10
TX_QUEUE_EMPTY	0x0A
TX_FILE	11
TX_QUEUE_FULL	0x0B
TX_TCP_IP	12
TX_SEMAPHORE_ERROR	0x0C
TX_MUTEX_SUSP	13
TX_NO_INSTANCE	0x0D
TX_THREAD_ERROR	0x0E
TX_PRIORITY_ERROR	0x0F
TX_16_ULONG	16
TX_NO_MEMORY	0x10
TX_START_ERROR	0x10
TX_DELETE_ERROR	0x11
TX_RESUME_ERROR	0x12
TX_CALLER_ERROR	0x13
TX_SUSPEND_ERROR	0x14
TX_TIMER_ERROR	0x15
TX_TICK_ERROR	0x16
TX_ACTIVATE_ERROR	0x17
TX_THRESH_ERROR	0x18
TX_SUSPEND_LIFTED	0x19
TX_WAIT_ABORTED	0x1A

ττ(ττ)	"τ"
TX_WAIT_ABORT_ERROR	0x1B
TX_MUTEX_ERROR	0x1C
TX_NOT_AVAILABLE	0x1D
TX_NOT_OWNED	0x1E
TX_INHERIT_ERROR	0x1F
TX_NOT_DONE	0x20
TX_CEILING_EXCEEDED	0x21
TX_INVALID_CEILING	0x22
TX_FEATURE_NOT_ENABLED	0xFF
TX_STACK_FILL	0xEFEFEFEFUL
TX_WAIT_FOREVER	0xFFFFFFFFUL

附录 C - Azure RTOS ThreadX 数据类型

2021/4/29 •

TX_BLOCK_POOL

```
typedef struct TX_BLOCK_POOL_STRUCT
{
  ULONG tx_block_pool_id;
  CHAR *tx_block_pool_name;
  ULONG tx_block_pool_available;
  ULONG tx_block_pool_total;
  UCHAR *tx_block_pool_available_list;
  UCHAR *tx_block_pool_start;
  ULONG tx_block_pool_size;
  ULONG tx_block_pool_block_size;
  struct TX_THREAD_STRUCT
  *tx_block_pool_suspension_list;
  ULONG tx_block_pool_suspended_count;
  struct TX_BLOCK_POOL_STRUCT
  *tx_block_pool_created_next,
  *tx_block_pool_created_previous;

  #ifdef TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO
  ULONG tx_block_pool_performance_allocate_count;
  ULONG tx_block_pool_performance_release_count;
  ULONG tx_block_pool_performance_suspension_count;
  ULONG tx_block_pool_performance_timeout_count;
  #endif
  TX_BLOCK_POOL_EXTENSION /* Port defined */
}
```

TX_BYTE_POOL

```
typedef struct TX_BYTE_POOL_STRUCT
{
  ULONG tx_byte_pool_id;
  CHAR *tx_byte_pool_name;
  ULONG tx_byte_pool_available;
  ULONG tx_byte_pool_fragments;
  UCHAR *tx_byte_pool_list;
  UCHAR *tx_byte_pool_search;
  UCHAR *tx_byte_pool_start;
  ULONG tx_byte_pool_size;
  struct TX_THREAD_STRUCT
  *tx_byte_pool_owner;
  struct TX_THREAD_STRUCT
  *tx_byte_pool_suspension_list;
  ULONG tx_byte_pool_suspended_count;
  struct TX_BYTE_POOL_STRUCT
  *tx_byte_pool_created_next,
  *tx_byte_pool_created_previous;

  #ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO
  ULONG tx_byte_pool_performance_allocate_count;
  ULONG tx_byte_pool_performance_release_count;
  ULONG tx_byte_pool_performance_merge_count;
  ULONG tx_byte_pool_performance_split_count;
  ULONG tx_byte_pool_performance_search_count;
  ULONG tx_byte_pool_performance_suspension_count;
  ULONG tx_byte_pool_performance_timeout_count;
  #endif
  TX_BYTE_POOL_EXTENSION /* Port defined */
}
```

TX_EVENT_FLAGS_GROUP

```
typedef struct TX_EVENT_FLAGS_GROUP_STRUCT
{
    ULONG tx_event_flags_group_id;
    CHAR *tx_event_flags_group_name;
    ULONG tx_event_flags_group_current;
    UINT tx_event_flags_group_reset_search;
    struct TX_THREAD_STRUCT *tx_event_flags_group_suspension_list;
    ULONG tx_event_flags_group_suspended_count;
    struct TX_EVENT_FLAGS_GROUP_STRUCT *tx_event_flags_group_created_next,
*tx_event_flags_group_created_previous;
    ULONG tx_event_flags_group_delayed_clear;

#ifdef TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO
    ULONG tx_event_flags_group_performance_set_count;
    ULONG tx_event_flags_group__performance_get_count;
    ULONG tx_event_flags_group___performance_suspension_count;
    ULONG tx_event_flags_group____performance_timeout_count;
#endif

#ifndef TX_DISABLE_NOTIFY_CALLBACKS
    VOID (*tx_event_flags_group_set_notify)(struct TX_EVENT_FLAGS_GROUP_STRUCT);
#endif
    TX_EVENT_FLAGS_GROUP_EXTENSION /* Port defined */
}
```

TX_MUTEX

```
typedef struct TX_MUTEX_STRUCT
{
    ULONG tx_mutex_id;
    CHAR *tx_mutex_name;
    ULONG tx_mutex_ownership_count;
    TX_THREAD *tx_mutex_owner;
    UINT tx_mutex_inherit;
    UINT tx_mutex_original_priority;
    struct TX_THREAD_STRUCT *tx_mutex_suspension_list;
    ULONG tx_mutex_suspended_count;
    struct TX_MUTEX_STRUCT *tx_mutex_created_next, *tx_mutex_created_previous;
    ULONG tx_mutex_highest_priority_waiting;
    struct TX_MUTEX_STRUCT *tx_mutex_owned_next, *tx_mutex_owned_previous;

#ifdef TX_MUTEX_ENABLE_PERFORMANCE_INFO
    ULONG tx_mutex_performance_put_count;
    ULONG tx_mutex_performance_get_count;
    ULONG tx_mutex_performance_suspension_count;
    ULONG tx_mutex_performance_timeout_count;
    ULONG tx_mutex_performance_priority_inversion_count;
    ULONG tx_mutex_performance__priority_inheritance_count;
#endif

    TX_MUTEX_EXTENSION /* Port defined */
}
```

TX_QUEUE

```
typedef struct TX_QUEUE_STRUCT
{
    ULONG tx_queue_id;
    CHAR *tx_queue_name;
    UINT tx_queue_message_size;
    ULONG tx_queue_capacity;
    ULONG tx_queue_enqueued;
    ULONG tx_queue_available_storage;
    ULONG *tx_queue_start;
    ULONG *tx_queue_end;
    ULONG *tx_queue_read;
    ULONG *tx_queue_write;
    struct TX_THREAD_STRUCT *tx_queue_suspension_list;
    ULONG tx_queue_suspended_count;
    struct TX_QUEUE_STRUCT *tx_queue_created_next, *tx_queue_created_previous;

#ifdef TX_QUEUE_ENABLE_PERFORMANCE_INFO
    ULONG tx_queue_performance_messages_sent_count;
    ULONG tx_queue_performance_messages_received_count;
    ULONG tx_queue_performance_empty_suspension_count;
    ULONG tx_queue_performance_full_suspension_count;
    ULONG tx_queue_performance_full_error_count;
    ULONG tx_queue_performance_timeout_count;
#endif

#ifndef TX_DISABLE_NOTIFY_CALLBACKS
    VOID *tx_queue_send_notify)(struct TX_QUEUE_STRUCT *);
#endif

    TX_QUEUE_EXTENSION /* Port defined */
}
```

TX_SEMAPHORE

```
typedef struct TX_SEMAPHORE_STRUCT
{
    ULONG tx_semaphore_id;
    CHAR *tx_semaphore_name;
    ULONG tx_semaphore_count;
    struct TX_THREAD_STRUCT *tx_semaphore_suspension_list;
    ULONG tx_semaphore_suspended_count;
    struct TX_SEMAPHORE_STRUCT *tx_semaphore_created_next, *tx_semaphore_created_previous;

#ifdef TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO
    ULONG tx_semaphore_performance_put_count;
    ULONG tx_semaphore_performance_get_count;
    ULONG tx_semaphore_performance_suspension_count;
    ULONG tx_semaphore_performance_timeout_count;
#endif

#ifndef TX_DISABLE_NOTIFY_CALLBACKS
    VOID (*tx_semaphore_put_notify)(struct TX_SEMAPHORE_STRUCT *);
#endif

    TX_SEMAPHORE_EXTENSION /* Port defined */
}
```

TX_THREAD

```
typedef struct TX_THREAD_STRUCT
{
    ULONG tx_thread_id;
    ULONG tx_thread_run_count;
    VOID *tx_thread_stack_ptr;
    VOID *tx_thread_stack_start;
    VOID *tx_thread_stack_end;
    ULONG tx_thread_stack_size;
    ULONG tx_thread_time_slice;
    ULONG tx_thread_new_time_slice;
    struct TX_THREAD_STRUCT *tx_thread_ready_next, *tx_thread_ready_previous;
    TX_THREAD_EXTENSION_0 /* Port defined */
    CHAR *tx_thread_name;
    UINT tx_thread_priority;
    UINT tx_thread_state;
    UINT tx_thread_delayed_suspend;
    UINT tx_thread_suspending;
    UINT tx_thread_preempt_threshold;
    VOID (*tx_thread_schedule_hook)(struct TX_THREAD_STRUCT *, ULONG);
    VOID (*tx_thread_entry)(ULONG);
    ULONG tx_thread_entry_parameter;
    TX_TIMER_INTERNAL tx_thread_timer;
    VOID (*tx_thread_suspend_cleanup)(struct TX_THREAD_STRUCT *);
    VOID *tx_thread_suspend_control_block;
    struct TX_THREAD_STRUCT *tx_thread_suspended_next, *tx_thread_suspended_previous;
    ULONG tx_thread_suspend_info;
    VOID *tx_thread_additional_suspend_info;
    UINT tx_thread_suspend_option;
    UINT tx_thread_suspend_status;
    TX_THREAD_EXTENSION_1 /* Port defined */
    struct TX_THREAD_STRUCT *tx_thread_created_next, *tx_thread_created_previous;
    TX_THREAD_EXTENSION_2 /* Port defined */
    VOID *tx_thread_filex_ptr;
    UINT tx_thread_user_priority;
    UINT tx_thread_user_preempt_threshold;
    UINT tx_thread_inherit_priority;
    ULONG tx_thread_owned_mutex_count;
    struct TX_MUTEX_STRUCT *tx_thread_owned_mutex_list;

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO
    ULONG tx_thread_performance_resume_count;
    ULONG tx_thread_performance_suspend_count;
    ULONG tx_thread_performance_solicited_preemption_count;
    ULONG tx_thread_performance_interrupt_preemption_count;
    ULONG tx_thread_performance_priority_inversion_count;
    struct TX_THREAD_STRUCT *tx_thread_performance_last_preempting_thread;
    ULONG tx_thread_performance_time_slice_count;
    ULONG tx_thread_performance_relinquish_count;
    ULONG tx_thread_performance_timeout_count;
    ULONG tx_thread_performance_wait_abort_count;
#endif

    VOID *tx_thread_stack_highest_ptr;

#ifndef TX_DISABLE_NOTIFY_CALLBACKS
    VOID (*tx_thread_entry_exit_notify) (struct TX_THREAD_STRUCT *, UINT);
#endif

    TX_THREAD_EXTENSION_3 /* Port defined */
    ULONG tx_thread_suspension_sequence;
    TX_THREAD_USER_EXTENSION
}
```

TX_TIMER

```
typedef struct TX_TIMER_STRUCT
{
    ULONG tx_timer_id;
    CHAR *tx_timer_name;
    TX_TIMER_INTERNAL tx_timer_internal;
    struct TX_TIMER_STRUCT *tx_timer_created_next, *tx_timer_created_previous;
    TX_TIMER_EXTENSION /* Port defined */

#ifdef TX_TIMER_ENABLE_PERFORMANCE_INFO
    ULONG tx_timer_performance_activate_count;
    ULONG tx_timer_performance_reactivate_count;
    ULONG tx_timer_performance_deactivate_count;
    ULONG tx_timer_performance_expiration_count;
    ULONG tx_timer_performance__expiration_adjust_count;
#endif
}
```

TX_TIMER_INTERNAL

```
typedef struct TX_TIMER_INTERNAL_STRUCT
{
    ULONG tx_timer_internal_remaining_ticks;
    ULONG tx_timer_internal_re_initialize_ticks;
    VOID (*tx_timer_internal_timeout_function)(ULONG);
    ULONG tx_timer_internal_timeout_param;
    struct TX_TIMER_INTERNAL_STRUCT *tx_timer_internal_active_next, *tx_timer_internal_active_previous;
    struct TX_TIMER_INTERNAL_STRUCT *tx_timer_internal_list_head;
    TX_TIMER_INTERNAL_EXTENSION /* Port defined */
}
```

附录 D - Azure RTOS ThreadX ASCII 字符代码

2021/4/30 •

十六进制 ASCII 字符代码

least significant nibble \ most significant nibble	0_	1_	2_	3_	4_	5_	6_	7_
_0	NUL	DLE	SP	0	@	P	'	p
_1	SOH	DC1	!	1	A	Q	a	q
_2	STX	DC2	"	2	B	R	b	r
_3	ETX	DC3	#	3	C	S	c	s
_4	EOT	DC4	$	4	D	T	d	t
_5	ENQ	NAK	%	5	E	U	e	u
_6	ACK	SYN	&	6	F	V	f	v
_7	BEL	ETB	'	7	G	W	g	w
_8	BS	CAN	(	8	H	X	h	x
_9	HT	EM	)	9	I	Y	i	y
_A	LF	SUB	*	:	J	Z	j	z
_B	VT	ESC	+	;	K	[	K	}
_C	FF	FS	,	<	L	\	l	|
_D	CR	GS	-	=	M	]	m	}
_E	SO	RS	.	>	N	^	n	~
_F	SI	US	/	?	O	_	o	DEL

第 1 章 - 概述

2021/4/29 •

ARMv8-M 体系结构引入了新的安全功能，包括 TrustZone，它允许将内存标记为安全或不安全。按照 ARM 的指导原则，ThreadX（和用户应用程序）设计为在非安全模式下运行。也可以在安全模式下运行 ThreadX（和用户应用程序）。为了与安全模式软件进行交互，需要一些新的 ThreadX API。本文档介绍了特定于 ARMv8-M 体系结构的这些 ThreadX 服务，其中包括 Cortex-M23、Cortex-M33、Cortex-M35P 和 Cortex-M55。

第 2 章 - 安装对 ARMv8-M 的 ThreadX 支持

2021/4/29 •

有额外的 ThreadX 源代码文件可支持 ARMv8-M 体系结构。

如果 ThreadX 将在安全模式下运行，则不需要这些额外的文件和 API。若要在安全模式下运行 ThreadX，请在 *tx_port.h _ 顶部或者在命令行或项目设置中定义符号 TX_SECURE_EXECUTION。确保为所有 c 和程序集文件定义了 _* TX_SECURE_EXECUTION。ThreadX 和用户应用程序将在安全模式下执行。

若要在非安全模式下运行 ThreadX 和用户应用程序，并支持不安全的可调用安全函数，请执行以下操作。

必须将文件 *tx_thread_secure_stack.c* 添加到安全的应用程序。

必须将以下文件添加到 ThreadX 库中。

- *tx_secure_interface.h*
- *txe_thread_secure_stack_allocate.c*
- *txe_thread_secure_stack_free.c*
- *tx_thread_secure_stack_allocate.s*
- *tx_thread_secure_stack_free.s*

下面两个文件替换 ThreadX 库中的通用文件。

- *tx_thread_stack_error_handler.c*
- *tx_thread_stack_error_notify.c*

ARMv8-M 的其他 ThreadX 源

下面介绍了 ARMv8-M TrustZone 体系结构的其他 ThreadX 文件。

CONTENTS	
tx_secure_interface.h	包含定义 ThreadX 非安全可调用函数的文件。
txe_thread_secure_stack_allocate.c	安全堆栈分配 API 的错误检查文件。
txe_thread_secure_stack_free.c	安全堆栈释放 API 的错误检查文件。
tx_thread_secure_stack_allocate.s	适用于安全堆栈分配服务的非安全 veneer。
tx_thread_secure_stack_free.s	适用于安全堆栈释放服务的非安全 veneer。
tx_thread_stack_error_handler.c	线程堆栈错误的处理程序。
tx_thread_stack_error_notify.c	注册用于处理线程堆栈错误的通知回调。

第 3 章 - 适用于 ARMv8-M 的 ThreadX API

2021/4/29 •

本章按字母顺序介绍了特定于 ARMv8-M 的 ThreadX 服务。它们的名称设计将所有相似的服务组合在一起。在以下说明的“返回值”部分中，以粗体显示的值不受用于禁用 API 错误检查的 NX_DISABLE_ERROR_CHECKING 定义影响，而不以粗体显示的值则会被完全禁用。

- tx_thread_secure_stack_allocate 在安全内存中分配线程堆栈。
- tx_thread_secure_stack_free 安全内存中的释放线程堆栈

tx_thread_secure_stack_allocate

在安全内存中分配线程堆栈。

原型

```
UINT tx_thread_secure_stack_allocate(
    TX_THREAD *thread_ptr,
    ULONG stack_size);
```

说明

此服务会在安全内存中分配大小为 stack_size 字节的堆栈。应为调用安全函数的每个线程调用此函数。

输入参数

- thread_ptr 指向之前创建的线程的指针。
- stack_size 安全堆栈的大小。

返回值

- **TX_SUCCESS** (0X00) 成功的请求。
- **TX_SIZE_ERROR** (0X05) 堆栈大小超出范围。
- **TX_THREAD_ERROR** (0X0E) 无效的线程指针。
- **TX_NO_MEMORY** (0X10) 无法分配内存。
- **NX_CALLER_ERROR** (0x13) 此服务的调用方无效。
- **TX_FEATURE_NOT_ENABLED** (0xFF) 系统已被编译为在安全模式下运行。

允许来自

初始化、线程

示例

```
/* Create thread.  */
tx_thread_create(&thread_0, "thread 0", thread_0_entry, 0, pointer, DEMO_STACK_SIZE, 1, 1, TX_NO_TIME_SLICE,
TX_AUTO_START);

/* Allocate secure stack so this thread can call secure functions.  */
status = tx_thread_secure_stack_allocate(&thread_0, 256);

/* If status is TX_SUCCESS the request was successful.  */
```

另请参阅

tx_thread_secure_stack_free

tx_thread_secure_stack_free

在安全内存中释放线程堆栈。

原型

```
UINT tx_thread_secure_stack_free(TX_THREAD *thread_ptr);
```

说明

此服务在安全内存中释放线程的安全堆栈。如果线程具有安全堆栈，并且线程不再需要调用安全函数或已删除线程，则应调用此函数。

输入参数

- thread_ptr 指向之前创建的线程的指针。

返回值

- **TX_SUCCESS** (0X00) 成功的请求。
- **TX_THREAD_ERROR** (0X0E) 无效的线程指针。
- **NX_CALLER_ERROR** (0x13) 此服务的调用方无效。
- **TX_FEATURE_NOT_ENABLED** (0xFF) 系统已被编译为在安全模式下运行。

允许来自

初始化、线程

示例

```
/* Free thread's secure stack.  */
status = tx_thread_secure_stack_free(&thread_0);

/* If status is TX_SUCCESS the request was successful.  */

/* Delete thread.  */
tx_thread_delete(&thread_0);
```

另请参阅

tx_thread_secure_stack_allocate

关于本指南

2021/4/29 •

本指南旨在全面介绍 Azure RTOS ThreadX SMP（Microsoft 的高性能嵌入式实时内核）。

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

组织

1	简要概述 ThreadX SMP 及其与实时嵌入式开发的关系。
2	介绍在应用程序中安装和使用 ThreadX SMP 的基本步骤（开箱即用）。
3	详细介绍 ThreadX SMP（高性能实时 SMP 内核）的功能操作。
4	详细介绍应用程序的 ThreadX SMP 接口。
5	介绍如何为 ThreadX SMP 应用程序编写 I/O 驱动程序。
6	介绍每个 ThreadX SMP 处理器支持包随附的演示应用程序。
A	ThreadX SMP API
B	ThreadX SMP 常量
C	ThreadX SMP 数据类型
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。-

> **IMPORTANT**
> 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。

> **WARNING**
> 警告符号提示开发人员注意应小心避免的情况，这些情况可能导致灾难性错误。

ThreadX SMP 数据类型

ThreadX SMP 中除了自定义的控制结构数据类型之外，还提供了一系列特殊的数据类型，用于 ThreadX SMP 服

务调用接口。这些特殊数据类型会直接与底层 C 编译器的数据类型相映射。这样做是为了确保不同 C 编译器之间的可移植性。有关准确的实现，请参阅分发磁盘随附的 tx_port.h 文件。

ThreadX SMP 服务调用数据类型及其关联含义的列表如下所示：

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

ThreadX SMP 源中还使用了其他数据类型。这些数据类型也位于 tx_port.h 文件中。

客户支持中心

支持电子邮件：azure-rtos-support@microsoft.com 网页：azure.com/rtos

最新产品信息

请访问 azure.com/rtos 网站，然后选择“支持”菜单选项，查找最新的联机支持信息，包括有关最新 ThreadX SMP 产品版本的信息。

我们需要你提供的信息

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 ThreadX SMP 所作的任何更改。
3. 可在分发的 tx_port.h_* 文件中找到的 *tx_version_id** 字符串的内容。此字符串将为我们提供有关运行时环境的重要信息。
4. RAM 中 _tx_build_options ULONG 变量的内容。此变量将为我们提供有关 ThreadX SMP 库生成方式的信息。

如对本指南有任何意见，联系方式如下：

如对本指南有任何意见和建议，请发送电子邮件至客户支持中心（邮件地址为 azure-rtos-support@microsoft.com），并在主题行中输入“ThreadX SMP 用户指南”。

第 1 章：Azure RTOS ThreadX SMP 简介

2021/4/29 •

Azure RTOS ThreadX SMP 是专门为嵌入式应用程序设计的高性能实时 SMP 内核。本章提供了该产品的简介，并说明了其应用和优势。

ThreadX SMP 的独特功能

ThreadX SMP 将对称多处理 (SMP) 技术引入到嵌入式应用程序。“准备好”运行的 ThreadX SMP 应用程序线程（具有不同优先级）在计划期间动态分配给可用处理器核心。这将导致真正的 SMP 处理，包括自动跨所有可用处理器核心对应用程序线程执行进行负载均衡。

与其他实时内核不同，ThreadX SMP 功能多样—通过使用强大的 CISC、RISC 和 DSP 处理器的应用程序，可以轻松地在基于小型微控制器的应用程序之间扩展。

ThreadX SMP 可基于其基础体系结构进行扩展。因为 ThreadX SMP 服务是作为 C 库实现的，所以只有应用程序实际使用的那些服务被引入了运行时映像。因此，ThreadX SMP 的实际大小完全由应用程序决定。对于大多数应用程序，ThreadX SMP 的指令映像的大小在 5 KB 至 20 KB 之间。

picokernel™ 体系结构

ThreadX SMP 服务没有像传统的微内核体系结构那样将内核函数相互层叠，而是直接将其插入核心。由此产生了最快的上下文切换和服务呼叫性能。我们将此非分层设计称为 picokernel 体系结构。

ANSI C 源代码

ThreadX SMP 主要是在 ANSI C 中编写的。用户需要少量的汇编语言来定制内核，从而满足基础目标处理器的需求。此设计使用户可以在很短的时间内（通常在几周内）将 ThreadX SMP 移植到新的处理器系列！

高级技术

下面是 ThreadX SMP 高级技术的亮点：

- 简单的 picokernel 体系结构
- 自动负载均衡
- 每线程处理器排除
- 自动扩展（占用空间少）
- 确定性处理
- 快速实时性能
- 抢先式和协作式计划
- 灵活的线程优先级支持 (32-1024)
- 动态系统对象创建
- 无限量的系统对象
- 经过优化的中断处理
- 抢占阈值 (Preemption-threshold™)
- 优先级继承
- 事件链接 (Event-chaining™)
- 快速软件计时器
- 运行时内存管理
- 运行时性能监视
- 运行时堆栈分析

- 内置系统跟踪
- 广泛的处理器支持
- 广泛的开发工具支持
- 字节顺序完全中性

不是黑盒

ThreadX SMP 的大多数发行版都包含完整的 C 源代码以及特定于处理器的汇编语言。这消除了许多商业内核所出现的"黑盒"问题。借助 ThreadX SMP，应用程序开发人员可以看到内核正在进行的确切操作。没有任何秘密可言！

源代码还允许进行特定于应用程序的修改。尽管不建议这么做，但如果真的需要，具备修改内核的能力当然是有益的。

对于习惯使用自己的内部内核的开发人员而言，这些功能尤其令人欣慰。他们期望拥有源代码和修改内核的能力。ThreadX SMP 是适用于这类开发人员的终极内核。

RTOS 标准

由于 ThreadX SMP 的多功能性、高性能的 picokernel 体系结构、先进的技术以及经证实的可移植性，目前已部署 ThreadX SMP 的设备超过了 20 亿。这实际上使 ThreadX SMP 成为深度嵌入式应用程序的 RTOS 标准。

安全认证

TÜV 认证

ThreadX SMP 已被 SGS-TÜV Saar 认证可用于安全关键型系统，并且符合 IEC61508 和 IEC-62304 标准。该认证证明：ThreadX SMP 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 ThreadX SMP 进行了认证，确定其可用于安全关键型汽车系统。此外，ThreadX SMP 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。

此外，SGS-TÜV Saar 已对 ThreadX SMP 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 azure-rtos-support@microsoft.com，了解 ThreadX SMP 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。

符合 MISRA C

MISRA C 是一组编程准则，面向使用 C 编程语言的关键系统。最初的 MISRA C 准则主要面向汽车业应用；但是，现在人们广泛认可 MISRA C 适用于任何安全关键应用。ThreadX SMP 符合 MISRA-C:2004 和 MISRA C:2012 的所有"必需"规则和"强制性"规则。ThreadX SMP 还符合除三个"公告"规则之外的所有规则。有关更多详细信息，请参阅 ThreadX_MISRA_Compliance.pdf 文档。

UL 认证

ThreadX SMP 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 E607301 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 azure-rtos-support@microsoft.com，了解 ThreadX SMP 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。

认证包

ThreadX SMP Certification Pack™ 是 100% 完整、统包式、行业特定的独立包，提供所有必要的 ThreadX SMP 证据，以认证或成功申报基于 ThreadX SMP 的产品，致力于达到安全关键型航空、医疗和工业系统所需的最高可靠性和关键性等级。支持的认证包括 DO-178B、ED-12B、DO-278、FDA510(k)、IEC-62304、IEC-60601、ISO-14971、UL-1998、IEC-61508、CENELEC EN50128、BS50128 和 49CFR236。有关认证包的更多信息，请联系 sales@expresslogic.com。

嵌入式应用程序

嵌入式应用程序在诸如无线通信设备、汽车引擎、激光打印机、医疗器械等产品内置的微处理器上执行。嵌入式应用程序的另一个区别在于它们的软件和硬件有着特定的用途。

实时软件

当对应用软件施加时间限制时，它被称为实时软件。大致说来，必须在确切的一段时间内执行其处理的软件被称为实时网络软件。由于嵌入式应用程序与外部事件的固有交互，因此几乎总是实时的。

多任务

如前所述，嵌入式应用程序有着特定的用途。为了实现此目的，软件必须执行各种任务。任务是执行特定职责的应用程序的半独立部分。同样，事实上某些任务比其他任务更重要。嵌入式应用程序的主要难题之一是在各种应用程序任务之间分配处理器。在竞争任务之间分配处理器是 ThreadX SMP 的主要目的。

任务与线程

必须进行任务的另一个区别。术语任务是以多种方式使用的。有时它意味着可单独加载的程序。在其他情况下，它可以指内部程序段。

在现代操作系统讨论中，有两个术语或多或少地替代了"任务"："进程"和"线程"。"进程"是具有自己的地址空间的完全独立的程序，而"线程"是在进程内执行的半独立程序段。线程共享相同的进程地址空间。与线程管理相关的开销是最小的。

大多数嵌入式应用程序负担不起与面向进程的成熟操作系统相关的开销(内存和性能)。此外，较小的微处理器并没有支持真正面向进程的操作系统的硬件体系结构。出于这些原因，ThreadX SMP 实现了一个线程模型，对于大多数实时嵌入式应用程序来说，该模型非常高效实用。

为避免混淆，ThreadX SMP 不使用"任务"这个术语。取而代之的是，使用更具描述性和现代性的名称"线程"。

ThreadX SMP 的好处

使用 ThreadX SMP 为嵌入式应用程序带来了许多好处。当然，主要好处在于如何为嵌入式应用程序线程分配处理时间。

自动负载均衡

ThreadX SMP 提供自动负载均衡(跨可用核心的线程执行)执行线程，这样可以尽可能轻松地利用多核处理器。

更高的响应能力

在出现像 ThreadX SMP 这样的实时内核之前，大多数嵌入式应用程序都使用简单的控制循环(通常来自 C main 函数内部)来分配处理时间。在非常小或简单的应用程序中，这种方法仍在使用。但是，在大型或复杂的应用程序中，这是不切实际的，因为对任何事件的响应时间是一个函数，是一次通过控制循环的传递的最差处理时间的函数。

更糟糕的是，每当对控制循环进行修改时，应用程序的时序特性都会发生变化。这使应用程序本身就不稳定，难以维护和改进。

ThreadX SMP 提供对重要外部事件的快速确定性响应时间。ThreadX SMP 通过其基于优先级的抢先式计划算法来实现这一点，该算法允许优先级较高的线程抢先于正在执行的优先级较低的线程。因此，最差的响应时间接近执行上下文切换所需的时间。这不仅是确定性的，而且还是非常快速的。

软件维护

ThreadX SMP 内核使应用程序开发人员能够专注于其应用程序线程的特定要求，而无需担心更改应用程序其他区域的计时。此功能还简化了使用 ThreadX SMP 的应用程序的修复或增强。

更高的吞吐量

解决控制循环响应时间问题的一种方法可以是添加更多轮询。这样可以提高响应能力，但仍不能保证最差响应时间保持不变，而且也不利于推动应用程序的未来修改。而且，由于额外的轮询，处理器现在正在执行更多不必要的处理。所有这些不必要的处理都会降低系统的总体吞吐量。

关于开销，有趣的一点是，许多开发人员都认为多线程环境(如 ThreadX SMP)会增加开销，并对总系统吞吐量产生负面影响。但是在某些情况下，多线程实际上通过消除在控制循环环境中发生的所有冗余轮询减少了开销。与多线程内核相关的开销通常是上下文切换所需时间的函数。如果上下文切换时间少于轮询过程，则 ThreadX SMP 提供的解决方案可能会减少开销并提高吞吐量。无论应用程序的复杂性和大小如何，这都使 ThreadX SMP 成为明智之选。

处理器隔离

ThreadX SMP 在应用程序与基础处理器之间提供可靠的独立于处理器的接口。这使开发人员能够专注于应用程序，而不是花费大量时间研究硬件详细信息。

划分应用程序

在基于控制循环的应用程序中，每个开发人员都必须对整个应用程序的运行时行为和要求有着深入的了解。这是因为处理器分配逻辑分散在整个应用程序中。随着应用程序的大小或复杂性的增加，所有开发人员都无法记住整个应用程序的精确处理要求。

ThreadX SMP 让每个开发人员摆脱了与处理器分配相关的烦恼，并允许他们专注于嵌入式应用程序的特定部分。此外，ThreadX SMP 强制将应用程序划分为明确定义的线程。将应用程序划分为多个线程本身可使开发变得更简单。

易用性

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

缩短上市时间

ThreadX SMP 的这一切优点都加速了软件开发过程。ThreadX SMP 负责处理大多数处理器问题和最常见的安全认证，从而从开发计划中省去了这些工作。所有这些都使上市时间缩短！

保护软件投资

由于其体系结构，ThreadX SMP 可轻松移植到新的处理器和/或开发工具环境。基于这一特性，再加上 ThreadX SMP 可以将应用程序与基础处理器的详细信息隔离的事实，从而使 ThreadX SMP 应用程序变得高度可移植。最终保证了应用程序的迁移路径，并保护了原始开发投资。

第 2 章 - 安装和使用 Azure RTOS ThreadX SMP

2021/4/30 •

本章旨在介绍与安装、设置和使用高性能 Azure RTOS ThreadX SMP 内核相关的各种问题。

主机注意事项

嵌入式软件通常是在 Windows 或 Linux (Unix) 主机计算机上开发的。在对应用程序进行编译和链接并将其放置在主机上之后，将应用程序下载到目标硬件，以执行它。

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

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

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

> **IMPORTANT**
> 请查看提供的 readme_threadx.txt 文件，了解其他主机系统注意事项和选项。

目标注意事项

ThreadX SMP 要求目标具有 2 KB 和 20 KB 只读内存 (ROM)。对于 ThreadX SMP 系统堆栈和其他全局数据结构，它还需要 1 到 2KB 的目标的随机访问内存 (RAM)。

对于与计时器相关的函数（如服务调用超时、时间切片和要运行的应用程序计时器），基础目标硬件必须提供定期中断源。如果处理器具有此功能，则 ThreadX SMP 会使用它。否则，如果目标处理器无法生成定期中断，则用户的硬件必须提供此功能。计时器中断的设置和配置通常位于 ThreadX SMP 分发的 tx_initialize_low_level 程序集文件中。

> **IMPORTANT**
> 即使没有可用的定期计时器中断源，ThreadX SMP 仍可正常运行。但是，与计时器相关的服务都无法正常运行。请查看提供的 readme_threadx.txt 文件，了解任何其他主机系统注意事项和/或选项。

产品分发

分发磁盘的确切内容取决于目标处理器、开发工具和购买的 ThreadX SMP 包。但是，下面列出了大多数产品分发共有的几个重要文件：

ThreadX_Express_Startup.pdf

此 PDF 提供了一个简单的四步过程，能够让 ThreadX SMP 在特定的目标处理器/板和特定的开发工具上运行。

readme_threadx.txt

文本文件，其中包含有关 ThreadX SMP 端口的特定信息（包括有关目标处理器和开发工具的信息）。

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

IMPORTANT

所有文件名均为小写。通过此命名约定可以更轻松地将命令转换为 Linux (Unix) 开发平台的命令。

ThreadX SMP 安装

ThreadX SMP 的安装非常简单。有关针对特定环境安装 ThreadX SMP 的特定信息，请参阅 ThreadX_Express_Startup.pdf 文件和 readme_threadx.txt 文件。

IMPORTANT

请确保备份 ThreadX SMP 分发磁盘，并将其存储在安全的位置。

IMPORTANT

应用程序软件需要访问 ThreadX SMP 库文件（通常为 tx.a 或 tx.lib）和 C 包含文件 tx_api.h 和 tx_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 ThreadX SMP

使用 ThreadX SMP 非常简单。基本上，应用程序代码必须在编译期间包含 tx_api.h，并与 ThreadX SMP 运行时库 tx.a（或tx.lib）链接 。

生成 ThreadX SMP 应用程序需要四个步骤：

在所有使用 ThreadX SMP 服务或数据结构的应用程序文件中包含 tx_api.h 文件。

创建标准 C main 函数。此函数必须最终调用 tx_kernel_enter 才能启动 ThreadX SMP。在输入内核之前，可能会添加不涉及 ThreadX SMP 的应用程序特定的初始化。

IMPORTANT

ThreadX SMP entry 函数 tx_kernel_enter 不返回。因此，请确保在其之后不进行任何处理或函数调用。

创建 tx_application_define 函数。这是创建初始系统资源的位置。系统资源的示例包括线程、队列、内存池、事件标志组、互斥体和信号灯。

编译应用程序源并与 ThreadX SMP 运行时库 tx.lib 链接。生成的映像可下载到目标并执行！

小型示例系统

第 28 页图 1 中的小型示例系统显示了优先级为 3 的单个线程的创建。线程执行，递增计数器，然后休眠一个时钟周期。此过程会永远继续下去。

```
#include              "tx_api.h"

unsigned long         my_thread_counter = 0;
TX_THREAD             my_thread;

main( )
{
      /* Enter the ThreadX SMP kernel. */
      tx_kernel_enter( );
}

void tx_application_define(void *first_unused_memory)
{

      /* Create my_thread! */
      tx_thread_create(&my_thread, "My Thread",
          my_thread_entry, 0x1234, first_unused_memory, 1024,
             3, 3, TX_NO_TIME_SLICE, TX_AUTO_START);
}

void my_thread_entry(ULONG thread_input)
{
      /* Enter into a forever loop. */
      while(1)
      {

            /* Increment thread counter. */
            my_thread_counter++;

            /* Sleep for 1 tick. */
            tx_thread_sleep(1);
      }
}
```

图 1 应用程序开发的模板

尽管这是一个简单的示例，但它为真正的应用程序开发提供了一个很好的模板。同样，有关其他详细信息，请参阅 readme_threadx.txt 文件。

疑难解答

每个 ThreadX SMP 端口都随演示应用程序一起提供。首先运行演示系统，无论是在实际的目标硬件上，还是在模拟环境中，这始终是个不错的主意。

> **IMPORTANT**
> 有关演示系统的详细信息，请参阅随分发提供的 readme_threadx.txt 文件。

如果演示系统未正确执行，以下是一些故障排除提示：

- 确定演示的运行量。
- 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）。
- 重新生成 ThreadX SMP 库，其中定义了 TX_ENABLE_STACK_CHECKING。这将启用内置的 ThreadX SMP 堆栈检查。
- 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。此类信息对于支持工程师非常有用。

按照第 12 页的“我们需要你提供的内容”中所述的过程发送在故障排除步骤中收集的信息。

配置选项

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

> **IMPORTANT**
> 只有当应用程序和 ThreadX SMP 库在构建时定义了 TX_INCLUDE_USER_DEFINE_FILE，才会应用在 tx_user.h 中定义的选项。

最小配置

对于最小的代码大小，应考虑以下 ThreadX SMP 配置选项（缺少所有其他选项）：

- TX_DISABLE_ERROR_CHECKING
- TX_DISABLE_PREEMPTION_THRESHOLD
- TX_DISABLE_NOTIFY_CALLBACKS
- TX_DISABLE_REDUNDANT_CLEARING
- TX_DISABLE_STACK_FILLING
- TX_NOT_INTERRUPTABLE
- TX_TIMER_PROCESS_IN_ISR

最快配置

为实现最快的执行，在以前的最小配置中使用相同的配置选项，但也将此选项考虑在内：

- TX_REACTIVATE_INLINE

有关 ThreadX SMP 特定版本的其他选项，请查看 readme_threadx.txt 文件 ***readme_threadx.txt***。第 28 页开头介绍了详细的配置选项。

全局时间源

对于其他 Azure RTOS 产品（FileX、NetX、GUIX、USBX 等），ThreadX SMP 定义表示一秒的 ThreadX SMP 计时器时钟周期数。其他产品基于此常量派生其时间要求。默认情况下，该值为 100（假设 10ms 定期中断）。用户可以通过在 tx_port.h 或在 IDE 或命令行中以所需值定义 TX_TIMER_TICKS_PER_SECOND 来覆盖此值。

详细的配置选项

- TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO：定义后，可以收集块池的性能信息。默认情况下，未定义此选项。
- TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO：定义后，可以收集字节池的性能信息。默认情况下，未定义此选项。
- TX_DISABLE_ERROR_CHECKING：跳过基本服务调用错误检查。在应用程序源中定义后，将禁用所有基本参数错误检查。这可能会将性能提高多达 30%，还可能会减小图像大小。

> **NOTE**
> 仅当应用程序可以绝对保证所有输入参数（包括从外部输入派生的输入参数）在所有情况下始终有效，才可以安全地禁用错误检查。如果在禁用错误检查的情况下向 API 提供无效输入，则生成的行为是未定义的，并且可能会导致内存损坏或系统崩溃。

> **NOTE**
> 不受禁用错误检查影响的 ThreadX SMP API 返回值在第 4 章的每个 API 说明的“返回值”部分中以粗体列出。如果通过使用 TX_DISABLE_ERROR_CHECKING 选项禁用了错误检查，则非加粗返回值无效。

