AUTOSAR OS

实战指南

原理 · 配置 · 嵌入式实战

基于 Infineon AURIX TC334 平台

吴延辉

序

嵌入式实时操作系统从来都不是一个轻松的话题。它藏在每一辆汽车的 ECU 里，默默调度着刹车、转向、引擎的控制任务，却鲜少有人真正愿意走近它、读懂它。对于大多数嵌入式工程师而言，AUTOSAR 是一个令人望而生畏的名字——厚重的规范文档、晦涩的术语、昂贵的商业工具链，像一道高墙将无数学习者挡在门外。

我自己也站在那堵墙的前面。

从 FreeRTOS 转向 AUTOSAR，面对 SWS_OS 规范中几百页的英文描述，面对 OSEK/OSEKtime/COM 各种子规范的纠缠，面对一个简单的任务激活竟然需要理解符合性类别、可扩展性等级、调度策略这么多前置概念——我感到深深的迷茫。市面上能找到的中文资料少之又少，大部分是学术论文式的概述，缺乏工程视角的解读；大部分书籍往往停留在规范层面，难以与实际的硬件、编译器、调试器建立起联系。

幸运的是，这个时代给了我一个前辈们不曾拥有的武器——人工智能。当我卡在某个概念上无法理解时，AI 可以帮我快速梳理规范的逻辑脉络；当我面对一段晦涩的内核代码时，AI 能逐行解释其背后的设计意图；当我需要验证一个猜想时，AI 可以帮我构建测试用例、分析可能的边界情况。它不是替代我去思考，而是像一个随时在线的导师，陪我一起啃那些原本可能让我放弃的硬骨头。可以说，如果没有 AI 的辅助，这本书应该不会诞生。

这本书不是对 AUTOSAR 规范的翻译，也不是对某个商业 OS 产品的说明。它是一个工程师写给工程师的实战手册。

需要说明的是， AUTOSAR OS 的开发通常依赖专业的配置工具（如 Vector DaVinci、EB tresos 等），这些工具通过图形化界面自动生成配置代码和 ARXML 文件。但工具本身的学习成本不低，且容易让初学者"知其然而不知其所以然"——点了几个选项，代码就生成了，却并不真正理解背后的原理。因此，本书选择了另一条路：所有 OS 配置均通过手写代码完成。从任务定义、中断注册到告警配置，每一步都亲手敲出来。这不是否定工具的价值，而是希望通过"裸手"的方式，让读者真正看清 AUTOSAR OS 的骨架和脉络。等你理解了每一个配置项的含义和作用，再回到工具中去，会发现一切都变得清晰而自然。

每一章都遵循同样的节奏：先讲清原理，再手写配置，然后剖析 API，最后在真实硬件上验证。我希望读者读到的不是抽象的概念，而是可编译、可运行、可触摸的代码和实验结果。

选择 Infineon AURIX TC334 作为配套硬件平台，说起来并没有什么宏大的理由——手头刚好有一块 TC334 Lite Kit 的 Demo 板。但缘分有时候就是这样，既然有了板子，就干脆从这块板子出发，把 AUTOSAR OS 的每一个模块都在它上面跑通、验证、记录下来。TC334 搭载的 TriCore TC1.6P 内核是当前汽车电子领域应用最广泛的架构之一，Lite Kit 价格也足够亲民，任何读者都可以轻松搭建自己的实验环境。

全书从 AUTOSAR OS 的基本概念出发，逐步深入到中断管理、任务调度、资源保护、事件机制，再到多核 OS、时序保护、低功耗策略等高级主题，最终以实战案例和性能优化收尾。这条路线图正是我自己走过的学习路径。如果你有 FreeRTOS 的基础，你会发现书中随处可见与 FreeRTOS 的对比，帮助你快速建立知识的映射关系。

感谢 AUTOSAR 组织建立了如此优秀的软件架构标准。感谢 Infineon 提供了高质量的 iLLD 驱动库和高性价比的开发板。感谢 i-soft 团队开源了 AUTOSAR OS 内核，让学习和研究成为可能。感谢每一位在技术社区中无私分享知识的同行者。

愿每一位翻开这本书的读者，都能在 AUTOSAR OS 的世界里找到属于自己的路。

吴延辉 2026 年 5 月 15 日凌晨，于上海。

目录

序 .... *A*
目录 .... *B*
本书目的 .... - 1 -
目标读者 .... - 1 -
本书结构 .... - 1 -
阅读建议 .... - 2 -
四段式结构说明 .... - 2 -
平台说明 .... - 2 -
OS 版本 .... - 3 -
术语表 .... - 3 -
第一章 AUTOSAR OS 简介 .... *- 5 -*
1.1 初识 AUTOSAR OS .... - 6 -
1.2 学习资源与参考资料 .... - 8 -
1.3 AUTOSAR OS 源码与配置初探 .... - 10 -
1.4 本章小结 .... - 16 -
第二章 AUTOSAR OS 移植与集成 .... *- 17 -*
2.1 准备工作 .... - 18 -
2.2 OS 集成步骤 .... - 19 -
2.3 集成验证 .... - 27 -
2.4 本章小结 .... - 31 -
第三章 AUTOSAR OS 系统配置 .... *- 32 -*
3.1 OS 配置概述（OsGeneralConfiguration） .... - 33 -
3.2 可扩展性类别（Scalability Class 1/2/3/4） .... - 33 -
3.3 符合性类别（BCC1/BCC2/ECC1/ECC2） .... - 34 -
3.4 OS 配置参数详解 .... - 36 -
3.5 本章小结 .... - 42 -
第四章 中断管理 .... *- 43 -*
4.1 AUTOSAR OS 中断模型 .... - 44 -
4.2 中断优先级与嵌套 .... - 45 -
4.3 开关中断 API .... - 47 -

4.4 临界区保护 - 50 -
4.5 中断配置实验 - 51 -
4.6 本章小结 - 54 -
第五章 任务基础 - 55 -
5.1 AUTOSAR OS 任务模型概述 - 56 -
5.2 基本任务（Basic Task）与扩展任务（Extended Task） - 56 -
5.3 任务状态模型 - 58 -
5.4 任务优先级与抢占 - 59 -
5.5 任务实现范式（Task 函数体结构） - 60 -
5.6 任务堆栈与上下文保存 - 61 -
5.7 任务自动启动（AUTOSTART） - 62 -
5.8 本章小结 - 64 -
第六章 任务 API 详解 - 65 -
6.1 任务管理 API 概览 - 66 -
6.2 任务激活与终止实验 - 70 -
6.3 任务链式激活实验 - 72 -
6.4 Task API 速查表 - 74 -
6.5 本章小结 - 75 -
第七章 调度器与调度表 - 76 -
7.1 调度器工作原理 - 77 -
7.2 调度表概念 - 80 -
7.3 调度表 API - 81 -
7.4 调度表实验 - 85 -
7.5 本章小结 - 89 -
第八章 计数器与报警 - 90 -
8.1 Counter 概念 - 91 -
8.2 Alarm 概念 - 92 -
8.3 Alarm API - 94 -
8.4 Alarm 实验 - 98 -
8.5 本章小结 - 102 -
第九章 资源管理 - 103 -
9.1 资源与优先级天花板协议（PCP） - 104 -

9.2 标准资源与内部资源 - 106 -
9.3 Resource API - 107 -
9.4 优先级反转问题与解决 - 109 -
9.5 资源管理实验 - 113 -
9.6 本章小结 - 115 -
第十章 事件机制 - 116 -
10.1 Event 概念（仅用于 Extended Task） - 117 -
10.2 Event API - 118 -
10.3 Event 与 Task 状态转换 - 121 -
10.4 事件机制实验 - 122 -
10.5 本章小结 - 125 -
第十一章 OS 应用与内存保护 - 126 -
11.1 OS-Application 概念 - 127 -
11.2 内存保护（Memory Protection） - 128 -
11.3 服务保护与时间保护 - 130 -
11.4 Protection Hook - 132 -
11.5 本章小结 - 136 -
第十二章 Hook 函数 - 137 -
12.1 Hook 函数概述 - 138 -
12.2 系统级 Hook - 138 -
12.3 任务级 Hook - 142 -
12.4 Hook 函数实验 - 144 -
12.5 本章小结 - 147 -
第十三章 多核 OS - 148 -
13.1 AUTOSAR 多核 OS 概述 - 149 -
13.2 核间通信机制 - 149 -
13.3 多核启动流程 - 153 -
13.4 跨核任务激活与事件设置 - 155 -
13.5 多核实验 - 157 -
13.6 从单核迁移到多核的实战指南 - 159 -
13.7 本章小结 - 163 -
第十四章 时间管理 - 164 -

14.1 OS 系统 Tick 与时钟源 - 165 -
14.2 GetCounterValue / GetElapsedValue（获取计数器值/已经过值） - 167 -
14.3 时间保护机制详解 - 168 -
14.4 时间管理实验 - 170 -
14.5 本章小结 - 173 -
第十五章 低功耗 OS - 174 -
15.1 MCU 低功耗模式概述 - 175 -
15.2 IdleTask 与后台循环中的低功耗处理 - 176 -
15.3 OS 与 EcuM 睡眠集成 - 178 -
15.4 低功耗实验 - 180 -
15.5 本章小结 - 183 -
第十六章 错误处理与调试 - 184 -
16.1 OS 错误码一览 - 185 -
16.2 ErrorHook 与错误日志 - 186 -
16.3 栈溢出检测 - 188 -
16.4 OS Trace 与运行时分析 - 189 -
16.5 常见问题排查 - 191 -
16.6 本章小结 - 194 -
第十七章 实战案例与工程模板 - 195 -
17.1 周期任务模板（10ms/50ms/100ms 周期任务） - 196 -
17.2 状态机任务模板（Event 驱动） - 200 -
17.3 生产者-消费者模式（中断采集→事件通知→任务处理） - 201 -
17.4 多优先级系统示例（安全关键/实时控制/后台诊断三层） - 203 -
17.5 从零搭建工程（基于 TC334 LiteKit） - 204 -
17.6 CAN/LIN 通信集成示例 - 205 -
17.7 本章小结 - 207 -
第十八章 性能优化与最佳实践 - 208 -
18.1 任务划分策略 - 209 -
18.2 中断响应优化 - 212 -
18.3 栈空间优化 - 215 -
18.4 CPU 负载分析 - 219 -
18.5 资源争用优化 - 224 -

18.6 内存布局优化（TC334 特有） - 227 -
18.7 常见陷阱与规避 - 230 -
18.8 本章小结 - 236 -
附录 A：AUTOSAR OS 全部 API 速查表 - 238 -
A.1 任务管理（7 个 API） - 238 -
A.2 事件管理（6 个 API） - 238 -
A.3 告警管理（5 个 API） - 239 -
A.4 计数器管理（3 个 API） - 239 -
A.5 资源管理（2 个 API） - 240 -
A.6 中断管理（9 个 API） - 240 -
A.7 调度表管理（6 个 API） - 241 -
A.8 系统管理（4 个 API） - 242 -
A.9 多核管理（5 个 API） - 243 -
A.10 自旋锁（3 个 API） - 244 -
A.11 Hook 函数（6 个） - 244 -
附录 B：常用 ARXML 配置模板 - 245 -
B.1 Task 配置模板 - 245 -
B.2 Alarm 配置模板 - 246 -
B.3 Counter 配置模板 - 246 -
B.4 Schedule Table 配置模板 - 247 -
B.5 ISR 配置模板 - 247 -
B.6 Resource 配置模板 - 248 -
B.7 Event 配置模板 - 248 -
附录 C：各主流工具链对比 - 248 -
C.1 工具链概览 - 248 -
C.2 典型使用流程对比 - 248 -
C.3 选型建议 - 249 -
附录 D：FreeRTOS 与 AUTOSAR OS 概念映射总结表 - 249 -
D.1 关键差异总结 - 251 -
D.2 常见迁移场景代码示例 - 251 -
D.3 迁移检查清单 - 254 -
附录 E：API 调用上下文兼容性完整矩阵 - 254 -

E.1 任务/事件/资源 API - 255 -
E.2 告警/计数器/调度表 API - 255 -
E.3 系统/多核/Spinlock API - 255 -
E.4 中断管理 API - 256 -

本书目的

本手册为正在从 FreeRTOS 过渡到 AUTOSAR OS 的嵌入式开发工程师提供系统性的学习指导。通过理论讲解、配置演示、API 详解和实验验证四个维度，帮助读者快速掌握 AUTOSAR OS 的核心概念与工程实践。

本书基于 Infineon TC334 Lite Kit 硬件平台和小满/iSoft AUTOSAR OS（LGPL 开源版本）内核，以实际可编译运行的工程代码为基础，确保所有示例均可在真实硬件上验证。

目标读者

- 具有 FreeRTOS 或其他 RTOS 使用经验的嵌入式 C 开发者
- 正在进行 AUTOSAR CP 平台开发的系统工程师
- 希望了解 AUTOSAR OS 调度机制和资源管理的软件架构师

阅读本书前，建议读者具备以下基础知识：

  - C 语言编程（熟悉指针、结构体、宏定义）
  - 嵌入式系统基本概念（中断、定时器、寄存器操作）
  - 至少一种 RTOS 的使用经验（FreeRTOS / μC/OS / RT-Thread 等）

本书结构

全书共 18 章 + 附录，按照由浅入深的逻辑组织为以下六个部分：

基础篇（第 1-3 章）

- 第 1 章 AUTOSAR OS 简介 —— OS 概述、与 FreeRTOS 对比、源码结构初探
- 第 2 章 AUTOSAR OS 移植与集成 —— 准备工作、集成步骤、启动验证
- 第 3 章 AUTOSAR OS 系统配置 —— 可扩展性类别、符合性类别、配置参数详解

核心机制（第 4-6 章）

- 第 4 章 中断管理 —— Cat1/Cat2 ISR、中断优先级、开关中断 API、临界区保护
- 第 5 章 任务基础 —— Basic/Extended Task、状态模型、优先级与抢占、栈与上下文
- 第 6 章 任务 API 详解 —— ActivateTask、TerminateTask、ChainTask、Schedule 等

调度与定时（第 7-8 章）

- 第 7 章 调度器与调度表 —— 优先级位图、Schedule Table、过期点、同步模式
- 第 8 章 计数器与报警 —— Counter 概念、Alarm API、周期/单次触发

同步与保护（第 9-11 章）

- 第 9 章 资源管理 —— 优先级天花板协议（PCP）、GetResource/ReleaseResource
- 第 10 章 事件机制 —— SetEvent/WaitEvent/ClearEvent、Extended Task 同步
- 第 11 章 OS 应用与内存保护 —— OS-Application、MPU、时间保护、ProtectionHook

高级特性（第 12-14 章）

- 第 12 章 Hook 函数 —— StartupHook、ErrorHook、PreTaskHook/PostTaskHook
- 第 13 章 多核 OS —— IOC、Spinlock、多核启动、跨核任务激活
- 第 14 章 时间管理 —— 系统 Tick、GetCounterValue/GetElapsedValue、时间保护

工程实践（第 15-18 章）

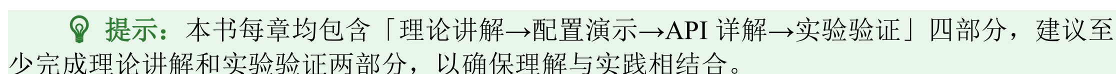
- 第 15 章 低功耗 OS —— MCU 低功耗模式、IdleHook、EcuM 睡眠集成
- 第 16 章 错误处理与调试 —— 错误码、ErrorHook、栈溢出检测、OS Trace
- 第 17 章 实战案例与工程模板 —— 周期任务、状态机、生产者-消费者、CAN 通信集成
- 第 18 章 性能优化与最佳实践 —— 任务划分、中断优化、栈空间、CPU 负载、常见陷阱

附录

- 附录 A：AUTOSAR OS 全部 API 速查表
- 附录 B：常用 ARXML 配置模板
- 附录 C：各主流工具链对比
- 附录 D：FreeRTOS 与 AUTOSAR OS 概念映射总结表
- 附录 E：API 调用上下文兼容性完整矩阵

阅读建议

- 建议按顺序阅读：从第 1 章开始，逐步建立 AUTOSAR OS 的完整知识体系
- 有 RTOS 经验者：可直接跳至感兴趣的章节，每章均可独立阅读
- 每章实验部分建议动手操作：在 TC334 Lite Kit 上编译运行，观察实际行为
- FreeRTOS 对比框：每章的「与 FreeRTOS 对比」框可帮助快速建立概念映射



提示：本书每章均包含「理论讲解→配置演示→API 详解→实验验证」四部分，建议至少完成理论讲解和实验验证两部分，以确保理解与实践相结合。

四段式结构说明

为保证学习效果，本书每章统一采用以下四段式结构：

- 【理论讲解】—— 阐述概念原理，对比 FreeRTOS 帮助理解
- 【配置演示】—— 展示 Os_Cfg.h / Os_Cfg.c 中的实际配置，含 OIL/ARXML 等效描述
- 【API 详解】—— 逐一讲解相关 API 的原型、参数、返回值、调用上下文与注意事项
- 【实验验证】—— 提供可在 TC334 硬件上运行的完整实验代码与预期结果分析

平台说明

本书所有示例均基于以下硬件和软件环境：

- 硬件平台：Infineon AURIX TC334 Lite Kit（KIT_A2G_TC334_LITE）
- MCU：TC334（TC33A，单核 TriCore TC1.6P，主频 300MHz）

- 编译器： Tricore-elf-gcc
- 驱动库：Infineon iLLD
- IDE：AURIX Development Studio (ADS)
- 调试器：DAP MiniWiggler

OS 版本

本书使用的 AUTOSAR OS 内核为小满/iSoft AUTOSAR OS（LGPL 开源版本）。

- 符合 AUTOSAR CP R19-11 OS 规范（SWS_OS）

【说明】本书基于 iSoft AUTOSAR OS 开源内核（遵循 R19-11 规范）编写，参考标准为 AUTOSAR CP R25-11 规范。部分 API 在 R20-11 至 R25-11 间已发生变更，已在相关位置标注。

【说明】本书基于 iSoft AUTOSAR OS 开源内核（遵循 R19-11 规范）编写，参考标准为 AUTOSAR CP R25-11 规范（AUTOSAR_CP_SWS_OS-25.pdf）。部分 API 在 R20-11 至 R25-11 版本间已发生变更（如 ControlIdle、StartNonAutosarCore 被移除），本书在相关位置已做标注。

- 支持可扩展性类别 SC1~SC4
- 支持符合性类别 BCC1/BCC2/ECC1/ECC2
- 支持多核（本书以单核 TC334 为基础讲解）
- 开源协议：LGPL v2.1

当前工程配置：SC1 + BCC2 + 单核，代表 AUTOSAR OS 最精简的可运行系统。

术语表

本术语表收录 AUTOSAR OS 相关核心术语，按英文字母排序。

术语（英文）	中文	定义/说明
ActivateTask	激活任务	将任务从挂起状态（SUSPENDED）转为就绪状态（READY）的 API
Alarm	报警	基于计数器（Counter）的定时触发机制，可周期性或单次触发
AUTOSAR	汽车开放系统架构	AUTomotive Open System ARchitecture，汽车电子软件标准化组织及其规范
BCC1	基本一致性类 1	Basic Conformance Class 1：仅支持 Basic Task，每个任务最大激活次数为 1
BCC2	基本一致性类 2	Basic Conformance Class 2：支持 Basic Task 多次激活排队
Cat1 ISR	第一类中断	不使用 OS 服务的高优先级中断，OS 不参与其上下文管理
Cat2 ISR	第二类中断	可使用 OS 服务的中断，由 OS 完全管理上下文保存/恢复与重调度
ChainTask	链式任务切换	终止当前任务并激活另一任务的原子操作，避免中间调度点
Counter	计数器	OS 中的时基对象，可为硬件或软件实现，驱动 Alarm 和 ScheduleTable
CSA	上下文保存区	Context Save Area，TriCore 硬件自动保存/恢复 CPU 上下文的机制
ECC1	扩展一致性类 1	Extended Conformance Class 1：支持 Extended Task（可等待事件），每个任务最大激活次数为 1
ECC2	扩展一致性类 2	Extended Conformance Class 2：支持 Extended Task 多次激活排队
EcuM	ECU 状态管	ECU State Manager，管理 ECU 启动、运行、睡眠和关机状态的

	理器	BSW 模块
Event	事件	用于 Extended Task 间同步的机制，通过位掩码（EventMaskType）标识
GetResource	获取资源	通过优先级天花板协议获取资源的 API，提升当前运行优先级
Hook	钩子函数	OS 在特定时机（启动、关闭、错误、任务切换等）调用的用户回调函数
IOC	OS 间通信	Inter-OS-Application Communication，跨 OS-Application 的安全通信机制
ISR	中断服务程序	Interrupt Service Routine，响应硬件中断请求的处理函数
MemoryProtection	内存保护	基于 MPU 的地址空间隔离，防止 OS-Application 间的非法内存访问（SC3/SC4）
MPU	内存保护单元	Memory Protection Unit，硬件级别的地址空间访问控制机制
OIL	OS 实现语言	OSEK Implementation Language，用于描述 OS 静态配置的标准化语言
Os_Cfg.h	OS 配置头文件	包含所有静态配置宏定义的头文件，决定 OS 运行时行为
OsApplication	OS 应用	任务/ISR/资源/计数器/报警的逻辑分组单元，用于权限隔离
OSEK	开放系统及接口	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug，AUTOSAR OS 的前身标准
PCP	优先级天花板协议	Priority Ceiling Protocol，防止优先级反转的资源管理机制
ReleaseResource	释放资源	释放先前获取的资源，恢复原始运行优先级
RES_SCHEDULER	调度器资源	特殊的内置资源，获取后阻止当前任务被更高优先级任务抢占
Resource	资源	用于互斥访问共享数据的 OS 对象，内置优先级天花板协议
ScheduleTable	调度表	基于计数器的同步调度机制，通过过期点（Expiry Point）在精确时刻触发动作
Spinlock	自旋锁	多核环境下的忙等互斥机制，通过硬件原子操作实现核间同步
STM	系统定时器	System Timer Module，TC3xx 硬件定时器，为 OS 提供系统时钟节拍
Task	任务	OS 调度的基本执行单元，分为 Basic Task 和 Extended Task
TerminateTask	终止任务	将当前正在运行的任务转为挂起状态（SUSPENDED）的 API
Timing Protection	时间保护	监控任务/ISR 执行时间的安全机制，超时触发 ProtectionHook（SC2/SC4）
Trusted	可信	具有完全硬件访问权限的 OsApplication 权限级别
NonTrusted	非可信	受限的 OsApplication 权限级别，由 MPU 强制限制其内存访问范围

第一章 AUTOSAR OS 简介

1.1 初识 AUTOSAR OS

1.1.1 什么是 AUTOSAR OS？

【理论讲解】

AUTOSAR OS（AUTomotive Open System ARchitecture Operating System）是由 AUTOSAR 联盟定义的汽车级实时操作系统规范。它是基于 OSEK/VDX 标准（ISO 17356）扩展而来的静态配置实时操作系统（RTOS），专为汽车电子控制单元（ECU）设计。

与 FreeRTOS 等通用嵌入式 RTOS 最大的区别在于：AUTOSAR OS 的所有内核对象（Task、Alarm、Counter、Resource 等）均在编译前通过配置工具静态确定，不允许在运行时动态创建或销毁。这种设计哲学源于汽车功能安全（ISO 26262）对确定性行为的严格要求。

【与 FreeRTOS 的核心差异对比】

特性	FreeRTOS	AUTOSAR OS
对象创建方式	动态创建（xTaskCreate）	静态配置（OIL/ARXML 生成）
优先级模型	数值越大优先级越高	数值越大优先级越高（0 为最低）
优先级反转保护	需手动使用 Mutex	内置 PCP（Priority Ceiling Protocol）
中断管理	用户自行管理	ISR Category 1/2 分类管理
内存保护	MPU 可选	SC3/SC4 强制内存保护
时间保护	无	SC2/SC4 执行时间预算监控
多核支持	SMP 为主	AMP 静态核分配
配置方式	代码中动态配置	OIL/ARXML 工具生成
安全认证	无官方认证	支持 ASIL-D 认证

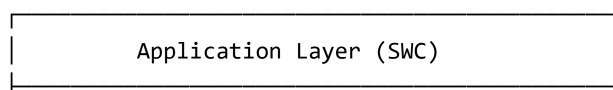
【关键概念速记】

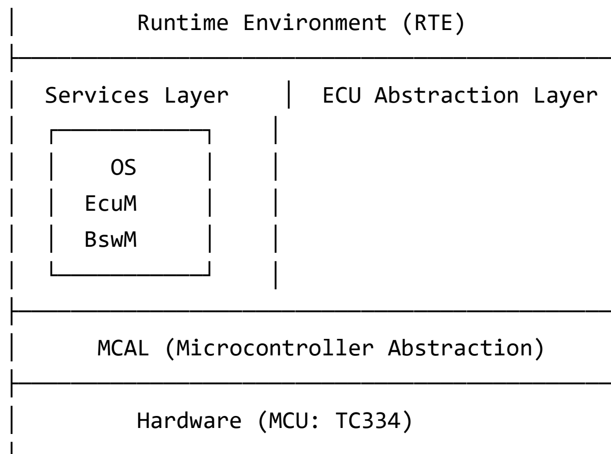
- Task（任务）：最基本的调度单元，分 Basic Task 和 Extended Task
- Alarm（告警）：基于 Counter 的定时触发机制，用于周期激活 Task
- Counter（计数器）：硬件/软件定时器抽象，驱动 Alarm 和 Schedule Table
- Resource（资源）：互斥访问共享资源，内置优先级天花板协议
- Event（事件）：仅 Extended Task 可等待，用于任务间同步
- ISR Category 2：由 OS 管理的中断，可调用 OS API

1.1.2 AUTOSAR OS 在整车架构中的定位

【理论讲解】

在 AUTOSAR 经典平台（Classic Platform, CP）分层架构中，OS 位于基础软件层（BSW）的服务层（Services Layer）。其层级关系如下：





OS 为 BSW 中的其他模块（如 EcuM、BswM、ComM 等）以及应用层 SWC（通过 RTE）提供任务调度、中断管理、时间服务和保护机制。在 AUTOSAR 架构中，OS 是最底层的运行时基础设施。

对于从 FreeRTOS 过渡的工程师而言，可以将 AUTOSAR OS 理解为一个“受约束的 RTOS”——它提供与 FreeRTOS 类似的调度能力，但在架构上强制要求静态配置、标准化接口和安全保护机制。

1.1.3 AUTOSAR OS 与 OSEK/VDX 的关系

【理论讲解】

AUTOSAR OS 规范脱胎于 OSEK/VDX OS 标准（1990 年代由欧洲汽车厂商联合制定）。二者关系可概括为：

• OSEK OS 定义了基础的任务管理、中断管理、资源管理和告警管理

• AUTOSAR OS 完全兼容 OSEK OS 的 API 和语义

• AUTOSAR OS 在 OSEK 基础上扩展了：

- 可扩展性类别（Scalability Class SC1~SC4）

- 多核支持（Multi-Core OS）

- 内存保护（Memory Protection）

- 时间保护（Timing Protection）

- OS-Application 隔离机制

- Schedule Table（调度表）

- IOC（Inter OS-Application Communication）

在本项目的 TC334 单核配置中，我们使用 SC1（无内存/时间保护）+ BCC2（基本任务，多次激活），这与传统 OSEK OS 的功能范围基本一致。随着对 AUTOSAR OS 理解的深入，后续章节将逐步介绍 SC2~SC4 的高级保护特性。

【OSEK 到 AUTOSAR 的演进对照】

OSEK/VDX 特性	AUTOSAR OS 扩展
BCC1/BCC2/ECC1/ECC2	保留，作为符合性类别
DeclareTask/DeclareAlarm	保留，通过 OIL/ARXML 配置
GetResource/ReleaseResource	保留，增加 Spinlock（多核）
ISR Category 1/2	保留，增加 ISR 映射到核
无多核	增加 Multi-Core OS
无内存保护	SC3/SC4 增加 MPU 保护
无时间保护	SC2/SC4 增加时间预算

1.1.4 AUTOSAR OS 的特点与优势

【理论讲解】

相比 FreeRTOS 等通用 RTOS，AUTOSAR OS 具备以下核心优势：

1. 静态确定性

所有 OS 对象在编译前完全确定。无 malloc/free，无动态对象创建。这使得系统行为在设计阶段即可完全预测——对于 ASIL-D 等级的功能安全应用至关重要。

2. 优先级天花板协议（PCP）

AUTOSAR OS 内置的 Resource 机制使用 PCP 防止优先级反转。当 Task 获取 Resource 时，其运行优先级被提升至资源天花板优先级，从而消除优先级反转问题。相比 FreeRTOS 的 Priority Inheritance，PCP 提供了更强的确定性保证。

3. 分层安全保护

通过可扩展性类别（SC1~SC4），AUTOSAR OS 可根据安全需求逐级增加保护：

- SC1：基础调度与中断管理（本项目当前配置）
- SC2：增加时间保护——防止 Task/ISR 超时
- SC3：增加内存保护——使用 MPU 隔离 OS-Application
- SC4：SC2 + SC3 的完整保护

4. 标准化接口

所有 AUTOSAR OS 实现遵循相同的 API 规范（SWS_OS），这意味着应用代码可以跨不同供应商的 OS 实现进行移植（如 Vector MICROSAR OS、EB tresos OS、ETAS RTA-OS 等）。

5. 多核原生支持

AUTOSAR OS 从规范层面定义了多核 OS 行为，包括跨核 Task 激活、Spinlock、核间同步等，为多核 MCU（如 TC3xx 系列的 TC397 六核）提供了标准化的并发编程模型。

1.2 学习资源与参考资料

1.2.1 AUTOSAR 官方规范文档（SWS_OS）

【参考资料】

AUTOSAR 官方发布的 OS 相关规范文档包括：

• AUTOSAR_SWS_OS（OS 软件规范）—— 核心 API 与行为定义

• AUTOSAR_SRS_OS（OS 需求规范）—— 功能需求追溯

• AUTOSAR_TPS_OS（OS 模板参数规范）—— OIL/ARXML 配置参数定义

• AUTOSAR_TPS_ECUConfiguration（ECU 配置模板）—— ARXML 参数定义

• AUTOSAR_EXP_LayeredSoftwareArchitecture —— 分层架构说明

获取路径：https://www.autosar.org/standards/ （需注册免费账号）

建议重点阅读 SWS_OS 中的以下章节：

- Chapter 7: API Specification
- Chapter 8: Sequence Diagrams
- Annex A: Conformance Classes

1.2.2 供应商工具文档

【参考资料】

主流 AUTOSAR OS 供应商及其文档资源：

供应商	产品名称	配置工具	文档获取
Vector	MICROSAR OS	DaVinci Developer/Configurator	Vector Customer Portal
EB (Elektrobit)	tresos AutoCore OS	tresos Studio	EB Support Portal
ETAS	RTA-OS	ISOLAR-A/B	ETAS Download Center
本项目	自研内核	手工配置	Os_Cfg.h / Os_Cfg.c

本项目采用自研 AUTOSAR OS 内核，配置通过手工编辑 Os_Cfg.h 宏定义实现。在商业项目中，配置通常通过供应商提供的 GUI 工具（如 DaVinci Configurator）生成。

1.2.3 目标 MCU 架构参考手册

【参考资料】

本项目基于 Infineon AURIX TC334 平台，需要参考的硬件文档：

• TC33x User Manual（UM）—— MCU 寄存器与外设说明

• TC3xx Architecture Manual —— TriCore 1.6P 指令集与 CSA 机制

• AURIX TC3xx iLLD User Manual —— 底层驱动库 API

• KIT_A2G_TC334_LITE Schematic —— 开发板原理图

其中对 OS 移植最关键的硬件知识点：

• CSA（Context Save Area）：TriCore 硬件上下文保存机制，OS 通过 CSA 链表实现任务切换

• STM（System Timer Module）：提供 OS 系统时钟节拍（本项目使用 STM0_SR0）

• SRC（Service Request Control）：中断路由与优先级配置

• MPU（Memory Protection Unit）：SC3/SC4 内存保护的硬件基础

1.3 AUTOSAR OS 源码与配置初探

1.3.1 OS 模块在 BSW 中的层级

【理论讲解】

在本项目的工程目录结构中，AUTOSAR OS 相关代码分布如下：

```
TC334_LK_Autosar_OS/
├── RTOS/                           # OS 内核源码
│   ├── Kernel/                     # 内核核心实现
│   │   ├── inc/                    # 内核头文件
│   │   └── src/                    # 内核源文件（21 个.c 文件）
│   ├── Extend/                     # 扩展功能（OS Monitor 等）
│   │   ├── inc/
│   │   └── src/
│   └── Portable/                   # 硬件移植层
│       ├── Processor/TriCore/TC1.6/   # TriCore 处理器适配
│       └── Mcu/Infineon/TC334/        # TC334 MCU 适配
├── Os_Cfg/                         # OS 配置文件
│   ├── Os_Cfg.h                    # 主配置头文件（宏定义）
│   ├── Os_Cfg.c                    # 配置数据实例化
│   ├── Os_CoreCfg.h/c              # 核相关配置
│   ├── Os_UserInf.c                # 用户 Hook 实现
│   ├── Os_Intvet.c                 # 中断向量配置
│   ├── Os_MemMap.h                 # 内存映射段定义
│   ├── Os_MprotCfg.h               # 内存保护配置
│   ├── Compiler.h                  # 编译器抽象
│   └── Os_Monitor.h                # OS 监控配置
├── App/                            # 应用任务代码
│   └── App_Tasks.c                 # Task_Init, Task_Blink
├── Cpu0_Main.c                     # 主函数（StartOS 入口）
└── Lcf_Gnuc_Tricore_Tc.lsl         # GCC 链接脚本
```

从 FreeRTOS 工程师的视角来理解这个结构：

• RTOS/Kernel/ 相当于 FreeRTOS 的 Source/ 目录

• RTOS/Portable/ 相当于 FreeRTOS 的 portable/GCC/ARM_CM4/ 移植层

• Os_Cfg/ 相当于 FreeRTOS 的 FreeRTOSConfig.h，但内容远比单个头文件丰富

• App_Tasks.c 中使用 TASK() 宏定义任务体，类似 FreeRTOS 中 xTaskCreate 注册的任务函数

1.3.2 生成代码与配置文件预览（Os_Cfg.h / Os_Cfg.c）

【配置演示】

Os_Cfg.h 是 AUTOSAR OS 的核心配置文件，所有 OS 行为均由其中的宏定义决定。以下是本项目 TC334 最小系统的关键配置摘录：

```
/* ===== Os_Cfg.h 核心配置摘录 ===== */

/* 核心配置 */
#define CFG_CORE_MAX                     (1U)     // 单核
#define OS_AUTOSAR_CORES                 1U       // AUTOSAR 管理的核心数理的核心数

/* 可扩展性类别 */
#define CFG_SC                           OS_SC1     // SC1: 基础功能

/* 符合性类别 */
#define CFG_CC                           OS_BCC2     // 基本任务，多次激活

/* 系统状态 */
#define CFG_STATUS                       OS_STATUS_STANDARD

/* 任务配置 */
#define CFG_TASK_MAX                     (3U)             // 总任务数任务数
#define CFG_PRIORITY_MAX_CORE0           (3U)             // 优先级最大值先级最大值
#define Task_Init               ((Os_TaskType)0x0000U)    // 初始化任务始化任务
#define Task_Blink              ((Os_TaskType)0x0001U)    // LED 闪烁任务烁任务
#define OS_TASK_IDLE_CORE0      ((Os_TaskType)0x0002U)    // 空闲任务闲任务

/* 中断配置 */
#define CFG_ISR_MAX                      (1U)     // ISR 总数数
#define CFG_ISR2_MAX                     (1U)     // ISR2 数量量
#define CFG_INT_NEST_ENABLE              TRUE     // 支持中断嵌套持中断嵌套

/* 计数器与告警 */
#define CFG_COUNTER_MAX                  (1U)     // 计数器数量数器数量
#define CFG_ALARM_MAX                    (1U)     // 告警数量警数量
#define AlarmBlink              ((Os_AlarmType)0x0000U)

/* Hook 配置 */
#define CFG_ERRORHOOK                    TRUE
#define CFG_STARTUPHOOK                  TRUE
#define CFG_SHUTDOWNHOOK                 TRUE

/* 调度策略 */
```

```
#define CFG_SCHED_POLICY    OS_PREEMPTIVE_MIXED    // 混合抢占合抢占

/* 栈监控 */
#define CFG_STACK_MONITOR             TRUE
```

【与 FreeRTOS 配置对比】

FreeRTOS 工程师熟悉的 FreeRTOSConfig.h 通常是一个 50~100 行的头文件。而 AUTOSAR OS 的配置远更细粒度——不仅定义系统参数，还静态声明每个 Task ID、Alarm ID、Counter ID 等。

FreeRTOS 配置	AUTOSAR OS 等效配置
configMAX_PRIORITIES	CFG_PRIORITY_MAX_CORE0 = 3
configTICK_RATE_HZ	CFG_REG_OSTIMER_VALUE_CORE0 (STM 比较值)
configUSE_PREEMPTION	CFG_SCHED_POLICY = OS_PREEMPTIVE_MIXED
configCHECK_FOR_STACK_OVERFLOW	CFG_STACK_MONITOR = TRUE
configUSE_IDLE_HOOK	IdleHook_Core0() (内置空闲任务)
N/A (动态创建)	CFG_TASK_MAX = 3 (静态声明)
N/A (无此概念)	CFG_SC = OS_SC1 (可扩展性类别)
N/A (无此概念)	CFG_CC = OS_BCC2 (符合性类别)

【API 详解 - 系统启动流程】

AUTOSAR OS 的启动通过 StartOS() API 触发，这是 OS 的唯一入口点：

```
/* Cpu0_Main.c - StartOS 调用 */
void core0_main(void)
{
  IfxCpu_enableInterrupts();
  IfxScuWdt_disableCpuWatchdog(...);
  IfxScuWdt_disableSafetyWatchdog(...);

  IfxCpu_emitEvent(&g_cpuSyncEvent);
  IfxCpu_waitEvent(&g_cpuSyncEvent, 1);

  /* StartOS() 之后不会返回 */
  StartOS(OSDEFAULTAPPMODE);

  while(1) { /* 不应到达 */ }
}
```

StartOS() 执行后的内部流程：

1. 初始化 OS 内核数据结构（TCB、CCB、ACB 等控制块）
2. 配置硬件定时器（STM0）作为系统时钟源
3. 调用 StartupHook()（如果 CFG_STARTUPHOOK = TRUE）
4. 激活所有 Autostart Task（Task_Init 和 OS_TASK_IDLE_CORE0）
5. 执行第一次调度，进入最高优先级就绪任务

【实验验证】

验证目标：确认 AUTOSAR OS 基础概念理解正确

实验 1-1：阅读 Os_Cfg.h 并回答以下问题

Q1: 当前工程配置了几个 Task？它们的优先级关系如何？

A1: 3 个 Task — Task_Init(优先级 2,最高)、Task_Blink(优先级 1)、IdleCore0(优先级 0,最低)

Q2: 系统使用哪种调度策略？

A2: OS_PREEMPTIVE_MIXED — Task_Init 为 NON_PREEMPTIVE，Task_Blink 为 FULL_PREEMPTIVE

Q3: AlarmBlink 如何被启动？

A3: 在 StartupHook() 中调用 SetRelAlarm(AlarmBlink, 500U, 500U)，即启动后 500ms 首次触发，之后每 500ms 周期触发

实验 1-2：对比 FreeRTOS 的等效实现

如果用 FreeRTOS 实现同样的 LED 闪烁功能，代码如下：

```
/* FreeRTOS 等效实现（仅供对比） */
void vTaskBlink(void *pvParams) {
  for(;;) {
    GPIO_Toggle(LED_PIN);
    vTaskDelay(pdMS_TO_TICKS(500));
  }
}
int main(void) {
  xTaskCreate(vTaskBlink, "Blink", 128, NULL, 1, NULL);
  vTaskStartScheduler(); // 类似 StartOS()
}
```

而 AUTOSAR OS 中：

- Task 不包含无限循环——执行完毕必须调用 TerminateTask()
- 周期行为通过 Alarm 激活 Task 实现，而非 Task 内部延时
- Task 的栈大小、优先级等属性在配置文件中静态定义

1.3.3 工程信息总览

【项目信息】

以下是本项目 TC334 AUTOSAR OS 最小系统的核心参数总览：

项目参数	配置值
MCU	Infineon TC334 (TC33A, 单核 TriCore TC1.6P)
编译器	HighTec GCC (tricore-elf-gcc)
OS 内核	小满/isoft AUTOSAR OS (LGPL)
可扩展类	SC1 (基础功能)
符合性类别	BCC2（基本任务，多次激活）
系统时钟	STM0 @ 100MHz, 1ms Tick
任务数	3 (Task_Init, Task_Blink, IdleTask)
ISR 数	1 (STM0 Cat2)
Counter	1 (SystemCounter, 硬件)
Alarm	1 (AlarmBlink)

此配置代表了 AUTOSAR OS 最精简的可运行系统——1 个硬件 Counter 驱动 1 个 Alarm 周期激活 Task_Blink，Task_Init 完成初始化后终止，IdleTask 作为最低优先级后台任务持续运行。

1.3.4 关键源文件索引表

【参考资料】

下表列出了本工程中所有与 AUTOSAR OS 相关的关键源文件，供开发者快速定位代码：

文件路径	功能说明
RTOS/Kernel/inc/Os.h	OS 公开 API 声明（92.6KB，全部服务接口）
RTOS/Kernel/inc/Os_Types.h	数据类型定义（140+类型）
RTOS/Kernel/inc/Os_Internal.h	内部变量和结构声明
RTOS/Kernel/inc/Os_ECode.h	错误代码定义（25 个）
RTOS/Kernel/inc/Os_Hook.h	Hook 函数接口
RTOS/Kernel/inc/Os_Macros.h	系统宏定义
RTOS/Kernel/inc/Os_Err.h	错误处理机制
RTOS/Kernel/inc/Os_Trace.h	跟踪系统
RTOS/Kernel/src/Os_Task.c	任务管理实现
RTOS/Kernel/src/Os_Alarm.c	告警管理实现
RTOS/Kernel/src/Os_Counter.c	计数器管理实现
RTOS/Kernel/src/Os_Event.c	事件管理实现
RTOS/Kernel/src/Os_Resource.c	资源管理实现
RTOS/Kernel/src/Os_Sched.c	调度器实现
RTOS/Kernel/src/Os_Hook.c	Hook 函数实现
RTOS/Extend/src/Os_ScheduleTable.c	调度表实现
RTOS/Extend/src/Os_Spinlock.c	自旋锁实现（多核）
RTOS/Portable/Processor/TriCore/TC1.6/src/Arch_Irq.c	中断底层处理
RTOS/Portable/Processor/TriCore/TC1.6/src/Arch_Context.c	上下文切换
RTOS/Portable/Processor/TriCore/TC1.6/src/Arch_Timer.c	定时器驱动
Os_Cfg/Os_Cfg.h	OS 配置宏定义（95+宏）
Os_Cfg/Os_Cfg.c	OS 配置数据实例
Os_Cfg/Os_CoreCfg.h	核心配置
Os_Cfg/Os_CoreCfg.c	核心配置数据
Lcf_Gnuc_Tricore_Tc.lsl	GCC 链接脚本（CSA/栈段）

提示：阅读源码时建议从 Os.h 的 API 声明入手，结合 Os_Cfg.h 的宏定义理解配置与内核的

关联，再深入 Kernel/src/ 的具体实现。

1.4 本章小结

AUTOSAR OS 是一个为汽车安全关键系统设计的静态配置 RTOS，它继承了 OSEK/VDX 的核心设计，并增加了多核支持、内存/时间保护等现代安全特性。对于 FreeRTOS 工程师而言，最重要的思维转变是从“运行时动态创建”转向“编译前静态配置”。后续章节将详细介绍如何将这套 OS 集成到 TC334 工程中并进行系统配置。

第二章 AUTOSAR OS 移植与集成

2.1 准备工作

2.1.1 准备基础工程（MCAL + EcuM 基础）

【理论讲解】

AUTOSAR OS 的移植需要一个已经能正常编译、烧录并运行的基础工程作为起点。在本项目中，我们基于 Infineon AURIX Development Studio (ADS) 创建的 TC334 iLLD 模板工程进行集成。

【配置演示 - 基础工程要求】

移植 AUTOSAR OS 前，基础工程需满足以下条件：

项目	要求	本工程实际值
开发板	AURIX TC3xx Lite Kit	KIT_A2G_TC334_LITE
编译器	HighTec GCC for TriCore	tricore-elf-gcc v4.9.x
IDE	AURIX Development Studio	ADS v1.10.24
底层驱动	iLLD (Infineon Low Level Driver)	iLLD TC33A
启动代码	Ifx_Ssw 启动序列	Ifx_Cfg_Ssw.c
链接脚本	GCC Linker Script	Lcf_Gnuc_Tricore_Tc.lsl
基础验证	能编译通过并烧录运行	LED 闪烁 Demo

与 FreeRTOS 移植的对比：FreeRTOS 移植通常只需将 Source/ 和 portable/ 目录加入工程，再配置 FreeRTOSConfig.h 即可。AUTOSAR OS 的集成更复杂，需要：

- 内核源码/库文件
- 处理器移植层（上下文切换、中断管理）
- MCU 适配层（定时器、MPU 等硬件初始化）
- 配置文件集（Os_Cfg.h/c、Os_CoreCfg.h/c、Compiler.h 等）
- 链接脚本修改（CSA、中断向量表、OS 栈段）

2.1.2 获取 OS 内核源码或库（供应商交付物）

【理论讲解】

商业 AUTOSAR OS 供应商通常以以下形式交付 OS 组件：

- 源码交付：完整的内核源代码（本项目方式）
- 库交付：预编译的 .a/.lib 文件 + 头文件
- 配置工具生成：通过 OIL/ARXML 文件由配置工具自动生成配置代码

本项目的 OS 内核源码结构如下：

```
RTOS/
├── Kernel/                         # 内核核心代码
│   ├── inc/                        # 内核头文件 (8 个)
```

```
│   │   ├── Os.h                    # OS 主头文件（应用层包含）
│   │   ├── Os_Internal.h           # 内核内部声明
│   │   ├── Os_Types.h              # OS 类型定义
│   │   ├── Os_CfgData.h            # 配置数据结构
│   │   └── ...                     # 其他内核头文件
│   └── src/                        # 内核源文件 (21 个)
│       ├── Os_Task.c               # Task 管理
│       ├── Os_Alarm.c              # Alarm 管理
│       ├── Os_Counter.c            # Counter 管理
│       ├── Os_Resource.c           # Resource 管理
│       ├── Os_Event.c              # Event 管理
│       ├── Os_Sched.c              # 调度器
│       ├── Os_Hook.c               # Hook 管理
│       └── ...                     # 其他核心模块
├── Extend/                         # 扩展功能
│   ├── inc/Arch_Extend.h           # 扩展接口
│   └── src/Arch_Extend.c           # OS Monitor 等
└── Portable/                       # 硬件移植层
    ├── Processor/TriCore/TC1.6/    # TriCore 处理器适配
    │   ├── inc/Arch_Processor.h    # 处理器抽象接口
    │   └── src/Arch_Processor.c    # 上下文切换/CSA 管理
    └── Mcu/Infineon/TC334/         # TC334 MCU 适配
        ├── inc/Arch_Mcu.h          # MCU 抽象接口
        └── src/Arch_Mcu.c          # STM 定时器/中断配置
```