- TX_DISABLE_NOTIFY_CALLBACKS：定义后，将对各种 ThreadX SMP 对象禁用通知回调。使用此选项可以略微减小代码大小并提高性能。默认情况下，未定义此选项。
- TX_DISABLE_PREEMPTION_THRESHOLD：定义后，将禁用抢占阈值功能，略微减小代码大小并提高性能。当然，抢占阈值功能不再可用。默认情况下，未定义此选项。
- TX_DISABLE_REDUNDANT_CLEARING：定义后，将删除使 ThreadX SMP 全局 C 数据结构初始化为零的逻辑。仅当编译器的初始化代码将所有未初始化的 C 全局数据设置为零时，才应使用此选项。在初始化期间，使用此选项可以略微减小代码大小并提高性能。默认情况下，未定义此选项。
- TX_DISABLE_STACK_FILLING：定义后，将禁止在创建时将 0xEF 值置于每个线程的堆栈的每个字节中。默认情况下，未定义此选项。
- TX_ENABLE_EVENT_TRACE：定义后，ThreadX SMP 启用事件收集代码，用于创建 TraceX 跟踪缓冲区。有关更多详细信息，请参阅 TraceX 用户指南。
- TX_ENABLE_STACK_CHECKING：定义后，将启用 ThreadX SMP 运行时堆栈检查，其中包括分析已使用的堆栈量，以及堆栈区域前后的数据模式"防护"检查。如果检测到堆栈错误，则会调用已注册应用程序堆栈错误处理程序。此选项会导致系统开销和代码大小略有增加。有关详细信息，请查看 tx_-thread_stack_error_notify API。默认情况下，未定义此选项。
- TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO：定义后，可以收集事件标志组的性能信息。默认情况下，未定义此选项。
- TX_INLINE_THREAD_RESUME_SUSPEND：定义后，ThreadX SMP 通过行内代码改进 tx_thread_resume 和 tx_thread_suspend API 调用 。这会增加代码大小，但会增强这两个 API 调用的性能。
- TX_MAX_PRIORITIES：定义 ThreadX SMP 的优先级别。合法值的范围为 32 到 1024（含），且必须能被 32 整除。增加支持的优先级别数量会使每组 32 个优先级别的 RAM 用量增加 128 字节。但是，对性能的影响可忽略不计。默认情况下，此值设置为 32 个优先级别。
- TX_MINIMUM_STACK：定义最小堆栈大小（以字节为单位）。它用于在创建线程时进行错误检查。默认值是端口特定的，位于 tx_port.h 中。
- TX_MISRA_ENABLE：定义后，ThreadX SMP 使用 MISRA C 兼容约定。有关详细信息，请参阅 ThreadX_MISRA_Compliance.pdf。
- TX_MUTEX_ENABLE_PERFORMANCE_INFO：定义后，可以收集互斥体的性能信息。默认情况下，未定义此选项。
- TX_NO_TIMER：定义后，将完全禁用 ThreadX SMP 计时器逻辑。这在无法使用 ThreadX SMP 计时器功能（线程睡眠、API 超时、时间切片和应用程序计时器）的情况下十分有用。
- TX_NOT_INTERRUPTABLE：定义后，ThreadX SMP 不会尝试最大程度地缩短中断锁定时间。这会提升执行速度，但会略微增加中断锁定时间。
- TX_QUEUE_ENABLE_PERFORMANCE_INFO：定义后，可以收集队列的性能信息。默认情况下，未定义此选项。
- TX_REACTIVATE_INLINE：定义后，执行 ThreadX SMP 计时器内联的激活，而不是使用函数调用。这可以提升性能，但会略微增加代码大小。默认情况下，未定义此选项。
- TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO：定义后，可以收集信号灯的性能信息。默认情况下，未定义此选项。
- TX_THREAD_ENABLE_PERFORMANCE_INFO：定义后，可以收集线程的性能信息。默认情况下，未定义此选项。
- TX_THREAD_SMP_CORE_MASK：定义核心排除的位映射掩码。例如，根据此定义，四核环境的值为 0xF。

- TX_THREAD_SMP_DEBUG_ENABLE：定义后，ThreadX SMP 调试信息保存在循环缓冲区中。
- TX_THREAD_SMP_DYNAMIC_CORE_MAX：定义后，实现可在运行时进行调整的动态最大核心数。
- TX_THREAD_SMP_EQUAL_PRIORITY：定义后，ThreadX SMP 仅计划并行的相同优先级线程。应在生成 ThreadX SMP 库之前定义。
- TX_THREAD_SMP_INTER_CORE_INTERRUPT：定义后，ThreadX SMP 将生成核心间中断。
- TX_THREAD_SMP_MAX_CORES：定义最大核心数。
- TX_THREAD_SMP_ONLY_CORE_0_DEFAULT：定义后，ThreadX SMP 默认所有线程和计时器都只在核心 0 上执行。应用程序可以调用核心排除 API 来替代这一点。应在生成 ThreadX SMP 库之前定义。
- TX_THREAD_SMP_WAKEUP_LOGIC：定义后，将调用唤醒核心“i”的应用程序宏。应在包含 tx_port.h 之前定义。
- TX_THREAD_SMP_WAKEUP(i)：将应用程序宏定义为唤醒核心“i”。应在包含 tx_port.h 之前定义。
- TX_TIMER_ENABLE_PERFORMANCE_INFO：定义后，可以收集计时器的性能信息。默认情况下，未定义此选项。
- TX_TIMER_PROCESS_IN_ISR：定义后，将消除 ThreadX SMP 的内部系统计时器线程。这会提升计时器事件和更小 RAM 要求的性能，因为不再需要计时器堆栈和控制块。但是，使用此选项会将所有计时器过期处理移动到计时器 ISR 级别。默认情况下，未定义此选项。

  > **NOTE**
  > 计时器允许的该服务可能得不到 ISR 的允许，因此，在使用此选项时可能不允许这样做。

- TX_TIMER_THREAD_PRIORITY：定义内部 ThreadX SMP 系统计时器线程的优先级。默认值为优先级 0，即 ThreadX SMP 中的最高优先级。默认值定义在 tx_port.h 中。
- TX_TIMER_THREAD_STACK_SIZE：定义内部 ThreadX SMP 系统计时器线程的堆栈大小（以字节为单位）。此线程处理所有线程睡眠请求以及所有服务调用超时。此外，从该上下文调用所有应用程序计时器回调例程。默认值是端口特定的，位于 tx_port.h 中。

ThreadX SMP 版本 ID

可在 readme_threadx.txt 文件中找到 ThreadX SMP 版本 ID。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 tx_version_id 来获取 ThreadX SMP 版本*。

第 3 章 - Azure RTO ThreadX SMP 的功能组件

2021/4/30 •

本章从功能角度介绍了高性能 Azure RTO ThreadX SMP 内核。每个功能组件都将采用易于理解的方式进行介绍。

执行概述

ThreadX SMP 应用程序包含四种类型的程序执行：初始化、线程执行、中断服务例程 (ISR) 和应用程序计时器。

第 45 页的图 1 显示了各种不同类型的程序执行。而本章的后续部分详细介绍了其中的每种类型。

初始化

顾名思义，这是 ThreadX SMP 应用程序中的第一种程序执行。初始化包括处理器重置与“线程调度循环”入口点之间的所有程序执行。

> **IMPORTANT**
> 初始化由核心 0（重置后默认运行的核心）执行或启动。

线程执行

初始化完成后，每个运行 ThreadX SMP 的核心都会进入其线程调度循环。调度循环会查找能够在该核心上执行的应用程序线程。找到准备就绪的线程后，ThreadX SMP 会将控制权转交给该线程。系统完成该线程或另一个优先级较高的线程变为就绪后，会将执行权转交回线程调度循环，以查找每个核心上下一个优先级最高的就绪线程。

这个持续执行和调度线程的过程是 ThreadX SMP 应用程序中最常见的程序执行类型。

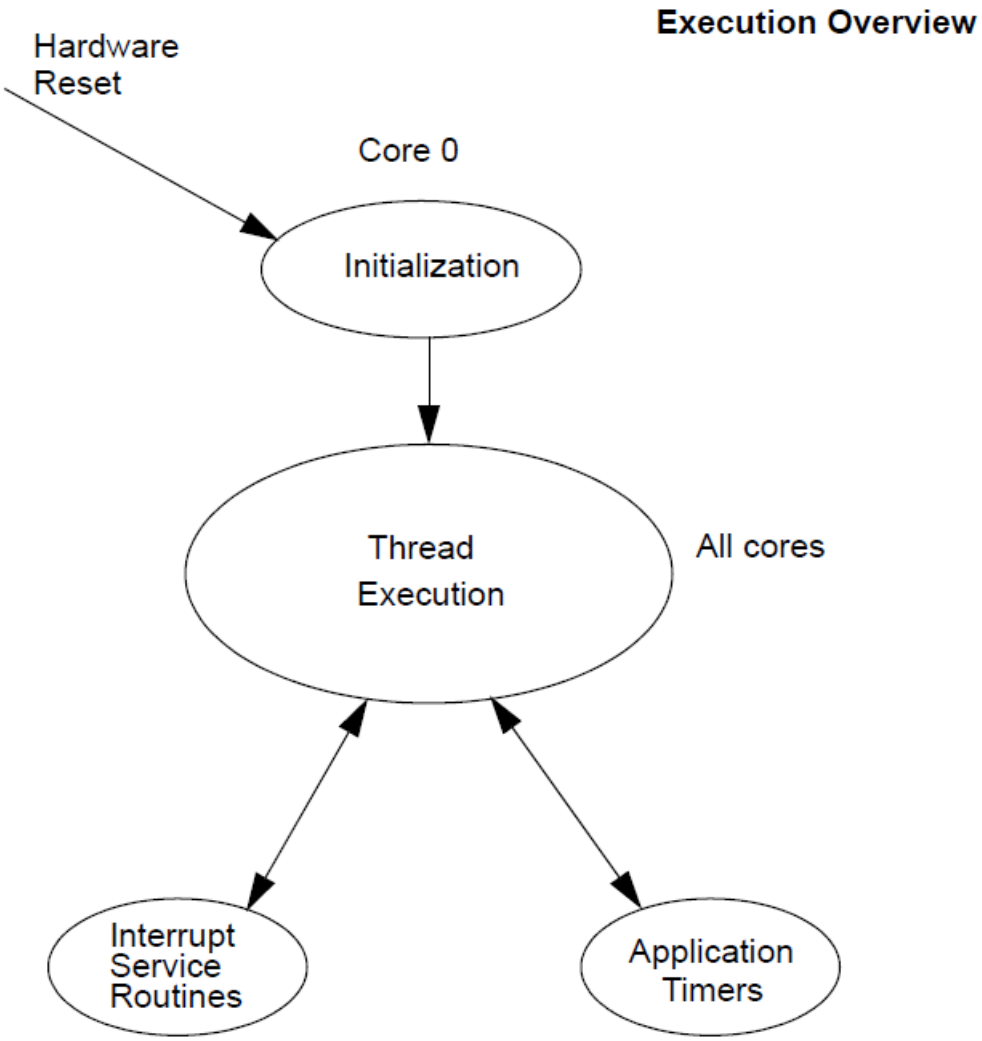


图 1. 程序执行类型

中断服务例程 (ISR)

中断是实时系统的基础。如果没有中断，系统将很难及时响应外部环境的变化。检测到中断时，处理器会保存当前程序执行的重要信息（通常在堆栈上），然后将控制权转交到预定义的程序区域。这个预定义的程序区域通常称为中断服务例程。

在大多数情况下，中断发生在线程执行期间（或线程调度循环中）。但也可能在执行 ISR 或应用程序计时器时发生中断。

所有核心都可以处理中断。中断到核心的映射由应用程序直接控制。默认情况下，系统会将 ThreadX SMP 计时器中断分配给核心 0 进行处理。有关此分配的实现，请参阅 tx_timer_interrupt.S 中的代码。

应用程序计时器

应用程序计时器与 ISR 类似，不同之处在于硬件实现（通常使用单个定期硬件中断）已对应用程序隐藏。应用程序使用此类计时器来执行超时、定期任务和/或监视器服务。与 ISR 一样，应用程序计时器最常中断线程执行。但与 ISR 不同的是，应用程序计时器无法相互中断。

> **NOTE**
> 应用程序计时器与线程一样，可排除在任何核心上的执行之外。

内存用量

ThreadX SMP 与应用程序一起驻留。因此，ThreadX SMP 的静态内存（或固定内存）使用情况取决于开发工具，例如编译器、链接器和定位器。动态内存（或运行时内存）使用情况由应用程序直接控制。

> **NOTE**
>
> ThreadX SMP 访问的所有内存都必须保持缓存一致，并且可从执行 ThreadX SMP 的所有核心进行访问。

静态内存使用情况

大多数开发工具将应用程序映像分为五个基本区域："指令"、"常数"、"已初始化的数据"、"未初始化的数据"和"系统堆栈"。第 47 页的图 2 显示了这些内存区域的示例。

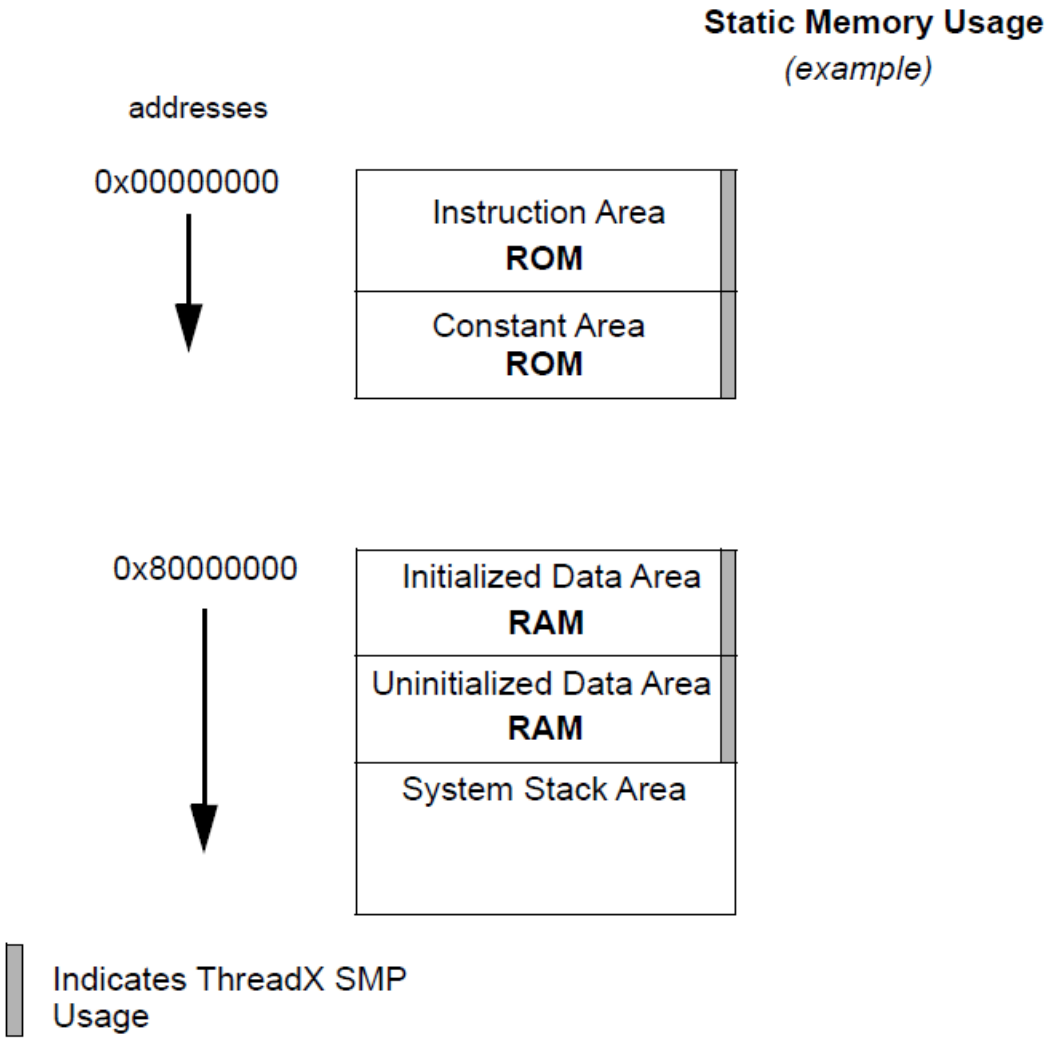


图 2. 内存区域示例

请务必注意，这只是一个示例。实际的静态内存布局特定于处理器、开发工具和基础硬件。

指令区域包含程序的所有处理器指令。此区域通常最大，一般位于 ROM 中。

常数区域包含各种已编译的常数，包括程序中定义或引用的字符串。此外，此区域包含已初始化的数据区域的"初始副本"。在编译器的初始化过程中，常数区域的这个部分用于设置 RAM 中已初始化的数据区域。常数区域通常在指令区域之后，一般位于 ROM 中。

已初始化的数据和未初始化的数据区域包含所有全局和静态变量。这些区域始终位于 RAM 中。

系统堆栈通常紧跟在初始化和未初始化的数据区域之后设置。系统堆栈供编译器在初始化期间使用，然后供 ThreadX SMP 在初始化期间使用，随后在 ISR 处理中使用。

动态内存使用情况

如前所述，动态内存使用情况由应用程序直接控制。与堆栈、队列和内存池关联的控制块和内存区域可以放置在目标内存空间中的任何位置。这是一项重要的功能，因为该功能有助于轻松使用不同类型的物理内存。

例如，假设目标硬件环境具有快速内存和慢速内存。如果应用程序需要额外的性能来处理高优先级线程，则可将其控制块 (TX_THREAD) 和堆栈放置在快速内存区域中，这可能会显著提高其性能。

初始化

了解初始化过程非常重要。初始硬件环境在此处设置。此外，还可在此处指定应用程序的初始个性化设置。

> **IMPORTANT**
> ThreadX SMP 尝试（尽可能）采用完整的开发工具的初始化过程。这样，以后就可以更轻松地升级到开发工具的新版本。

系统重置向量

所有微处理器都具有重置逻辑。当（硬件或软件）发生重置时，将从特定内存位置检索应用程序入口点的地址。检索到入口点后，处理器会将控制权转交到该位置。

应用程序入口点通常用本机程序集语言编写，并且通常由开发工具提供（至少采用模板形式）。在某些情况下，ThreadX SMP 会附带入口程序的特殊版本。

开发工具初始化

完成低级初始化后，系统将控制权转交给开发工具的高级初始化。此处通常设置了已初始化的全局变量和静态 C 变量。请记住，其初始值将从常数区域检索。具体的初始化处理将特定于开发工具。

main 函数

完成开发工具初始化后，系统将控制权转交给用户提供的 main 函数。此时，应用程序会控制接下来执行的操作。对于大多数应用程序，main 函数只调用 tx_kernel_enter，这是 ThreadX SMP 的入口。但是，应用程序可在进入 ThreadX SMP 之前执行初步处理（通常用于硬件初始化）。

> **IMPORTANT**
> 对 tx_kernel_enter 的调用不会返回结果，因此请勿在此后执行任何处理！

tx_kernel_enter

entry 函数协调各种内部 ThreadX SMP 数据结构的初始化，然后调用应用程序的定义函数 tx_application_define。

当 tx_application_define 返回时，控制权将转交给线程调度循环。这标志着初始化结束！

应用程序定义函数

tx_application_define 函数定义所有初始应用程序线程、队列、信号灯、互斥锁、事件标志、内存池和计时器。在应用程序的正常操作过程中，还可以在线程中创建和删除系统资源。但是，所有初始应用程序资源都在此处定义。

值得一提的是，tx_application_define 函数只有一个输入参数。“第一个可用”的 RAM 地址是该函数唯一的输入参数。该地址通常用作线程堆栈、队列和内存池的初始运行时内存分配起点。

> **IMPORTANT**
> 初始化完成后，只有正在执行的线程才能创建和删除系统资源（包括其他线程）。因此，在初始化期间必须至少创建一个线程。

中断

在整个初始化过程中，中断处于禁用状态。如果应用程序以某种方式启用中断，则可能出现不可预知的行为。第 52 页的图 3 显示了从系统重置到特定于应用程序的初始化的整个初始化过程。

线程执行

调度和执行应用程序线程是 ThreadX SMP 最重要的活动。线程通常定义为具有专用用途的半独立程序段。所有线程的组合处理构成了应用程序。

线程在初始化或线程执行期间通过调用 tx_thread_create 来动态创建。创建的线程处于“就绪”或“已挂起”状态。

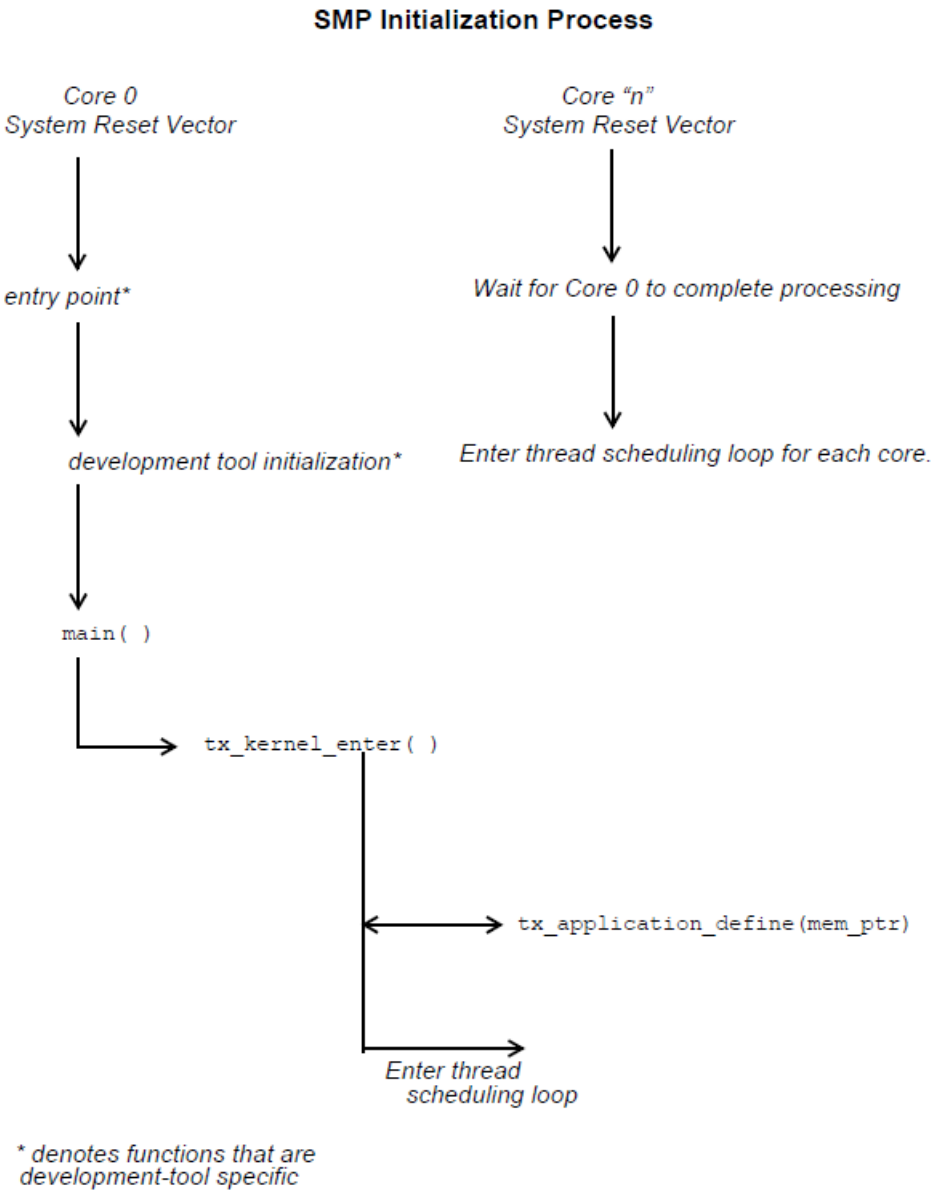


图 3. SMP 初始化过程

线程执行状态

了解线程的不同处理状态是了解整个多线程环境的关键要素。ThreadX SMP 有五个不同的线程状态：“就绪”、“已挂起”、“正在执行”、“已终止”和“已完成”。图 4 显示了 ThreadX SMP 的线程状态转换图。

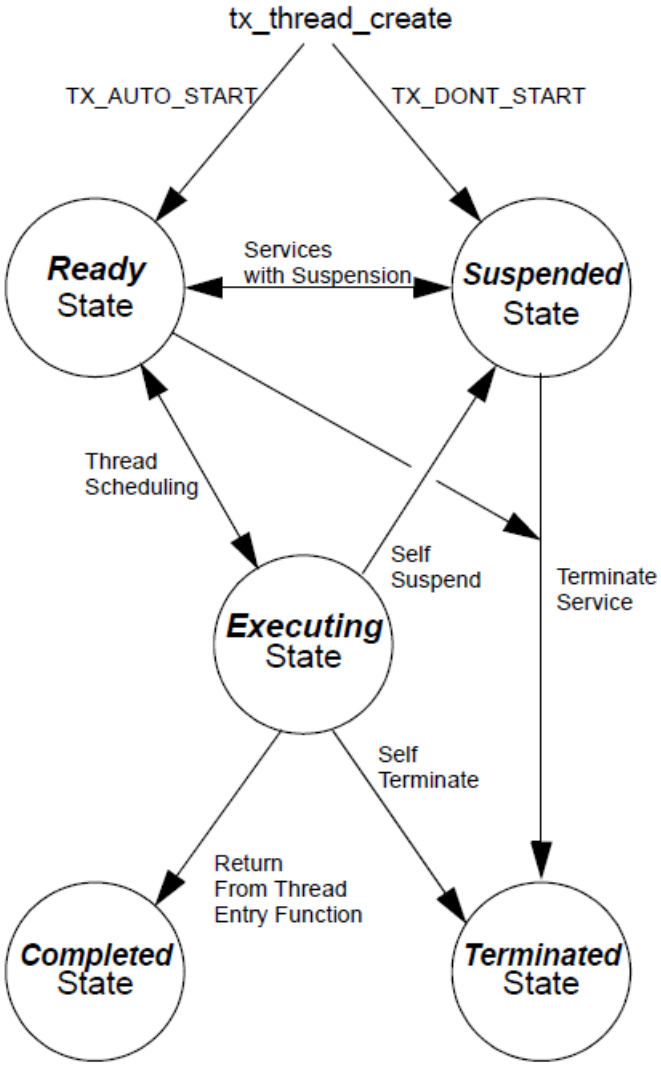


图 4. 线程状态转换

线程准备好执行时处于"就绪"状态。在处于就绪状态的线程中，只有优先级最高的就绪线程才会执行。发生这种情况时，ThreadX SMP 会执行该线程，然后将其状态更改为"正在执行"。

如果更高优先级的线程已准备就绪，正在执行的线程会恢复为"就绪"状态。然后执行新的高优先级就绪线程，并将其逻辑状态更改为"正在执行"。每次发生线程抢占时，即会在"就绪"和"正在执行"之间转换。

在任意指定时刻，只有一个线程处于"正在执行"状态。这是因为处于"正在执行"状态的线程可以控制基础处理器。

处于"已挂起"状态的线程不符合执行条件。导致处于"已挂起"状态的原因包括挂起时间、队列消息、信号灯、互斥锁、事件标志、内存和基本线程挂起。排除导致挂起的原因后，线程将恢复"就绪"状态。

处于"已完成"状态的线程是指已完成其处理任务并从其 entry 函数返回的线程。entry 函数在线程创建期间指定。处于"已完成"状态的线程无法再次执行。

线程处于"已终止"状态，因为另一个线程或该线程本身调用了 tx_thread_terminate 服务。处于"已终止"状态的线程无法再次执行。

> **IMPORTANT**
>
> 如果需要重新启动已完成或已终止的线程，应用程序必须先删除该线程。然后才能重新创建并重新启动该线程。

线程进入/退出通知

某些应用程序可能会发现，在特定线程首次进入、完成或终止时收到通知非常有利。ThreadX SMP 通过 tx_thread_entry_exit_notify 服务提供此功能。此服务为特定线程注册应用程序通知函数，ThreadX SMP 会在该线程开始运行、完成或终止时调用该函数。应用程序通知函数在调用后可以执行特定于应用程序的处理。这通常涉及通过 ThreadX SMP 同步基元通知此事件的另一个应用程序线程。

线程优先级

如前所述，线程是具有专用用途的半独立程序段。但是，所有线程在创建时并非完全相同！ 某些线程具有比其他线程更重要的专用用途。这种异类的线程重要性是嵌入式实时应用程序的标志。

ThreadX SMP 在创建线程时通过分配表示其"优先级"的数值来确定线程的重要性。ThreadX SMP 的最大优先级数可在 32 到 1024（增量为 32）之间进行配置。实际的最大优先级数由 TX_MAX_PRIORITIES 常数在 ThreadX SMP 库的编译过程中确定。设置更多优先级并不会显著增加处理开销。但是，每个包含 32 个优先级的组额外需要 128 字节的 RAM 进行管理。例如，32 个优先级需要 128 字节的 RAM，64 个优先级需要 256 字节的 RAM，而 96 个优先级需要 384 字节的 RAM。

默认情况下，ThreadX SMP 具有 32 个优先级，范围从优先级 0 到优先级 31。

数值越小，优先级越高。因此，优先级 0 表示最高优先级，而优先级 (TX_MAX_PRIORITIES-1) 表示最低优先级。

通过协作调度或时间切片，多个线程可以具有相同的优先级。此外，线程优先级还可以在运行时更改。

线程调度

ThreadX SMP 根据线程的优先级来调度线程。优先级最高的就绪线程最先执行。如果有多个具有相同优先级的线程准备就绪，则按照"先进先出"(FIFO) 的方式执行。

默认情况下，ThreadX SMP 在"n"个可用处理器上安排"n"个优先级最高的线程。如果只需对相同优先级的就绪线程进行并发处理，则必须在定义 TX_THREAD_SMP_EQUAL_PRIORITY 的情况下生成 ThreadX SMP 库。

> **NOTE**
> 通过生成已定义 TX_THREAD_SMP_ONLY_CORE_0_DEFAULT 的 ThreadX SMP 库，所有线程才能在最初默认为只在核心 0 上运行。

轮循调度

ThreadX SMP 支持通过"轮循"调度处理具有相同优先级的多个线程。此过程通过以协作方式调用 tx_thread_relinquish 来实现。此服务为相同优先级的所有其他就绪线程提供了在 tx_thread_relinquish 调用方再次执行之前执行的机会。

时间切片

"时间切片"是轮循调度的另一种形式。时间片指定线程在不放弃处理器的情况下可以执行的最大计时器时钟周期数（计时器中断）。在 ThreadX SMP 中，时间切片按每个线程提供。线程的时间片在创建时分配，可在运行时修改。当时间片过期时，具有相同优先级的所有其他就绪线程有机会在时间切片线程重新执行之前执行。

当线程挂起、放弃、执行导致抢占的 ThreadX SMP 服务调用或自身经过时间切片后，该线程将获得一个新的线程时间片。

当时间切片的线程被抢占时，该线程将在其剩余的时间片内比具有相同优先级的其他就绪线程更早恢复执行。

> **IMPORTANT**
> 使用时间切片会产生少量系统开销。由于时间切片仅适用于多个线程具有相同优先级的情况，因此不应为具有唯一优先级的线程分配时间片。

优先

抢占是为了支持优先级更高的线程而暂时中断正在执行的线程的过程。此过程对正在执行的线程不可见。当更

高优先级的线程完成时，控制权将转交回发生抢占的确切位置。

这是实时系统中一项非常重要的功能，因为该功能有助于快速响应重要的应用程序事件。尽管抢占是一项非常重要的功能，但也可能导致各种问题，包括资源不足、开销过大和优先级倒置。

Preemption-Threshold™

为了缓解抢占的一些固有问题，ThreadX SMP 提供了一个独特的高级功能，名为“抢占式阀值”。

抢占式阀值允许线程指定禁用抢占的优先级“上限”。优先级高于上限的线程仍可以执行抢占，但不允许优先级低于上限的线程执行抢占。

例如，假设优先级为 20 的线程只与一组优先级介于 15 到 20 之间的线程进行交互。在其关键部分中，优先级为 20 的线程可将其抢占式阀值设置为 15，从而防止该线程和与之交互的所有线程发生抢占。这仍允许（优先级介于 0和 14 之间）真正重要的线程在其关键部分处理中抢占此线程的资源，这会使处理的响应速度更快。

当然，仍有可能通过将其抢占式阀值设置为 0 来为线程禁用所有抢占。此外，抢占式阀值可以在运行时更改。

> **IMPORTANT**
> 使用抢占式阀值禁用指定线程的时间切片。

优先级继承

ThreadX SMP 还支持其互斥服务内的可选优先级继承，本章稍后将对此进行介绍。优先级继承允许低优先级线程暂时假设正在等待归较低优先级线程所有的互斥锁的高优先级线程的优先级。借助此功能，应用程序可以消除中间优先级线程的抢占，从而避免出现不确定的优先级倒置。当然，也可以使用“抢占式阀值”获得类似的结果。

线程创建

应用程序线程在初始化或执行其他应用程序线程的过程中创建。应用程序可以创建的线程数量没有限制。

线程控制块 TX_THREAD

每个线程的特征都包含在其控制块中。此结构在 tx_api.h 文件中定义。

线程的控制块可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

在其他区域查找控制块需要更加小心，就像所有动态分配的内存一样。如果在 C 函数内分配控制块，与之相关联的内存就是调用线程堆栈的一部分。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，而不管另一个线程是否正将其用于控制块！

在大多数情况下，应用程序不知道线程控制块的内容。但在某些情况下，尤其是在调试过程中，观察特定成员会很有用。下面是一些更有用的控制块成员：

- tx_thread_run_count 包含记录线程已调用次数的计数器。计数器增加表示正在调度和执行线程。
- tx_thread_state 包含相关线程的状态。下面列出了可能的线程状态：
  - TX_READY(0x00)
  - TX_COMPLETED(0x01)
  - TX_TERMINATED(0x02)
  - TX_SUSPENDED(0x03)
  - TX_SLEEP(0x04)
  - TX_QUEUE_SUSP(0x05)
  - TX_SEMAPHORE_SUSP(0x06)
  - TX_EVENT_FLAG (0x07)
  - TX_BLOCK_MEMORY(0x08)

- TX_BYTE_MEMORY (0x09)
- TX_MUTEX_SUSP(0x0D)

> **IMPORTANT**
> 当然，线程控制块中还有很多有趣的字段，包括堆栈指针、时间片值、优先级等。欢迎用户查看控制块成员，但严格禁止修改！

> **IMPORTANT**
> 本节前面提到的“正在执行”状态没有等同的状态。这不是必需的，因为在给定时间只有一个正在执行的线程。正在执行的线程的状态也是 TX_READY。

当前正在执行的线程

如前所述，在任何给定时间都只有一个正在执行的线程。有多种方法可以识别正在执行的线程，具体取决于发出请求的线程。

通过调用 tx_thread_identify，程序段可以获取正在执行的线程的控制块地址。这在从多个线程执行的应用程序代码的共享部分中很有用。

在调试会话中，用户可以检查 ThreadX SMP 内部指针数组 _tx_thread_current_ptr [core]。该数组包含当前正在执行的线程的控制块地址。如果此指针为 NULL，则不执行任何应用程序线程；也就是说，ThreadX SMP 在其调度循环中等待线程准备就绪。

线程堆栈区域

每个线程都必须有自己的堆栈，用于保存其上次执行和编译器使用的上下文。大多数 C 编译器使用堆栈来执行函数调用和临时分配局部变量。第 61 页的图 5 显示了典型的线程堆栈。

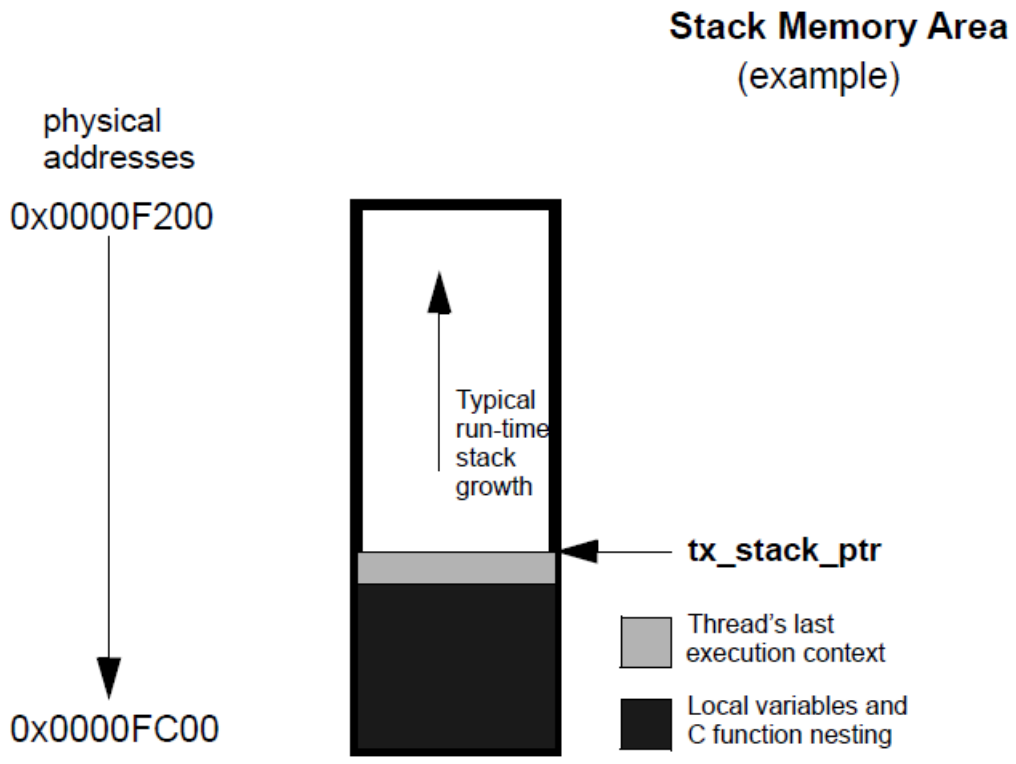


图 5. 典型的线程堆栈

线程堆栈在内存中的位置取决于应用程序。堆栈区域在线程创建期间指定，可以位于目标地址空间的任意位置。

这是一项重要的功能，因为应用程序可借助该功能，通过将重要线程堆栈置于高速 RAM 中来提高线程的性能。

应该设置多大的堆栈是有关线程的最常见问题之一。线程的堆栈区域必须大到足以容纳最坏情况下的函数调用嵌套、局部变量分配，并保存其最后一个执行上下文。

最小堆栈大小 TX_MINIMUM_STACK 由 ThreadX SMP 定义。这种大小的堆栈支持保存线程的上下文、最少量的函数调用和局部变量分配。

但对大多数线程来说，最小的堆栈大小太小，用户必须通过检查函数调用嵌套和局部变量分配来确定最坏情况下的大小要求。当然，最好从较大的堆栈区域开始。

调试应用程序后，如果内存不足，可以调整线程堆栈大小。常用的技巧是在创建线程之前，使用诸如 (0xEFEF) 之类易于识别的数据模式预设所有堆栈区域。在应用程序经过全面测试后，可以对堆栈区域进行检查，具体方法是通过查找堆栈区域（其中的数据模式仍保持不变）来查看实际使用了多少堆栈。图 6 显示了堆栈在线程完全执行后预设为 0xEFEF。

> **IMPORTANT**
>
> 默认情况下，ThreadX SMP 使用值 0xEF 初始化每个线程堆栈的每个字节。

内存缺陷

线程的堆栈需求可能很大。因此，务必要将应用程序设计为可以容纳数量合理的线程。此外，必须注意避免在线程中过度使用堆栈。应避免使用递归算法和大型本地数据结构。

在大多数情况下，溢出的堆栈会导致线程执行损坏其堆栈区域相邻

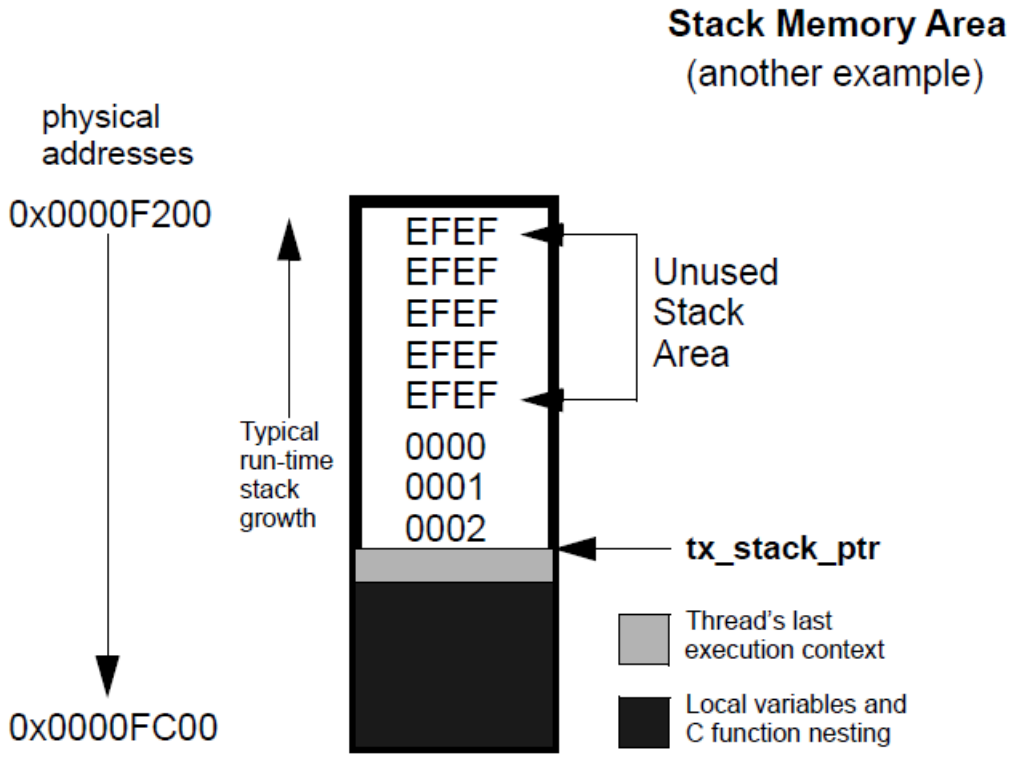


图 6. 堆栈预设为 0xEFEF

（通常在此位置之前）的内存。其结果不可预测，但大多会导致程序计数器出现反常的变化。这通常称为“深陷细枝末节”。当然，防止出现这种情况的唯一方法是确保所有线程堆栈足够大。

可选的运行时堆栈检查

ThreadX SMP 提供了在运行时检查每个线程的堆栈是否损坏的功能。默认情况下，ThreadX SMP 在创建过程中使用 0xEF 数据模式填充线程堆栈的每个字节。如果应用程序生成了已定义 *TX_ENABLE_STACK_CHECKING_ 的 ThreadX SMP 库，ThreadX SMP 会检查每个线程的堆栈是否因为线程挂起或恢复而损坏。如果检测到堆栈损坏，ThreadX SMP 将调用在调用 _tx_thread_stack_error_notify* 时指定的应用程序堆栈错误处理例程。否则，如果未指定堆栈错误处理程序，ThreadX SMP 将调用内部 _tx_thread_stack_error_handler 例程。

重新进入

多线程处理的真正优点之一是，可以从多个线程中调用相同的 C 函数。这提供了强大的功能，还有助于减少代码空间。但是，此功能要求从多个线程调用的 C 函数是"可重入"的函数。

基本上，可重入函数将调用方的返回地址存储在当前堆栈上，并且不依赖于先前设置的全局或静态 C 变量。大多数编译器将返回地址放在堆栈上。因此，应用程序开发人员必须只关心如何使用"全局"和"静态"变量。

非重入函数的一个例子是在标准 C 库中找到的字符串标记函数"strtok"。此函数在后续调用时记住了前面的字符串指针。此函数通过静态字符串指针实现此功能。如果从多个线程调用此函数，则最有可能返回无效的指针。

线程优先级缺陷

选择线程优先级是多线程处理最重要的方面之一。有时很容易根据感知的线程重要性概念来分配优先级，而不是确定运行时到底需要什么。滥用线程优先级会导致其他线程资源枯竭、产生优先级倒置、减少处理带宽，致使应用程序出现难以理解的运行时行为。

如前所述，ThreadX SMP 提供基于优先级的抢占式调度算法。优先级较低的线程只能在没有更高优先级的线程可以执行时才会执行。如果优先级较高的线程始终准备就绪，则不会执行优先级较低的线程。这种情况称为"线程资源不足"。

大多数线程资源不足的问题都是在调试初期检测到的，可通过确保优先级较高的线程不会连续执行来解决。另外，还可以在应用程序中添加此逻辑，以便逐渐提高资源不足的线程的优先级，直到有机会执行这些线程。

与线程优先级相关的另一个缺陷是"优先级倒置"。当优先级较高的线程由于优先级较低的线程占用其所需资源而挂起时，将会发生优先级倒置。当然，在某些情况下，有必要让两个优先级不同的线程共享一个公用资源。如果这些线程是唯一处于活动状态的线程，优先级倒置时间就与低优先级线程占用资源的时间息息相关。这种情况既具有确定性又非常正常。不过，如果在这种优先级倒置的情况，中等优先级的线程变为活动状态，优先级倒置时间就不再确定，并且可能导致应用程序失败。

主要有三种不同的方法可防止 ThreadX SMP 中出现不确定的优先级倒置。首先，在设计应用程序优先级选择和运行时行为时，可以采用能够防止出现优先级倒置问题的方式。其次，优先级较低的线程可以利用"抢占式阈值"来阻止中等优先级线程在其与优先级较高的线程共享资源时执行抢占。最后，使用 ThreadX SMP 互斥对象保护系统资源的线程可以利用可选的互斥"优先级继承"来消除不确定的优先级倒置。

优先级开销

要减少多线程处理开销，最容易被忽视的一种方法是减少上下文切换的次数。如前所述，当优先级较高的线程执行优先于正在执行的线程时，则会发生上下文切换。值得一提的是，在发生外部事件（例如中断）和执行线程发出服务调用时，优先级较高的线程可能变为就绪状态。

为了说明线程优先级对上下文切换开销的影响，假设有一个包含三个线程的环境，这些线程分别命名为 thread_1、thread_2 和 thread_3。进一步假定所有线程都处于等待消息的挂起状态。当 thread_1 收到消息后，立即将其转发给 thread_2。随后，thread_2 将此消息转发给 thread_3。Thread_3 只是丢弃此消息。每个线程处理其消息后，则会返回并等待另一个消息。

执行这三个线程所需的处理存在很大的差异，具体取决于其优先级。如果所有线程都具有相同优先级，则会在执行每个线程之前发生一次上下文切换。当每个线程在空消息队列中挂起时，将会发生上下文切换。

但是，如果 thread_2 的优先级高于 thread_1，thread_3 的优先级高于 thread_2，上下文切换的次数将增加一倍。这是因为当其检测到优先级更高的线程现已准备就绪时，会在 tx_queue_send 服务中执行另一次上下文切换。

ThreadX SMP 抢占式阈值机制可以避免出现这些额外的上下文切换，并且仍支持前面提到的优先级选择。这是

一项重要的功能，因为该功能允许在调度期间使用多个线程优先级，同时避免在线程执行期间出现一些不需要的上下文切换。

运行时线程性能信息

ThreadX SMP 提供可选的运行时线程性能信息。如果 ThreadX SMP 库和应用程序是在定义 TX_THREAD_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX SMP 会累积以下信息：

整个系统的总数：

- 线程恢复数
- 线程挂起数
- 服务调用抢占次数
- 中断抢占次数
- 优先级倒置数
- 时间切片数
- 放弃次数
- 线程超时数
- 挂起中止数
- 空闲系统返回数
- 非空闲系统返回数

每个线程的总数：

- 恢复数
- 挂起数
- 服务调用抢占次数
- 中断抢占次数
- 优先级倒置数
- 时间切片数
- 线程放弃
- 线程超时数
- 挂起中止数

此信息在运行时通过 tx_thread_performance_info_get 和 tx_thread_performance_system_info_get 服务提供。线程性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。例如，如果服务调用抢占数量相对较多，可能表明线程的优先级和/或抢占式阈值过低。此外，如果空闲系统返回次数相对较少，可能表明优先级较低的线程没有完全挂起。

调试缺陷

调试多线程应用程序更为困难，因为同一个程序代码可以通过多个线程执行。在这种情况下，只使用断点可能不够。调试器还必须使用条件断点观察当前线程指针数组 _tx_thread_current_ptr [core]，以查看调用线程是否为要调试的线程。

其中很多工作都是使用各种开发工具供应商提供的多线程支持包进行处理。因为这些支持并设计简单，因此将 ThreadX SMP 与不同的开发工具集成相对容易。

堆栈大小始终是多线程处理的重要调试主题。如果观察到无法解释的行为，通常最好是增加所有线程的堆栈大小，尤其是要执行的最后一个线程的堆栈大小！

> **IMPORTANT**
> 使用定义的 TX_ENABLE_STACK_CHECKING 生成 ThreadX SMP 库也是个好办法。这将有助于在处理过程中尽早排除堆栈损坏问题！

消息队列

消息队列是 ThreadX SMP 中线程间通信的主要方式。消息队列中可以驻留一个或多个消息。保留单个消息的消息队列通常称为“邮箱”。

消息通过 tx_queue_send 复制到队列，然后通过 tx_queue_receive 从队列中复制。唯一的例外是在等待空队列中的消息时线程会挂起。在这种情况下，发送到队列的下一条消息将直接放入该线程的目标区域。

每个消息队列都是一个公用资源。ThreadX SMP 对如何使用消息队列没有任何限制。

创建消息队列

消息队列由应用程序线程在初始化期间或运行时创建。应用程序中的消息队列数没有限制。

消息大小

每个消息队列都支持许多固定大小的消息。可用的消息大小为 1 到 16 个 32 位的字（含）。消息大小在创建队列时指定。

超过 16 个字的应用程序消息必须通过指针传递。为此，可以创建消息大小为 1 个字的队列（足以容纳一个指针），然后发送和接收消息指针，而不是整个消息。

消息队列容量

队列可以容纳的消息数是消息大小与创建期间提供的内存区域大小的函数。队列的总消息容量的计算方法是，将每条消息中的字节数除以所提供的内存区域的总字节数。

例如，如果支持消息大小为 1 个 32 位字（4 个字节）的消息队列，是使用 100 字节内存区域创建的，则其容量为 25 条消息。

队列内存区域

如前所述，用于缓冲消息的内存区域在队列创建期间指定。与 ThreadX SMP 中的其他内存区域一样，该区域可以位于目标地址空间的任何位置。

这是一项重要的功能，因为它为应用程序提供了相当大的灵活性。例如，应用程序可能会在高速 RAM 中定位重要队列的内存区域，从而提高性能。

线程挂起

尝试从队列发送或接收消息时，应用程序线程可能会挂起。线程挂起通常涉及等待来自空队列的消息。但是，线程也可能在尝试向已满队列发送消息时挂起。

解决挂起的条件后，请求的服务完成，等待的线程也相应恢复。如果同一队列中的多个线程挂起，这些线程将按照挂起的顺序 (FIFO) 恢复。

不过，如果应用程序在取消线程挂起的队列服务之前调用 tx_queue_prioritize，还可以恢复优先级。队列设置优先级服务将优先级最高的线程置于挂起列表的前面，让所有其他挂起的线程采用相同的 FIFO 顺序。

超时也可用于所有队列挂起。从根本上说，超时会指定线程保持挂起状态的最大计时器时钟周期数。如果发生超时，则会恢复线程，该服务会返回相应的错误代码。

队列发送通知

某些应用程序可能会发现，在将消息放入队列时收到通知十分有利。ThreadX SMP 通过 tx_thread_entry_exit_notify 服务提供此功能。此服务将提供的应用程序通知函数注册到指定的队列。只要有消息发送到队列，ThreadX SMP 就会调用此应用程序通知函数。应用程序通知函数内的确切处理由应用程序决定；但这通常包括恢复相应的线程以处理新消息。

队列 Event-chaining™

ThreadX SMP 中的通知功能可用于链接各种同步事件。当单个线程必须处理多个同步事件时，这通常很有用。

例如，假设单个线程负责处理来自五个不同队列的消息，还必须在没有可用消息时挂起。通过为每个队列注册应用程序通知函数，并引入额外的计数信号灯，即可轻松实现这一点。具体而言，每次调用应用程序通知函数时，

该函数就会执行 tx_semaphore_put(信号灯计数表示所有五个队列中的消息总数)。处理线程通过 tx_semaphore_get 服务挂起此信号灯。当信号灯可用时(在这种情况下是消息可用时),即会恢复处理线程。随后,它会询问每个队列来获取一条消息,处理找到的消息,然后执行另一个 tx_semaphore_get,以等待下一条消息。在不使用事件链的情况下实现这一点非常困难,可能需要更多线程和/或附加的应用程序代码。

通常情况下,"事件链"会减少线程、降低开销和减少 RAM 要求。它还提供了一种非常灵活的机制来处理更复杂系统的同步要求。

运行时队列性能信息

ThreadX SMP 提供可选的运行时队列性能信息。如果 ThreadX SMP 库和应用程序是在定义 TX_QUEUE_ENABLE_PERFORMANCE_INFO 的情况下生成的,ThreadX SMP 会累积以下信息:

整个系统的总数:

- 发送的消息数
- 收到的消息数
- 队列为空的挂起数
- 队列已满的挂起数
- 队列已满返回的错误数(未指定挂起)
- 队列超时数

每个队列的总数:

- 发送的消息数
- 收到的消息数
- 队列为空的挂起数
- 队列已满的挂起数
- 队列已满返回的错误数(未指定挂起)
- 队列超时数

此信息在运行时通过 tx_queue_performance_info_get 和 tx_queue_performance_system_info_get 服务提供。队列性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。例如,如果"队列已满的挂起数"相对较多,则表明增加队列大小可能有好处。

队列控制块 TX_QUEUE

每个消息队列的特征都可在其控制块中找到。控制块包含受关注的信息,例如队列中的消息数。此结构在 tx_api.h 文件中定义。

消息队列控制块也可以位于内存中的任意位置,但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

消息目标缺陷

如前所述,消息将在队列区域和应用程序数据区域之间复制。请务必确保收到消息的目标大到足以容纳整个消息。否则,可能会损坏消息目标后面的内存。

> **WARNING**
> 如果堆栈上的消息目标太小,就特别致命,没有什么比损坏函数的返回地址更重要!

计数信号灯

ThreadX SMP 提供 32 位计数信号灯,其值范围在 0 到 4,294,967,295 之间。计数信号灯有两个操作:tx_semaphore_get 和 tx_semaphore_put。执行获取操作会将信号灯数量减一。如果信号灯为 0,获取操作不会成功。获取操作的逆操作是放置操作。该操作会将信号灯数量加一。

每个计数信号灯都是一个公用资源。ThreadX SMP 对如何使用计数信号灯没有任何限制。

计数信号灯通常用于"互相排斥"。不过，也可将计数信号灯用作事件通知的方法。

互相排斥

互相排斥用于控制线程对某些应用程序区域（也称为"关键部分"或"应用程序资源"）的访问。将信号灯用于互相排斥时，信号灯的"当前计数"表示允许访问的线程总数。在大多数情况下，用于互相排斥的计数信号灯的初始值为 1，这意味着每次只有一个线程可以访问关联的资源。只有 0 或 1 值的计数信号灯通常称为"二进制信号灯"。

> **IMPORTANT**
> 如果使用二进制信号灯，用户必须阻止同一个线程对其已拥有的信号灯执行获取操作。第二个获取操作将失败，并且可能导致调用线程无限期挂起和资源永久不可用。

事件通知

还可以采用生成者-使用者的方式，将计数信号灯用作事件通知。使用者尝试获取计数信号灯，而生成者则在有可用的信息时增加信号灯。此类信号灯的初始值通常为 0，此值不会在生成者为使用者准备好信息之前增加。用于事件通知的信号灯也可能从使用 *tx_semaphore_ceiling_put* 服务调用中获益。此服务确保信号灯计数值永远不会超过调用中提供的值。

创建计数信号灯

计数信号灯由应用程序线程在初始化期间或运行时创建。信号灯的初始计数在创建过程中指定。应用程序中计数信号灯的数量没有限制。

线程挂起

尝试对当前计数为 0 的信号灯执行获取操作时，应用程序线程可能会挂起。

执行放置操作后，才会执行挂起线程的获取操作并恢复该线程。如果同一计数信号灯上挂起多个线程，这些线程将按照挂起的顺序 (FIFO) 恢复。

不过，如果应用程序在取消线程挂起的信号灯放置调用之前调用 tx_semaphore_prioritize，还可以恢复优先级。信号灯设置优先级服务将优先级最高的线程放于挂起列表的前面，同时让所有其他挂起的线程采用相同的 FIFO 顺序。

信号灯放置通知

某些应用程序可能会发现，在放置信号灯时收到通知十分有利。ThreadX SMP 通过 tx_semaphore_put_notify 服务提供此功能。此服务将提供的应用程序通知函数注册到指定的信号灯。只要放置了信号灯，ThreadX SMP 就会调用此应用程序通知函数。应用程序通知函数内的确切处理由应用程序决定；但这通常包括恢复相应的线程以处理新信号灯放置事件。

信号灯 Eventchaining™

ThreadX SMP 中的通知功能可用于链接各种同步事件。当单个线程必须处理多个同步事件时，这通常很有用。

例如，应用程序可以为每个对象注册一个通知例程，而不是为队列消息、事件标志和信号灯而挂起单独的线程。在调用后，应用程序通知例程会恢复单个线程，该线程可以询问每个对象以查找并处理新事件。

通常情况下，"事件链"会减少线程、降低开销和减少 RAM 要求。它还提供了一种非常灵活的机制来处理更复杂系统的同步要求。

运行时信号灯性能信息

ThreadX SMP 提供可选的运行时信号灯性能信息。如果 ThreadX SMP 库和应用程序是在定义 TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX SMP 会累积以下信息：

整个系统的总数：

- 信号灯放置数

- 信号灯获取数
- 信号灯获取挂起数
- 信号灯获取超时数

每个信号灯的总数：

- 信号灯放置数
- 信号灯获取数
- 信号灯获取挂起数
- 信号灯获取超时数

此信息在运行时通过 tx_semaphore_performance_info_get 和 tx_semaphore_performance_system_info_get 服务提供。信号灯性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。例如，“信号灯获取超时数”相对较高可能表明其他线程占用资源的时间太长。

信号灯控制块 TX_SEMAPHORE

每个计数信号灯的特征都可在其控制块中找到。该控制块包含诸如当前的信号灯计数等信息。此结构在 tx_api.h 文件中定义。

信号灯控制块可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

抱死

与用于互相排斥的信号灯相关的最有趣且最危险的缺陷之一是“抱死”。抱死或“死锁”是指两个或多个线程在尝试获取归对方所有的信号灯时无限期挂起的情况。

这种情况最好用两个线程、两个信号灯的示例来说明。假设第一个线程拥有第一个信号灯，第二个线程拥有第二个信号灯。如果第一个线程尝试获取第二个信号灯，同时第二个线程尝试获取第一个信号灯，这两个线程就会进入死锁状态。此外，如果这些线程永远保持挂起状态，与之关联的资源也会永久锁定。第 78 页的图 7 说明了这个例子。

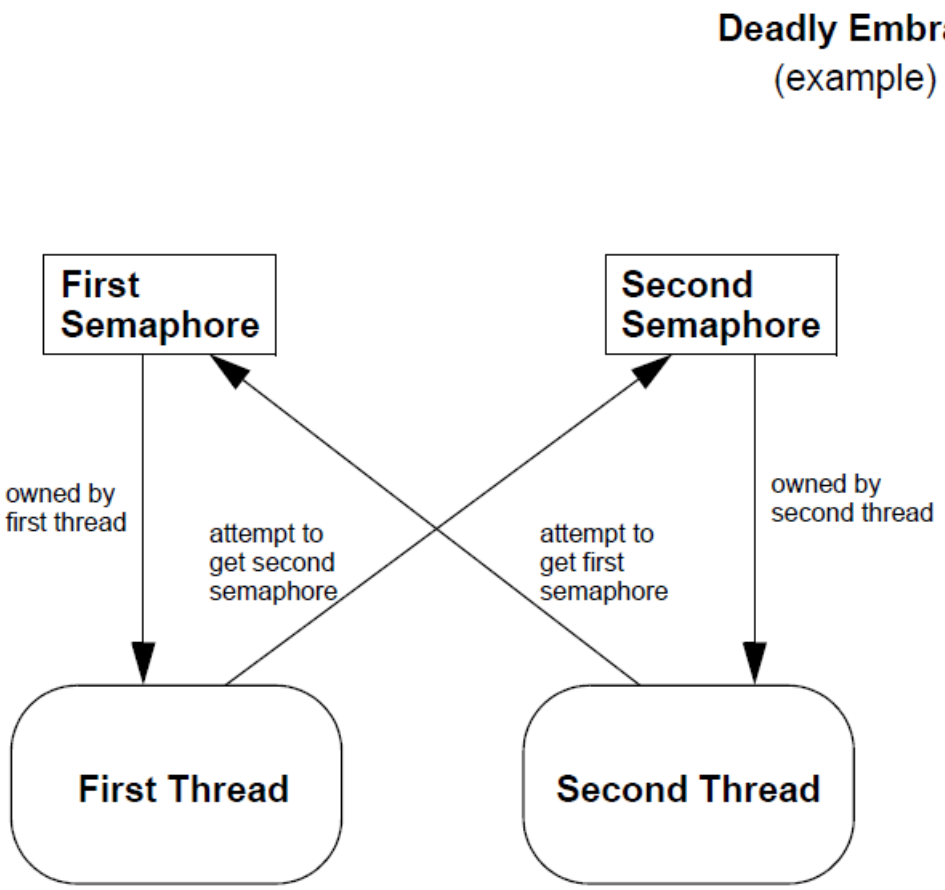


图 7. 挂起线程的示例

对于实时系统，可以通过对线程获取信号灯的方式设置一些限制来防止抱死。线程每次只能拥有一个信号灯。或者，如果线程按照相同的顺序收集多个信号灯，则可以拥有这些信号灯。在前面的示例中，如果第一个和第二个线程按顺序获取第一个和第二个信号灯，则可防止抱死。

> **IMPORTANT**
> 也可以使用与获取操作关联的挂起超时从抱死状态中恢复。

优先级倒置

与互相排斥信号灯相关的另一个缺陷是优先级倒置。第 64 页的"线程优先级缺陷"更全面地讨论了本主题。

根本问题源自这种情况，即低优先级线程拥有较高优先级线程所需的信号灯。这本身很正常。但是，它们之间具有优先级的线程可能会导致优先级倒置持续不确定的时间。这种情况可通过以下方式处理：谨慎选择线程优先级，使用抢占式阀值，并将拥有该资源的线程的优先级暂时提升到高优先级线程的级别。

Mutexes

除了信号灯，ThreadX SMP 还提供互斥对象。互斥锁实质上是二进制信号灯，这意味着每次只有一个线程可以拥有一个互斥锁。此外，同一个线程可能会在拥有互斥锁时多次成功执行互斥获取操作，准确来说是 4,294,967,295 次。互斥对象有两个操作：tx_mutex_get*_ 和 tx_mutex_put_***。获取操作获得不归另一个线程所有的互斥锁，而放置操作释放以前获得的互斥锁。对于要释放互斥锁的线程，放置操作数必须等于先前的获取操作数。

每个互斥锁都是一个公用资源。ThreadX SMP 对如何使用互斥锁没有任何限制。

ThreadX 互斥锁仅用于"互相排斥"。与计数信号灯不同，互斥锁不能作为事件通知的方法。

互斥锁互相排斥

与“计数信号灯”一节的讨论类似，互相排斥用于控制线程对特定应用程序区域（也称为“关键部分”或“应用程序资源”）的访问。如果可用，ThreadX SMP 互斥锁的所有权计数就为 0。线程获得互斥锁后，在互斥时每成功执行一次“获取”操作，所有权计数就会递增一次，每成功执行一次“放置”操作则会递减一次。

创建互斥锁

ThreadX SMP 互斥锁由应用程序线程在初始化期间或运行时创建。互斥锁的初始条件始终是“可用”。还可以在选择“优先级继承”的情况下创建互斥锁。

线程挂起

当尝试对已归另一个线程所有的互斥锁执行获取操作时，应用程序线程可能会挂起。

当拥有线程执行了相同数量的获取操作后，将执行挂起线程的获取操作，为其提供互斥锁所有权，并恢复线程。如果多个线程在同一互斥锁上挂起，这些线程将按照挂起的相同顺序 (FIFO) 恢复。

但是，如果在创建期间选择了互斥优先级继承，则会自动执行优先级恢复。如果应用程序在取消线程挂起的互斥锁放置调用之前调用 tx_mutex_prioritize，还可以恢复优先级。互斥设置优先级服务将优先级最高的线程置于挂起列表的前面，让所有其他挂起的线程采用相同的 FIFO 顺序。

运行时互斥性能信息

ThreadX SMP 提供可选的运行时互斥性能信息。如果 ThreadX SMP 库和应用程序是在定义 TX_MUTEX_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX SMP 会累积以下信息：

整个系统的总数：

- 互斥锁放置数
- 互斥锁获取数
- 互斥锁获取挂起数
- 互斥锁获取超时数
- 互斥锁优先级倒置数
- 互斥锁优先级继承数

每个互斥锁的总数：

- 互斥锁放置数
- 互斥锁获取数
- 互斥锁获取挂起数
- 互斥锁获取超时数
- 互斥锁优先级倒置数
- 互斥锁优先级继承数

此信息在运行时通过 tx_mutex_performance_info_get 和 tx_mutex_performance_system_info_get 服务提供。互斥性能信息可用于确定应用程序是否正常运行。此信息对于优化应用程序也很有用。例如，“互斥锁获取超时数”相对较高可能表明其他线程占用资源的时间太长。

互斥控制块 TX_MUTEX

每个互斥锁的特征都可在其控制块中找到。该控制块包含诸如当前互斥锁所有权计数以及拥有互斥锁的线程的指针等信息。此结构在 tx_api.h 文件中定义。

互斥控制块可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

抱死

与互斥锁所有权相关的最有趣且最危险的缺陷之一是“抱死”。抱死或“死锁”是指两个或多个线程在尝试获取归其他线程所有的互斥锁时无限期挂起的情况。第 77 页讨论的“抱死”及其补救措施也完全适用于互斥对象。

优先级倒置

如前所述，与互相排斥相关的主要缺陷是优先级倒置。第 64 页的"线程优先级缺陷"更全面地讨论了本主题。

根本问题源自这种情况，即低优先级线程拥有较高优先级线程所需的信号灯。这本身很正常。但是，它们之间具有优先级的线程可能会导致优先级倒置持续不确定的时间。与前面讨论的信号灯不同，ThreadX SMP 互斥对象具有可选的"优先级继承"。优先级继承的基本思路是，优先级较低的线程将其优先级暂时提升为需要归低优先级线程所有的相同互斥锁的高优先级线程的优先级。当优先级较低的线程释放互斥锁时，即会恢复为其原始优先级，并为优先级较高的线程提供互斥锁所有权。此功能可将倒置量限制为拥有互斥锁的较低优先级线程的时间，以此来消除不确定的优先级倒置。当然，本章前面讨论的用于处理不确定的优先级倒置的技巧也适用于互斥。

事件标志

事件标志为线程同步提供了强大的工具。每个事件标志由一个位表示。事件标志按照 32 个一组的形式排列。

线程可以同时对组中的所有 32 个事件标记执行操作。事件由 tx_event_flags_set 设置，由 tx_event_flags_get 检索。

可通过在当前事件标志和新事件标志之间执行逻辑 AND/OR 运算来设置事件标志。逻辑运算的类型（AND 或 OR）在 tx_event_flags_set 调用中指定。

可使用类似的逻辑选项来检索事件标志。获取请求可以指定需要所有指定的事件标志（一个逻辑 AND）。获取请求还可以指定任何指定的事件标志都满足该请求（一个逻辑 OR）。与事件标志检索相关的逻辑运算类型在 tx_event_flags_get 调用中指定。

> **IMPORTANT**
> 如果 TX_OR_CLEAR 或 TX_AND_CLEAR 由该请求指定，则使用满足获取请求的事件标志（例如，设置为零）。

每个事件标志组都是一个公用资源。ThreadX SMP 对如何使用事件标志组没有任何限制。

创建事件标志组

事件标志组由应用程序线程在初始化期间或运行时创建。创建事件标志组时，组中的所有事件标志均设置为零。应用程序中事件标志组的数量没有限制。

线程挂起

尝试从组中获取事件标志的任意逻辑组合时，应用程序线程可能会挂起。设置事件标志后，将检查所有挂起线程的获取请求。所有现已包含所需事件标志的线程都会恢复。

> **IMPORTANT**
> 设置事件标志组的事件标志时，将检查该事件标志组中所有已挂起的线程。当然，这会带来额外的开销。因此，最好将使用同一事件标志组的线程数限制为合理的数量。

事件标志设置通知

某些应用程序可能会发现，在设置事件标志时收到通知十分有利。ThreadX SMP 通过 tx_event_flags_set_notify 服务提供此功能。此服务提供的应用程序通知函数注册到指定的事件标志组。只要在组中设置了事件标志，ThreadX SMP 就会调用此应用程序通知函数。应用程序通知函数内的确切处理由应用程序决定，但这通常包括恢复相应的线程以处理新事件标志。

事件标志 Event-chaining™

ThreadX SMP 中的通知功能可用于"链接"各种同步事件。当单个线程必须处理多个同步事件时，这通常很有用。

例如，应用程序可以为每个对象注册一个通知例程，而不是为队列消息、事件标志和信号灯而挂起单独的线程。在调用后，应用程序通知例程会恢复单个线程，该线程可以询问每个对象以查找并处理新事件。

通常情况下，"事件链"会减少线程、降低开销和减少 RAM 要求。它还提供了一种非常灵活的机制来处理更复杂系统的同步要求。

运行时事件标志性能信息

ThreadX SMP 提供可选的运行时事件标志性能信息。如果 ThreadX SMP 库和应用程序是在定义 TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX SMP 会累积以下信息：

整个系统的总数：

- 事件标志集数
- 事件标志获取数
- 事件标志获取挂起数
- 事件标志获取超时数

每个事件标志组的总数：

- 事件标志集数
- 事件标志获取数
- 事件标志获取挂起数
- 事件标志获取超时数

此信息在运行时通过 tx_event_flags_performance_info_get 和 tx_event_flags_performance_system_info_get 服务提供。事件标志性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。例如，tx_event_flags_get 服务中的超时次数相对较多，可能表明事件标志挂起超时时间太短。

事件标志组控制块 TX_EVENT_FLAGS_GROUP

每个事件标志组的特征都可在其控制块中找到。该控制块包含诸如当前事件标志设置和事件中挂起的线程数等信息。此结构在 tx_api.h 文件中定义。

事件组控制块可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

内存块池

在实时应用程序中，采用快速且确定的方式分配内存始终是一项挑战。考虑到这一点，ThreadX SMP 提供了创建和管理多个固定大小的内存块池的功能。

由于内存块池由固定大小的块组成，因此永远不会出现任何碎片问题。当然，碎片会导致出现本质上不确定的行为。此外，分配和释放固定大小内存块所需的时间与简单的链接列表操作所需的时间相当。另外，还可以在可用列表的开头完成内存块分配和取消分配。这可以提供最快的链接列表处理速度，并且有助于将实际的内存块保存在缓存中。

缺乏灵活性是固定大小内存池的主要缺点。池的块大小必须足够大，才能处理其用户最坏情况下的内存需求。当然，如果对同一个池发出了许多大小不同的内存请求，则可能会浪费内存。一种可能的解决方案是创建多个不同的内存块池，这些池包含不同大小的内存块。

每个内存块池都是一个公用资源。ThreadX SMP 对如何使用池没有任何限制。

创建内存块池

内存块池由应用程序线程在初始化期间或运行时创建。应用程序中内存块池的数量没有限制。

内存块大小

如前所述，内存块池包含许多固定大小的块。块大小（以字节为单位）在创建池时指定。

> **IMPORTANT**
> ThreadX SMP 为池中的每个内存块增加了少量开销(C 指针的大小)。此外，ThreadX SMP 可能需要填充块大小，从而确保每个内存块的开头能够正确对齐。

池容量

池中的内存块数是在创建过程中提供的内存区域的块大小和总字节数的函数。池容量的计算方法是将块大小(包括填充和指针开销字节)除以提供的内存区域的总字节数。

池的内存区域

如前所述，块池的内存区域在创建时指定。与 ThreadX SMP 中的其他内存区域一样，该区域可以位于目标地址空间的任何位置。

这是一项重要的功能，因为它提供了相当大的灵活性。例如，假设某个通信产品有一个用于 I/O 的高速内存区域。将此内存区域设置为 ThreadX SMP 内存块池，即可轻松对其进行管理。

线程挂起

在等待空池中的内存块时，应用程序线程可能会挂起。当块返回到池时，将为挂起的线程提供此块，并恢复线程。

如果同一内存块池中挂起多个线程，这些线程将按挂起的顺序 (FIFO) 恢复。

不过，如果应用程序在取消线程挂起的块释放调用之前调用 tx_block_pool_prioritize，还可以恢复优先级。块池设置优先级服务将优先级最高的线程置于挂起列表的前面，让所有其他挂起的线程采用相同的 FIFO 顺序。

运行时块池性能信息

ThreadX SMP 提供可选的运行时块池性能信息。如果 ThreadX SMP 库和应用程序是在定义 TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX SMP 会累积以下信息：

整个系统的总数：

- 已分配的块数
- 已释放的块数
- 分配挂起数
- 分配超时数

每个块池的总数：

- 已分配的块数
- 已释放的块数
- 分配挂起数
- 分配超时数

此信息在运行时通过 tx_block_pool_performance_info_get 和 tx_block_pool_performance_system_info_get 服务提供。块池性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。例如，“分配挂起数”相对较高可能表明块池太小。

内存块池控制块 TX_BLOCK_POOL

每个内存块池的特征都可在其控制块中找到。该控制块包含诸如可用的内存块数和内存池块大小等信息。此结构在 tx_api.h 文件中定义。

池控制块也可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

覆盖内存块

务必确保已分配内存块的用户不会在其边界之外写入。如果发生这种情况，则会损坏其相邻的内存区域(通常是

后续区域）。其结果不可预测，通常很严重！

内存字节池

ThreadX SMP 内存字节池与标准 C 堆类似。与标准 C 堆的不同之处在于，该内存字节池可以包含多个内存字节池。此外，线程可在池中挂起，直到请求的内存可用为止。

内存字节池的分配与传统的 malloc 调用类似，其中包含所需的内存量（以字节为单位）。内存采用“首次适应”的方式从池中分配；例如，使用满足请求的第一个可用内存块。此块中多余的内存会转换为新块，并放回可用内存列表中。此过程称为“碎片”。

相邻的可用内存块在后续的分配搜索过程中“合并”为一个足够大的可用内存块。此过程称为“碎片整理”。

每个内存字节池都是一个公用资源。除了不能从 ISR 调用内存字节服务之外，ThreadX SMP 对如何使用池没有任何限制。

创建内存字节池

内存字节池由应用程序线程在初始化期间或运行时创建。应用程序中内存字节池的数量没有限制。

池容量

内存字节池中可分配的字节数略小于创建期间指定的字节数。这是因为可用内存区域的管理带来了一些开销。池中的每个可用内存块都需要相当于两个 C 指针的开销。此外，创建的池包含两个块：一个较大的可用块和在内存区域末端永久分配的一个较小的块。这个分配块用于提高分配算法的性能。这样就无需在合并期间持续检查池区域末端。

在运行时，池中的开销通常会增加。如果分配奇数字节数，系统会加以填充，以确保正确对齐下一个内存块。此外，随着池变得更加零碎，开销也会增加。

池的内存区域

内存字节池的内存区域在创建过程中指定。与 ThreadX SMP 中的其他内存区域一样，该区域可以位于目标地址空间的任何位置。

这是一项重要的功能，因为它提供了相当大的灵活性。例如，如果目标硬件有高速内存区域和低速内存区域，用户可以通过在每个区域中创建池来管理这两个区域的内存分配。

线程挂起

在等待池中的内存字节时，应用程序线程可能会挂起。当有足够的连续内存可用时，将为已挂起的线程提供其请求的内存，并且恢复线程。

如果同一内存字节池中挂起多个线程，则按这些线程挂起的顺序 (FIFO) 为其提供内存（恢复）。

不过，如果应用程序在信号灯发出取消线程挂起的字节释放调用之前调用 tx_byte_pool_prioritize，还可以恢复优先级。字节池设置优先级服务将最高优先级的线程置于挂起列表的前面，让所有其他挂起的线程采用相同的 FIFO 顺序。

运行时字节池性能信息

ThreadX SMP 提供可选的运行时字节池性能信息。如果 ThreadX SMP 库和应用程序是在定义 TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX SMP 会累积以下信息：

整个系统的总数：

- 分配数
- 版本
- 搜索的片段数
- 合并的片段数
- 创建的片段数
- 分配挂起数

- 分配超时数

每个字节池的总数：

- 分配数
- 版本
- 搜索的片段数
- 合并的片段数
- 创建的片段数
- 分配挂起数
- 分配超时数

此信息在运行时通过 tx_byte_pool_performance_info_get 和 tx_byte_pool_performance_system_info_get 服务提供。字节池性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。例如，“分配挂起数”相对较高可能表明字节池太小。

内存字节池控制块 TX_BYTE_POOL

每个内存字节池的特征都可在其控制块中找到。该控制块包含诸如池中可用的字节数等有用的信息。此结构在 tx_api.h 文件中定义。

池控制块也可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

非确定性行为

尽管内存字节池提供了最灵活的内存分配，但这些池也受一些非确定性行为的影响。例如，内存字节池可能有 2,000 字节的可用内存，但可能无法满足 1,000 字节的分配请求。这是因为无法保证有多少可用字节是连续的。即使存在 1,000 字节可用块，也不能保证找到此块需要多长时间。完全有可能需要搜索整个内存池来查找这个 1,000 字节块。

> **IMPORTANT**
> 因此，通常应避免在需要确定性实时行为的区域中使用内存字节服务。许多应用程序在初始化或运行时配置期间预先分配其所需的内存。

覆盖内存块

务必确保已分配内存的用户不会在其边界之外写入。如果发生这种情况，则会损坏其相邻的内存区域（通常是后续区域）。其结果不可预测，通常很严重！

应用程序计时器

快速响应异步外部事件是嵌入式实时应用程序中最重要的功能。但是，其中的许多应用程序还必须按预定的时间间隔执行某些活动。

借助 ThreadX SMP 应用程序计时器，应用程序能够按特定的时间间隔执行应用程序 C 函数。应用程序计时器也可能只过期一次。这种类型的计时器称为“单次计时器”，而重复间隔计时器称为“定期计时器”。

每个应用程序计时器都是一个公用资源。ThreadX SMP 对如何使用应用程序计时器没有任何限制。

> **IMPORTANT**
> 通过 tx_timer_smp_core_exclude API，可将应用程序计时器排除在任何核心上的执行之外。

计时器间隔

在 ThreadX SMP 中，时间间隔通过定期计时器中断来测量。每个计时器中断称为计时器时钟周期。计时器时钟

周期之间的实际时间由应用程序指定，但 10 毫秒是大多数实现的标准时间。定期计时器设置通常位于 tx_initialize_low_level 程序集文件中。

值得一提的是，基础硬件必须能够生成定期中断，应用程序计时器才会正常运行。在某些情况下，处理器具有内置的定期中断功能。如果处理器没有此功能，用户的主板必须包含可生成定期中断的外围设备。

> **IMPORTANT**
> 即使没有定期中断源，ThreadX SMP 仍可正常工作。但随后会禁用所有与计时器相关的处理。这包括时间切片、挂起超时和计时器服务。

计时器准确性

计时器过期时间根据时钟周期指定。达到每个计时器时钟周期时，指定到期值将减一。由于应用程序计时器可在计时器中断（或计时器时钟周期）之前启用，因此，实际过期时间可能会提前一个时钟周期。

如果计时器时钟周期速率为 10 毫秒，应用程序计时器可能会提前 10 毫秒过期。与 1 秒计时器相比，这对 10 毫秒计时器更重要。当然，增加计时器中断频率会减少此误差范围。

计时器执行

应用程序计时器按照其激活的顺序执行。例如，如果创建了三个具有相同过期值的计时器并已激活，这些计时器对应的过期函数将保证按它们激活的顺序执行。

创建应用程序计时器

应用程序计时器由应用程序线程在初始化期间或运行时创建。应用程序中应用程序计时器的数量没有限制。

运行时应用程序计时器性能信息

ThreadX SMP 提供可选的运行时应用程序计时器性能信息。如果 ThreadX SMP 库和应用程序是在定义 TX_TIMER_ENABLE_PERFORMANCE_INFO 的情况下生成的，ThreadX SMP 会累积以下信息：