【与 FreeRTOS 移植层的类比】

FreeRTOS 移植层	**AUTOSAR OS 对应层**	**职责**
portable/GCC/ARM_CM4/port.c	Portable/Processor/TriCore/TC1.6/	上下文切换、Tick 中断
N/A	Portable/Mcu/Infineon/TC334/	MCU 特定硬件初始化
FreeRTOSConfig.h	Os_Cfg/ 目录	系统配置参数
Source/tasks.c	Kernel/src/Os_Task.c	Task 调度实现
Source/timers.c	Kernel/src/Os_Alarm.c	定时触发机制

2.2 OS 集成步骤

2.2.1 向工程添加 OS 相关文件

【配置演示】

步骤 1：将 RTOS/ 目录复制到工程根目录

步骤 2：创建 Os_Cfg/ 目录，放置配置文件

步骤 3：配置工程 Include Path

需要添加的 Include Path（在 ADS 中配置）：

```
# AURIX_GCC_Compiler-Include_paths__-I_.opt 内容
-I"../RTOS/Kernel/inc"
-I"../RTOS/Extend/inc"
-I"../RTOS/Portable/Processor/TriCore/TC1.6/inc"
-I"../RTOS/Portable/Mcu/Infineon/TC334/inc"
-I"../Os_Cfg"
-I"../Configurations"
```

步骤 4：将以下源文件加入编译：

目录	文件数	说明
RTOS/Kernel/src/	21 个 .c	内核核心实现
RTOS/Extend/src/	1 个 .c	OS Monitor 扩展
RTOS/Portable/Processor/TriCore/TC1.6/src/	1 个 .c	上下文切换
RTOS/Portable/Mcu/Infineon/TC334/src/	1 个 .c	MCU 硬件适配
Os_Cfg/	5 个 .c	配置数据实例化
App/	1 个 .c	应用任务

2.2.2 配置工具链与链接脚本

【配置演示】

链接脚本 (Lcf_Gnuc_Tricore_Tc.lsl) 是 OS 移植中最关键的硬件配置文件。它定义了 OS 运行所需的内存布局。以下是与 OS 相关的关键配置：

1) CSA 配置（Context Save Area）

TriCore 架构使用硬件 CSA 链表实现上下文保存与恢复，这是 OS 任务切换的硬件基础：

```
/* Lcf_Gnuc_Tricore_Tc.lsl - CSA 配置 */
LCF_CSA0_SIZE = 16k;        /* CSA 区域大小: 16KB */
LCF_USTACK0_SIZE = 2k;      /* 用户栈: 2KB */
LCF_ISTACK0_SIZE = 2k;      /* 中断栈: 2KB */

/* CSA 位于 DSPR0 末尾 */
LCF_CSA0_OFFSET = (LCF_DSPR0_SIZE - 1k - LCF_CSA0_SIZE);
/* 中断栈在 CSA 下方 */
LCF_ISTACK0_OFFSET = (LCF_CSA0_OFFSET - 256 - LCF_ISTACK0_SIZE);
/* 用户栈在中断栈下方 */
LCF_USTACK0_OFFSET = (LCF_ISTACK0_OFFSET - 256 - LCF_USTACK0_SIZE);
```

CSA 大小计算：每个 CSA 节点占 64 字节，16KB 可容纳 256 个 CSA 节点。每次函数调用消耗 1 个 Upper CSA + 1 个 Lower CSA，因此 16KB 支持约 128 层函数嵌套。

2) OS 栈段配置

链接脚本中为每个 Task 和 ISR 分配专用栈空间：

```
/* Lcf_Gnuc_Tricore_Tc.lsl - AUTOSAR OS Stack Sections */
```

```
CORE_ID = CPU0;
SECTIONS
{
  .OS_STACK_Task_Init (NOLOAD) : ALIGN(8)
  {
    PROVIDE(__OS_STACK_Task_Init_BEGIN = .);
    *(.bss.OS_STACK_Task_Init)
    PROVIDE(__OS_STACK_Task_Init_END = .);
  } > dsram0

  .OS_STACK_Task_Blink (NOLOAD) : ALIGN(8)
  {
    PROVIDE(__OS_STACK_Task_Blink_BEGIN = .);
    *(.bss.OS_STACK_Task_Blink)
    PROVIDE(__OS_STACK_Task_Blink_END = .);
  } > dsram0

  .OS_STACK_Task_Idle (NOLOAD) : ALIGN(8)
  {
    PROVIDE(__OS_STACK_Task_Idle_BEGIN = .);
    *(.bss.OS_STACK_Task_Idle)
    PROVIDE(__OS_STACK_Task_Idle_END = .);
  } > dsram0

  .OS_ISR_STACK_Core0 (NOLOAD) : ALIGN(8)
  {
    PROVIDE(__OS_ISR_STACK_Core0_BEGIN = .);
    *(.bss.OS_ISR_STACK_Core0)
    PROVIDE(__OS_ISR_STACK_Core0_END = .);
  } > dsram0
}
```

注意事项：

• 所有 OS 栈均位于 DSRAM0（本地高速 SRAM），确保访问速度

• ALIGN(8) 保证 8 字节对齐，满足 TriCore 栈对齐要求

• 每个 Task 栈独立分配，便于栈监控检测溢出

3) 中断向量表配置

TC334 支持 256 个中断向量，链接脚本中为每个向量分配 32 字节空间：

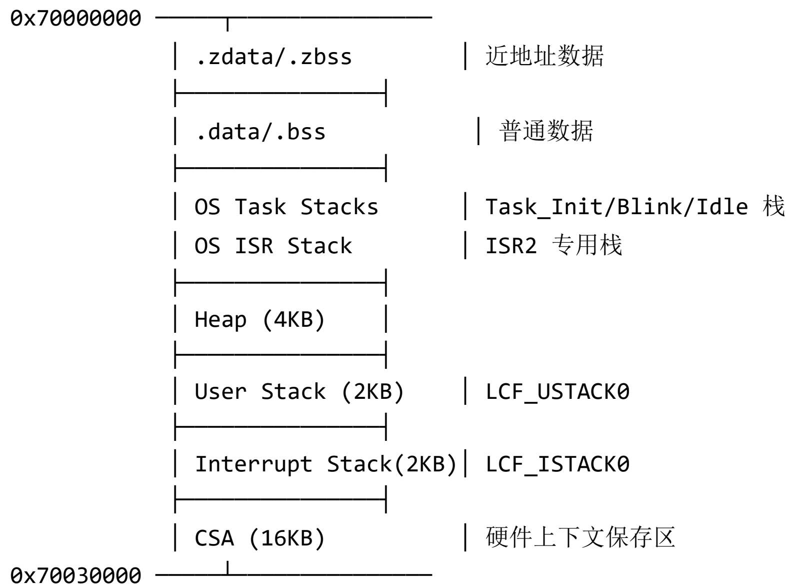
```
/* 中断向量表起始地址 */
LCF_INTVEC0_START = 0x801FE000;
__INTTAB_CPU0 = LCF_INTVEC0_START;

/* 每个中断入口占 32 字节 (0x20) */
.inttab_tc0_000 (__INTTAB_CPU0 + 0x0000) : { . = ALIGN(8); KEEP(*(.intvec_tc0_0)); }
.inttab_tc0_001 (__INTTAB_CPU0 + 0x0020) : { . = ALIGN(8); KEEP(*(.intvec_tc0_1)); }
```

```
/* ... 256 个入口 ... */
```

OS 的系统定时器 ISR（STM0_SR0）通过 Os_Intvet.c 中的代码注册到对应的中断向量位置。

4) 内存布局总览

TC334 DSPR0 (192KB) 的内存布局如下：



2.2.3 OIL 或 ARXML 配置生成

【理论讲解】

商业 AUTOSAR OS 通常使用 OIL（OSEK Implementation Language）或 ARXML（AUTOSAR XML）文件描述 OS 配置，然后通过配置工具自动生成 C 代码。

本项目采用手工配置方式，直接编写 Os_Cfg.h/Os_Cfg.c，其效果等同于工具生成。以下是配置流程对比：

方式	流程	优势	劣势
OIL 配置	OIL 文件 → OS Generator → C 代码	标准化、可移植	需工具支持
ARXML 配置	ARXML → BSW Generator → C 代码	与整车配置集成	复杂度高
手工配置	直接编写 Os_Cfg.h/c	灵活、快速原型验证	易出错、不可移植

【配置演示 - Os_Cfg.c 中的任务配置实例】

以下代码展示了 Os_Cfg.c 中 Task 的静态配置结构：

```
/* Os_Cfg.c - Task 配置数据 */
static const Os_TaskCfgType Os_TaskCfgCore0[CFG_TASK_MAX_CORE0] = {
  {
```

```
        &Os_TaskEntry_Task_Init,    /* osTaskEntry: 任务入口函数 */
        Os_GetObjLocalId(Task_Init),/* osTaskStackId: 栈索引 */
        1U,                         /* osTaskActivation: 最大激活次数 */
        2U,                         /* osTaskPriority: 优先级 */
        OSDEFAULTAPPMODE,           /* osTaskAutoStartMode: 自动启动 */
        OS_PREEMPTIVE_NON,          /* osTaskSchedule: 不可抢占 */
    },
    {
        &Os_TaskEntry_Task_Blink,   /* osTaskEntry */
        Os_GetObjLocalId(Task_Blink),/* osTaskStackId */
        1U,                         /* osTaskActivation */
        1U,                         /* osTaskPriority */
        OS_NULL_APPMODE,            /* osTaskAutoStartMode: 手动激活 */
        OS_PREEMPTIVE_FULL,         /* osTaskSchedule: 完全抢占 */
    },
    {
        &Os_TaskEntry_IdleCore0,    /* osTaskEntry */
        Os_GetObjLocalId(OS_TASK_IDLE_CORE0),
        1U,                         /* osTaskActivation */
        0U,                         /* osTaskPriority: 最低 */
        OS_ALL_APPMODE,             /* osTaskAutoStartMode: 所有模式启动 */
        OS_PREEMPTIVE_FULL,         /* osTaskSchedule */
    }
};
```

【API 详解 - 配置引用的关键类型】

- Os_TaskCfgType: Task 配置结构体，包含入口、栈、优先级、调度属性
- Os_AlarmCfgType: Alarm 配置，关联 Counter 和回调动作
- Os_CounterCfgType: Counter 配置，定义最大值、节拍基数
- Os_IsrCfgType: ISR 配置，定义中断源、类型、是否嵌套

2.2.4 手写 Os_Cfg.c 配置指南

【理论讲解】

在没有 OIL/ARXML 配置工具的情况下，需要手动编写 Os_Cfg.c 来实例化 OS 配置数据。以下 6 个步骤将引导你从零开始构建一个完整的 Os_Cfg.c 文件。

步骤一：定义任务入口函数声明

在 Os_Cfg.c 顶部声明所有 Task 的入口函数，使用 DeclareTask 宏：

```
/* Os_Cfg.c - Step 1: 任务声明 */
#include "Os.h"
#include "Os_Cfg.h"

DeclareTask(Task_Init);
```

```
DeclareTask(Task_Blink);
/* IdleTask 由内核自动声明，无需手动 Declare */
```

• 常见错误：忘记 DeclareTask 导致编译报 'implicit declaration' 警告

• DeclareTask 宏展开后生成 extern void Os_TaskEntry_XXX(void)，必须与 App_Tasks.c 中的 TASK(XXX) 名称一致

步骤二：定义栈空间

为每个 Task 分配静态栈数组，大小以 word (4 字节) 为单位：

```
/* Step 2: 栈空间定义 */
#define TASK_INIT_STACK_SIZE    128U    /* 128 words = 512 bytes */
#define TASK_BLINK_STACK_SIZE   128U
#define IDLE_TASK_STACK_SIZE    64U     /* 空闲任务栈可较小 */

static uint32 Task_Init_Stack[TASK_INIT_STACK_SIZE];
static uint32 Task_Blink_Stack[TASK_BLINK_STACK_SIZE];
static uint32 Idle_Task_Stack[IDLE_TASK_STACK_SIZE];
```

• 常见错误：栈分配过小导致运行时栈溢出（Trap E 类异常）

• 建议：初始分配 128 words (512B)，开启 CFG_STACK_MONITOR 后根据实际使用量调整

• 注意：若使用链接脚本分配栈段（如 .OS_STACK_Task_Init），则此处可不定义数组，而是用 extern 引用链接器符号

步骤三：填充任务配置数组 Os_TaskCfg[]

这是 Os_Cfg.c 的核心，定义每个 Task 的属性：

```
/* Step 3: 任务配置数组 */
const Os_TaskCfgType Os_TaskCfg_Core0[CFG_TASK_MAX_CORE0] = {
  /* Task_Init: 优先级 2, 非抢占, 自启动 */
  {
    Os_TaskEntry_Task_Init,   /* osTaskEntry: 入口函数 */
    0U,                       /* osTaskStackId */
    1U,                       /* osTaskActivation: 最大激活次数 */
    2U,                       /* osTaskPriority: 优先级 */
    OSDEFAULTAPPMODE,         /* osTaskAutoStartMode: 自动启动 */
    OS_TASK_NON_PREEMPTIVE,   /* osTaskSchedule: 不可抢占 */
  },
  /* Task_Blink: 优先级 1, 可抢占, 不自启 */
  {
    Os_TaskEntry_Task_Blink,  /* osTaskEntry */
    1U,                       /* osTaskStackId */
    1U,                       /* osTaskActivation */
    1U,                       /* osTaskPriority */
    0U,                       /* osTaskAutoStartMode: 手动激活 */
```

```
    OS_TASK_FULL_PREEMPTIVE,  /* osTaskSchedule: 完全抢占 */
  },
  /* IdleTask: 优先级 0, 可抢占, 全模式自启 */
  {
    Os_IdleTask_Core0,        /* osTaskEntry */
    2U,                       /* osTaskStackId */
    1U,                       /* osTaskActivation */
    0U,                       /* osTaskPriority: 最低 */
    OS_MODE_ALL,              /* osTaskAutoStartMode: 所有模式启动 */
    OS_TASK_FULL_PREEMPTIVE,  /* osTaskSchedule */
  },
};
```

• 常见错误：osTaskEntry 函数名与 DeclareTask 不匹配，导致链接时 undefined reference

• 常见错误：数组元素数与 CFG_TASK_MAX_CORE0 不一致，导致越界访问

• 注意：任务顺序必须与 Os_Cfg.h 中的 Task ID 宏定义一一对应

步骤四：配置计数器和告警

Counter 提供时间基准，Alarm 基于 Counter 触发动作：

```
/* Step 4: 计数器配置 */
const Os_CounterCfgType Os_CounterCfg_Core0[CFG_COUNTER_MAX_CORE0] = {
  {
    65535U,             /* osCounterMaxAllowedValue */
    1U,                 /* osCounterMinCycle */
    1U,                 /* osCounterTicksPerBase */
    COUNTER_HARDWARE,   /* osCounterType: 硬件计数器 */
  },
};

/* 告警配置 */
const Os_AlarmCfgType Os_AlarmCfg_Core0[CFG_ALARM_MAX_CORE0] = {
  {
    SystemCounter,      /* osAlarmCounterRef: 关联的 Counter */
    ALARM_ACTION_ACTIVATETASK, /* osAlarmAction: 激活任务 */
    Task_Blink,         /* osAlarmTaskRef: 目标 Task */
    0U,                 /* osAlarmAutoStart: 非自动启动 */
  },
};
```

• 常见错误：osCounterMaxAllowedValue 设置过小，导致 Alarm 周期超过计数器范围

• 常见错误：osAlarmCounterRef 与实际 Counter ID 不匹配，导致 Alarm 不触发

• 说明：本项目中 Alarm 通过 StartupHook 中的 SetRelAlarm() 启动，而非配置自动启动

步骤五：配置优先级队列

AUTOSAR OS 使用静态优先级队列管理就绪 Task，每个优先级对应一个队列槽位：

```
/* Step 5: 优先级队列配置 */
/* 就绪队列内存，每个优先级一个槽位 */
static Os_ReadyQueueType Os_ReadyQueue_Core0[CFG_PRIORITY_MAX_CORE0];

/* 优先级到任务的映射表 */
const Os_PriorityTableType Os_PrioTable_Core0[CFG_PRIORITY_MAX_CORE0] = {
    { 1U, &Os_ReadyQueue_Core0[0] }, /* Priority 0: IdleTask */
    { 1U, &Os_ReadyQueue_Core0[1] }, /* Priority 1: Task_Blink */
    { 1U, &Os_ReadyQueue_Core0[2] }, /* Priority 2: Task_Init */
};
```

• 常见错误：优先级槽位数与 CFG_PRIORITY_MAX_CORE0 不一致，导致调度器异常

• 注意：队列槽位的 capacity 字段表示该优先级同时就绪的最大 Task 数（BCC2 支持多次激活）

步骤六：配置 ISR

Cat2 ISR 需要在 Os_Cfg.c 中注册，以便 OS 管理其上下文：

```
/* Step 6: ISR 配置 */
const Os_IsrCfgType Os_IsrCfg_Core0[CFG_ISR_MAX_CORE0] = {
    {
        ISR_Stm0Sr0,            /* isrEntry: ISR 入口函数 */
        OS_ISR_CATEGORY_2,      /* isrCategory: Cat2 由 OS 管理 */
        11U,                    /* isrPriority: SRC 中断优先级 */
    },
};
```

• 常见错误：isrPriority 与 SRC 寄存器中的实际配置不一致，导致中断无法被 OS 正确识别

• 常见错误：将 Cat1 ISR 注册为 Cat2，导致 OS 在 ISR 中不必要地保存/恢复上下文

• 说明：ISR 入口函数通常在 Os_Intvet.c 中通过 ISR(Stm0Sr0) 宏定义

【配置文件完整性检查清单】

检查项	对应步骤	常见错误
Task 数量与 CFG_TASK_MAX_CORE0 一致	步骤三	数组越界或未使用的槽位
每个 Task 有对应栈分配	步骤二	栈指针为 NULL 导致 Trap
Counter 数量与 CFG_COUNTER_MAX_CORE0 一致	步骤四	Alarm 无法关联计数器
Alarm 的 TaskRef 引用有效 Task ID	步骤四	激活不存在的 Task
优先级槽位数与 CFG_PRIORITY_MAX_CORE0 一致	步骤五	调度器遍历越界

ISR 数量与 CFG_ISR_MAX_CORE0 一致	步骤六	中断入口未注册

提示：所有 CFG_*_MAX_CORE0 宏在 Os_Cfg.h 中定义。修改 Os_Cfg.c 时必须同步检查 Os_Cfg.h 中的宏值，确保二者一致。

2.3 集成验证

2.3.1 最小系统启动验证

【实验验证】

集成完成后，第一步是验证 OS 能正常启动并进入调度。最小系统验证的目标：

- StartOS() 能正常执行不崩溃
- StartupHook() 被正确调用
- Autostart Task 能被激活执行
- Idle Task 正常循环运行

【配置演示 - Hook 函数实现】

在 Os_UserInf.c 中实现必要的 Hook 函数：

```
/* Os_UserInf.c - StartupHook 实现 */
void StartupHook(void)
{
    /* 启动 AlarmBlink: 500 ticks 初始延迟, 500 ticks 周期 */
    (void)SetRelAlarm(AlarmBlink, 500U, 500U);
}

/* ErrorHook - 错误钩子 */
void ErrorHook(StatusType Error)
{
    (void)Error;
    /* 调试时可在此设置断点或输出日志 */
}

/* ShutdownHook - 关闭钩子 */
void ShutdownHook(StatusType Error)
{
    (void)Error;
}

/* IdleHook_Core0 - 空闲钩子 */
void IdleHook_Core0(void)
{
    /* Idle Task 体内调用，可放置省电等待指令 */
```

```
}
```

【配置演示 - 应用任务实现】

在 App/App_Tasks.c 中实现 Task 体：

```
/* App_Tasks.c - LED 闪烁演示 */
#include "Os.h"
#include "IfxPort.h"

#define LED_PORT  &MODULE_P00
#define LED_PIN   5u

/* Task_Init: 初始化任务（优先级 2, 自动启动） */
TASK(Task_Init)
{
  IfxPort_setPinMode(LED_PORT, LED_PIN,
                IfxPort_Mode_outputPushPullGeneral);
  IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_high);
  TerminateTask(); /* 必须调用! 不同于 FreeRTOS 的无限循环 */
}

/* Task_Blink: LED 翻转任务（优先级 1, 由 AlarmBlink 每 500ms 激活） */
TASK(Task_Blink)
{
  IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_toggled);
  TerminateTask();
}
```

关键差异（与 FreeRTOS 对比）：

- AUTOSAR Task 体使用 TASK() 宏定义，而非普通函数指针
- Basic Task 必须以 TerminateTask() 或 ChainTask() 结束
- 不允许在 Task 体内使用无限循环（Idle Task 除外）
- 周期性行为通过 Alarm 周期激活实现，而非 Task 内 vTaskDelay

2.3.2 验证结果分析

【实验验证】

编译并烧录后，预期行为：

时间点	事件	验证方法
T=0	core0_main() 调用 StartOS(OSDEFAULTAPPMODE)	断点确认执行到 StartOS
T+0.1ms	OS 初始化完成，调用 StartupHook()	断点确认 SetRelAlarm 执行
T+0.2ms	Task_Init 被调度执行（最高优先	LED GPIO 配置成功

	级）	
T+0.3ms	Task_Init 调用 TerminateTask()，进入 IdleTask	空闲状态稳定
T+500ms	AlarmBlink 触发，激活 Task_Blink	LED 第一次翻转
T+1000ms	AlarmBlink 再次触发	LED 第二次翻转
持续	每 500ms Task_Blink 被激活一次	LED 以 1Hz 闪烁

【常见问题排查】

症状	可能原因	解决方案
StartOS 后立即崩溃	CSA 未正确初始化	检查链接脚本 CSA 段配置
进入 Trap	栈溢出或非法地址访问	增大 Task 栈分配、检查指针
StartupHook 未调用	CFG_STARTUPHOOK 未设为 TRUE	检查 Os_Cfg.h 宏定义
Alarm 未触发	Counter 未得到 Tick 驱动	检查 STM 中断是否正确配置
LED 不闪烁	GPIO 未配置或 Task 未激活	断点确认 Task_Blink 执行
编译报错类型重复定义	Os_Types.h 与 Std_Types.h 冲突	确保 Os_Cfg.h 包含 Std_Types.h

【调试技巧】

1. 在 StartupHook() 中设置断点，确认 OS 初始化流程正常
2. 在 ErrorHook() 中设置断点，任何 OS API 错误会触发此 Hook
3. 使用 ADS 的 Trace 功能观察任务切换序列
4. 检查 CSA 链表是否完整（查看 FCX/PCX 寄存器）

【启动流程时序图】

```
core0_main()
    │
    ├── IfxCpu_enableInterrupts()
    ├── IfxScuWdt_disableCpuWatchdog()
    ├── IfxScuWdt_disableSafetyWatchdog()
    ├── IfxCpu_emitEvent() / IfxCpu_waitEvent()
    │
    └── StartOS(OSDEFAULTAPPMODE)
            │
            ├── Os_InitKernel()          // 初始化任务始化 TCB/CCB/ACB
            ├── Os_InitHardwareTimer()   // 配置 STM0
            ├── StartupHook()            // 用户钩子
            │       └── SetRelAlarm(AlarmBlink, 500, 500)
            ├── Os_ActivateAutoTasks()   // 激活 Task_Init + IdleCore0
            └── Os_StartSchedule()       // 进入调度
                    │
                    ├── Task_Init (Priority 2)  // 最高优先级先执行
                    │       ├── 配置 GPIO
                    │       └── TerminateTask()
                    │
                    └── IdleCore0 (Priority 0)  // 无其他就绪 Task
                            └── IdleHook_Core0() 循环
```

2.4 本章小结

本章介绍了将 AUTOSAR OS 集成到 TC334 工程的完整流程，包括源码结构理解、链接脚本配置、OS 配置数据实例化以及最小系统验证。与 FreeRTOS 移植相比，AUTOSAR OS 的集成复杂度更高，但带来了更强的确定性和安全保证。下一章将深入讲解 OS 系统配置参数的含义与调整方法。

第三章 AUTOSAR OS 系统配置

3.1 OS 配置概述（OsGeneralConfiguration）

【理论讲解】

AUTOSAR OS 的系统配置是通过 Os_Cfg.h 中的宏定义来完成的。每个宏定义对应 AUTOSAR SWS_OS 规范中的一个配置参数。这些配置在编译时就确定了 OS 的所有行为——这是 AUTOSAR OS 静态配置哲学的核心体现。

本项目 TC334 工程的完整配置参数概览：

配置类别	参数	当前值	说明
核心	CFG_CORE_MAX	1	单核配置
核心	OS_AUTOSAR_CORES	1	AUTOSAR 管理的核数
可扩展性	CFG_SC	OS_SC1	基础功能（无保护）
符合性	CFG_CC	OS_BCC2	基本任务+多次激活
状态	CFG_STATUS	OS_STATUS_STANDARD	标准模式
任务	CFG_TASK_MAX	3	Task_Init+Task_Blink+Idle
任务	CFG_PRIORITY_MAX_CORE0	3	3 个优先级等级
中断	CFG_ISR_MAX	1	1 个 ISR (STM0)
中断	CFG_ISR2_MAX	1	1 个 ISR Category 2
中断	CFG_INT_NEST_ENABLE	TRUE	支持中断嵌套
计数器	CFG_COUNTER_MAX	1	SystemCounter
告警	CFG_ALARM_MAX	1	AlarmBlink
调度	CFG_SCHED_POLICY	OS_PREEMPTIVE_MIXED	混合抢占
Hook	CFG_ERRORHOOK	TRUE	错误钩子启用
Hook	CFG_STARTUPHOOK	TRUE	启动钩子启用
Hook	CFG_SHUTDOWNHOOK	TRUE	关闭钩子启用
监控	CFG_STACK_MONITOR	TRUE	栈监控启用

与 FreeRTOS 的对比：FreeRTOS 通过 FreeRTOSConfig.h 中的 configXXX 宏定义系统行为，但 AUTOSAR OS 的配置远更细粒度，不仅包含系统参数，还包含每个 OS 对象的静态声明。这使得编译器能在编译期就完成完整性检查。

3.2 可扩展性类别（Scalability Class 1/2/3/4）

【理论讲解】

AUTOSAR OS 定义了 4 个可扩展性类别（Scalability Class, SC），代表不同级别的安全保护能力。类别越高，提供的保护机制越完善，但也带来更高的资源开销和配置复杂度。

【配置演示 - SC1~SC4 对比】

特性	SC1 (当前)	SC2	SC3	SC4
基础调度	√	√	√	√
Counter/Alarm	√	√	√	√
Resource (PCP)	√	√	√	√
ISR Cat 1/2	√	√	√	√
Schedule Table	√	√	√	√

时间保护 (TP)	✗	✓	✗	✓
内存保护 (MPU)	✗	✗	✓	✓
OS-Application	✗	✗	✓	✓
服务保护	✗	✗	✓	✓
典型 ASIL 等级	QM/ASIL-A	ASIL-B	ASIL-C/D	ASIL-D
硬件要求	无特殊要求	需 TP 定时器	需 MPU	需 MPU+TP 定时器
RAM 开销	低	中	高	最高

【配置演示 - 本项目 SC1 配置】

```
/* Os_Cfg.h - 可扩展性类别配置 */
#define CFG_SC                          OS_SC1

/* SC1 下禁用的功能 */
#define CFG_TIMING_PROTECTION_ENABLE    FALSE    /* SC2/SC4 才 */
#define CFG_MEMORY_PROTECTION_ENABLE    FALSE    /* SC3/SC4 才 */
#define CFG_SERVICE_PROTECTION_ENABLE   FALSE    /* SC3/SC4 才 */
#define CFG_OSAPPLICATION_MAX           0U       /* SC3/SC4 才 */
#define CFG_PROTECTIONHOOK              FALSE    /* SC2/SC3/SC4 才 */
```

SC1 选择理由：

- TC334 为单核 MCU，不需要多核隔离
- 当前为学习/原型验证阶段，无安全等级要求
- SC1 资源开销最低，适合资源受限的 TC334（192KB DSRAM）

【升级到 SC2 需要的改动】

如果需要时间保护，升级到 SC2 的关键配置改动：

```
/* 升级到 SC2 的配置改动示例 */
#define CFG_SC                              OS_SC2
#define CFG_TIMING_PROTECTION_ENABLE        TRUE
#define CFG_TIMING_PROTECTION_ENABLE_CORE0  TRUE
#define CFG_PROTECTIONHOOK                  TRUE

/* 每个 Task/ISR 需配置时间预算 */
/* 在 Os_Cfg.c 中添加 Timing Protection 配置 */
```

时间保护会监控每个 Task 和 ISR2 的执行时间、锁定时间和中断禁止时间，超时时触发 ProtectionHook。

3.3 符合性类别（BCC1/BCC2/ECC1/ECC2）

【理论讲解】

符合性类别（Conformance Class）定义了 OS 支持的 Task 类型和调度能力。这是从 OSEK/VDX 继承而来的概念，用于定义 OS 实现的功能边界。

【配置演示 - BCC/ECC 对比表】

特性	BCC1	BCC2 (当前)	ECC1	ECC2
任务类型	Basic Task	Basic Task	Basic + Extended	Basic + Extended
多次激活	✗ (单次)	✓ (多次)	✗ (单次)	✓ (多次)
同优先级多 Task	✗	✓	✗	✓
Event 机制	✗	✗	✓	✓
WaitEvent/SetEvent	✗	✗	✓	✓
每任务独立栈	可共享	可共享	必须独立	必须独立
典型应用	简单轮询系统	周期任务系统	事件驱动系统	复杂事件系统

【配置演示 - 本项目 BCC2 配置】

```
/* Os_Cfg.h - 符合性类别配置 */
#define CFG_CC                          OS_BCC2
#define CFG_EXTENDED_TASK_MAX           (0U)    /* 无扩展任务 */
#define CFG_EXTENDED_TASK_MAX_CORE0     (0U)

/* BCC2 允许多次激活，激活队列大小配置 */
/* Os_Cfg.c 中的激活队列定义 */
static Os_TaskType Os_ActivateQueue_Core0_0[1];    /* Idle: 单 */
static Os_TaskType Os_ActivateQueue_Core0_1[2];    /* Task_Blink: 最 2 次 */
static Os_TaskType Os_ActivateQueue_Core0_2[2];    /* Task_Init: 最 2 次 */
```

BCC2 选择理由：

• LED 闪烁场景不需要 Event 等待机制

• 多次激活允许 Alarm 在上一次 Task_Blink 未完成时再次激活

• Basic Task 不需要独立栈（理论上可共享），节省 RAM

【与 FreeRTOS 的对比】

FreeRTOS 中所有 Task 都是 Extended Task 的等价（每个 Task 都有独立栈，可以 Block）。AUTOSAR OS 将 Task 分为两类：

• Basic Task：不能调用 WaitEvent()，类似一个一次性执行的函数

• Extended Task：可以调用 WaitEvent() 进入等待状态，类似 FreeRTOS 的常规 Task

在 FreeRTOS 中，你可能习惯于在 Task 中使用 xQueueReceive() 或 ulTaskNotifyTake() 等待事件。在 AUTOSAR OS 的 BCC2 中，这种模式无法使用——Task 必须执行完毕就结束。如果需要事件等待，必须升级到 ECC1/ECC2。

【升级到 ECC2 的配置示例】

```
/* 升级到 ECC2 的配置改动 */
#define CFG_CC                          OS_ECC2
```

```
#define CFG_EXTENDED_TASK_MAX           (1U)
#define CFG_EXTENDED_TASK_MAX_CORE0     (1U)

/* 定义 Event */
#define Evt_DataReady    ((Os_EventMaskType)0x01U)

/* Extended Task 示例 */
TASK(Task_DataProc)
{
  for(;;) {
    WaitEvent(Evt_DataReady);  /* 等待事件 */
    ClearEvent(Evt_DataReady);
    /* 处理数据... */
  }
  /* Extended Task 可以使用无限循环 */
}
```

3.4 OS 配置参数详解

3.4.1 OsNumberOfCores

【理论讲解】

定义 AUTOSAR OS 管理的 CPU 核心数量。在多核 MCU（如 TC397（6 核））中，可能只有部分核运行 AUTOSAR OS。

【配置演示】

```
/* Os_Cfg.h - 核心配置 */
#define CFG_CORE_MAX                    (1U)                /* 物 */
#define OS_AUTOSAR_CORES                1U                  /* AUTOSAR OS 管 */
#define OS_CORE_ID_MASTER               ((Os_CoreIdType)0U)  /* 主 */
#define OS_CORE_ID_0                    ((Os_CoreIdType)0U)  /* Core0 ID */
#define CFG_CORE0_AUTOSAROS_ENABLE      TRUE                /* Core0 启 OS */
```

TC334 是单核 MCU，因此 CFG_CORE_MAX = 1。如果迁移到 TC397（6 核），需要：

- 将 CFG_CORE_MAX 修改为实际使用的核数
- 为每个核配置对应的 Task、ISR、Counter 等
- 配置核间通信机制（Spinlock、IOC）

【API 详解 - 多核相关 API】

- GetCoreID() — 获取当前核 ID
- StartNonAutosarCore(CoreID) — 启动非 AUTOSAR 核
- GetSpinlock(SpinlockID) — 获取跨核自旋锁

• ReleaseSpinlock(SpinlockID) — 释放自旋锁

3.4.2 OsStackMonitoring

【理论讲解】

栈监控是 AUTOSAR OS 的重要安全特性，用于检测 Task 和 ISR 的栈溢出。与 FreeRTOS 的 configCHECK_FOR_STACK_OVERFLOW 类似，但 AUTOSAR OS 提供更精细的控制。

【配置演示】

```
/* Os_Cfg.h - 栈监控配置 */
#define CFG_STACK_CHECK                  FALSE    /* 运(开) */
#define CFG_STACK_MONITOR                TRUE     /* 栈 */
```

CFG_STACK_MONITOR 的工作原理：

1. OS 初始化时，将每个 Task 栈空间填充特定模式（如 0xDEADBEEF）
2. 在任务切换时检查栈底的 Magic Word 是否被破坏
3. 如果检测到溢出，触发 ProtectionHook（SC2+）或 ErrorHook（SC1）

与 FreeRTOS 对比：

特性	FreeRTOS	AUTOSAR OS
栈溢出检测	configCHECK_FOR_STACK_OVERFLOW = 1/2	CFG_STACK_MONITOR = TRUE
检测时机	任务切换时	任务切换时
检测方式	栈顶指针/水印模式	栈底 Magic Word
溢出处理	调用回调函数	触发 Hook + 可配置响应

【实验验证 - 栈分配实例】

本项目的 Task 栈分配：

```
/* Os_Cfg.c - Task 栈分配 */
static Os_StackDataType Os_TaskStack_Task_Init[128];  /* 128x4=512 字节 */
static Os_StackDataType Os_TaskStack_Task_Blink[128]; /* 128x4=512 字节 */
static Os_StackDataType Os_TaskStack_Idle_Core0[64];  /* 64x4=256 字节 */
static Os_StackDataType Os_SysStack_Core0[256];       /* 系统栈: 1KB */
static Os_StackDataType Os_SysTimer_Stack_Core0[256]; /* ISR 栈: 1KB */
```

栈大小设计原则：

• Task 栈需容纳最深函数调用链的局部变量

• 预留 20%~30% 余量防止溢出

• ISR 栈需考虑中断嵌套深度

• TriCore CSA 机制已处理了寄存器保存，栈主要用于局部变量

3.4.3 OsProtectionHook

【理论讲解】

ProtectionHook 是 AUTOSAR OS SC2/SC3/SC4 提供的保护机制回调。当发生以下违规时，OS 会调用 ProtectionHook：

- Task/ISR 执行时间超过时间预算（SC2/SC4）
- Task/ISR 访问未授权内存区域（SC3/SC4）
- Task/ISR 调用未授权的 OS 服务（SC3/SC4）
- 栈溢出检测（当 CFG_STACK_MONITOR=TRUE）

【配置演示】

```
/* Os_Cfg.h - ProtectionHook 配置 */
#define CFG_PROTECTIONHOOK              FALSE    /* SC1 下 */

/* 当升级到 SC2+ 时启用 */
/* #define CFG_PROTECTIONHOOK           TRUE */

/* ProtectionHook 实现示例 (SC2+) */
ProtectionReturnType ProtectionHook(StatusType FatalError)
{
  switch(FatalError) {
    case E_OS_PROTECTION_TIME:
      /* 时间保护违规: 终止当前 Task */
      return PRO_TERMINATETASKISR;
    case E_OS_PROTECTION_MEMORY:
      /* 内存保护违规: 关闭对应 OS-Application */
      return PRO_TERMINATEAPPL;
    case E_OS_STACKFAULT:
      /* 栈溢出: 关闭系统 */
      return PRO_SHUTDOWN;
    default:
      return PRO_SHUTDOWN;
  }
}
```

【API 详解 - ProtectionHook 返回值】

返回值	含义	使用场景
PRO_IGNORE	忽略违规，继续执行	仅调试用
PRO_TERMINATETASKISR	终止当前 Task/ISR	时间超时
PRO_TERMINATEAPPL	终止整个 OS-Application	内存违规
PRO_TERMINATEAPPL_RESTART	终止并重启 OS-Application	可恢复故障
PRO_SHUTDOWN	关闭整个 OS	严重错误

3.4.4 其他关键配置宏

【配置演示 - 调度策略】

```
/* 调度策略配置 */
#define CFG_SCHED_POLICY    OS_PREEMPTIVE_MIXED

/* 三种调度策略: */
/* OS_PREEMPTIVE_FULL   - 完全抢占 (类似 FreeRTOS 默认) */
/* OS_PREEMPTIVE_NON    - 完全不可抢占 */
/* OS_PREEMPTIVE_MIXED  - 混合模式 (每个 Task 单独配置) */
```

本项目使用 MIXED 模式：

- Task_Init: OS_PREEMPTIVE_NON（初始化时不可抢占，确保原子性）
- Task_Blink: OS_PREEMPTIVE_FULL（可被更高优先级任务抢占）
- IdleCore0: OS_PREEMPTIVE_FULL（可被任何任务抢占）

【配置演示 - 系统定时器】

```
/* 系统定时器配置 */
#define CFG_SYSTEM_TIMER_ENABLE             TRUE
#define CFG_SYSTEM_TIMER_ENABLE_CORE0       TRUE
#define CFG_REG_OSTIMER_VALUE_CORE0         (100000U)    /* STM 比 */

/* TC334 的 STM 时钟频率 = 100MHz */
/* 100000 / 100MHz = 1ms 节拍 */
/* 即 SystemCounter 每 1ms 递增一次 */
```

Counter 配置与 Alarm 的关系：

- SystemCounter: 1ms 节拍，最大值 2147483647
- AlarmBlink: 基于 SystemCounter，500 ticks = 500ms
- 因此 LED 每 500ms 翻转一次，周期 = 1s，频率 = 1Hz

【配置演示 - Hook 配置总览】

```
/* Hook 配置 */
#define CFG_ERRORHOOK                       TRUE     /* 任 OS API 报 */
#define CFG_PRETASKHOOK                     FALSE    /* Task 切 */
#define CFG_POSTTASKHOOK                    FALSE    /* Task 切 */
#define CFG_STARTUPHOOK                     TRUE     /* StartOS()后 */
#define CFG_SHUTDOWNHOOK                    TRUE     /* ShutdownOS()时 */
#define CFG_PROTECTIONHOOK                  FALSE    /* 保(SC2+) */
```

Hook 的作用总结：

Hook	触发时机	典型用途
StartupHook	OS 启动后、调度前	初始化 Alarm、配置外设
ErrorHook	OS API 返回错误时	错误日志、调试断点
ShutdownHook	ShutdownOS()调用时	资源清理、错误记录
PreTaskHook	任务切入前	Trace、运行时分析
PostTaskHook	任务切出后	Trace、CPU 负载统计
ProtectionHook	保护违规时	故障处理决策

【配置演示 - 系统状态模式】

```
/* 系统状态模式 */
#define CFG_STATUS    OS_STATUS_STANDARD

/* 两种模式: */
/* OS_STATUS_STANDARD  - 生产模式，减少运行时检查 */
/* OS_STATUS_EXTENDED  - 开发模式，完整参数检查 */
```

在 EXTENDED 模式下，每个 OS API 调用都会检查：

• 参数有效性（Task ID 是否合法、Alarm 是否存在）

• 调用上下文是否合法（是否在 ISR 中调用了仅 Task 可用的 API）

• 状态一致性（是否重复获取 Resource）

建议：开发阶段使用 EXTENDED，量产时切换为 STANDARD 以减少运行时开销。

【实验验证 - 配置修改实践】

实验 3-1：将 CFG_STATUS 从 STANDARD 改为 EXTENDED，然后故意向 ActivateTask() 传入非法 Task ID，观察是否触发 ErrorHook。

```
/* 实验 3-1: 验证 EXTENDED 模式的参数检查 */
/* 步骤 1: 修改 Os_Cfg.h */
#define CFG_STATUS    OS_STATUS_EXTENDED

/* 步骤 2: 在 Task_Init 中添加测试代码 */
TASK(Task_Init)
{
  StatusType ret;
  /* 故意传入非法 Task ID (999) */
  ret = ActivateTask((Os_TaskType)999U);
  /* EXTENDED 模式下，返回 E_OS_ID，并触发 ErrorHook */
  /* STANDARD 模式下，行为未定义 */
  TerminateTask();
}
```

实验 3-2：修改 Alarm 周期，观察 LED 闪烁频率变化

```
/* 实验 3-2: 修改 AlarmBlink 周期 */
/* 在 StartupHook() 中修改参数 */
void StartupHook(void)
{
   /* 原始: 500ms 周期 -> LED 1Hz 闪烁 */
   /* (void)SetRelAlarm(AlarmBlink, 500U, 500U); */

   /* 修改: 100ms 周期 -> LED 5Hz 闪烁 */
   (void)SetRelAlarm(AlarmBlink, 100U, 100U);
}
```

实验 3-3：禁用 CFG_STARTUPHOOK，观察系统行为

```
/* 实验 3-3: 禁用 StartupHook */
#define CFG_STARTUPHOOK    FALSE

/* 结果: StartupHook 不会被调用 */
/* AlarmBlink 不会被启动 -> Task_Blink 永远不会被激活 */
/* LED 不会闪烁，系统停留在 IdleTask */
```

3.5 本章小结

本章详细解析了 AUTOSAR OS 的核心配置参数，包括可扩展性类别（SC1~SC4）、符合性类别（BCC/ECC）、核配置、栈监控、保护机制和调度策略。这些配置共同决定了 OS 的运行行为和安全保护级别。对于 FreeRTOS 工程师而言，最重要的理解是：AUTOSAR OS 通过细粒度的静态配置实现了对系统行为的完全控制，这是其实现功能安全认证的基础。

第四章 中断管理

4.1 AUTOSAR OS 中断模型

AUTOSAR OS 将中断分为两大类（Category 1 和 Category 2），这一分类决定了 OS 对 ISR 的管理程度。理解这两类中断是掌握 AUTOSAR OS 实时行为的基础。

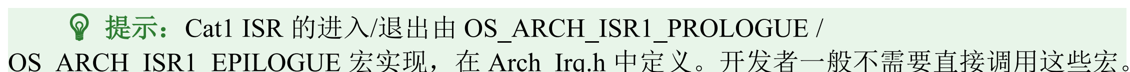
在本项目（TC334 单核移植）中，系统定时器 STM0 使用 Category 2 中断（OS_ISR_CATEGORY2），由 OS 完全管理其上下文保存与恢复。

4.1.1 Category 1 ISR（OS 不管理）

Category 1 ISR（Cat1 ISR）是最轻量的中断处理方式：OS 不参与其调度与上下文管理。

- 不调用任何 OS 服务（ActivateTask、SetEvent 等均禁止）
- 不触发重调度——ISR 返回后直接恢复被打断的上下文
- OS 内部嵌套计数器 Os_IntNestISR1 用于跟踪 Cat1 嵌套深度
- 进入 Cat1 ISR 时，OS 仅切换到系统栈（Os_ArchSwitch2System）

适用场景：对延迟极度敏感、不需要与任务交互的硬件中断（例如高速 ADC 采样、编码器脉冲计数）。



提示：Cat1 ISR 的进入/退出由 OS_ARCH_ISR1_PROLOGUE / OS_ARCH_ISR1_EPILOGUE 宏实现，在 Arch_Irq.h 中定义。开发者一般不需要直接调用这些宏。

4.1.2 Category 2 ISR（OS 管理）

Category 2 ISR（Cat2 ISR）是 AUTOSAR OS 完全管理的中断：OS 负责上下文保存/恢复、嵌套控制，并在 ISR 退出时触发重调度。

- 可调用 OS 服务：ActivateTask()、SetEvent()、GetResource() 等
- OS 保存当前任务的 PCX（Previous Context）到 Os_TaskCBExt
- 切换到 ISR 专用栈（Os_ArchSwitch2ISR2Stk）
- 退出时若有更高优先级任务就绪，OS 执行重调度（OS_ARCH_RESUME_CONTEXT）
- 嵌套计数器 Os_IntNestISR2 跟踪 Cat2 嵌套深度

在本项目中，系统定时器 ISR 配置如下：

配置项	值	说明
OsIsrSrc	OS_ISR_STM0_SR0	STM0 比较寄存器 0 中断源
OsIsrSrcType	OS_ARCH_INT_CPU0	绑定到 CPU0
OsIsrCatType	OS_ISR_CATEGORY2	Cat2，OS 完全管理
OsNestedEnable	FALSE	该 ISR 不允许被嵌套

4.1.3 ISR 与 Task 的关系

在 AUTOSAR OS 中，ISR 的优先级始终高于任何 Task。即使系统中最高优先级的 Task 正在运行，任何已使能的中断（Cat1 或 Cat2）发生时都会立即抢占 Task。

- Task 优先级范围：0（最低，Idle）~ CFG_PRIORITY_MAX_CORE0-1
- ISR2 使用硬件中断优先级（IPL），其值域与 Task 优先级独立
- Cat2 ISR 退出时会检查就绪队列，可能触发 Task 切换
- Cat1 ISR 退出后直接返回被打断的 Task，不触发调度

与 FreeRTOS 对比：ISR 与 Task 关系

特性	FreeRTOS	AUTOSAR OS
ISR 中	必须使用 FromISR 后缀 API (xTaskNotifyFromISR 等)	Cat2 ISR 可直接调用

调用 OS		ActivateTask/SetEvent 等
ISR 优先级域	configMAX_SYSCALL_INTERRUPT_PRIORITY 以下可用 OS API	CFG_ISR2_IPL_MAX_CORE0 定义 Cat2 优先级天花板
ISR 分类	不区分，统一处理	Cat1（OS 不管理） Cat2（OS 管理）

表：Cat1 ISR vs Cat2 ISR 对 OS 服务 API 的调用限制（依据 SWS_Os_00075）

API	Task	Cat1 ISR	Cat2 ISR	Hook
ActivateTask	✓	✗	✓	✗
TerminateTask	✓	✗	✗	✗
ChainTask	✓	✗	✗	✗
Schedule	✓	✗	✗	✗
GetTaskID	✓	✗	✓	✓
GetTaskState	✓	✗	✓	✓
SetEvent	✓	✗	✓	✗
ClearEvent	✓	✗	✗	✗
WaitEvent	✓	✗	✗	✗
GetEvent	✓	✗	✓	✓
GetResource	✓	✗	✓	✗
ReleaseResource	✓	✗	✓	✗
SetRelAlarm	✓	✗	✓	✗
SetAbsAlarm	✓	✗	✓	✗
CancelAlarm	✓	✗	✓	✗
GetAlarm	✓	✗	✓	✗
GetAlarmBase	✓	✗	✓	✓
GetCounterValue	✓	✗	✓	✗
GetElapsedValue	✓	✗	✓	✗
DisableAllInterrupts	✓	✓	✓	✓
EnableAllInterrupts	✓	✓	✓	✓
SuspendAllInterrupts	✓	✓	✓	✓
ResumeAllInterrupts	✓	✓	✓	✓
SuspendOSInterrupts	✓	✗	✓	✓
ResumeOSInterrupts	✓	✗	✓	✓
GetISRID	✓	✗	✓	✓
GetSpinlock	✓	✗	✓	✗
ReleaseSpinlock	✓	✗	✓	✗
ShutdownOS	✓	✗	✓	✓