整个系统的总数：

- 激活数
- 停用数
- 重新激活（定期计时器）
- expirations
- 过期调整

每个应用程序计时器的总数：

- 激活数
- 停用数
- 重新激活（定期计时器）
- expirations
- 过期调整

此信息在运行时通过 tx_timer_performance_info_get 和 tx_timer_performance_system_info_get 服务提供。应用程序计时器性能信息在确定应用程序是否正常运行时非常有用。此信息对于优化应用程序也很有用。

应用程序计时器控制块 TX_TIMER

每个应用程序计时器的特征都可在其控制块中找到。该控制块包含诸如 32 位过期标识值等有用信息。此结构在 tx_api.h 文件中定义。

应用程序计时器控制块可以位于内存中的任意位置，但最常见的是通过在任何函数的作用域外部定义该控件块来使其成为全局结构。

计时器过多

默认情况下，应用程序计时器在优先级为 0 时运行的隐藏系统线程中执行，该线程的优先级通常比任何应用程序线程都高。因此，在应用程序计时器内进行处理应保持最小值。

如果可能，还应尽可能避免使用在每个时钟周期过期的计时器。这种情况可能导致应用程序的开销过大。

> **WARNING**
> 如前所述，应用程序计时器在隐藏的系统线程中执行。因此，请不要在应用程序计时器的过期函数内执行任何 ThreadX SMP 服务调用时选择挂起。

相对时间

除了前面所述的应用程序计时器，ThreadX SMP 还提供单个连续递增的 32 位时钟周期计数器。每次发生计时器中断时，时钟周期计数器或“时间”就会加一。

应用程序可以通过分别调用 tx_time_get 和 tx_time_set 来读取或设置此 32 位计数器。此时钟周期计数器的使用完全由应用程序确定。ThreadX SMP 不在内部使用此计时器。

中断

快速响应异步事件是嵌入式实时应用程序的主函数。应用程序知道此类事件是因为硬件中断而出现的。

中断是处理器执行中的异步更改。发生中断时，处理器通常会将当前执行的一小部分保存在堆栈上，并将控制权转交给相应的中断向量。中断向量基本上只是负责处理特定类型中断的例程的地址。确切的中断处理过程特定于处理器。

中断控制

使用 tx_interrupt_control 服务，应用程序可以启用和禁用中断。上一个中断启用/禁用状态由此服务返回。值得一提的是，中断控制只会影响当前正在执行的程序段。例如，如果某个线程禁用中断，这些中断仅在该线程执行期间保持禁用状态。

> **WARNING**
> 不可屏蔽的中断 (NMI) 是无法通过硬件禁用的中断。此类中断可供 ThreadX SMP 应用程序使用。但是，不允许应用程序的 NMI 处理例程使用 ThreadX SMP 上下文管理或任何 API 服务。ThreadX SMP 托管中断

ThreadX SMP 提供具有完整中断管理的应用程序。此管理包括保存和还原中断执行的上下文。此外，ThreadX SMP 允许在中断服务例程 (ISR) 内调用特定服务。下面是允许从应用程序 ISR 调用的 ThreadX SMP 服务的列表：

- tx_block_allocate
- tx_block_pool_info_get
- tx_block_pool_prioritize
- tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_release
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize
- tx_event_flags_info_get
- tx_event_flags_get
- tx_event_flags_set

- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set_notify
- tx_interrupt_control
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send
- tx_semaphore_get
- tx_queue_send_notify
- tx_semaphore_ceiling_put
- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_thread_identify
- tx_semaphore_put_notify
- tx_thread_entry_exit_notify
- tx_thread_info_get
- tx_thread_resume
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_stack_error_notify
- tx_thread_wait_abort
- tx_time_get
- tx_time_set
- tx_timer_activate
- tx_timer_change
- tx_timer_deactivate
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

> **WARNING**
> 不允许从 ISR 中暂停。因此，从 ISR 发出的所有 ThreadX SMP 服务调用的 wait_option 参数必须设置为 TX_NO_WAIT。

ISR 模板

若要管理应用程序中断，必须在应用程序 ISR 的开头和结尾调用多个 ThreadX SMP 实用程序。中断处理的确切格式因端口而异。有关管理 ISR 的具体说明，请参阅安装盘上的 readme_threadx.txt 文件。

以下小代码段是大多数 ThreadX SMP 托管 ISR 的典型代码段。在大多数情况下，此处理采用汇编语言。

```
_application_ISR_vector_entry:
;保存上下文并通过
;调用 ISR entry 函数来
;使用 ThreadX SMP。
CALL _tx_thread_context_save

;ISR 现在可以调用 ThreadX SMP
;服务及其 C 函数

;完成 ISR 后，
;将通过调用上下文还原函数
;（或线程抢占）还原
;上下文。控制权不返回！
JUMP _tx_thread_context_restore
```

高频中断

某些中断发生的频率很高，导致在每次中断时保存和还原完整上下文消耗过多的处理带宽。在这种情况下，应用程序通常有一种小型汇编语言 ISR，用于对大多数此类高频中断执行一定量的处理。

在某个时间点之后，这种小型 ISR 可能需要与 ThreadX SMP 交互。这种交互通过调用上述模板所述的入口和出口函数来实现。

中断延迟

ThreadX SMP 在短时间内锁定中断。禁用中断的最大时间与保存或还原线程上下文所需的时间大致相同。

第 4 章 - Azure RTOS ThreadX SMP 服务的说明

2021/4/30 •

本章按字母顺序介绍所有 Azure RTOS ThreadX SMP 服务。它们的名称设计为将所有相似的服务组合在一起。在以下说明的“返回值”部分中，以粗体显示的值不受用于禁用 API 错误检查的 NX_DISABLE_ERROR_CHECKING 定义影响，而不以粗体显示的值则会被完全禁用。此外，“可以抢占”标题下列出的“是”表示调用该服务可能会恢复更高优先级的线程，从而抢占调用线程 。

- tx_block_allocate：分配固定大小的内存块
- tx_block_pool_create：创建固定大小的内存块池
- tx_block_pool_delete：删除内存块池
- tx_block_pool_info_get：检索有关块池的信息
- tx_block_pool_performance_info_get：获取块池性能信息
- tx_block_pool_performance_system_info_get：获取块池系统性能信息
- tx_block_pool_prioritize：设置块池挂起列表的优先级
- tx_block_release：释放固定大小的内存块
- tx_byte_allocate：分配内存的字节数
- tx_byte_pool_create：创建内存字节池
- tx_byte_pool_delete：删除内存字节池
- tx_byte_pool_info_get：检索有关字节池的信息
- tx_byte_pool_performance_info_get：获取字节池性能信息
- tx_byte_pool_performance_system_info_get：获取字节池系统性能信息
- tx_byte_pool_prioritize：设置字节池挂起列表的优先级
- tx_byte_release：将字节释放回内存池
- tx_event_flags_create创建事件标志组
- tx_event_flags_delete：删除事件标志组
- tx_event_flags_get：从事件标志组获取事件标志
- tx_event_flags_info_get：检索有关事件标志组的信息
- tx_event_flags_performance_info_get：获取事件标志组性能信息
- tx_event_flags_performance_system_info_get：检索系统性能信息
- tx_event_flags_set：在事件标志组中设置事件标志
- tx_event_flags_set_notify：在设置事件标志时通知应用程序
- tx_interrupt_control：启用和禁用中断
- tx_mutex_create：创建互斥锁
- tx_mutex_delete：删除互斥锁
- tx_mutex_get：获取互斥锁的所有权
- tx_mutex_info_get：检索有关互斥锁的信息
- tx_mutex_performance_info_get：获取互斥锁性能信息
- tx_mutex_performance_system_info_get：获取互斥锁系统性能信息
- tx_mutex_prioritize：设置互斥锁挂起列表的优先级
- tx_mutex_put：释放互斥锁的所有权
- tx_queue_create：创建消息队列
- tx_queue_delete：删除消息队列
- tx_queue_flush：清空消息队列中的消息
- tx_queue_front_send：将消息发送到队列的前面

- tx_queue_info_get：检索有关队列的信息
- tx_queue_performance_info_get：获取队列性能信息
- tx_queue_performance_system_info_get：获取队列系统性能信息
- tx_queue_prioritize：设置队列挂起列表的优先级
- tx_queue_receive：从消息队列获取消息
- tx_queue_send：将消息发送到消息队列
- tx_queue_send_notify：在将消息发送到队列时通知应用程序
- tx_semaphore_ceiling_put：将实例放入具有上限的计数信号灯
- tx_semaphore_create：创建计数信号灯
- tx_semaphore_delete：删除计数信号灯
- tx_semaphore_get：从计数信号灯获取实例
- tx_semaphore_info_get：检索有关信号灯的信息
- tx_semaphore_performance_info_get：获取信号灯性能信息
- tx_semaphore_performance_system_info_get：获取信号灯系统性能信息
- tx_semaphore_prioritize：设置信号灯挂起列表的优先级
- tx_semaphore_put：将实例放入计数信号灯
- tx_semaphore_put_notify：放置信号灯时通知应用程序
- tx_thread_create：创建应用程序线程
- tx_thread_delete：删除应用程序线程
- tx_thread_entry_exit_notify：在线程进入和退出时通知应用程序
- tx_thread_identify：检索指向当前正在执行的线程的指针
- tx_thread_info_get：检索有关线程的信息
- tx_thread_performance_info_get：获取线程性能信息
- tx_thread_performance_system_info_get：获取线程系统性能信息
- tx_thread_preemption_change：更改应用程序线程的抢占阈值
- tx_thread_priority_change：更改应用程序线程的优先级
- tx_thread_relinquish：将控制权让给其他应用程序线程
- tx_thread_reset：重置线程
- tx_thread_resume：恢复挂起的应用程序线程
- tx_thread_sleep：在指定的时间内挂起当前线程
- tx_thread_smp_core_exclude：在一组内核上排除线程执行
- tx_thread_smp_core_exclude_get：获取线程的当前内核排除
- tx_thread_smp_core_get：检索调用方当前正在执行的内核
- tx_thread_stack_error_notify：注册线程堆栈错误通知回调
- tx_thread_suspend：挂起应用程序线程
- tx_thread_terminate：终止应用程序线程
- tx_thread_time_slice_change：更改应用程序线程的时间片
- tx_thread_wait_abort：中止指定线程的挂起
- tx_time_get：检索当前时间
- tx_time_set：设置当前时间
- tx_timer_activate：激活应用程序计时器
- tx_timer_change：更改应用程序计时器
- tx_timer_create：创建应用程序计时器
- tx_timer_deactivate：停用应用程序计时器
- tx_timer_delete：删除应用程序计时器
- tx_timer_info_get：检索有关应用程序计时器的信息

- tx_timer_performance_info_get：获取计时器性能信息
- tx_timer_performance_system_info_get：获取计时器系统性能信息
- tx_timer_smp_core_exclude：在一组内核上排除计时器执行
- tx_timer_smp_core_exclude_get：获取计时器的当前内核排除

tx_block_allocate

分配固定大小的内存块

原型

```
UINT tx_block_allocate(TX_BLOCK_POOL *pool_ptr, VOID **block_ptr,
                       ULONG wait_option);
```

说明

此服务从指定的内存池中分配固定大小的内存块。内存块的实际大小是在创建内存池的过程中确定的。

> **WARNING**
> 务必确保应用程序代码不在已分配的内存块之外写入。如果发生这种情况，则会损坏其相邻的内存块（通常是后续块）。结果不可预测，且通常很严重！

参数

- pool_ptr：指向之前创建的内存块池的指针。
- block_ptr：指向目标块的指针。成功分配时，已分配内存块的地址将放置在此参数所指向的位置。
- wait_option：定义此服务在没有任何内存块可用时的行为方式。等待选项定义如下：
  - TX_NO_WAIT：(0x00000000)
  - TX_WAIT_FOREVER：(0xFFFFFFFF)
  - 超时值：(0x00000001 到 0xFFFFFFFE)

  如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从非线程（例如初始化、计时器或 ISR）调用服务，则这是唯一有效的选项。

  选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到内存块可用为止。

  如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待内存块时调用线程保持挂起的最大计时器时钟周期数。

返回值

- TX_SUCCESS：(0x00) 成功分配内存块。
- TX_DELETED：(0x01) 线程挂起时删除了内存块池。
- TX_NO_MEMORY：(0x10) 服务无法在指定的等待时间内分配内存块。
- TX_WAIT_ABORTED：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- TX_POOL_ERROR：(0x02) 内存块池指针无效。
- TX_PTR_ERROR：(0x03) 指向目标的指针无效。
- TX_WAIT_ERROR：(0x04) 从非线程调用时指定了 TX_NO_WAIT 以外的等待选项。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_BLOCK_POOL   my_pool;
unsigned char*memory_ptr;
UINT         status;

/* Allocate a memory block from my_pool. Assume that the
   pool has already been created with a call to
   tx_block_pool_create. */
status = tx_block_allocate(&my_pool, (VOID **) &memory_ptr,
                           TX_NO_WAIT);

/* If status equals TX_SUCCESS, memory_ptr contains the
   address of the allocated block of memory. */
```

另请参阅

- tx_block_pool_create
- tx_block_pool_delete
- tx_block_pool_info_get
- tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_pool_prioritize
- tx_block_release

tx_block_pool_create

创建固定大小内存块的池

原型

```
UINT tx_block_pool_create(TX_BLOCK_POOL *pool_ptr,
                          CHAR *name_ptr, ULONG block_size,
                          VOID *pool_start, ULONG pool_size);
```

说明

此服务创建固定大小内存块的池。使用以下公式将指定的内存区域划分为尽可能多的固定大小内存块：
总块数 =（总字节数）/（块大小 + sizeof(void *)）

> **IMPORTANT**
> 每个内存块包含一个开销指针，该指针对用户不可见，在前面的公式中用“sizeof(void *)”表示。

参数

- pool_ptr：指向内存块池控制块的指针。
- name_ptr：指向内存块池名称的指针。
- block_size：每个内存块中的字节数。
- pool_start：内存块池的起始地址。起始地址必须与 ULONG 数据类型的大小一致。
- pool_size：内存块池可用的总字节数。

返回值

- TX_SUCCESS：(0x00) 成功创建内存块池。
- TX_POOL_ERROR：(0x02) 内存块池指针无效。指针为 NULL 或已创建池。
- TX_PTR_ERROR：(0x03) 池的起始地址无效。

- TX_SIZE_ERROR：(0x05) 池大小无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

初始化和线程

可以抢占

否

示例

```
TX_BLOCK_POOL  my_pool;
UINT           status;

/* Create a memory pool whose total size is 1000 bytes
   starting at address 0x100000. Each block in this
   pool is defined to be 50 bytes long. */
status = tx_block_pool_create(&my_pool, "my_pool_name",
               50, (VOID *) 0x100000, 1000);

/* If status equals TX_SUCCESS, my_pool contains 18
   memory blocks of 50 bytes each. The reason
   there are not 20 blocks in the pool is
   because of the one overhead pointer associated with each
   block. */
```

另请参阅

- tx_block_allocate
- tx_block_pool_delete
- tx_block_pool_info_get
- tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_pool_prioritize
- tx_block_release

tx_block_pool_delete

删除内存块池

原型

```
UINT tx_block_pool_delete(TX_BLOCK_POOL *pool_ptr);
```

说明

此服务删除指定的块内存池。所有挂起并等待此池中的内存块的线程都将恢复，并获得 TX_DELETED 返回状态。

> **IMPORTANT**
> 应用程序负责管理与池关联的内存区域，该内存区域在此服务完成后可用。此外，应用程序必须阻止使用已删除的池或其以前的内存块。

参数

- pool_ptr：指向之前创建的内存块池的指针。

返回值

- TX_SUCCESS：(0x00) 成功删除内存块池。
- TX_POOL_ERROR：(0x02) 内存块池指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程数

可以抢占

是

示例

```
TX_BLOCK_POOLmy_pool;
UINT           status;

    /* Delete entire memory block pool. Assume that the pool
      has already been created with a call to
      tx_block_pool_create. */
    status =  tx_block_pool_delete(&my_pool);

    /* If status equals TX_SUCCESS, the memory block pool is
       deleted. */
```

另请参阅

- tx_block_allocate
- tx_block_pool_create
- tx_block_pool_info_get
- tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_pool_prioritize
- tx_block_release

tx_block_pool_info_get

检索有关块池的信息

原型

```
UINT tx_block_pool_info_get(TX_BLOCK_POOL *pool_ptr, CHAR **name,
                            ULONG *available, ULONG *total_blocks,
                            TX_THREAD **first_suspended,
                            ULONG *suspended_count,
                            TX_BLOCK_POOL **next_pool);
```

说明

此服务检索所指定块内存池的相关信息。

参数

- pool_ptr：指向之前创建的内存块池的指针。
- name：指向块池名称的指针。
- available：指向块池中的可用块数的指针。
- total_blocks：指向块池中的总块数的指针。
- first_suspended：指向此块池的挂起列表上第一个线程的指针。
- suspended_count：指向此块池中当前挂起的线程数的指针。

- next_pool：指向下一个已创建的块池的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功检索块池信息。
- TX_POOL_ERROR：(0x02) 内存块池指针无效。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_BLOCK_POOL    my_pool;
CHAR             *name;
ULONG            available;
ULONG            total_blocks;
TX_THREAD        *first_suspended;
ULONG            suspended_count;
TX_BLOCK_POOL    *next_pool;
UINT             status;

/* Retrieve information about the previously created
   block pool "my_pool." */
status = tx_block_pool_info_get(&my_pool, &name,
                &available,&total_blocks,
                &first_suspended, &suspended_count,
                &next_pool);

/* If status equals TX_SUCCESS, the information requested is
   valid. */
```

另请参阅

- tx_block_allocate
- tx_block_pool_create
- tx_block_pool_delete
- tx_block_pool_info_get
- tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_pool_prioritize
- tx_block_release

tx_block_pool_performance_info_get

获取块池性能信息

原型

```
UINT tx_block_pool_performance_info_get(TX_BLOCK_POOL *pool_ptr,
       ULONG *allocates, ULONG *releases,
       ULONG *suspensions, ULONG *timeouts));
```

说明

此服务检索所指定内存块池的相关性能信息。

> **IMPORTANT**
>
> 必须使用已定义的 TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- pool_ptr：指向之前创建的内存块池的指针。
- allocates：指向对此池执行的分配请求数的指针。
- releases：指向对此池执行的释放请求数的指针。
- suspensions：指向此池的线程分配挂起数的指针。
- timeouts：指向此池的分配挂起超时次数的指针。

> **IMPORTANT**
>
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取块池性能信息。
- TX_PTR_ERROR：(0x03) 块池指针无效。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_BLOCK_POOL     my_pool;
ULONG             allocates;
ULONG             releases;
ULONG             suspensions;
ULONG             timeouts;

/* Retrieve performance information on the previously created block
   pool. */
status = tx_block_pool_performance_info_get(&my_pool, &allocates,
                                            &releases,
                &suspensions,
                &timeouts);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_block_allocate
- tx_block_pool_create
- tx_block_pool_delete
- tx_block_pool_info_get
- tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_release

tx_block_pool_performance_system_info_get

获取块池系统性能信息

原型

```
UINT tx_block_pool_performance_system_info_get(ULONG *allocates,
       ULONG *releases, ULONG *suspensions, ULONG *timeouts);
```

说明

此服务检索有关应用程序中所有内存块池的性能信息。

> **IMPORTANT**
> 必须使用已定义的 TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- allocates：指向对所有块池执行的分配请求总数的指针。
- releases：指向对所有块池执行的释放请求总数的指针。
- suspensions：指向所有块池的线程分配挂起总次数的指针。
- timeouts：指向所有块池的分配挂起超时总次数的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取块池系统性能信息。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
ULONG         allocates;
ULONG         releases;
ULONG         suspensions;
ULONG         timeouts;

/* Retrieve performance information on all the block pools in
   the system. */
status = tx_block_pool_performance_system_info_get(&allocates,
                    &releases,&suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_block_allocate
- tx_block_pool_create
- tx_block_pool_delete
- tx_block_pool_info_get
- tx_block_pool_performance_info_get

- tx_block_pool_prioritize
- tx_block_release

tx_block_pool_prioritize

设置块池挂起列表的优先级

原型

```
UINT tx_block_pool_prioritize(TX_BLOCK_POOL *pool_ptr);
```

说明

此服务将此池中某个内存块已挂起的最高优先级线程放在挂起列表前面。所有其他线程保持它们挂起时的相同 FIFO 顺序。

参数

- pool_ptr：指向内存块池控制块的指针。

返回值

- TX_SUCCESS：(0x00) 成功为块池设置优先级。
- TX_POOL_ERROR：(0x02) 内存块池指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_BLOCK_POOL my_pool;
UINT          status;

/* Ensure that the highest priority thread will receive
   the next free block in this pool. */
status = tx_block_pool_prioritize(&my_pool);

/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_block_release call will wake up this thread. */
```

另请参阅

- tx_block_allocate
- tx_block_pool_create
- tx_block_pool_delete
- tx_block_pool_info_get
- tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_release

tx_block_release

释放固定大小的内存块

原型

```
UINT tx_block_release(VOID *block_ptr);
```

说明

此服务将以前分配的块释放回其关联的内存池。如果有一个或多个线程挂起并等待此池中的内存块，则为第一个挂起的线程提供此内存块并恢复该线程。

> **IMPORTANT**
> 内存块区域已释放回池中后，应用程序应阻止使用该内存块区域。

参数

- block_ptr：指向之前分配的内存块的指针。

返回值

- TX_SUCCESS：(0x00) 成功释放内存块。
- TX_PTR_ERROR：(0x03) 指向内存块的指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_BLOCK_POOLmy_pool;
unsigned char*memory_ptr;
UINT         status;

/* Release a memory block back to my_pool. Assume that the
   pool has been created and the memory block has been
   allocated. */
status = tx_block_release((VOID *) memory_ptr);

/* If status equals TX_SUCCESS, the block of memory pointed
   to by memory_ptr has been returned to the pool. */
```

另请参阅

- tx_block_allocate
- tx_block_pool_create
- tx_block_pool_delete
- tx_block_pool_info_get
- tx_block_pool_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_pool_prioritize

tx_byte_allocate

分配内存的字节数

原型

```
UINT tx_byte_allocate(TX_BYTE_POOL *pool_ptr,
                          VOID **memory_ptr, ULONG memory_size,
                          ULONG wait_option);
```

说明

此服务从指定的内存字节池中分配指定的字节数。

> **WARNING**
> 务必确保应用程序代码不在已分配的内存块之外写入。如果发生这种情况，则会损坏其相邻的内存块（通常是后续块）。结果不可预测，且通常很严重！

> **IMPORTANT**
> 此服务的性能是块大小和池中碎片量的函数。因此，在执行时间关键线程期间不应使用此服务。

参数

- pool_ptr：指向之前创建的内存池的指针。
- memory_ptr：指向目标内存的指针。成功分配时，已分配内存区域的地址将放置在此参数所指向的位置。
- memory_size：所请求的字节数。
- wait_option：定义此服务在没有足够内存可用时的行为方式。等待选项定义如下：
  - TX_NO_WAIT：(0x00000000)
  - TX_WAIT_FOREVER：(0xFFFFFFFF)
  - 超时值：（0x00000001 到 0xFFFFFFFE）

  如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从初始化调用服务，则这是唯一有效的选项。

  选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到有足够内存可用为止。

  如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待内存时调用线程保持挂起的最大计时器时钟周期数。

返回值

- TX_SUCCESS：(0x00) 成功分配内存。
- TX_DELETED：(0x01) 线程挂起时删除了内存池。
- TX_NO_MEMORY：(0X10) 服务无法在指定的等待时间内分配内存。
- TX_WAIT_ABORTED：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- TX_POOL_ERROR：(0x02) 内存池指针无效。
- TX_PTR_ERROR：(0x03) 指向目标的指针无效。
- TX_SIZE_ERROR：(0x05) 所请求的大小为零或超过池大小。
- TX_WAIT_ERROR：(0x04) 从非线程调用时指定了 TX_NO_WAIT 以外的等待选项。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

初始化和线程

可以抢占

是

示例

```
TX_BYTE_POOL my_pool;
unsigned char*memory_ptr;
UINT         status;

/* Allocate a 112 byte memory area from my_pool. Assume
   that the pool has already been created with a call to
   tx_byte_pool_create. */
status =  tx_byte_allocate(&my_pool, (VOID **) &memory_ptr,
                      112, TX_NO_WAIT);

/* If status equals TX_SUCCESS, memory_ptr contains the
   address of the allocated memory area. */
```

另请参阅

- tx_byte_pool_create
- tx_byte_pool_delete
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize
- tx_byte_release

tx_byte_pool_create

创建内存字节池

原型

```
UINT tx_byte_pool_create(TX_BYTE_POOL *pool_ptr,
                         CHAR *name_ptr, VOID *pool_start,
                         ULONG pool_size);
```

说明

此服务在指定的区域中创建内存字节池。基本上来说，池最初包含一个非常大的可用块。但是，随着不断进行分配，池将分成更小的块。

参数

- pool_ptr：指向内存池控制块的指针。
- name_ptr：指向内存池名称的指针。
- pool_start：内存池的起始地址。起始地址必须与 ULONG 数据类型的大小一致。
- pool_size：内存池可用的总字节数。

返回值

- TX_SUCCESS：(0x00) 成功创建内存池。
- TX_POOL_ERROR：(0x02) 内存池指针无效。指针为 NULL 或已创建池。
- TX_PTR_ERROR：(0x03) 池的起始地址无效。
- TX_SIZE_ERROR：(0x05) 池大小无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

初始化和线程

可以抢占

否

示例

```
TX_BYTE_POOL my_pool;
UINT         status;

/* Create a memory pool whose total size is 2000 bytes
   starting at address 0x500000. */
status =  tx_byte_pool_create(&my_pool, "my_pool_name",
             (VOID *) 0x500000, 2000);

/* If status equals TX_SUCCESS, my_pool is available for
   allocating memory. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_delete
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize
- tx_byte_release

tx_byte_pool_delete

删除内存字节池

原型

```
UINT tx_byte_pool_delete(TX_BYTE_POOL *pool_ptr);
```

说明

此服务删除指定的内存字节池。所有挂起并等待此池中的内存的线程都将恢复，并获得 TX_DELETED 返回状态。

> **IMPORTANT**
> 应用程序负责管理与池关联的内存区域，该内存区域在此服务完成后可用。此外，应用程序必须阻止使用已删除的池或先前从中分配的内存。

参数

- pool_ptr：指向之前创建的内存池的指针。

返回值

- TX_SUCCESS：(0X00) 成功删除内存池。
- TX_POOL_ERROR：(0x02) 内存池指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程数

可以抢占

是

示例

```
TX_BYTE_POOL my_pool;
UINT         status;

/* Delete entire memory pool. Assume that the pool has already
   been created with a call to tx_byte_pool_create. */
status =   tx_byte_pool_delete(&my_pool);

/* If status equals TX_SUCCESS, memory pool is deleted. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_create
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize
- tx_byte_release

tx_byte_pool_info_get

检索有关字节池的信息

原型

```
UINT tx_byte_pool_info_get(TX_BYTE_POOL *pool_ptr, CHAR **name,
                           ULONG *available, ULONG *fragments,
                           TX_THREAD **first_suspended,
                           ULONG *suspended_count,
                           TX_BYTE_POOL **next_pool);
```

说明

此服务检索所指定内存字节池的相关信息。

参数

- pool_ptr：指向之前创建的内存池的指针。
- name：指向字节池名称的指针。
- available：指向池中的可用字节数的指针。
- fragments：指向字节池中内存片段总数的指针。
- first_suspended：指向此字节池的挂起列表上第一个线程的指针。
- suspended_count：指向此字节池中当前挂起的线程数的指针。
- next_pool：指向下一个已创建的字节池的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功检索池信息。
- TX_POOL_ERROR：(0x02) 内存池指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_BYTE_POOL my_pool;
CHAR         *name;
ULONG        available;
ULONG        fragments;
TX_THREAD    *first_suspended;
ULONG        suspended_count;
TX_BYTE_POOL *next_pool;
UINT         status;

/* Retrieve information about the previously created
   block pool "my_pool." */
status =  tx_byte_pool_info_get(&my_pool, &name,
             &available, &fragments,
             &first_suspended, &suspended_count,
             &next_pool);

/* If status equals TX_SUCCESS, the information requested is
   valid. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_create
- tx_byte_pool_delete
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize
- tx_byte_release

tx_byte_pool_performance_info_get

获取字节池性能信息

原型

```
UINT tx_byte_pool_performance_info_get(TX_BYTE_POOL *pool_ptr,
        ULONG *allocates, ULONG *releases,
        ULONG *fragments_searched, ULONG *merges, ULONG *splits,
        ULONG *suspensions, ULONG *timeouts);
```

说明

此服务检索所指定内存字节池的相关性能信息。

> **IMPORTANT**
> 必须使用已定义的 TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- pool_ptr：指向之前创建的内存字节池的指针。

- allocates：指向对此池执行的分配请求数的指针。
- releases：指向对此池执行的释放请求数的指针。
- fragments_searched：指向对此池执行分配请求期间搜索到的内部内存片段数的指针。
- merges：指向对此池执行分配请求期间合并的内部内存块数的指针。
- splits：指向对此池执行分配请求期间创建的内部内存块拆分（片段）数的指针。
- suspensions：指向此池的线程分配挂起数的指针。
- timeouts：指向此池的分配挂起超时次数的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取字节池性能信息。
- TX_PTR_ERROR：(0x03) 字节池指针无效。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_BYTE_POOL     my_pool;
ULONG            fragments_searched;
ULONG            merges;
ULONG            splits;
ULONG            allocates;
ULONG            releases;
ULONG            suspensions;
ULONG            timeouts;

/* Retrieve performance information on the previously created byte
   pool.  */
status =  tx_byte_pool_performance_info_get(&my_pool,
                &fragments_searched,
                &merges, &splits,
                &allocates, &releases,
                      &suspensions,&timeouts);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_create
- tx_byte_pool_delete
- tx_byte_pool_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize
- tx_byte_release

tx_byte_pool_performance_system_info_get

获取字节池系统性能信息

原型

```
UINT  tx_byte_pool_performance_system_info_get(ULONG *allocates,
        ULONG *releases, ULONG *fragments_searched, ULONG *merges,
        ULONG *splits, ULONG *suspensions, ULONG *timeouts);;
```

说明

此服务检索系统中所有内存字节池的相关性能信息。

> **IMPORTANT**
> 必须使用已定义的 TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- allocates：指向对此池执行的分配请求数的指针。
- releases：指向对此池执行的释放请求数的指针。
- fragments_searched：指向对所有字节池执行分配请求期间搜索到的内部内存片段总数的指针。
- merges：指向对所有字节池执行分配请求期间合并的内部内存块总数的指针。
- splits：指向对所有字节池执行分配请求期间创建的内部内存块拆分（片段）总数的指针。
- suspensions：指向所有字节池的线程分配挂起总次数的指针。
- timeouts：指向所有字节池的分配挂起超时总次数的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取字节池性能信息。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
ULONG         fragments_searched;
ULONG         merges;
ULONG         splits;
ULONG         allocates;
ULONG         releases;
ULONG         suspensions;
ULONG         timeouts;

/* Retrieve performance information on all byte pools in the
   system. */
status =
tx_byte_pool_performance_system_info_get(&fragments_searched,
                &merges, &splits, &allocates, &releases,
                &suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_create
- tx_byte_pool_delete
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_prioritize
- tx_byte_release

tx_byte_pool_prioritize

设置字节池挂起列表的优先级

原型

```
UINT tx_byte_pool_prioritize(TX_BYTE_POOL *pool_ptr);
```

说明

此服务将此池中内存已挂起的最高优先级线程放在挂起列表前面。所有其他线程保持它们挂起时的相同 FIFO 顺序。

参数

- pool_ptr：指向内存池控制块的指针。

返回值

- TX_SUCCESS：(0x00) 成功为内存池设置优先级。
- TX_POOL_ERROR：(0x02) 内存池指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_BYTE_POOL my_pool;
UINT         status;

/* Ensure that the highest priority thread will receive
   the next free memory from this pool. */
status = tx_byte_pool_prioritize(&my_pool);

/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_byte_release call will wake up this thread,
   if there is enough memory to satisfy its request. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_create
- tx_byte_pool_delete
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get

- tx_byte_release

tx_byte_release

将字节释放回内存池

原型

```
UINT tx_byte_release(VOID *memory_ptr);
```

说明

此服务将以前分配的内存区域释放回其关联的池。如果有一个或多个挂起的线程正在等待此池中的内存，则会为每个挂起的线程分配内存，并继续执行直到内存耗尽或不再有任何挂起的线程为止。向挂起的线程分配内存这一过程始终从挂起的第一个线程开始。

> **IMPORTANT**
> 应用程序必须防止在释放内存区域后使用该区域。

参数

- memory_ptr：指向之前分配的内存区域的指针。

返回值

- TX_SUCCESS：(0x00) 成功释放内存。
- TX_PTR_ERROR：(0x03) 内存区域指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

初始化和线程

可以抢占

是

示例

```
unsigned char    *memory_ptr;
UINT             status;

/* Release a memory back to my_pool. Assume that the memory
   area was previously allocated from my_pool. */
status =  tx_byte_release((VOID *) memory_ptr);

/* If status equals TX_SUCCESS, the memory pointed to by
   memory_ptr has been returned to the pool. */
```

另请参阅

- tx_byte_allocate
- tx_byte_pool_create
- tx_byte_pool_delete
- tx_byte_pool_info_get
- tx_byte_pool_performance_info_get
- tx_byte_pool_performance_system_info_get
- tx_byte_pool_prioritize

tx_event_flags_create

创建事件标志组

原型

```
UINT tx_event_flags_create(TX_EVENT_FLAGS_GROUP *group_ptr,
                           CHAR *name_ptr);
```

说明

此服务创建包含 32 个事件标志的事件标志组。组中的 32 个事件标志全部初始化为零。每个事件标志由一个位表示。

参数

- group_ptr：指向事件标志组控制块的指针。
- name_ptr：指向事件标志组名称的指针。

返回值

- TX_SUCCESS：(0x00) 成功创建事件组。
- TX_GROUP_ERROR：(0x06) 事件组指针无效。指针为 NULL 或已创建事件组。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

初始化和线程

可以抢占

否

示例

```
TX_EVENT_FLAGS_GROUP my_event_group;
UINT         status;

/* Create an event flags group. */
status = tx_event_flags_create(&my_event_group,
            "my_event_group_name");

/* If status equals TX_SUCCESS, my_event_group is ready
   for get and set services. */
```

另请参阅

- tx_event_flags_delete
- tx_event_flags_get
- tx_event_flags_info_get
- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set
- tx_event_flags_set_notify

tx_event_flags_delete

删除事件标志组

原型

```
UINT tx_event_flags_delete(TX_EVENT_FLAGS_GROUP *group_ptr);
```

说明

此服务删除指定的事件标志组。所有挂起并等待此组中的事件的线程都将恢复，并获得 TX_DELETED 返回状态。

> **IMPORTANT**
>
> 在删除事件标志组之前，应用程序必须确保完成（或禁用）此事件标志组的设置通知回调。此外，应用程序必须阻止将来再使用已删除的事件标志组。

参数

- group_ptr：指向以前创建的事件标志组的指针。

返回值

- TX_SUCCESS：(0x00) 成功删除事件标志组。
- TX_GROUP_ERROR：(0x06) 事件标志组指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程数

可以抢占

是

示例

```
TX_EVENT_FLAGS_GROUP my_event_flags_group;
UINT                 status;

/* Delete event flags group. Assume that the group has
   already been created with a call to
   tx_event_flags_create. */
status = tx_event_flags_delete(&my_event_flags_group);

/* If status equals TX_SUCCESS, the event flags group is
   deleted. */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_get
- tx_event_flags_info_get
- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set
- tx_event_flags_set_notify

tx_event_flags_get

从事件标志组获取事件标志

原型

```
UINT tx_event_flags_get(TX_EVENT_FLAGS_GROUP *group_ptr,
                        ULONG requested_flags, UINT get_option,
                        ULONG *actual_flags_ptr, ULONG wait_option);
```

说明

此服务从指定事件标志组检索事件标志。每个事件标志组都包含 32 个事件标志。每个标志由一个位表示。此服务可以检索由输入参数选择的各种事件标志组合。

参数

- group_ptr：指向以前创建的事件标志组的指针。
- requested_flags：32 位无符号变量，表示请求的事件标志。
- get_option：指定是所请求的所有事件标志均为必需还是任何事件标志为必需。以下是有效的选择：
  - TX_AND：(0x02)
  - TX_AND_CLEAR：(0x03)
  - TX_OR：(0x00)
  - TX_OR_CLEAR：(0x01)

  如果选择 TX_AND 或 TX_AND_CLEAR，则会指定所有事件标志在组中都必须存在。如果选择 TX_OR 或 TX_OR_CLEAR，则会指定任何事件标志都符合要求。如果指定 TX_AND_CLEAR 或 TX_OR_CLEAR，则会清除满足请求的事件标志（设置为零）。
- actual_flags_ptr：指向在其中放置检索到的事件标志的目标的指针。请注意，实际获得的标志可能包含未请求的标志。
- wait_option：定义未设置所选事件标志时服务的行为方式。等待选项定义如下：
  - TX_NO_WAIT：(0x00000000)
  - TX_WAIT_FOREVER：(0xFFFFFFFF)
  - 超时值：（0x00000001 到 0xFFFFFFFE）

  如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从非线程（例如初始化、计时器或 ISR）调用服务，则这是唯一有效的选项。

  选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到事件标志可用为止。

  如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待事件标志时调用线程保持挂起的最大计时器时钟周期数。

返回值

- TX_SUCCESS：(0x00) 成功获取事件标志。
- TX_DELETED：(0x01) 线程挂起时删除了事件标志组。
- TX_NO_EVENTS：(0x07) 服务无法在指定的等待时间内获取指定的事件。
- TX_WAIT_ABORTED：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- TX_GROUP_ERROR：(0x06) 事件标志组指针无效。
- TX_PTR_ERROR：(0x03) 指向实际事件标志的指针无效。
- TX_WAIT_ERROR：(0x04) 从非线程调用时指定了 TX_NO_WAIT 以外的等待选项。
- TX_OPTION_ERROR：(0x08) 指定的 get_option 无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_EVENT_FLAGS_GROUP my_event_flags_group;
ULONG         actual_events;
UINT          status;

/* Request that event flags 0, 4, and 8 are all set. Also,
   if they are set they should be cleared. If the event
   flags are not set, this service suspends for a maximum of
   20 timer-ticks. */
status = tx_event_flags_get(&my_event_flags_group, 0x111,
                      TX_AND_CLEAR, &actual_events, 20);

/* If status equals TX_SUCCESS, actual_events contains the
   actual events obtained. */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_info_get
- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set
- tx_event_flags_set_notify

tx_event_flags_info_get

检索有关事件标志组的信息

原型

```
UINT tx_event_flags_info_get(TX_EVENT_FLAGS_GROUP *group_ptr,
                         CHAR **name, ULONG *current_flags,
                         TX_THREAD **first_suspended,
                         ULONG *suspended_count,
                         TX_EVENT_FLAGS_GROUP **next_group);
```

说明

此服务检索所指定事件标志组的相关信息。

参数

- group_ptr：指向事件标志组控制块的指针。
- name：指向事件标志组名称的指针。
- current_flags：指向事件标志组中当前设置的标志的指针。
- first_suspended：指向此事件标志组的挂起列表上第一个线程的指针。
- suspended_count：指向此事件标志组中当前挂起的线程数的指针。
- next_group：指向下一个已创建的事件标志组的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功检索事件组信息。

- TX_GROUP_ERROR：(0x06) 事件组指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_EVENT_FLAGS_GROUPmy_event_group;
CHAR          *name;
ULONG         current_flags;
TX_THREAD     *first_suspended;
ULONG         suspended_count;
TX_EVENT_FLAGS_GROUP*next_group;
UINT          status;

/* Retrieve information about the previously created
   event flags group "my_event_group." */
status =  tx_event_flags_info_get(&my_event_group, &name,
             &current_flags,
             &first_suspended, &suspended_count,
             &next_group);

/* If status equals TX_SUCCESS, the information requested is
   valid. */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_get
- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set
- tx_event_flags_set_notify

tx_event_flags_performance info_get

获取事件标志组性能信息

原型

```
UINT tx_event_flags_performance_info_get(TX_EVENT_FLAGS_GROUP
                        *group_ptr, ULONG *sets, ULONG *gets,
                        ULONG *suspensions, ULONG *timeouts);
```

说明

此服务检索所指定事件标志组的相关性能信息。

> **IMPORTANT**
> 必须使用已定义的 TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- group_ptr：指向以前创建的事件标志组的指针。
- sets：指向对此组执行的事件标志设置请求数的指针。
- gets：指向对此组执行的事件标志获取请求数的指针。
- suspensions：指向此组的线程事件标志获取挂起数的指针。
- timeouts：指向此组的事件标志获取挂起超时次数的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取事件标志组性能信息。
- TX_PTR_ERROR：(0x03) 事件标志组指针无效。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_EVENT_FLAGS_GROUPmy_event_flag_group;
ULONG           sets;
ULONG           gets;
ULONG           suspensions;
ULONG           timeouts;

/* Retrieve performance information on the previously created event
   flag group. */
status =  tx_event_flags_performance_info_get(&my_event_flag_group,
   &sets, &gets, &suspensions,
   &timeouts);

/* If status is TX_SUCCESS the performance information was successfully
   retrieved. */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_get
- tx_event_flags_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set
- tx_event_flags_set_notify

tx_event_flags_performance_system_info_get

检索系统性能信息

原型

```
UINT  tx_event_flags_performance_system_info_get(ULONG *sets,
        ULONG *gets,ULONG *suspensions, ULONG *timeouts);
```

说明

此服务检索系统中所有事件标志组的相关性能信息。

> **IMPORTANT**
> 必须使用已定义的 TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- sets：指向对所有组执行的事件标志设置请求总数的指针。
- gets：指向对所有组执行的事件标志获取请求总数的指针。
- suspensions：指向所有组的线程事件标志获取挂起总数的指针。
- timeouts：指向所有组的事件标志获取挂起超时总次数的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取事件标志系统性能信息。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
ULONG         sets;
ULONG         gets;
ULONG         suspensions;
ULONG         timeouts;

/* Retrieve performance information on all previously created event
   flag groups. */
status = tx_event_flags_performance_system_info_get(&sets, &gets,
                &suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_get
- tx_event_flags_info_get
- tx_event_flags_performance_info_get
- tx_event_flags_set
- tx_event_flags_set_notify

tx_event_flags_set

在事件标志组中设置事件标志

原型

```
UINT tx_event_flags_set(TX_EVENT_FLAGS_GROUP *group_ptr,
                        ULONG  flags_to_set,UINT set_option);
```

说明

此服务根据指定的 set_option 设置或清除事件标志组中的事件标志。所有现已满足其事件标志请求的挂起线程都将恢复。

参数

- group_ptr：指向以前创建的事件标志组控制块的指针。
- flags_to_set：根据所选的 set_option，指定要设置或清除的事件标志。
- set_option：指定是对所指定的事件标志与该组的当前事件标志进行“AND”运算还是“OR”运算。以下是有效的选择：
  - TX_AND：(0x02)
  - TX_OR：(0x00) 如果选择 TX_AND，则会指定对所指定的事件标志与该组的当前事件标志进行“AND”运算。此选项通常用于清除组中的事件标志。否则，如果指定了 TX_OR，则对所指定的事件标志与该组的当前事件标志进行“OR”运算。

返回值

- TX_SUCCESS：(0x00) 成功设置事件标志。
- TX_GROUP_ERROR：(0x06) 指向事件标志组的指针无效。
- TX_OPTION_ERROR：(0x08) 指定的 set_option 无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_EVENT_FLAGS_GROUPmy_event_flags_group;
UINT         status;

/* Set event flags 0, 4, and 8. */
status =  tx_event_flags_set(&my_event_flags_group,
                             0x111, TX_OR);

/* If status equals TX_SUCCESS, the event flags have been
   set and any suspended thread whose request was satisfied
   has been resumed. */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_get
- tx_event_flags_info_get
- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set_notify

tx_event_flags_set_notify

设置事件标志时通知应用程序

原型

```
UINT tx_event_flags_set_notify(TX_EVENT_FLAGS_GROUP *group_ptr,
       VOID (*events_set_notify)(TX_EVENT_FLAGS_GROUP *));
```

说明

此服务注册一个通知回调函数，只要在指定的事件标志组中设置了一个或多个事件标志，就会调用该函数。通知回调的处理由应用程序定义。

> **NOTE**
> 不允许应用程序的事件标志设置通知回调调用任何带有挂起选项的 ThreadX SMP API。

参数

- group_ptr：指向以前创建的事件标志组的指针。
- events_set_notify：指向应用程序的事件标志设置通知函数的指针。如果此值为 TX_NULL，则禁用通知。

返回值

- TX_SUCCESS：(0x00) 成功注册事件标志设置通知。
- TX_GROUP_ERROR：(0x06) 事件标志组指针无效。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时禁用了通知功能。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_EVENT_FLAGS_GROUPmy_group;

/* Register the "my_event_flags_set_notify" function for monitoring
   event flags set in the event flags group "my_group." */
status =  tx_event_flags_set_notify(&my_group,
                my_event_flags_set_notify);

/* If status is TX_SUCCESS the event flags set notification function
   was successfully registered. */

void my_event_flags_set_notify(TX_EVENT_FLAGS_GROUP *group_ptr)
   /* One or more event flags was set in this group! */
```

另请参阅

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_get
- tx_event_flags_info_get
- tx_event_flags_performance_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set

tx_interrupt_control

启用和禁用中断

原型

```
UINT tx_interrupt_control(UINT new_posture);
```

说明

此服务启用或禁用由输入参数 new_posture 指定的中断。

> **IMPORTANT**
> 如果从应用程序线程调用此服务，中断状态将保留在该线程的上下文中。例如，如果线程调用此例程来禁用中断，然后挂起，则当该线程恢复后，将重新禁用中断。

> **WARNING**
> 在初始化期间，不应使用此服务来启用中断！ 这样做可能会导致不可预知的结果。

参数

- new_posture：此参数指定是禁用还是启用中断。合法值包括 TX_INT_DISABLE 和 TX_INT_ENABLE。这些参数的实际值特定于端口。此外，某些处理体系结构可能还支持其他中断禁用状态。有关更多详细信息，请参阅分发磁盘上提供的 readme_threadx.txt 文件。

返回值

- previous posture：此服务将以前的中断状态返回给调用方。这允许服务用户在中断被禁用后恢复先前的状态。

获准方式

线程、计时器和 ISR

可以抢占

否

示例

```
UINT my_old_posture;

/* Lockout interrupts */
my_old_posture =  tx_interrupt_control(TX_INT_DISABLE);

/* Perform critical operations that need interrupts
   locked-out.... */

/* Restore previous interrupt lockout posture. */
tx_interrupt_control(my_old_posture);
```

另请参阅

无

tx_mutex_create

创建互斥锁

原型

```
UINT tx_mutex_create(TX_MUTEX *mutex_ptr,
                     CHAR *name_ptr, UINT priority_inherit);
```

说明

此服务为线程间互斥创建一个互斥锁，用于资源保护。

参数

- mutex_ptr：指向互斥锁控制块的指针。
- name_ptr：指向互斥锁名称的指针。
- priority_inherit：指定此互斥锁是否支持优先级继承。如果此值为 TX_INHERIT，则支持优先级继承。但是，如果指定了 TX_NO_INHERIT，则此互斥锁不支持优先级继承。

返回值

- TX_SUCCESS：(0x00) 成功创建互斥锁。
- TX_MUTEX_ERROR：(0x1C) 互斥锁指针无效。指针为 NULL 或已创建互斥锁。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。
- TX_INHERIT_ERROR：(0x1F) 优先级继承参数无效。

获准方式

初始化和线程

可以抢占

否

示例

```
TX_MUTEX     my_mutex;
UINT         status;

/* Create a mutex to provide protection over a
   common resource. */
status =  tx_mutex_create(&my_mutex,"my_mutex_name",
                          TX_NO_INHERIT);

/* If status equals TX_SUCCESS, my_mutex is ready for
   use. */
```

另请参阅

- tx_mutex_delete
- tx_mutex_get
- tx_mutex_info_get
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_prioritize
- tx_mutex_put

tx_mutex_delete

删除互斥锁

原型

```
UINT tx_mutex_delete(TX_MUTEX *mutex_ptr);
```

说明

此服务删除指定的互斥锁。所有挂起并等待互斥锁的线程都将恢复，并获得 TX_DELETED 返回状态。

> **IMPORTANT**
> 应用程序负责阻止使用已删除的互斥锁。

参数

- mutex_ptr：指向之前创建的互斥锁的指针。

返回值

- TX_SUCCESS：(0x00) 成功删除互斥锁。
- TX_MUTEX_ERROR：(0x1C) 互斥锁指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程数

可以抢占

是

示例

```
TX_MUTEX     my_mutex;
UINT         status;

/* Delete a mutex. Assume that the mutex
   has already been created. */
status =  tx_mutex_delete(&my_mutex);

/* If status equals TX_SUCCESS, the mutex is
   deleted. */
```

另请参阅

- tx_mutex_create
- tx_mutex_get
- tx_mutex_info_get
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_prioritize
- tx_mutex_put

tx_mutex_get

获取互斥锁的所有权

原型

```
UINT tx_mutex_get(TX_MUTEX *mutex_ptr, ULONG wait_option);
```

说明

此服务尝试获取指定互斥锁的独占所有权。如果调用线程已拥有互斥锁，则内部计数器会递增，并返回成功状态。

如果互斥锁由另一个线程拥有，并且该线程具有更高的优先级，并且在创建互斥锁时指定了优先级继承，则低优先级线程的优先级将临时提升到调用线程的优先级。

> **IMPORTANT**
>
> 对于拥有互斥锁并指定了优先级继承的低优先级线程，在拥有互斥锁期间，其优先级绝不能由外部线程修改。

参数

- mutex_ptr：指向之前创建的互斥锁的指针。
- wait_option：定义该互斥锁已由另一个线程拥有时此服务的行为方式。等待选项定义如下：
  - TX_NO_WAIT：(0x00000000)
  - TX_WAIT_FOREVER：(0xFFFFFFFF)
  - 超时值：(0x00000001 到 0xFFFFFFFE)

  如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从初始化调用服务，则这是唯一有效的选项。

  选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到互斥锁可用为止。

  如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待互斥锁时调用线程保持挂起的最大计时器时钟周期数。

返回值

- TX_SUCCESS：(0x00) 成功执行互斥锁获取操作。
- TX_DELETED：(0x01) 线程挂起时删除了互斥锁。
- TX_NOT_AVAILABLE：(0x1D) 服务无法在指定的等待时间内获得互斥锁的所有权。
- TX_WAIT_ABORTED：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- TX_MUTEX_ERROR：(0x1C) 互斥锁指针无效。
- TX_WAIT_ERROR：(0x04) 从非线程调用时指定了 TX_NO_WAIT 以外的等待选项。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

初始化、线程和计时器

可以抢占

是

示例

```
TX_MUTEX     my_mutex;
UINT         status;

/* Obtain exclusive ownership of the mutex "my_mutex".
   If the mutex "my_mutex" is not available, suspend until it
   becomes available. */
status =  tx_mutex_get(&my_mutex, TX_WAIT_FOREVER);
```

另请参阅

- tx_mutex_create
- tx_mutex_delete
- tx_mutex_info_get
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_prioritize
- tx_mutex_put

tx_mutex_info_get

检索有关互斥锁的信息

原型

```
UINT tx_mutex_info_get(TX_MUTEX *mutex_ptr, CHAR **name,
                       ULONG *count, TX_THREAD **owner,
                       TX_THREAD **first_suspended,
                       ULONG *suspended_count, TX_MUTEX **next_mutex);
```

说明

此服务从指定的互斥锁检索信息。

参数

- mutex_ptr：指向互斥锁控制块的指针。
- name：指向互斥锁名称的指针。
- count：指向互斥锁所有权计数的指针。
- owner：指向所属线程的指针。
- first_suspended：指向此互斥锁的挂起列表上第一个线程的指针。
- suspended_count：指向此互斥锁中当前挂起的线程数的指针。
- next_mutex：指向下一个已创建的互斥锁的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功检索互斥锁信息。
- TX_MUTEX_ERROR：(0x1C) 互斥锁指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_MUTEX     my_mutex;
CHAR         *name;
ULONG        count;
TX_THREAD    *owner;
TX_THREAD    *first_suspended;
ULONG        suspended_count;
TX_MUTEX     *next_mutex;
UINT         status;

/* Retrieve information about the previously created
   mutex "my_mutex." */
status =  tx_mutex_info_get(&my_mutex, &name,
                            &count, &owner,
                            &first_suspended, &suspended_count,
                            &next_mutex);

/* If status equals TX_SUCCESS, the information requested is
   valid. */
```

另请参阅

- tx_mutex_create
- tx_mutex_delete
- tx_mutex_get
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_prioritize
- tx_mutex_put

tx_mutex_performance_info_get

获取互斥锁性能信息

原型

```
UINT tx_mutex_performance_info_get(TX_MUTEX *mutex_ptr, ULONG *puts,
       ULONG *gets, ULONG *suspensions, ULONG *timeouts,
       ULONG *inversions, ULONG *inheritances);
```

说明

此服务检索所指定互斥锁的相关性能信息。

> **IMPORTANT**
> 必须使用已定义的 TX_MUTEX_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- mutex_ptr：指向之前创建的互斥锁的指针。
- puts：指向对此互斥锁执行的放置请求数的指针。
- gets：指向对此互斥锁执行的获取请求数的指针。
- suspensions：指向此互斥锁的线程互斥锁获取挂起数的指针。
- timeouts：指向此互斥锁的互斥锁获取挂起超时次数的指针。
- inversions：指向此互斥锁的线程优先级倒置数的指针。
- inheritances：指向此互斥锁的线程优先级继承操作数的指针。

> **IMPORTANT**
>
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取互斥锁性能信息。
- TX_PTR_ERROR：(0x03) 互斥锁指针无效。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_MUTEX     my_mutex;
ULONG        puts;
ULONG        gets;
ULONG        suspensions;
ULONG        timeouts;
ULONG        inversions;
ULONG        inheritances;

/* Retrieve performance information on the previously created
   mutex. */
status =  tx_mutex_performance_info_get(&my_mutex_ptr, &puts, &gets,
                &suspensions, &timeouts, &inversions,
                &inheritances);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_mutex_create
- tx_mutex_delete
- tx_mutex_get
- tx_mutex_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_prioritize
- tx_mutex_put

tx_mutex_performance_system_info_get

获取互斥锁系统性能信息

原型

```
UINT  tx_mutex_performance_system_info_get(ULONG *puts, ULONG *gets,
        ULONG *suspensions, ULONG *timeouts,
        ULONG *inversions, ULONG *inheritances);
```

说明

此服务检索系统中所有互斥锁的相关性能信息。

IMPORTANT

必须使用已定义的 TX_MUTEX_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- puts：指向对所有互斥锁执行的放置请求总数的指针。
- gets：指向对所有互斥锁执行的获取请求总数的指针。
- suspensions：指向所有互斥锁的线程互斥锁获取挂起总数的指针。
- timeouts：指向所有互斥锁的互斥锁获取挂起超时总次数的指针。
- inversions：指向所有互斥锁的线程优先级倒置总数的指针。
- inheritances：指向所有互斥锁的线程优先级继承操作总数的指针。

IMPORTANT

为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取互斥锁系统性能信息。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
ULONG         puts;
ULONG         gets;
ULONG         suspensions;
ULONG         timeouts;
ULONG         inversions;
ULONG         inheritances;

/* Retrieve performance information on all previously created
   mutexes. */
status = tx_mutex_performance_system_info_get(&puts, &gets,
                &suspensions, &timeouts,
                &inversions, &inheritances);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_mutex_create
- tx_mutex_delete
- tx_mutex_get
- tx_mutex_info_get
- tx_mutex_performance_info_get
- tx_mutex_prioritize
- tx_mutex_put

tx_mutex_prioritize

设置互斥锁挂起列表的优先级

原型

```
UINT tx_mutex_prioritize(TX_MUTEX *mutex_ptr);
```

说明

此服务将为拥有互斥锁而挂起的最高优先级线程放在挂起列表前面。所有其他线程保持它们挂起时的相同 FIFO 顺序。

参数

- mutex_ptr：指向之前创建的互斥锁的指针。

返回值

- TX_SUCCESS：(0x00) 成功为互斥锁设置优先级。
- TX_MUTEX_ERROR：(0x1C) 互斥锁指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_MUTEX     my_mutex;
UINT         status;

/* Ensure that the highest priority thread will receive
   ownership of the mutex when it becomes available. */
status = tx_mutex_prioritize(&my_mutex);

/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_mutex_put call that releases ownership of the
   mutex will give ownership to this thread and wake it
   up. */
```

另请参阅

- tx_mutex_create
- tx_mutex_delete
- tx_mutex_get
- tx_mutex_info_get
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_put

tx_mutex_put

释放互斥锁的所有权

原型

```
UINT tx_mutex_put(TX_MUTEX *mutex_ptr);
```

说明

此服务递减指定互斥锁的所有权计数。如果所有权计数为零，则互斥锁可用。

> **IMPORTANT**
> 如果在创建互斥锁期间选择了优先级继承，则释放线程的优先级将恢复为最初获得互斥锁所有权时的优先级。在拥有互斥锁期间对释放线程所做的任何其他优先级更改都可以撤消。

参数

- mutex_ptr：指向之前创建的互斥锁的指针。

返回值

- TX_SUCCESS：(0x00) 成功释放互斥锁。
- TX_NOT_OWNED：(0x1E) 互斥锁不归调用方所有。
- TX_MUTEX_ERROR：(0x1C) 互斥锁指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

初始化、线程和计时器

可以抢占

是

示例

```
TX_MUTEX         my_mutex;
UINT             status;
/* Release ownership of "my_mutex." */
status =  tx_mutex_put(&my_mutex);

/* If status equals TX_SUCCESS, the mutex ownership
   count has been decremented and if zero, released. */
```

另请参阅

- tx_mutex_create
- tx_mutex_delete
- tx_mutex_get
- tx_mutex_info_get
- tx_mutex_performance_info_get
- tx_mutex_performance_system_info_get
- tx_mutex_prioritize

tx_queue_create

创建消息队列

原型

```
UINT tx_queue_create(TX_QUEUE *queue_ptr, CHAR *name_ptr,
                         UINT message_size,
                         VOID *queue_start, ULONG queue_size);
```

说明

此服务创建通常用于线程间通信的消息队列。消息总数是根据指定的消息大小和队列中的字节总数来计算的。

> **IMPORTANT**
> 如果队列内存区域中指定的总字节数不能被指定的消息大小整除，则不会使用内存区域中剩余的字节。

参数

- queue_ptr：指向消息队列控制块的指针。
- name_ptr：指向消息队列名称的指针。
- message_size：指定队列中每条消息的大小。消息大小从 1 个 32 位字到 16 个 32 位字不等。有效的消息大小选项是介于 1 到 16（含）之间的数值。
- queue_start：消息队列的起始地址。起始地址必须与 ULONG 数据类型的大小一致。
- queue_size：可用于消息队列的总字节数。

返回值

- TX_SUCCESS：(0x00) 成功创建消息队列。
- TX_QUEUE_ERROR：(0x09) 消息队列指针无效。指针为 NULL 或已创建队列。
- TX_PTR_ERROR：(0x03) 消息队列的起始地址无效。
- TX_SIZE_ERROR：(0x05) 消息队列大小无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

初始化和线程

可以抢占

否

示例

```
TX_QUEUE     my_queue;
UINT         status;

/* Create a message queue whose total size is 2000 bytes
   starting at address 0x300000. Each message in this
   queue is defined to be 4 32-bit words long. */
status = tx_queue_create(&my_queue, "my_queue_name",
            4, (VOID *) 0x300000, 2000);

/* If status equals TX_SUCCESS, my_queue contains room
   for storing 125 messages (2000 bytes/ 16 bytes per
   message). */
```

另请参阅

- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_delete

删除消息队列

原型

```
UINT tx_queue_delete(TX_QUEUE *queue_ptr);
```

说明

此服务删除指定的消息队列。所有挂起并等待此队列中的消息的线程都将恢复，并获得 TX_DELETED 返回状态。

> **IMPORTANT**
> 在删除队列之前，应用程序必须确保完成（或禁用）此队列的任何发送通知回调。此外，应用程序必须阻止将来再使用已删除的队列。

应用程序还负责管理与队列关联的内存区域，该内存区域在此服务完成后可用。

参数

- queue_ptr：指向以前创建的消息队列的指针。

返回值

- TX_SUCCESS：(0x00) 成功删除消息队列。
- TX_QUEUE_ERROR：(0x09) 消息队列指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程数

可以抢占

是

示例

```
TX_QUEUE     my_queue;
UINT         status;

/* Delete entire message queue. Assume that the queue
   has already been created with a call to
   tx_queue_create. */
status = tx_queue_delete(&my_queue);

/* If status equals TX_SUCCESS, the message queue is
   deleted. */
```

另请参阅

- tx_queue_create
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive

- tx_queue_send
- tx_queue_send_notify

tx_queue_flush

清空消息队列中的消息

原型

```
UINT tx_queue_flush(TX_QUEUE *queue_ptr);
```

说明

此服务删除存储在指定消息队列中的所有消息。如果队列已满，则丢弃所有挂起线程的消息。然后，每个挂起的线程都将恢复，并且返回状态会指示消息发送成功。如果队列为空，则此服务不执行任何操作。

参数

- queue_ptr：指向以前创建的消息队列的指针。

返回值

- TX_SUCCESS：(0x00) 成功刷新消息队列。
- TX_QUEUE_ERROR：(0x09) 消息队列指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_QUEUE     my_queue;
UINT         status;

/* Flush out all pending messages in the specified message
   queue. Assume that the queue has already been created
   with a call to tx_queue_create. */
status =  tx_queue_flush(&my_queue);

/* If status equals TX_SUCCESS, the message queue is
    empty. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_front_send

将消息发送到队列的前面

原型

```
UINT tx_queue_front_send(TX_QUEUE *queue_ptr,
                         VOID *source_ptr, ULONG wait_option);
```

说明

此服务将消息发送到指定消息队列的前端位置。将消息从源指针指定的内存区域复制到队列的前面。

参数

- queue_ptr：指向消息队列控制块的指针。
- source_ptr：指向消息的指针。
- wait_option：定义在消息队列已满时服务的行为方式。等待选项定义如下：
  - TX_NO_WAIT：(0x00000000)
  - TX_WAIT_FOREVER：(0xFFFFFFFF)
  - 超时值：(0x00000001 到 0xFFFFFFFE)

  如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从非线程（例如初始化、计时器或 ISR）调用服务，则这是唯一有效的选项。

  选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到队列中有空间为止。

  如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待队列中的空间时调用线程保持挂起的最大计时器时钟周期数。

返回值

- TX_SUCCESS：(0x00) 成功发送消息。
- TX_DELETED：(0x01) 线程挂起时删除了消息队列。
- TX_QUEUE_FULL：(0x0B) 服务无法发送消息，因为在指定的等待时间内队列已满。
- TX_WAIT_ABORTED：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- TX_QUEUE_ERROR：(0x09) 消息队列指针无效。
- TX_PTR_ERROR：(0x03) 消息的源指针无效。
- TX_WAIT_ERROR：(0x04) 从非线程调用时指定了 TX_NO_WAIT 以外的等待选项。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_QUEUE     my_queue;
UINT         status;
ULONG        my_message[4];

/* Send a message to the front of "my_queue." Return
   immediately, regardless of success. This wait
   option is used for calls from initialization, timers,
   and ISRs. */
status = tx_queue_front_send(&my_queue, my_message,
            TX_NO_WAIT);

/* If status equals TX_SUCCESS, the message is at the front
   of the specified queue. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_info_get

检索有关队列的信息

原型

```
UINT tx_queue_info_get(TX_QUEUE *queue_ptr, CHAR **name,
        ULONG *enqueued, ULONG *available_storage
        TX_THREAD **first_suspended, ULONG *suspended_count,
        TX_QUEUE **next_queue);
```

说明

此服务检索所指定消息队列的相关信息。

参数

- queue_ptr：指向以前创建的消息队列的指针。
- name：指向队列名称的指针。
- enqueued：指向队列中的当前消息数的指针。
- available_storage：指向队列当前可容纳的消息数的指针。
- first_suspended：指向此队列的挂起列表上第一个线程的指针。
- suspended_count：指向此队列中当前挂起的线程数的指针。
- next_queue：指向下一个已创建的队列的指针。

> **IMPORTANT**
>
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取队列信息。
- TX_QUEUE_ERROR：(0x09) 消息队列指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_QUEUE     my_queue;
CHAR         *name;
ULONG        enqueued;
ULONG        available_storage;
TX_THREAD    *first_suspended;
ULONG        suspended_count;
TX_QUEUE     *next_queue;
UINT         status;

/* Retrieve information about the previously created
   message queue "my_queue." */
status = tx_queue_info_get(&my_queue, &name,
            &enqueued, &available_storage,
            &first_suspended, &suspended_count,
            &next_queue);

/* If status equals TX_SUCCESS, the information requested is
   valid. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_performance_info_get

获取队列性能信息

原型

```
UINT  tx_queue_performance_info_get(TX_QUEUE *queue_ptr,
        ULONG *messages_sent, ULONG *messages_received,
        ULONG *empty_suspensions, ULONG *full_suspensions,
        ULONG *full_errors, ULONG *timeouts);
```

说明

此服务检索所指定消息队列的相关性能信息。

IMPORTANT

必须使用已定义的 TX_QUEUE_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- queue_ptr：指向之前创建的队列的指针。
- messages_sent：指向对此队列执行的发送请求数的指针。
- messages_received：指向对此队列执行的接收请求数的指针。
- empty_suspensions：指向此队列的队列为空挂起数的指针。
- full_suspensions：指向此队列的队列已满挂起数的指针。
- full_errors：指向此队列的队列已满错误数的指针。
- timeouts：指向此队列的线程挂起超时次数的指针。

IMPORTANT

为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取队列性能信息。
- TX_PTR_ERROR：(0x03) 队列指针无效。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_QUEUE     my_queue;
ULONG        messages_sent;
ULONG        messages_received;
ULONG        empty_suspensions;
ULONG        full_suspensions;
ULONG        full_errors;
ULONG        timeouts;

/* Retrieve performance information on the previously created
   queue. */
status = tx_queue_performance_info_get(&my_queue, &messages_sent,
            &messages_received, &empty_suspensions,
            &full_suspensions, &full_errors, &timeouts);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize

- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_performance_system_info_get

获取队列系统性能信息

原型

```
UINT  tx_queue_performance_system_info_get(ULONG *messages_sent,
        ULONG *messages_received, ULONG *empty_suspensions,
        ULONG *full_suspensions, ULONG *full_errors,
        ULONG *timeouts);
```

说明

此服务检索系统中所有队列的相关性能信息。

> **IMPORTANT**
> 必须使用已定义的 TX_QUEUE_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- messages_sent：指向对所有队列执行的发送请求总数的指针。
- messages_received：指向对所有队列执行的接收请求总数的指针。
- empty_suspensions：指向所有队列的队列为空挂起总数的指针。
- full_suspensions：指向所有队列的队列已满挂起总数的指针。
- full_errors：指向所有队列的队列已满错误总数的指针。
- timeouts：指向所有队列的线程挂起超时总次数的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取队列系统性能信息。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
ULONG         messages_sent;
ULONG         messages_received;
ULONG         empty_suspensions;
ULONG         full_suspensions;
ULONG         full_errors;
ULONG         timeouts;

/* Retrieve performance information on all previously created
   queues. */
status = tx_queue_performance_system_info_get(&messages_sent,
            &messages_received, &empty_suspensions,
            &full_suspensions, &full_errors, &timeouts);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_prioritize

设置队列挂起列表的优先级

原型

```
UINT tx_queue_prioritize(TX_QUEUE *queue_ptr);
```

说明

此服务将此队列中消息已挂起的最高优先级线程(或者将消息)放在挂起列表前面。所有其他线程保持它们挂起时的相同 FIFO 顺序。

参数

- queue_ptr:指向以前创建的消息队列的指针。

返回值

- TX_SUCCESS:(0x00) 成功为队列设置优先级。
- TX_QUEUE_ERROR:(0x09) 消息队列指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_QUEUE     my_queue;
UINT         status;

/* Ensure that the highest priority thread will receive
   the next message placed on this queue. */
status = tx_queue_prioritize(&my_queue);

/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_queue_send or tx_queue_front_send call made
   to this queue will wake up this thread. */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_receive
- tx_queue_send
- tx_queue_send_notify

tx_queue_receive

从消息队列获取消息

原型

```
UINT tx_queue_receive(TX_QUEUE *queue_ptr,
                          VOID *destination_ptr, ULONG wait_option);
```

说明

此服务从指定的消息队列检索消息。将检索到的消息从队列复制到目标指针指定的内存区域中。然后从队列中删除该消息。

> **WARNING**
> 指定的目标内存区域必须足够大以容纳消息；也就是说，destination_ptr 指向的消息目标必须至少与此队列的消息大小一样大。否则，如果目标不够大，则下一个内存区域会发生内存损坏。

参数

- queue_ptr：指向以前创建的消息队列的指针。
- destination_ptr：要将消息复制到的位置。
- wait_option：定义在消息队列为空时服务的行为方式。等待选项定义如下：
  - TX_NO_WAIT：(0x00000000)
  - TX_WAIT_FOREVER：(0xFFFFFFFF)
  - 超时值：(0x00000001 到 0xFFFFFFFE)

  如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从非线程（例如初始

化、计时器或 ISR）调用服务，则这是唯一有效的选项。

选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到消息可用为止。

如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待消息时调用线程保持挂起的最大计时器时钟周期数。

返回值

- TX_SUCCESS：(0x00) 成功检索消息。
- TX_DELETED：(0x01) 线程挂起时删除了消息队列。
- TX_QUEUE_EMPTY：(0x0A) 服务无法检索消息，因为队列在指定的等待时间内为空。
- TX_WAIT_ABORTED：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- TX_QUEUE_ERROR：(0x09) 消息队列指针无效。
- TX_PTR_ERROR：(0x03) 消息的目标指针无效。
- TX_WAIT_ERROR：(0x04) 从非线程调用时指定了 TX_NO_WAIT 以外的等待选项。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_QUEUE     my_queue;
UINT         status;
ULONG my_message[4];

/* Retrieve a message from "my_queue." If the queue is
   empty, suspend until a message is present. Note that
   this suspension is only possible from application
   threads. */
status =  tx_queue_receive(&my_queue, my_message,
                           TX_WAIT_FOREVER);

/* If status equals TX_SUCCESS, the message is in
   "my_message." */
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_send
- tx_queue_send_notify

tx_queue_send

将消息发送到消息队列

原型

```
UINT tx_queue_send(TX_QUEUE *queue_ptr,
                           VOID *source_ptr, ULONG wait_option);
```

说明

此服务将消息发送到指定的消息队列。将发送的消息从源指针指定的内存区域复制到队列。

参数

- queue_ptr：指向以前创建的消息队列的指针。
- source_ptr：指向消息的指针。
- wait_option：定义在消息队列已满时服务的行为方式。等待选项定义如下：
  - TX_NO_WAIT：(0x00000000)
  - TX_WAIT_FOREVER：(0xFFFFFFFF)
  - 超时值：(0x00000001 到 0xFFFFFFFE)

  如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从非线程（例如初始化、计时器或 ISR）调用服务，则这是唯一有效的选项。

  选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到队列中有空间为止。

  如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待队列中的空间时调用线程保持挂起的最大计时器时钟周期数。

返回值

- TX_SUCCESS：(0x00) 成功发送消息。
- TX_DELETED：(0x01) 线程挂起时删除了消息队列。
- TX_QUEUE_FULL：(0x0B) 服务无法发送消息，因为在指定的等待时间内队列已满。
- TX_WAIT_ABORTED：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- TX_QUEUE_ERROR：(0x09) 消息队列指针无效。
- TX_PTR_ERROR：(0x03) 消息的源指针无效。
- TX_WAIT_ERROR：(0x04) 从非线程调用时指定了 TX_NO_WAIT 以外的等待选项。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_QUEUE my_queue;
UINT status;
ULONG my_message[4];

/* Send a message to "my_queue." Return immediately,
   regardless of success. This wait option is used for
   calls from initialization, timers, and ISRs. */
status =  tx_queue_send(&my_queue, my_message, TX_NO_WAIT);

/* If status equals TX_SUCCESS, the message is in the
   queue. */
```

另请参阅

- tx_queue_create

- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send_notify

tx_queue_send_notify

在将消息发送到队列时通知应用程序

原型

```
UINT  tx_queue_send_notify(TX_QUEUE *queue_ptr,
        VOID (*queue_send_notify)(TX_QUEUE *));
```

说明

此服务注册一个通知回调函数，每当消息发送到指定的队列时，就会调用该函数。通知回调的处理由应用程序定义。

> **NOTE**
> 不允许应用程序的队列发送通知回调调用任何带有挂起选项的 ThreadX SMP API。

参数

- queue_ptr：指向之前创建的队列的指针。
- queue_send_notify：指向应用程序的队列发送通知函数的指针。如果此值为 TX_NULL，则禁用通知。

返回值

- TX_SUCCESS：(0x00) 成功注册队列发送通知。
- TX_QUEUE_ERROR：(0x09) 队列指针无效。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时禁用了通知功能。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_QUEUE         my_queue;

/* Register the "my_queue_send_notify" function for monitoring
   messages sent to the queue "my_queue." */
status = tx_queue_send_notify(&my_queue, my_queue_send_notify);

/* If status is TX_SUCCESS the queue send notification function was
   successfully registered. */
void my_queue_send_notify(TX_QUEUE *queue_ptr)
{
/* A message was just sent to this queue! */
}
```

另请参阅

- tx_queue_create
- tx_queue_delete
- tx_queue_flush
- tx_queue_front_send
- tx_queue_info_get
- tx_queue_performance_info_get
- tx_queue_performance_system_info_get
- tx_queue_prioritize
- tx_queue_receive
- tx_queue_send

tx_semaphore_ceiling_put

将实例放入具有上限的计数信号灯

原型

```
UINT  tx_semaphore_ceiling_put(TX_SEMAPHORE *semaphore_ptr,
        ULONG ceiling);
```

说明

此服务将一个实例放入指定的计数信号灯，这实际上会将计数信号灯增加 1。如果计数信号灯的当前值大于或等于指定的上限，则不会放入该实例，并将返回 TX_CEILING_EXCEEDED 错误。

参数

- semaphore_ptr：指向之前创建的信号灯的指针。
- ceiling：允许的信号灯上限（有效值的范围介于 1 到 0xFFFFFFFF 之间）。

返回值

- TX_SUCCESS：(0x00) 成功指定信号灯上限。
- TX_CEILING_EXCEEDED：(0x21) 放置请求超过上限。
- TX_INVALID_CEILING：(0x22) 为上限提供的值 (0) 无效。
- TX_SEMAPHORE_ERROR：(0x0C) 信号灯指针无效。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_SEMAPHORE     my_semaphore;

/* Increment the counting semaphore "my_semaphore" but make sure
   that it never exceeds 7 as specified in the call. */
status = tx_semaphore_ceiling_put(&my_semaphore, 7);

/* If status is TX_SUCCESS the semaphore count has been
```

另请参阅

- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_get

- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_create

创建计数信号灯

原型

```
UINT tx_semaphore_create(TX_SEMAPHORE *semaphore_ptr,
                         CHAR *name_ptr, ULONG initial_count);
```

说明

此服务创建用于线程间同步的计数信号灯。初始信号灯计数指定为输入参数。

参数

- semaphore_ptr：指向信号灯控制块的指针。
- name_ptr：指向信号灯名称的指针。
- initial_count：指定此信号灯的初始计数。合法值的范围为 0x00000000 至 0xFFFFFFFF。

返回值

- TX_SUCCESS：(0x00) 成功创建信号灯。
- TX_SEMAPHORE_ERROR：(0x0C) 信号灯指针无效。指针为 NULL 或已创建信号灯。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

初始化和线程

可以抢占

否

示例

```
TX_SEMAPHORE my_semaphore;
UINT         status;

/* Create a counting semaphore whose initial value is 1.
   This is typically the technique used to make a binary
   semaphore. Binary semaphores are used to provide
   protection over a common resource. */
status = tx_semaphore_create(&my_semaphore,
            "my_semaphore_name", 1);

/* If status equals TX_SUCCESS, my_semaphore is ready for
   use. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_delete
- tx_semaphore_get

- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_delete

删除计数信号灯

原型

```
UINT tx_semaphore_delete(TX_SEMAPHORE *semaphore_ptr);
```

说明

此服务删除指定的计数信号灯。所有挂起并等待信号灯实例的线程都将恢复，并获得 TX_DELETED 返回状态。

> **IMPORTANT**
> 在删除信号灯之前，应用程序必须确保完成（或禁用）此信号灯的放置通知回调。此外，应用程序必须阻止将来再使用已删除的信号灯。

参数

- semaphore_ptr：指向之前创建的信号灯的指针。

返回值

- TX_SUCCESS：(0x00) 成功删除计数信号灯。
- TX_SEMAPHORE_ERROR：(0x0C) 计数信号灯指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程数

可以抢占

是

示例

```
TX_SEMAPHORE my_semaphore;
UINT         status;

/* Delete counting semaphore. Assume that the counting
   semaphore has already been created. */
status = tx_semaphore_delete(&my_semaphore);

/* If status equals TX_SUCCESS, the counting semaphore is
   deleted. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semaphore_get

- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_get

从计数信号灯获取实例

原型

```
UINT tx_semaphore_get(TX_SEMAPHORE *semaphore_ptr,
                      ULONG wait_option)
```

说明

此服务从指定的计数信号灯检索实例（单个计数）。因此，指定信号灯的计数将减少 1。

参数

- semaphore_ptr：指向之前创建的计数信号灯的指针。
- wait_option：定义在没有任何信号灯实例可用（即信号灯计数为零）时服务的行为方式。等待选项定义如下：
  - TX_NO_WAIT：(0x00000000)
  - TX_WAIT_FOREVER：(0xFFFFFFFF)
  - 超时值：（0x00000001 到 0xFFFFFFFE）

  如果选择 TX_NO_WAIT，则无论此服务是否成功，都会导致立即从此服务返回。如果从非线程（例如初始化、计时器或 ISR）调用服务，则这是唯一有效的选项。

  选择 TX_WAIT_FOREVER 会导致调用线程无限期挂起，直到信号灯实例可用为止。

  如果选择一个数值（1 到 0xFFFFFFFE），则会指定在等待信号灯实例时调用线程保持挂起的最大计时器时钟周期数。

返回值

- TX_SUCCESS：(0x00) 成功检索信号灯实例。
- TX_DELETED：(0x01) 线程挂起时删除了计数信号灯。
- TX_NO_INSTANCE：(0x0D) 服务无法检索计数信号灯的实例（信号灯计数在指定的等待时间内为零）。
- TX_WAIT_ABORTED：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- TX_SEMAPHORE_ERROR：(0x0C) 计数信号灯指针无效。
- TX_WAIT_ERROR：(0x04) 从非线程调用时指定了 TX_NO_WAIT 以外的等待选项。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_SEMAPHORE my_semaphore;
UINT         status;

/* Get a semaphore instance from the semaphore
   "my_semaphore." If the semaphore count is zero,
   suspend until an instance becomes available.
   Note that this suspension is only possible from
   application threads. */
status =  tx_semaphore_get(&my_semaphore, TX_WAIT_FOREVER);

/* If status equals TX_SUCCESS, the thread has obtained
   an instance of the semaphore. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semahore_delete
- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_info_get

检索有关信号灯的信息

原型

```
UINT tx_semaphore_info_get(TX_SEMAPHORE *semaphore_ptr,
                           CHAR **name, ULONG *current_value,
                           TX_THREAD **first_suspended,
                           ULONG *suspended_count,
                           TX_SEMAPHORE **next_semaphore)
```

说明

此服务检索所指定信号灯的相关信息。

参数

- semaphore_ptr：指向信号灯控制块的指针。
- name：指向信号灯名称的指针。
- current_value：指向当前信号灯的计数的指针。
- first_suspended：指向此信号灯的挂起列表上第一个线程的指针。
- suspended_count：指向此信号灯中当前挂起的线程数的指针。
- next_semaphore：指向下一个已创建的信号灯的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功检索信号灯信息。
- TX_SEMAPHORE_ERROR：(0x0C) 信号灯指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_SEMAPHORE my_semaphore;
CHAR         *name;
ULONG        current_value;
TX_THREAD    *first_suspended;
ULONG        suspended_count;
TX_SEMAPHORE *next_semaphore;
UINT         status;

/* Retrieve information about the previously created
   semaphore "my_semaphore." */
status =  tx_semaphore_info_get(&my_semaphore, &name,
                      &current_value,
                      &first_suspended, &suspended_count,
                      &next_semaphore);

/* If status equals TX_SUCCESS, the information requested is
   valid. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_performance_info_get

获取信号灯性能信息

原型