关键规则（SWS_Os_00075）：

- Cat1 ISR 不受 OS 管理，仅允许调用 Disable/Enable/Suspend/ResumeAllInterrupts
- Cat2 ISR 由 OS 管理，可调用大部分 OS 服务（除 TerminateTask/ChainTask/Schedule/WaitEvent/ClearEvent）
- ErrorHook/StartupHook/ShutdownHook 中可调用的 API 受限（见附录 A）

4.2 中断优先级与嵌套

4.2.1 硬件中断优先级配置

TriCore 架构的中断优先级通过 SRC（Service Request Control）寄存器配置。每个中断源有一个

8-bit 优先级号（SRPN），值越大优先级越高。

- SRPN = 0 表示中断禁止
- TriCore TC334 支持最多 255 个中断优先级
- OS 系统定时器 STM0 的中断优先级由 OS 在初始化时自动配置

在 Os_Cfg.h 中，与 ISR 优先级相关的关键宏定义：

```
#define CFG_ISR_MAX                 (1U)    // 总任务数 ISR 数量
#define CFG_ISR2_MAX                (1U)    // Cat2 ISR 数量
#define CFG_ISR2_IPL_MAX_CORE0      (2U)    // Core0 Cat2 最高 IPL
#define CFG_INT_NEST_ENABLE         TRUE    // 启用中断嵌套
```

4.2.2 OS 中断优先级天花板协议

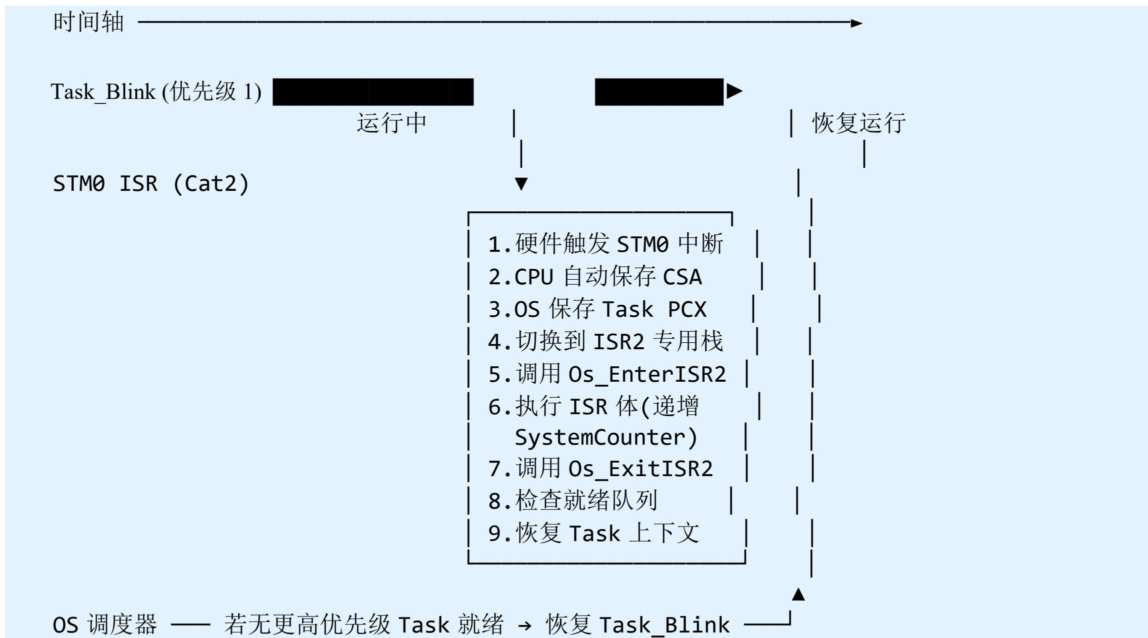
AUTOSAR OS 使用优先级天花板协议（Priority Ceiling Protocol, PCP）来管理中断资源的访问。CFG_ISR2_IPL_MAX_CORE0 定义了 Cat2 ISR 的最高中断优先级等级（IPL）。

- 当 Task 或低优先级 ISR2 获取中断资源时，OS 将当前 IPL 提升到资源的天花板优先级
- 这确保了持有资源的执行体不会被需要相同资源的更高优先级 ISR2 抢占
- 资源释放后，IPL 恢复到之前的值

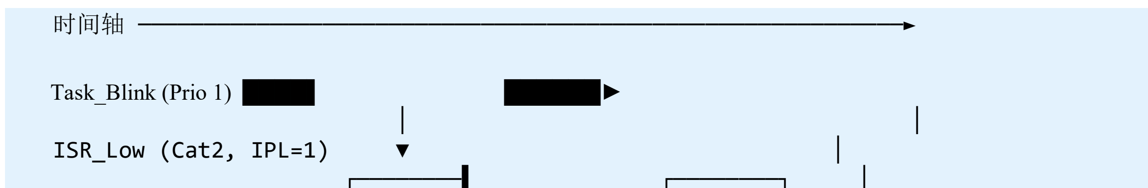
类似于 FreeRTOS 中的 configMAX_SYSCALL_INTERRUPT_PRIORITY，但 AUTOSAR OS 的机制更加细粒度——每个 Resource 可以有自己的天花板优先级。

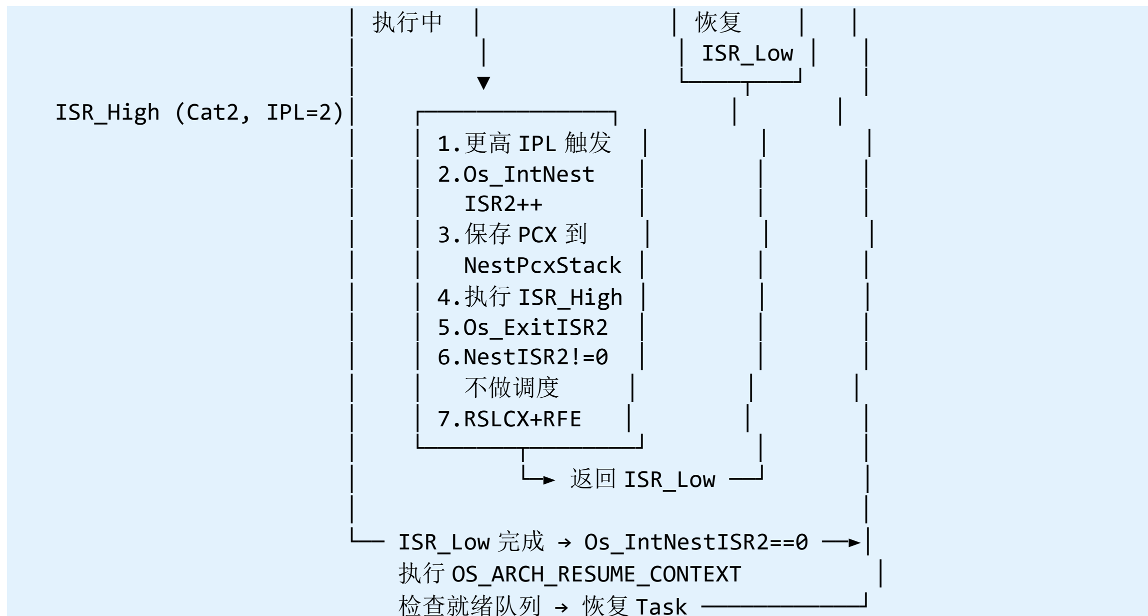
⚠ **注意：**优先级高于 CFG_ISR2_IPL_MAX_CORE0 的中断不受 OS 管理（视为 Cat1），其中不能调用任何 OS 服务。

【时序图】场景 1：Cat2 ISR 抢占 Task 执行



【时序图】场景 2：中断嵌套（高优先级 Cat2 抢占低优先级 Cat2）





4.3 开关中断 API

AUTOSAR OS 提供三组开关中断 API，适用于不同场景。理解它们的区别对于正确实现临界区至关重要。

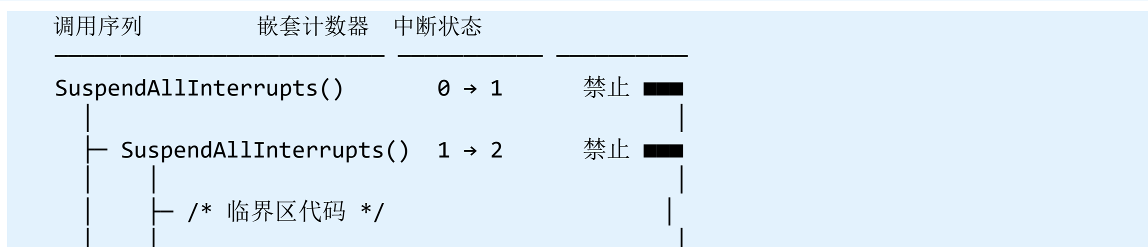
4.3.1 SuspendAllInterrupts / ResumeAllInterrupts（挂起/恢复所有中断）

挂起和恢复所有中断（包括 Cat1 和 Cat2）。支持嵌套调用。

属性	说明
函数原型	void SuspendAllInterrupts(void) void ResumeAllInterrupts(void)
参数	无
返回值	无
嵌套支持	是 — OS 内部维护嵌套计数器 (Os_SuspendAllCount)
影响范围	所有中断（Cat1 + Cat2 + 非 OS 管理中断）
适用场景	需要绝对原子操作的临界区：访问硬件寄存器、多核共享变量
可调用上下文	Task、Cat2 ISR、ErrorHook、各类 Hook

```
void MyFunction(void) {
    SuspendAllInterrupts();     // 嵌套层级 1
    /* 临界区操作... */
    SuspendAllInterrupts();     // 嵌套层级 2
    /* 更深层临界区... */
    ResumeAllInterrupts();      // 嵌套层级回到 1
    ResumeAllInterrupts();      // 嵌套层级回到 0，中断恢复
}
```

【时序图】场景 3：SuspendAllInterrupts / ResumeAllInterrupts 嵌套计数机制



```
    │    └ ResumeAllInterrupts() 2 → 1    禁止 ███
    │                                          │
    ├ /* 外层临界区代码 */                      │
    │                                          │
    └ ResumeAllInterrupts()  1 → 0      使能 ░░░

关键规则:
 · Suspend 和 Resume 必须严格配对
 · 仅当计数器回到 0 时，中断才真正恢复
 · 配对不匹配会导致中断永久关闭（系统挂死）
 · 或中断过早开启（临界区保护失效）
```

⚠ 注意：SuspendAllInterrupts/ResumeAllInterrupts 必须严格配对。嵌套不匹配将导致中断永久关闭（系统死锁）或临界区保护失效。

4.3.2 SuspendOSInterrupts / ResumeOSInterrupts（挂起/恢复 OS 中断）

仅挂起/恢复 Cat2 中断，Cat1 中断不受影响。同样支持嵌套。

属性	说明
函数原型	void SuspendOSInterrupts(void) void ResumeOSInterrupts(void)
参数	无
返回值	无
嵌套支持	是
影响范围	仅 Cat2 中断
适用场景	保护 Task 与 Cat2 ISR 之间的共享数据 同时允许 Cat1 ISR 继续响应
实现机制	通过 Os_ArchSetIpl() 将 IPL 提升到 Cat2 最高优先级

💡 提示：如果你的临界区只需要防止 Cat2 ISR 的干扰，应优先使用 SuspendOSInterrupts 而非 SuspendAllInterrupts，以减少对系统实时性的影响。

4.3.3 DisableAllInterrupts / EnableAllInterrupts（禁用/启用所有中断）

最简单的开关中断 API，不支持嵌套。直接操作 CPU 全局中断使能位。

属性	说明
函数原型	void DisableAllInterrupts(void) void EnableAllInterrupts(void)
参数	无
返回值	无
嵌套支持	否 — 不能嵌套调用
影响范围	所有中断
适用场景	极短的临界区（几条指令） 不存在嵌套调用的简单场景
底层实现	Os_ArchDisableInt() / Os_ArchEnableInt() 即 _disable() / _enable()

与 FreeRTOS 对比：临界区保护

特性	FreeRTOS	AUTOSAR OS
进入临界区	taskENTER_CRITICAL() (关中断,支持嵌套)	SuspendAllInterrupts() (关全部,支持嵌套)
ISR 安全临界区	taskENTER_CRITICAL_FROM_ISR()	SuspendOSInterrupts() (仅关 Cat2)
简单关中断	portDISABLE_INTERRUPTS()	DisableAllInterrupts() (不支持嵌套)

三组 API 的选择指南：

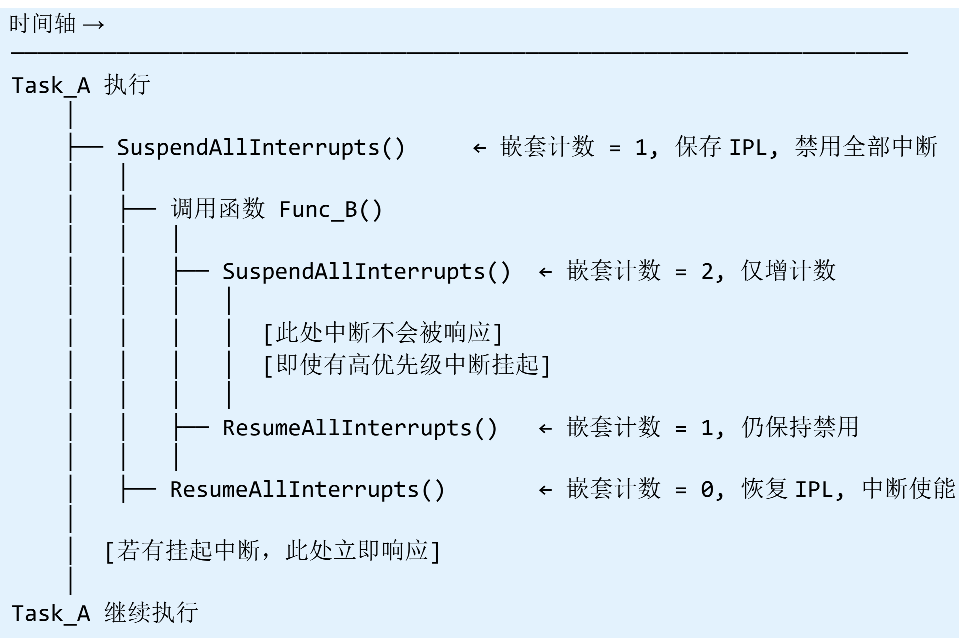
API 组	嵌套	影响范围	推荐场景
DisableAll/EnableAll	否	全部中断	极短临界区（<10 条指令）
SuspendAll/ResumeAll	是	全部中断	需要嵌套的全局临界区

SuspendOS/ResumeOS	是	仅 Cat2	保护 Task-ISR2 共享数据

4.3.4 ISR 嵌套计数机制的完整时序图

以下时序图展示了 SuspendAllInterrupts 在多层函数调用中嵌套使用时的完整行为：

【时序图】SuspendAllInterrupts 嵌套层级跟踪



关键规则：
1. SuspendAllInterrupts/ResumeAllInterrupts 必须严格配对
2. 嵌套调用安全：内层调用仅增减计数，不操作硬件
3. 仅当计数归零时才真正恢复中断状态
4. SuspendOSInterrupts 使用独立计数器，两者互不影响
5. DisableAllInterrupts/EnableAllInterrupts 不支持嵌套！

中断管理 API 使用规则总结

```
┌──────────────────────────────────────────────┐
│ 中断管理 API 使用规则                          │
├──────────────────────────────────────────────┤
│ 1. Disable/Enable: 不可嵌套，最简单的临界区     │
│ 2. Suspend/Resume All: 可嵌套，保护所有中断资源 │
│ 3. Suspend/Resume OS:  可嵌套，仅保护 Cat2 中断 │
│ 4. Cat1 ISR 在 SuspendOS 期间仍可执行          │
│ 5. 嵌套计数由 OS 内部维护，用户不可访问          │
│ 6. 调用顺序必须符合 LIFO（后进先出）            │
└──────────────────────────────────────────────┘
```

⚠ **注意：** 违反 LIFO 调用顺序（例如先 SuspendAll 再 SuspendOS 再 ResumeAll）属于未定义行为，可能导致中断状态混乱。务必确保每对 Suspend/Resume 按 LIFO 顺序配对。

4.3.5 单个中断源控制 API

AUTOSAR OS 还提供了对单个中断源进行使能/禁止/清除挂起的 API：

EnableInterruptSource

属性	说明
函数原型	StatusType EnableInterruptSource(ISRType ISRID, boolean ClearPending)
参数 ISRID	ISR 标识符（在 Os_Cfg.h 中定义）
参数 ClearPending	TRUE = 使能前清除挂起位 FALSE = 保留挂起状态
返回值	E_OK：成功 E_OS_ID：无效 ISRID E_OS_NOFUNC：ISR 已使能
适用场景	运行时动态使能特定中断源

DisableInterruptSource

属性	说明
函数原型	StatusType DisableInterruptSource(ISRType ISRID)
参数 ISRID	ISR 标识符
返回值	E_OK：成功 E_OS_ID：无效 ISRID E_OS_NOFUNC：ISR 已禁止
适用场景	运行时动态禁止特定中断源

ClearPendingInterrupt

属性	说明
函数原型	StatusType ClearPendingInterrupt(ISRType ISRID)
参数 ISRID	ISR 标识符
返回值	E_OK：成功 E_OS_ID：无效 ISRID E_OS_NOFUNC：无挂起中断
适用场景	在使能中断前清除历史挂起请求

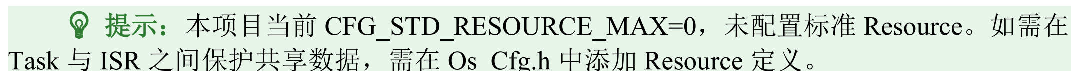
4.4 临界区保护

4.4.1 GetResource / ReleaseResource（中断资源）

AUTOSAR OS 的 Resource 机制使用优先级天花板协议（PCP）来防止优先级反转。当用于中断保护时，GetResource 会将当前 IPL 提升到资源的天花板优先级。

属性	说明
函数原型	StatusType GetResource(ResourceType ResID) StatusType ReleaseResource(ResourceType ResID)
参数 ResID	资源标识符（静态配置于 Os_Cfg.h）
GetResource 返回值	E_OK：成功获取 E_OS_ID：无效 ResID E_OS_ACCESS：权限不足
ReleaseResource 返回值	E_OK：成功释放 E_OS_ID：无效 ResID E_OS_NOFUNC：未持有该资源 E_OS_ACCESS：释放顺序错误（LIFO）
注意事项	· 必须按 LIFO 顺序释放（后获取先释放） · 不能在 ISR 退出前仍持有资源 · 本项目 CFG_STD_RESOURCE_MAX=0，未配置标准资源

```
/* 使用 Resource 保护共享数据示例 */
TASK(Task_A) {
    GetResource(ResShared);
    /* 访问共享数据 —— 此时 IPL 被提升 */
    /* 低于天花板优先级的 ISR2 被阻止 */
    ReleaseResource(ResShared);
    TerminateTask();
}
```



提示：本项目当前 CFG_STD_RESOURCE_MAX=0，未配置标准 Resource。如需在 Task 与 ISR 之间保护共享数据，需在 Os_Cfg.h 中添加 Resource 定义。

4.4.2 Spinlock（多核场景）

Spinlock 是 AUTOSAR OS 为多核系统提供的硬件级互斥机制。在单核场景（如本项目 CFG_CORE_MAX=1）下通常不使用。

- CFG_SPINLOCK_MAX=0：本项目未配置 Spinlock
- Spinlock 使用忙等待（busy-wait），仅适用于极短的多核临界区
- API: GetSpinlock() / ReleaseSpinlock() / TryToGetSpinlock()
- 嵌套获取必须按照全局配置的顺序，否则可能导致死锁

> ⚠ **注意**：Spinlock 仅在多核 AUTOSAR OS 中使用。单核项目应使用 SuspendAllInterrupts 或 GetResource 代替。

4.5 中断配置实验

4.5.1 实验设计

本实验演示 Cat2 ISR 与 Task 的交互：通过 STM0 定时中断触发 Alarm，Alarm 回调中激活 Task_Blink，实现 LED 500ms 闪烁。同时演示 SuspendAllInterrupts 的使用。

实验目标

- 理解 Cat2 ISR 的配置和行为
- 观察 ISR → Alarm → ActivateTask 的触发链
- 验证 SuspendAllInterrupts / ResumeAllInterrupts 的嵌套机制

Os_Cfg.h 关键配置

```
/* ISR 配置 */
#define CFG_ISR_MAX              (1U)
#define CFG_ISR2_MAX             (1U)
#define CFG_INT_NEST_ENABLE      TRUE
#define CFG_ISR2_IPL_MAX_CORE0   (2U)

/* 系统定时器 */
#define CFG_SYSTEM_TIMER_ENABLE        TRUE
#define CFG_SYSTEM_TIMER_ENABLE_CORE0  TRUE
#define CFG_REG_OSTIMER_VALUE_CORE0    (100000U)    // 1ms @ 100MHz
```

Os_Cfg.c ISR 配置数据

```
static const Os_IsrCfgType Os_IsrCfgCore0[CFG_ISR_MAX_CORE0] = {
    {
        OS_ISR_STM0_SR0,      /* OsIsrSrc — STM0 比较匹配中断 */
        OS_ARCH_INT_CPU0,     /* OsIsrSrcType — 绑定 CPU0 */
        OS_ISR_CATEGORY2,     /* OsIsrCatType — Cat2 */
        FALSE,                /* OsNestedEnable — 不允许被嵌套 */
    },
};
```

Alarm 配置（周期触发 Task_Blink）

```
/* Os_Cfg.c 中的 Alarm 回调 */
static void AlarmCallback_AlarmBlink(void) {
    (void)ActivateTask(Task_Blink);
}

/* StartupHook() (Os_UserInf.c) 中启动 Alarm */
```

```
SetRelAlarm(AlarmBlink, 500, 500);  // 500ms 后首次触发，周期 500ms
```

实验代码：SuspendAllInterrupts 嵌套演示

```
#include "Os.h"
#include "IfxPort.h"

#define LED_PORT  &MODULE_P00
#define LED_PIN   5u

/* 共享变量，被 Task 和 ISR 间接访问 */
static volatile uint32 g_isrCount = 0;

TASK(Task_Blink) {
    uint32 localCount;

    /* 嵌套关中断演示 */
    SuspendAllInterrupts();        /* 嵌套层级: 0→1 */
    {
        SuspendAllInterrupts();    /* 嵌套层级: 1→2 */
        localCount = g_isrCount;   /* 安全读取 */
        ResumeAllInterrupts();     /* 嵌套层级: 2→1 */
    }
    /* 此处中断仍然关闭 (层级=1) */
    IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_toggled);
    ResumeAllInterrupts();         /* 嵌套层级: 1→0, 中断恢复 */

    TerminateTask();
}
```

4.5.2 运行结果分析

将程序下载到 TC334 Lite Kit 后，可以观察到以下行为：

- LED（P00.5）每 500ms 切换一次状态，表明 Alarm 周期触发正常
- SuspendAllInterrupts 嵌套期间，STM0 中断被延迟——中断不丢失，但被推迟到 ResumeAllInterrupts 后执行
- 因为临界区非常短（仅几条指令），对系统时基精度的影响可忽略

调试验证方法：

- 在调试器中设置断点于 Os_ArchSystemTimerCore0()，观察 ISR 触发频率
- 在 SuspendAllInterrupts() 前后分别读取 STM0 计数器，验证关中断时间
- 检查 Os_IntNestISR2 变量，确认嵌套计数正确归零

实验执行流程：

```
系统启动
  → StartOS(OSDEFAULTAPPMODE)
  → Task_Init 自启动 (优先级 2, 初始化 GPIO)
  → StartupHook() 调用 SetRelAlarm(AlarmBlink, 500, 500)
  → Task_Init 调用 TerminateTask()
  → OS_TASK_IDLE_CORE0 运行 (优先级 0)
  → [每 1ms] STM0 Cat2 ISR 触发 → IncrementHardCounter(SystemCounter)
  → [每 500ms] AlarmBlink 到期 → AlarmCallback_AlarmBlink()
                               → ActivateTask(Task_Blink)
  → Task_Blink 运行 → LED 翻转 → TerminateTask()
  → 返回 Idle Task
```

```
→ 循环...
```

4.6 本章小结

本章系统讲解了 AUTOSAR OS 的中断管理机制。核心内容包括：Category 1 ISR 不受 OS 管理、响应最快但不能调用 OS API；Category 2 ISR 由 OS 管理，支持调用有限的 OS 服务。中断优先级通过天花板协议确保实时性，开关中断 API（SuspendAllInterrupts、SuspendOSInterrupts、DisableAllInterrupts）各有适用场景和嵌套规则。临界区保护可通过 GetResource 或多核 Spinlock 实现。实验验证了 Cat1/Cat2 ISR 的优先级关系和嵌套行为。与 FreeRTOS 相比，AUTOSAR OS 的中断管理更强调静态配置和确定性。

> ⚠ **注意：**在生产代码中，临界区应尽可能短。长时间关闭中断会影响系统定时器精度和实时响应。

第五章 任务基础

5.1 AUTOSAR OS 任务模型概述

AUTOSAR OS 的任务（Task）是应用程序的基本执行单元。与 FreeRTOS 中动态创建的 Task 不同，AUTOSAR OS 的所有 Task 在编译时静态配置（通过 OIL 文件或 ARXML），运行时不能创建或删除。

- Task 的数量、优先级、栈大小、抢占策略均在编译时确定
- Task 使用 TASK(TaskName) { ... } 宏定义函数体
- 每个 Task 有唯一的 TaskID（Os_TaskType），在 Os_Cfg.h 中定义为宏
- Task 函数体不能使用 return 语句，必须以 TerminateTask() 或 ChainTask() 结束

本项目配置了 3 个 Task：

Task	TaskID	优先级	抢占	激活数	栈大小	自动启动
Task_Init	0x0000	2	NON（非抢占）	1	128 字(512B)	OSDEFAULTAPPMODE
Task_Blink	0x0001	1	FULL（全抢占）	1	128 字(512B)	禁用
OS_TASK_IDLE_CORE0	0x0002	0	FULL（全抢占）	1	64 字(256B)	所有模式

与 FreeRTOS 对比：任务创建

特性	FreeRTOS	AUTOSAR OS
创建方式	xTaskCreate() 动态创建 或 xTaskCreateStatic() 静态	静态配置（OIL/ARXML） 编译时确定
删除任务	vTaskDelete() 可运行时删除	不支持删除 TerminateTask()仅终止当前执行
任务函数	void task(void *pvParam) 可以 return	TASK(name) { ... } 禁止 return
任务标识	TaskHandle_t（指针）	TaskType（uint16 整数 ID）

5.2 基本任务（Basic Task）与扩展任务（Extended Task）

AUTOSAR OS 将 Task 分为两类：Basic Task 和 Extended Task。这一区分决定了 Task 能否使用事件（Event）等待机制。

5.2.1 Basic Task 的特性

Basic Task 是最简单的任务类型，不支持等待事件。

- 状态集合：Suspended → Ready → Running → Suspended（三状态模型）
- 不能调用 WaitEvent()——调用会返回错误
- 一旦开始执行，要么运行完毕（TerminateTask），要么被更高优先级 Task/ISR 抢占
- 内存占用更小（不需要事件控制块）
- 本项目中所有 Task 均为 Basic Task（CFG_EXTENDED_TASK_MAX=0）

Basic Task 适用于「一次性执行」的场景，例如初始化、周期性数据采集、LED 控制等。

```
/* Basic Task 典型实现 */
TASK(Task_Blink) {
    IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_toggled);
```

```
    TerminateTask();  /* 必须以此结束 */
}
```

5.2.2 Extended Task 的特性（等待事件）

Extended Task 在 Basic Task 的基础上增加了 Waiting 状态，可以通过 WaitEvent() 主动让出 CPU 等待事件。

- 状态集合：Suspended → Ready → Running → Waiting → Ready（四状态模型）
- 可调用 WaitEvent()——Task 进入 Waiting 状态，CPU 切换到其他就绪 Task
- 通过 SetEvent() 唤醒——可由其他 Task 或 Cat2 ISR 调用
- 需要更多内存（事件掩码、等待掩码控制块）
- 本项目未使用 Extended Task（CFG_EXTENDED_TASK_MAX=0）

```
/* Extended Task 典型实现（本项目未使用，仅作参考） */
TASK(Task_CommHandler) {
    for (;;) {
        WaitEvent(EVT_DATA_RECEIVED);  /* 进入 Waiting 状态 */
        ClearEvent(EVT_DATA_RECEIVED);
        /* 处理接收到的数据... */
    }
    /* Extended Task 可以使用无限循环 */
}
```

💡 **提示：**Extended Task 的行为类似 FreeRTOS 中使用 xTaskNotifyWait() 或 xQueueReceive() 阻塞的任务。但 AUTOSAR OS 使用事件掩码（Event Mask）而非通知值或队列。

与 FreeRTOS 对比：Basic vs Extended Task

特性	FreeRTOS	AUTOSAR OS
阻塞等待	所有 Task 都可阻塞 (Queue/Semaphore/Notify)	仅 Extended Task 可等待 Basic Task 无 Waiting 状态
事件机制	xTaskNotify (32-bit 值) xEventGroupWaitBits	SetEvent/WaitEvent (事件掩码 EventMaskType)
内存开销	统一 Task 结构体	Extended Task 需额外 事件控制块

5.2.3 激活次数（osTaskActivation）与 BCC1/BCC2 区别

AUTOSAR OS 将 Basic Task 的符合性类（Conformance Class）分为 BCC1 和 BCC2，其核心区别在于是否支持同一任务的多次激活排队：

特性	BCC1	BCC2
osTaskActivation	= 1（固定）	> 1（可配置）
重复激活	第二次 ActivateTask 返回 E_OS_LIMIT	激活请求入队列排队
任务终止后	回到 SUSPENDED	自动取出队列中下一次激活请求
典型应用	单次激活即够的任务	高频触发且不能丢失请求的任务

```
/* BCC1 场景：osTaskActivation = 1 */
ActivateTask(Task_Blink);  // → E_OK, Task_Blink 进入 Ready
ActivateTask(Task_Blink);  // → E_OS_LIMIT! Task_Blink 已激活，第二次被拒绝

/* BCC2 场景：osTaskActivation = 2 */
ActivateTask(Task_Proc);   // → E_OK, 激活计数=1
ActivateTask(Task_Proc);   // → E_OK, 激活计数=2 (排队)
ActivateTask(Task_Proc);   // → E_OS_LIMIT, 超出最大激活数
```

```
// Task_Proc 第一次 TerminateTask 后，自动从队列取出第二次激活请求并重新进入 Ready
```

本项目配置（BCC2，CFG_CC = BCC2）：

- Task_Init: osTaskActivation = 1（初始化任务，无需多次激活）
- Task_Blink: osTaskActivation = 1（虽是 BCC2，但 500ms 周期足够执行完毕，无需排队）
- OS_TASK_IDLE_CORE0: osTaskActivation = 1（系统空闲任务，始终运行）

提示： 如果 Task_Blink 的激活周期远小于执行时间（例如 1ms 激活一次但执行需 2ms），应将 osTaskActivation 设为 2 以避免丢失激活请求。

5.3 任务状态模型

5.3.1 Basic Task 状态：Ready / Running / Suspended

Basic Task 有三种状态，状态转换由 OS 服务调用触发：

状态	枚举值	含义
SUSPENDED	TASK_STATE_SUSPENDED (2)	未激活，不参与调度
READY	TASK_STATE_READY (1)	已激活，等待 CPU
RUNNING	TASK_STATE_RUNNING (3)	正在 CPU 上执行

状态转换：

- SUSPENDED → READY：ActivateTask() 或 ChainTask() 或 AUTOSTART
- READY → RUNNING：调度器选中（最高优先级就绪任务）
- RUNNING → READY：被更高优先级 Task 抢占
- RUNNING → SUSPENDED：TerminateTask() 或 ChainTask()

5.3.2 Extended Task 状态：加上 Waiting

Extended Task 在 Basic Task 的三状态基础上增加了 WAITING 状态：

状态	枚举值	含义
WAITING	TASK_STATE_WAITING (0)	等待事件，不参与调度但保留上下文
READY	TASK_STATE_READY (1)	事件到达或已激活，等待 CPU
RUNNING	TASK_STATE_RUNNING (3)	正在执行
SUSPENDED	TASK_STATE_SUSPENDED (2)	未激活

新增状态转换：

- RUNNING → WAITING：WaitEvent()——主动等待，让出 CPU
- WAITING → READY：SetEvent()——事件被设置，Task 重新就绪

还有一个内部辅助状态 TASK_STATE_START (4)，用于 OS 内部标记任务首次启动。

5.4 任务优先级与抢占

AUTOSAR OS 的任务优先级约定与 FreeRTOS 相同：数值越大，优先级越高。优先级 0 是最低的，通常分配给 Idle Task。

本项目的优先级分配：

Task	优先级	说明
OS_TASK_IDLE_CORE0	0	系统空闲任务，最低优先级
Task_Blink	1	LED 闪烁任务
Task_Init	2	初始化任务，最高优先级

优先级相关配置：

```
#define CFG_PRIORITY_MAX_CORE0    (3U)    // 0, 1, 2 共 3 级
#define CFG_PRIORITY_GROUP_CORE0  (1U)    // 1 个
```

5.4.1 全抢占式调度（Full Preemptive）

设置 osTaskSchedule = OS_PREEMPTIVE_FULL 的 Task 可以在任何时刻被更高优先级 Task 抢占。

- 当更高优先级 Task 变为 Ready（如被 ISR 中的 ActivateTask 激活），立即触发重调度
- 被抢占的 Task 保存上下文，进入 Ready 状态
- 本项目中 Task_Blink 和 OS_TASK_IDLE_CORE0 使用全抢占

全抢占模式提供最佳的实时响应，但需要每个 Task 有独立的栈空间保存上下文。

5.4.2 非抢占式调度（Non Preemptive）

设置 osTaskSchedule = OS_PREEMPTIVE_NON 的 Task 不会被其他 Task 抢占，只有以下情况才会发生 Task 切换：

- Task 主动调用 TerminateTask()、ChainTask()、WaitEvent() 或 Schedule()
- 中断发生（ISR 始终可以抢占 Task，但 ISR 退出后会返回当前 Task）

本项目中 Task_Init 使用非抢占调度（OS_PREEMPTIVE_NON），确保初始化过程不被其他 Task 打断。

⚠ **注意：** 非抢占 Task 运行期间，即使有更高优先级 Task 就绪，也不会发生 Task 级抢占。但硬件中断（Cat1/Cat2 ISR）仍然可以打断非抢占 Task。

5.4.3 混合抢占

当系统中同时存在全抢占和非抢占 Task 时，称为混合抢占调度。本项目正是这种配置：

```
#define CFG_SCHED_POLICY  OS_PREEMPTIVE_MIXED
```

混合抢占规则：

- 全抢占 Task 运行时：可被更高优先级 Task 抢占
- 非抢占 Task 运行时：不被其他 Task 抢占，仅被 ISR 抢占
- 非抢占 Task 调用 Schedule() 时：显式让出调度点，若有更高优先级 Task 就绪则切换

与 FreeRTOS 对比：调度策略

特性	FreeRTOS	AUTOSAR OS
抢占模式	configUSE_PREEMPTION=1 全抢占或协作式二选一	每个 Task 独立配置 FULL/NON 可混合

协作调度点	taskYIELD()	Schedule()
优先级方向	数值大=高优先级	数值大=高优先级（相同）
同优先级	时间片轮转 (configUSE_TIME_SLICING)	FIFO（先激活先执行） 无时间片

5.5 任务实现范式（Task 函数体结构）

AUTOSAR OS 的 Task 函数体有严格的编写规范，与 FreeRTOS 有显著差异：

5.5.1 Basic Task 范式

```
/* 正确的 Basic Task 实现 */
TASK(MyBasicTask) {
    /* 1. 执行业务逻辑 */
    DoSomeWork();

    /* 2. 必须以 TerminateTask() 或 ChainTask() 结束 */
    TerminateTask();
    /* 此行之后的代码永远不会执行 */
}
```

⚠ **注意：** Task 函数体禁止使用 return 语句！TASK() 宏展开后，函数签名不是普通函数。使用 return 会导致未定义行为（通常是系统崩溃）。

5.5.2 Extended Task 范式

```
/* Extended Task 可以使用无限循环 */
TASK(MyExtendedTask) {
    for (;;) {
        WaitEvent(MY_EVENT);       /* 阻塞等待 */
        ClearEvent(MY_EVENT);
        ProcessData();
    }
    /* 不需要 TerminateTask() —— 循环永不结束 */
}
```

5.5.3 错误示例

```
/* ✗ 错误：使用 return */
TASK(BadTask1) {
    DoWork();
    return;  /* 严禁！未定义行为 */
}

/* ✗ 错误：遗漏 TerminateTask */
TASK(BadTask2) {
    DoWork();
    /* 函数末尾没有 TerminateTask → 执行到非法地址 */
}

/* ✗ 错误：Basic Task 中使用无限循环但不等待事件 */
TASK(BadTask3) {
    for (;;) {
```

```
            DoWork();  /* 永远不释放 CPU，低优先级 Task 饿死 */
        }
    }
```

5.5.4 Task 实现范式与 FreeRTOS 根本差异

以下对比展示了同一功能（LED 500ms 闪烁）在 FreeRTOS 和 AUTOSAR OS 中的实现差异：

FreeRTOS Task 实现（无限循环模式）

```
void vTask_Blink(void *pvParameters) {
    for (;;) {
        HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
        vTaskDelay(pdMS_TO_TICKS(500));  // 让出 CPU 500ms
    }
    // 永远不会到达这里
}
```

AUTOSAR OS Basic Task 实现（单次执行模式）

```
TASK(Task_Blink) {
    /* 执行一次任务逻辑 */
    Dio_FlipChannel(DioConf_DioChannel_LED);

    TerminateTask();  // 必须！任务回到 SUSPENDED 状态
    /* 永远不会到达这里 */
}
// 周期性由 Alarm 驱动：AlarmBlink 每 500ms ActivateTask
```

根本差异说明

维度	FreeRTOS	AUTOSAR OS
任务生命周期	永久运行(while(1))	单次执行后终止
周期性实现	vTaskDelay()内部等待	Alarm 外部周期激活
任务结束	vTaskDelete(可选)	TerminateTask(必须)
不调用结束 API	任务继续运行	未定义行为(OS 可能崩溃)
栈复用	不能(每个任务独占)	Basic Task 可复用栈(SC3/4)
状态模型	Ready/Running/ Blocked/Suspended	Ready/Running/Suspended (+Waiting 仅 Extended Task)

⚠ **注意：** AUTOSAR OS Basic Task 中绝对不能使用 while(1) 循环！原因：1) Basic Task 无等待机制，while(1) 会永远占用 CPU；2) 低优先级任务将永远无法执行（除非配置为可抢占）；3) OS 调度器无法正常工作；4) 如需周期任务，使用 Alarm + ActivateTask 模式。

5.6 任务堆栈与上下文保存

在 TriCore 架构上，上下文保存/恢复由硬件自动完成，使用 CSA（Context Save Area）机制。这是 TriCore 与 ARM Cortex-M 的一个重要区别。

5.6.1 CSA 机制

- 每个上下文块（CSA）大小为 64 字节（16 × 32-bit 寄存器）
- CSA 分为 Upper Context 和 Lower Context 两部分
- Upper Context（由硬件自动保存）：A10-A15, D8-D15, PCXI, PSW
- Lower Context（由 CALL 指令自动保存）：A2-A7, D0-D7, A11(RA)

●FCX（Free Context List）：指向空闲 CSA 链表头

●PCX（Previous Context）：指向前一个已保存的上下文

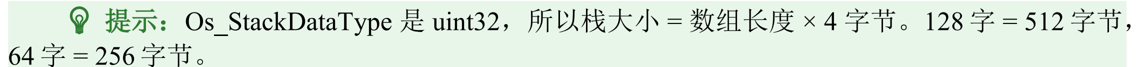
当发生中断或函数调用时，TriCore CPU 自动将当前上下文保存到 FCX 指向的空闲 CSA 块中。

5.6.2 Task 栈配置

虽然 CSA 机制处理了寄存器保存，但每个 Task 仍然需要独立的栈空间用于局部变量和函数调用链。

Task	栈数组	大小（字）	大小（字节）
Task_Init	Os_TaskStack_Task_Init[128]	128	512
Task_Blink	Os_TaskStack_Task_Blink[128]	128	512
OS_TASK_IDLE_CORE0	Os_TaskStack_Idle_Core0[64]	64	256
系统栈	Os_SysStack_Core0[256]	256	1024
ISR2 栈(定时器)	Os_SysTimer_Stack_Core0[256]	256	1024

```
/* Os_Cfg.c 中的栈定义 */
static Os_StackDataType Os_TaskStack_Task_Init[128];  /* 512 bytes */
static Os_StackDataType Os_TaskStack_Task_Blink[128]; /* 512 bytes */
static Os_StackDataType Os_TaskStack_Idle_Core0[64];  /* 256 bytes */

/* 栈描述结构 */
static const Os_StackType Os_TaskStackCfgCore0[] = {
    {OS_STACK_TOP(Os_TaskStack_Task_Init),
     OS_STACK_BOTTOM(Os_TaskStack_Task_Init)},
    {OS_STACK_TOP(Os_TaskStack_Task_Blink),
     OS_STACK_BOTTOM(Os_TaskStack_Task_Blink)},
    {OS_STACK_TOP(Os_TaskStack_Idle_Core0),
     OS_STACK_BOTTOM(Os_TaskStack_Idle_Core0)},
};
```


提示：Os_StackDataType 是 uint32，所以栈大小 = 数组长度 × 4 字节。128 字 = 512 字节，64 字 = 256 字节。

与 FreeRTOS 对比：上下文保存

特性	FreeRTOS	AUTOSAR OS
机制	软件保存寄存器到栈 (PendSV 中保存 R4-R11 等)	CSA 硬件自动保存 64 字节/块，FCX/PCX 管理
栈用途	寄存器保存 + 局部变量	仅局部变量和调用链 寄存器由 CSA 保存
栈溢出检测	configCHECK_FOR_STACK_OVERFLOW	CFG_STACK_CHECK CFG_STACK_MONITOR

5.7 任务自动启动（AUTOSTART）

AUTOSAR OS 支持在 StartOS() 时自动激活指定 Task，无需显式调用 ActivateTask()。自动启动通过 osTaskAutoStartMode 配置。

Task	AutoStartMode	说明
Task_Init	OSDEFAULTAPPMODE	仅在默认模式下自启
Task_Blink	OS_NULL_APPMODE	不自动启动
OS_TASK_IDLE_CORE0	OS_ALL_APPMODE	所有模式下自启

AppMode（应用模式）定义：

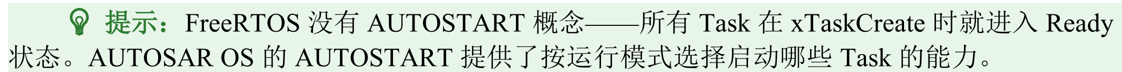
```
#define DONOTCARE          ((Os_AppModeType)0x0U)    // 不
#define OSDEFAULTAPPMODE   ((Os_AppModeType)0x1U)    // 默
```

启动流程：

- ●main() 调用 StartOS(OSDEFAULTAPPMODE)
- ●OS 检查每个 Task 的 osTaskAutoStartMode
- ●若 Task 的 AutoStartMode & 当前 AppMode != 0，则自动 ActivateTask
- ●Task_Init（优先级 2）和 Idle（优先级 0）被激活
- ●调度器启动，Task_Init 优先运行

5.8 本章小结

本章全面介绍了 AUTOSAR OS 的任务模型。核心内容包括：Basic Task 仅有 Ready/Running/Suspended 三种状态，不能阻塞等待；Extended Task 增加 Waiting 状态，可通过 WaitEvent 挂起。任务优先级在配置时静态确定，支持全抢占、非抢占和混合调度策略。Task 函数体必须以 TerminateTask() 或 ChainTask() 结束，禁止使用 while(1) 死循环（这是与 FreeRTOS 最根本的范式差异）。任务堆栈基于 TC334 的 CSA 机制实现上下文保存，栈大小在配置时静态分配。AUTOSTART 机制允许指定任务在 StartOS 后自动进入 Ready 状态。



提示：FreeRTOS 没有 AUTOSTART 概念——所有 Task 在 xTaskCreate 时就进入 Ready 状态。AUTOSAR OS 的 AUTOSTART 提供了按运行模式选择启动哪些 Task 的能力。

第六章 任务 API 详解

6.1 任务管理 API 概览

AUTOSAR OS 提供一组简洁的 Task 管理 API。与 FreeRTOS 的 xTask* 系列函数相比，AUTOSAR OS 的 API 更少但语义更严格。所有 API 返回 StatusType 错误码（E_OK 表示成功）。

API	功能	可调用上下文
ActivateTask()	激活一个处于 SUSPENDED 的 Task	Task, Cat2 ISR
TerminateTask()	终止当前正在运行的 Task	仅 Task
ChainTask()	终止当前 Task 并激活目标 Task	仅 Task
GetTaskID()	获取当前运行 Task 的 ID	Task, Cat2 ISR, Hook
GetTaskState()	获取指定 Task 的当前状态	Task, Cat2 ISR, Hook
Schedule()	显式触发调度点（非抢占 Task 用）	仅 Task

6.1.1 ActivateTask()（激活任务）

属性	说明
函数原型	StatusType ActivateTask(TaskType TaskID)
参数 TaskID	目标任务标识符 （Os_Cfg.h 中定义的宏，如 Task_Blink）
返回值	E_OK：激活成功 E_OS_LIMIT：激活次数超过 osTaskActivation 限制 E_OS_ID：无效的 TaskID E_OS_CALLEVEL：在错误的上下文中调用 E_OS_DISABLEDINT：中断被禁止时调用 E_OS_ACCESS：无权访问该 Task E_OS_CORE：跨核访问错误
适用场景	· 在 Task 中激活另一个 Task · 在 Cat2 ISR 中激活 Task · 在 Alarm 回调中激活 Task
注意事项	· 若 Task 已处于 READY/RUNNING 且 osTaskActivation=1，返回 E_OS_LIMIT · 若 osTaskActivation>1 (BCC2/ECC2)，支持多次激活排队 · 激活后是否立即抢占取决于优先级和当前 Task 的抢占属性 · 本项目为 BCC2 (CFG_CC=OS_BCC2)，支持多次激活

```
/* 在 Alarm 回调中激活 Task_Blink */
static void AlarmCallback_AlarmBlink(void) {
    (void)ActivateTask(Task_Blink);  /* Task_Blink: 0x0001 */
}

/* 在另一个 Task 中激活 */
TASK(Task_Init) {
    StatusType ret = ActivateTask(Task_Blink);
    if (ret != E_OK) {
        /* 处理错误: E_OS_LIMIT 等 */
    }
    TerminateTask();
}
```