```
UINT  tx_semaphore_performance_info_get(TX_SEMAPHORE *semaphore_ptr,
        ULONG *puts, ULONG *gets,
        ULONG *suspensions, ULONG *timeouts);
```

说明

此服务检索所指定信号灯的相关性能信息。

> **NOTE**
>
> 必须使用已定义的 TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- semaphore_ptr：指向之前创建的信号灯的指针。
- puts：指向对此信号灯执行的放置请求数的指针。
- gets：指向对此信号灯执行的获取请求数的指针。
- suspensions：指向此信号灯的线程挂起数的指针。
- timeouts：指向此信号灯的线程挂起超时次数的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取信号灯性能信息。
- TX_PTR_ERROR：(0x03) 信号灯指针无效。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_SEMAPHORE     my_semaphore;
ULONG            puts;
ULONG            gets;
ULONG            suspensions;
ULONG            timeouts;

/* Retrieve performance information on the previously created
   semaphore. */
status =  tx_semaphore_performance_info_get(&my_semaphore, &puts,
               &gets, &suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_get
- tx_semaphore_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_performance_system_info_get

获取信号灯系统性能信息

原型

```
UINT tx_semaphore_performance_system_info_get(ULONG *puts,
       ULONG *gets, ULONG *suspensions, ULONG *timeouts);
```

说明

此服务检索系统中所有信号灯的相关性能信息。

> **IMPORTANT**
> 必须使用已定义的 TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- puts：指向对所有信号灯执行的放置请求总数的指针。
- gets：指向对所有信号灯执行的获取请求总数的指针。
- suspensions：指向所有信号灯的线程挂起总数的指针。
- timeouts：指向所有信号灯的线程挂起超时总次数的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取信号灯系统性能信息。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
ULONG         puts;
ULONG         gets;
ULONG         suspensions;
ULONG         timeouts;

/* Retrieve performance information on all previously created
   semaphores. */
status = tx_semaphore_performance_system_info_get(&puts, &gets,
               &suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_get
- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_prioritize
- tx_semaphore_put
- tx_semaphore_put_notify

tx_semaphore_prioritize

设置信号灯挂起列表的优先级

原型

```
UINT tx_semaphore_prioritize(TX_SEMAPHORE *semaphore_ptr);
```

说明

此服务将信号灯实例已挂起的最高优先级线程放在挂起列表前面。所有其他线程保持它们挂起时的相同 FIFO 顺序。

参数

- semaphore_ptr：指向之前创建的信号灯的指针。

返回值

- TX_SUCCESS：(0x00) 成功为信号灯设置优先级。
- TX_SEMAPHORE_ERROR：(0x0C) 计数信号灯指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_SEMAPHORE my_semaphore;
UINT         status;

/* Ensure that the highest priority thread will receive
   the next instance of this semaphore. */
status =  tx_semaphore_prioritize(&my_semaphore);

/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_semaphore_put call made to this semaphore will
   wake up this thread. */
```

另请参阅

- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_get
- tx_semaphore_info_get
- tx_semaphore_put

tx_semaphore_put

将实例放入计数信号灯

原型

```
UINT tx_semaphore_put(TX_SEMAPHORE *semaphore_ptr);
```

说明

此服务将一个实例放入指定的计数信号灯，这实际上会将计数信号灯增加 1。

IMPORTANT

如果在信号灯全部为一 (OxFFFFFFFF) 时调用此服务，则新的放置操作将导致信号灯重置为零。

参数

- semaphore_ptr：指向之前创建的计数信号灯控制块的指针。

返回值

- TX_SUCCESS：(0x00) 成功放置信号灯。
- TX_SEMAPHORE_ERROR：(0x0C) 计数信号灯指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_SEMAPHORE     my_semaphore;
UINT             status;

/* Increment the counting semaphore "my_semaphore." */
status =  tx_semaphore_put(&my_semaphore);

/* If status equals TX_SUCCESS, the semaphore count has
   been incremented. Of course, if a thread was waiting,
   it was given the semaphore instance and resumed. */
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_get
- tx_semaphore_put_notify

tx_semaphore_put_notify

放置信号灯时通知应用程序

原型

```
UINT  tx_semaphore_put_notify(TX_SEMAPHORE *semaphore_ptr,
        VOID (*semaphore_put_notify)(TX_SEMAPHORE *));
```

说明

此服务注册一个通知回调函数，每当放置指定的信号灯时，就会调用该函数。通知回调的处理由应用程序定义。

> **NOTE**
>
> 不允许应用程序的信号灯通知回调调用任何带有挂起选项的 ThreadX SMP API。

参数

- semaphore_ptr：指向之前创建的信号灯的指针。
- semaphore_put_notify：指向应用程序的信号灯放置通知函数的指针。如果此值为 TX_NULL，则禁用通知。

返回值

- TX_SUCCESS：(0x00) 成功注册信号灯放置通知。
- TX_SEMAPHORE_ERROR：(0x0C) 信号灯指针无效。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时禁用了通知功能。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_SEMAPHORE     my_semaphore;

/* Register the "my_semaphore_put_notify" function for monitoring
   the put operations on the semaphore "my_semaphore." */
status =  tx_semaphore_put_notify(&my_semaphore,
                my_semaphore_put_notify);

/* If status is TX_SUCCESS the semaphore put notification function
   was successfully registered. */

void my_semaphore_put_notify(TX_SEMAPHORE *semaphore_ptr)
{
   /* The semaphore was just put! */
}
```

另请参阅

- tx_semaphore_ceiling_put
- tx_semaphore_create
- tx_semaphore_delete
- tx_semaphore_get
- tx_semaphore_info_get
- tx_semaphore_performance_info_get
- tx_semaphore_performance_system_info_get
- tx_semaphore_prioritize
- tx_semaphore_put

tx_thread_create

创建应用程序线程

原型

```
UINT tx_thread_create(TX_THREAD *thread_ptr,
                      CHAR *name_ptr, VOID (*entry_function)(ULONG),
                      ULONG entry_input, VOID *stack_start,
                      ULONG stack_size, UINT priority,
                      UINT preempt_threshold, ULONG time_slice,
                      UINT auto_start);
```

说明

此服务创建一个应用程序线程，该线程在指定的任务入口函数处开始执行。堆栈、优先级、抢占阈值和时间片都是由输入参数指定的属性。此外，还指定了线程的初始执行状态。

参数

- thread_ptr：指向线程控制块的指针。
- name_ptr：指向线程名称的指针。
- entry_function：指定线程执行的初始 C 函数。当线程从此入口函数返回时，它将处于完成状态并无限期挂起。
- entry_input：首次执行时传递给线程的入口函数的 32 位值。该输入的用途完全由应用程序决定。
- stack_start：堆栈的内存区域的起始地址。
- stack_size：堆栈内存区域中的字节数。线程的堆栈区域必须足够大以容纳最坏情况下的函数调用嵌套和局部变量使用情况。
- priority：线程的优先级数值。合法值的范围为 0 至 TX_MAX_PRIORITES-1，其中 0 表示最高优先级。
- preempt_threshold：禁用抢占的最高优先级（0 至 TX_MAX_PRIORITIES-1）。仅允许高于此级别的优先级抢占该线程。该值必须小于或等于指定的优先级。如果设置为等于线程优先级的值，将禁用抢占阈值。
- time_slice：在优先级相同的其他就绪线程有机会运行之前，允许该线程运行的计时器时钟周期数。请注意，使用抢占阈值将禁用时间片。合法时间片值的范围为 1 至 0xFFFFFFFF（含）。值为 TX_NO_TIME_SLICE（值为 0）将禁用此线程的时间片。

  > **IMPORTANT**
  > 使用时间片会产生少量系统开销。由于时间片仅适用于多个线程具有相同优先级的情况，因此不应为具有唯一优先级的线程分配时间片。

- auto_start：指定线程是立即启动还是置于挂起状态。合法选项为 TX_AUTO_START (0x01) 和 TX_DONT_START (0x00) 。如果指定了 TX_DONT_START，应用程序随后必须调用 tx_thread_resume 以便线程运行。

返回值

- TX_SUCCESS：(0x00) 成功创建线程。
- TX_THREAD_ERROR：(0x0E) 线程控制指针无效。指针为 NULL 或已创建线程。
- TX_PTR_ERROR：(0x03) 入口点的起始地址无效或堆践区域无效（通常为 NULL）。
- TX_SIZE_ERROR：(0x05) 堆栈区域大小无效。线程必须至少有 TX_MINIMUM_STACK 字节才能执行。
- TX_PRIORITY_ERROR：(0x0F) 线程优先级无效，该值超出范围（0 至 TX_MAX_PRIORITIES-1）。
- TX_THRESH_ERROR：(0x18) 指定的抢占阈值无效。此值必须是小于或等于线程初始优先级的有效优先级。
- TX_START_ERROR：(0x10) 自动启动选择无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

初始化和线程

可以抢占

是

示例

```
TX_THREAD     my_thread;
UINT          status;

/* Create a thread of priority 15 whose entry point is
   "my_thread_entry". This thread's stack area is 1000
   bytes in size, starting at address 0x400000. The
   preemption-threshold is setup to allow preemption of threads
   with priorities ranging from 0 through 14. Time-slicing is
   disabled. This thread is automatically put into a ready
   condition. */
status =  tx_thread_create(&my_thread, "my_thread_name",
                my_thread_entry, 0x1234,
                (VOID *) 0x400000, 1000,
                15, 15, TX_NO_TIME_SLICE,
                TX_AUTO_START);

/* If status equals TX_SUCCESS, my_thread is ready
   for execution! */
...

/* Thread's entry function. When "my_thread" actually
   begins execution, control is transferred to this
   function. */
VOID my_thread_entry (ULONG initial_input)
{

    /* When we get here, the value of initial_input is
    0x1234. See how this was specified during
    creation. */

    /* The real work of the thread, including calls to
    other function should be called from here! */

    /* When this function returns, the corresponding
    thread is placed into a "completed" state. */
}
```

另请参阅

- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend

- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_delete

删除应用程序线程

原型

```
UINT tx_thread_delete(TX_THREAD *thread_ptr);
```

说明

此服务删除指定的应用程序线程。由于指定的线程必须处于已终止或已完成状态，因此不能从尝试删除自身的线程中调用此服务。

> **IMPORTANT**
> 应用程序负责管理与线程堆栈关联的内存区域，该内存区域在此服务完成后可用。此外，应用程序必须阻止使用已删除的线程。

参数

- thread_ptr：指向以前创建的应用程序线程的指针。

返回值

- TX_SUCCESS：(0x00) 成功删除线程。
- TX_THREAD_ERROR：(0x0E) 应用程序线程指针无效。
- TX_DELETE_ERROR：(0x11) 指定的线程未处于已终止或已完成状态。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程和计时器

可以抢占

否

示例

```
TX_THREAD     my_thread;
UINT          status;

/* Delete an application thread whose control block is
   "my_thread". Assume that the thread has already been
   created with a call to tx_thread_create. */
status =  tx_thread_delete(&my_thread);

/* If status equals TX_SUCCESS, the application thread is
   deleted. */
```

另请参阅

- tx_thread_create
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get

- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_entry_exit_notify

在线程进入和退出时通知应用程序

原型

```
UINT  tx_thread_entry_exit_notify(TX_THREAD *thread_ptr,
        VOID (*entry_exit_notify)(TX_THREAD *, UINT));
```

说明

此服务注册一个通知回调函数，每当进入或退出指定的线程时，就会调用该函数。通知回调的处理由应用程序定义。

> **NOTE**
> 不允许应用程序的线程进入/退出通知回调调用任何带有挂起选项的 ThreadX SMP API。

参数

- thread_ptr：指向之前创建的线程的指针。
- entry_exit_notify：指向应用程序的线程进入/退出通知函数的指针。进入/退出通知函数的第二个参数指定是否存在入口或出口。值 TX_THREAD_ENTRY (0x00) 表示已进入线程，而值 TX_THREAD_EXIT (0x01) 表示已退出线程。如果此值为 TX_NULL，则禁用通知。

返回值

- TX_SUCCESS：(0x00) 成功注册线程进入/退出通知函数。
- TX_THREAD_ERROR (0x0E) 线程指针无效。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时禁用了通知功能。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_THREAD         my_thread;

/* Register the "my_entry_exit_notify" function for monitoring
   the entry/exit of the thread "my_thread." */
status =  tx_thread_entry_exit_notify(&my_thread,
                my_entry_exit_notify);

/* If status is TX_SUCCESS the entry/exit notification function was
   successfully registered. */
void my_entry_exit_notify(TX_THREAD *thread_ptr, UINT condition)
{

    /* Determine if the thread was entered or exited. */
    if (condition == TX_THREAD_ENTRY)
                /* Thread entry! */
    else if (condition == TX_THREAD_EXIT)
         /* Thread exit! */
}
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_identify

检索指向当前正在执行的线程的指针

原型

```
TX_THREAD* tx_thread_identify(VOID);
```

说明

此服务将返回指向当前正在执行的线程的指针。如果没有正在执行的线程，则此服务返回空指针。

> **IMPORTANT**
>
> 如果从 ISR 调用此服务，则返回值表示在执行中断处理程序之前运行的线程。

参数

无

返回值

- 线程指针：指向当前正在执行的线程的指针。如果没有正在执行的线程，则返回值为 TX_NULL。

获准方式

线程和 ISR

可以抢占

否

示例

```
TX_THREAD     *my_thread_ptr;

/* Find out who we are! */
my_thread_ptr =  tx_thread_identify();

/* If my_thread_ptr is non-null, we are currently executing
   from that thread or an ISR that interrupted that thread.
   Otherwise, this service was called
   from an ISR when no thread was running when the
   interrupt occurred.  */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_info_get

检索有关线程的信息

原型

```
UINT tx_thread_info_get(TX_THREAD *thread_ptr, CHAR **name,
                        UINT *state, ULONG *run_count,
                        UINT *priority,
                        UINT *preemption_threshold,
                        ULONG *time_slice,
                        TX_THREAD **next_thread,
                        TX_THREAD **suspended_thread);
```

说明

此服务检索有关指定线程的信息。

参数

- thread_ptr：指向线程控制块的指针。
- name：指向线程名称的指针。
- state：指向线程的当前执行状态的指针。可能的值如下：
  - TX_READY：(0x00)
  - TX_COMPLETED：(0x01)
  - TX_TERMINATED：(0x02)
  - TX_SUSPENDED：(0x03)
  - TX_SLEEP：(0x04)
  - TX_QUEUE_SUSP：(0x05)
  - TX_SEMAPHORE_SUSP：(0x06)
  - TX_EVENT_FLAG：(0x07)
  - TX_BLOCK_MEMORY：(0x08)
  - TX_BYTE_MEMORY：(0x09)
  - TX_MUTEX_SUSP：(0x0D)
- run_count：指向线程的运行计数的指针。
- priority：指向线程优先级的指针。
- preemption_threshold：指向线程抢占阈值的指针。
- time_slice：指向线程的时间片的指针。
- next_thread：指向下一个已创建的线程的指针。
- suspended_thread：指向挂起列表中的下一个线程的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功检索线程信息。
- TX_THREAD_ERROR：(0x0E) 线程控制指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_THREAD my_thread;
CHAR *name;
UINT state;
ULONG run_count;
UINT priority;
UINT preemption_threshold;
UINT time_slice;
TX_THREAD *next_thread;
TX_THREAD *suspended_thread;
UINT status;

/* Retrieve information about the previously created
   thread "my_thread." */
status =  tx_thread_info_get(&my_thread, &name,
                  &state, &run_count,
            &priority, &preemption_threshold,
                  &time_slice, &next_thread,&suspended_thread);
/* If status equals TX_SUCCESS, the information requested is
   valid. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_performance_info_get

获取线程性能信息

原型

```
UINT  tx_thread_performance_info_get(TX_THREAD *thread_ptr,
        ULONG *resumptions, ULONG *suspensions,
        ULONG *solicited_preemptions, ULONG *interrupt_preemptions,
        ULONG *priority_inversions, ULONG *time_slices,
        ULONG *relinquishes, ULONG *timeouts, ULONG *wait_aborts,
        TX_THREAD **last_preempted_by);
```

说明

此服务检索有关指定线程的性能信息。

> **IMPORTANT**
> 必须使用已定义的 TX_THREAD_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- thread_ptr：指向之前创建的线程的指针。
- resumptions：指向此线程的恢复数的指针。
- suspensions：指向此线程的挂起数的指针。
- solicited_preemptions：指向由于此线程进行的 ThreadX API 服务调用而导致的抢占数的指针。
- interrupt_preemptions：指向此线程由于中断处理而导致的抢占数的指针。
- priority_inversions：指向此线程的优先级倒置数的指针。
- time_slices：指向此线程的时间片数的指针。
- relinquishes：指向此线程执行的线程放弃次数的指针。
- timeouts：指向此线程的挂起超时次数的指针。
- wait_aborts：指向对此线程执行的等待中止数的指针。
- last_preempted_by：指向最后抢占此线程的线程的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取线程性能信息。
- TX_PTR_ERROR：(0x03) 线程指针无效。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_THREAD     my_thread;
ULONG         resumptions;
ULONG         suspensions;
ULONG         solicited_preemptions;
ULONG         interrupt_preemptions;
ULONG         priority_inversions;
ULONG         time_slices;
ULONG         relinquishes;
ULONG         timeouts;
ULONG         wait_aborts;
TX_THREAD     *last_preempted_by;

/* Retrieve performance information on the previously created
   thread. */
status = tx_thread_performance_info_get(&my_thread, &resumptions,
                &suspensions,
                &solicited_preemptions, &interrupt_preemptions,
                &priority_inversions, &time_slices,
                &relinquishes, &timeouts,
                &wait_aborts, &last_preempted_by);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_performance_system_info_get

获取线程系统性能信息

原型

```
UINT tx_thread_performance_system_info_get(ULONG *resumptions,
       ULONG *suspensions, ULONG *solicited_preemptions,
       ULONG *interrupt_preemptions, ULONG *priority_inversions,
       ULONG *time_slices, ULONG *relinquishes, ULONG *timeouts,
       ULONG *wait_aborts, ULONG *non_idle_returns,
       ULONG *idle_returns);
```

说明

此服务检索系统中所有线程的性能信息。

> **IMPORTANT**
> 必须使用已定义的 TX_THREAD_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- resumptions：指向线程恢复总数的指针。
- suspensions：指向线程挂起总数的指针。
- solicited_preemptions：指向由于线程调用 ThreadX API 服务而导致的线程抢占总数的指针。
- interrupt_preemptions：指向由于中断处理而导致的线程抢占总数的指针。
- priority_inversions：指向线程优先级倒置总数的指针。
- resumptions：指向线程时间片总数的指针。
- relinquishes：指向线程放弃总次数的指针。
- timeouts：指向线程挂起超时总次数的指针。
- wait_aborts：指向线程等待中止总数的指针。
- non_idle_returns：指向线程在另一个线程准备好执行时返回到系统的次数的指针。
- idle_returns：指向线程在没有任何其他线程准备好执行（系统空闲）时返回到系统的次数的指针。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取线程系统性能信息。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
ULONG         resumptions;
ULONG         suspensions;
ULONG         solicited_preemptions;
ULONG         interrupt_preemptions;
ULONG         priority_inversions;
ULONG         time_slices;
ULONG         relinquishes;
ULONG         timeouts;
ULONG         wait_aborts;
ULONG         non_idle_returns;
ULONG         idle_returns;

/* Retrieve performance information on all previously created
   thread. */
status = tx_thread_performance_system_info_get(&resumptions,
                &suspensions,
                &solicited_preemptions, &interrupt_preemptions,
                &priority_inversions, &time_slices, &relinquishes,
                &timeouts, &wait_aborts, &non_idle_returns,
                &idle_returns);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_preemption_change

更改应用程序线程的抢占阈值

原型

```
UINT tx_thread_preemption_change(TX_THREAD *thread_ptr,
                         UINT new_threshold, UINT *old_threshold);
```

说明

此服务更改指定线程的抢占阈值。抢占阈值阻止指定的线程被等于或小于抢占阈值的线程抢占。

> **IMPORTANT**
> 使用抢占阈值会禁用指定线程的时间片。

参数

- thread_ptr：指向以前创建的应用程序线程的指针。
- new_threshold：新的抢占阈值优先级（0 至 TX_MAX_PRIORITIES-1）。
- old_threshold：指向恢复为上一个抢占阈值的位置的指针。

返回值

- TX_SUCCESS：(0x00) 成功更改抢占阈值。
- TX_THREAD_ERROR：(0x0E) 应用程序线程指针无效。
- TX_THRESH_ERROR：(0x18) 指定的新抢占阈值是无效的线程优先级（值不介于 0 到 TX_MAX_PRIORITIES-1 之间）或大于（优先级更低）当前线程优先级。
- TX_PTR_ERROR：(0x03) 指向之前的抢占阈值存储位置的指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程和计时器

可以抢占

是

示例

```
TX_THREAD     my_thread;
UINT          my_old_threshold;
UINT          status;

/* Disable all preemption of the specified thread. The
   current preemption-threshold is returned in
   "my_old_threshold". Assume that "my_thread" has
   already been created. */
status = tx_thread_preemption_change(&my_thread,
                             0, &my_old_threshold);

/* If status equals TX_SUCCESS, the application thread is
   non-preemptable by another thread. Note that ISRs are
   not prevented by preemption disabling. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep

- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_priority_change

更改应用程序线程的优先级

原型

```
UINT tx_thread_priority_change(TX_THREAD *thread_ptr,
                           UINT new_priority, UINT *old_priority);
```

说明

此服务更改指定线程的优先级。有效优先级的范围为 0 至 TX_MAX_PRIORITES-1，其中 0 表示最高优先级。

> **IMPORTANT**
> 指定线程的抢占阈值会自动设置为新的优先级。如果需要新的阈值，则必须在此调用之后使用 tx_thread_preemption_change 服务。

参数

- thread_ptr：指向以前创建的应用程序线程的指针。
- new_priority：新的线程优先级（0 至 TX_MAX_PRIORITIES-1）。
- old_priority：指向恢复为上一个线程优先级的位置的指针。

返回值

- TX_SUCCESS：(0x00) 成功更改优先级。
- TX_THREAD_ERROR：(0x0E) 应用程序线程指针无效。
- TX_PRIORITY_ERROR：(0x0F) 指定的新优先级无效（值不介于 0 到 TX_MAX_PRIORITIES-1 之间）。
- TX_PTR_ERROR：(0x03) 指向之前的优先级存储位置的指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程和计时器

可以抢占

是

示例

```
TX_THREAD     my_thread;
UINT          my_old_priority;
UINT          status;

/* Change the thread represented by "my_thread" to priority
   0. */
status = tx_thread_priority_change(&my_thread,
                            0, &my_old_priority);

/* If status equals TX_SUCCESS, the application thread is
   now at the highest priority level in the system. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_relinquish

将控制权让给其他应用程序线程

原型

```
VOID tx_thread_relinquish(VOID);
```

说明

此服务将处理器控制权让给其他具有相同或更高优先级的就绪线程。

> **IMPORTANT**
> 除了将控制权让给具有相同优先级的线程以外，此服务还将控制权让给由于当前线程的抢占阈值设置而阻止执行的最高优先级线程。

参数

无

返回值

无

获准方式

线程数

可以抢占

是

示例

```
ULONG run_counter_1 =  0;
ULONG run_counter_2 =  0;

/* Example of two threads relinquishing control to
   each other in an infinite loop. Assume that
   both of these threads are ready and have the same
   priority. The run counters will always stay within one
   of each other. */

VOID  my_first_thread(ULONG thread_input)
{
    /* Endless loop of relinquish. */
    while(1)
    {

        /* Increment the run counter. */
        run_counter_1++;

        /* Relinquish control to other thread. */
        tx_thread_relinquish();
    }
}

VOID my_second_thread(ULONG thread_input)
{
    /* Endless loop of relinquish. */
    while(1)
    {
        /* Increment the run counter. */
        run_counter_2++;

        /* Relinquish control to other thread. */
        tx_thread_relinquish();
    }
}
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_reset

重置线程

原型

```
UINT tx_thread_reset(TX_THREAD *thread_ptr);
```

说明

此服务重置指定的线程，使其在创建线程时定义的入口点执行。线程必须处于 TX_COMPLETED 或 TX_TERMINATED 状态才能重置

> **IMPORTANT**
> 线程必须恢复才能再次执行。

参数

- mutex_ptr：指向之前创建的线程的指针。

返回值

- TX_SUCCESS：(0x00) 成功重置线程。
- TX_NOT_DONE：(0x20) 指定的线程未处于 TX_COMPLETED 或 TX_TERMINATED 状态。
- TX_THREAD_ERROR (0x0E) 线程指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程数

示例

```
TX_THREAD     my_thread;

/* Reset the previously created thread "my_thread." */
status = tx_thread_reset(&my_thread);

/* If status is TX_SUCCESS the thread is reset. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate

- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_resume

恢复挂起的应用程序线程

原型

```
UINT tx_thread_resume(TX_THREAD *thread_ptr);
```

说明

此服务恢复或准备执行之前由 tx_thread_suspend 调用挂起的线程。此外，此服务将恢复在创建时未启用自动启动的线程。

参数

- thread_ptr：指向挂起的应用程序线程的指针。

返回值

- TX_SUCCESS：(0x00) 成功恢复线程。
- TX_SUSPEND_LIFTED：(0x19) 之前设置的延迟挂起被解除。
- TX_THREAD_ERROR：(0x0E) 应用程序线程指针无效。
- TX_RESUME_ERROR：(0x12) 指定的线程未挂起，或者以前被 tx_thread_suspend 以外的服务挂起。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_THREAD     my_thread;
UINT          status;

/* Resume the thread represented by "my_thread". */
status =  tx_thread_resume(&my_thread);

/* If status equals TX_SUCCESS, the application thread is
   now ready to execute. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset

- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_sleep

在指定的时间内挂起当前线程

原型

```
UINT tx_thread_sleep(ULONG timer_ticks);
```

说明

此服务使调用线程在指定的计时器时钟周期数内挂起。与计时器时钟周期相关的物理时间量特定于应用程序。此服务只能从应用程序线程调用。

参数

- timer_ticks：挂起调用应用程序线程的计时器时钟周期数，范围为 0 至 0xFFFFFFFF。如果指定 0，服务将立即返回。

返回值

- TX_SUCCESS：(0x00) 线程成功进入睡眠状态。
- TX_WAIT_ABORTED：(0x1A) 挂起状态由其他线程、计时器或 ISR 中止。
- TX_CALLER_ERROR：(0x13) 从非线程调用了服务。

获准方式

线程数

可以抢占

是

示例

```
UINT status;

/* Make the calling thread sleep for 100
   timer-ticks. */
status = tx_thread_sleep(100);

/* If status equals TX_SUCCESS, the currently running
   application thread slept for the specified number of
   timer-ticks. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get

- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_smp_core_exclude

在一组内核上排除线程执行

原型

```
UINT  tx_thread_smp_core_exclude(TX_THREAD *thread_ptr,
                          ULONG exclusion_map);
```

说明

此函数排除指定的线程在称为"exclusion_map"的位映射中指定的内核上执行。"exclusion_map"中的每个位代表一个内核(位 0 代表内核 0，依此类推)。如果设置了位，则会排除指定的线程在对应的内核上执行。

> **IMPORTANT**
> 如果使用处理器排除，则可能会在线程中对内核映射逻辑进行附加处理，以便找到最佳匹配项。此处理受就绪线程数的限制。

参数

- thread_ptr：指向要更改内核排除的线程的指针。
- exclusion_map：位映射，其中的位表示排除对应的内核。如果提供值 0，则线程可以在任何内核上执行(默认设置)。

返回值

- TX_SUCCESS：(0x00) 成功排除内核。
- TX_THREAD_ERROR (0x0E) 线程指针无效。

获准方式

初始化、ISR、线程和计时器

示例

```
/* Exclude core 0 for "Thread 0". */
tx_thread_smp_core_exclude(&thread_0, 0x01);
```

另请参阅

- tx_thread_smp_core_exclude_get
- tx_thread_smp_core_get

tx_thread_smp_core_exclude_get

获取线程的当前内核排除

原型

```
UINT tx_thread_smp_core_exclude_get(TX_THREAD *thread_ptr,
                        ULONG *exclusion_map_ptr);
```

说明

此函数返回当前的内核排除列表。

参数

- thread_ptr：指向要从中检索内核排除的线程的指针。
- exclusion_map_ptr：当前内核排除位映射的目标。

返回值

- TX_SUCCESS：(0x00) 成功检索线程的内核排除。
- TX_THREAD_ERROR (0x0E) 线程指针无效。
- TX_PTR_ERROR：(0x03) 指向排除目标的指针无效。

获准方式

初始化、ISR、线程和计时器

示例

```
ULONGexcluded_cores;
/* Retrieve the core exclusion for "Thread 0". */
tx_thread_smp_core_exclude_get(&thread_0, &excluded_cores);
```

另请参阅

- tx_thread_smp_core_exclude
- tx_thread_smp_core_get

tx_thread_smp_core_get

检索调用方的当前执行内核

原型

```
UINT  tx_thread_smp_core_get(void);
```

说明

此函数返回执行该服务的内核的内核 ID。

参数

无

返回值

- core_id：当前执行内核的 ID（0 至 TX_THREAD_SMP_MAX_CORES-1）

获准方式

初始化、ISR、线程和计时器

示例

```
UINTcore;
/* Pickup the currently executing core. */
core = tx_thread_smp_core_get();

/* At this point, “core” contains the executing core ID. */
```

另请参阅

- tx_thread_smp_core_exclude
- tx_thread_smp_core_exclude_get

tx_thread_stack_error_notify

注册线程堆栈错误通知回调

原型

```
UINT tx_thread_stack_error_notify(VOID (*error_handler)(TX_THREAD *));
```

说明

此服务注册用于处理线程堆栈错误的通知回调函数。当 ThreadX SMP 在执行过程中检测到线程堆栈错误时，它将调用此通知函数来处理该错误。对错误的处理措施由应用程序完全定义。可以采取任何措施，包括挂起冲突线程和重置整个系统。

> **IMPORTANT**
> 必须使用已定义的 TX_ENABLE_STACK_CHECKING 生成 ThreadX SMP 库，此服务才能返回性能信息。

参数

- error_handler：指向应用程序的堆栈错误处理函数的指针。如果此值为 TX_NULL，则禁用通知。

返回值

- TX_SUCCESS：(0x00) 成功重置线程。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
void my_stack_error_handler(TX_THREAD *thread_ptr);

/* Register the "my_stack_error_handler" function with ThreadX SMP
   so that thread stack errors can be handled by the application. */
status =  tx_thread_stack_error_notify(my_stack_error_handler);

/* If status is TX_SUCCESS the stack error handler is registered.*/
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify

- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_suspend

挂起应用程序线程

原型

```
UINT tx_thread_suspend(TX_THREAD *thread_ptr);
```

说明

此服务挂起指定的应用程序线程。线程可以调用此服务来挂起自身。

> **IMPORTANT**
> 如果指定的线程已由于其他原因而挂起，则会在内部保持此挂起，直到之前的挂起被解除为止。发生这种情况时，将对指定的线程执行无条件挂起。其他无条件挂起请求不起作用。

挂起后，必须由 tx_thread_resume 恢复该线程才能再次执行。

参数

- thread_ptr：指向应用程序线程的指针。

返回值

- TX_SUCCESS：(0x00) 线程成功挂起。
- TX_THREAD_ERROR：(0x0E) 应用程序线程指针无效。
- TX_SUSPEND_ERROR：(0x14) 指定的线程处于已终止或已完成状态。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_THREAD     my_thread;
UINT          status;

/* Suspend the thread represented by "my_thread". */
status = tx_thread_suspend(&my_thread);

/* If status equals TX_SUCCESS, the application thread is
   unconditionally suspended. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_terminate
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_terminate

终止应用程序线程

原型

```
UINT tx_thread_terminate(TX_THREAD *thread_ptr);
```

说明

不管是否挂起线程，此服务都将终止指定的应用程序线程。线程可以调用此服务来终止自身。

> **IMPORTANT**
> 终止后，必须重置该线程才能再次执行。

> **WARNING**
> 应用程序负责确保线程处于适合终止的状态。例如，在关键应用程序处理期间，或者在此类处理可能会处于未知状态的其他中间件组件中，不应终止线程。

参数

- thread_ptr：指向应用程序线程的指针。

返回值

- TX_SUCCESS：(0x00) 线程成功终止。
- TX_THREAD_ERROR：(0x0E) 应用程序线程指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程和计时器

可以抢占

是

示例

```
TX_THREAD     my_thread;
UINT          status;

/* Terminate the thread represented by "my_thread". */
status =  tx_thread_terminate(&my_thread);

/* If status equals TX_SUCCESS, the thread is terminated
   and cannot execute again until it is reset. */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_time_slice_change
- tx_thread_wait_abort

tx_thread_time_slice_change

更改应用程序线程的时间片

原型

```
UINT tx_thread_time_slice_change(TX_THREAD *thread_ptr,
                          ULONG new_time_slice, ULONG *old_time_slice);
```

说明

此服务更改指定应用程序线程的时间片。为线程选择时间片可确保在具有相同或更高优先级的其他线程有机会

执行之前，该线程的执行时间不会超过指定的计时器时钟周期数。

> **IMPORTANT**
> 使用抢占阈值会禁用指定线程的时间片。

参数

- thread_ptr：指向应用程序线程的指针。
- new_time_slice：新的时间片值。合法值包括 TX_NO_TIME_SLICE 和从 1 到 0xFFFFFFFF 的数值。
- old_time_slice：指向用于存储指定线程先前时间片值的位置的指针。

返回值

- TX_SUCCESS：(0x00) 成功设置时间片。
- TX_THREAD_ERROR：(0x0E) 应用程序线程指针无效。
- TX_PTR_ERROR：(0x03) 指向之前的时间片存储位置的指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程和计时器

可以抢占

否

示例

```
TX_THREAD     my_thread;
ULONG         my_old_time_slice;
UINT          status;

/* Change the time-slice of the thread associated with
   "my_thread" to 20. This will mean that "my_thread"
   can only run for 20 timer-ticks consecutively before
   other threads of equal or higher priority get a chance
   to run. */
status = tx_thread_time_slice_change(&my_thread, 20,
                              &my_old_time_slice);

/* If status equals TX_SUCCESS, the thread's time-slice
   has been changed to 20 and the previous time-slice is
   in "my_old_time_slice." */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify
- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume

- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_wait_abort

tx_thread_wait_abort

中止指定线程的挂起

原型

```
UINT tx_thread_wait_abort(TX_THREAD *thread_ptr);
```

说明

此服务将中止指定线程的睡眠状态或任何其他对象挂起。如果中止等待，则从线程正在等待的服务返回 TX_WAIT_ABORTED 值。

> **IMPORTANT**
> 此服务不会释放由 tx_thread_suspend 服务执行的显式挂起。

参数

- thread_ptr：指向以前创建的应用程序线程的指针。

返回值

- TX_SUCCESS：(0x00) 线程等待成功中止。
- TX_THREAD_ERROR：(0x0E) 应用程序线程指针无效。
- TX_WAIT_ABORT_ERROR：(0x1b) 指定的线程未处于等待状态。

获准方式

初始化、线程、计时器和 ISR

可以抢占

是

示例

```
TX_THREAD     my_thread;
UINT          status;

/* Abort the suspension condition of "my_thread." */
status =  tx_thread_wait_abort(&my_thread);

/* If status equals TX_SUCCESS, the thread is now ready
   again, with a return value showing its suspension
   was aborted (TX_WAIT_ABORTED). */
```

另请参阅

- tx_thread_create
- tx_thread_delete
- tx_thread_entry_exit_notify
- tx_thread_identify

- tx_thread_info_get
- tx_thread_performance_info_get
- tx_thread_performance_system_info_get
- tx_thread_preemption_change
- tx_thread_priority_change
- tx_thread_relinquish
- tx_thread_reset
- tx_thread_resume
- tx_thread_sleep
- tx_thread_stack_error_notify
- tx_thread_suspend
- tx_thread_terminate
- tx_thread_time_slice_change

tx_time_get

检索当前时间

原型

```
ULONG tx_time_get(VOID);
```

说明

此服务返回内部系统时钟的内容。每个计时器时钟周期将内部系统时钟增加 1。系统时钟在初始化期间设置为零，并且可以通过服务 tx_time_set 更改为特定值。

> **IMPORTANT**
> 每个计时器时钟周期代表的实际时间特定于应用程序。

参数

无

返回值

- 系统时钟计时周期：内部自由运行系统时钟的值。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
ULONG current_time;

/* Pickup the current system time, in timer-ticks. */
current_time =  tx_time_get();

/* Current time now contains a copy of the internal system
   clock. */
```

另请参阅

- tx_time_set

tx_time_set

设置当前时间

原型

```
VOID tx_time_set(ULONG new_time);
```

说明

此服务将内部系统时钟设置为指定值。每个计时器时钟周期将内部系统时钟增加 1。

> **IMPORTANT**
> 每个计时器时钟周期代表的实际时间特定于应用程序。

参数

- new_time：要添加到系统时钟的新时间，合法值的范围为 0 至 0xFFFFFFFF。

返回值

无

获准方式

线程、计时器和 ISR

可以抢占

否

示例

```
/* Set the internal system time to 0x1234. */
tx_time_set(0x1234);

/* Current time now contains 0x1234 until the next timer
   interrupt. */
```

另请参阅

- tx_time_get

tx_timer_activate

激活应用程序计时器

原型

```
UINT tx_timer_activate(TX_TIMER *timer_ptr);
```

说明

此服务激活指定的应用程序计时器。同时过期的计时器的过期例程将按其激活顺序执行。

> **NOTE**
> 过期的一次性计时器必须通过 tx_timer_change 重置，然后才能再次激活。

参数

- timer_ptr：指向以前创建的应用程序计时器的指针。

返回值

- TX_SUCCESS：(0x00) 成功激活应用程序计时器。
- TX_TIMER_ERROR：(0x15) 应用程序计时器指针无效。
- TX_ACTIVATE_ERROR：(0x17) 计时器已处于活动状态或是已过期的一次性计时器。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_TIMER     my_timer;
UINT         status;

/* Activate an application timer. Assume that the
   application timer has already been created. */
status = tx_timer_activate(&my_timer);

/* If status equals TX_SUCCESS, the application timer is
   now active. */
```

另请参阅

- tx_timer_change
- tx_timer_create
- tx_timer_deactivate
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

tx_timer_change

更改应用程序计时器

原型

```
UINT tx_timer_change(TX_TIMER *timer_ptr,
                         ULONG initial_ticks, ULONG reschedule_ticks);
```

说明

此服务更改指定应用程序计时器的过期特性。必须先停用该计时器，然后才能调用此服务。

> **IMPORTANT**
> 在此服务之后，需要调用 tx_timer_activate 服务，以便再次启动计时器。

参数

- timer_ptr：指向计时器控制块的指针。
- initial_ticks：指定计时器过期的初始时钟周期数。合法值的范围为 1 至 0xFFFFFFFF。
- reschedule_ticks：指定第一个计时器过期后所有计时器过期的时钟周期数。如果此参数为零，则会使计时器成为一次性计时器。否则，对于周期性计时器，合法值的范围为 1 至 0xFFFFFFFF。

> **NOTE**
> 过期的一次性计时器必须通过 tx_timer_change 重置，然后才能再次激活。

返回值

- TX_SUCCESS：(0x00) 成功更改应用程序计时器。
- TX_TIMER_ERROR：(0x15) 应用程序计时器指针无效。
- TX_TICK_ERROR：(0x16) 为初始时钟周期数提供的值（零）无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程、计时器和 ISR

可以抢占

否

示例

```
TX_TIMER         my_timer;
UINT             status;

/* Change a previously created and now deactivated timer
   to expire every 50 timer ticks, including the initial
   expiration. */
status =  tx_timer_change(&my_timer,50, 50);

/* If status equals TX_SUCCESS, the specified timer is
   changed to expire every 50 ticks. */

/* Activate the specified timer to get it started again. */
status = tx_timer_activate(&my_timer);
```

另请参阅

- tx_timer_activate
- tx_timer_create
- tx_timer_deactivate
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

tx_timer_create

创建应用程序计时器

原型