与 FreeRTOS 对比：任务激活

特性	FreeRTOS	AUTOSAR OS
激活 API	vTaskResume(handle) xTaskNotifyGive(handle)	ActivateTask(TaskID) 状态: SUSPENDED→READY
多次激活	不支持概念 Task 始终存在	BCC2: osTaskActivation>1 可排队多次激活
ISR 中激活	xTaskNotifyFromISR() (必须用 FromISR 后缀)	ActivateTask() (Cat2 ISR 直接调用)

⚠ **注意：** E_OS_DISABLEDINT 触发场景（SWS_Os_00053）： 在 DisableAllInterrupts() 保护区间内调用 ActivateTask 将返回此错误。

```
/* 错误示例 */
DisableAllInterrupts();
```

```
    ret = ActivateTask(Task_Blink);  // 返回 E_OS_DISABLEDINT，激活失败！
    EnableAllInterrupts();

    /* 正确做法 */
    EnableAllInterrupts();            // 先恢复中断
    ret = ActivateTask(Task_Blink);  // 正常激活
```

同理，SuspendAllInterrupts() 和 SuspendOSInterrupts() 期间也不可调用。此规则适用于所有需要调度的 OS 服务 API。

ActivateTaskAsyn()（异步激活，RPC 场景）

属性	说明
函数原型	StatusType ActivateTaskAsyn(TaskType TaskID)
参数	同 ActivateTask
返回值	同 ActivateTask
适用场景	多核 RPC（远程过程调用）场景 本项目单核，通常不使用
与 ActivateTask 区别	异步执行，不等待目标核确认

6.1.2 TerminateTask()（终止任务）

属性	说明
函数原型	StatusType TerminateTask(void)
参数	无
返回值	正常情况不返回！ 仅在出错时返回： E_OS_RESOURCE：仍持有 Resource E_OS_SPINLOCK：仍持有 Spinlock E_OS_CALLEVEL：在错误上下文调用
适用场景	Basic Task 执行完毕时调用 是 Basic Task 函数体的最后一条语句
注意事项	· 调用后不会返回——OS 切换到下一个就绪 Task · 调用前必须释放所有已获取的 Resource · 仅能在 Task 中调用（不能在 ISR 或 Hook 中调用） · Task 状态变为 SUSPENDED

```
    TASK(Task_Blink) {
        /* 业务逻辑 */
        IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_toggled);

        TerminateTask();  /* 不会返回！ */
        /* 以下代码永远不会执行 */
    }
```

⚠ **注意：**TerminateTask() 不会返回。如果你在 TerminateTask() 之后写了代码，它永远不会执行。如果 TerminateTask() 意外返回（持有 Resource 等），应视为严重错误。

与 FreeRTOS 对比：任务终止

特性	FreeRTOS	AUTOSAR OS
终止 API	vTaskDelete(handle) 可删除任何 Task 释放动态内存	TerminateTask() 仅终止当前 Task Task 变为 SUSPENDED
终止后状态	Task 被彻底删除 内存回收	Task 进入 SUSPENDED 可被再次 ActivateTask
返回行为	无返回值概念	正常不返回 出错时返回 StatusType

⚠ **注意：**Spinlock 约束（SWS_Os_00147，多核场景）： 若 Task 在持有 Spinlock 期间调用 TerminateTask，OS 返回 E_OS_SPINLOCK 而不终止任务。

```
    /* 错误示例 */
    GetSpinlock(Spinlock_SharedData);
      shared_buffer[0] = new_value;
```

```
TerminateTask();  // 返回 E_OS_SPINLOCK! 任务不会终止!
// 代码继续执行...

/* 正确做法 */
GetSpinlock(Spinlock_SharedData);
  shared_buffer[0] = new_value;
ReleaseSpinlock(Spinlock_SharedData);  // 先释放 Spinlock
TerminateTask();  // 正常终止
```

此规则同样适用于 ChainTask()。

6.1.3 ChainTask()（链式激活任务）

属性	说明
函数原型	StatusType ChainTask(TaskType TaskID)
参数 TaskID	目标任务标识符（可以是自己）
返回值	正常情况不返回！ 出错时： E_OS_LIMIT：目标 Task 激活次数已满 E_OS_ID：无效 TaskID E_OS_RESOURCE：仍持有 Resource E_OS_CALLEVEL：错误上下文 E_OS_CORE：跨核错误
适用场景	终止当前 Task 并立即激活另一个 Task 等价于 TerminateTask() + ActivateTask() 的原子操作
注意事项	· 原子操作：不会出现两者之间的调度间隙 · 可以 ChainTask 到自己——等价于重新执行 · 若 ChainTask 到自己且 osTaskActivation=1，不会返回 E_OS_LIMIT · 调用前必须释放所有 Resource

```
/* ChainTask: 终止自己并激活 Task_Blink */
TASK(Task_Init) {
    InitHardware();
    ChainTask(Task_Blink);  /* 不返回 */
}

/* ChainTask 到自己：周期性重执行 */
TASK(Task_Periodic) {
    ProcessData();
    ChainTask(Task_Periodic);  /* 重新激活自己 */
}
```

提示：ChainTask(self) 是让 Basic Task 重复执行的标准模式。与在 Task 中使用无限循环不同，ChainTask 会释放 CPU 让更高优先级 Task 有机会运行。

ChainTask vs ActivateTask+TerminateTask 的差异（osTaskActivation=1 时）

场景：Task_A 想终止自身并激活 Task_B

```
/* 方式 1（错误 - 当 activation=1 时） */
ActivateTask(Task_B);  // 若 Task_B 已处于 READY → E_OS_LIMIT
TerminateTask();       // Task_A 终止

/* 方式 2（正确 - ChainTask 原子操作） */
ChainTask(Task_B);     // 原子操作：先终止 Task_A，再激活 Task_B
                       // 即使 Task_B 的 activation=1 也能成功
                       // 因为 ChainTask 保证终止在先、激活在后
```

ChainTask 的独特优势：

- 原子性：不存在“已终止但还未激活”的中间状态
- 自链接：ChainTask(当前 TaskID) 等效于 TerminateTask + ActivateTask(自身)

●无竞态：避免在 ActivateTask 和 TerminateTask 之间被抢占

6.1.4 GetTaskID() / GetTaskState()（获取任务 ID/状态）

GetTaskID

属性	说明
函数原型	StatusType GetTaskID(TaskRefType TaskID)
参数 TaskID	输出参数，指向 TaskType 变量的指针 调用后写入当前运行 Task 的 ID
返回值	E_OK：成功 若在 ISR/Hook 中调用，TaskID 设为 INVALID_TASK
适用场景	在通用函数中判断当前执行上下文 在 Hook 函数中获取出错的 Task
注意事项	在 ISR 上下文中调用时，TaskID 被设为 INVALID_TASK

```
void MySharedFunction(void) {
    TaskType currentTask;
    GetTaskID(&currentTask);
    if (currentTask == Task_Init) {
        /* 在 Task_Init 上下文中执行 */
    } else if (currentTask == Task_Blink) {
        /* 在 Task_Blink 上下文中执行 */
    }
}
```

GetTaskState

属性	说明
函数原型	StatusType GetTaskState(TaskType TaskID, TaskStateRefType State)
参数 TaskID	查询的目标 Task
参数 State	输出参数，写入 Task 状态 TASK_STATE_RUNNING / READY / SUSPENDED / WAITING
返回值	E_OK：成功 E_OS_ID：无效 TaskID E_OS_ACCESS：无权访问
适用场景	在激活 Task 前检查其当前状态 调试和监控

```
TaskStateType state;
GetTaskState(Task_Blink, &state);
if (state == SUSPENDED) {
    ActivateTask(Task_Blink);  /* 仅在挂起时激活 */
}
```

6.1.5 Schedule()（主动让出处理器）

属性	说明
函数原型	StatusType Schedule(void)
参数	无
返回值	E_OK：成功（可能发生了调度，也可能没有） E_OS_RESOURCE：仍持有 Resource E_OS_SPINLOCK：仍持有 Spinlock E_OS_CALLEVEL：错误上下文
适用场景	非抢占 Task 中的显式调度点 允许更高优先级 Task 有机会运行
注意事项	· 仅在非抢占 Task 中有意义 · 全抢占 Task 调用 Schedule 也合法但效果等同于普通抢占点 · 调用前必须释放所有 Resource

```
/* 非抢占 Task 中使用 Schedule 提供调度点 */
TASK(Task_Init) {  /* OS_PREEMPTIVE_NON */
    Phase1_Init();
    Schedule();       /* 让出调度 — 若有更高优先级 Task 则执行 */
    Phase2_Init();
    Schedule();       /* 再次让出 */
```

```
        Phase3_Init();
        TerminateTask();
    }
```

与 FreeRTOS 对比：显式调度

特性	FreeRTOS	AUTOSAR OS
让出 API	taskYIELD() 立即触发 PendSV	Schedule() 检查就绪队列
延迟让出	vTaskDelay(ticks) vTaskDelayUntil()	无对应 API 需使用 Alarm 周期激活
调用约束	任意 Task 中调用	调用前必须释放 所有 Resource 和 Spinlock

6.2 任务激活与终止实验

6.2.1 实验设计

本实验演示 ActivateTask 和 TerminateTask 的基本使用，在 Task_Init 中激活 Task_Blink 并设置周期性 Alarm。

实验目标

- 掌握 ActivateTask 的调用方式和返回值处理
- 理解 TerminateTask 的「不返回」特性
- 观察 Task 状态转换：SUSPENDED → READY → RUNNING → SUSPENDED

OIL/ARXML 配置片段（等效静态配置）

```
/* OIL 语法表示（等效于 Os_Cfg.h/Os_Cfg.c 的配置） */
TASK Task_Init {
    PRIORITY = 2;
    SCHEDULE = NON;           /* 非抢占 */
    ACTIVATION = 1;
    AUTOSTART = TRUE {
        APPMODE = OSDEFAULTAPPMODE;
    };
    STACKSIZE = 512;          /* 字节 */
};

TASK Task_Blink {
    PRIORITY = 1;
    SCHEDULE = FULL;          /* 全抢占 */
    ACTIVATION = 1;
    AUTOSTART = FALSE;
    STACKSIZE = 512;
};

ALARM AlarmBlink {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK {
        TASK = Task_Blink;
    };
    AUTOSTART = FALSE;
};
```

实验代码

```
#include "Os.h"
#include "IfxPort.h"
```

```
#define LED_PORT  &MODULE_P00
#define LED_PIN   5u

TASK(Task_Init) {
    /* 初始化 LED GPIO */
    IfxPort_setPinMode(LED_PORT, LED_PIN,
                       IfxPort_Mode_outputPushPullGeneral);
    IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_high);

    /* 设置周期 Alarm: 500ms 后首次触发，周期 500ms */
    SetRelAlarm(AlarmBlink, 500, 500);

    /* 验证 ActivateTask 返回值 */
    StatusType ret = ActivateTask(Task_Blink);
    /* 此时 Task_Blink 进入 READY，但因 Task_Init 是 NON-PREEMPTIVE */
    /* 不会立即切换到 Task_Blink */

    TerminateTask();  /* Task_Init 结束，调度器选择 Task_Blink */
}

TASK(Task_Blink) {
    IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_toggled);
    TerminateTask();  /* 返回 SUSPENDED，等待下次 Alarm 激活 */
}
```

6.2.2 运行结果分析

程序运行后的调度序列：

```
1. StartOS(OSDEFAULTAPPMODE)
   → Task_Init (AUTOSTART, 优先级 2) 进入 READY
   → OS_TASK_IDLE_CORE0 (AUTOSTART) 进入 READY

2. 调度器启动，Task_Init (优先级 2) 运行
   → 初始化 GPIO
   → SetRelAlarm(AlarmBlink, 500, 500)
   → ActivateTask(Task_Blink) → 返回 E_OK
     Task_Blink 进入 READY (但 Task_Init 是 NON-PREEMPTIVE，不切换)
   → TerminateTask() → Task_Init 进入 SUSPENDED

3. Task_Blink (优先级 1) 运行
   → LED 翻转
   → TerminateTask() → Task_Blink 进入 SUSPENDED

4. OS_TASK_IDLE_CORE0 (优先级 0) 运行
   → 空闲循环

5. [500ms 后] STM0 ISR 触发 → SystemCounter++
   → AlarmBlink 到期 → AlarmCallback_AlarmBlink()
   → ActivateTask(Task_Blink) → Task_Blink 进入 READY
   → ISR 退出 → 调度 → Task_Blink 运行 → LED 翻转
   → TerminateTask() → 返回 Idle

6. 循环步骤 5，LED 每 500ms 闪烁一次
```

- 观察 LED 闪烁频率应为约 1Hz（500ms 亮 + 500ms 灭）
- 使用调试器断点验证 ActivateTask 返回 E_OK
- 在 Task_Blink 中的 TerminateTask 处设断点，确认每次触发执行一次

6.3 任务链式激活实验

6.3.1 实验设计

本实验演示 ChainTask 的使用，通过任务链实现初始化序列：Task_Init → Task_Blink → 正常运行。

实验目标

- 掌握 ChainTask 的语义：终止当前 + 激活目标（原子操作）
- 理解 ChainTask 与 TerminateTask + ActivateTask 的区别
- 验证 ChainTask 到自身的行为

实验代码

```
#include "Os.h"
#include "IfxPort.h"

#define LED_PORT  &MODULE_P00
#define LED_PIN   5u

static volatile uint32 g_chainCount = 0;

TASK(Task_Init) {
    /* 初始化硬件 */
    IfxPort_setPinMode(LED_PORT, LED_PIN,
                       IfxPort_Mode_outputPushPullGeneral);
    IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_high);

    /* 设置 Alarm */
    SetRelAlarm(AlarmBlink, 500, 500);

    /* ChainTask: 终止 Task_Init 并激活 Task_Blink */
    /* 这是原子操作，不会出现调度间隙 */
    ChainTask(Task_Blink);  /* 不返回 */
}

TASK(Task_Blink) {
    g_chainCount++;
    IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_toggled);

    if (g_chainCount < 3) {
        /* 前 3 次：ChainTask 到自己，立即重新执行 */
        ChainTask(Task_Blink);  /* 不返回 */
    } else {
        /* 之后正常终止，等待 Alarm 周期激活 */
        TerminateTask();
    }
}
```

等效 OIL 配置

```
TASK Task_Init {
    PRIORITY = 2;
    SCHEDULE = NON;
    ACTIVATION = 1;
    AUTOSTART = TRUE { APPMODE = OSDEFAULTAPPMODE; };
    STACKSIZE = 512;
};

TASK Task_Blink {
    PRIORITY = 1;
    SCHEDULE = FULL;
    ACTIVATION = 1;
    AUTOSTART = FALSE;
    STACKSIZE = 512;
};
```

6.3.2 运行结果分析

ChainTask 实验的执行序列：

```
1. Task_Init 启动
   → 初始化 GPIO, 设置 Alarm
   → ChainTask(Task_Blink)
     原子操作: Task_Init→SUSPENDED, Task_Blink→READY

2. Task_Blink 第 1 次运行 (g_chainCount=1)
   → LED 翻转 (亮)
   → g_chainCount < 3 → ChainTask(Task_Blink)
     Task_Blink→SUSPENDED→READY（原子）

3. Task_Blink 第 2 次运行 (g_chainCount=2)
   → LED 翻转 (灭)
   → ChainTask(Task_Blink)

4. Task_Blink 第 3 次运行 (g_chainCount=3)
   → LED 翻转 (亮)
   → g_chainCount >= 3 → TerminateTask()
   → Task_Blink 进入 SUSPENDED

5. Idle Task 运行
   → 等待 Alarm 周期激活 Task_Blink

6. [500ms 后] Alarm → ActivateTask(Task_Blink)
   → 正常闪烁模式开始
```

关键观察点：

- ChainTask 到自身时，Task 会完整重新执行（从函数头开始），局部变量重新初始化
- g_chainCount 作为 static 变量，在 ChainTask 到自身时不会重置
- ChainTask 到自身与 ActivateTask(self) + TerminateTask() 的区别：前者是原子操作，不会产生调度间隙
- 初始化阶段 LED 快速闪烁 3 次（无延迟），之后进入 500ms 周期闪烁

⚠ **注意：** ChainTask 到自身时，若 osTaskActivation=1，不会返回 E_OS_LIMIT。但若先 ActivateTask(self) 再 TerminateTask()，第一步就会因为 Task 已在 RUNNING 而返回 E_OS_LIMIT （除非 osTaskActivation>1）。这是 ChainTask 的独特优势。

6.4 Task API 速查表

API	原型	返回	关键特性
ActivateTask	StatusType ActivateTask (TaskType TaskID)	E_OK E_OS_LIMIT E_OS_ID	激活 SUSPENDED→READY 支持多次激活(BCC2)
TerminateTask	StatusType TerminateTask (void)	不返回 (出错时返回)	终止当前 Task RUNNING→SUSPENDED
ChainTask	StatusType ChainTask (TaskType TaskID)	不返回 (出错时返回)	原子终止+激活 可 Chain 到自身
GetTaskID	StatusType GetTaskID (TaskRefType TaskID)	E_OK	获取当前 TaskID ISR 中返回 INVALID_TASK
GetTaskState	StatusType GetTaskState (TaskType TaskID, TaskStateRefType State)	E_OK E_OS_ID	查询任意 Task 状态
Schedule	StatusType Schedule (void)	E_OK E_OS_RESOURCE	显式调度点 非抢占 Task 专用

6.5 本章小结

本章详细讲解了 AUTOSAR OS 任务管理的核心 API。ActivateTask() 将任务从 Suspended 激活到 Ready 状态；TerminateTask() 终止当前任务并释放资源；ChainTask() 在终止当前任务的同时激活目标任务，实现原子级任务切换。GetTaskID()/GetTaskState() 用于运行时查询，Schedule() 在非抢占任务中手动触发调度点。通过两组实验验证了任务激活/终止的完整生命周期和链式激活的执行时序。与 FreeRTOS 的动态创建/删除不同，AUTOSAR OS 的任务在编译时静态定义，运行时仅做状态切换，确保了确定性和可分析性。

第七章 调度器与调度表

7.1 调度器工作原理

AUTOSAR OS 调度器基于固定优先级抢占式调度（Fixed-Priority Preemptive Scheduling）。调度器在以下时机被触发：任务激活（ActivateTask）、任务终止（TerminateTask/ChainTask）、释放资源（ReleaseResource）、等待/设置事件（WaitEvent/SetEvent）以及中断返回。

7.1.1 优先级队列

本项目配置了 3 个优先级等级（CFG_PRIORITY_MAX_CORE0 = 3），每个优先级对应一个就绪队列：

优先级	队列容量	关联 Task	优先级掩码
0（最低）	1	OS_TASK_IDLE_CORE0	0x0001
1	2	Task_Blink	0x0002
2（最高）	2	Task_Init	0x0004

调度器使用位图（Bitmap）机制快速查找最高优先级就绪任务：

```
/* Os_Cfg.c - 优先级掩码配置 */
static const Os_PriorityType Os_PrioMaskCore0[CFG_PRIORITY_MAX_CORE0] = {
    0x0001U,   /* priority 0 */
    0x0002U,   /* priority 1 */
    0x0004U,   /* priority 2 */
};

/* 调度器通过 ReadyMap 位图快速定位最高优先级：
   Os_ReadyMap_Core0 中对应优先级位置 1 表示该优先级有就绪任务
   使用 CLZ (Count Leading Zeros) 指令快速找到最高位 */
```

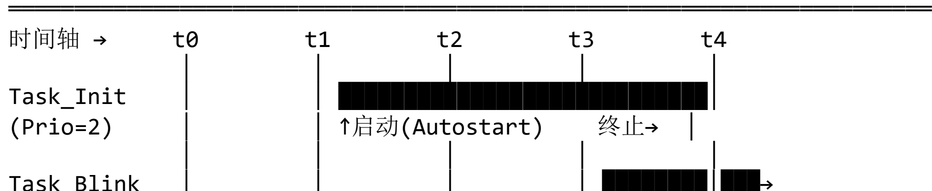
调度策略配置为混合抢占（OS_PREEMPTIVE_MIXED）：

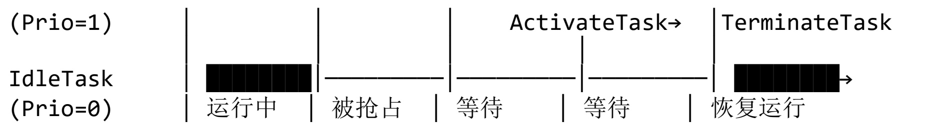
```
/* Os_Cfg.h */
#define CFG_SCHED_POLICY   OS_PREEMPTIVE_MIXED

/* 各任务的抢占属性：
 * Task_Init:  OS_PREEMPTIVE_NON  (不可抢占)
 * Task_Blink: OS_PREEMPTIVE_FULL (完全抢占)
 * IdleTask:   OS_PREEMPTIVE_FULL (完全抢占) */
```

混合抢占模式下，每个任务可以独立配置为可抢占或不可抢占。Task_Init 配置为不可抢占，一旦开始运行就不会被其他任务打断（仅 ISR 可中断）。Task_Blink 配置为完全抢占，如果有更高优先级任务就绪，会被立即抢占。

时序图：

时序图 1：优先级抢占调度（高优先级 Task 抢占低优先级 Task）





说明：
t0: StartOS()后，IdleTask 自动启动（优先级 0，总是就绪）
t1: Task_Init 被 Autostart 激活（优先级 2），抢占 IdleTask
t2: Task_Init 继续执行（注：SetRelAlarm 已在 StartOS()期间由 StartupHook()完成）继续执行（注：SetRelAlarm 已在 StartOS()期间由 StartupHook()完成）周期报警
t3: Task_Init 调用 TerminateTask()终止，调度器选择 Task_Blink
（如果此时 Task_Blink 已被激活）；否则回到 IdleTask
t4: AlarmBlink 触发 ActivateTask(Task_Blink)，Task_Blink 抢占 IdleTask

7.1.2 同优先级 FIFO 策略

当多个任务具有相同优先级时，AUTOSAR OS 采用先进先出（FIFO）策略。先被激活的任务先获得执行权。这与 FreeRTOS 的时间片轮转（Round-Robin）有本质区别：

- AUTOSAR OS：同优先级任务按激活顺序排队，一旦开始运行就执行到终止或被更高优先级抢占
- FreeRTOS：同优先级任务通过时间片轮转共享 CPU（configUSE_TIME_SLICING）

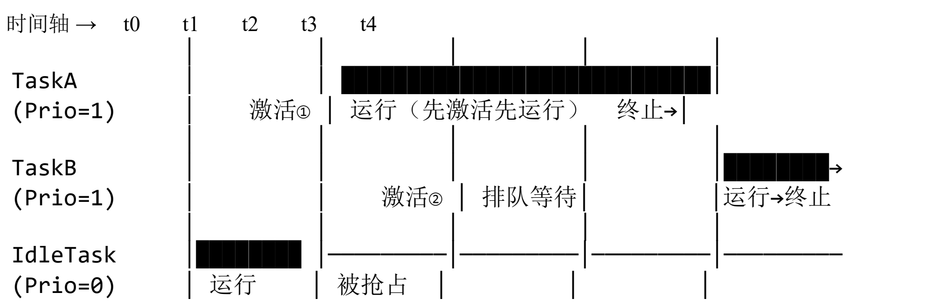
本项目中 Task_Blink 队列容量为 2，支持 BCC2 多次激活队列化：

```
/* Os_Cfg.c */
static Os_TaskType Os_ActivateQueue_Core0_1[2]; /* priority 1: 容量 2 */

/* 场景：Alarm 每 500ms 激活一次 Task_Blink
 * 如果 Task_Blink 执行时间<500ms，队列中最多 1 个实例
 * 如果执行时间≈500ms，可能出现 2 个排队实例 */
```

时序图：

时序图 2：同优先级 FIFO 调度

假设：TaskA 和 TaskB 同为优先级 1，队列容量=2



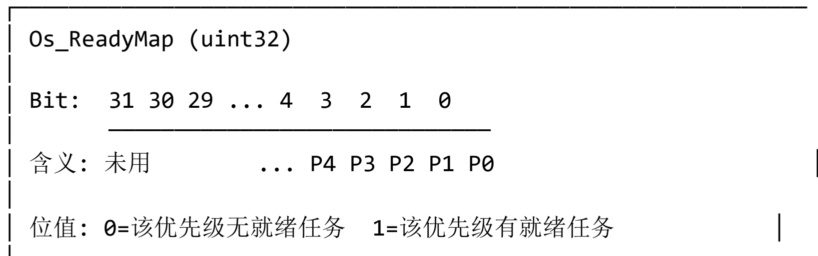
说明：
t0: IdleTask 运行中
t1: TaskA 被激活（先入队），抢占 IdleTask 开始运行
t2: TaskB 被激活（后入队），排在 TaskA 后面等待
t3: TaskA 执行完毕调用 TerminateTask()
t4: 调度器按 FIFO 选择 TaskB 开始运行
注意：不存在时间片切换，TaskA 必须主动终止后 TaskB 才能运行

7.1.3 优先级位图机制（Priority Bitmap）

AUTOSAR OS 使用位图(Bitmap)实现 O(1) 时间复杂度的最高优先级查找。这是调度器高效运行的核心数据结构。

数据结构

Os_ReadyMap 是一个 uint32 类型的位图变量，每一位对应一个优先级等级：



当前项目配置（3 个优先级）

每个优先级对应一个预计算的掩码值：

```
Os_PrioMaskCore0[0] = 0x0001  → 优先级 0 (IdleTask)
Os_PrioMaskCore0[1] = 0x0002  → 优先级 1 (Task_Blink)
Os_PrioMaskCore0[2] = 0x0004  → 优先级 2 (Task_Init)
```

调度器查找最高优先级算法

利用 TriCore CLZ (Count Leading Zeros) 硬件指令，单条指令即可找到最高优先级：

```
/* TriCore CLZ (Count Leading Zeros) 指令 */
highest_prio = 31 - __CLZ(Os_ReadyMap);

示例：
 Os_ReadyMap = 0x00000006 (二进制: ...0110)
 → 优先级 1 和 2 有就绪任务
 __CLZ(0x06) = 29
 highest_prio = 31 - 29 = 2
 → 调度优先级 2 的任务(Task_Init)

性能优势：
 传统方法: 遍历优先级队列 O(N)
 位图+CLZ: 单条指令 O(1)
 TriCore CLZ 硬件指令: 1 个时钟周期
```

任务就绪/移除时的位图操作

任务状态变化时通过简单的位操作更新位图：

```
/* 任务激活：置位对应优先级 */
ActivateTask: Os_ReadyMap |= Os_PrioMaskCore0[task_prio]

/* 任务终止：若该优先级队列为空则清位 */
TerminateTask: 若该优先级队列为空
        Os_ReadyMap &= ~Os_PrioMaskCore0[task_prio]
```

与 FreeRTOS 对比

FreeRTOS 也使用类似的位图机制（uxTopReadyPriority），但在 Cortex-M 上使用 CLZ 指令，TriCore 上对应 CLZ 指令功能相同。

核心差异：AUTOSAR OS 的位图是编译时确定大小（静态），FreeRTOS 的 configMAX_PRIORITIES 可动态配置。

特性	AUTOSAR OS	FreeRTOS
位图变量	Os_ReadyMap (uint32，静态)	uxTopReadyPriority (可配置大小)
查找指令	TriCore CLZ (1 cycle)	Cortex-M CLZ / 软件模拟
优先级数	编译时固定（≤32）	configMAX_PRIORITIES（可动态配置）
置位/清位	Os_PrioMask[] 预计算掩码	portRECORD_READY_PRIORITY() 宏

7.2 调度表概念

Schedule Table 是 AUTOSAR OS 独有的时间驱动调度机制，用于在固定时间点触发一系列动作（激活任务、设置事件）。它弥补了 Alarm 只能触发单一动作的局限性。

7.2.1 调度表（ScheduleTable）简介

Schedule Table 定义了一组按时间顺序排列的过期点（Expiry Point），每个过期点可以包含多个动作。调度表绑定到一个 Counter，随 Counter 递增自动推进。

核心特性：

- 静态配置：所有调度表在 OIL/ARXML 中配置，编译时确定
- 周期性或单次运行：可配置为到达终点后自动重复或停止
- 可切换：运行中的调度表可以切换到另一个调度表（NextScheduleTable）
- 支持同步：与外部时间源（如全局时间、FlexRay）保持同步

注意：本项目当前未配置 Schedule Table（CFG_SCHEDTBL_MAX = 0），但内核代码完整支持。以下内容基于 AUTOSAR OS 标准和内核源码分析。

7.2.2 过期点（Expiry Point）

Expiry Point 是调度表中的时间锚点，定义了相对于调度表起始的偏移（Offset）以及到达该偏移时需执行的动作列表。

动作类型：

- 激活任务（ActivateTask）：将指定任务加入就绪队列
- 设置事件（SetEvent）：为扩展任务设置指定事件掩码

配置示例（OIL 格式）：

```
SCHEDULETABLE MotorControlTable {
  COUNTER = SystemCounter;
  AUTOSTART = FALSE;
  PERIODIC = TRUE;
  LENGTH = 100;  /* 调度表总长度：100 ticks */

  EXPIRY_POINT EP_0 {
    OFFSET = 0;
    ACTION = ACTIVATETASK { TASK = Task_ReadSensor; };
  };
  EXPIRY_POINT EP_1 {
    OFFSET = 25;
    ACTION = ACTIVATETASK { TASK = Task_Calculate; };
  };
  EXPIRY_POINT EP_2 {
    OFFSET = 50;
    ACTION = ACTIVATETASK { TASK = Task_Actuate; };
    ACTION = SETEVENT { TASK = Task_Monitor; EVENT = Evt_CycleComplete; };
  };
};
```

7.2.3 同步模式（隐式同步与显式同步）

AUTOSAR OS 定义了两种同步模式：

隐式同步（Implicit Synchronization）：

- 调度表使用 StartScheduleTableAbs() 以绝对计数值启动
- 假设系统使用的 Counter 与驱动源完全一致，无需额外同步操作
- 适用于本地硬件定时器驱动的场景

显式同步（Explicit Synchronization）：

- 调度表使用 StartScheduleTableSynchron() 启动，进入 WAITING 状态
- 需要外部提供同步计数值（SyncScheduleTable API）
- OS 自动调整过期点偏移以追赶或减速对齐外部时间
- 适用于 FlexRay 全局时间同步、多 ECU 协同调度

特性	隐式同步	显式同步
启动 API	StartScheduleTableAbs()	StartScheduleTableSynchron()
同步源	本地 Counter（无需额外输入）	外部时间源（需 SyncScheduleTable）
初始状态	RUNNING	WAITING → RUNNING_AND_SYNCHRONOUS
适用场景	单 ECU 本地调度	多 ECU 时间同步（FlexRay 等）
复杂度	低	高（需处理偏差补偿）

7.3 调度表 API

7.3.1 StartScheduleTableRel / StartScheduleTableAbs（启动调度表）

StartScheduleTableRel

```
StatusType StartScheduleTableRel(
  ScheduleTableType ScheduleTableID,
  TickType Offset
);
```

参数	类型	说明
ScheduleTableID	ScheduleTableType	调度表标识符（配置生成的宏）
Offset	TickType	相对于当前 Counter 值的偏移量

返回值：

- E_OK：成功启动
- E_OS_ID：无效的 ScheduleTableID
- E_OS_VALUE：Offset = 0 或超出范围
- E_OS_STATE：调度表不在 STOPPED 状态

适用场景：需要在当前时刻之后的某个相对偏移启动调度表，不关心绝对起始时刻。

注意事项：Offset 不能为 0（规范要求），第一个过期点将在 Counter 当前值 + Offset + 第一个 EP 的 Offset 时触发。

StartScheduleTableAbs

```
StatusType StartScheduleTableAbs(
  ScheduleTableType ScheduleTableID,
  TickType Start
);
```

参数	类型	说明
ScheduleTableID	ScheduleTableType	调度表标识符
Start	TickType	Counter 的绝对起始值

返回值：同 StartScheduleTableRel

适用场景：需要在精确的 Counter 绝对值处启动，用于隐式同步。

注意事项：Start 值会被 Counter 的 MaxAllowedValue 取模，第一个过期点在 Start + 第一个 EP 的 Offset 时触发。

与 FreeRTOS 对比：

- FreeRTOS 无直接等价概念。最接近的是 xTimerStart()，但 Software Timer 只能触发单一回调
- AUTOSAR Schedule Table 可在一个时间表中编排多个任务的激活时序，无需多个独立定时器

7.3.2 StopScheduleTable（停止调度表）

```
StatusType StopScheduleTable(
    ScheduleTableType ScheduleTableID
);
```

参数	类型	说明
ScheduleTableID	ScheduleTableType	要停止的调度表标识符

返回值：

- E_OK：成功停止
- E_OS_ID：无效 ID
- E_OS_NOFUNC：调度表已经是 STOPPED 状态

适用场景：运行中需要立即终止调度表（如模式切换、故障处理）。

注意事项：停止后状态变为 STOPPED，已排队但未触发的过期点将被丢弃。再次启动需重新调用 Start API。

与 FreeRTOS 对比：

- 等价于 xTimerStop()，但 Schedule Table 停止会同时取消所有编排的未来动作

7.3.3 NextScheduleTable（切换调度表）

```
StatusType NextScheduleTable(
    ScheduleTableType ScheduleTableID_From,
    ScheduleTableType ScheduleTableID_To
);
```

参数	类型	说明
ScheduleTableID_From	ScheduleTableType	当前运行的调度表
ScheduleTableID_To	ScheduleTableType	下一个要切换到的调度表

返回值：

- E_OK：切换请求已接受
- E_OS_ID：无效 ID
- E_OS_NOFUNC：From 表不在 RUNNING/RUNNING_AND_SYNCHRONOUS 状态
- E_OS_STATE：To 表不在 STOPPED 状态

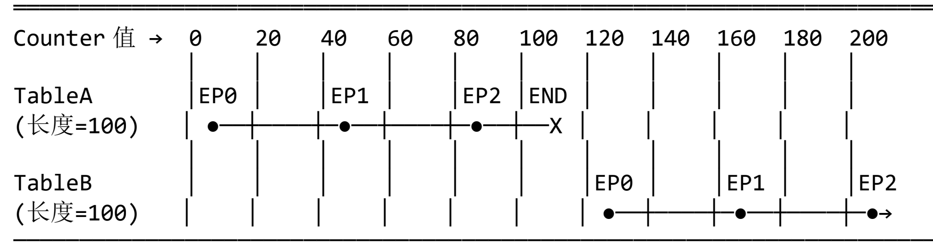
适用场景：在当前调度表运行完一个完整周期后，无缝切换到另一个调度表（如从正常模式切换到降级模式）。

注意事项：

- 切换在 From 表到达终点时自动发生
- From 表状态变为 STOPPED，To 表状态变为 RUNNING
- 多次调用 NextScheduleTable 会覆盖之前的切换请求

时序图：

时序图 3：NextScheduleTable 切换时序



操作序列：
1. StartScheduleTableRel(TableA, 1) @ Counter=0
2. NextScheduleTable(TableA, TableB) @ 任意时刻（Counter=50 处调用）
3. Counter=100 时 TableA 到达 END → 自动停止 → TableB 立即从 EP0 开始

关键点：
- NextScheduleTable 是"预约"操作，不会立即切换
- 切换发生在 From 表的末尾（LENGTH 处）
- To 表从其第一个 Expiry Point 开始执行

7.3.4 GetScheduleTableStatus（获取调度表状态）

```
StatusType GetScheduleTableStatus(
    ScheduleTableType ScheduleTableID,
    ScheduleTableStatusRefType ScheduleStatus
);
```

参数	类型	说明
ScheduleTableID	ScheduleTableType	查询的调度表
ScheduleStatus	ScheduleTableStatusRefType	输出参数，返回当前状态

状态枚举值：

状态	含义
SCHEDULETABLE_STOPPED	调度表已停止，未运行
SCHEDULETABLE_NEXT	调度表已被设置为下一个（等待 From 表结束）
SCHEDULETABLE_WAITING	显式同步模式下等待同步信号
SCHEDULETABLE_RUNNING	正在运行（非同步或隐式同步）
SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS	运行中且已与外部源同步

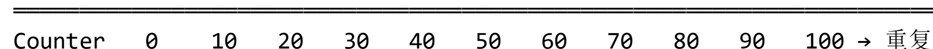
适用场景：诊断当前调度状态，条件性切换逻辑。

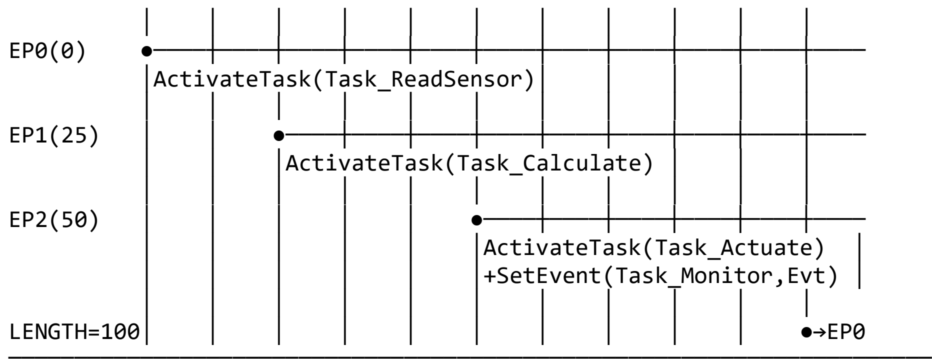
与 FreeRTOS 对比：

- FreeRTOS 使用 xTimerIsTimerActive() 仅能判断定时器是否激活
- AUTOSAR 提供更丰富的状态信息，支持复杂的状态机管理

时序图：

时序图 4：Schedule Table 的 Expiry Point 触发时序





说明：
- 周期性调度表(PERIODIC=TRUE)，到达 LENGTH 后自动从 EP0 重新开始
- 每个 EP 在其 OFFSET 对应的 Counter 值处精确触发
- 一个 EP 可以包含多个动作（如 EP2 同时激活任务和设置事件）
- 动作在 Counter ISR 上下文中执行，然后调度器选择最高优先级任务运行

7.4 调度表实验

7.4.1 实验设计

实验目标：验证 Schedule Table 的基本功能，包括启动、过期点触发、切换和停止。

实验环境：

- 硬件：TC334 Lite Kit
- OS 配置：BCC2/SC1，SystemCounter（1ms/tick）
- 新增配置：2 个 ScheduleTable + 3 个 Task

OIL 配置（新增部分）：

```
/* --- 新增任务配置 --- */
TASK Task_Sensor {
    PRIORITY = 3;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    AUTOSTART = FALSE;
    STACKSIZE = 128;
};
TASK Task_Compute {
    PRIORITY = 3;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    AUTOSTART = FALSE;
    STACKSIZE = 128;
};
TASK Task_Output {
    PRIORITY = 3;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    AUTOSTART = FALSE;
    STACKSIZE = 128;
```

```
};

/* --- Schedule Table 配置 --- */
SCHEDULETABLE SchedTable_Normal {
    COUNTER = SystemCounter;
    PERIODIC = TRUE;
    LENGTH = 100;  /* 100ms 周期 */
    AUTOSTART = NONE;
    LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION = FALSE;

    EXPIRY_POINT EP_Sensor {
        OFFSET = 0;
        ACTION = ACTIVATETASK { TASK = Task_Sensor; };
    };
    EXPIRY_POINT EP_Compute {
        OFFSET = 30;
        ACTION = ACTIVATETASK { TASK = Task_Compute; };
    };
    EXPIRY_POINT EP_Output {
        OFFSET = 60;
        ACTION = ACTIVATETASK { TASK = Task_Output; };
    };
};

SCHEDULETABLE SchedTable_Degraded {
    COUNTER = SystemCounter;
    PERIODIC = TRUE;
    LENGTH = 200;  /* 200ms 周期（降级模式，周期拉长） */
    AUTOSTART = NONE;
    LOCAL_TO_GLOBAL_TIME_SYNCHRONIZATION = FALSE;

    EXPIRY_POINT EP_Sensor_D {
        OFFSET = 0;
        ACTION = ACTIVATETASK { TASK = Task_Sensor; };
    };
    EXPIRY_POINT EP_Output_D {
        OFFSET = 100;
        ACTION = ACTIVATETASK { TASK = Task_Output; };
    };
};
```

应用代码（实验入口）：

```
#include "Os.h"

/* 全局标志用于观测调度行为 */
volatile uint32 g_SensorCount = 0;
volatile uint32 g_ComputeCount = 0;
volatile uint32 g_OutputCount = 0;
volatile uint8  g_DegradedMode = 0;

TASK(Task_Sensor) {
    g_SensorCount++;
    /* 读取传感器数据 */
    TerminateTask();
}

TASK(Task_Compute) {
```

```
    g_ComputeCount++;
    /* 执行控制算法 */
    TerminateTask();
}

TASK(Task_Output) {
    g_OutputCount++;
    /* 输出 PWM 控制信号 */
    TerminateTask();
}

TASK(Task_Init) {
    /* 启动正常模式调度表 */
    StartScheduleTableRel(SchedTable_Normal, 1);

    /* 设置 500ms 后的报警用于演示模式切换 */
    SetRelAlarm(AlarmModeSwitch, 500, 0);
    TerminateTask();
}

/* Alarm 回调：切换到降级模式 */
void AlarmCallback_ModeSwitch(void) {
    if (!g_DegradedMode) {
        g_DegradedMode = 1;
        NextScheduleTable(SchedTable_Normal, SchedTable_Degraded);
    }
}
```

7.4.2 运行结果分析

预期运行行为：

阶段一（0~500ms）- 正常模式：

- 每 100ms 一个周期：t=0 激活 Sensor，t=30 激活 Compute，t=60 激活 Output
- 500ms 内执行 5 个完整周期，g_SensorCount=5, g_ComputeCount=5, g_OutputCount=5

阶段二（500ms+）- 降级模式切换：

- 500ms 时 AlarmModeSwitch 触发，调用 NextScheduleTable
- 等待 SchedTable_Normal 当前周期结束后切换到 SchedTable_Degraded
- 降级模式：每 200ms 仅激活 Sensor 和 Output（跳过 Compute）

调试器观测方法：

```
/* 在调试器的 Watch 窗口中观察以下变量 */
g_SensorCount   // 持续递增（两种模式都有）
g_ComputeCount  // 在切换后停止递增
g_OutputCount   // 持续递增（两种模式都有）
g_DegradedMode  // 0→1 表示模式切换发生
```

```
/* 使用 GetScheduleTableStatus 验证状态 */
ScheduleTableStatusType status;
GetScheduleTableStatus(SchedTable_Normal, &status);
/* 切换前: SCHEDULETABLE_RUNNING */
/* 调用 NextScheduleTable 后: 仍为 RUNNING(等待周期结束) */
/* 切换完成后: SCHEDULETABLE_STOPPED */

GetScheduleTableStatus(SchedTable_Degraded, &status);
/* 切换前: SCHEDULETABLE_NEXT */
/* 切换完成后: SCHEDULETABLE_RUNNING */
```

与 FreeRTOS 实现对比：

如果在 FreeRTOS 中实现类似的多任务时间编排，需要：

```
/* FreeRTOS 等价实现（复杂度显著更高） */
TimerHandle_t timerSensor, timerCompute, timerOutput;

void vTimerSensorCB(TimerHandle_t t) {
  xTaskNotifyGive(hTaskSensor);
}
void vTimerComputeCB(TimerHandle_t t) {
  xTaskNotifyGive(hTaskCompute);
}
void vTimerOutputCB(TimerHandle_t t) {
  xTaskNotifyGive(hTaskOutput);
}

/* 需要 3 个独立的 Software Timer + 精确的启动时序控制 */
/* 模式切换时需要逐一停止/修改/重启所有定时器 */
/* 无法保证多个 Timer 回调之间的精确时序关系 */
```

7.5 本章小结

Schedule Table 将多个任务的时间关系封装在一个配置对象中，切换模式只需一个 API 调用（NextScheduleTable），而 FreeRTOS 需要手动管理多个独立定时器，代码复杂度和出错概率更高。

第八章 计数器与报警

8.1 Counter 概念

Counter（计数器）是 AUTOSAR OS 的时间度量单位。它本质上是一个单调递增的整数值，由硬件或软件事件驱动递增。Alarm 和 Schedule Table 都依赖 Counter 来确定触发时机。

8.1.1 硬件计数器与软件计数器

AUTOSAR OS 支持两种类型的计数器：

硬件计数器（COUNTER_HARDWARE）：

- 由硬件定时器中断驱动递增
- OS 内部通过 ISR 调用 Os_IncrementHardCounter() 推进
- 精度由硬件定时器的时钟源和分频决定
- 本项目使用 STM0（System Timer Module 0）作为硬件计数器源

软件计数器（COUNTER_SOFTWARE）：

- 由应用代码调用 IncrementCounter() 手动递增
- 适用于事件计数（如 CAN 帧到达计数）或由外部触发的不规则时基
- 不占用硬件定时器资源

本项目的 SystemCounter 配置（来自 Os_Cfg.c）：

```
/* Os_Cfg.c - Counter 配置 */
static const Os_CounterCfgType Os_CounterCfgCore0[CFG_COUNTER_MAX_CORE0] = {
  {
    2147483647U,   /* osCounterMaxAllowedValue (0x7FFFFFFF) */
    1U,            /* osCounterMinCycle */
    1U,            /* osCounterTicksPerBase */
    1000U,         /* osSecondsPerTick（单位：微秒，即 1ms） */
    COUNTER_HARDWARE, /* osCounterType */
  },
};

/* SystemCounter 由 STM0 硬件定时器驱动 */
void Os_ArchSystemTimerCore0(void) {
  (void)Os_IncrementHardCounter(SystemCounter);
}
/* STM0 配置: fSTM=100MHz, 比较值=100000 → 中断周期=1ms */
/* Os_Cfg.h: CFG_REG_OSTIMER_VALUE_CORE0 = 100000U */
```

8.1.2 计数器属性（MaxAllowedValue、TicksPerBase 等）

每个 Counter 有以下关键属性：

属性	本项目值	说明
MaxAllowedValue	2147483647 (0x7FFFFFFF)	计数器最大值，到达后回绕到 0

MinCycle	1	Alarm 允许的最小周期值
TicksPerBase	1	每个"基准 tick"对应的硬件 tick 数
SecondsPerTick	1000µs (1ms)	每个 tick 对应的实际时间
Type	COUNTER_HARDWARE	硬件计数器（STM0 驱动）

时间转换宏（Os_Cfg.h）：

```
/* 将 tick 转换为各时间单位 */
#define OS_TICKS2NS_SystemCounter(ticks)  ((ticks)*1000*1000)
#define OS_TICKS2US_SystemCounter(ticks)  ((ticks)*1000)
#define OS_TICKS2MS_SystemCounter(ticks)  ((ticks)*1000/1000)
#define OS_TICKS2SEC_SystemCounter(ticks) ((ticks)*1000/1000000)

/* 将各时间单位转换为 tick */
#define OS_NS2TICKS_SystemCounter(ns)     ((ns)/1000/1000)
#define OS_US2TICKS_SystemCounter(us)     ((us)/1000)
#define OS_MS2TICKS_SystemCounter(ms)     ((ms)*1000/1000)
#define OS_SEC2TICKS_SystemCounter(sec)   ((sec)*1000000/1000)
```

与 FreeRTOS 对比：

特性	FreeRTOS	AUTOSAR OS
系统时基	xTickCount (SysTick 驱动)	Counter (硬件/软件)
时间转换	pdMS_TO_TICKS(ms)	OS_MS2TICKS_SystemCounter(ms)
最大计数	configUSE_16_BIT_TICKS 决定	MaxAllowedValue 配置
多计数器	仅 1 个系统 Tick	支持多个独立 Counter
计数器类型	仅硬件(SysTick)	硬件+软件两种

8.2 Alarm 概念

Alarm（报警器）是 AUTOSAR OS 中基于 Counter 的定时触发机制。当绑定的 Counter 到达指定值时，Alarm 执行预设的动作。

8.2.1 Alarm 与 Counter 的关系

每个 Alarm 必须绑定到一个 Counter。Alarm 的定时精度完全取决于其绑定 Counter 的 tick 周期。

工作流程：

- 1. Counter 由硬件中断或软件调用递增
- 2. OS 在 Counter 递增时检查是否有 Alarm 到期
- 3. 到期的 Alarm 执行其配置的动作
- 4. 如果是循环 Alarm，自动重新装载下次触发值

本项目的 Alarm 配置：

```
/* Os_Cfg.h */
#define CFG_ALARM_MAX       (1U)    /* 系 1 个 Alarm */
#define CFG_AUTO_ALARM_MAX  (0U)    /* 无 Alarm */
```

```
#define AlarmBlink          ((Os_AlarmType)0x0000U)

/* Os_Cfg.c - Alarm 配置结构 */
static const Os_AlarmCfgType Os_AlarmCfgCore0[CFG_ALARM_MAX_CORE0] = {
    {
      NULL_PTR,                /* osAlarmAutostartRef (无自动启动) */
      &AlarmCallback_AlarmBlink,      /* osAlarmCallback */
      Os_GetObjLocalId(SystemCounter), /* osAlarmCounter */
    },
};
```

8.2.2 Alarm 动作类型（激活 Task、设置 Event、回调）

AUTOSAR OS 规范定义了三种 Alarm 到期动作：

动作类型	说明	执行上下文	配置方式
激活 Task	调用 ActivateTask()将目标 Task 加入就绪队列	ISR/内核上下文	OIL: ACTION=ACTIVATETASK
设置 Event	调用 SetEvent()为扩展 Task 设置事件掩码	ISR/内核上下文	OIL: ACTION=SETEVENT
回调函数	调用用户定义的回调函数	ISR 上下文(中断关闭)	OIL: ACTION=ALARMCALLBACK

本项目使用回调函数方式，在回调中激活任务：

```
/* Os_Cfg.c - AlarmBlink 的回调实现 */
static void AlarmCallback_AlarmBlink(void) {
  (void)ActivateTask(Task_Blink);
}

/* 等价的 OIL 配置（如果直接使用 ACTIVATETASK 动作）：
 * ALARM AlarmBlink {
 *    COUNTER = SystemCounter;
 *    ACTION = ACTIVATETASK { TASK = Task_Blink; };
 * };
 */
```

注意事项：

- 回调函数在 Counter ISR 上下文中执行，必须尽可能短
- 回调中不能调用阻塞 API（如 WaitEvent）
- 回调中可以调用 ActivateTask、SetEvent 等非阻塞 OS 服务

8.2.3 Alarm 自动启动配置

Alarm 可以配置为系统启动时自动启动（Autostart），无需应用代码手动调用 SetRelAlarm/SetAbsAlarm。

本项目 AlarmBlink 未配置自动启动（CFG_AUTO_ALARM_MAX = 0），而是在 Task_Init 中手动启动：

```
/* App/App_Tasks.c - 手动启动 Alarm */
TASK(Task_Init) {
    /* 初始化 LED 引脚 */
    IfxPort_setPinModeOutput(LED_PORT, LED_PIN, ...);

    /* 启动周期性 Alarm：首次 500ms 后触发，之后每 500ms 循环 */
    SetRelAlarm(AlarmBlink, 500, 500);

    TerminateTask();
}

/* 如果配置为自动启动(OIL 示例)：
 * ALARM AlarmBlink {
 *     COUNTER = SystemCounter;
 *     AUTOSTART = TRUE {
 *         APPMODE = OSDEFAULTAPPMODE;
 *         ALARMTIME = 500;
 *         CYCLETIME = 500;
 *     };
 * };
 */
```

8.3 Alarm API

8.3.1 SetRelAlarm / SetAbsAlarm

SetRelAlarm

```
StatusType SetRelAlarm(
    AlarmType AlarmID,
    TickType increment,
    TickType cycle
);
```

参数	类型	说明
AlarmID	AlarmType	报警器标识符（如 AlarmBlink）
increment	TickType	相对于当前 Counter 值的首次触发偏移
cycle	TickType	循环周期：0=一次性，>0=循环触发

返回值：

- E_OK：成功设置
- E_OS_ID：无效的 AlarmID
- E_OS_VALUE：increment=0 或 cycle<MinCycle（cycle!=0 时）
- E_OS_STATE：Alarm 已经在运行中

适用场景：设定相对延时的周期性或一次性报警。最典型用法是周期性任务激活。

注意事项：

- increment 不能为 0（规范要求最小为 1）

- cycle=0 表示单次触发，触发后 Alarm 自动回到停止状态
- cycle>0 时必须 cycle >= MinCycle（本项目 MinCycle=1）
- 如果 Alarm 已在运行，必须先 CancelAlarm 再重设

本项目实际使用：

```
/* 设置 AlarmBlink：500ms 后首次触发，之后每 500ms 循环 */
SetRelAlarm(AlarmBlink, 500, 500);
/* 等价于：在 Counter 当前值+500 处首次触发
 * 之后每隔 500 个 tick(=500ms)再次触发 */
```

SetAbsAlarm

```
StatusType SetAbsAlarm(
    AlarmType AlarmID,
    TickType start,
    TickType cycle
);
```

参数	类型	说明
AlarmID	AlarmType	报警器标识符
start	TickType	Counter 的绝对触发值
cycle	TickType	循环周期：0=一次性，>0=循环触发

返回值：同 SetRelAlarm

适用场景：需要在 Counter 的精确绝对值处触发（如同步启动多个 Alarm）。

注意事项：start 值如果小于当前 Counter 值，则在 Counter 回绕后触发。

与 FreeRTOS 对比：

操作	FreeRTOS	AUTOSAR OS
创建周期定时	xTimerCreate() + xTimerStart()	SetRelAlarm(id, delay, cycle)
创建单次定时	xTimerCreate(period,pdFALSE,...)	SetRelAlarm(id, delay, 0)
修改周期	xTimerChangePeriod()	CancelAlarm() + SetRelAlarm()
执行上下文	Timer Task(低优先级)	Counter ISR / 内核上下文
最大定时器数	configTIMER_QUEUE_LENGTH	CFG_ALARM_MAX(编译时固定)

8.3.2 CancelAlarm

```
StatusType CancelAlarm(
    AlarmType AlarmID
);
```

参数	类型	说明
AlarmID	AlarmType	要取消的报警器

返回值：

- E_OK：成功取消
- E_OS_ID：无效的 AlarmID
- E_OS_NOFUNC：Alarm 不在运行状态

适用场景：停止正在运行的周期性 Alarm（如停止 LED 闪烁）或取消未触发的单次 Alarm。
注意事项：取消后 Alarm 回到停止状态，可以重新通过 SetRelAlarm/SetAbsAlarm 设置。

与 FreeRTOS 对比：等价于 xTimerStop()。

8.3.3 GetAlarm / GetAlarmBase

GetAlarm

```
StatusType GetAlarm(
   AlarmType AlarmID,
   TickRefType Tick
);
```

参数	类型	说明
AlarmID	AlarmType	查询的报警器
Tick	TickRefType	输出：距下次触发的剩余 tick 数

返回值：

- E_OK：查询成功
- E_OS_ID：无效 ID
- E_OS_NOFUNC：Alarm 未运行（最常见的错误）

适用场景：诊断 Alarm 剩余时间，判断是否需要提前干预。

GetAlarmBase

```
StatusType GetAlarmBase(
   AlarmType AlarmID,
   AlarmBaseRefType Info
);
```

AlarmBaseType 结构定义：

```
typedef struct {
   Os_TickType maxallowedvalue;  /* Counter 最大允许值 */
   Os_TickType ticksperbase;     /* 每基准 tick 的硬件 tick 数 */
   Os_TickType mincycle;         /* Alarm 最小周期 */
} Os_AlarmBaseType;
```

参数	类型	说明
AlarmID	AlarmType	查询的报警器
Info	AlarmBaseRefType	输出：Alarm 绑定 Counter 的属性信息

返回值：

- E_OK：成功
- E_OS_ID：无效 ID

适用场景：获取 Alarm 绑定 Counter 的属性，用于计算安全的 increment/cycle 值。

注意事项：返回的是 Counter 的属性，不是 Alarm 自身的运行状态。始终返回 E_OK（只要 ID 有效），无论 Alarm 是否运行。

使用示例：

```
AlarmBaseType alarmInfo;
GetAlarmBase(AlarmBlink, &alarmInfo);
/* alarmInfo.maxallowedvalue = 2147483647 */
/* alarmInfo.ticksperbase = 1 */
/* alarmInfo.mincycle = 1 */

/* 安全设置 Alarm 的 cycle 值 */
TickType safeCycle = (desiredMs >= alarmInfo.mincycle) ?
            desiredMs : alarmInfo.mincycle;
```

Counter 相关 API（补充）：

IncrementCounter

```
StatusType IncrementCounter(
  CounterType CounterID
);
```

仅用于软件计数器（COUNTER_SOFTWARE）。每次调用使 Counter 值 +1，并检查是否有 Alarm/ScheduleTable 到期。硬件计数器由 OS 内部 ISR 驱动，不可手动调用。

GetCounterValue

```
StatusType GetCounterValue(
  CounterType CounterID,
  TickRefType Value
);
```

获取指定 Counter 的当前值。可用于时间戳记录或手动计时。

GetElapsedValue

```
StatusType GetElapsedValue(
  CounterType CounterID,
  TickRefType Value,
  TickRefType ElapsedValue
);
```

计算从 Value（输入时为上次记录值）到当前 Counter 值的经过量。自动处理回绕。Value 参数会被更新为当前值。

使用示例：

```
TickType startVal, elapsed;
GetCounterValue(SystemCounter, &startVal);
/* ... 执行一些操作 ... */
GetElapsedValue(SystemCounter, &startVal, &elapsed);
/* elapsed = 经过的 tick 数（处理了回绕） */
```

/* startVal 已更新为当前 Counter 值 */

8.4 Alarm 实验

8.4.1 实验设计

实验目标：验证 Alarm 的设置、触发、取消和查询功能，并观测与 Counter 的交互行为。

实验环境：

- 硬件：TC334 Lite Kit（板载 LED P00.6）
- OS 配置：现有配置（BCC2/SC1，SystemCounter 1ms/tick）
- 使用现有 AlarmBlink + 新增 AlarmDiag 用于对比

OIL 配置（扩展部分）：

```
ALARM AlarmBlink {
    COUNTER = SystemCounter;
    ACTION = ALARMCALLBACK { ALARMCALLBACKNAME = "AlarmCallback_AlarmBlink"; };
    AUTOSTART = FALSE;
};

/* 新增：诊断 Alarm，单次触发 */
ALARM AlarmDiag {
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK { TASK = Task_Diag; };
    AUTOSTART = FALSE;
};

TASK Task_Diag {
    PRIORITY = 2;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    AUTOSTART = FALSE;
    STACKSIZE = 128;
};
```

实验代码：

```
#include "Os.h"
#include "IfxPort.h"

#define LED_PORT  &MODULE_P00
#define LED_PIN   6u

volatile uint32 g_BlinkCount = 0;
volatile uint32 g_DiagCount = 0;
volatile TickType g_AlarmRemaining = 0;
volatile TickType g_CounterValue = 0;

/* AlarmBlink 回调：Toggle LED */
static void AlarmCallback_AlarmBlink(void) {
```

```c
    (void)ActivateTask(Task_Blink);
}

TASK(Task_Blink) {
    g_BlinkCount++;
    IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_toggled);
    TerminateTask();
}

TASK(Task_Diag) {
    g_DiagCount++;
    /* 诊断任务：记录系统状态 */
    GetCounterValue(SystemCounter, (TickRefType)&g_CounterValue);

    /* 查询 AlarmBlink 剩余时间 */
    StatusType ret = GetAlarm(AlarmBlink, (TickRefType)&g_AlarmRemaining);
    if (ret == E_OS_NOFUNC) {
        g_AlarmRemaining = 0xFFFFFFFF; /* 标记未运行 */
    }
    TerminateTask();
}

TASK(Task_Init) {
    /* 初始化 LED */
    IfxPort_setPinModeOutput(LED_PORT, LED_PIN,
        IfxPort_OutputMode_pushPull, IfxPort_OutputIdx_general);
    IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_high);

    /* 启动周期性 LED 闪烁 Alarm: 500ms 周期 */
    SetRelAlarm(AlarmBlink, 500, 500);

    /* 启动单次诊断 Alarm: 2000ms 后触发一次 */
    SetRelAlarm(AlarmDiag, 2000, 0);

    TerminateTask();
}
```

ARXML 等价配置片段（供参考）：

```xml
<ECUC-MODULE-CONFIGURATION-VALUES>
 <SHORT-NAME>Os</SHORT-NAME>
 <CONTAINERS>
  <ECUC-CONTAINER-VALUE>
   <SHORT-NAME>SystemCounter</SHORT-NAME>
   <PARAMETER-VALUES>
    <ECUC-NUMERICAL-PARAM-VALUE>
     <DEFINITION-REF>OsCounterMaxAllowedValue</DEFINITION-REF>
     <VALUE>2147483647</VALUE>
    </ECUC-NUMERICAL-PARAM-VALUE>
    <ECUC-NUMERICAL-PARAM-VALUE>
     <DEFINITION-REF>OsCounterMinCycle</DEFINITION-REF>
     <VALUE>1</VALUE>
    </ECUC-NUMERICAL-PARAM-VALUE>
    <ECUC-TEXTUAL-PARAM-VALUE>
     <DEFINITION-REF>OsCounterType</DEFINITION-REF>
     <VALUE>HARDWARE</VALUE>
    </ECUC-TEXTUAL-PARAM-VALUE>
   </PARAMETER-VALUES>
```

```xml
      </ECUC-CONTAINER-VALUE>
      <ECUC-CONTAINER-VALUE>
        <SHORT-NAME>AlarmBlink</SHORT-NAME>
        <REFERENCE-VALUES>
          <ECUC-REFERENCE-VALUE>
            <DEFINITION-REF>OsAlarmCounterRef</DEFINITION-REF>
            <VALUE-REF>/Os/SystemCounter</VALUE-REF>
          </ECUC-REFERENCE-VALUE>
        </REFERENCE-VALUES>
      </ECUC-CONTAINER-VALUE>
    </CONTAINERS>
  </ECUC-MODULE-CONFIGURATION-VALUES>
```

8.4.2 运行结果分析

预期运行时序：

时间(ms)	事件	说明
0	StartOS() → Task_Init 运行	系统启动
0	SetRelAlarm(AlarmBlink,500,500)	注册周期 Alarm
0	SetRelAlarm(AlarmDiag,2000,0)	注册单次 Alarm
0	Task_Init → TerminateTask()	初始化完成
500	AlarmBlink 触发 → ActivateTask(Task_Blink)	第 1 次 LED 翻转
1000	AlarmBlink 触发 → ActivateTask(Task_Blink)	第 2 次 LED 翻转
1500	AlarmBlink 触发 → ActivateTask(Task_Blink)	第 3 次 LED 翻转
2000	AlarmDiag 触发 → ActivateTask(Task_Diag)	诊断任务执行(仅 1 次)
2000	AlarmBlink 触发 → ActivateTask(Task_Blink)	第 4 次 LED 翻转
2500+	AlarmBlink 继续周期触发	AlarmDiag 已停止

调试器验证要点：

```
/* Watch 窗口观察变量 */
g_BlinkCount      // 每 500ms 递增 1：0→1→2→3→...
g_DiagCount       // 2000ms 时变为 1，之后不再变化
g_AlarmRemaining  // Task_Diag 读取时 AlarmBlink 的剩余 tick
g_CounterValue    // Task_Diag 读取时 SystemCounter 的当前值

/* 断点验证 */
// 在 AlarmCallback_AlarmBlink 设置断点 → 每 500ms 命中一次
// 在 Task_Diag 入口设置断点 → 仅命中一次(t=2000ms)

/* GetAlarm 验证 */
TickType remaining;
StatusType ret = GetAlarm(AlarmBlink, &remaining);
// ret = E_OK, remaining = 距下次触发的 tick 数 (0~499)

ret = GetAlarm(AlarmDiag, &remaining);
// t<2000: ret=E_OK, remaining=距触发的剩余 tick
// t>2000: ret=E_OS_NOFUNC (单次 Alarm 已停止)
```

常见问题排查：

现象	可能原因	解决方法
LED 不闪烁	SetRelAlarm 未调用或参数	检查 Task_Init 执行，确认 increment>0

	错误	
闪烁频率不对	Counter tick 周期理解错误	确认 1tick=1ms，500tick=500ms
GetAlarm 返回 E_OS_NOFUNC	查询时 Alarm 未运行	确保在 SetRelAlarm 之后、CancelAlarm 之前查询
Alarm 触发但 Task 不执行	Task 激活次数超限	检查 ACTIVATION 配置，确保不超过最大激活数

与 FreeRTOS 实现对比：

```
/* FreeRTOS 等价实现 */
TimerHandle_t hTimerBlink;

void vBlinkCallback(TimerHandle_t xTimer) {
   /* 注意：在 Timer Task 上下文中执行，非 ISR */
   xTaskNotifyGive(hTaskBlink);
}

void vAppInit(void) {
   hTimerBlink = xTimerCreate("Blink", pdMS_TO_TICKS(500),
                       pdTRUE, NULL, vBlinkCallback);
   xTimerStart(hTimerBlink, 0);
}

/* 关键差异：
 * 1. FreeRTOS Timer 回调在 Timer Daemon Task 中执行（低优先级）
 *    AUTOSAR Alarm 回调在 Counter ISR 中执行（高优先级、低延迟）
 * 2. FreeRTOS 需要 Timer 命令队列，有排队延迟
 *    AUTOSAR Alarm 直接在 Counter 递增时同步检查
 * 3. FreeRTOS 可动态创建/删除 Timer
 *    AUTOSAR Alarm 数量编译时固定(CFG_ALARM_MAX) */
```

8.5 本章小结

Counter 和 Alarm 构成了 AUTOSAR OS 时间管理的核心框架。Counter 提供硬件级别的精确时基，Alarm 在此基础上实现确定性的定时触发。相比 FreeRTOS 的 Software Timer，AUTOSAR Alarm 具有更低的触发延迟和更强的确定性，适合对时序要求严格的汽车电子应用。

第九章 资源管理

9.1 资源与优先级天花板协议（PCP）

9.1.1 理论讲解

在多任务实时操作系统中，共享资源（如全局变量、硬件外设寄存器等）的互斥访问是核心问题。AUTOSAR OS 采用优先级天花板协议（Priority Ceiling Protocol, PCP）来解决资源竞争中的优先级反转和死锁问题。

PCP 的核心思想：

- 每个资源在配置时被赋予一个「天花板优先级」（Ceiling Priority），等于所有可能访问该资源的任务/ISR 中的最高优先级
- 当任务/ISR 获取资源时，其运行优先级立即被提升至该资源的天花板优先级
- 释放资源后，运行优先级恢复到获取资源前的值
- 由于获取资源后优先级已是最高，不会被同样需要该资源的任务抢占，从而避免了优先级反转

PCP 相比优先级继承（Priority Inheritance）的优势：

- 预防死锁：由于优先级提升是确定性的，不会产生循环等待
- 确定性调度：天花板优先级在配置时确定，运行时行为可预测
- 最多一次阻塞：高优先级任务最多只会被低优先级任务阻塞一次（获取资源的持续时间）

AUTOSAR OS 资源管理的关键约束：

- 资源的获取和释放必须遵循 LIFO（后进先出）顺序，即嵌套获取的资源必须按相反顺序释放
- 不允许在持有资源时调用 TerminateTask()、ChainTask() 或 WaitEvent()
- 资源必须在获取它的同一个函数层级内释放（不能跨任务传递）
- 中断资源（Interrupt Resource）获取后会禁止优先级低于天花板的中断

【时序图 1】无 PCP 时的优先级反转场景

```
时间轴 →

Task_H(高) :        |====|....blocked....|=====>完成
                ↑请求 R(被 L 持有)      ↑L 释放 R 后 H 获得 R

Task_M(中) :            |==============>完成
                   ↑抢占 L(M 不用 R)

Task_L(低) : |==|##R##|...被 M 抢占...|##R##|=>释放 R
          ↑运行 ↑获取 R              ↑M 完成后 L 继续

问题：Task_H 被无关的 Task_M 间接阻塞（优先级反转）
    H 的阻塞时间 = M 的执行时间 + L 持有 R 的剩余时间（不可预测）
```

【时序图 2】PCP 机制解决优先级反转

```
时间轴 →

天花板优先级(R) = Task_H 的优先级

Task_H(高) :        |========================>完成
                ↑请求 R(L 已释放，立即获得)

Task_M(中) :           (无法抢占，因为 L 的运行优先级=H)
                   ...等待...|============>完成
                           ↑L 释放 R 后恢复原优先级

Task_L(低) : |==|##R##(运行优先级=H)|=>释放 R
          ↑运行 ↑获取 R,优先级提升到 Ceiling

结果：Task_H 的阻塞时间 = L 持有 R 的剩余时间（确定性，可分析）
    Task_M 不能抢占持有资源的 Task_L
```

9.1.2 配置演示

当前项目配置（Os_Cfg.h）：

```c
/* -------------------Resource Definition----------------- */
#define     CFG_USERESSCHEDULER                     TRUE
#define     CFG_RESOURCE_MAX                        (0U)
#define     CFG_STD_RESOURCE_MAX                    (0U)
#define     CFG_INTERNAL_RESOURCE_MAX               (0U)
#define     CFG_RESOURCE_MAX_CORE0                  (0U)
#define     CFG_STD_RESOURCE_MAX_CORE0              (0U)
#define     CFG_INTERNAL_RESOURCE_MAX_CORE0         (0U)
#define     CFG_CRITICAL_ZONE_MAX                   3U
```

说明：当前项目为最小配置，未定义用户资源。CFG_USERESSCHEDULER=TRUE 表示系统隐含一个 RES_SCHEDULER 资源，任务获取后可阻止调度（等效于非抢占段）。

若需添加资源，典型的 ARXML/OIL 配置片段如下：

```
/* OIL 配置示例 */
RESOURCE SharedData {
  RESOURCEPROPERTY = STANDARD;
};

TASK Task_Writer {
  PRIORITY = 2;
  RESOURCE = SharedData;
  ...
};

TASK Task_Reader {
  PRIORITY = 4;
  RESOURCE = SharedData;
  ...
};
/* 生成的天花板优先级 = max(2, 4) = 4 */
```

对应生成的 Os_Cfg.h 宏变化：

```
#define     CFG_RESOURCE_MAX            (1U)
#define     CFG_STD_RESOURCE_MAX        (1U)
#define     SharedData                  ((Os_ResourceType)0x0000U)
```

9.2 标准资源与内部资源

9.2.1 标准资源的使用

标准资源（Standard Resource）是应用程序显式获取和释放的资源，通过 GetResource()/ReleaseResource() API 操作。

标准资源的特点：

- 可被任务（Task）和二类中断（ISR2）获取
- 获取后运行优先级提升到资源的天花板优先级
- 支持嵌套获取多个不同资源（必须按 LIFO 顺序释放）
- RES_SCHEDULER 是一种特殊的标准资源，天花板优先级等于系统最高任务优先级

资源配置结构体（Os_ResourceCfgType）：

```
typedef struct {
  Os_PriorityType      ceiling;        /* 天花板优先级 */
  Os_ResourceOccupyType resourceOccupyType; /* 占用类型 */
} Os_ResourceCfgType;
```

资源占用类型枚举：

```
typedef enum {
  OCCUPIED_BY_TASK            = 0, /* 仅被任务占用 */
  OCCUPIED_BY_INTERRUPT        = 1, /* 仅被中断占用 */
  OCCUPIED_BY_TASK_OR_INTERRUPT = 2  /* 被任务或中断占用 */
} Os_ResourceOccupyType;
```

资源控制块（Os_RCBType）管理运行时状态：

```
typedef struct {
  Os_PriorityType    savePrio;   /* 保存获取前的优先级 */
  uint8              saveCount;  /* 嵌套计数 */
  Os_CallLevelType   saveLevel;  /* 保存调用层级 */
} Os_RCBType;
```

9.2.2 内部资源的作用

内部资源（Internal Resource）是 AUTOSAR OS 特有的概念，其获取/释放由操作系统自动完成，不需要应用程序显式调用 API。

内部资源的工作机制：

- 任务从 SUSPENDED/WAITING 进入 RUNNING 状态时，OS 自动获取其绑定的内部资源
- 任务因调度切换离开 RUNNING 状态时，OS 自动释放其内部资源
- 绑定同一内部资源的所有任务形成一个「非抢占组」——组内任务不能互相抢占

内部资源的典型应用场景：

- 实现「组内协作调度」：绑定同一内部资源的任务之间采用协作式调度，只有调用 Schedule() 时才切换
- 创建非抢占域而不影响其他任务的抢占性
- 替代混合调度策略中对单个任务设置 NON_PREEMPTIVE 属性的需求

【时序图 3】内部资源对调度行为的影响

```
配置：Task_A(优先级 2), Task_B(优先级 3) 绑定同一内部资源 IR(天花板=3)
    Task_C(优先级 4) 不绑定 IR

Task_C(4) :              |===========>完成
                      ↑激活，优先级>IR 天花板，可抢占

Task_B(3) : (IR 天花板=3)
        激活但无法抢占 A(A 运行优先级=3)
        ...等待...|=====>完成
              ↑A 调用 Schedule()后 B 获得 CPU

Task_A(2) : |============|→调用 Schedule()|
        ↑运行(自动获取 IR,运行优先级提升到 3)

效果：绑定 IR 的 A 和 B 互不抢占;未绑定 IR 的 C(优先级>天花板)可抢占 A/B
```

FreeRTOS 对比：FreeRTOS 没有内部资源的概念。若需实现类似的非抢占组行为，需要手动调用 vTaskSuspendAll()/xTaskResumeAll() 或使用 taskENTER_CRITICAL()，但这些方法影响范围更大且缺乏配置时的静态分析能力。

9.3 Resource API

9.3.1 GetResource()（获取资源）

函数原型

```
StatusType GetResource(ResourceType ResID);
```

功能描述

获取指定资源，将调用者的运行优先级提升至该资源的天花板优先级。若资源已被占用或参数非法，返回对应错误码。

参数说明

参数	方向	类型	说明
ResID	IN	ResourceType (uint16)	资源标识符，由配置工具生成

返回值

错误码	数值	触发条件
E_OK	0	成功获取资源
E_OS_ID	3	ResID 超出范围（仅 Extended Status）
E_OS_ACCESS	1	调用者无权访问该资源，或资源已被占用（Extended Status）
E_OS_CALLEVEL	2	从非法上下文调用（如 Hook 函数中）
E_OS_CORE	21	跨核访问资源（多核场景）

适用场景

- 保护共享全局变量的读写操作
- 保护硬件外设寄存器的原子操作序列
- 获取 RES_SCHEDULER 以临时创建非抢占段

注意事项

- 嵌套获取多个资源时，后获取的资源天花板优先级必须 >= 先获取的资源
- 获取资源后不能调用 WaitEvent()、TerminateTask()、ChainTask()
- ISR2 获取资源时，资源天花板中断优先级必须 >= 该 ISR 的优先级
- 在 SC3/SC4 下，时间保护会监控资源占用时间

与 FreeRTOS 对比

特性	FreeRTOS	AUTOSAR OS
API	xSemaphoreTake(mutex, timeout)	GetResource(ResID)
协议	优先级继承（Priority Inheritance）	优先级天花板（PCP）
超时	支持（portMAX_DELAY 为永久等待）	不支持超时，立即返回
阻塞	获取失败时任务阻塞等待	不阻塞，返回错误码
嵌套	支持递归互斥量	支持多资源嵌套（LIFO）
死锁预防	不预防（依赖开发者）	PCP 天然预防死锁

9.3.2 ReleaseResource()（释放资源）

函数原型

```
StatusType ReleaseResource(ResourceType ResID);
```

功能描述

释放之前获取的资源，恢复调用者的运行优先级到获取该资源之前的值。可能触发重新调度。

参数说明

参数	方向	类型	说明
ResID	IN	ResourceType (uint16)	要释放的资源标识符

返回值

错误码	数值	触发条件
E_OK	0	成功释放资源
E_OS_ID	3	ResID 超出范围
E_OS_NOFUNC	5	资源未被占用，或 LIFO 释放顺序错误
E_OS_ACCESS	1	调用者不是该资源的当前持有者
E_OS_CALLEVEL	2	从非法上下文调用

适用场景

- 完成共享资源访问后释放
- 释放 RES_SCHEDULER 恢复正常抢占调度

注意事项

- 必须按 LIFO 顺序释放嵌套获取的资源
- 释放后优先级恢复，若有更高优先级任务就绪，将立即发生调度切换
- 中断中释放资源后，若有更高优先级中断挂起，将在返回时触发

与 FreeRTOS 对比

特性	FreeRTOS	AUTOSAR OS
API	xSemaphoreGive(mutex)	ReleaseResource(ResID)
优先级恢复	继承协议恢复（可能不立即）	PCP 立即恢复到保存值
调度触发	taskYIELD_IF_USING_PREEMPTION()	自动触发重新调度
LIFO 要求	无（单独互斥量）	严格 LIFO

9.4 优先级反转问题与解决

优先级反转（Priority Inversion）是实时系统中的经典问题，发生在高优先级任务因低优先级任务持有资源而被阻塞，同时中间优先级任务却能运行的情况。

场景设定：

- Task_H: 高优先级 (Priority=3)
- Task_M: 中优先级 (Priority=2)
- Task_L: 低优先级 (Priority=1)
- Resource_A: 共享资源

9.4.1 场景 1：无保护 — 优先级反转问题

无界优先级反转的必要条件：

- 至少三个不同优先级的任务（L < M < H）
- L 和 H 共享某个资源 R
- M 与资源 R 无关但优先级介于 L 和 H 之间

- 时序：L 持有 R → H 请求 R（阻塞）→ M 抢占 L → H 的阻塞时间取决于 M

【时序图 5】无保护场景 — 优先级反转

时间 → t1 t2 t3 t4 t5 t6 t7 t8

─────────────────────────────

Task_H: .... [等待资源] [等待] [等待] [运行] ....
Task_M: .... [抢占 L] [运行] ....
Task_L: [获取 R] [运行] [运行] [被 H 等待] [被 M 抢占] [挂起] [释放 R] ....

─────────────────────────────

问题分析：
t1: Task_L 获取 Resource_A
t3: Task_H 就绪，抢占 Task_L（但此时 Resource 被 L 持有）
t4: Task_H 尝试获取 Resource_A → 阻塞等待
t5: Task_M 就绪 → 抢占 Task_L（M 优先级 > L 优先级）
★ 问题：Task_M 不需要资源却阻塞了 Task_H！
t6: Task_M 运行完毕
t7: Task_L 恢复，释放 Resource_A
t8: Task_H 终于获得资源

反转时间 = t5~t7（Task_H 被不相关的 Task_M 延迟）

后果：

- H 的最大阻塞时间不可预测（无界）
- 实时系统的可调度性分析失败
- 严重时可导致任务超时、看门狗复位

9.4.2 场景 2：PCP 保护 — 无优先级反转

Resource_A 天花板优先级 = max(Task_H, Task_L) = 3

【时序图 6】PCP 保护场景 — 无优先级反转

时间 → t1 t2 t3 t4 t5

─────────────────────────────

Task_H: .... [运行→获取 R] [运行]
Task_M: .... [就绪但无法抢占] [运行]
Task_L: [获取 R,提升到 P3] [运行 P3] [释放 R,恢复 P1] [被 H 抢占] ....

─────────────────────────────

PCP 解决方案：
t1: Task_L 调用 GetResource(Resource_A)
→ OS 将 Task_L 优先级提升到天花板值 3
t2: Task_M 就绪，但 Task_L 当前优先级=3 > Task_M=2
★ Task_M 无法抢占！优先级反转被消除
t3: Task_L 调用 ReleaseResource(Resource_A)
→ OS 恢复 Task_L 优先级到原始值 1
t4: Task_H 就绪并抢占，直接获取空闲的资源
t5: Task_H 执行完毕后，Task_M 才得到执行

```
总延迟 = 仅 Task_L 持有资源的时间（确定性、有界）
```

PCP 的核心价值：将不可预测的阻塞时间转化为可分析的、有界的延迟，满足实时系统的确定性要求。

9.4.3 资源 LIFO 释放规则

资源的获取和释放必须严格遵循后进先出（LIFO）顺序，即嵌套获取的资源必须按相反顺序释放。

【图示】资源嵌套规则

```
正确用法（LIFO/栈式）：

  GetResource(Res_A);       // 获取 A
    GetResource(Res_B);     // 嵌套获取 B
      /* 临界区代码 */
    ReleaseResource(Res_B); // 先释放 B（后进先出）
  ReleaseResource(Res_A);   // 再释放 A

错误用法（交叉释放）：

  GetResource(Res_A);
    GetResource(Res_B);
  ReleaseResource(Res_A);   // ✗ 错误！应先释放 B
    ReleaseResource(Res_B); // ✗ 错误！

违反 LIFO 规则 → E_OS_NOFUNC 错误
```

违反 LIFO 规则时，ReleaseResource() 返回 E_OS_NOFUNC 错误码，资源不会被释放。这是 AUTOSAR OS 的强制约束，确保资源管理的确定性和死锁预防。

9.4.4 Spinlock 与 Task/ISR 生命周期约束（SWS_Os_00147）

AUTOSAR OS R25-11 明确规定：Task 或 Cat2 ISR 在持有 Spinlock 期间，以下 API 调用将被拒绝并返回 E_OS_SPINLOCK：

【表】持有 Spinlock 期间的禁止操作

禁止 API	行为	原因
TerminateTask()	返回 E_OS_SPINLOCK，任务不终止	持有者终止将导致其他核永久自旋
ChainTask()	返回 E_OS_SPINLOCK，任务不终止	ChainTask 隐含终止当前任务
Schedule()	返回 E_OS_SPINLOCK，不触发调度	让出 CPU 将导致 Spinlock 被长时间持有
WaitEvent()	返回 E_OS_SPINLOCK，不进入等待	进入 WAITING 状态等效于无限期持有锁

设计原理：Spinlock 是忙等机制，若持有者被挂起/终止，其他核将永久自旋，导致系统死锁。因此 OS 在编译时/运行时双重保护。

正确的 Spinlock 使用模式：

```
GetSpinlock(MySpinlock);
  /* 仅执行极短的数据访问操作 */
  shared_data = local_result;
ReleaseSpinlock(MySpinlock);
/* Spinlock 释放后才能调用上述 API */
TerminateTask();
```

Spinlock 嵌套规则（SWS_Os_00148）

多个 Spinlock 必须按严格顺序获取和释放（LIFO）：

```
GetSpinlock(Lock_A);       // 先获取 A
  GetSpinlock(Lock_B);     // 再获取 B
  ReleaseSpinlock(Lock_B); // 先释放 B
ReleaseSpinlock(Lock_A);   // 再释放 A
```

违反嵌套顺序 → E_OS_NESTING_DEADLOCK

尝试获取已被同核持有的锁 → E_OS_INTERFERENCE_DEADLOCK

错误码	触发条件	说明
E_OS_SPINLOCK	持有 Spinlock 期间调用 TerminateTask/ChainTask/Schedule/WaitEvent	SWS_Os_00147
E_OS_NESTING_DEADLOCK	Spinlock 嵌套获取顺序违反	SWS_Os_00148
E_OS_INTERFERENCE_DEADLOCK	同核重复获取同一 Spinlock	自死锁检测

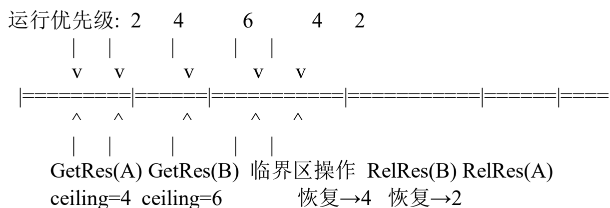
9.4.5 PCP 解决方案在内核中的实现

本 AUTOSAR OS 内核中 GetResource 的核心实现逻辑（简化）：

```
/* Os_Resource.c - GetResource 核心逻辑 */
StatusType GetResource(ResourceType ResID) {
    Os_RCBType* pRcb = &Os_RCB[ResID];
    Os_ResourceCfgType* pCfg = &Os_ResourceCfg[ResID];

    /* 保存当前优先级 */
    pRcb->savePrio = Os_TCB[runningTask].taskRunPrio;
    pRcb->saveLevel = Os_SCB.sysOsLevel;

    /* 提升到天花板优先级 */
    Os_TCB[runningTask].taskRunPrio = pCfg->ceiling;

    /* 压入资源栈（LIFO） */
    taskCriticalZoneStack[zoneCount++] = ResID;
    taskCriticalZoneType[zoneCount] = OBJECT_RESOURCE;

    return E_OK;
}
```

【时序图 4】嵌套资源获取/释放的 LIFO 顺序

Task_A (基础优先级=2)



资源栈状态: [] → [A] → [A,B] → [A,B] → [A] → []

规则：后获取的 B(ceiling=6) >= 先获取的 A(ceiling=4) ✓
释放顺序必须是 B → A（LIFO）

taskCriticalZoneStack 与 taskCriticalZoneType 数组：

```
/* Os_TCBType 中的资源栈字段 */
uint16          taskCriticalZoneStack[CFG_CRITICAL_ZONE_MAX]; /* 最大嵌套深度=3 */
uint16          taskCriticalZoneCount;                  /* 当前栈深度 */
Os_ObjectTypeType taskCriticalZoneType[CFG_CRITICAL_ZONE_MAX]; /* 区分 Resource/Spinlock */
```

9.5 资源管理实验

9.5.1 实验设计

实验目标：验证 PCP 机制下资源获取/释放的行为，观察优先级提升和调度切换。

实验配置（需修改 Os_Cfg.h）：

```
/* 增加资源配置 */
#define     CFG_RESOURCE_MAX        (1U)
#define     CFG_STD_RESOURCE_MAX    (1U)
#define     SharedLED               ((Os_ResourceType)0x0000U)

/* 增加任务配置 */
#define     CFG_TASK_MAX            (5U)
#define     Task_Low                ((Os_TaskType)0x0003U)
#define     Task_High               ((Os_TaskType)0x0004U)
```

实验代码（App_ResourceExp.c）：

```
#include "Os.h"
#include "IfxPort.h"

/* 共享资源：LED 控制序列 */
static volatile uint32 g_LedSequence = 0;

/* Task_Low: 优先级=1, 获取 SharedLED 后执行长操作 */
TASK(Task_Low) {
  StatusType ret;
  ret = GetResource(SharedLED);
  if (ret == E_OK) {
    /* 持有资源期间,运行优先级提升到天花板 */
    g_LedSequence |= 0x01;  /* 标记低优先级段 */
```

```
        /* 模拟耗时操作 */
        for (volatile int i = 0; i < 10000; i++) {}
        g_LedSequence |= 0x04;  /* 标记完成 */
        ReleaseResource(SharedLED);
    }
    TerminateTask();
}

/* Task_High: 优先级=3, 也需要 SharedLED */
TASK(Task_High) {
    StatusType ret;
    g_LedSequence |= 0x10;  /* 标记 H 开始 */
    ret = GetResource(SharedLED);
    if (ret == E_OK) {
        g_LedSequence |= 0x20;  /* 标记 H 持有资源 */
        IfxPort_togglePin(&MODULE_P00, 6);
        ReleaseResource(SharedLED);
    }
    g_LedSequence |= 0x40;  /* 标记 H 完成 */
    TerminateTask();
}
```

9.5.2 运行结果分析

预期执行序列：

- 1. Task_Low 先激活并获取 SharedLED，优先级提升到天花板（=3）
- 2. Task_High 被激活，但由于 Task_Low 运行优先级已=3，Task_High 无法抢占
- 3. Task_Low 完成操作，释放 SharedLED，优先级恢复为 1
- 4. 调度器切换到 Task_High（优先级 3 > 1）
- 5. Task_High 获取 SharedLED（此时已空闲），执行后释放

g_LedSequence 最终值分析：

```
预期值: 0x75
  bit0 (0x01): Task_Low 获取资源后设置     ✓
  bit2 (0x04): Task_Low 完成操作         ✓
  bit4 (0x10): Task_High 开始执行        ✓
  bit5 (0x20): Task_High 获取资源        ✓
  bit6 (0x40): Task_High 完成           ✓
```

验证要点：

- 如果 bit2 在 bit4 之前被设置，说明 PCP 成功阻止了 Task_High 的抢占
- 通过调试器断点或 Trace 功能验证优先级提升/恢复时机
- g_LedSequence 的位设置顺序证明了 LIFO 资源栈的正确性

如果禁用 PCP（假设使用无保护的互斥实现），可能观察到：

```
错误值: 0x35 (bit2 缺失或在 bit4 之后)
  说明 Task_High 在 Task_Low 持有资源期间抢占了它
```

9.6 本章小结

本章深入讲解了 AUTOSAR OS 的资源管理机制。优先级天花板协议（PCP）通过将持有资源的任务提升至天花板优先级，从根本上消除优先级反转和死锁问题。标准资源需显式调用 GetResource/ReleaseResource 并遵循 LIFO 释放规则；内部资源由 OS 在任务调度时自动获取/释放，用于实现任务组内非抢占。资源的天花板优先级在配置时静态计算，运行时开销为 O(1)。实验对比了无保护与 PCP 保护场景下的优先级反转问题，验证了资源管理的正确性。与 FreeRTOS 的互斥量（带优先级继承）相比，PCP 提供了更强的确定性保证。

第十章 事件机制

10.1 Event 概念（仅用于 Extended Task）

10.1.1 理论讲解

事件（Event）是 AUTOSAR OS 中实现任务间同步的轻量级机制。与 FreeRTOS 的 Event Group 不同，AUTOSAR OS 的事件绑定到特定的扩展任务（Extended Task），不是独立对象。

核心概念：

- 事件是一组位掩码（Bitmask），每个位代表一个事件标志
- 事件类型为 uint64，最多支持 64 个独立事件位
- 只有 Extended Task 可以等待事件（调用 WaitEvent）
- 任何任务或 ISR2 都可以设置事件（调用 SetEvent）
- Extended Task 自身清除其事件（调用 ClearEvent）

Extended Task 与 Basic Task 的区别：

特性	Basic Task (BCC1/BCC2)	Extended Task (ECC1/ECC2)
状态	RUNNING, READY, SUSPENDED	RUNNING, READY, WAITING, SUSPENDED
等待事件	不支持	支持（WaitEvent）
多次激活	BCC2 支持	不支持（最多 1 次激活）
释放 CPU 方式	TerminateTask/ChainTask	TerminateTask/ChainTask/WaitEvent
配置	CFG_TASK_MAX	CFG_EXTENDED_TASK_MAX

当前项目配置：

```
#define     CFG_EXTENDED_TASK_MAX       (0U)    /* 当 */
#define     CFG_EXTENDED_TASK_MAX_CORE0 (0U)
```

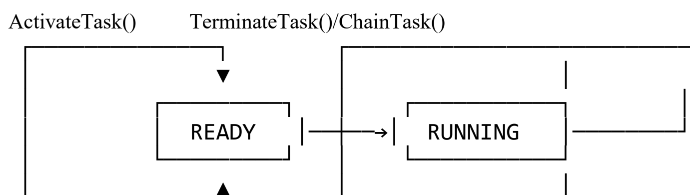
事件控制块（ECB）结构体：

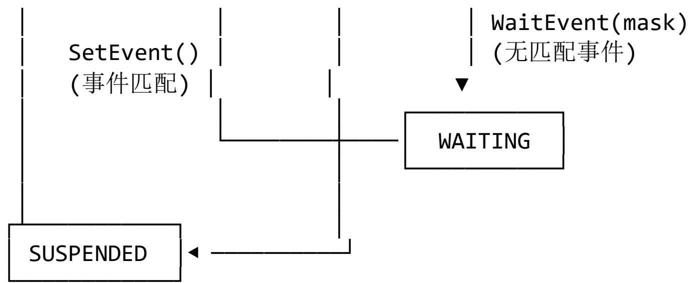
```
typedef struct {
   Os_EventMaskType eventSetEvent;      /* 已设置的事件位集合 */
   Os_EventMaskType eventWaitEvent;     /* 正在等待的事件位集合 */
   boolean        eventIsWaitAllEvents; /* TRUE=AND 等待, FALSE=OR 等待 */
} Os_ECBType;
```

事件类型定义：

```
typedef uint64     Os_EventType;       /* 事件类型 */
typedef uint64     Os_EventMaskType;    /* 事件掩码类型 */
typedef Os_EventMaskType* Os_EventMaskRefType; /* 事件掩码指针 */
```

【状态转换图】Extended Task 的事件驱动状态转换





10.2 Event API

10.2.1 SetEvent()（设置事件）

函数原型

StatusType SetEvent(TaskType TaskID, EventMaskType Mask);

功能描述

设置目标扩展任务的指定事件位。如果目标任务正在等待的事件与新设置的事件匹配（OR 语义），则将目标任务从 WAITING 状态转移到 READY 状态，并可能触发调度。

参数说明

参数	方向	类型	说明
TaskID	IN	TaskType (uint16)	目标扩展任务的 ID
Mask	IN	EventMaskType (uint64)	要设置的事件位掩码

返回值

错误码	数值	触发条件
E_OK	0	成功设置事件
E_OS_ID	3	TaskID 无效或不是扩展任务
E_OS_ACCESS	1	调用者无权设置目标任务的事件
E_OS_STATE	7	目标任务处于 SUSPENDED 状态

适用场景

- ISR2 通知扩展任务有数据到达
- 任务间同步：一个任务完成操作后通知另一个任务继续
- 周期性 Alarm 回调中设置事件唤醒等待任务

注意事项

- 可以同时设置多个事件位（OR 多个掩码）
- 已设置的事件位不会因再次设置而丢失（幂等性）
- 目标任务必须不在 SUSPENDED 状态

- 多核场景下使用 SetEventAsyn() 进行异步 RPC 设置

与 FreeRTOS 对比

特性	FreeRTOS	AUTOSAR OS
API	xEventGroupSetBits(group, bits)	SetEvent(TaskID, Mask)
事件归属	Event Group 是独立对象	事件绑定到特定 Extended Task
谁可设置	任何任务/ISR	任何任务/ISR2
位宽	24 位(32 位系统)	64 位
ISR 安全	xEventGroupSetBitsFromISR()	SetEvent() 直接可用

10.2.2 ClearEvent()（清除事件）

函数原型

StatusType ClearEvent(EventMaskType Mask);

功能描述

清除当前扩展任务的指定事件位。只有任务自己可以清除自己的事件。

参数说明

参数	方向	类型	说明
Mask	IN	EventMaskType (uint64)	要清除的事件位掩码

返回值

错误码	数值	触发条件
E_OK	0	成功清除事件
E_OS_ACCESS	1	调用者不是扩展任务
E_OS_CALLEVEL	2	在 ISR 或 Hook 中调用

注意事项

- 典型用法：WaitEvent() 返回后立即 ClearEvent() 清除已处理的事件
- 不清除事件可能导致下次 WaitEvent() 立即返回（事件仍置位）
- ISR 不能调用 ClearEvent()

10.2.3 WaitEvent()（等待事件）

函数原型

StatusType WaitEvent(EventMaskType Mask);