```
UINT tx_timer_create(TX_TIMER *timer_ptr, CHAR *name_ptr,
                        VOID (*expiration_function)(ULONG),
                        ULONG expiration_input, ULONG initial_ticks,
                        ULONG reschedule_ticks, UINT auto_activate)
```

说明

此服务创建具有指定过期函数和定期的应用程序计时器。

参数

- timer_ptr：指向计时器控制块的指针
- name_ptr：指向计时器名称的指针。
- expiration_function：在计时器过期时要调用的应用程序函数。
- expiration_input：在计时器过期时要传递到过期函数的输入。
- initial_ticks：指定计时器过期的初始时钟周期数。合法值的范围为 1 至 0xFFFFFFFF。
- reschedule_ticks：指定第一个计时器过期后所有计时器过期的时钟周期数。如果此参数为零，则会使计时器成为一次性计时器。否则，对于周期性计时器，合法值的范围为 1 至 0xFFFFFFFF。

  > **NOTE**
  > 一次性计时器过期后，必须通过 tx_timer_change 将其重置，然后才能再次激活。

- auto_activate：确定创建期间是否自动激活计时器。如果此值为 TX_AUTO_ACTIVATE (0x01)，则激活计时器。否则，如果选择了值 TX_NO_ACTIVATE (0x00)，则所创建的计时器处于非活动状态。在这种情况下，随后需要调用 tx_timer_activate 服务来实际启动计时器。

返回值

- TX_SUCCESS：(0x00) 成功创建应用程序计时器。
- TX_TIMER_ERROR：(0x15) 应用程序计时器指针无效。指针为 NULL 或已创建计时器。
- TX_TICK_ERROR：(0x16) 为初始时钟周期数提供的值（零）无效。
- TX_ACTIVATE_ERROR：(0x17) 选择的激活无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

初始化和线程

可以抢占

否

示例

```
TX_TIMER     my_timer;
UINT         status;

/* Create an application timer that executes
   "my_timer_function" after 100 ticks initially and then
   after every 25 ticks. This timer is specified to start
   immediately! */
status =  tx_timer_create(&my_timer,"my_timer_name",
                my_timer_function, 0x1234, 100, 25,
                TX_AUTO_ACTIVATE);

/* If status equals TX_SUCCESS, my_timer_function will
   be called 100 timer ticks later and then called every
   25 timer ticks. Note that the value 0x1234 is passed to
   my_timer_function every time it is called. */
```

另请参阅

- tx_timer_activate
- tx_timer_change
- tx_timer_deactivate
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

tx_timer_deactivate

停用应用程序计时器

原型

```
UINT tx_timer_deactivate(TX_TIMER *timer_ptr);
```

说明

此服务停用指定的应用程序计时器。如果计时器已停用，则此服务不起作用。

参数

- timer_ptr：指向以前创建的应用程序计时器的指针。

返回值

- TX_SUCCESS：(0x00) 成功停用应用程序计时器。
- TX_TIMER_ERROR：(0x15) 应用程序计时器指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_TIMER     my_timer;
UINT         status;

/* Deactivate an application timer. Assume that the
   application timer has already been created. */
status =  tx_timer_deactivate(&my_timer);

/* If status equals TX_SUCCESS, the application timer is
   now deactivated. */
```

另请参阅

- tx_timer_activate
- tx_timer_change
- tx_timer_create
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

tx_timer_delete

删除应用程序计时器

原型

```
UINT tx_timer_delete(TX_TIMER *timer_ptr);
```

说明

此服务删除指定的应用程序计时器。

> **IMPORTANT**
> 应用程序负责阻止使用已删除的计时器。

参数

- timer_ptr：指向以前创建的应用程序计时器的指针。

返回值

- TX_SUCCESS：(0x00) 成功删除应用程序计时器。
- TX_TIMER_ERROR：(0x15) 应用程序计时器指针无效。
- NX_CALLER_ERROR：(0x13) 此服务的调用方无效。

获准方式

线程数

可以抢占

否

示例

```
TX_TIMER     my_timer;
UINT         status;

/* Delete application timer. Assume that the application
   timer has already been created. */
status =  tx_timer_delete(&my_timer);

/* If status equals TX_SUCCESS, the application timer is
   deleted. */
```

另请参阅

- tx_timer_activate
- tx_timer_change
- tx_timer_create
- tx_timer_deactivate
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

tx_timer_info_get

检索有关应用程序计时器的信息

原型

```
UINT tx_timer_info_get(TX_TIMER *timer_ptr, CHAR **name,
                       UINT *active, ULONG *remaining_ticks,
                       ULONG *reschedule_ticks,
                       TX_TIMER **next_timer)
```

说明

此服务检索有关指定应用程序计时器的信息。

参数

- timer_ptr：指向以前创建的应用程序计时器的指针。
- name：指向计时器名称的指针。
- active：指向计时器活动指示的指针。如果计时器处于非活动状态或从计时器本身调用此服务，则返回 TX_FALSE 值。否则，如果计时器处于活动状态，则返回 TX_TRUE 值。
- remaining_ticks：指向在计时器过期前剩余的计时器时钟周期数的指针。
- reschedule_ticks：指向将用于自动重新计划此计时器的计时器时钟周期数的指针。如果该值为零，则计时器是一次性的，不会重新计划。
- next_timer：指向下一个已创建的应用程序计时器的指针。

> **NOTE**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功检索计时器信息。
- TX_TIMER_ERROR：(0x15) 应用程序计时器指针无效。

获准方式

初始化、线程、计时器和 ISR

可以抢占

否

示例

```
TX_TIMER     my_timer;
CHAR         *name;
UINT         active;
ULONG        remaining_ticks;
ULONG        reschedule_ticks;
TX_TIMER     *next_timer;
UINT         status;

/* Retrieve information about the previously created
   application timer "my_timer." */
status =  tx_timer_info_get(&my_timer, &name,
                            &active,&remaining_ticks,
                &reschedule_ticks,
                         &next_timer);

/* If status equals TX_SUCCESS, the information requested is
   valid. */
```

另请参阅

- tx_timer_activate
- tx_timer_change
- tx_timer_create
- tx_timer_deactivate
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_info_get
- tx_timer_performance_system_info_get

tx_timer_performance_info_get

获取计时器性能信息

原型

```
UINT  tx_timer_performance_info_get(TX_TIMER *timer_ptr,
        ULONG *activates, ULONG *reactivates,
        ULONG *deactivates, ULONG *expirations,
        ULONG *expiration_adjusts);
```

说明

此服务检索有关指定应用程序计时器的性能信息。

> **IMPORTANT**
> 必须使用已定义的 TX_TIMER_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- timer_ptr：指向之前创建的计时器的指针。

- activates：指向对此计时器执行的激活请求数的指针。
- reactivates：指向对此周期性计时器执行的自动重新激活次数的指针。
- deactivates：指向对此计时器执行的停用请求数的指针。
- expirations：指向此计时器的过期次数的指针。
- expiration_adjusts：指向对此计时器执行的内部过期调整次数的指针。这些调整是在计时器中断处理中完成的，针对大于默认计时器列表大小的计时器（默认情况下即为过期时间大于 32 个时钟周期的计时器）。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取计时器性能信息。
- TX_PTR_ERROR：(0x03) 计时器指针无效。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
TX_TIMER     my_timer;
ULONG        activates;
ULONG        reactivates;
ULONG        deactivates;
ULONG        expirations;
ULONG        expiration_adjusts;

/* Retrieve performance information on the previously created
   timer.  */
status =  tx_timer_performance_info_get(&my_timer, &activates,
                &reactivates,&deactivates, &expirations,
                &expiration_adjusts);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_timer_activate
- tx_timer_change
- tx_timer_create
- tx_timer_deactivate
- tx_timer_delete
- tx_timer_info_get
- tx_timer_performance_system_info_get

tx_timer_performance_system_info_get

获取计时器系统性能信息

原型

```
UINT  tx_timer_performance_system_info_get(ULONG *activates,
        ULONG *reactivates, ULONG *deactivates,
        ULONG *expirations, ULONG *expiration_adjusts);
```

说明

此服务检索有关系统中所有应用程序计时器的性能信息。

> **IMPORTANT**
> 必须使用已定义的 TX_TIMER_ENABLE_PERFORMANCE_INFO 生成 ThreadX SMP 库和应用程序，此服务才能返回性能信息。

参数

- activates：指向对所有计时器执行的激活请求总数的指针。
- reactivates：指向对所有周期性计时器执行的自动重新激活总次数的指针。
- deactivates：指向对所有计时器执行的停用请求总数的指针。
- expirations：指向所有计时器的过期总次数的指针。
- expiration_adjusts：指向对所有计时器执行的内部过期调整总次数的指针。这些调整是在计时器中断处理中完成的，针对大于默认计时器列表大小的计时器（默认情况下即为过期时间大于 32 个时钟周期的计时器）。

> **IMPORTANT**
> 为任何参数提供 TX_NULL 表示该参数不是必需的。

返回值

- TX_SUCCESS：(0x00) 成功获取计时器系统性能信息。
- TX_FEATURE_NOT_ENABLED：(0xFF) 在编译系统时未启用性能信息。

获准方式

初始化、线程、计时器和 ISR

示例

```
ULONG     activates;
ULONG     reactivates;
ULONG     deactivates;
ULONG     expirations;
ULONG     expiration_adjusts;

/* Retrieve performance information on all previously created
   timers.  */
status =  tx_timer_performance_system_info_get(&activates,
                &reactivates, &deactivates, &expirations,
                &expiration_adjusts);
/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

另请参阅

- tx_timer_activate
- tx_timer_change
- tx_timer_create
- tx_timer_deactivate
- tx_timer_delete

- tx_timer_info_get
- tx_timer_performance_info_get

tx_timer_smp_core_exclude

在一组内核上排除计时器执行

原型

```
UINT  tx_timer_smp_core_exclude(TX_TIMER *timer_ptr, ULONG exclusion_map);
```

说明

此函数排除指定的计时器在称为"exclusion_map"的位映射中指定的内核上执行。"exclusion_map"中的每个位代表一个内核(位 0 代表内核 0，依此类推)。如果设置了位，则会排除指定的计时器在对应的内核上执行。

> **IMPORTANT**
> 如果使用处理器排除，则可能会在线程中对内核映射逻辑进行附加处理，以便找到最佳匹配项。此处理受就绪线程数的限制。

参数

- timer_ptr：指向要更改内核排除的计时器的指针。
- exclusion_map：位映射，其中的位表示排除对应的内核。如果提供值 0，则计时器可以在任何内核上执行(默认设置)。

返回值

- TX_SUCCESS：(0x00) 成功排除内核。
- TX_TIMER_ERROR：(0x0E) 计时器指针无效。

获准方式

初始化、ISR、线程和计时器

示例

```
/* Exclude core 0 for "Timer 0". */
tx_timer_smp_core_exclude(&timer_0, 0x01);
```

另请参阅

- tx_timer_smp_core_exclude_get

tx_timer_smp_core_exclude_get

获取计时器的当前内核排除

原型

```
UINT tx_timer_smp_core_exclude_get(TX_TIMER *timer_ptr,
                        ULONG *exclusion_map_ptr);
```

说明

此函数返回当前的内核排除列表。

参数

- timer_ptr：指向要从中检索内核排除的计时器的指针。
- exclusion_map_ptr：当前内核排除位映射的目标。

返回值

- TX_SUCCESS：(0x00) 成功检索计时器的内核排除。
- TX_TIMER_ERROR：(0x0E) 计时器指针无效。
- TX_PTR_ERROR：(0x03) 指向排除目标的指针无效。

获准方式

初始化、ISR、线程和计时器

示例

```
ULONGexcluded_cores;

/* Retrieve the core exclusion for "Timer 0". */
tx_timer_smp_core_exclude_get(&timer_0,&excluded_cores);
```

另请参阅

- tx_timer_smp_core_exclude

第 5 章 - 适用于 Azure RTOS ThreadX SMP 的设备驱动程序

2021/4/29 •

本章介绍适用于 Azure RTOS ThreadX SMP 的设备驱动程序。本章介绍的信息旨在帮助开发人员编写特定于应用程序的驱动程序。

设备驱动程序简介

与外部环境的通信是大多数嵌入式应用程序的重要组成部分。此通信通过嵌入式应用程序软件可访问的硬件设备实现。负责管理此类设备的软件组件通常称为设备驱动程序。

嵌入式实时系统中的设备驱动程序本质上依赖于应用程序。这是因为以下两个主要原因：目标硬件非常多样化，对各个实时应用程序施加的性能要求同样巨大。因此，实际上不可能提供一组可满足每个应用程序的要求的通用驱动程序。由于这些原因，本章中的信息旨在帮助用户自定义现成的 ThreadX SMP 设备驱动程序以及编写自己的特定驱动程序。

驱动程序函数

ThreadX SMP 设备驱动程序由八个基本函数区域组成，如下所示：

- 驱动程序初始化
- 驱动程序控制
- 驱动程序访问
- 驱动程序输入
- 驱动程序输出
- 驱动程序中断
- 驱动程序状态
- 驱动程序终止

每个驱动程序函数区域都为可选，但初始化除外。此外，每个区域中的确切处理特定于设备驱动程序。

驱动程序初始化

此函数区域负责初始化实际的硬件设备和驱动程序的内部数据结构。在初始化完成之前，不允许调用其他驱动程序服务。

> **IMPORTANT**
> 通常可通过 tx_application_define 函数或初始化线程调用驱动程序的初始化函数组件。

驱动程序控制

驱动程序初始化并准备好运行后，此函数区域负责运行时控制。通常，运行时控制包括对基础硬件设备进行更改。例如，更改串行设备的波特率或查找磁盘上的新扇区。

驱动程序访问

某些设备驱动程序只能通过单个应用程序线程调用。在这种情况下，不需要此函数区域。但是，在多个线程需要同时访问驱动程序的应用程序中，必须通过在设备驱动程序中添加分配/释放功能来控制这些线程的交互。或者，应用程序也可以使用信号灯来控制驱动程序访问，并避免驱动程序内部出现额外的开销和复杂情况。

驱动程序输入

此函数区域负责所有设备输入。与驱动程序输入相关的主要问题通常涉及如何缓冲输入以及线程如何等待此类输入。

驱动程序输出

此函数区域负责所有设备输出。与驱动程序输出相关的主要问题通常涉及如何缓冲输出以及线程如何等待执行输出。

驱动程序中断

大多数实时系统都依赖硬件中断向驱动程序通知设备输入、输出、控制和错误事件。中断可为此类外部事件提供有保证的响应时间。驱动程序软件可能会定期检查外部硬件以查找此类事件，而不依赖中断。这种技术称为轮询。它的实时性低于中断，但轮询对于一些实时性较低的应用程序可能有意义。

驱动程序状态

此函数区域负责提供与驱动程序操作关联的运行时状态和统计信息。此函数区域管理的信息通常包括以下内容：

- 当前设备状态
- 输入字节数
- 输出字节数
- 设备错误计数

驱动程序终止

此函数区域为可选。仅当驱动程序和/或物理硬件设备需要关闭时，才需要此函数区域。终止后，在重新初始化之前不能再次调用驱动程序。

简单驱动程序示例

举例是介绍设备驱动程序的最佳方式。在此示例中，驱动程序假定简单的串行硬件设备具有配置寄存器、输入寄存器和输出寄存器。此简单驱动程序示例阐释了初始化、输入、输出和中断函数区域。

简单驱动程序初始化

简单驱动程序的 tx_sdriver_initialize 函数可创建两个计数信号灯，用于管理驱动程序的输入和输出操作。当串行硬件设备收到字符时，输入信号灯由输入 ISR 设置。因此，所创建的输入信号灯的初始计数为零。

而输出信号灯用于指示串行硬件传输寄存器的可用性。创建输出信号灯时，将其值设为 1，表示传输寄存器在初始时可用。

初始化函数还负责为输入和输出通知安装低级别中断向量处理程序。与其他 ThreadX SMP 中断服务例程一样，低级别处理程序在调用简单驱动程序 ISR 之前必须调用 *_tx_thread_context_save。驱动程序 ISR 返回后，低级别处理程序必须调用 _*_tx_thread_context_restore**。

> **IMPORTANT**
> 请务必在调用任何其他驱动程序函数之前调用初始化。通常可通过 tx_application_define 调用驱动程序初始化。

有关简单驱动程序的初始化源代码，请参阅第 306 页上的图 9。

```
VOID    tx_sdriver_initialize(VOID)
{

    /* Initialize the two counting semaphores used to control
       the simple driver I/O. */
    tx_semaphore_create(&tx_sdriver_input_semaphore,
                        "simple driver input semaphore", 0);
    tx_semaphore_create(&tx_sdriver_output_semaphore,
                        "simple driver output semaphore", 1);

    /* Setup interrupt vectors for input and output ISRs.
       The initial vector handling should call the ISRs
       defined in this file. */

    /* Configure serial device hardware for RX/TX interrupt
       generation, baud rate, stop bits, etc. */
}
```

图 9. 简单驱动程序初始化

简单驱动程序输入

简单驱动程序的输入以输入信号灯为中心。系统收到串行设备输入中断时，会设置输入信号灯。如果一个或多个线程正在等待从驱动程序接收字符，则会恢复等待时间最长的线程。如果没有任何线程正在等待，则信号灯会保持不变，直到线程调用驱动程序输入函数为止。

系统在处理简单驱动程序输入时有几个限制。其中最重要的一点是，系统可能会删除输入字符。之所以有这种可能性，是因为系统在处理前一个字符之前无法缓冲到达的输入字符。可以通过添加输入字符缓冲区来轻松解决此限制。

> **IMPORTANT**
>
> 只有线程可以调用 tx_sdriver_input 函数。

图 10 显示了与简单驱动程序输入关联的源代码。

```
UCHAR    tx_sdriver_input(VOID)
{

    /* Determine if there is a character waiting. If not,
        suspend. */
    tx_semaphore_get(&tx_sdriver_input_semaphore,
                                            TX_WAIT_FOREVER;
    /* Return character from serial RX hardware register. */
    return(*serial_hardware_input_ptr);
}

VOID    tx_sdriver_input_ISR(VOID)
{
    /* See if an input character notification is pending. */
    if (!tx_sdriver_input_semaphore.tx_semaphore_count)
    {
        /* If not, notify thread of an input character. */
        tx_semaphore_put(&tx_sdriver_input_semaphore);
    }
}
```

图 10. 简单驱动程序输入

简单驱动程序输出

当串行设备的传输寄存器空闲时，输出处理将使用输出信号灯发出信号。在将输出字符实际写入到设备之前，系

统会获取输出信号灯的相关信息。如果该信号灯不可用，则表示之前的传输尚未完成。

输出 ISR 负责处理传输完成中断。处理输出 ISR 等同于设置输出信号灯，从而允许输出另一个字符。

> **IMPORTANT**
> 只有线程可以调用 tx_sdriver_output函数。

图 11 显示了与简单驱动程序输出关联的源代码。

```
VOID     tx_sdriver_output(UCHAR alpha)
{

    /* Determine if the hardware is ready to transmit a
       character. If not, suspend until the previous output
        completes. */
    tx_semaphore_get(&tx_sdriver_output_semaphore,
                                          TX_WAIT_FOREVER);
    /* Send the character through the hardware. */
    *serial_hardware_output_ptr = alpha;
}

VOID     tx_sdriver_output_ISR(VOID)
{
    /* Notify thread last character transmit is
        complete. */
    tx_semaphore_put(&tx_sdriver_output_semaphore);
}
```

图 11. 简单驱动程序输出

简单驱动程序的缺点

此简单设备驱动程序示例阐释了 ThreadX SMP 设备驱动程序的基本概念。但是，由于简单设备驱动程序无法解决数据缓冲问题或任何开销问题，因此它不能完全代表实际 ThreadX SMP 驱动程序。下一节介绍了与设备驱动程序相关的一些更高级的问题。

驱动程序高级问题

如前所述，设备驱动程序具有与应用程序不同的要求。某些应用程序可能需要缓冲大量数据，而其他应用程序可能由于设备中断频率较高而需要优化的驱动程序 ISR。

I/O 缓冲

实时嵌入式应用程序中的数据缓冲需要进行大量规划。有些设计由基础硬件设备决定。如果设备提供基本的字节 I/O，则简单的循环缓冲区可能符合要求。但是，如果设备提供块 I/O、DMA I/O 或数据包 I/O，则可能需要缓冲区管理方案。

循环字节缓冲区

循环字节缓冲区通常在管理简单串行硬件设备（例如 UART）的驱动程序中使用。这种情况下最常使用两个循环缓冲区：一个用于输入，另一个用于输出。

每个循环字节缓冲区都包含一个字节内存区域（通常是一个 UCHAR 数组）、一个读指针和一个写指针。如果读指针和写指针引用缓冲区中的同一个内存位置，则将缓冲区视为空的。驱动程序初始化会将缓冲区读指针和缓冲区写指针设置为缓冲区的起始地址。

循环缓冲区输入

输入缓冲区用于在应用程序准备就绪之前保存到达的字符。收到输入字符（通常是在中断服务例程中）时，将从硬件设备检索新字符，并将其放入输入缓冲区中写指针所指向的位置。然后，写指针将前进到缓冲区中的下一个位置。如果下一个位置超出了缓冲区的末尾，则写指针将设置为缓冲区的开头。如果新的写指针与读指针相同，

则通过取消写指针前进来解决队列已满状况。

向驱动程序发送的应用程序输入字节请求首先检查输入缓冲区的读指针和写指针。如果读指针与写指针相同，则缓冲区为空。否则，如果读指针不相同，则会从输入缓冲区中复制读指针所指向的字节，并且读指针会前进到下一个缓冲区位置。如果新的读指针超出了缓冲区的末尾，则将其重置为开头。图 12 显示了循环输入缓冲区的逻辑。

```
UCHAR     tx_input_buffer[MAX_SIZE];
UCHAR     tx_input_write_ptr;
UCHAR     tx_input_read_ptr;

/* Initialization.  */
tx_input_write_ptr =  &tx_input_buffer[0];
tx_input_read_ptr =    &tx_input_buffer[0];

/* Input byte ISR... UCHAR alpha has character from device. */
save_ptr =  tx_input_write_ptr;
*tx_input_write_ptr++ =  alpha;
if (tx_input_write_ptr > &tx_input_buffer[MAX_SIZE-1])
    tx_input_write_ptr =  &tx_input_buffer[0];  /* Wrap */
if (tx_input_write_ptr == tx_input_read_ptr)
    tx_input_write_ptr =  save_ptr;  /* Buffer full */

/* Retrieve input byte from buffer... */
if (tx_input_read_ptr != tx_input_write_ptr)
{
    alpha =  *tx_input_read_ptr++;
    if (tx_input_read_ptr > &tx_input_buffer[MAX_SIZE-1])
        tx_input_read_ptr =  &tx_input_buffer[0];
}
```

图 12. 循环输入缓冲区的逻辑

> **IMPORTANT**
> 为了确保操作的可靠性，在操作循环输入缓冲区以及循环输出缓冲区的读指针和写指针时可能需要锁定中断。

循环输出缓冲区

输出缓冲区用于在硬件设备完成发送前一个字节之前保存到达用于输出的字符。输出缓冲区处理类似于输入缓冲区处理，不同之处在于传输完成中断处理操作输出读指针，而应用程序输出请求使用输出写指针。在其他方面，输出缓冲区处理与输入缓冲区处理相同。图 13 显示了循环输出缓冲区的逻辑。

```
UCHAR     tx_output_buffer[MAX_SIZE];
UCHAR     tx_output_write_ptr;
UCHAR     tx_output_read_ptr;

/* Initialization.  */
tx_output_write_ptr =  &tx_output_buffer[0];
tx_output_read_ptr =   &tx_output_buffer[0];

/* Transmit complete ISR... Device ready to send. */
if (tx_output_read_ptr != tx_output_write_ptr)
{
    *device_reg =  *tx_output_read_ptr++;
    if (tx_output_read_reg > &tx_output_buffer[MAX_SIZE-1])
    tx_output_read_ptr =  &tx_output_buffer[0];
}

/* Output byte driver service. If device busy, buffer! */
save_ptr =  tx_output_write_ptr;
*tx_output_write_ptr++ =  alpha;
if (tx_output_write_ptr > &tx_output_buffer[MAX_SIZE-1])
    tx_output_write_ptr =  &tx_output_buffer[0];  /* Wrap */
if (tx_output_write_ptr == tx_output_read_ptr)
    tx_output_write_ptr =  save_ptr;  /* Buffer full!  */
```

图 13. 循环输出缓冲区的逻辑

缓冲区 I/O 管理

为了提高嵌入式微处理器的性能，许多外围设备使用软件提供的缓冲区来传输和接收数据。在某些实现中，可能会使用多个缓冲区来传输或接收单个数据包。

I/O 缓冲区的大小和位置由应用程序和/或驱动程序软件确定。通常，缓冲区的大小是固定的，并在 ThreadX SMP 块内存池中进行管理。图 14 介绍了典型 I/O 缓冲区以及管理缓冲区分配的 ThreadX SMP 块内存池。

```
typedef struct TX_IO_BUFFER_STRUCT
{

      struct TX_IO_BUFFER_STRUCT *tx_next_packet;
    struct TX_IO_BUFFER_STRUCT *tx_next_buffer;
      UCHAR  tx_buffer_area[TX_MAX_BUFFER_SIZE];
} TX_IO_BUFFER;

TX_BLOCK_POOL tx_io_block_pool;

/* Create a pool of I/O buffers. Assume that the pointer
   "free_memory_ptr"points to an available memory area that
   is 64 KBytes in size. */

tx_block_pool_create(&tx_io_block_pool,
                  "Sample IO Driver Buffer Pool",
                  free_memory_ptr, 0x10000,
                  sizeof(TX_IO_BUFFER));
```

图 14. I/O 缓冲区

TX_IO_BUFFER

typedef TX_IO_BUFFER 包含两个指针。*tx_next_packet_ 指针用于在输入列表或输出列表中链接多个数据包。_tx_next_buffer* 指针用于将构成来自设备的单个数据包的缓冲区链接到一起。从池中分配缓冲区时，这两个指针都设置为 NULL。此外，某些设备可能需要另一个字段来指示实际包含数据的缓冲区大小。

缓冲 I/O 的优点

缓冲区 I/O 方案的优点是什么？ 最大的优点是不会在设备寄存器和应用程序的内存之间复制数据。相反，驱动

程序会为设备提供一系列缓冲区指针。物理设备 I/O 直接利用提供的缓冲区内存。

使用处理器复制信息的输入数据包或输出数据包会产生非常高的开销，在任何高吞吐量 I/O 情况下应避免使用这种方法。

缓冲 I/O 方法的另一个优点是输入列表和输出列表不会出现已满状况。所有可用的缓冲区在任何时候都可以位于任一列表中。这与本章前面介绍的简单字节循环缓冲区相反。每个缓冲区都具有固定大小，该大小在编译时确定。

缓冲驱动程序的职责

缓冲设备驱动程序只负责管理 I/O 缓冲区的链接列表。对于在应用程序软件准备就绪之前接收的数据包，会维护一个输入缓冲区列表。相反，对于发送速率比硬件设备的处理速度更快的数据包，会维护一个输出缓冲区列表。314 页上的图 15 显示了包含数据包以及构成每个数据包的缓冲区的输入列表和输出列表的简单链接。

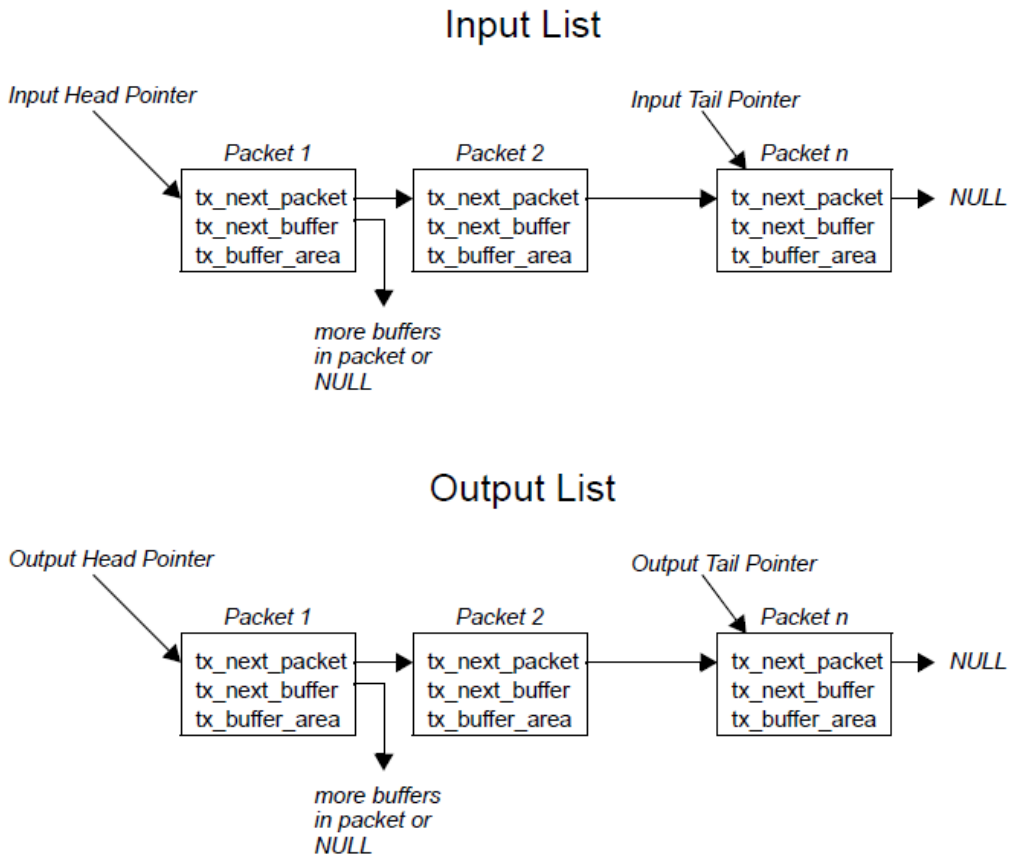


图 15. 输入列表-输出列表

应用程序使用相同的 I/O 缓冲区与缓冲驱动程序交互。传输时，应用程序软件会为驱动程序提供一个或多个缓冲区用于传输数据。当应用程序软件请求输入时，驱动程序将在 I/O 缓冲区中返回输入数据。

> **IMPORTANT**
>
> 在某些应用程序中，构建要求应用程序为驱动程序的输入缓冲区交换可用缓冲区的驱动程序输入接口可能会很有用。这可以减少驱动程序内部的一些缓冲区分配处理。

中断管理

在某些应用程序中，设备中断频率太高可能会禁止以 C 源代码格式编写 ISR，或者在每次中断时禁止与 ThreadX SMP 交互。例如，如果需要 25us 来保存并还原中断的上下文，则当中断频率为 50us 时，建议不要执行完整上下文保存。在这种情况下，可以使用小型汇编语言 ISR 来处理大多数设备中断。此低开销 ISR 仅在必要时才与

ThreadX SMP 交互。

在第 3 章末尾的中断管理讨论中可以找到类似的讨论。

> **NOTE**
>
> 如果要使用中断锁定来保护驱动程序的关键部分驱动程序的 ISR 代码，则必须始终在处理 ISR 的相同内核上执行线程级驱动程序代码。否则，应使用内部 ThreadX SMP 基元（如 TX_DISABLE 和 TX_RESTORE），这也是内置的核心保护。

线程挂起

在本章前面介绍的简单驱动程序示例中，输入服务的调用方在某个字符不可用时会暂停。在某些应用程序中，这可能是无法接受的。

例如，如果负责处理驱动程序输入的线程还承担其他职责，则只在驱动程序输入时暂停可能无法起作用。相反，需要对驱动程序进行自定义，使其以向线程发送其他处理请求的相似方式请求处理。

在大多数情况下，输入缓冲区放置在链接列表中，而输入事件消息会发送到线程的输入队列。

第 6 章 - Azure RTOS ThreadX SMP 的演示系统

2021/4/29 •

本章介绍所有 Azure RTOS ThreadX SMP 处理器支持包随附的演示系统。

概述

每个 ThreadX SMP 产品分发均包含一个演示系统，该演示系统可在所有受支持的微处理器上运行。

此示例系统在分发文件 demo_threadx.c 中定义，旨在说明如何在嵌入式多线程环境中使用 ThreadX SMP。演示包括初始化、八个线程、一个字节池、一个块池、一个队列、一个信号灯、一个互斥体和一个事件标志组。

> **IMPORTANT**
> 除线程的堆栈大小外，该演示应用程序在所有 ThreadX SMP 支持的处理器上均相同。

demo_threadx.c 的完整列表（包括在本章其余部分引用的行号）显示在第 324 页及其之后的页上。

应用程序定义

完成基本的 ThreadX SMP 初始化后，将执行 tx_application_define 函数。该函数负责设置所有初始系统资源，包括线程、队列、信号灯、互斥体、事件标志和内存池。

演示系统的 tx_application_define（第 60-164 行）按以下顺序创建演示对象：

```
byte_pool_0
thread_0
thread_1
thread_2
thread_3
thread_4
thread_5
thread_6
thread_7
queue_0
semaphore_0
event_flags_0
mutex_0
block_pool_0
```

该演示系统不会创建任何其他的附加 ThreadX SMP 对象。但是，实际应用程序可能会在运行时在执行线程内部创建系统对象。

初始执行

所有线程均使用 TX_AUTO_START 选项创建。因此，这些线程初步准备就绪，可以执行。tx_application_define 完成后，控制权将转移到线程计划程序，并从该处转移到各个线程。

线程执行顺序取决于它们的优先级和创建顺序。在演示系统中，thread_0 *首先执行，因为它具有最高优先级（它是使用优先级 1 创建的）。在 thread_0* 挂起后，将执行 thread_5，然后执行 thread_3、thread_4、thread_6、thread_7、thread_1，最后执行 thread_2 。

IMPORTANT

尽管 thread_3 和 thread_4 具有相同的优先级(都是使用优先级 8 创建的),但 thread_3 先执行。这是因为 thread_3 在 thread_4 之前创建并准备就绪。相同优先级的线程以 FIFO 方式执行。

线程 0

函数 thread_0_entry 标记线程的入口点(第 167-190 行)。Thread_0 是演示系统中要执行的第一个线程。其处理方式很简单:递增其计数器,休眠 10 个计时器时钟周期,设置一个事件标志以唤醒 thread_5,然后重复该序列。

Thread_0 是系统中优先级最高的线程。当它请求的休眠过期时,它会抢占演示中任何其他正在执行的线程。

线程 1

函数 thread_1_entry 标记线程的入口点(第 193-216 行)。Thread_1 是演示系统中要执行的倒数第二个线程。其处理包括:递增其计数器,将消息发送到 thread_2(通过 queue_0),然后重复该序列。请注意,只要 queue_0 已满,thread_1 就会挂起(第 207 行)。

线程 2

函数 thread_2_entry 标记线程的入口点(第 219-243 行)。Thread_2 是演示系统中要执行的最后一个线程。其处理包括:递增其计数器,从 thread_1(通过 queue_0)获取消息,然后重复该序列。请注意,只要 queue_0 为空,thread_2 就会挂起(第 233 行)。

虽然 thread_1 和 thread_2 在演示系统中都具有最低优先级(优先级 16),但它们也是大多数时候准备好执行的唯一线程。它们还是以时间分片方式创建的唯一线程(第 87 和 93 行)。在执行另一个线程之前,每个线程最多可执行 4 个计时器时钟周期。

线程 3 和 4

函数 thread_3_and_4_entry 标记 thread_3 和 thread_4 的入口点(第 246-280 行)。*这两个线程的优先级均为 8,这使它们成为演示系统中要执行的第三个和第四个线程。每个线程的处理是相同的:递增其计数器,获取 semaphore_0,*休眠 2 个计时器时钟周期,释放 semaphore_0 ,然后重复该序列。请注意,只要 semaphore_0 不可用,所有线程均会挂起(第 264 行)。

同时,两个线程在主处理中使用相同函数。这不会带来任何问题,因为它们都有自己唯一的堆栈,并且 C 天生是可重入函数。每个线程通过检查线程输入参数(第 258 行)来确定自己是哪一个线程,该参数可在创建线程时设置(第 102 行和第 109 行)。

IMPORTANT

在线程执行过程中获取当前线程点,并将其与控制块的地址进行比较,以确定线程标识,这一操作也很合理。

线程 5

函数 thread_5_entry 标记线程的入口点(第 283-305 行)。Thread_5 是演示系统中要执行的第二个线程。其处理包括:递增其计数器,从 thread_0 获取事件标志(通过 event_flags_0),然后重复该序列。请注意,只要 event_flags_0 中的事件标志不可用,thread_5 就会挂起(第 298 行)。

线程 6 和 7

函数 thread_6_and_7_entry 标记 thread_6 和 thread_7 的入口点(第 307-358 行)。*这两个线程的优先级均为 8,这使它们成为演示系统中要执行的第五个和第六个线程。每个线程的处理是相同的:递增其计数器,两次获取*

mutex_0，休眠 2 个计时器时钟周期，释放 mutex_0 两次，然后重复该序列。请注意，只要 mutex_0 不可用，所有线程均会挂起（第 325 行）。

同时，两个线程在主处理中使用相同函数。这不会带来任何问题，因为它们都有自己唯一的堆栈，并且 C 天生是可重入函数。每个线程通过检查线程输入参数（第 319 行）来确定自己是哪一个线程，该参数可在创建线程时设置（第 126 行和第 133 行）。

观看演示

每个演示线程均会递增自己的唯一计数器。可以检查以下计数器以核实演示的操作：

```
thread_0_counter
thread_1_counter
thread_2_counter
thread_3_counter
thread_4_counter
thread_5_counter
thread_6_counter
thread_7_counter
```

在演示执行过程中，每个计数器都应继续增加，其中，thread_1_counter 和 thread_2_counter 的增加速度最快。

分发文件：demo_threadx.c

本部分显示了 demo_threadx.c 的完整列表，包括本章中引用的行号。

```
000 /* This is a small demo of the high-performance ThreadX SMP kernel. It includes examples of eight
001 threads of different priorities, using a message queue, semaphore, mutex, event flags group,
002 byte pool, and block pool. */
003
004 #include "tx_api.h"
005
006 #define DEMO_STACK_SIZE         1024
007 #define DEMO_BYTE_POOL_SIZE     9120
008 #define DEMO_BLOCK_POOL_SIZE    100
009 #define DEMO_QUEUE_SIZE         100
010
011 /* Define the ThreadX SMP object control blocks...  */
012
013 TX_THREAD               thread_0;
014 TX_THREAD               thread_1;
015 TX_THREAD               thread_2;
016 TX_THREAD               thread_3;
017 TX_THREAD               thread_4;
018 TX_THREAD               thread_5;
019 TX_THREAD               thread_6;
020 TX_THREAD               thread_7;
021 TX_QUEUE                queue_0;
022 TX_SEMAPHORE            semaphore_0;
023 TX_MUTEX                mutex_0;
024 TX_EVENT_FLAGS_GROUP    event_flags_0;
025 TX_BYTE_POOL            byte_pool_0;
026 TX_BLOCK_POOL           block_pool_0;
027
028 /* Define the counters used in the demo application...  */
029
030 ULONG                    thread_0_counter;
031 ULONG                    thread_1_counter;
032 ULONG                    thread_1_messages_sent;
033 ULONG                    thread_2_counter;
034 ULONG                    thread_2_messages_received;
035 ULONG                    thread_3_counter;
036 ULONG                    thread_4_counter;
```

```
ULONG                   thread_5_counter;
ULONG                   thread_6_counter;
ULONG                   thread_7_counter;

/* Define thread prototypes.  */

void    thread_0_entry(ULONG thread_input);
void    thread_1_entry(ULONG thread_input);
void    thread_2_entry(ULONG thread_input);
void    thread_3_and_4_entry(ULONG thread_input);
void    thread_5_entry(ULONG thread_input);
void    thread_6_and_7_entry(ULONG thread_input);


/* Define main entry point.  */

int main()
{

    /* Enter the ThreadX SMP kernel. */
    tx_kernel_enter();
}

/* Define what the initial system looks like. */
void    tx_application_define(void *first_unused_memory)
{

CHAR    *pointer;

   /* Create a byte memory pool from which to allocate the thread stacks. */
   tx_byte_pool_create(&byte_pool_0, "byte pool 0", first_unused_memory,
                       DEMO_BYTE_POOL_SIZE);

   /* Put system definition stuff in here, e.g., thread creates and other assorted
      create information. */

   /* Allocate the stack for thread 0. */
   tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

   /* Create the main thread. */
   tx_thread_create(&thread_0, "thread 0", thread_0_entry, 0,
                       pointer, DEMO_STACK_SIZE,
                       1, 1, TX_NO_TIME_SLICE, TX_AUTO_START);

   /* Allocate the stack for thread 1. */
   tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

   /* Create threads 1 and 2. These threads pass information through a ThreadX SMP
      message queue. It is also interesting to note that these threads have a time
      slice. */
   tx_thread_create(&thread_1, "thread 1", thread_1_entry, 1,
                       pointer, DEMO_STACK_SIZE,
                       16, 16, 4, TX_AUTO_START);