功能描述

等待指定事件位中的任意一个被设置（OR 语义）。如果指定的事件位都未设置，任务进入 WAITING 状态并释放 CPU。如果有匹配事件已设置，立即返回不会阻塞。

参数说明

参数	方向	类型	说明
Mask	IN	EventMaskType (uint64)	等待的事件位掩码（OR 语义）

返回值

错误码	数值	触发条件
E_OK	0	事件匹配，任务继续执行
E_OS_ACCESS	1	调用者不是扩展任务
E_OS_RESOURCE	6	调用者持有资源（不允许）
E_OS_CALLEVEL	2	在 ISR 或 Hook 中调用

适用场景

- 等待外设数据就绪通知
- 等待定时器超时事件
- 等待多个事件源中的任意一个

注意事项

- 调用 WaitEvent() 时不能持有任何资源（否则返回 E_OS_RESOURCE）
- 只有扩展任务可以调用
- 等待使用 OR 语义：Mask 中任一位匹配即返回
- 对应的 AND 语义使用 WaitAllEvents()

与 FreeRTOS 对比

特性	FreeRTOS	AUTOSAR OS
API	xEventGroupWaitBits(group, bits, clearOnExit, waitAll, timeout)	WaitEvent(Mask)
超时	支持（tick 数或 portMAX_DELAY）	不支持超时（永久等待）
自动清除	clearOnExit 参数控制	不自动清除，需手动 ClearEvent()
AND/OR	waitAll 参数选择	WaitEvent=OR, WaitAllEvents=AND
资源限制	无	不能持有资源

10.2.4 GetEvent()（获取事件状态）

函数原型

```
StatusType GetEvent(TaskType TaskID, EventMaskRefType Event);
```

功能描述

获取指定扩展任务当前的事件掩码值（已设置的事件位集合）。可用于轮询事件状态而不

阻塞。

参数说明

参数	方向	类型	说明
TaskID	IN	TaskType (uint16)	目标扩展任务 ID
Event	OUT	EventMaskRefType (uint64*)	输出：当前事件掩码

返回值

错误码	数值	触发条件
E_OK	0	成功获取事件掩码
E_OS_ID	3	TaskID 无效或非扩展任务
E_OS_ACCESS	1	无访问权限
E_OS_STATE	7	目标任务处于 SUSPENDED

扩展 API - WaitAllEvents()：

StatusType WaitAllEvents(EventMaskType Mask);

功能：等待 Mask 中指定的所有事件位全部被设置后才返回（AND 语义）。在 ECB 中通过 eventIsWaitAllEvents = TRUE 标识。

扩展 API - SetEventAsyn()：

StatusType SetEventAsyn(TaskType TaskID, EventMaskType Mask);

功能：多核场景下的异步事件设置。通过 RPC 机制将 SetEvent 请求发送到目标任务所在的核处理，调用者不等待完成即返回。

10.3 Event 与 Task 状态转换

事件机制对扩展任务状态机的影响：

- RUNNING → WAITING：调用 WaitEvent(Mask)，且 (eventSetEvent & Mask) == 0 时触发
- WAITING → READY：其他任务/ISR 调用 SetEvent()，使得 (eventSetEvent & eventWaitEvent) != 0 时触发
- WAITING 期间：eventSetEvent 可能被多次 SetEvent 累加（位 OR），但只在匹配 waitEvent 时触发状态转换

内核实现关键逻辑（Os_Event.c 简化）：

```
/* SetEvent 核心逻辑 */
void Os_SetEvent_Internal(TaskType TaskID, EventMaskType Mask) {
  Os_ECB[TaskID].eventSetEvent |= Mask;  /* 累加事件位 */

  if (Os_TCB[TaskID].taskState == TASK_STATE_WAITING) {
    if (Os_ECB[TaskID].eventIsWaitAllEvents == FALSE) {
      /* OR 语义：任一位匹配 */
      if ((Os_ECB[TaskID].eventSetEvent & Os_ECB[TaskID].eventWaitEvent) != 0) {
```

```
            Os_EventTaskDispatch(TaskID); /* WAITING→READY */
        }
    } else {
        /* AND 语义：所有位匹配 */
        if ((Os_ECB[TaskID].eventSetEvent & Os_ECB[TaskID].eventWaitEvent)
            == Os_ECB[TaskID].eventWaitEvent) {
            Os_EventTaskDispatch(TaskID);
        }
    }
  }
}

/* WaitEvent 核心逻辑 */
void Os_WaitEvent_Internal(EventMaskType Mask) {
    TaskType taskId = Os_SCB.sysRunningTaskID;
    Os_ECB[taskId].eventWaitEvent = Mask;
    Os_ECB[taskId].eventIsWaitAllEvents = FALSE;

    if ((Os_ECB[taskId].eventSetEvent & Mask) == 0) {
        /* 无匹配事件，进入 WAITING */
        Os_TCB[taskId].taskState = TASK_STATE_WAITING;
        Os_ReadyQueueRemove(taskId);  /* 从就绪队列移除 */
        Os_Dispatch();               /* 触发调度 */
    }
    /* 有匹配事件，直接返回（不阻塞）*/
}
```

10.4 事件机制实验

10.4.1 实验设计

实验目标：验证事件的设置、等待、清除机制，观察 Extended Task 的状态转换。

实验配置（需修改 Os_Cfg.h）：

```
/* 启用扩展任务 */
#define     CFG_CC                      OS_ECC2
#define     CFG_EXTENDED_TASK_MAX       (1U)
#define     CFG_EXTENDED_TASK_MAX_CORE0 (1U)

/* 定义事件 */
#define     Evt_DataReady    ((EventMaskType)0x01U)
#define     Evt_Timeout      ((EventMaskType)0x02U)
#define     Evt_Error        ((EventMaskType)0x04U)

/* 定义扩展任务 */
#define     Task_Handler     ((Os_TaskType)0x0003U)
```

实验代码（App_EventExp.c）：

```
#include "Os.h"
#include "IfxPort.h"
```

```
static volatile uint32 g_EventLog = 0;

/* Task_Handler: Extended Task，等待事件 */
TASK(Task_Handler) {
    StatusType ret;
    EventMaskType events;

    while (1) {
        /* 等待数据就绪或超时事件（OR 语义）*/
        ret = WaitEvent(Evt_DataReady | Evt_Timeout);
        if (ret == E_OK) {
            /* 获取当前事件状态 */
            GetEvent(Task_Handler, &events);

            if (events & Evt_DataReady) {
                g_EventLog |= 0x01; /* 记录数据处理 */
                IfxPort_togglePin(&MODULE_P00, 5);
                ClearEvent(Evt_DataReady);
            }
            if (events & Evt_Timeout) {
                g_EventLog |= 0x02; /* 记录超时处理 */
                ClearEvent(Evt_Timeout);
            }
        }
    }
}

/* Task_Producer: Basic Task，生产数据并通知 */
TASK(Task_Producer) {
    g_EventLog |= 0x10;
    /* 通知 Handler 有数据就绪 */
    SetEvent(Task_Handler, Evt_DataReady);
    g_EventLog |= 0x20;
    TerminateTask();
}

/* ISR2: 定时器中断中设置超时事件 */
ISR(ISR_Timer_Timeout) {
    SetEvent(Task_Handler, Evt_Timeout);
}
```

10.4.2 运行结果分析

预期执行序列：

- 1. Task_Handler 启动后调用 WaitEvent()，进入 WAITING 状态
- 2. Task_Producer 被 Alarm 激活，执行 SetEvent(Evt_DataReady)
- 3. Task_Handler 被唤醒（WAITING→READY→RUNNING），处理数据事件
- 4. Task_Handler 清除事件后再次调用 WaitEvent()，重新进入 WAITING
- 5. ISR_Timer_Timeout 触发，SetEvent(Evt_Timeout) 唤醒 Handler

g_EventLog 分析：

预期最终值: 0x33
bit0 (0x01): 数据就绪事件已处理 ✓
bit1 (0x02): 超时事件已处理 ✓
bit4 (0x10): Producer 开始执行 ✓
bit5 (0x20): Producer SetEvent 后 ✓

关键验证点：

- WaitEvent() 在无匹配事件时确实使任务进入 WAITING（通过调试器查看任务状态）
- SetEvent() 能够唤醒 WAITING 状态的任务
- ClearEvent() 后再次 WaitEvent() 会再次阻塞（事件已清除）
- ISR2 中可以正常调用 SetEvent()

常见错误场景：

- 忘记 ClearEvent() → 下次 WaitEvent() 立即返回，导致忙等待
- 在 Basic Task 中调用 WaitEvent() → 返回 E_OS_ACCESS
- 持有资源时调用 WaitEvent() → 返回 E_OS_RESOURCE

10.5 本章小结

本章讲解了 AUTOSAR OS 的事件机制。Event 仅用于 Extended Task，通过位掩码实现任务间的同步通信。SetEvent() 设置目标任务的事件标志并可唤醒等待中的任务；WaitEvent() 使当前任务进入 Waiting 状态直到指定事件被设置；ClearEvent() 清除事件标志；GetEvent() 查询当前事件状态。事件机制实现了生产者-消费者模式：生产者通过 SetEvent 通知，消费者通过 WaitEvent 等待。实验验证了事件驱动的任务同步流程和状态转换时序。与 FreeRTOS 的任务通知相比，AUTOSAR OS 事件更轻量但仅限于 Extended Task 使用。

第十一章 OS 应用与内存保护

11.1 OS-Application 概念

11.1.1 Trusted 与 Non-Trusted Application

OS-Application 是 AUTOSAR OS 中用于对操作系统对象进行逻辑分组和保护域划分的核心概念。它是 SC3（Scalability Class 3）和 SC4 配置等级下实现功能安全隔离的基础。

OS-Application 的定义：

- 一个 OS-Application 是一组 OS 对象（Tasks、ISRs、Alarms、Counters、ScheduleTables）的逻辑容器
- 每个 OS 对象有且仅属于一个 OS-Application
- OS-Application 确定对象间的访问权限和保护边界

Trusted 与 Non-Trusted Application 的区别：

特性	Trusted Application	Non-Trusted Application
运行模式	Supervisor Mode（完全硬件访问）	User Mode（受限硬件访问）
内存保护	可选（OsTrustedAppWithProtection）	强制启用
时间保护	可延迟处理违规	立即处理违规
系统服务	可调用所有 OS 服务	受限（不能调用 ShutdownOS 等）
外设访问	不受限	仅配置的外设区域
Trusted Function	可提供供 Non-Trusted 调用	不能提供
典型用途	底层驱动、安全监控	应用层功能软件

当前项目配置（SC1，无 OS-Application）：

```
#define     CFG_OSAPPLICATION_MAX     0U
#define     CFG_SC                    OS_SC1
```

注意：SC1 和 SC2 不使用 OS-Application 机制。只有在 SC3/SC4 配置下才需要定义 OS-Application。本章讲解完整机制原理，供系统升级到 SC3/SC4 时参考。

11.1.2 Application 与对象归属

每个 OS 对象归属于某个 OS-Application，由配置工具生成归属映射：

```
/* Os_ObjectAppCfgType：对象到 Application 的映射 */
typedef struct {
  const ApplicationType* accAppRef;       /* 允许访问的 App 列表 */
  ApplicationType      hostApp;       /* 拥有该对象的 App */
  ApplicationType      accAppRefNodeCnt; /* 访问列表长度 */
} Os_ObjectAppCfgType;
```

Application 配置结构体（Os_ApplicationCfgType）核心字段：

```
typedef struct {
  const Os_AppHookCfgType OsApplicationHooks; /* App 专属 Hook */
  const Os_AppObjectRefType** OsAppObjectRef; /* 对象引用列表 */
  CoreIdType OsHostCore;          /* 所属核 */
  uint16     OsAppTaskCnt;        /* App 内任务数量 */
```

```
    uint16    OsAppIsrRefCnt;       /* App 内 ISR 数量 */
    const Os_TaskType OsRestartTask; /* 重启任务 ID */
    boolean   OsTrusted;          /* 是否可信 */
    boolean   OsTrustedAppWithProtection;  /* 可信 App 是否启用保护 */
    boolean   OsTrustedApplicationDelayTimingViolationCall; /* 延迟违规处理 */
} Os_ApplicationCfgType;
```

Application 状态机：

状态	枚举值	含义
APPLICATION_ACCESSIBLE	0	正常运行，对象可被访问
APPLICATION_RESTARTING	1	正在重启（由 TerminateApplication + RESTART 触发）
APPLICATION_TERMINATED	2	已终止，对象不可访问

跨 Application 访问检查（服务保护）：

- 每次 OS 服务调用时，内核检查调用者所在 Application 是否有权访问目标对象
- 无权限时返回 E_OS_ACCESS
- 通过 CheckObjectAccess(AppID, ObjectType, ObjectID) 可主动查询

11.2 内存保护（Memory Protection）

11.2.1 MPU 配置与栈监控

AUTOSAR OS 的内存保护依赖硬件 MPU（Memory Protection Unit）。在 TC334 的 TriCore 架构中，使用数据保护寄存器组（DPR）和代码保护寄存器组（CPR）实现。

TC334 TriCore MPU 特性：

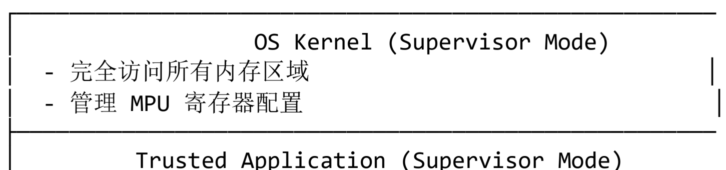
- 数据保护范围寄存器（DPR）：定义可读/可写的数据区域
- 代码保护范围寄存器（CPR）：定义可执行的代码区域
- 保护集合（Protection Set）：User Mode 下生效的保护配置
- 内核有多组保护寄存器可快速切换（任务切换时更换保护集）

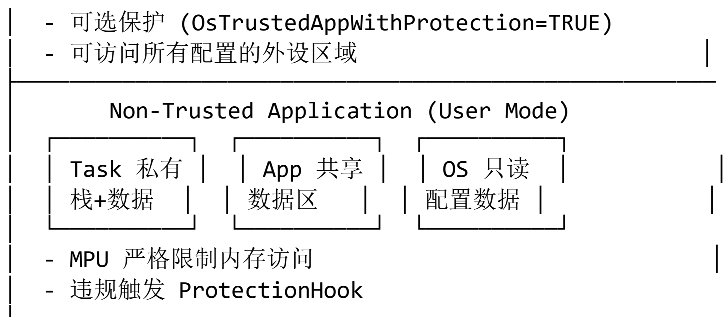
当前项目配置：

```
#define     CFG_MEMORY_PROTECTION_ENABLE    FALSE    /* SC1 不 */
#define     CFG_STACK_CHECK                 FALSE
#define     CFG_STACK_MONITOR               TRUE     /* 软 */
```

SC3/SC4 下的内存保护架构：

【架构图】SC3/SC4 内存保护层次





内存保护相关的地址结构体：

```
/* Task 的内存区域配置 */
typedef struct {
   uint8* Task_ADDR_START;  /* 任务私有数据起始地址 */
   uint8* Task_ADDR_END;    /* 任务私有数据结束地址 */
} OS_TASK_ADDR;

/* ISR 的内存区域配置 */
typedef struct {
   uint8* ISR_ADDR_START;
   uint8* ISR_ADDR_END;
} OS_ISR_ADDR;

/* Application 的外设区域配置 */
typedef struct {
   uint8* APP_ADDR_START;
   uint8* APP_ADDR_END;
} OS_APP_ADDR;
```

栈监控机制（CFG_STACK_MONITOR=TRUE 时生效）：

- 在任务栈底部放置特定模式字（Magic Pattern）
- 任务切换时检查模式字是否被覆盖
- 检测到栈溢出时触发 E_OS_STACKFAULT (14) 错误
- 这是软件检测方法，不如 MPU 硬件保护精确但开销更低

11.2.2 跨 Application 访问权限

SC3/SC4 下的访问控制模型：

- 同一 Application 内的对象可自由互相访问
- 跨 Application 访问需要在配置中显式声明允许列表（accAppRef）
- OS 内核在每次服务调用时执行权限检查

CheckObjectAccess API：

```
ObjectAccessType CheckObjectAccess(
   ApplicationType ApplID,
   ObjectTypeType  ObjectType,
   Os_AppObjectId  ObjectID
);
```

```
// 返回: ACCESS 或 NO_ACCESS
```

对象类型枚举（用于访问检查）：

```
typedef enum {
    OBJECT_TASK          = 0,
    OBJECT_ISR           = 1,
    OBJECT_ALARM         = 2,
    OBJECT_COUNTER       = 3,
    OBJECT_SCHEDULETABLE = 4,
    OBJECT_RESOURCE      = 5,
    OBJECT_SPINLOCK      = 6,
} Os_ObjectTypeType;
```

11.3 服务保护与时间保护

11.3.1 执行时间预算（Execution Budget）

时间保护（Timing Protection）是 SC2/SC4 配置等级下的功能，确保任务和 ISR 的执行时间不超过配置的预算值。

当前项目配置：

```
#define    CFG_TIMING_PROTECTION_ENABLE        FALSE
#define    CFG_TIMING_PROTECTION_ENABLE_CORE0  FALSE
#define    CFG_REG_TP_TIMER_VALUE_CORE0        (100000U)    /* TP 定 */
```

执行时间预算的工作原理：

- 为每个任务/ISR2 配置最大执行时间预算（osTaskExecutionBudget / osIsrExeBudget）
- 任务/ISR 开始执行时启动硬件定时器
- 任务被抢占时暂停计时，恢复执行时继续计时
- 如果累计执行时间超过预算，触发 E_OS_PROTECTION_TIME (16)

任务时间保护配置结构体：

```
typedef struct {
    const Os_TaskResLockType* osTaskResLockRef;   /* 资源锁预算列表 */
    Os_TickType osTaskAllInterruptLockBudget;     /* 禁止全部中断的最大时间 */
    Os_TickType osTaskExecutionBudget;            /* 执行时间预算 */
    Os_TickType osTaskOsInterruptLockBudget;      /* 禁止 OS 中断的最大时间 */
    Os_TickType osTaskTimeFrame;                  /* 到达间隔时间 */
    uint16      osTaskResLockCnt;                 /* 资源锁预算数量 */
} Os_TaskTmProtCfgType;
```

时间保护控制块数据：

```
typedef struct {
    Os_TickType osTpTime;     /* 当前计时值 */
    Os_TickType osTpBudget;   /* 配置的预算值 */
    boolean osIsTpStart;      /* 是否正在计时 */
```

```
  uint8   osWhoHook;      /* 标识哪个保护触发了 Hook */
} Os_TmProtCbDataDef;
```

11.3.2 到达间隔时间（Time Frame）

到达间隔时间保护确保任务/ISR 不会被过于频繁地激活，防止因激活风暴导致系统过载。

工作原理：

- 配置 osTaskTimeFrame / osIsrTimeFrame 为最小到达间隔
- 每次任务激活或 ISR 触发时记录时间戳
- 如果两次激活间隔小于 TimeFrame，触发 E_OS_PROTECTION_ARRIVAL (20)

适用场景：

- 防止外部中断信号毛刺导致 ISR2 频繁触发
- 防止软件错误导致任务被循环激活
- 保障系统的最坏情况可调度性（WCET 分析）

11.3.3 锁占用时间（Locking Time）

锁占用时间保护监控任务/ISR 持有资源或禁止中断的最大时间，防止长时间阻塞系统。

三种锁占用时间监控：

锁类型	配置字段	违规错误码
资源锁（GetResource）	osTaskResLockBudget / osIsrResLockBudget	E_OS_PROTECTION_LOCKED (17)
SuspendAllInterrupts	osTaskAllInterruptLockBudget	E_OS_PROTECTION_LOCKED (17)
SuspendOSInterrupts	osTaskOsInterruptLockBudget	E_OS_PROTECTION_LOCKED (17)

资源锁时间保护配置：

```
typedef struct {
  Os_TickType     OsTaskResLockBudget; /* 该资源的最大占用时间 */
  Os_ResourceType OsTaskResLockResRef; /* 被监控的资源 ID */
} Os_TaskResLockType;
```

与 FreeRTOS 对比：

- FreeRTOS：无时间保护机制，依赖看门狗和开发者自律
- AUTOSAR OS SC2/SC4：硬件定时器驱动的精确时间监控
- FreeRTOS：configCHECK_FOR_STACK_OVERFLOW 仅检测栈溢出
- AUTOSAR OS：全面的执行预算、到达间隔、锁时间三重保护

11.4 Protection Hook

11.4.1 ProtectionHook 回调机制

【API 版本说明】PRO_TERMINATEAPPL_RESTART 在 R22-11 起已移除 OS-Application 重启选项。PRO_PREVENT_ARRIVAL_RATE 为 R24-11 新增。

ProtectionHook 是 OS 在检测到保护违规时调用的用户回调函数，允许应用决定如何处理违规。

函数原型：

```
ProtectionReturnType ProtectionHook(StatusType Fatalerror);
```

当前配置：

```
#define     CFG_PROTECTIONHOOK     FALSE     /* SC1 未 */
```

Fatalerror 参数的可能值：

错误码	数值	含义	触发源
E_OS_STACKFAULT	14	栈溢出/下溢	栈监控检测到异常
E_OS_PROTECTION_MEMORY	15	非法内存访问	MPU 异常
E_OS_PROTECTION_TIME	16	执行时间超预算	TP 定时器超时
E_OS_PROTECTION_LOCKED	17	锁占用超时	资源/中断锁超时
E_OS_PROTECTION_EXCEPTION	18	硬件异常	非法指令/除零等
E_OS_PROTECTION_ARRIVAL	20	到达间隔过短	激活频率超限

ProtectionHook 的返回值（决定 OS 的后续动作）：

返回值	枚举值	OS 动作
PRO_IGNORE	0	忽略错误继续执行（仅内存保护异常可用）
PRO_TERMINATETASKISR	1	终止触发错误的 Task/ISR
PRO_TERMINATEAPPL	2	终止该 Task/ISR 所属的 OS-Application（不重启）
PRO_TERMINATEAPPL_RESTART	3	终止并重启 OS-Application（执行 RestartTask）
PRO_SHUTDOWN	4	关闭整个 OS（调用 ShutdownOS）

```
/* ProtectionReturnType 枚举定义 (Os_Types.h) */
typedef enum {
    PRO_IGNORE              = 0,
    PRO_TERMINATETASKISR      = 1,
    PRO_TERMINATEAPPL         = 2,
    PRO_TERMINATEAPPL_RESTART = 3,  /* ⚠ R22-11 起移除 OS-Application 重启选项 */
    PRO_PREVENT_ARRIVAL_RATE = 5,  /* ✦ R24-11 新增：阻止到达率错误 */
  PRO_SHUTDOWN            = 4
} Os_ProtectionReturnType;
```

ProtectionHook 的调用上下文：

- 在保护违规发生后、OS 执行恢复动作前调用
- 运行在 OS_LEVEL_PROTECTIONHOOK 上下文中
- 不可递归（Hook 内再次违规将直接 ShutdownOS）

- Hook 内可调用的 OS 服务非常有限（GetApplicationID, GetISRID 等信息获取类）

11.4.2 保护实验

实验目标：演示 SC3/SC4 配置下的保护机制行为。

实验配置（升级到 SC3 需修改）：

```
/* 切换到 SC3 */
#define     CFG_SC                          OS_SC3
#define     CFG_MEMORY_PROTECTION_ENABLE    TRUE
#define     CFG_SERVICE_PROTECTION_ENABLE   TRUE
#define     CFG_PROTECTIONHOOK              TRUE
#define     CFG_TIMING_PROTECTION_ENABLE    TRUE
#define     CFG_OSAPPLICATION_MAX           2U

/* 定义 OS-Application */
#define     App_Trusted     ((Os_ApplicationType)0x0000U)
#define     App_User        ((Os_ApplicationType)0x0001U)
```

实验代码（App_ProtectionExp.c）：

```
#include "Os.h"

/* 全局保护违规计数器 */
static volatile uint32 g_ProtViolationCount = 0;
static volatile StatusType g_LastProtError = 0;

/* ProtectionHook 实现 */
ProtectionReturnType ProtectionHook(StatusType Fatalerror) {
  g_ProtViolationCount++;
  g_LastProtError = Fatalerror;

  switch (Fatalerror) {
    case E_OS_PROTECTION_MEMORY:
      /* 内存违规：终止违规任务 */
      return PRO_TERMINATETASKISR;

    case E_OS_PROTECTION_TIME:
      /* 时间超限：终止并重启 Application */
      return PRO_TERMINATEAPPL_RESTART;

    case E_OS_PROTECTION_LOCKED:
      /* 锁超时：终止任务 */
      return PRO_TERMINATETASKISR;

    case E_OS_STACKFAULT:
      /* 栈溢出：关闭系统 */
      return PRO_SHUTDOWN;

    default:
      return PRO_SHUTDOWN;
```

```
    }
}

/* 测试任务 1：触发内存保护违规 */
TASK(Task_MemViolation) {
    /* 尝试写入不属于本 App 的内存区域 */
    volatile uint32* illegal_addr = (uint32*)0xD0000000;
    *illegal_addr = 0x12345678; /* 触发 MPU 异常 */
    /* 执行不到此处 - ProtectionHook 会终止本任务 */
    TerminateTask();
}

/* 测试任务 2：触发时间保护违规 */
TASK(Task_TimeViolation) {
    /* 执行时间超过配置的 Budget */
    volatile uint32 count = 0;
    while (1) { count++; } /* 死循环，超时后被 TP 终止 */
    TerminateTask();
}

/* 测试任务 3：正常任务,验证保护不影响正常执行 */
TASK(Task_Normal) {
    /* 在自己的内存区域内操作 */
    static uint32 myData = 0;
    myData++;
    TerminateTask();
}
```

预期实验结果：

- 1. Task_MemViolation 访问非法地址 → MPU 触发异常 → ProtectionHook(E_OS_PROTECTION_MEMORY) → 返回 PRO_TERMINATETASKISR → 任务被终止
- 2. Task_TimeViolation 死循环 → TP 定时器超时 → ProtectionHook(E_OS_PROTECTION_TIME) → 返回 PRO_TERMINATEAPPL_RESTART → Application 重启
- 3. Task_Normal 正常执行完毕 → 无违规发生
- 4. g_ProtViolationCount 最终 >= 2（至少两次保护违规被捕获）

关键验证点：

- MPU 异常能被正确捕获并路由到 ProtectionHook
- ProtectionHook 的返回值确实控制了 OS 的恢复动作
- PRO_TERMINATEAPPL_RESTART 能正确重启 Application 并执行 RestartTask
- 保护机制不影响正常任务的执行

Scalability Class 对照表（总结本章涉及的保护特性）：

保护特性	**SC1**	**SC2**	**SC3**	**SC4**

OS-Application	×	×	√	√
内存保护（MPU）	×	×	√	√
服务保护	×	×	√	√
时间保护	×	√	×	√
ProtectionHook	×	√	√	√
栈监控（软件）	√	√	√	√

11.5 本章小结

本章介绍了 AUTOSAR OS 的应用隔离与保护机制（SC3/SC4）。OS-Application 将任务、ISR、资源等对象分组管理，分为 Trusted（可访问所有内存）和 Non-Trusted（受 MPU 限制）两类。内存保护通过 MPU 硬件实现栈区、数据区的访问隔离，防止故障扩散。服务保护限制 Non-Trusted Application 可调用的 OS 服务；时间保护通过 Execution Budget、Time Frame、Locking Time 三个维度检测任务超时和资源死锁。ProtectionHook 在违规发生时回调，支持终止任务、重启 Application 或关闭 OS 等恢复策略。当前实验基于 SC1 运行，高级保护特性通过条件编译预留了升级路径。

当前项目运行在 SC1，所有高级保护特性均未启用。但代码中通过条件编译（#if (OS_SC3==CFG_SC) || (OS_SC4==CFG_SC)）已包含完整的 SC3/SC4 保护实现，升级时只需修改 CFG_SC 宏和配置相关参数即可激活。

第十二章 Hook 函数

12.1 Hook 函数概述

Hook 函数是 AUTOSAR OS 提供的一种回调机制，允许用户在 OS 特定事件发生时插入自定义处理逻辑。Hook 函数由 OS 内核在特定时机自动调用，用户只需实现函数体即可。

AUTOSAR OS 定义了以下几类 Hook 函数：

- 系统级 Hook：StartupHook、ShutdownHook、ErrorHook、ProtectionHook
- 任务级 Hook：PreTaskHook、PostTaskHook
- 应用级 Hook（SC3/SC4）：Application-specific StartupHook/ShutdownHook/ErrorHook

Hook 函数的调用级别（OS Level）定义如下：

Hook 函数	OS Level 枚举值	数值
ErrorHook	OS_LEVEL_ERRORHOOK	5
PreTaskHook	OS_LEVEL_PRETASKHOOK	6
PostTaskHook	OS_LEVEL_POSTTASKHOOK	7
StartupHook	OS_LEVEL_STARTUPHOOK	8
ShutdownHook	OS_LEVEL_SHUTDOWNHOOK	9
ProtectionHook	OS_LEVEL_PROTECTIONHOOK	11

Hook 函数运行在特殊的 OS 上下文中，不属于任何 Task 或 ISR。在 Hook 中可调用的 API 受到严格限制，具体允许调用的 API 请参考 AUTOSAR OS 规范。

当前项目 Hook 配置（Os_Cfg.h）：

```
CFG_ERRORHOOK          = TRUE   /* ErrorHook 已启用 */
CFG_PRETASKHOOK        = FALSE  /* PreTaskHook 未启用 */
CFG_POSTTASKHOOK       = FALSE  /* PostTaskHook 未启用 */
CFG_STARTUPHOOK        = TRUE   /* StartupHook 已启用 */
CFG_SHUTDOWNHOOK       = TRUE   /* ShutdownHook 已启用 */
CFG_PROTECTIONHOOK     = FALSE  /* ProtectionHook 未启用(SC1 无需) */
CFG_READYTASKHOOK      = FALSE  /* ReadyTaskHook 未启用 */
CFG_APPL_STARTUPHOOK   = FALSE  /* 应用级 StartupHook 未启用 */
CFG_APPL_ERRORHOOK     = FALSE  /* 应用级 ErrorHook 未启用 */
CFG_APPL_SHUTDOWNHOOK  = FALSE  /* 应用级 ShutdownHook 未启用 */
```

与 FreeRTOS 对比：FreeRTOS 仅提供有限的 Hook 机制（vApplicationIdleHook、vApplicationTickHook、vApplicationStackOverflowHook、vApplicationMallocFailedHook），而 AUTOSAR OS 的 Hook 体系更加系统化，覆盖了 OS 生命周期的各个关键节点。

12.2 系统级 Hook

12.2.1 StartupHook（启动钩子）

StartupHook 在 StartOS() 执行过程中、调度器启动前被调用。此时 OS 已完成内部初始化（Counter、Alarm、Task 等数据结构已就绪），但尚未开始调度。

函数原型

```
void StartupHook(void);
```

项目	说明
功能	系统启动时的用户初始化回调
调用时机	StartOS() 内部，调度器启动前
参数	无
返回值	无
调用上下文	OS_LEVEL_STARTUPHOOK (Level 8)
启用条件	CFG_STARTUPHOOK = TRUE
典型用途	设置 Alarm（SetRelAlarm/SetAbsAlarm）、初始化外设、配置 GPIO
允许调用的 API	SetRelAlarm, SetAbsAlarm, GetActiveApplicationMode 等
禁止调用的 API	ActivateTask, WaitEvent, TerminateTask 等调度类 API

调用时序

```
main()
  └ StartOS(OSDEFAULTAPPMODE)
       ├ Os_InitSystem()          // 内核数据初始化
       ├ Os_InitTask()            // Task TCB 初始化
       ├ Os_InitAlarm()           // Alarm 初始化
       ├ Os_InitCounter()         // Counter 初始化
       ├ StartupHook()            // ◂ — 用户代码执行点
       │    └ SetRelAlarm(AlarmBlink, 500, 500)
       ├ SynPoint(1)              // 多核同步(单核为空操作)
       └ Os_StartScheduler()      // 启动调度器 → 最高优先级 Task
```

当前项目实现

```
/* Os_UserInf.c */
void StartupHook(void)
{
    /* 启动周期 Alarm：延迟 500 ticks 后，每 500 ticks 触发一次 */
    (void)SetRelAlarm(AlarmBlink, 500U, 500U);

    /* 可在此初始化硬件外设 */
    /* IfxPort_setPinModeOutput(&MODULE_P10, 2, ...); */
}
```

注意事项：

- StartupHook 中可调用 ActivateTask()、SetRelAlarm() 等有限 API，激活的任务将在调度器启动后获得执行
- 多核系统中，每个核都会调用各自的 StartupHook，需通过 GetCoreID() 区分
- StartupHook 执行时间应尽量短，避免延迟系统启动

12.2.2 ShutdownHook（关闭钩子）

ShutdownHook 在 ShutdownOS() 执行过程中被调用，是系统关闭前的最后一个用户代码执行点。

函数原型

void ShutdownHook(StatusType Error);

项目	说明
功能	系统关闭时的清理回调
调用时机	ShutdownOS() 内部，中断已禁止后
参数 Error	触发关闭的错误码（由 ShutdownOS(Error) 传入）
返回值	无（函数返回后系统进入死循环）
调用上下文	OS_LEVEL_SHUTDOWNHOOK (Level 9)
启用条件	CFG_SHUTDOWNHOOK = TRUE
典型用途	记录错误日志、保存关键数据到 NVM、关闭外设
允许调用的 API	GetActiveApplicationMode 等少数 API

调用时序

```
ShutdownOS(Error)
  ├─ DisableAllInterrupts()     // 禁止所有中断
  ├─ ShutdownHook(Error)        // ◄── 用户代码执行点
  │     └─ 用户清理操作
  └─ for(;;) {}                 // 系统永久停止
```

实现示例

```
void ShutdownHook(StatusType Error)
{
  /* 记录关闭原因 */
  volatile StatusType shutdownReason = Error;

  /* 关闭所有 LED */
  /* IfxPort_setPinLow(&MODULE_P10, 2); */

  /* 可选：写入 NVM 保存诊断信息 */
  (void)shutdownReason;
}
```

12.2.3 ErrorHook（错误钩子）

ErrorHook 在任何 OS API 返回错误时被自动调用。它是 AUTOSAR OS 错误处理体系的核心组件，提供了集中式的错误捕获机制。

函数原型

void ErrorHook(StatusType Error);

项目	说明
功能	集中式 OS API 错误回调
调用时机	任何 OS API 返回非 E_OK 错误码时
参数 Error	API 返回的错误码
返回值	无
调用上下文	OS_LEVEL_ERRORHOOK (Level 5)
启用条件	CFG_ERRORHOOK = TRUE

典型用途	错误日志记录、DTC 上报、系统降级处理
可获取信息	OSErrorGetServiceId() - 出错的 API ID
注意事项	ErrorHook 自身调用 API 出错时不会再次触发 ErrorHook（防止递归）

ErrorHook 调用流程

```
用户调用 ActivateTask(InvalidTaskID)
  └ 内核检测到 TaskID 无效
       ├ err = E_OS_ID
       ├ if (CFG_ERRORHOOK == TRUE)
       │    └ ErrorHook(E_OS_ID)    // ◂ — 用户代码
       └ return E_OS_ID
```

获取出错 API 信息

在 ErrorHook 中可通过以下宏获取出错 API 的详细信息：

```
void ErrorHook(StatusType Error)
{
  OSServiceIdType serviceId = OSErrorGetServiceId();

  switch (serviceId)
  {
  case OSServiceId_ActivateTask:
    /* 获取出错时的参数 */
    /* TaskType errTask = OSError_ActivateTask_TaskID(); */
    break;
  case OSServiceId_SetRelAlarm:
    break;
  default:
    break;
  }

  /* 记录错误信息 */
  volatile StatusType lastError = Error;
  (void)lastError;
}
```

常见触发 ErrorHook 的错误码：

错误码	数值	含义
E_OS_ACCESS	1	访问权限错误
E_OS_CALLEVEL	2	调用上下文错误
E_OS_ID	3	无效的 OS 对象 ID
E_OS_LIMIT	4	超过激活次数限制
E_OS_NOFUNC	5	操作无效（如释放未获取的资源）
E_OS_RESOURCE	6	Task 未释放资源就终止
E_OS_STATE	7	对象状态错误
E_OS_VALUE	8	参数值超出范围

12.2.4 ProtectionHook（SC3/SC4）

ProtectionHook 是内存保护和时间保护的错误处理接口，仅在 Scalability Class 3/4 中可用。当前项目为 SC1，未启用此 Hook。

函数原型

ProtectionReturnType ProtectionHook(StatusType Fatalerror);

项目	说明
功能	保护违规处理回调
触发条件	内存访问违规(MPU 异常)、时间保护超时、栈溢出
参数 Fatalerror	E_OS_PROTECTION_MEMORY / E_OS_PROTECTION_TIME / E_OS_PROTECTION_ARRIVAL / E_OS_PROTECTION_LOCKED / E_OS_STACKFAULT / E_OS_PROTECTION_EXCEPTION
返回值	PRO_IGNORE / PRO_TERMINATETASKISR / PRO_TERMINATEAPPL / PRO_TERMINATEAPPL_RESTART / PRO_SHUTDOWN
启用条件	CFG_PROTECTIONHOOK = TRUE (SC3/SC4)

返回值含义

返回值	行为
PRO_IGNORE	忽略错误（仅限 E_OS_PROTECTION_ARRIVAL）
PRO_TERMINATETASKISR	终止出错的 Task/ISR
PRO_TERMINATEAPPL	终止出错 Task 所属的 OS-Application
PRO_TERMINATEAPPL_RESTART	终止并重启 OS-Application
PRO_SHUTDOWN	关闭整个 OS（调用 ShutdownOS）

12.3 任务级 Hook

12.3.1 PreTaskHook（任务切入前钩子）

PreTaskHook 在任务状态转为 RUNNING 之后（即调度器选中该任务后）、任务用户代码执行之前被调用。

函数原型

void PreTaskHook(void);

项目	说明
功能	任务开始运行前的回调
调用时机	Task 从 READY → RUNNING 转换后，Task 入口函数前
参数	无
返回值	无
调用上下文	OS_LEVEL_PRETASKHOOK (Level 6)
启用条件	CFG_PRETASKHOOK = TRUE
可调用 API	GetTaskID, GetTaskState 等
典型用途	任务执行时间统计、任务切换跟踪、调试日志

调用时序

```
调度器选中 TaskA
  ├─ TaskA.state = RUNNING
  ├─ PreTaskHook()              // ◄── 每次 TaskA 获得 CPU 时
  │     └─ GetTaskID(&taskId)   // 获取当前任务 ID
  └─ TaskA 用户代码开始执行
```

实现示例

```
void PreTaskHook(void)
{
  TaskType currentTask;
  (void)GetTaskID(&currentTask);

  /* 记录任务开始时间 */
  (void)GetCounterValue(SystemCounter, &taskStartTime[currentTask]);

  /* 任务切换计数 */
  taskSwitchCount++;
}
```

注意：PreTaskHook 会在每次任务被调度执行时调用（包括被抢占后恢复执行），因此执行频率可能很高。Hook 实现应尽量简短，避免影响系统实时性。

12.3.2 PostTaskHook（任务切出后钩子）

PostTaskHook 在任务离开 RUNNING 状态之前被调用，即任务即将被抢占或终止时执行。

函数原型

```
void PostTaskHook(void);
```

项目	说明
功能	任务离开运行状态前的回调
调用时机	Task 从 RUNNING 离开前（被抢占、等待事件、终止）
参数	无
返回值	无
调用上下文	OS_LEVEL_POSTTASKHOOK (Level 7)
启用条件	CFG_POSTTASKHOOK = TRUE
可调用 API	GetTaskID, GetTaskState 等
典型用途	计算任务执行时间、资源使用审计

调用时序

```
TaskA 正在运行...
  ├─ TerminateTask() 或被高优先级任务抢占
  ├─ PostTaskHook()          // ◄── TaskA 离开 RUNNING 前
  │     └─ 计算 TaskA 执行时间
  └─ TaskA.state = SUSPENDED / READY / WAITING
```

实现示例：任务执行时间统计

```
void PostTaskHook(void)
{
  TaskType currentTask;
  TickType elapsed;

  (void)GetTaskID(&currentTask);
```

```
    /* 计算任务执行时间 */
    (void)GetElapsedValue(SystemCounter,
                          &taskStartTime[currentTask],
                          &elapsed);

    /* 更新最大执行时间记录 */
    if (elapsed > taskMaxExecTime[currentTask])
    {
        taskMaxExecTime[currentTask] = elapsed;
    }
}
```

PreTaskHook 与 PostTaskHook 配合使用，可以精确测量每个任务的执行时间。这对于验证时间保护参数（Execution Budget）的合理性非常有价值。

12.4 Hook 函数实验

12.4.1 实验设计

本实验演示如何使用 Hook 函数实现运行时错误监控和任务执行时间统计。

实验目标

- 验证 StartupHook 在 OS 启动时被正确调用
- 验证 ErrorHook 能捕获 API 调用错误
- 使用 PreTaskHook/PostTaskHook 统计任务执行时间
- 验证 ShutdownHook 在系统关闭时被调用

配置修改

在 Os_Cfg.h 中启用所有 Hook：

```
/* Os_Cfg.h - 实验配置 */
#define CFG_ERRORHOOK        TRUE
#define CFG_PRETASKHOOK      TRUE    /* 修: FALSE
#define CFG_POSTTASKHOOK     TRUE    /* 修: FALSE
#define CFG_STARTUPHOOK      TRUE
#define CFG_SHUTDOWNHOOK     TRUE
```

实验代码

```
/* ===== 全局变量 ===== */
static volatile uint32 hookCallCount[5] = {0};  /* 各 Hook 调用计数 */
/* 0:Startup, 1:Shutdown, 2:Error, 3:PreTask, 4:PostTask */

static TickType taskStartTime[CFG_TASK_MAX];
static TickType taskMaxExecTime[CFG_TASK_MAX];
static volatile uint32 errorLog[16];
static volatile uint8 errorLogIndex = 0;
```

```
/* ===== StartupHook ===== */
void StartupHook(void)
{
    hookCallCount[0]++;

    /* 初始化统计数组 */
    for (uint32 i = 0; i < CFG_TASK_MAX; i++)
    {
        taskStartTime[i] = 0U;
        taskMaxExecTime[i] = 0U;
    }

    /* 启动周期 Alarm */
    (void)SetRelAlarm(AlarmBlink, 500U, 500U);
}

/* ===== ShutdownHook ===== */
void ShutdownHook(StatusType Error)
{
    hookCallCount[1]++;
    volatile StatusType reason = Error;
    (void)reason;
}

/* ===== ErrorHook ===== */
void ErrorHook(StatusType Error)
{
    hookCallCount[2]++;

    if (errorLogIndex < 16U)
    {
        errorLog[errorLogIndex] = (uint32)Error;
        errorLogIndex++;
    }
}

/* ===== PreTaskHook ===== */
void PreTaskHook(void)
{
    hookCallCount[3]++;

    TaskType tid;
    (void)GetTaskID(&tid);
    (void)GetCounterValue(SystemCounter, &taskStartTime[tid]);
}

/* ===== PostTaskHook ===== */
void PostTaskHook(void)
{
    hookCallCount[4]++;

    TaskType tid;
    TickType elapsed;
    (void)GetTaskID(&tid);
    (void)GetElapsedValue(SystemCounter,
                    &taskStartTime[tid], &elapsed);
    if (elapsed > taskMaxExecTime[tid])
    {
```

```
        taskMaxExecTime[tid] = elapsed;
    }
}
```

测试任务

```
/* Task_Init - 触发一次 ErrorHook */
void Os_TaskEntry_Task_Init(void)
{
    /* 故意使用无效参数触发 ErrorHook */
    StatusType err = ActivateTask(0xFFU); /* 无效 TaskID */
    /* ErrorHook(E_OS_ID) 将被自动调用 */

    (void)err;
    (void)TerminateTask();
}
```

12.4.2 运行结果分析

通过调试器观察以下全局变量来验证 Hook 函数的正确执行：

预期结果

变量	预期值	说明
hookCallCount[0]	1	StartupHook 被调用 1 次
hookCallCount[1]	0	ShutdownHook 未被触发（系统正常运行）
hookCallCount[2]	≥1	ErrorHook 至少被调用 1 次（Task_Init 触发）
hookCallCount[3]	递增	PreTaskHook 每次任务调度时递增
hookCallCount[4]	递增	PostTaskHook 每次任务离开时递增
errorLog[0]	3 (E_OS_ID)	第一个错误为无效 Task ID
taskMaxExecTime[x]	>0	各任务最大执行时间(ticks)

调试方法

- 在调试器中设置断点于各 Hook 函数入口
- 观察 hookCallCount 数组验证调用次数
- 通过 errorLog 数组追溯历史错误
- 通过 taskMaxExecTime 数组评估各任务执行时间

性能影响分析

启用 Hook 函数会引入额外的运行时开销：

Hook	开销来源	影响程度
StartupHook	仅启动时调用一次	可忽略
ShutdownHook	仅关闭时调用一次	可忽略
ErrorHook	仅错误时调用	正常运行无影响
PreTaskHook	每次任务调度时调用	中等（取决于任务切换频率）
PostTaskHook	每次任务离开时调用	中等（取决于任务切换频率）

12.5 本章小结

本章详细讲解了 AUTOSAR OS 的 Hook 回调函数机制。系统级 Hook 包括：StartupHook 在 OS 启动后、调度器运行前执行硬件初始化；ShutdownHook 在 OS 关闭时执行清理和错误记录；ErrorHook 在 API 返回错误时统一捕获并记录错误信息；ProtectionHook（SC3/SC4）在保护违规时决定恢复策略。任务级 Hook 包括：PreTaskHook 在任务进入 RUNNING 后、用户代码执行前调用，PostTaskHook 在任务离开 RUNNING 状态前调用，常用于运行时性能分析和任务切换追踪。实验验证了各 Hook 的触发时机和执行顺序。建议在开发阶段启用 Hook 进行调试分析，量产版本关闭以减少开销。

建议：在开发和调试阶段启用 PreTaskHook/PostTaskHook 进行性能分析，在量产版本中关闭以减少开销。ErrorHook 和 StartupHook 建议始终保持启用。

第十三章 多核 OS

13.1 AUTOSAR 多核 OS 概述

AUTOSAR OS 从 R4.0 版本开始引入多核扩展。在多核系统中，每个核心运行独立的 OS 实例，拥有独立的调度队列、Task 集合和中断管理。OS 对象（Task、ISR、Counter、Alarm 等）在配置阶段静态绑定到特定核心。

TC3xx 多核硬件架构

特性	说明
CPU0 (Master Core)	上电后第一个运行的核，负责启动其他核
CPU1/CPU2 (Slave Core)	需由 CPU0 通过 StartCore() 启动
共享内存 (Shared RAM)	多核数据交换区域，需 Spinlock 保护
IPC 机制	核间中断 + 共享内存实现 RPC 调用
SCU (System Control Unit)	管理多核启动、复位和低功耗模式

AUTOSAR 多核 OS 核心设计原则

- 每个核心独立调度：独立的 Ready Queue、Running Task、ISR 管理
- OS 对象绑定核心：Task/ISR/Counter/Alarm 在配置时绑定到特定 Core
- 核间通信标准化：通过 IOC 实现跨核数据传输
- 自旋锁保护共享资源：Spinlock 用于多核互斥访问
- 启动同步屏障：Barrier 确保多核启动同步

当前项目配置

```
/* Os_Cfg.h - 当前为单核配置 */
#define CFG_CORE_MAX                (1U)
#define OS_AUTOSAR_CORES            1U
#define CFG_CORE0_AUTOSAROS_ENABLE  TRUE
#define CFG_SPINLOCK_MAX            (0U)     /* 无 Spinlock */
#define CFG_IOC_MAX                 (0U)     /* 无 IOC */
```

与 FreeRTOS 对比

特性	FreeRTOS SMP	AUTOSAR OS Multi-Core
任务分配	任务可在核间迁移	任务静态绑定到核
核间通信	无专用机制，使用 Queue/Semaphore	IOC 专用跨核通信通道
互斥保护	portENTER_CRITICAL_FROM_ISR	GetSpinlock / Spinlock API
启动方式	所有核同时启动调度器	Master 核依次启动 Slave 核
对象归属	共享所有 OS 对象	OS 对象严格归属特定核

13.2 核间通信机制

13.2.1 IOC（Inter-OS-Application Communication）

IOC 是 AUTOSAR OS 提供的标准核间通信机制，支持 Queued（队列式）和 Unqueued（非队列式）两种通信模型。IOC 在内部通过 Spinlock 保护共享缓冲区，确保跨核数据一致性。

IOC 通信模型

模型	特点	适用场景
Queued (队列式)	FIFO 缓冲区，数据不丢失（直到满）	流式数据传输，如传感器数据流
Unqueued (非队列式)	最新值覆盖旧值，始终保存最新数据	状态同步，如共享标志位

IOC API 详解

【IocSendQueue】向队列型 IOC 通道发送数据

StatusType IocSendQueue(IocType IocID, const Ioc_DataType* Data);

项目	说明
参数 IocID	IOC 通道标识符
参数 Data	指向待发送数据的指针
返回 IOC_E_OK (0)	发送成功
返回 IOC_E_LIMIT (130)	队列已满
返回 IOC_E_NOK (1)	通用失败
内部机制	跨核时使用 Spinlock 保护共享缓冲区

【IocReceiveQueue】从队列型 IOC 通道接收数据

StatusType IocReceiveQueue(IocType IocID, Ioc_DataType* Data);

项目	说明
参数 IocID	IOC 通道标识符
参数 Data	输出数据缓冲区指针
返回 IOC_E_OK (0)	接收成功
返回 IOC_E_NO_DATA (131)	队列为空，无数据可读

【IocWrite / IocRead】非队列式写入/读取

StatusType IocWrite(IocType IocID, const Ioc_DataType* Data);
StatusType IocRead(IocType IocID, Ioc_DataType* Data);

项目	说明
IocWrite 返回 IOC_E_LOST_DATA (64)	上次数据未被读取即被覆盖
IocRead 返回 IOC_E_NO_DATA (131)	无数据可读
适用场景	跨核状态同步，如最新传感器值

【IocEmptyQueue / IocOverwrite】

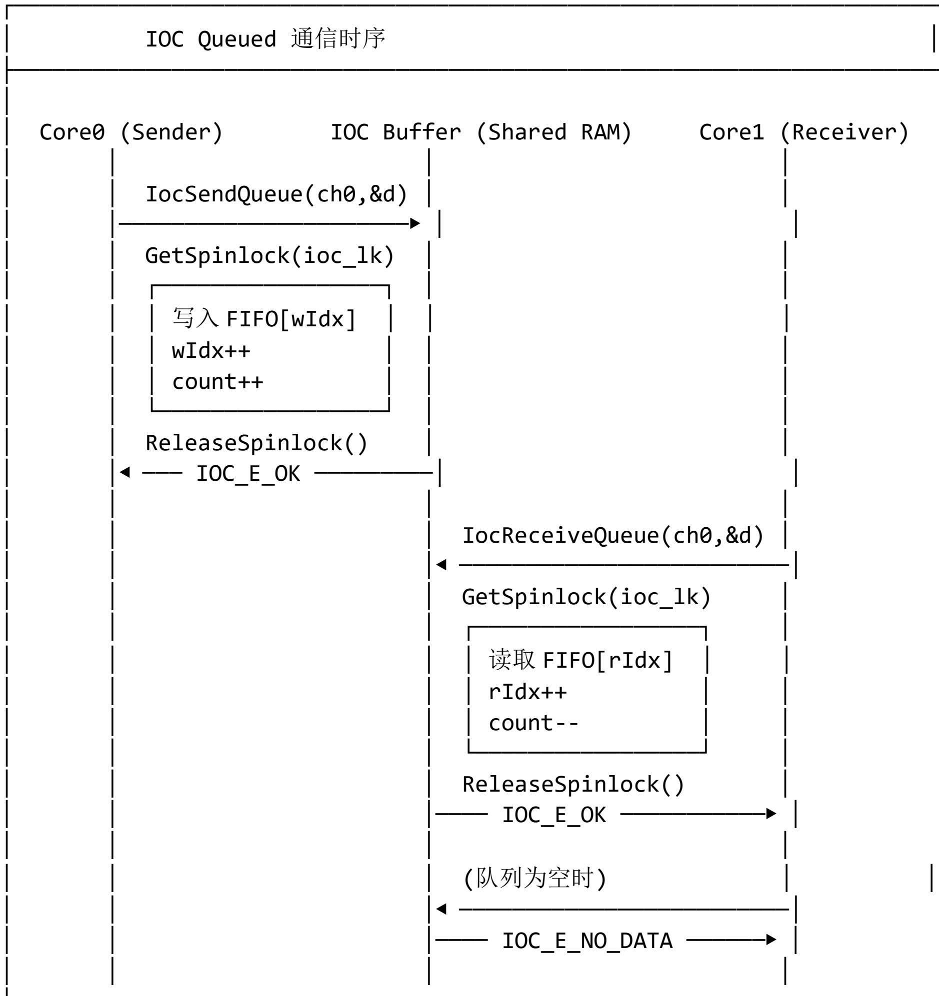
StatusType IocEmptyQueue(IocType IocID); // 清空队列
StatusType IocOverwrite(IocType IocID, const Ioc_DataType* Data); // 无条件覆盖

IOC 错误码汇总

错误码	数值	含义
IOC_E_OK	0	操作成功
IOC_E_NOK	1	通用失败
IOC_E_LOST_DATA	64	数据被覆盖（Unqueued Write）
IOC_E_LIMIT	130	队列已满

IOC_E_NO_DATA	131	无数据可读
IOC_E_LENGTH	132	数据长度错误

【时序图】IOC 通信时序（发送方 → 缓冲区 → 接收方）



IOC 内部保护机制

```
/* Os_Ioc.c - 跨核 IOC 使用内部 Spinlock 保护 */
#if (OS_AUTOSAR_CORES > 1U)
#define IOC_LOCK(com)                                                    \
  if (CROSS_CORE_COM == Os_IocCommunicationCfg[com].IocCfgComMark) \
    { Os_GetInternalSpinlock(&Os_IocSpinlock[com]); }
#define IOC_UNLOCK(com)                                                  \
  if (CROSS_CORE_COM == Os_IocCommunicationCfg[com].IocCfgComMark) \
    { Os_ReleaseInternalSpinlock(&Os_IocSpinlock[com]); }
#else
#define IOC_LOCK(com)    /* 单核为空宏 */
#define IOC_UNLOCK(com)
#endif
```

13.2.2 Spinlock（自旋锁）

Spinlock（自旋锁）是 AUTOSAR OS 多核环境中保护共享资源的同步原语。与 Mutex 不同，

Spinlock 采用忙等待（busy-wait）策略，适用于短临界区保护。

GetSpinlock

StatusType GetSpinlock(SpinlockIdType SpinlockId);

项目	说明
功能	获取自旋锁，若锁被占用则忙等待
参数 SpinlockId	自旋锁标识符
返回 E_OK	获取成功
返回 E_OS_ID	无效的 SpinlockId
返回 E_OS_INTERFERENCE_DEADLOCK	同核重复获取（自死锁检测）
返回 E_OS_NESTING_DEADLOCK	嵌套顺序错误（交叉死锁检测）
返回 E_OS_ACCESS	无访问权限
返回 E_OS_CALLEVEL	调用上下文错误
禁止操作	获取后不得调用 WaitEvent/TerminateTask/Schedule

ReleaseSpinlock

StatusType ReleaseSpinlock(SpinlockIdType SpinlockId);

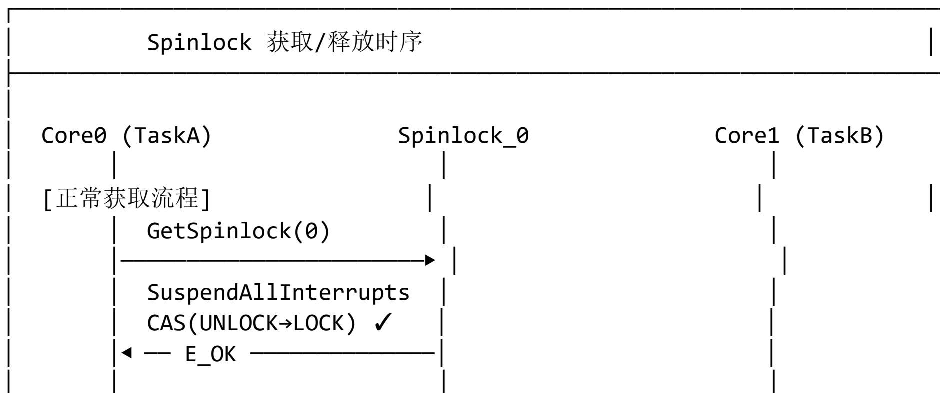
项目	说明
功能	释放自旋锁
返回 E_OK	释放成功
返回 E_OS_STATE	锁未由当前 Task/ISR 占用
返回 E_OS_NOFUNC	LIFO 释放顺序违反
约束	必须与 GetSpinlock 成对调用，遵循 LIFO 释放顺序

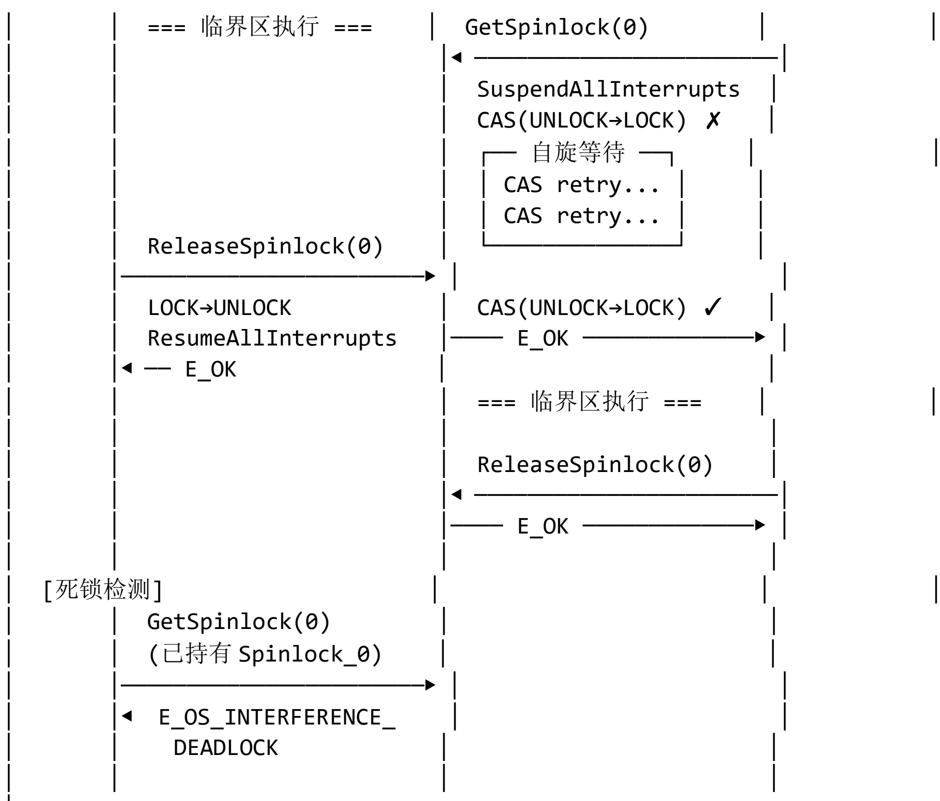
TryToGetSpinlock

StatusType TryToGetSpinlock(SpinlockIdType SpinlockId, TryToGetSpinlockType* Success);

项目	说明
功能	非阻塞式尝试获取自旋锁
参数 Success	TRYTOGETSPINLOCK_SUCCESS=获取成功 / TRYTOGETSPINLOCK_NOSUCCESS=锁被占用
返回 E_OK	API 调用成功（需检查 Success 参数判断是否获取锁）
适用场景	ISR 中或不允许长时间等待的场景

【时序图】Spinlock 获取/释放与死锁检测





Spinlock 内部实现

```
static void Os_GetSpinlock(SpinlockIdType SpinlockId)
{
  /* 1. 根据 SpinlockMethod 锁定中断 */
  switch (pSpinlockCfg->SpinlockMethod) {
  case LOCK_ALL_INTERRUPTS:    Os_SuspendAllInterrupts(); break;
  case LOCK_CAT2_INTERRUPTS:   Os_SuspendOSInterrupts();  break;
  case LOCK_WITH_RES_SCHEDULER: Os_GetResource(RES_SCHEDULER); break;
  case LOCK_NOTHING:           break;
  }

  /* 2. 原子 CAS 操作获取锁（忙等待） */
  do {
    result = Os_CmpSwapW(&Os_Spinlock[SpinlockId],
                         OS_SPINLOCK_UNLOCK, OS_SPINLOCK_LOCK);
  } while (result > 0u);  /* 自旋直到获取成功 */
}
```

13.3 多核启动流程

13.3.1 StartCore（启动核心）

StartCore

void StartCore(CoreIdType CoreID, StatusType* Status);

项目	说明
功能	启动一个受 AUTOSAR OS 管理的核心

参数 CoreID	要启动的核心 ID
参数 Status	输出参数，返回操作结果
返回 E_OS_ACCESS	在 StartOS() 之后调用
返回 E_OS_ID	CoreID 超过 OS_AUTOSAR_CORES
返回 E_OS_STATE	目标核已启动
调用时机	仅在 StartOS() 之前的 main() 中调用
实现原理	写入目标核 PCPTR (Program Counter Pointer) 寄存器

StartNonAutosarCore（⚠ R20-11 已移除）

void StartNonAutosarCore(CoreIdType CoreID, StatusType* Status); /* ⚠ R20-11 已移除，仅 R19-11 及之前版本可用 */ /* ⚠ R20-11 已移除 */

项目	说明
功能	启动一个不受 AUTOSAR OS 管理的核心（运行裸机代码）
调用时机	可在 StartOS() 之后调用
编译条件	CFG_CORE_MAX > 1 时才编译

GetCoreID / GetNumberOfActivatedCores

```
CoreIdType GetCoreID(void);              // 获取当前核逻辑 ID
uint32 GetNumberOfActivatedCores(void);    // 获取已激活核数量
```

ControlIdle（⚠ R25-11 已移除） / ShutdownAllCores

StatusType ControlIdle(CoreIdType CoreID, IdleModeType IdleMode); /* ⚠ R25-11 已移除此 API */ /* ⚠ R25-11 已移除 */

void ShutdownAllCores(StatusType Error);

API	功能
ControlIdle	控制指定核在空闲时的行为（运行/休眠）
ShutdownAllCores	关闭所有核的 OS，通过 RPC 通知每个核执行 ShutdownOS

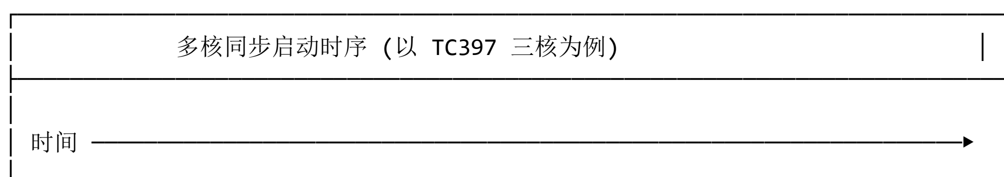
13.3.2 核同步屏障

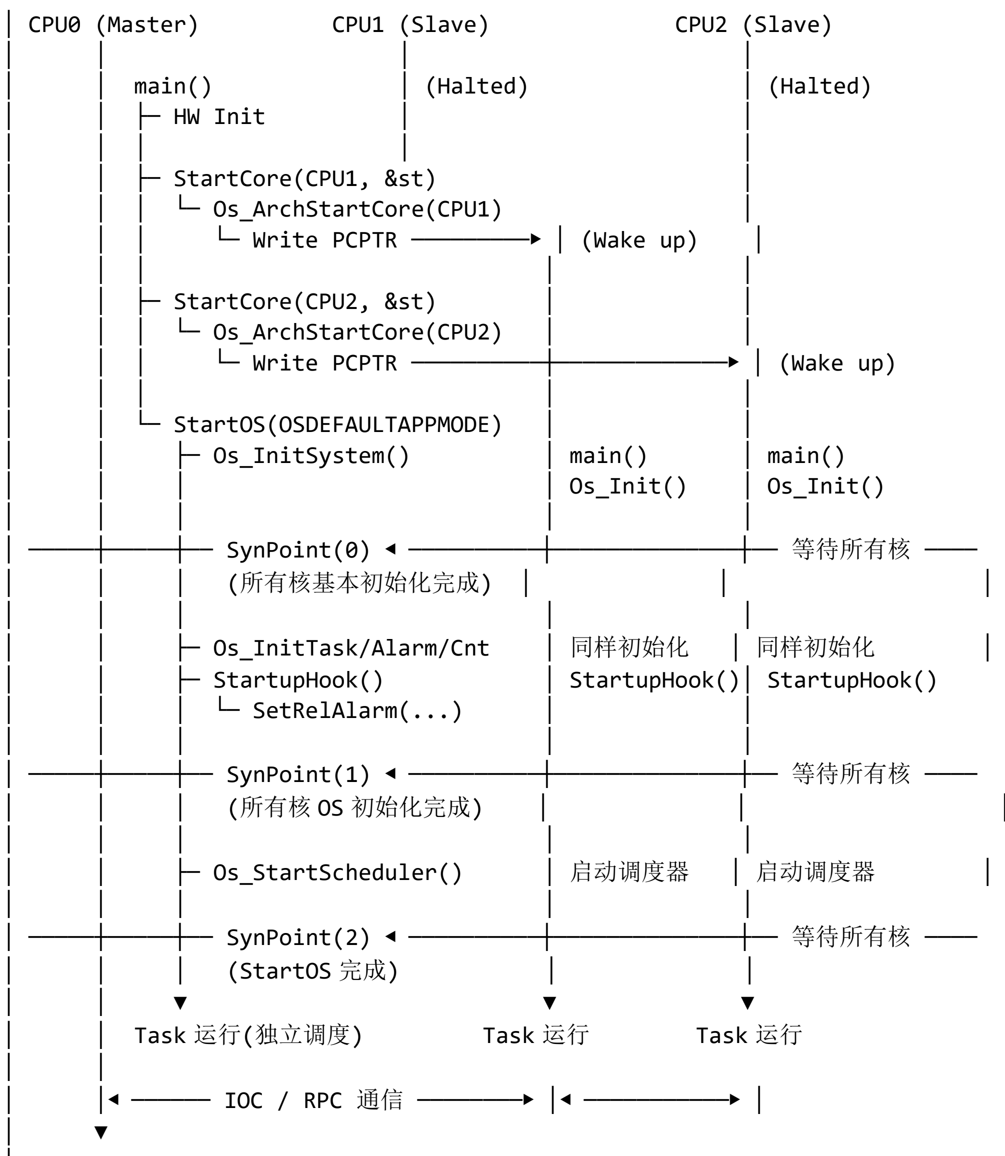
AUTOSAR OS 定义了 4 个同步点（SynPoint 0-3），用于多核启动和关闭过程的同步：

同步点	阶段	用途
SynPoint 0	启动	所有核完成基本初始化后同步
SynPoint 1	启动	所有核完成 OS 初始化后同步
SynPoint 2	启动	标记 StartOS() 完成
SynPoint 3	关闭	ShutdownAllCores() 中同步所有核关闭

同步屏障原理：每个核到达同步点后设置自己的标志位，然后轮询等待其他核的标志位。所有核的标志位都设置后，屏障解除。

【时序图】多核同步启动流程（StartCore → 屏障 → StartOS）





13.4 跨核任务激活与事件设置

在多核系统中，一个核心上的 Task/ISR 可以激活另一个核心上的 Task 或设置事件。AUTOSAR OS 内部通过 RPC（Remote Procedure Call）机制实现跨核操作。

跨核 API

```
StatusType ActivateTaskAsyn(TaskType TaskID);  // 异步跨核激活
StatusType SetEventAsyn(TaskType TaskID, EventMaskType Mask); // 异步跨核事件
```

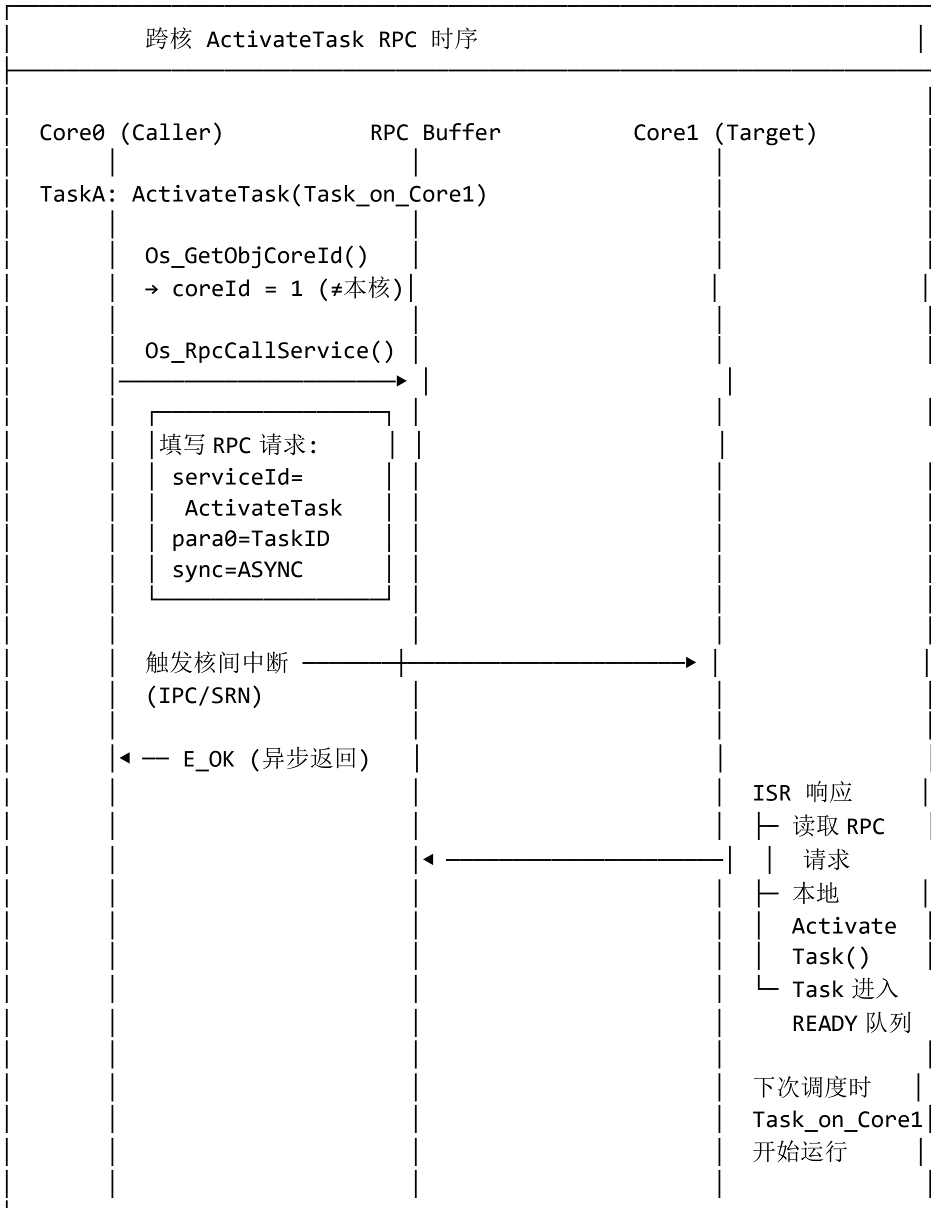
RPC 机制原理

当 ActivateTask() 检测到目标 Task 不在本核时，自动转为 RPC 调用：

```
/* Os_Task.c - 跨核路径 */
#if (OS_AUTOSAR_CORES > 1)
  Os_CoreIdType coreId = Os_GetObjCoreId(TaskID);
```

```
  if (coreId != Os_SCB.sysCore)
  {
    RpcInputType rpcData = {
      .sync        = RPC_ASYNC,
      .remoteCoreId = coreId,
      .serviceId    = OSServiceId_ActivateTask,
      .srvPara0     = (uint32)TaskID,
    };
    err = Os_RpcCallService(&rpcData);
  }
#endif
```

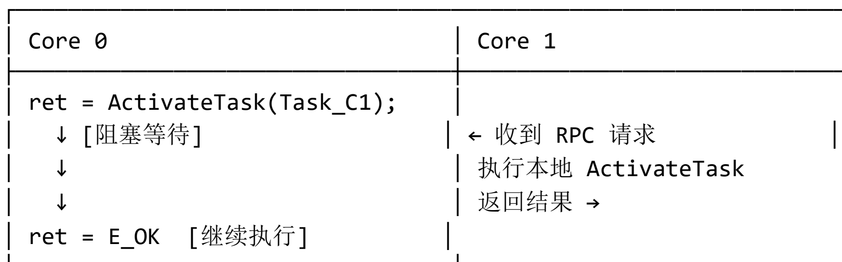
【时序图】跨核 ActivateTask 的 RPC 机制时序



13.4.1 跨核 ActivateTask 的同步阻塞语义（SWS_Os_00598）

⚠ 重要：AUTOSAR OS 的跨核 ActivateTask 是同步操作，非异步！与 FreeRTOS SMP 不同，AUTOSAR OS 跨核 API 调用遵循同步阻塞语义。

同步阻塞行为时序



语义要点

- 1. 调用核（Core 0）会阻塞，直到目标核完成激活操作
- 2. 返回值来自目标核的实际执行结果（E_OK / E_OS_LIMIT / E_OS_ID 等）
- 3. 若目标核未启动或不响应 → 返回 E_OS_CORE
- 4. 阻塞期间，调用核的中断仍可响应（取决于实现）

与 FreeRTOS SMP 的对比

维度	AUTOSAR OS Multi-Core	FreeRTOS SMP
跨核激活方式	同步阻塞 RPC	异步通知（yield 触发）
返回时机	目标核完成后返回	立即返回
错误感知	可立即感知激活失败	无法感知远端结果
延迟确定性	取决于目标核响应速度	极低延迟（仅写队列）
适用场景	安全关键（需确认）	高吞吐（可容忍丢失）

注意：本项目源码中的 ActivateTaskAsyn() 是非标准扩展 API，提供类似 FreeRTOS 的异步语义，但不属于 AUTOSAR 官方规范。上述时序图中的 sync=ASYNC 为本项目扩展实现，标准 ActivateTask() 默认为同步模式。

13.5 多核实验

13.5.1 实验设计

本实验以 TC397（三核）为假设平台，设计双核 IOC 通信实验：

- CPU0：运行周期任务 Task_Sender，通过 IOC 发送递增计数器到 CPU1
- CPU1：运行 Task_Receiver，接收 IOC 数据并通过 LED 显示状态
- 使用 Spinlock 保护跨核共享变量

配置修改（Os_Cfg.h）

```
#define CFG_CORE_MAX                    (2U)    /* 2 核 */
#define OS_AUTOSAR_CORES                2U
```

```
#define CFG_CORE0_AUTOSAROS_ENABLE  TRUE
#define CFG_CORE1_AUTOSAROS_ENABLE  TRUE
#define CFG_SPINLOCK_MAX            (1U)       /* 1 个 Spinlock */
#define CFG_IOC_MAX                 (1U)    /* 1 个 IOC 通 */

/* Core1 任务定义 */
#define CFG_TASK_MAX_CORE1          (2U)
#define Task_Receiver               ((Os_TaskType)0x1000U)
#define OS_TASK_IDLE_CORE1          ((Os_TaskType)0x1001U)

/* IOC / Spinlock 定义 */
#define Ioc_Channel_0               ((IocType)0U)
#define Spinlock_0                  ((SpinlockIdType)0U)
```

实验代码

【main.c - 双核启动】

```
int main(void)
{
  StatusType status;

  /* 硬件初始化 */
  IfxScuWdt_disableCpuWatchdog(IfxScuWdt_getCpuWatchdogPassword());

  /* 启动 CPU1（必须在 StartOS 之前） */
  StartCore(OS_CORE_ID_1, &status);
  if (status != E_OK) { for(;;){} }

  /* CPU0 启动 OS */
  StartOS(OSDEFAULTAPPMODE);
  return 0;
}
```

【Task_Sender - CPU0 发送任务】

```
void Os_TaskEntry_Task_Sender(void)
{
  static uint32 counter = 0U;
  Ioc_DataType sendData = (Ioc_DataType)counter;

  StatusType result = IocSendQueue(Ioc_Channel_0, &sendData);
  if (result == IOC_E_LIMIT) {
    /* 队列满，可选择丢弃或等待 */
    IocEmptyQueue(Ioc_Channel_0);
  }

  counter++;
  (void)TerminateTask();
}
```

【Task_Receiver - CPU1 接收任务】

```
void Os_TaskEntry_Task_Receiver(void)
{
  Ioc_DataType recvData;
  StatusType result = IocReceiveQueue(Ioc_Channel_0, &recvData);
```

```
    if (IOC_E_OK == result) {
        if (recvData & 1U)
            IfxPort_setPinHigh(&MODULE_P10, 2);  /* LED ON */
        else
            IfxPort_setPinLow(&MODULE_P10, 2);   /* LED OFF */
    }

    (void)TerminateTask();
}
```

【Spinlock 保护共享变量】

```
static volatile uint32 sharedCounter = 0U;

void IncrementSharedCounter(void)
{
    (void)GetSpinlock(Spinlock_0);
    sharedCounter++;
    (void)ReleaseSpinlock(Spinlock_0);
}
```

13.5.2 运行结果分析

验证项目

验证项	方法	预期结果
双核启动	GetCoreID() 在 StartupHook 中打印	CPU0=0, CPU1=1
IOC 通信	CPU0 发送递增计数	CPU1 LED 按节拍闪烁
Spinlock	两核同时递增 sharedCounter	最终值 = 两核操作次数之和
Barrier 同步	GPIO 翻转 + 示波器	SynPoint 处两核几乎同时通过
ShutdownAllCores	调用后观察	所有核进入死循环

调试技巧

- 使用 UDE/Lauterbach TRACE32 同时连接多核调试
- 在各核 StartupHook 中设断点，验证启动顺序
- IOC 通信验证：在 IocSendQueue 前后设断点观察缓冲区状态
- Spinlock 验证：同时暂停两核，检查 Os_Spinlock[] 数组状态

常见问题

问题	原因	解决方案
Slave 核不启动	StartCore 在 StartOS 之后调用	确保 StartCore 在 StartOS 前
IOC 数据错误	未使用 Spinlock 保护	检查 IOC 配置的 CrossCore 标记
E_OS_INTERFERENCE_DEADLOCK	同核重复获取同一 Spinlock	检查代码逻辑，避免递归获取
E_OS_NESTING_DEADLOCK	Spinlock 嵌套顺序错误	严格按 ID 升序获取 Spinlock

13.6 从单核迁移到多核的实战指南

当前项目配置为单核（CFG_CORE_MAX=1），以下是升级到多核（如 TC397 双核/三核）的

具体步骤。

13.6.1 修改核心配置宏（Os_Cfg.h）

```
// 修改前（单核）
#define CFG_CORE_MAX              1U
#define OS_AUTOSAR_CORES          1U
#define CFG_CORE0_AUTOSAROS_ENABLE TRUE

// 修改后（双核）
#define CFG_CORE_MAX              2U
#define OS_AUTOSAR_CORES          2U
#define CFG_CORE0_AUTOSAROS_ENABLE TRUE
#define CFG_CORE1_AUTOSAROS_ENABLE TRUE

// 新增 Core1 配置
#define CFG_TASK_MAX_CORE1        2U
#define CFG_ISR_MAX_CORE1         1U
#define CFG_COUNTER_MAX_CORE1     1U
#define CFG_ALARM_MAX_CORE1       1U
```

13.6.2 启用 Spinlock

```
// Os_Cfg.h
#define CFG_SPINLOCK_MAX          2U    // 定 2 个

// Os_Cfg.c 新增
const Os_SpinlockCfgType Os_SpinlockCfg[CFG_SPINLOCK_MAX] = {
    {OS_SPINLOCK_LOCK_CAT_ALL},    // Spinlock_SharedData
    {OS_SPINLOCK_LOCK_CAT_OS},     // Spinlock_OSResource
};

// 使用示例
StatusType ret;
GetSpinlock(Spinlock_SharedData);
 /* 跨核共享数据访问 */
 shared_buffer[idx] = new_value;
ReleaseSpinlock(Spinlock_SharedData);
```

13.6.3 配置 IOC 通信通道

```
// Os_Cfg.h
#define CFG_IOC_MAX               1U    // 1 个 IOC 通

// 典型 IOC 使用模式
// Core0 (发送方)
Std_ReturnType IocWrite_Channel0(const IocDataType* data) {
  // OS 内部使用 Spinlock 保护
  return IOC_E_OK;
}

// Core1 (接收方)
Std_ReturnType IocRead_Channel0(IocDataType* data) {
  return IOC_E_OK;
```

```
}
```

13.6.4 多核启动代码

```
// Cpu0_Main.c (主核)
int core0_main(void) {
    StatusType status;

    // 1. 启动从核
    StartCore(OS_CORE_ID_1, &status);
    if (status != E_OK) { /* 错误处理 */ }

    // 2. 启动 OS（内部有核间同步屏障）
    StartOS(OSDEFAULTAPPMODE);
    return 0;  // 不会到达
}

// Cpu1_Main.c (从核)
int core1_main(void) {
    // 从核等待主核的 StartCore 调用
    StartOS(OSDEFAULTAPPMODE);
    return 0;
}
```

13.6.5 跨核任务激活

```
// Core0 中激活 Core1 上的任务
StatusType ret = ActivateTask(Task_Core1_Worker);
// OS 内部通过 RPC 机制转发到 Core1

// 跨核事件设置
SetEvent(Task_Core1_Extended, EVT_DATA_READY);
```

13.6.6 迁移检查清单

- □ 确定任务-核心分配策略（性能关键任务放哪个核）
- □ 识别跨核共享数据（需 Spinlock/IOC 保护）
- □ 配置核间中断（用于 RPC 机制）
- □ 更新链接脚本（每个核独立的栈和 CSA 段）
- □ 验证中断源与核的绑定关系
- □ 测试核同步启动时序
- □ 压力测试跨核通信延迟

13.6.7 常见陷阱

- 1. 忘记为跨核共享变量添加 volatile 修饰
- 2. Spinlock 未按获取顺序释放（导致 E_OS_NESTING_DEADLOCK）

- 3. IOC 缓冲区大小不足（导致 IOC_E_LIMIT）

13.7 本章小结

本章系统介绍了 AUTOSAR OS 的多核支持。多核 OS 架构中每个核运行独立的 OS 实例，共享全局配置。核间通信通过 IOC（支持带队列和不带队列两种模式）和 Spinlock 实现，Spinlock 使用硬件原子指令保证临界区互斥。多核启动流程由 Master Core 调用 StartCore 唤醒 Slave Core，通过 StartOS 同步屏障确保所有核同步进入运行状态。跨核任务激活具有同步阻塞语义（SWS_Os_00598），确保远程核完成激活后才返回。实验验证了双核协同工作和核间通信机制。本章还提供了从单核迁移到多核的完整实战指南和检查清单。

第十四章 时间管理

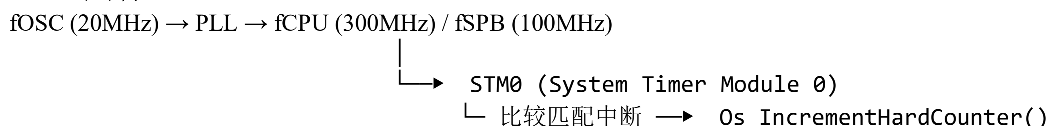
14.1 OS 系统 Tick 与时钟源

AUTOSAR OS 的时间管理基于 Counter 机制。SystemCounter 是最核心的硬件计数器，由底层硬件定时器（STM）驱动，为 Alarm 和 Schedule Table 提供时基。

TC334 系统定时器（STM）

TC334 的 SystemCounter 由 STM0（System Timer Module 0）硬件驱动。STM 是一个 64 位自由运行递增计数器，频率为系统外设总线时钟（fSPB）。

TC334 时钟树:



SystemCounter 配置

当前项目 SystemCounter 配置（Os_Cfg.c）：

```
static const Os_CounterCfgType Os_CounterCfgCore0[] = {
  {
    2147483647U,      /* MaxAllowedValue: 2^31-1 */
    1U,               /* MinCycle: 最小周期 1 tick */
    1U,               /* TicksPerBase: 每次递增 1 tick */
    1000U,            /* SecondsPerTick: 1000 μs = 1 ms */
    COUNTER_HARDWARE, /* CounterType: 硬件计数器 */
  },
};
```

参数	值	含义
MaxAllowedValue	2147483647 (0x7FFFFFFF)	计数器最大值
MinCycle	1	Alarm 最小周期 1 tick
TicksPerBase	1	每次 ISR 递增 1 tick
SecondsPerTick	1000 ns	1 tick 的时间长度 = 1 us
CounterType	COUNTER_HARDWARE	由硬件定时器 ISR 驱动

定时器 ISR 配置

```
/* STM0 定时器周期 */
#define CFG_REG_OSTIMER_VALUE_CORE0    (100000U)

/* 计算: fSPB = 100MHz, 周期 = 100000/100MHz = 1ms */
/* 因此: 每 1ms 触发一次 ISR, 递增 SystemCounter 一次 */
/* 即: 1 tick = 1 ms */
```

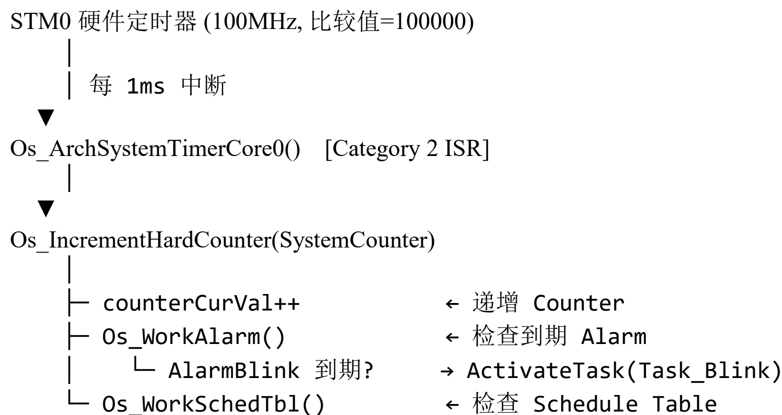
注意：虽然 SecondsPerTick = 1000 表示 1 tick = 1000 μs = 1 ms，与 STM ISR 周期一致（CFG_REG_OSTIMER_VALUE_CORE0 = 100000，fSPB = 100MHz，周期 = 1ms）。

系统定时器 ISR

```
/* Os_Cfg.c - 定时器 ISR 回调 */
void Os_ArchSystemTimerCore0(void)
```

```
{
    (void)Os_IncrementHardCounter(SystemCounter);
}

/* ISR 配置 */
static const Os_IsrCfgType Os_IsrCfgCore0[] = {
    {
        OS_ISR_STM0_SR0,     /* STM0 Service Request 0 */
        OS_ARCH_INT_CPU0,    /* CPU0 中断 */
        OS_ISR_CATEGORY2,    /* Category 2 ISR */
        FALSE,               /* 不允许嵌套 */
    },
};
```

Counter 驱动流程



时间转换宏

Os_Cfg.h 定义了 tick 与时间单位之间的转换宏：

```
#define OS_TICKS2NS_SystemCounter(ticks)   ((ticks)*1000*1000)
#define OS_TICKS2US_SystemCounter(ticks)   ((ticks)*1000)
#define OS_TICKS2MS_SystemCounter(ticks)   ((ticks)*1000/1000)    /* = ticks */
#define OS_TICKS2SEC_SystemCounter(ticks)  ((ticks)*1000/1000000)
#define OS_NS2TICKS_SystemCounter(ns)      ((ns)/1000/1000)
#define OS_US2TICKS_SystemCounter(us)      ((us)/1000)
#define OS_MS2TICKS_SystemCounter(ms)      ((ms)*1000/1000)       /* = ms */
#define OS_SEC2TICKS_SystemCounter(sec)    ((sec)*1000000/1000)
```

实际使用中，由于 1 tick = 1 ms，OS_TICKS2MS 和 OS_MS2TICKS 的换算结果均等于输入值本身。

与 FreeRTOS 对比：FreeRTOS 使用 configTICK_RATE_HZ 定义 tick 频率（常见 1000Hz = 1ms），通过 pdMS_TO_TICKS() 宏转换。AUTOSAR OS 的 Counter 更灵活，支持多个独立计数器和不同时基。

14.2 GetCounterValue / GetElapsedValue（获取计数器值/已经过值）

14.2.1 GetCounterValue（获取计数器当前值）

```
StatusType GetCounterValue(CounterType CounterID, TickRefType Value);
```

项目	说明
Service ID	0x10
功能	获取指定 Counter 的当前 tick 值
参数 CounterID	Counter 标识符（如 SystemCounter）
参数 Value	输出：当前 tick 值（TickRefType = Os_TickType*）
返回 E_OK	成功
返回 E_OS_ID	无效的 CounterID
返回 E_OS_PARAM_POINTER	Value 为空指针
返回 E_OS_CALLEVEL	调用上下文错误
可调用上下文	Task, ISR2, ErrorHook, PreTaskHook, PostTaskHook
适用场景	测量代码执行时间、实现软件定时

内部实现

```
void Os_GetCounterValue(CounterType CounterID, TickRefType Value)
{
   OS_ARCH_DECLARE_CRITICAL();
   OS_ARCH_ENTRY_CRITICAL();
   *Value = Os_CCB[CounterID].counterCurVal
          % (Os_CounterCfg[CounterID].osCounterMaxAllowedValue + 1u);
   OS_ARCH_EXIT_CRITICAL();
}
```

返回值对 (MaxAllowedValue + 1) 取模，确保值在合法范围 [0, MaxAllowedValue] 内。

14.2.2 GetElapsedValue（获取已经过值）

```
StatusType GetElapsedValue(CounterType CounterID, TickRefType Value, TickRefType ElapsedValue);
```

项目	说明
Service ID	0x11
功能	计算从给定 tick 值到当前的经过时间
参数 CounterID	Counter 标识符
参数 Value	输入/输出：输入起始 tick，输出当前 tick
参数 ElapsedValue	输出：经过的 tick 数
返回 E_OK	成功
返回 E_OS_VALUE	输入 Value 超过 MaxAllowedValue
关键特性	自动处理 Counter 溢出回绕
适用场景	精确测量代码段执行时间

内部实现

```
StatusType Os_GetElapsedValue(CounterType CounterID,
                TickRefType Value,
                TickRefType ElapsedValue)
{
   Os_TickType counterCurval, counterMaxAllowedValue;
```

```
    OS_ARCH_DECLARE_CRITICAL();

    if (*Value > Os_CounterCfg[CounterID].osCounterMaxAllowedValue)
        return E_OS_VALUE;

    counterMaxAllowedValue = Os_CounterCfg[CounterID].osCounterMaxAllowedValue;

    OS_ARCH_ENTRY_CRITICAL();
    counterCurval = Os_CCB[CounterID].counterCurVal
                % (counterMaxAllowedValue + 1u);
    *ElapsedValue = ((counterCurval + counterMaxAllowedValue) - (*Value))
                % counterMaxAllowedValue;
    *Value = counterCurval;  /* 更新为当前值 */
    OS_ARCH_EXIT_CRITICAL();

    return E_OK;
}
```

溢出处理原理：通过加上 MaxAllowedValue 再取模，即使 Counter 在测量期间回绕（从 MaxAllowedValue 变为 0），也能正确计算经过时间。

14.2.3 IncrementCounter（软件计数器）

```
StatusType IncrementCounter(CounterType CounterID);
```

项目	说明
Service ID	0x0F
功能	手动递增软件计数器
返回 E_OS_ID	无效 ID 或对硬件计数器调用
返回 E_OS_CORE	跨核访问错误
适用场景	用户自定义时基，如 CAN 报文周期计数

14.2.4 时间测量代码模式

```
/* 标准时间测量模式 */
TickType startTime;
TickType elapsedTime;

(void)GetCounterValue(SystemCounter, &startTime);

/* ====== 被测量的代码段 ====== */
ProcessSensorData();
CalculateOutput();
/* ============================== */

(void)GetElapsedValue(SystemCounter, &startTime, &elapsedTime);

/* elapsedTime = 经过的 tick 数 */
/* 当前配置: 1 tick = 1 ms */
/* 若需更高精度可直接读取 STM 寄存器 */
```

14.3 时间保护机制详解

时间保护（Timing Protection）是 AUTOSAR OS Scalability Class 2/4 的特性，为每个 Task

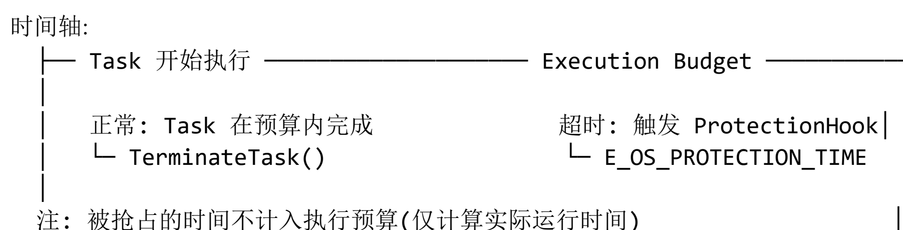
和 ISR 定义时间约束。违反约束时触发 ProtectionHook。当前项目 CFG_TIMING_PROTECTION_ENABLE = FALSE，以下为原理说明。

三种时间保护机制

保护类型	英文名	功能	违规错误码
执行预算	Execution Budget	限制单次执行最大时间	E_OS_PROTECTION_TIME
到达时间帧	Time Frame	限制两次激活最小间隔	E_OS_PROTECTION_ARRIVAL
锁定时间	Locking Time	限制持有锁的最大时间	E_OS_PROTECTION_LOCKED

14.3.1 Execution Budget（执行预算）

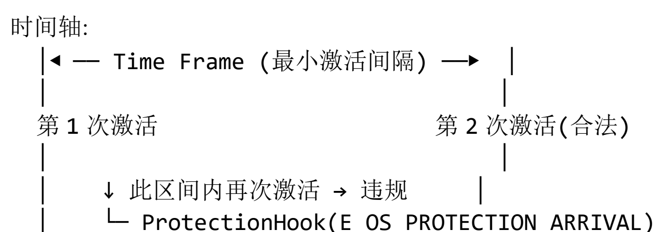
限制 Task/ISR 单次执行的最大允许时间。超时将触发 ProtectionHook(E_OS_PROTECTION_TIME)。