   /* Allocate the stack for thread 2. */
   tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
   tx_thread_create(&thread_2, "thread 2", thread_2_entry, 2,
                       pointer, DEMO_STACK_SIZE,
                       16, 16, 4, TX_AUTO_START);

   /* Allocate the stack for thread 3. */
   tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

   /* Create threads 3 and 4. These threads compete for a ThreadX SMP counting semaphore.
      An interesting thing here is that both threads share the same instruction area. */
   tx_thread_create(&thread_3, "thread 3", thread_3_and_4_entry, 3,
                       pointer, DEMO_STACK_SIZE,
                       8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

```

```
106    /* Allocate the stack for thread 4. */
107    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
108
109    tx_thread_create(&thread_4, "thread 4", thread_3_and_4_entry, 4,
110                             pointer, DEMO_STACK_SIZE,
111                             8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);
112
113    /* Allocate the stack for thread 5. */
114    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
115
116    /* Create thread 5. This thread simply pends on an event flag, which will be set
117       by thread_0. */
118    tx_thread_create(&thread_5, "thread 5", thread_5_entry, 5,
119                             pointer, DEMO_STACK_SIZE,
120                             4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);
121
122    /* Allocate the stack for thread 6. */
123    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
124
125    /* Create threads 6 and 7. These threads compete for a ThreadX SMP mutex. */
126    tx_thread_create(&thread_6, "thread 6", thread_6_and_7_entry, 6,
127                             pointer, DEMO_STACK_SIZE,
128                             8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);
129
130    /* Allocate the stack for thread 7. */
131    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
132
133    tx_thread_create(&thread_7, "thread 7", thread_6_and_7_entry, 7,
134                             pointer, DEMO_STACK_SIZE,
135                             8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);
136
137    /* Allocate the message queue. */
138    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_QUEUE_SIZE*sizeof(ULONG), TX_NO_WAIT);
139
140    /* Create the message queue shared by threads 1 and 2. */
141    tx_queue_create(&queue_0, "queue 0", TX_1_ULONG, pointer, DEMO_QUEUE_SIZE*sizeof(ULONG));
142
143    /* Create the semaphore used by threads 3 and 4. */
144    tx_semaphore_create(&semaphore_0, "semaphore 0", 1);
145
146    /* Create the event flags group used by threads 1 and 5. */
147    tx_event_flags_create(&event_flags_0, "event flags 0");
148
149    /* Create the mutex used by thread 6 and 7 without priority inheritance. */
150    tx_mutex_create(&mutex_0, "mutex 0", TX_NO_INHERIT);
151
152    /* Allocate the memory for a small block pool. */
153    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_BLOCK_POOL_SIZE, TX_NO_WAIT);
154
155    /* Create a block memory pool to allocate a message buffer from. */
156    tx_block_pool_create(&block_pool_0, "block pool 0", sizeof(ULONG), pointer,
157                 DEMO_BLOCK_POOL_SIZE);
158
159    /* Allocate a block and release the block memory. */
160    tx_block_allocate(&block_pool_0, &pointer, TX_NO_WAIT);
161
162    /* Release the block back to the pool. */
163    tx_block_release(pointer);
164 }
165
166    /* Define the test threads. */
167    void thread_0_entry(ULONG thread_input)
168 {
169
170 UINT status;
171
172
173    /* This thread simply sits in while-forever-sleep loop. */
174     while(1)
```

```
175    {
176
177    /* Increment the thread counter. */
178    thread_0_counter++;
179
180    /* Sleep for 10 ticks. */
181    tx_thread_sleep(10);
182
183    /* Set event flag 0 to wakeup thread 5. */
184    status = tx_event_flags_set(&event_flags_0, 0x1, TX_OR);
185
186    /* Check status. */
187    if (status != TX_SUCCESS)
188       break;
189    }
190 }
191
192
193 void    thread_1_entry(ULONG thread_input)
194 {
195
196 UINT    status;
197
198
199    /* This thread simply sends messages to a queue shared by thread 2. */
200    while(1)
201    {
202
203        /* Increment the thread counter. */
204        thread_1_counter++;
205
206        /* Send message to queue 0. */
207        status = tx_queue_send(&queue_0, &thread_1_messages_sent, TX_WAIT_FOREVER);
208
209        /* Check completion status. */
210        if (status != TX_SUCCESS)
211           break;
212
213        /* Increment the message sent. */
214        thread_1_messages_sent++;
215    }
216 }
217
218
219 void     thread_2_entry(ULONG thread_input)
220 {
221
222 ULONG     received_message;
223 UINT      status;
224
225     /* This thread retrieves messages placed on the queue by thread 1. */
226     while(1)
227     {
228
229        /* Increment the thread counter. */
230        thread_2_counter++;
231
232        /* Retrieve a message from the queue. */
233        status = tx_queue_receive(&queue_0, &received_message, TX_WAIT_FOREVER);
234
235        /* Check completion status and make sure the message is what we
236           expected. */
237        if ((status != TX_SUCCESS) || (received_message != thread_2_messages_received))
238           break;
239
240        /* Otherwise, all is okay. Increment the received message count. */
241        thread_2_messages_received++;
242     }
243 }
```

```
243 }
244
245
246 void     thread_3_and_4_entry(ULONG thread_input)
247 {
248
249 UINT     status;
250
251
252     /* This function is executed from thread 3 and thread 4. As the loop
253        below shows, these function compete for ownership of semaphore_0. */
254     while(1)
255     {
256
257         /* Increment the thread counter. */
258         if (thread_input == 3)
259            thread_3_counter++;
260         else
261            thread_4_counter++;
262
263         /* Get the semaphore with suspension. */
264         status = tx_semaphore_get(&semaphore_0, TX_WAIT_FOREVER);
265
266         /* Check status. */
267         if (status != TX_SUCCESS)
268            break;
269
270         /* Sleep for 2 ticks to hold the semaphore. */
271         tx_thread_sleep(2);
272
273         /* Release the semaphore. */
274         status = tx_semaphore_put(&semaphore_0);
275
276         /* Check status. */
277         if (status != TX_SUCCESS)
278             break;
279     }
280 }
281
282
283 void     thread_5_entry(ULONG thread_input)
284 {
285
286 UINT     status;
287 ULONG    actual_flags;
288
289
290     /* This thread simply waits for an event in a forever loop. */
291     while(1)
292     {
293
294         /* Increment the thread counter. */
295         thread_5_counter++;
296
297         /* Wait for event flag 0. */
298         status = tx_event_flags_get(&event_flags_0, 0x1, TX_OR_CLEAR,
299                                     &actual_flags, TX_WAIT_FOREVER);
300
301         /* Check status. */
302         if ((status != TX_SUCCESS) || (actual_flags != 0x1))
303            break;
304     }
305 }
306
307 void     thread_6_and_7_entry(ULONG thread_input)
308 {
309
310 UINT     status;
311
312
```

```
312
313     /* This function is executed from thread 6 and thread 7. As the loop
314         below shows, these function compete for ownership of mutex_0. */
315     while(1)
316     {
317
318         /* Increment the thread counter. */
319         if (thread_input == 6)
320            thread_6_counter++;
321         else
322            thread_7_counter++;
323
324         /* Get the mutex with suspension. */
325         status = tx_mutex_get(&mutex_0, TX_WAIT_FOREVER);
326
327         /* Check status. */
328         if (status != TX_SUCCESS)
329            break;
330
331         /* Get the mutex again with suspension. This shows
332            that an owning thread may retrieve the mutex it
333            owns multiple times. */
334         status = tx_mutex_get(&mutex_0, TX_WAIT_FOREVER);
335
336         /* Check status. */
337         if (status != TX_SUCCESS)
338             break;
339
340         /* Sleep for 2 ticks to hold the mutex. */
341         tx_thread_sleep(2);
342
343         /* Release the mutex. */
344         status = tx_mutex_put(&mutex_0);
345
346         /* Check status. */
347         if (status != TX_SUCCESS)
348            break;
349
350         /* Release the mutex again. This will actually
351            release ownership since it was obtained twice. */
352         status = tx_mutex_put(&mutex_0);
353
354         /* Check status. */
355         if (status != TX_SUCCESS)
356            break;
357     }
358 }
```

附录 A - Azure RTO ThreadX SMP API 服务

2021/4/30 •

Entry 函数

```
VOID    tx_kernel_enter(VOID);
```

块内存服务

```
UINT    tx_block_allocate(TX_BLOCK_POOL *pool_ptr,
           VOID **block_ptr, ULONG wait_option);

UINT    tx_block_pool_create(TX_BLOCK_POOL *pool_ptr,
           CHAR *name_ptr, ULONG block_size,
           VOID *pool_start, ULONG pool_size);

UINT    tx_block_pool_delete(TX_BLOCK_POOL *pool_ptr);

UINT    tx_block_pool_info_get(TX_BLOCK_POOL
           *pool_ptr,
           CHAR **name,
           ULONG *available_blocks, ULONG
           *total_blocks,
           TX_THREAD **first_suspended,
           ULONG *suspended_count,
           TX_BLOCK_POOL **next_pool);

UINT
           tx_block_pool_performance_info_get(TX_BLOC
           K_POOL *pool_ptr,
           ULONG *allocates, ULONG *releases, ULONG
           *suspensions,
           ULONG *timeouts);

UINT
           tx_block_pool_performance_system_info_get(
           ULONG *allocates,
           ULONG *releases, ULONG *suspensions, ULONG
           *timeouts);

UINT        tx_block_pool_prioritize(TX_BLOCK_POOL
              *pool_ptr);

UINT        tx_block_release(VOID *block_ptr);
```

字节内存服务

```
UINT     tx_byte_allocate(TX_BYTE_POOL *pool_ptr,
            VOID **memory_ptr,
            ULONG memory_size, ULONG wait_option);

UINT     tx_byte_pool_create(TX_BYTE_POOL *pool_ptr,
            CHAR *name_ptr,
            VOID *pool_start, ULONG pool_size);

UINT     tx_byte_pool_delete(TX_BYTE_POOL *pool_ptr);

UINT     tx_byte_pool_info_get(TX_BYTE_POOL *pool_ptr,
            CHAR **name, ULONG *available_bytes,
            ULONG *fragments, TX_THREAD
            **first_suspended,
            ULONG *suspended_count,
            TX_BYTE_POOL **next_pool);

UINT     tx_byte_pool_performance_info_get(TX_BYTE_POOL
            *pool_ptr,
            ULONG *allocates,
            ULONG *releases, ULONG *fragments_searched,
            ULONG *merges,
            ULONG *splits, ULONG *suspensions, ULONG
            *timeouts);

UINT     tx_byte_pool_performance_system_info_get(ULONG
            *allocates,
            ULONG *releases, ULONG *fragments_searched,
            ULONG *merges, ULONG *splits, ULONG
            *suspensions, ULONG *timeouts);

UINT     tx_byte_pool_prioritize(TX_BYTE_POOL
            *pool_ptr);

UINT     tx_byte_release(VOID *memory_ptr);
```

事件标志服务

```
UINT     tx_event_flags_create(TX_EVENT_FLAGS_GROUP
            *group_ptr,
            CHAR *name_ptr);

UINT     tx_event_flags_delete(TX_EVENT_FLAGS_GROUP
            *group_ptr);

UINT     tx_event_flags_get(TX_EVENT_FLAGS_GROUP
            *group_ptr,
            ULONG requested_flags, UINT get_option,
            ULONG *actual_flags_ptr, ULONG
            wait_option);

UINT     tx_event_flags_info_get(TX_EVENT_FLAGS_GROUP
            *group_ptr,
            CHAR **name, ULONG *current_flags,
            TX_THREAD **first_suspended,
            ULONG *suspended_count,
            TX_EVENT_FLAGS_GROUP **next_group);

UINT
            tx_event_flags_performance_info_get(TX_EVE
            NT_FLAGS_GROUP *group_ptr, ULONG *sets,
            ULONG *gets, ULONG *suspensions,
            ULONG *timeouts);

UINT
            tx_event_flags_performance_system_info_get
            (ULONG *sets, ULONG *gets,
            ULONG *suspensions, ULONG *timeouts);

UINT     tx_event_flags_set(TX_EVENT_FLAGS_GROUP
            *group_ptr,
            ULONG flags_to_set, UINT set_option);

UINT     tx_event_flags_set_notify(TX_EVENT_FLAGS_GROUP
            *group_ptr,
            VOID
            (*events_set_notify)(TX_EVENT_FLAGS_GROUP
            *));
```

中断控制

```
UINT     tx_interrupt_control(UINT new_posture);
```

互斥服务

```
UINT     tx_mutex_create(TX_MUTEX *mutex_ptr, CHAR
            *name_ptr,
            UINT inherit);

UINT     tx_mutex_delete(TX_MUTEX *mutex_ptr);

UINT     tx_mutex_get(TX_MUTEX *mutex_ptr, ULONG
            wait_option);

UINT     tx_mutex_info_get(TX_MUTEX *mutex_ptr, CHAR
            **name,
            ULONG *count, TX_THREAD **owner,
            TX_THREAD **first_suspended,
            ULONG *suspended_count,
            TX_MUTEX **next_mutex);

UINT     tx_mutex_performance_info_get(TX_MUTEX
            *mutex_ptr, ULONG *puts, ULONG *gets, ULONG
            *suspensions, ULONG *timeouts,
            ULONG *inversions, ULONG *inheritances);

UINT     tx_mutex_performance_system_info_get(ULONG
            *puts, ULONG *gets,
            ULONG *suspensions, ULONG *timeouts, ULONG
            *inversions,
            ULONG *inheritances);

UINT     tx_mutex_prioritize(TX_MUTEX *mutex_ptr);

UINT     tx_mutex_put(TX_MUTEX *mutex_ptr);
```

队列服务

```
UINT     tx_queue_create(TX_QUEUE *queue_ptr, CHAR
            *name_ptr,
            UINT message_size, VOID *queue_start,
            ULONG queue_size);

UINT     tx_queue_delete(TX_QUEUE *queue_ptr);

UINT     tx_queue_flush(TX_QUEUE *queue_ptr);

UINT     tx_queue_front_send(TX_QUEUE *queue_ptr, VOID
            *source_ptr,
            ULONG wait_option);

UINT     tx_queue_info_get(TX_QUEUE *queue_ptr, CHAR
            **name,
            ULONG *enqueued, ULONG *available_storage,
            TX_THREAD **first_suspended,
            ULONG *suspended_count, TX_QUEUE
            **next_queue);

UINT     tx_queue_performance_info_get(TX_QUEUE
            *queue_ptr,
            ULONG *messages_sent, ULONG
            *messages_received,
            ULONG *empty_suspensions, ULONG
            *full_suspensions,
            ULONG *full_errors, ULONG *timeouts);

UINT     tx_queue_performance_system_info_get(ULONG
            *messages_sent,
            ULONG *messages_received, ULONG
            *empty_suspensions,
            ULONG *full_suspensions, ULONG
            *full_errors,
            ULONG *timeouts);

UINT     tx_queue_prioritize(TX_QUEUE *queue_ptr);

UINT     tx_queue_receive(TX_QUEUE *queue_ptr,
            VOID *destination_ptr, ULONG wait_option);

UINT     tx_queue_send(TX_QUEUE *queue_ptr, VOID
            *source_ptr,
            ULONG wait_option);

UINT     tx_queue_send_notify(TX_QUEUE *queue_ptr, VOID
            (*queue_send_notify)(TX_QUEUE *));
```

信号灯服务

```
UINT     tx_semaphore_ceiling_put(TX_SEMAPHORE
            *semaphore_ptr,
            ULONG ceiling);

UINT     tx_semaphore_create(TX_SEMAPHORE
            *semaphore_ptr,
            CHAR *name_ptr, ULONG initial_count);

UINT     tx_semaphore_delete(TX_SEMAPHORE
            *semaphore_ptr);

UINT     tx_semaphore_get(TX_SEMAPHORE *semaphore_ptr,
            ULONG wait_option);

UINT     tx_semaphore_info_get(TX_SEMAPHORE
            *semaphore_ptr, CHAR **name,
            ULONG *current_value,
            TX_THREAD **first_suspended,
            ULONG *suspended_count,
            TX_SEMAPHORE **next_semaphore);

UINT     tx_semaphore_performance_info_get(TX_SEMAPHORE
            *semaphore_ptr,
            ULONG *puts, ULONG *gets, ULONG
            *suspensions,
            ULONG *timeouts);

UINT     tx_semaphore_performance_system_info_get(ULONG
            *puts,
            ULONG *gets, ULONG *suspensions, ULONG
            *timeouts);

UINT     tx_semaphore_prioritize(TX_SEMAPHORE
            *semaphore_ptr);

UINT     tx_semaphore_put(TX_SEMAPHORE *semaphore_ptr);

UINT     tx_semaphore_put_notify(TX_SEMAPHORE
            *semaphore_ptr,
            VOID (*semaphore_put_notify)(TX_SEMAPHORE
            *));
```

线程控制服务

```
UINT     tx_thread_create(TX_THREAD *thread_ptr,
            CHAR *name_ptr,
            VOID (*entry_function)(ULONG), ULONG
            entry_input,
            VOID *stack_start, ULONG stack_size,
            UINT priority, UINT preempt_threshold,
            ULONG time_slice, UINT auto_start);

UINT     tx_thread_delete(TX_THREAD *thread_ptr);

UINT     tx_thread_entry_exit_notify(TX_THREAD
            *thread_ptr,
            VOID (*thread_entry_exit_notify)(TX_THREAD
            *, UINT));

TX_THREAD *tx_thread_identify(VOID);

UINT     tx_thread_info_get(TX_THREAD *thread_ptr, CHAR
            **name,
            UINT *state, ULONG *run_count, UINT
            *priority,
```

```
            UINT *preemption_threshold, ULONG
            *time_slice,
            TX_THREAD **next_thread,
            TX_THREAD **next_suspended_thread);

UINT     tx_thread_performance_info_get(TX_THREAD
            *thread_ptr,
            ULONG *resumptions, ULONG *suspensions,
            ULONG *solicited_preemptions,
            ULONG *interrupt_preemptions,
            ULONG *priority_inversions,ULONG
            *time_slices, ULONG *relinquishes, ULONG
            *timeouts,
            ULONG *wait_aborts, TX_THREAD
            **last_preempted_by);

UINT     tx_thread_performance_system_info_get(ULONG
            *resumptions,
            ULONG *suspensions,
            ULONG *solicited_preemptions,
            ULONG *interrupt_preemptions,
            ULONG *priority_inversions,ULONG
            *time_slices, ULONG *relinquishes, ULONG
            *timeouts,
            ULONG *wait_aborts, ULONG
            *non_idle_returns,
            ULONG *idle_returns);

UINT     tx_thread_preemption_change(TX_THREAD
            *thread_ptr,
            UINT new_threshold, UINT *old_threshold);

UINT     tx_thread_priority_change(TX_THREAD
            *thread_ptr,
            UINT new_priority, UINT *old_priority);

VOID     tx_thread_relinquish(VOID);

UINT     tx_thread_reset(TX_THREAD *thread_ptr);

UINT     tx_thread_resume(TX_THREAD *thread_ptr);

UINT     tx_thread_sleep(ULONG timer_ticks);

UINT     tx_thread_smp_core_exclude(TX_THREAD
            *thread_ptr,
            ULONG exclusion_map);

UINT     tx_thread_smp_core_exclude_get(TX_THREAD
            *thread_ptr,
            ULONG *exclusion_map_ptr);

UINT     tx_thread_smp_core_get(void);

UINT     tx_thread_stack_error_notify
            VOID(*stack_error_handler)(TX_THREAD *));

UINT     tx_thread_suspend(TX_THREAD *thread_ptr);

UINT     tx_thread_terminate(TX_THREAD *thread_ptr);

UINT     tx_thread_time_slice_change(TX_THREAD
            *thread_ptr,
            ULONG new_time_slice, ULONG
            *old_time_slice);

UINT     tx_thread_wait_abort(TX_THREAD *thread_ptr);
```

时间服务

```
ULONG     tx_time_get(VOID);
VOID      tx_time_set(ULONG new_time);
```

计时器服务

```
UINT     tx_timer_activate(TX_TIMER *timer_ptr);

UINT     tx_timer_change(TX_TIMER *timer_ptr,
            ULONG initial_ticks,
            ULONG reschedule_ticks);

UINT     tx_timer_create(TX_TIMER *timer_ptr,
            CHAR *name_ptr,
            VOID (*expiration_function)(ULONG),
            ULONG expiration_input, ULONG
            initial_ticks,
            ULONG reschedule_ticks, UINT
            auto_activate);

UINT     tx_timer_deactivate(TX_TIMER *timer_ptr);

UINT     tx_timer_delete(TX_TIMER *timer_ptr);

UINT     tx_timer_info_get(TX_TIMER *timer_ptr, CHAR
            **name,
            UINT *active, ULONG *remaining_ticks,
            ULONG *reschedule_ticks,
            TX_TIMER **next_timer);

UINT     tx_timer_performance_info_get(TX_TIMER
            *timer_ptr, ULONG *activates,
            ULONG *reactivates, ULONG *deactivates,
            ULONG *expirations,
            ULONG *expiration_adjusts);

UINT     tx_timer_performance_system_info_get
            ULONG *activates, ULONG *reactivates,
            ULONG *deactivates, ULONG *expirations,
            ULONG *expiration_adjusts);

UINT     tx_timer_smp_core_exclude(TX_TIMER *timer_ptr,
            ULONG exclusion_map);

UINT     tx_timer_smp_core_exclude_get(TX_TIMER
            *timer_ptr,
            ULONG *exclusion_map_ptr);
```

附录 B - Azure RTOS ThreadX SMP 常量

2021/4/29 •

字母顺序列表

```
TX_1_ULONG                  1
TX_2_ULONG                  2
TX_4_ULONG                  4
TX_8_ULONG                  8
TX_16_ULONG                 16
TX_ACTIVATE_ERROR           0x17
TX_AND                      2
TX_AND_CLEAR                3
TX_AUTO_ACTIVATE            1
TX_AUTO_START               1
TX_BLOCK_MEMORY             8
TX_BYTE_MEMORY              9
TX_CALLER_ERROR             0x13
TX_CEILING_EXCEEDED         0x21
TX_COMPLETED                1
TX_DELETE_ERROR             0x11
TX_DELETED                  0x01
TX_DONT_START               0
TX_EVENT_FLAG               7
TX_FALSE                    0
TX_FEATURE_NOT_ENABLED      0xFF
TX_FILE                     11
TX_GROUP_ERROR              0x06
TX_INHERIT                  1
TX_INHERIT_ERROR            0x1F
TX_INVALID_CEILING          0x22
TX_IO_DRIVER                10
TX_LOOP_FOREVER             1
TX_MUTEX_ERROR              0x1C
TX_MUTEX_SUSP               13
TX_NO_ACTIVATE              0
TX_NO_EVENTS                0x07
TX_NO_INHERIT               0
TX_NO_INSTANCE              0x0D
TX_NO_MEMORY                0x10
TX_NO_TIME_SLICE            0
TX_NO_WAIT                  0
TX_NOT_AVAILABLE            0x1D
TX_NOT_DONE                 0x20
TX_NOT_OWNED                0x1E
TX_NULL                     0
TX_OPTION_ERROR             0x08
TX_OR                       0
TX_OR_CLEAR                 1
TX_POOL_ERROR               0x02
TX_PRIORITY_ERROR           0x0F
TX_PTR_ERROR                0x03
TX_QUEUE_EMPTY              0x0A
TX_QUEUE_ERROR              0x09
TX_QUEUE_FULL               0x0B
TX_QUEUE_SUSP               5
TX_READY                    0
TX_RESUME_ERROR             0x12
TX_SEMAPHORE_ERROR          0x0C
TX_SEMAPHORE_SUSP           6
TX_SIZE_ERROR               0x05
TX_SLEEP                    4
```

```
TX_STACK_FILL
0xEFEFEFEFUL
TX_START_ERROR              0x10
TX_SUCCESS                  0x00
TX_SUSPEND_ERROR            0x14
TX_SUSPEND_LIFTED           0x19
TX_SUSPENDED                3
TX_TCP_IP                   12
TX_TERMINATED               2
TX_THREAD_ENTRY             0
TX_THREAD_ERROR             0x0E
TX_THREAD_EXIT              1
TX_THRESH_ERROR             0x18
TX_TICK_ERROR               0x16
TX_TIMER_ERROR              0x15
TX_TRUE                     1
TX_WAIT_ABORT_ERROR         0x1B
TX_WAIT_ABORTED             0x1A
TX_WAIT_ERROR               0x04
TX_WAIT_FOREVER
0xFFFFFFFFUL
```

按值列出

```
TX_DONT_START               0
TX_FALSE                    0
TX_NO_ACTIVATE              0
TX_NO_INHERIT               0
TX_NO_TIME_SLICE            0
TX_NO_WAIT                  0
TX_NULL                     0
TX_OR                       0
TX_READY                    0
TX_SUCCESS                  0x00
TX_THREAD_ENTRY             0
TX_1_ULONG                  1
TX_AUTO_ACTIVATE            1
TX_AUTO_START               1
TX_COMPLETED                1
TX_INHERIT                  1
TX_LOOP_FOREVER             1
TX_DELETED                  0x01
TX_OR_CLEAR                 1
TX_THREAD_EXIT              1
TX_TRUE                     1
TX_2_ULONG                  2
TX_AND                      2
TX_POOL_ERROR               0x02
TX_TERMINATED               2
TX_AND_CLEAR                3
TX_PTR_ERROR                0x03
TX_SUSPENDED                3
TX_4_ULONG                  4
TX_SLEEP                    4
TX_WAIT_ERROR               0x04
TX_QUEUE_SUSP               5
TX_SIZE_ERROR               0x05
TX_GROUP_ERROR              0x06
TX_SEMAPHORE_SUSP           6
TX_EVENT_FLAG               7
TX_NO_EVENTS                0x07
TX_8_ULONG                  8
TX_BLOCK_MEMORY             8
TX_OPTION_ERROR             0x08
TX_BYTE_MEMORY              9
TX_QUEUE_ERROR              0x09
```

```
TX_IO_DRIVER                10
TX_QUEUE_EMPTY              0x0A
TX_FILE                     11
TX_QUEUE_FULL               0x0B
TX_TCP_IP                   12
TX_SEMAPHORE_ERROR          0x0C
TX_MUTEX_SUSP               13
TX_NO_INSTANCE              0x0D
TX_THREAD_ERROR             0x0E
TX_PRIORITY_ERROR           0x0F
TX_16_ULONG                 16
TX_NO_MEMORY                0x10
TX_START_ERROR              0x10
TX_DELETE_ERROR             0x11
TX_RESUME_ERROR             0x12
TX_CALLER_ERROR             0x13
TX_SUSPEND_ERROR            0x14
TX_TIMER_ERROR              0x15
TX_TICK_ERROR               0x16
TX_ACTIVATE_ERROR           0x17
TX_THRESH_ERROR             0x18
TX_SUSPEND_LIFTED           0x19
TX_WAIT_ABORTED             0x1A
TX_WAIT_ABORT_ERROR         0x1B
TX_MUTEX_ERROR              0x1C
TX_NOT_AVAILABLE            0x1D
TX_NOT_OWNED                0x1E
TX_INHERIT_ERROR            0x1F
TX_NOT_DONE                 0x20
TX_CEILING_EXCEEDED         0x21
TX_INVALID_CEILING          0x22
TX_FEATURE_NOT_ENABLED      0xFF
TX_STACK_FILL
0xEFEFEFEFUL
TX_WAIT_FOREVER
0xFFFFFFFFUL
```

附录 C - Azure RTOS ThreadX SMP 数据类型

2021/4/30 •

TX_BLOCK_POOL

```
typedef struct TX_BLOCK_POOL_STRUCT
{
    ULONG tx_block_pool_id;
    CHAR *tx_block_pool_name;
    ULONG tx_block_pool_available;
    ULONG tx_block_pool_total;
    UCHAR *tx_block_pool_available_list;
    UCHAR *tx_block_pool_start;
    ULONG tx_block_pool_size;
    ULONG tx_block_pool_block_size;
    struct TX_THREAD_STRUCT

*tx_block_pool_suspension_list;
    ULONG tx_block_pool_suspended_count;
    struct TX_BLOCK_POOL_STRUCT

*tx_block_pool_created_next,

*tx_block_pool_created_previous;

#ifdef TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO
    ULONG
tx_block_pool_performance_allocate_count;
    ULONG
tx_block_pool_performance_release_count;
    ULONG
tx_block_pool_performance_suspension_count;
    ULONG
tx_block_pool_performance_timeout_count;
#endif

    TX_BLOCK_POOL_EXTENSION /* Port defined */

} TX_BLOCK_POOL;
```

TX_BYTE_POOL

```
typedef struct TX_BYTE_POOL_STRUCT
{
    ULONG tx_byte_pool_id;
    CHAR *tx_byte_pool_name;
    ULONG tx_byte_pool_available;
    ULONG tx_byte_pool_fragments;
    UCHAR *tx_byte_pool_list;
    UCHAR *tx_byte_pool_search;
    UCHAR *tx_byte_pool_start;
    ULONG tx_byte_pool_size;
    struct TX_THREAD_STRUCT
                      *tx_byte_pool_owner;
    struct TX_THREAD_STRUCT

*tx_byte_pool_suspension_list;
    ULONG
tx_byte_pool_suspended_count;
    struct TX_BYTE_POOL_STRUCT

*tx_byte_pool_created_next,

*tx_byte_pool_created_previous;

#ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO
    ULONG
tx_byte_pool_performance_allocate_count;
    ULONG
tx_byte_pool_performance_release_count;
    ULONG tx_byte_pool_performance_merge_count;
    ULONG tx_byte_pool_performance_split_count;
    ULONG tx_byte_pool_performance_search_count;
    ULONG
tx_byte_pool_performance_suspension_count;
    ULONG
tx_byte_pool_performance_timeout_count;
#endif

    TX_BYTE_POOL_EXTENSION /* Port defined */
} TX_BYTE_POOL;
```

TX_EVENT_FLAGS_GROUP

```
typedef struct TX_EVENT_FLAGS_GROUP_STRUCT
{
    ULONG tx_event_flags_group_id;
    CHAR *tx_event_flags_group_name;
    ULONG tx_event_flags_group_current;
    UINT tx_event_flags_group_reset_search;
    struct TX_THREAD_STRUCT

*tx_event_flags_group_suspension_list;
    ULONG
tx_event_flags_group_suspended_count;
    struct TX_EVENT_FLAGS_GROUP_STRUCT
*tx_event_flags_group_created_next,

*tx_event_flags_group_created_previous;
    ULONG
tx_event_flags_group_delayed_clear;

#ifdef TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO
    ULONG
tx_event_flags_group_performance_set_count;
    ULONG
tx_event_flags_group__performance_get_count;
    ULONG
tx_event_flags_group___performance_suspension_co
unt;
    ULONG
tx_event_flags_group____performance_timeout_coun
t;
#endif

#ifndef TX_DISABLE_NOTIFY_CALLBACKS

    VOID
(*tx_event_flags_group_set_notify)(struct
TX_EVENT_FLAGS_GROUP_STRUCT *);
#endif

    TX_EVENT_FLAGS_GROUP_EXTENSION /* Port
defined  */
} TX_EVENT_FLAGS_GROUP;
```

TX_MUTEX

```
typedef struct TX_MUTEX_STRUCT
{
    ULONG tx_mutex_id;
    CHAR *tx_mutex_name;
    ULONG tx_mutex_ownership_count;
    TX_THREAD *tx_mutex_owner;
    UINT tx_mutex_inherit;
    UINT tx_mutex_original_priority;
    struct TX_THREAD_STRUCT
                      *tx_mutex_suspension_list;
    ULONG tx_mutex_suspended_count;
    struct TX_MUTEX_STRUCT
                      *tx_mutex_created_next,

*tx_mutex_created_previous;
    ULONG tx_mutex_highest_priority_waiting;
    struct TX_MUTEX_STRUCT
                      *tx_mutex_owned_next,
                      *tx_mutex_owned_previous;

#ifdef TX_MUTEX_ENABLE_PERFORMANCE_INFO
    ULONG tx_mutex_performance_put_count;
    ULONG tx_mutex_performance_get_count;
    ULONG tx_mutex_performance_suspension_count;
    ULONG tx_mutex_performance_timeout_count;
    ULONG
tx_mutex_performance_priority_inversion_count;
    ULONG
tx_mutex_performance__priority_inheritance_count
;
#endif

    TX_MUTEX_EXTENSION /* Port defined */

} TX_MUTEX;
```

TX_QUEUE

```
typedef struct TX_QUEUE_STRUCT
{
    ULONG tx_queue_id;
    CHAR *tx_queue_name;
    UINT tx_queue_message_size;
    ULONG tx_queue_capacity;
    ULONG tx_queue_enqueued;
    ULONG tx_queue_available_storage;
    ULONG *tx_queue_start;
    ULONG *tx_queue_end;
    ULONG *tx_queue_read;
    ULONG *tx_queue_write;
    struct TX_THREAD_STRUCT
                      *tx_queue_suspension_list;
    ULONG tx_queue_suspended_count;
    struct TX_QUEUE_STRUCT
                      *tx_queue_created_next,

*tx_queue_created_previous;

#ifdef TX_QUEUE_ENABLE_PERFORMANCE_INFO
    ULONG
tx_queue_performance_messages_sent_count;
    ULONG
tx_queue_performance_messages_received_count;
    ULONG
tx_queue_performance_empty_suspension_count;
    ULONG
tx_queue_performance_full_suspension_count;
    ULONG tx_queue_performance_full_error_count;
    ULONG tx_queue_performance_timeout_count;
#endif

#ifndef TX_DISABLE_NOTIFY_CALLBACKS
    VOID *tx_queue_send_notify)(struct
TX_QUEUE_STRUCT *);
#endif

    TX_QUEUE_EXTENSION /* Port defined  */

} TX_QUEUE;
```

TX_SEMAPHORE

```
typedef struct TX_SEMAPHORE_STRUCT
{
    ULONG tx_semaphore_id;
    CHAR *tx_semaphore_name;
    ULONG tx_semaphore_count;
    struct TX_THREAD_STRUCT

*tx_semaphore_suspension_list;
    ULONG tx_semaphore_suspended_count;
    struct TX_SEMAPHORE_STRUCT

*tx_semaphore_created_next,

*tx_semaphore_created_previous;

#ifdef TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO
    ULONG tx_semaphore_performance_put_count;
    ULONG tx_semaphore_performance_get_count;
    ULONG
tx_semaphore_performance_suspension_count;
    ULONG
tx_semaphore_performance_timeout_count;
#endif

#ifndef TX_DISABLE_NOTIFY_CALLBACKS
    VOID (*tx_semaphore_put_notify)(struct
TX_SEMAPHORE_STRUCT *);
#endif

    TX_SEMAPHORE_EXTENSION /* Port defined  */

} TX_SEMAPHORE;
```

TX_THREAD

```
typedef struct TX_THREAD_STRUCT
{
    ULONG tx_thread_id;
    ULONG tx_thread_run_count;
    VOID *tx_thread_stack_ptr;
    VOID *tx_thread_stack_start;
    VOID *tx_thread_stack_end;
    ULONG tx_thread_stack_size;
    ULONG tx_thread_time_slice;
    ULONG tx_thread_new_time_slice;
    struct TX_THREAD_STRUCT
                      *tx_thread_ready_next,
                      *tx_thread_ready_previous;

    TX_THREAD_EXTENSION_0  /* Port defined  */

    CHAR *tx_thread_name;
    UINT tx_thread_priority;
    UINT tx_thread_state;
    UINT tx_thread_delayed_suspend;
    UINT tx_thread_suspending;
    UINT tx_thread_preempt_threshold;
    VOID (*tx_thread_schedule_hook)(struct
TX_THREAD_STRUCT *, ULONG);
    VOID (*tx_thread_entry)(ULONG);
    ULONG tx_thread_entry_parameter;
    TX_TIMER_INTERNAL tx_thread_timer;
    VOID (*tx_thread_suspend_cleanup)(struct
TX_THREAD_STRUCT *);
    VOID *tx_thread_suspend_control_block;
```

```
    struct TX_THREAD_STRUCT
                        *tx_thread_suspended_next,

*tx_thread_suspended_previous;
    ULONG tx_thread_suspend_info;
    VOID *tx_thread_additional_suspend_info;
    UINT tx_thread_suspend_option;
    UINT tx_thread_suspend_status;

    TX_THREAD_EXTENSION_1 /* Port defined  */

    struct TX_THREAD_STRUCT
                        *tx_thread_created_next,

*tx_thread_created_previous;
    UINT tx_thread_smp_core_mapped;
    ULONG tx_thread_smp_core_control;
    UINT tx_thread_smp_core_executing;

    TX_THREAD_EXTENSION_2 /* Port defined  */

    ULONG tx_thread_smp_cores_excluded;
    ULONG tx_thread_smp_cores_allowed;

    VOID *tx_thread_filex_ptr;

    UINT tx_thread_user_priority;
    UINT tx_thread_user_preempt_threshold;
    UINT tx_thread_inherit_priority;
    ULONG tx_thread_owned_mutex_count;
    struct TX_MUTEX_STRUCT
*tx_thread_owned_mutex_list;

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO
    ULONG tx_thread_performance_resume_count;
    ULONG tx_thread_performance_suspend_count;
    ULONG
tx_thread_performance_solicited_preemption_count
;
    ULONG
tx_thread_performance_interrupt_preemption_count
;
    ULONG
tx_thread_performance_priority_inversion_count;
    struct TX_THREAD_STRUCT

*tx_thread_performance_last_preempting_thread;
    ULONG
tx_thread_performance_time_slice_count;
    ULONG
tx_thread_performance_relinquish_count;
    ULONG tx_thread_performance_timeout_count;
    ULONG
tx_thread_performance_wait_abort_count;
#endif
    VOID *tx_thread_stack_highest_ptr;
#ifndef TX_DISABLE_NOTIFY_CALLBACKS
    VOID (*tx_thread_entry_exit_notify)
                        (struct TX_THREAD_STRUCT
*, UINT);
#endif

    TX_THREAD_EXTENSION_3 /* Port defined  */
    ULONG tx_thread_suspension_sequence;

    TX_THREAD_USER_EXTENSION

} TX_THREAD;
```

TX_TIMER

```
typedef struct TX_TIMER_STRUCT
{
    ULONG tx_timer_id;
    CHAR *tx_timer_name;
    TX_TIMER_INTERNAL tx_timer_internal;
    struct TX_TIMER_STRUCT
                      *tx_timer_created_next,

*tx_timer_created_previous;

    TX_TIMER_EXTENSION  /* Port defined  */

#ifdef TX_TIMER_ENABLE_PERFORMANCE_INFO
    ULONG tx_timer_performance_activate_count;
    ULONG tx_timer_performance_reactivate_count;
    ULONG tx_timer_performance_deactivate_count;
    ULONG tx_timer_performance_expiration_count;
    ULONG
tx_timer_performance__expiration_adjust_count;
#endif

} TX_TIMER;
```

TX_TIMER_INTERNAL

```
typedef struct TX_TIMER_INTERNAL_STRUCT
{
    ULONG tx_timer_internal_remaining_ticks;
    ULONG tx_timer_internal_re_initialize_ticks;
    VOID
(*tx_timer_internal_timeout_function)(ULONG);
    ULONG tx_timer_internal_timeout_param;
    struct TX_TIMER_INTERNAL_STRUCT
    *tx_timer_internal_active_next,

*tx_timer_internal_active_previous;
    struct TX_TIMER_INTERNAL_STRUCT

*tx_timer_internal_list_head;
    ULONG tx_timer_internal_smp_cores_excluded
    TX_TIMER_INTERNAL_EXTENSION /* Port defined
*/

} TX_TIMER_INTERNAL;
```

附录 D - Azure RTOS ASCII 字符代码

2021/4/30 •

十六进制 ASCII 字符代码

least significant nibble \ most significant nibble	0_	1_	2_	3_	4_	5_	6_	7_
_0	NUL	DLE	SP	0	@	P	'	p
_1	SOH	DC1	!	1	A	Q	a	q
_2	STX	DC2	"	2	B	R	b	r
_3	ETX	DC3	#	3	C	S	c	s
_4	EOT	DC4	$	4	D	T	d	t
_5	ENQ	NAK	%	5	E	U	e	u
_6	ACK	SYN	&	6	F	V	f	v
_7	BEL	ETB	'	7	G	W	g	w
_8	BS	CAN	(	8	H	X	h	x
_9	HT	EM	)	9	I	Y	i	y
_A	LF	SUB	*	:	J	Z	j	z
_B	VT	ESC	+	;	K	[	K	}
_C	FF	FS	,	<	L	\	l	|
_D	CR	GS	-	=	M	]	m	}
_E	SO	RS	.	>	N	^	n	~
_F	SI	US	/	?	O	_	o	DEL