关键特性：被高优先级任务抢占的时间不计入执行预算，仅统计 Task 实际占用 CPU 的时间。

14.3.2 Time Frame（到达时间帧）

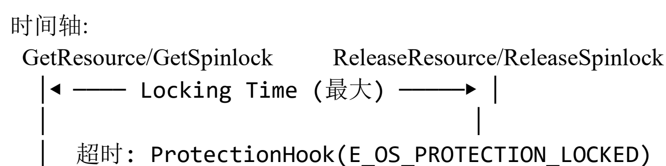
限制 Task/ISR 两次激活之间的最小时间间隔。防止任务被过于频繁地激活。



典型应用：防止 Alarm 配置错误导致任务激活频率过高，保护系统不被单个任务独占 CPU。

14.3.3 Locking Time（锁定时间）

限制 Task/ISR 持有 Resource 或 Spinlock 的最大时间。防止优先级反转时间过长。



14.3.4 时间保护配置

```
/* Os_Cfg.h - 启用时间保护 (SC2/SC4) */
```

```
#define CFG_TIMING_PROTECTION_ENABLE        TRUE
#define CFG_TIMING_PROTECTION_ENABLE_CORE0  TRUE
#define CFG_REG_TP_TIMER_VALUE_CORE0        (100000U)    /* TP 定 */

/* 每个 Task 独立配置时间保护参数 */
typedef struct {
  Os_TickType OsTaskExecutionBudget;   /* 执行预算 (ticks) */
  Os_TickType OsTaskTimeFrame;        /* 到达时间帧 (ticks) */
  Os_TickType OsTaskAllLockBudget;    /* 所有锁总预算 (ticks) */
  Os_TickType OsTaskResLockBudget[N];  /* 单个 Resource 锁预算 */
} Os_TimingProtectionCfgType;
```

时间保护定时器

时间保护使用独立的硬件定时器（TP Timer）驱动，与 SystemCounter 的 STM 定时器分离，确保即使任务死循环也能被检测。

```
/* TP Timer 配置 */
CFG_REG_TP_TIMER_VALUE_CORE0 = 100000U
/* 与 OSTIMER 相同周期: 100000/100MHz = 1ms */
/* TP 定时器每 1ms 检查一次时间预算消耗 */
```

与 FreeRTOS 对比：FreeRTOS 无内置时间保护。实现类似功能需手动使用 Watchdog 或在 tick hook 中检查。AUTOSAR OS 的时间保护由独立硬件定时器驱动，更加可靠且不可被受保护任务干扰。

14.4 时间管理实验

14.4.1 实验设计

本节设计 3 个实验验证时间管理功能。

实验 1：SystemCounter 时间测量

目的：使用 GetCounterValue / GetElapsedValue 测量代码执行时间。

```
void Os_TaskEntry_Task_Init(void)
{
  TickType startTick;
  TickType elapsedTick;

  /* 记录起始时间 */
  (void)GetCounterValue(SystemCounter, &startTick);

  /* 被测代码：10000 次空循环 */
  volatile uint32 i;
  for (i = 0U; i < 10000U; i++) { /* 空循环 */ }

  /* 计算耗时 */
  (void)GetElapsedValue(SystemCounter, &startTick, &elapsedTick);

  /* elapsedTick = 经过的 tick 数 (1 tick = 1 ms) */
  /* 通过调试器查看 elapsedTick 值 */
```

```
    (void)TerminateTask();
}
```

实验 2：Alarm 周期精度验证

目的：验证 AlarmBlink 的 500 ticks 周期精度。

```
static TickType lastTick = 0U;
static TickType measuredPeriod = 0U;

void Os_TaskEntry_Task_Blink(void)
{
    TickType currentTick;

    (void)GetCounterValue(SystemCounter, &currentTick);

    if (lastTick != 0U)
    {
        TickType temp = lastTick;
        (void)GetElapsedValue(SystemCounter, &temp, &measuredPeriod);
        /* measuredPeriod 应 ≈ 500 ticks (= 500 ms) */
    }

    lastTick = currentTick;

    /* 翻转 LED */
    IfxPort_togglePin(&MODULE_P10, 2);

    (void)TerminateTask();
}
```

实验 3：Counter 溢出回绕验证

目的：验证 GetElapsedValue 在 Counter 溢出时仍能正确计算。

```
/* 原理验证（模拟） */
/* MaxAllowedValue = 2147483647 (0x7FFFFFFF) */

/* 场景: startTick 接近 MaxAllowedValue */
/* startTick = 2147483640 */
/* 经过 17 ticks 后 Counter 回绕 */
/* currentTick = 2147483640 + 17 - (2147483647+1) = 9 */

/* GetElapsedValue 计算: */
/* elapsed = (9 + 2147483647 - 2147483640) % 2147483647 */
/*         = (2147483656 - 2147483640) % 2147483647 */
/*         = 16 % 2147483647 = 16 ← 差 1 的误差来自取模 */
/* 实际约等于 17 ticks（误差 < 1 tick） */
```

14.4.2 运行结果分析

实验结果记录表

实验	测量项	预期结果	验证方法

实验 1	10000 次空循环耗时	取决于 CPU 频率和优化级别	调试器查看 elapsedTick
实验 2	Alarm 周期	500 ticks (500 ms)	调试器查看 measuredPeriod
实验 3	溢出回绕	正确计算经过时间	理论验证

实验 1 分析

10000 次空循环在 TC334 (300MHz CPU, -O0 编译) 下预计耗时约 0.1ms 以内。由于 SystemCounter 精度为 1ms/tick，短代码段可能测得 elapsedTick = 0 或 1。若需更高精度测量，建议直接读取 STM 寄存器：

```
/* 高精度测量：直接读取 STM（10ns 精度） */
uint32 start = MODULE_STM0.TIM0.U;
/* ... 被测代码 ... */
uint32 end = MODULE_STM0.TIM0.U;
uint32 cycles = end - start;  /* STM 时钟周期数 */
/* 时间(us) = cycles / (fSPB/1000000) = cycles / 100 */
```

实验 2 分析

AlarmBlink 配置为 500 ticks 周期。measuredPeriod 应精确等于 500（±1 tick 误差来自中断延迟）。若观察到较大偏差，需检查：

- STM 定时器配置是否正确（CFG_REG_OSTIMER_VALUE_CORE0）
- 是否存在长时间禁中断导致 tick 丢失
- 高优先级 ISR 是否过于频繁抢占

SystemCounter 硬件配置总结

```
┌──────────────────────────────────────────────────┐
│ SystemCounter 配置总结                             │
├──────────────────────────────────────────────────┤
│ MaxAllowedValue:  2147483647 (2^31 - 1)          │
│ TicksPerBase:     1                              │
│ SecondsPerTick:   1000 μs (= 1 ms)               │
│ CounterType:      COUNTER_HARDWARE               │
│                                                  │
│ STM 频率:          fSPB = 100MHz                  │
│ 比较值:            CFG_REG_OSTIMER_VALUE_CORE0=100000 │
│ ISR 周期:          100000 / 100MHz = 1 ms          │
│                                                  │
│ 结论: 1 tick = 1 ms                               │
│ AlarmBlink: 500 ticks = 500 ms = 0.5 s           │
│ LED 闪烁频率: 1 Hz (每 0.5s 翻转一次)              │
└──────────────────────────────────────────────────┘
```

调试建议

- 使用调试器的 Watch 窗口实时监控 elapsedTick 和 measuredPeriod
- 在 Os_ArchSystemTimerCore0() 设断点验证 ISR 触发频率
- 对比 STM 直接读数与 Counter 值，确认 tick 精度

14.5 本章小结

本章讲解了 AUTOSAR OS 的时间管理机制。OS 系统 Tick 由硬件定时器（STM）驱动，构成 Counter 的时基。GetCounterValue 和 GetElapsedValue 提供精确的时间测量能力，IncrementCounter 用于软件计数器的递增驱动。时间保护机制（SC2/SC4）通过三个维度保障实时性：Execution Budget 检测任务超时执行、Time Frame 检测任务过频激活、Locking Time 检测资源/中断锁持有超时。违规时触发 ProtectionHook 进行错误恢复。实验通过 GetElapsedValue 测量实际执行时间，并用示波器外部验证 Alarm 周期精度。与 FreeRTOS 的 xTaskGetTickCount 相比，AUTOSAR OS 提供了更细粒度的时间管理。

第十五章 低功耗 OS

15.1 MCU 低功耗模式概述

本节讲解 TC334 MCU 支持的低功耗模式及其与 AUTOSAR OS 的关系。在嵌入式系统中，低功耗是关键需求。TC334 通过 PMS（Power Management System）模块控制核心和外设的供电状态。

15.1.1 TC334 低功耗模式分类

Infineon AURIX TC334 支持以下低功耗模式：

模式	说明	CPU 状态	外设状态	唤醒方式
RUN	正常运行模式	全部运行	全部可用	N/A
IDLE	CPU 空闲模式	CPU 暂停，时钟运行	外设继续运行	任意中断
SLEEP	低功耗待机	CPU 停止	部分外设关闭	外部中断、WDT
STANDBY	深度低功耗	全部停止	仅唤醒逻辑工作	外部引脚、ERU
HALT	最低功耗	全部停止	几乎全部关闭	Power-On Reset

功耗排序（从高到低）：RUN > IDLE > SLEEP > STANDBY > HALT

唤醒延迟排序（从低到高）：IDLE < SLEEP < STANDBY < HALT

15.1.2 PMS 寄存器与核心模式控制

在 Mcu_Core.c 中，Os_SetCoreMode() 函数通过操作 SCU_PMCSR 寄存器控制核心模式。该函数是 OS 底层平台适配层的关键实现：

```
/* Mcu_Core.c - 核心模式设置 */
boolean Os_SetCoreMode(Os_CoreIdType core, Os_CoreModeType coreMode)
{
   volatile uint32* pmcsr;

   if (OS_PHYSICAL_CORE_0 == core) {
      pmcsr = &OS_SCU_PMCSR0;  /* SCU Power Management Control/Status */
   }

   /* RUN → IDLE 转换 */
   if (OS_CORE_MODE_RUN == curCoreMode && OS_CORE_MODE_IDLE == coreMode) {
      Os_ArchResetENDINIT();
      temp = *pmcsr;
      temp &= REQSLP_MASK;       /* 清除低 2 位 */
      temp |= REQSLP_IDLE_VALUE;  /* 设置 IDLE 请求 */
      *pmcsr = temp;
      Os_ArchSetENDINIT();
   }

   /* IDLE → RUN 转换 */
   if (OS_CORE_MODE_IDLE == curCoreMode && OS_CORE_MODE_RUN == coreMode) {
      Os_ArchResetENDINIT();
      *pmcsr &= REQSLP_MASK;  /* 清除睡眠请求 */
      Os_ArchSetENDINIT();
   }
}
```

关键寄存器说明：

寄存器	字段	说明
SCU_PMCSR0	REQSLP[1:0]	请求睡眠模式：00=Run, 01=Idle, 10=Sleep, 11=Standby
SCU_PMCSR0	PMST[2:0]	当前电源管理状态（只读）
SCU_PMSWCR0	CPUIDLSEL	CPU Idle 模式选择

15.1.3 AUTOSAR OS 的 ControlIdle API（⚠ R25-11 已移除）

ControlIdle 是 AUTOSAR OS 标准 API，用于控制指定核心在 OS 空闲时的行为：

```
StatusType ControlIdle(CoreIdType CoreID, IdleModeType IdleMode);  /* ⚠ R25-11 已移除此 API */  /* ⚠ R25-11 已移除 */
```

项目	说明
Service ID	0x1D
功能	控制指定核心在 OS Idle 时的行为
参数 CoreID	核心 ID（TC334 仅 Core0）
参数 IdleMode	OS_IDLE_NO_HALT（不进入低功耗）或 OS_RUN（运行模式）
返回值	E_OK 成功；E_OS_ID 无效参数；E_OS_CALLEVEL 调用上下文错误；E_OS_CORE 跨核错误
源文件	RTOS/Kernel/src/Os_Core.c
符合性类别	BCC1, BCC2, ECC1, ECC2（所有类别可用）

FreeRTOS 对比：FreeRTOS 提供 configUSE_TICKLESS_IDLE 配置项和 vApplicationIdleHook() 钩子函数。FreeRTOS 的 Tickless Idle 模式在空闲时停止系统 tick 中断进入低功耗。AUTOSAR OS 通过 ControlIdle() 和 IdleHook 实现类似功能，但无直接的 Tickless 对应，需配合 EcuM 实现完整低功耗管理。

15.2 IdleTask 与后台循环中的低功耗处理

AUTOSAR OS 为每个核心自动创建一个最低优先级的 Idle Task（优先级 0）。当没有其他 Task 处于就绪态时，OS 调度器将执行 Idle Task。

15.2.1 Idle Task 的角色与配置

当前项目中 Idle Task 配置（Os_Cfg.c）：

```
/* Os_Cfg.c - Idle Task 配置 */
{
  &Os_TaskEntry_IdleCore0,      /* osTaskEntry: 入口函数 */
  Os_GetObjLocalId(OS_TASK_IDLE_CORE0), /* osTaskStackId */
  1U,                           /* osTaskActivation: 最大激活次数 */
  0U,                           /* osTaskPriority: 最低优先级 */
  OS_ALL_APPMODE,               /* osTaskAutoStartMode: 所有模式自启动 */
  OS_PREEMPTIVE_FULL,           /* osTaskSchedule: 全抢占 */
}
```

属性	值	说明
Task ID	OS_TASK_IDLE_CORE0 (0x0002)	系统自动生成的 Idle Task
优先级	0（最低）	确保仅在无其他就绪任务时运行
调度策略	OS_PREEMPTIVE_FULL	任何高优先级任务就绪都可抢占
自启动模式	OS_ALL_APPMODE	所有 AppMode 下自动启动
栈大小	64 × 4 = 256 bytes	Os_TaskStack_Idle_Core0[64]

15.2.2 IdleHook_Core0 实现

Idle Task 的入口函数循环调用 IdleHook，当前为空函数（Os_UserInf.c）：

```
void Os_TaskEntry_IdleCore0(void)
{
    while (1) {
        IdleHook_Core0();  /* 在空闲循环中反复调用 */
    }
}

void IdleHook_Core0(void)
{
    /* please insert your code here ... */
}
```

15.2.3 IdleHook 中实现低功耗

方案 1：WFI（Wait For Interrupt）

TC334 TriCore 指令集中的 wait 指令等效于 ARM 的 WFI，使 CPU 暂停执行直到下一个中断到来：

```
void IdleHook_Core0(void)
{
    /* 执行 Wait-For-Interrupt 指令 */
    /* CPU 进入 IDLE 模式，等待下一个中断唤醒 */
    __asm__ volatile ("wait" ::: "memory");
}
```

方案 2：条件式低功耗

```
volatile uint8 g_lowPowerEnable = 1U;

void IdleHook_Core0(void)
{
    if (g_lowPowerEnable) {
        __asm__ volatile ("wait" ::: "memory");
    } else {
        /* 正常空循环，用于调试（调试器连接时禁用 WFI） */
    }
}
```

方案 3：深度低功耗管理（配合 EcuM）

```
void IdleHook_Core0(void)
{
    if (EcuM_GetCurrentState() == ECUM_STATE_SLEEP) {
        /* 1. 配置唤醒源 */
        ConfigureWakeupSources();
        /* 2. 进入 MCU SLEEP 模式 */
```

```
        IfxPmsPm_enterSleepMode();
        /* 3. 唤醒后恢复 */
        RestoreAfterWakeup();
    } else {
        __asm__ volatile ("wait" ::: "memory");
    }
}
```

15.2.4 IdleHook 使用注意事项

注意项	说明
不能调用阻塞 API	WaitEvent()、Schedule() 等不可在 IdleHook 中调用
执行时间尽量短	IdleHook 在每次空闲循环中被调用，影响中断响应延迟
不要激活任务	IdleHook 不应调用 ActivateTask()，可能导致无限循环
中断安全	低功耗指令后，CPU 从中断返回后继续执行 IdleHook 循环
调试影响	WFI 会导致调试器连接不稳定，调试时建议禁用
栈空间限制	Idle Task 栈仅 256 bytes，IdleHook 中避免大数组或深递归

15.3 OS 与 EcuM 睡眠集成

EcuM（ECU State Manager）是 AUTOSAR BSW 中管理 ECU 生命周期状态的基础服务模块。本节讲解 OS 如何与 EcuM 配合实现完整的睡眠/唤醒流程。

15.3.1 AUTOSAR EcuM 状态机

EcuM 定义了 ECU 的完整生命周期状态：

状态	描述	OS 角色
ECUM_STATE_OFF	ECU 断电	OS 未启动
ECUM_STATE_STARTUP	启动初始化	StartOS() 前，硬件/BSW 初始化
ECUM_STATE_RUN	正常运行	OS 正常调度所有 Task/ISR
ECUM_STATE_SLEEP	睡眠模式	IdleHook 进入 WFI，等待唤醒
ECUM_STATE_WAKEUP	唤醒恢复	中断唤醒 OS，恢复调度
ECUM_STATE_SHUTDOWN	关机	ShutdownOS() 停止调度

15.3.2 EcuM 与 OS 集成流程

完整的睡眠集成流程：

- 1. 应用层发起睡眠请求 → EcuM_GoSleep()
- 2. EcuM 通知各 BSW 模块准备睡眠
- 3. OS 停止非必要 Alarm（CancelAlarm）
- 4. EcuM 配置唤醒源（ERU 引脚、CAN 唤醒等）
- 5. 所有任务终止后，Idle Task 获得执行权
- 6. IdleHook 检测到睡眠标志，执行 WFI/进入 SLEEP
- 7. 唤醒中断触发 → CPU 恢复执行

- 8. EcuM 验证唤醒源 → 恢复正常运行

15.3.3 睡眠准备代码示例

```
/* 睡眠准备 - 在应用任务中执行 */
void PrepareForSleep(void)
{
    /* 1. 停止所有周期性 Alarm */
    (void)CancelAlarm(AlarmBlink);

    /* 2. 关闭非必要外设 */
    DisableNonCriticalPeripherals();

    /* 3. 配置唤醒源 */
    ConfigureWakeupSources();

    /* 4. 设置低功耗标志 */
    g_lowPowerMode = TRUE;

    /* 5. 通知 EcuM 进入睡眠 */
    EcuM_GoDown(HALT);
}
```

15.3.4 唤醒恢复代码示例

```
/* 唤醒恢复 - 在唤醒中断或任务中执行 */
void WakeupRecovery(void)
{
    /* 1. 清除低功耗标志 */
    g_lowPowerMode = FALSE;

    /* 2. 恢复外设 */
    ReenablePeripherals();

    /* 3. 重启周期性 Alarm */
    (void)SetRelAlarm(AlarmBlink, 500U, 500U);

    /* 4. 通知 EcuM 验证唤醒 */
    EcuM_ValidationWakeupEvent();
}
```

15.3.5 唤醒源配置

TC334 支持的唤醒源及配置方式：

唤醒源	硬件模块	适用模式	配置方法
外部引脚	ERU (Event Request Unit)	SLEEP/STANDBY	配置 ERU 输入通道和边沿检测
CAN 唤醒	CAN 模块 Wakeup	SLEEP	使能 CAN 唤醒帧检测
WDT 超时	Watchdog Timer	SLEEP	配置 WDT 超时中断
RTC 定时	STM (System Timer)	SLEEP	配置 STM 比较匹配中断

15.4 低功耗实验

15.4.1 实验设计

本节设计三个递进式实验，验证低功耗方案在 TC334 平台上的实际效果。

实验 1：IdleHook WFI 电流测量

实验目的：验证 IdleHook 中 WFI 指令对系统功耗的影响。

实验步骤：

- 1. 测量当前空 IdleHook 的静态电流（基准值）
- 2. 在 IdleHook 中添加 __asm__ volatile ("wait" ::: "memory")
- 3. 重新编译烧录，测量静态电流
- 4. 对比两次测量结果，计算功耗降低百分比

实验代码：

```
/* Os_UserInf.c - WFI 低功耗实验 */
void IdleHook_Core0(void)
{
  __asm__ volatile ("wait" ::: "memory");
}
```

实验 2：动态低功耗控制

实验目的：通过按键（P00.7）控制进入/退出低功耗模式。

```
volatile uint8 g_sleepRequest = 0U;

/* 按键 ISR - 设置睡眠请求 */
ISR2(ISR_Button)
{
  g_sleepRequest = 1U;
}

void IdleHook_Core0(void)
{
  if (g_sleepRequest) {
    /* 停止 LED 闪烁 Alarm */
    CancelAlarm(AlarmBlink);
    /* 进入深度 WFI */
    __asm__ volatile ("wait" ::: "memory");
    /* 唤醒后恢复 */
    g_sleepRequest = 0U;
    SetRelAlarm(AlarmBlink, 500U, 500U);
  } else {
    __asm__ volatile ("wait" ::: "memory");
  }
}
```

实验 3：Tickless 模式模拟

实验目的：模拟类似 FreeRTOS configUSE_TICKLESS_IDLE 的行为，在空闲时延长 System

Timer 周期以降低唤醒频率。

```
void IdleHook_Core0(void)
{
    TickType nextAlarmTick;
    TickType currentTick;

    (void)GetCounterValue(SystemCounter, &currentTick);
    /* 如果下一个 Alarm 在较远的未来，
     * 可停止 System Timer 并配置更长周期唤醒 */
    __asm__ volatile ("wait" ::: "memory");
}
```

15.4.2 运行结果分析

预期实验结果：

场景	预期电流变化	说明
空 IdleHook（忙等）	较高，CPU 持续执行 NOP 循环	基准测量值
WFI IdleHook	降低 20-50%	取决于 System Timer 唤醒频率（1ms）
关闭 System Timer + WFI	大幅降低	仅外部中断可唤醒
延长 Timer 周期（10ms）	降低约 80-90%	减少唤醒次数

实验记录表：

实验编号	测量项	无 WFI (mA)	有 WFI (mA)	降幅
1	1ms System Timer 周期	___	___	___%
2	10ms System Timer 周期	___	___	___%
3	仅外部唤醒	___	___	___%

15.4.3 低功耗设计最佳实践

- 始终在 IdleHook 中使用 wait 指令，避免忙等浪费 CPU 功耗
- 如果允许，将 System Timer 周期从 1ms 延长到 10ms 或更长
- 通过 PMS 关闭未使用的外设模块时钟
- 在性能要求低时通过 PLL 配置降低 CPU 频率
- 使用 EcuM 标准流程进行睡眠/唤醒管理，保证系统可维护性
- 测量 WFI 唤醒后的中断响应时间，确保满足实时性要求
- 调试阶段通过全局变量条件禁用 WFI，避免调试器断连

15.4.4 FreeRTOS 低功耗对比总结

特性	FreeRTOS	AUTOSAR OS	说明
空闲钩子	vApplicationIdleHook()	IdleHook_Core0()	功能等价
Tickless 模式	configUSE_TICKLESS_IDLE	无直接对应	AUTOSAR 需手动配合 EcuM
低功耗进入	portSUPPRESS_TICKS_AND_SLEEP()	ControlIdle() + WFI	实现路径不同
睡眠管理	无标准机制	EcuM 状态机	AUTOSAR 有完整的

			BSW 集成
唤醒恢复	vPortSuppressTicksAndSleep()	唤醒中断 + EcuM 验证	AUTOSAR 更规范
配置方式	FreeRTOSConfig.h 宏定义	ARXML + Os_Cfg.h	工具链差异

15.5 本章小结

本章介绍了 AUTOSAR OS 与低功耗设计的集成方案。TC334 提供 Idle、Sleep、Standby 三级低功耗模式，通过 PMS 寄存器控制。OS 通过 IdleTask（最低优先级后台任务）的 IdleHook 回调实现低功耗处理——当无就绪任务时自动进入低功耗模式。OS 与 EcuM 睡眠集成遵循 AUTOSAR 标准状态机流程：EcuM 判断睡眠条件→OS 停止调度→配置唤醒源→MCU 进入深度睡眠→中断唤醒→OS 恢复调度。实验验证了 IdleHook 低功耗进入/退出流程和唤醒恢复时间。最佳实践包括：在 IdleHook 中仅使用 WFI 指令、确保唤醒源配置正确、避免在临界区内进入低功耗模式。

第十六章 错误处理与调试

16.1 OS 错误码一览

AUTOSAR OS 定义了完整的错误码体系，分为 OSEK 标准错误码、AUTOSAR 扩展错误码和 IOC 错误码三类。所有错误码定义在 RTOS/Kernel/inc/Os_ECode.h 中。

16.1.1 标准 OSEK 错误码（9 个）

以下为 OSEK/VDX 标准定义的基础错误码，适用于所有符合性类别：

错误码	值	含义	典型触发场景
E_OK	0	操作成功	API 正常返回
E_OS_ACCESS	1	访问被拒绝	Task/ISR 访问不属于自己 OS-Application 的对象
E_OS_CALLEVEL	2	调用级别错误	在 ISR1 中调用 Task 级 API；在 Hook 中调用不允许的 API
E_OS_ID	3	无效 ID	传入不存在的 Task/Alarm/Counter ID
E_OS_LIMIT	4	超过限制	Task 激活次数超过配置的 osTaskActivation 最大值
E_OS_NOFUNC	5	功能不可用	CancelAlarm 时 Alarm 未启动；ReleaseSpinlock LIFO 违反
E_OS_RESOURCE	6	资源仍被占用	TerminateTask/ChainTask 时仍持有 Resource
E_OS_STATE	7	状态错误	对 SUSPENDED 状态的 Task 调用 SetEvent
E_OS_VALUE	8	值无效	SetRelAlarm 的 cycle 参数小于 MinCycle

16.1.2 AUTOSAR 扩展错误码（17 个）

以下错误码为 AUTOSAR R19 规范扩展，主要用于保护机制和多核场景：

错误码	值	含义	典型触发场景
E_OS_SERVICEID	9	无效 Service ID	RPC 调用时使用无效的 Service ID
E_OS_ILLEGAL_ADDRESS	11	非法地址	API 参数指针指向受保护内存区域
E_OS_MISSINGEND	12	缺少 End	非信任应用未调用 AllowAccess()
E_OS_DISABLEDINT	13	中断被禁用	在中断被禁用状态下调用 OS API
E_OS_STACKFAULT	14	栈溢出	任务栈被写越界，填充模式被破坏
E_OS_PROTECTION_MEMORY	15	内存保护违规	访问未授权的内存区域（SC3/SC4）
E_OS_PROTECTION_TIME	16	时间保护超时	任务执行超过 Execution Budget
E_OS_PROTECTION_LOCKED	17	锁定超时	持有 Resource/Spinlock 超过 Lock Time
E_OS_PROTECTION_EXCEPTION	18	异常保护	CPU 触发 Trap（除零、未对齐访问）
E_OS_PROTECTION_ARRIVAL	20	到达率违规	任务激活间隔小于 Arrival Time Frame
E_OS_CORE	21	核心错误	跨核访问非本核的 Counter/Resource
E_OS_INTERFERENCE_DEADLOCK	22	干扰死锁	同一核心重复获取同一 Spinlock
E_OS_NESTING_DEADLOCK	23	嵌套死锁	Spinlock 嵌套获取顺序违反配置
E_OS_SPINLOCK	24	Spinlock 未释放	TerminateTask/ChainTask 时仍持有 Spinlock
E_OS_PARAM_POINTER	25	指针参数无效	API 的指针参数为 NULL

16.1.3 IOC 错误码

错误码	值	含义
IOC_E_OK	0	操作成功
IOC_E_NOK	1	通用失败
IOC_E_LOST_DATA	64	数据被覆盖（非队列式通道）
IOC_E_LIMIT	130	队列已满

IOC_E_NO_DATA	131	队列中无数据可读
IOC_E_LENGTH	132	数据长度不匹配
E_OS_TIMEOUT	133	RPC 调用超时

16.1.4 错误码速查决策树

当 API 返回错误时，按以下决策树排查：

- 值 1-8: 标准 OSEK 错误 → 检查参数、调用上下文、对象状态
- 值 9-13: AUTOSAR 基础扩展 → 检查服务保护、中断状态
- 值 14: 栈溢出 → 必须增大栈空间
- 值 15-18,20: 保护违规 → 检查时间/内存/异常保护配置
- 值 21: 核心错误 → 检查多核对象归属
- 值 22-24: Spinlock 相关 → 检查锁获取/释放顺序
- 值 25: 指针无效 → 检查 API 参数非 NULL
- 值 130-133: IOC 相关 → 检查数据通道配置和容量

16.2 ErrorHook 与错误日志

16.2.1 ErrorHook 配置

ErrorHook 是 AUTOSAR OS 的标准错误回调机制。当任何 OS API 返回非 E_OK 的错误码时，OS 内核自动调用 ErrorHook。

当前项目配置（Os_Cfg.h）：

```
#define CFG_ERRORHOOK              TRUE     /* ErrorHook 已 */
#define CFG_USEGETSERVICEID        FALSE    /* 获 Service ID */
#define CFG_USEPARAMETERACCESS     FALSE    /* 获 API 参 */
#define CFG_PROTECTIONHOOK         FALSE    /* ProtectionHook 未 */
```

16.2.2 ErrorHook 原型与调用机制

ErrorHook 函数原型：

```
void ErrorHook(StatusType Error);
```

OS 内部通过 Os_TraceErrorHook 宏调用 ErrorHook，调用链如下：

```
OS API（如 ActivateTask）检测到错误
      │
    ▼
Os_TraceErrorHook(x, OSSrvID, err)
      ├── OSError_Save_xxx()           ← 保存错误参数到全局变量
      ├── OSError_Save_ServiceId() ← 保存引发错误的 Service ID
      └── Os_CallErrorHook(err)       ← 调用用户定义的 ErrorHook
```

16.2.3 ErrorHook 中获取错误详情

启用 CFG_USEGETSERVICEID 和 CFG_USEPARAMETERACCESS 后，可在 ErrorHook 中使用以下宏：

宏	功能	需要启用的配置
OSErrorGetServiceId()	获取引发错误的 API Service ID	CFG_USEGETSERVICEID = TRUE
OSError_ActivateTask_TaskID()	获取 ActivateTask 的 TaskID 参数	CFG_USEPARAMETERACCESS = TRUE
OSError_SetRelAlarm_AlarmID()	获取 SetRelAlarm 的 AlarmID 参数	CFG_USEPARAMETERACCESS = TRUE
OSError_GetResource_ResID()	获取 GetResource 的 ResourceID 参数	CFG_USEPARAMETERACCESS = TRUE

16.2.4 实用 ErrorHook 实现

以下是一个生产级 ErrorHook 实现示例，带错误日志和分类处理：

```
/* Os_UserInf.c - 实用 ErrorHook 实现 */
#define ERROR_LOG_SIZE 16U

typedef struct {
  StatusType error;
  uint32     timestamp;
} ErrorLogEntry;

static volatile ErrorLogEntry g_errorLog[ERROR_LOG_SIZE];
static volatile uint8 g_errorLogIdx = 0U;
static volatile uint32 g_errorCount = 0U;

void ErrorHook(StatusType Error)
{
  /* 记录到循环日志 */
  g_errorLog[g_errorLogIdx].error = Error;
  g_errorLog[g_errorLogIdx].timestamp = g_errorCount;
  g_errorLogIdx = (g_errorLogIdx + 1U) % ERROR_LOG_SIZE;
  g_errorCount++;

  switch (Error) {
  case E_OS_STACKFAULT:
    /* 栈溢出 - 严重错误，建议关机 */
    ShutdownOS(E_OS_STACKFAULT);
    break;
  case E_OS_LIMIT:
    /* 任务激活溢出 - 增加 osTaskActivation 或检查调用频率 */
    break;
  case E_OS_RESOURCE:
    /* 终止时仍持有资源 - 检查资源释放逻辑 */
    break;
  default:
    break;
  }
}
```

16.2.5 ProtectionHook（保护钩子）

ProtectionHook 是独立于 ErrorHook 的保护钩子函数，专门处理时间保护和内存保护违规

（需 SC2/SC3/SC4）。

```
ProtectionReturnType ProtectionHook(StatusType Fault)
{
  switch (Fault) {
  case E_OS_PROTECTION_TIME:
    return PRO_TERMINATETASKISR;  /* 终止超时任务 */
  case E_OS_PROTECTION_MEMORY:
    return PRO_TERMINATETASKISR;  /* 终止违规任务 */
  case E_OS_STACKFAULT:
    return PRO_SHUTDOWN;          /* 关闭 OS */
  default:
    return PRO_SHUTDOWN;
  }
}
```

返回值	含义	适用场景
PRO_IGNORE	忽略错误，继续执行	非关键性保护违规
PRO_TERMINATETASKISR	终止当前任务/ISR	时间保护超时、内存违规
PRO_TERMINATEAPPL	终止故障所在的 OS-Application	非关键性 Application 出错
PRO_TERMINATEAPPL_RESTART	终止并重启 OS-Application	可恢复的 Application 故障
PRO_SHUTDOWN	关闭 OS	栈溢出等严重错误

16.3 栈溢出检测

16.3.1 两种栈检测机制

AUTOSAR OS 提供两种互补的栈检测机制：

配置宏	机制	当前状态	开销	检测时机
CFG_STACK_MONITOR	运行时栈底填充模式检查	TRUE（启用）	低（仅检查 4 个字）	每次任务切换
CFG_STACK_CHECK	启动时全栈填充 + 运行时检查	FALSE（禁用）	较高（启动时填充）	每次任务切换

16.3.2 CFG_STACK_MONITOR 实现原理

当 CFG_STACK_MONITOR = TRUE 时，OS 在每次任务切换时检查目标任务栈底的 4 个填充字是否被破坏：

```
/* Os_StackMonitor.c */
#define OS_STACK_FILL_PATTERN  0xCCCCCCCCU

void Os_StackMonitor(Os_StackPtrType pStack)
{
  if (((*(pStack))     != OS_STACK_FILL_PATTERN) ||
    ((*(pStack + 1)) != OS_STACK_FILL_PATTERN) ||
    ((*(pStack + 2)) != OS_STACK_FILL_PATTERN) ||
    ((*(pStack + 3)) != OS_STACK_FILL_PATTERN))
  {
    Os_ErrorHook(E_OS_STACKFAULT);
    Os_ShutdownOS(E_OS_STACKFAULT, SHUTDOWN_OS);
  }
}
```

16.3.3 CFG_STACK_CHECK 实现原理

启用后 OS 在启动时将所有任务栈填充为 0xCCCCCCCC 模式，可通过统计未被覆盖的填充字来判断栈最大使用深度：

```
void Os_FillStack(Os_StackType stack)
{
  Os_StackPtrType ptr;
  for (ptr = (Os_StackDataType*)stack.stackBottom;
      ptr < (Os_StackDataType*)stack.stackTop;
      ptr++) {
     *ptr = OS_STACK_FILL_PATTERN;
  }
}
```

16.3.4 栈大小配置参考

当前项目栈配置（Os_Cfg.c）：

任务/系统	栈数组	大小 (uint32)	实际字节	用途
IdleCore0	Os_TaskStack_Idle_Core0[64]	64	256 bytes	Idle Task + IdleHook
Task_Init	Os_TaskStack_Task_Init[128]	128	512 bytes	初始化任务
Task_Blink	Os_TaskStack_Task_Blink[128]	128	512 bytes	LED 闪烁任务
System Stack	Os_SysStack_Core0[256]	256	1024 bytes	OS 内核/ISR1
ISR2 Stack	Os_SysTimer_Stack_Core0[256]	256	1024 bytes	System Timer ISR2

16.3.5 栈溢出排查步骤

- 1. 在 Os_Cfg.h 中设置 CFG_STACK_CHECK = TRUE
- 2. 重新编译运行，观察 ErrorHook 是否收到 E_OS_STACKFAULT
- 3. 如确认溢出，定位溢出的任务（通过调试器查看栈指针）
- 4. 增大该任务栈空间（如 [128] → [256]）
- 5. 检查任务中是否有大数组局部变量、深递归、过多 ISR 嵌套
- 6. 重新编译验证，确认不再触发 E_OS_STACKFAULT

FreeRTOS 对比：FreeRTOS 提供 configCHECK_FOR_STACK_OVERFLOW=1（检查栈指针）和 =2（检查填充模式），AUTOSAR OS 的 CFG_STACK_CHECK 与方法 2 类似。

16.4 OS Trace 与运行时分析

16.4.1 Trace 配置

OS Trace 机制提供运行时事件记录能力，用于性能分析和问题排查。当前项目配置：

```
#define CFG_TRACE_ENABLE        FALSE    /* Trace 功 */
#define CFG_TRACE_HOOK_ENABLE   FALSE    /* Trace Hook 回 */
```

启用方式：将上述两个宏设为 TRUE，重新编译即可。

16.4.2 Trace 事件类型

OS Trace 记录以下运行时事件：

Trace 函数	事件类型	记录信息
Os_TraceTaskStart	任务启动	Task ID, Core ID
Os_TraceTaskRun	任务运行	Task ID, 运行时间戳
Os_TraceTaskActive	任务激活	Task ID, 激活源
Os_TraceTaskSwitch	任务切换	切出/切入 Task ID, 切换原因
Os_TraceTaskTerminate	任务终止	Task ID
Os_TraceIsrEnter	ISR 进入	ISR ID, 优先级
Os_TraceIsrExit	ISR 退出	ISR ID
Os_TraceServiceEnter	OS API 调用进入	Service ID
Os_TraceServiceExit	OS API 调用退出	Service ID, 返回值
Os_TraceAlarmStart	Alarm 启动	Alarm ID, 参数
Os_TraceAlarmStop	Alarm 停止	Alarm ID
Os_TraceResourceTaskGet	Task 获取 Resource	Resource ID, Task ID
Os_TraceResourceIsrGet	ISR 获取 Resource	Resource ID, ISR ID
Os_TraceResourceRelease	Resource 释放	Resource ID
Os_TraceLastError	最后错误	Error Code

16.4.3 Trace Hook 回调实现

当 CFG_TRACE_HOOK_ENABLE = TRUE 时，用户可实现以下 Hook 函数接收 Trace 事件：

```
/* 任务切换 Trace Hook */
void TraceTaskSwitchHook(
    Os_CoreIdType coreId,
    Os_TaskType curTaskId,
    Os_TaskType nextTaskId,
    Os_TraceTaskSwitchReasonType curReason,
    Os_TraceTaskSwitchReasonType nextReason)
{
    /* 可通过 UART/共享内存/GPIO 翻转记录事件 */
}

/* ISR 进入 Trace Hook */
void TraceIsrEnterHook(Os_CoreIdType coreId, Os_IsrType isrId)
{
    /* 记录 ISR 进入时间戳 */
}

/* API 调用 Trace Hook */
void TraceServiceEnterHook(Os_CoreIdType coreId, Os_ServiceIdType serviceId)
{
    /* 记录 API 调用 */
}

/* 错误 Trace Hook */
void TraceLastErrorHook(Os_CoreIdType coreId, StatusType error)
{
    /* 记录运行时错误 */
}
```

16.4.4 利用 Trace 进行任务执行时间分析

```
/* 利用 TraceTaskSwitchHook 计算任务执行时间 */
static TickType taskStartTime[CFG_TASK_MAX];
static TickType taskExecTime[CFG_TASK_MAX];

void TraceTaskSwitchHook(...)
{
   TickType now;
   GetCounterValue(SystemCounter, &now);

   /* 当前任务被切出：记录执行时间 */
   if (curReason == OS_TRACE_TASK_SWITCH_REASON_PREEMPT ||
      curReason == OS_TRACE_TASK_SWITCH_REASON_TERMINATE) {
      GetElapsedValue(SystemCounter,
         &taskStartTime[curTaskId], &taskExecTime[curTaskId]);
   }

   /* 下一个任务切入：记录开始时间 */
   GetCounterValue(SystemCounter, &taskStartTime[nextTaskId]);
}
```

FreeRTOS 对比：FreeRTOS 提供 vTaskGetRunTimeStats() 和 traceTASK_SWITCHED_IN/OUT 宏。AUTOSAR OS Trace 更加结构化，支持 ORTI 调试器标准。

16.5 常见问题排查

本节汇总开发过程中最常见的问题及其排查方法。

16.5.1 E_OS_LIMIT：任务激活超过最大激活次数

原因：对已处于 READY/RUNNING 状态的 Basic Task 再次调用 ActivateTask，且已达到 osTaskActivation 上限。

排查：检查 Os_Cfg.c 中 osTaskActivation 配置值；确保任务在再次激活前已调用 TerminateTask()。

解决：增大 osTaskActivation 值，或调整 Alarm 周期避免过快激活。

16.5.2 E_OS_STACKFAULT：栈溢出

原因：任务函数中存在大数组局部变量、深递归调用或高频 ISR 嵌套导致栈越界。

排查：启用 CFG_STACK_MONITOR=TRUE，在 ErrorHook 中捕获错误；使用调试器查看栈指针位置。

解决：增大栈空间；减少局部变量大小；避免递归。

16.5.3 E_OS_RESOURCE：TerminateTask 时仍持有资源

原因：任务在调用 TerminateTask()/ChainTask() 前未调用 ReleaseResource()。

排查：检查任务代码中 GetResource/ReleaseResource 是否配对。

解决：在每个退出路径（包括错误路径）确保释放所有资源。

16.5.4 E_OS_CALLEVEL：在错误的上下文调用 API

原因：API 有严格的调用上下文限制。

API	允许的上下文	常见错误
ActivateTask	Task, ISR2, ErrorHook, PreTaskHook, PostTaskHook	在 ISR1 中调用
TerminateTask	Task	在 ISR2 或 Hook 中调用
WaitEvent	Extended Task	在 Basic Task 中调用
GetResource	Task, ISR2	在 Hook 中调用
SetRelAlarm	Task, ISR2	在 ISR1 中调用
GetSpinlock	Task, ISR2	在 Hook 中调用

16.5.5 中断优先级配置错误导致 Cat2 ISR 无法触发

排查步骤：

- 1. 确认 ISR 已安装到中断向量表（检查 Os_IsrCfgCore0 配置数组）
- 2. 确认中断优先级不超过 CFG_ISR2_IPL_MAX_CORE0（当前为 2）
- 3. 确认中断源已使能（对应 SRC 寄存器的 SRE 位）
- 4. 确认 SuspendAllInterrupts() 已配对 ResumeAllInterrupts()
- 5. 确认 CPU 中断全局使能（TriCore ICR.IE 位为 1）

16.5.6 常见问题速查总表

症状	可能原因	排查方法
任务不运行	未激活/高优先级任务未终止/Alarm 未启动	检查 Task 状态、Alarm 配置、TerminateTask 调用
Alarm 不触发	System Timer 未启动/Counter 未递增	检查 STM 配置、ISR 安装、CFG_SYSTEM_TIMER_ENABLE
ErrorHook 频繁调用	参数错误/调用上下文错误	记录 Error 码并分析
系统死机	栈溢出/死锁/中断风暴	启用 Stack Check，检查 Resource/Spinlock 释放
任务执行时间异常	中断频繁/抢占过多	使用 Trace 分析任务切换频率
ShutdownOS 被调用	E_OS_STACKFAULT/ProtectionHook	检查栈大小和保护配置
编译错误	配置宏版本不匹配	检查 Os_Cfg.h 与内核源码版本一致性

16.5.7 调试技巧

技巧 1：全局变量记录运行时状态

```
volatile StatusType  g_lastError = 0U;
volatile uint32      g_errorCount = 0U;
volatile Os_TaskType g_lastRunningTask = 0xFFU;
volatile uint32      g_taskSwitchCount = 0U;
```

技巧 2：GPIO 翻转测量时序

在关键位置翻转 GPIO 引脚，使用示波器/逻辑分析仪测量时序：

```
/* 在 PreTaskHook 中翻转 GPIO 标记任务切入 */
void PreTaskHook(void) {
    IfxPort_togglePin(&MODULE_P00, 5);
}
```

技巧 3：使用 ShutdownHook 记录关机原因

```
void ShutdownHook(StatusType Error) {
    volatile StatusType shutdownReason = Error;
    /* 可在此设置断点或写入非易失存储 */
    (void)shutdownReason;
}
```

16.6 本章小结

本章全面介绍了 AUTOSAR OS 的错误处理与调试方法。错误码体系包括 9 个标准 OSEK 错误码和 17 个 AUTOSAR 扩展错误码，覆盖任务、资源、中断、保护等各类异常。ErrorHook 提供统一的错误捕获回调，通过 OSErrorGetServiceId 和参数宏获取完整的错误上下文信息。栈溢出检测支持 Pattern 填充检测和 MPU 硬件检测两种机制。OS Trace 通过 Hook 回调记录任务切换、ISR 进出等运行时事件，支持执行时间分析和时序可视化。本章还汇总了 5 类常见问题（E_OS_LIMIT、E_OS_STACKFAULT、E_OS_RESOURCE、E_OS_CALLEVEL、中断优先级配置错误）的典型原因和排查步骤。与 FreeRTOS 的 configASSERT 和 StackOverflowHook 相比，AUTOSAR OS 提供了更系统化的错误处理框架。

FreeRTOS 对比：FreeRTOS 使用 configASSERT() 宏和 vApplicationStackOverflowHook() 提供类似功能。AUTOSAR OS 的错误处理更加结构化和规范化，适合车规级系统的错误管理需求。

第十七章 实战案例与工程模板

本章通过一系列由浅入深的工程实例，帮助读者将前面章节学到的 AUTOSAR OS 知识真正落地到 TC334 LiteKit 硬件平台上。每个案例均基于本书配套源码工程，采用渐进式教学方式——从最简单的单任务配置开始，逐步增加复杂度，最终形成完整可编译的工程模板。

17.1 周期任务模板（10ms/50ms/100ms 周期任务）

17.1.1 设计目标

在嵌入式实时控制系统中，周期任务是最常见的执行模式。本节构建一个典型的三级周期任务系统：

任务	周期	优先级	典型用途
Task_10ms	10ms	4（最高）	电机控制、PID 运算
Task_50ms	50ms	3	传感器采样、滤波处理
Task_100ms	100ms	2	状态监控、诊断上报

设计原则：周期越短 → 实时性要求越高 → 优先级越高（AUTOSAR SWS_Os_00033：数值越大优先级越高）。

17.1.2 Step 1：单个 10ms 周期任务（最简配置）

从零开始，仅配置一个 10ms 周期任务，理解 Alarm 驱动周期任务的核心机制。

Os_Cfg.h（关键宏定义）：

```
    /* ===== Step 1: 单个 10ms 周期任务 ===== */

    /* 系统时钟：STM 产生 1ms 一次的 tick（CFG_REG_OSTIMER_VALUE_CORE0 = 100000）*/
    /* 即 SystemCounter 每 1ms 递增一次 */
    #define      CFG_REG_OSTIMER_VALUE_CORE0            (100000U)    /* 100MHz / 100000 =
1ms */

    /* 任务数量：Task_10ms + Idle = 2 */
    #define      CFG_TASK_MAX                           (2U)
    #define      CFG_TASK_MAX_CORE0                     (2U)
    #define      CFG_EXTENDED_TASK_MAX                   (0U)    /*
    #define      CFG_EXTENDED_TASK_MAX_CORE0             (0U)

    /* 任务 ID 定义（类型为 Os_TaskType，本质是 uint16） */
    #define      Task_10ms                              ((Os_TaskType)0x0000U)
    #define      OS_TASK_IDLE_CORE0                     ((Os_TaskType)0x0001U)

    /* 优先级：0=Idle, 1=Task_10ms（数值越大越高） */
    #define      CFG_PRIORITY_MAX_CORE0                 (2U)    /*
    #define      CFG_PRIORITY_GROUP_CORE0               (1U)    /*

    /* Alarm 配置：1 个 Alarm 驱动 Task_10ms */
    #define      CFG_ALARM_MAX                          (1U)
    #define      CFG_ALARM_MAX_CORE0                    (1U)
    #define      Alarm_10ms                             ((Os_AlarmType)0x0000U)

    /* Counter 配置：1 个硬件 Counter（由 STM 驱动） */
    #define      CFG_COUNTER_MAX                        (1U)
    #define      CFG_COUNTER_MAX_CORE0                  (1U)
    #define      SystemCounter                          ((Os_CounterType)0x0000U)

    /* Hooks */
    #define      CFG_STARTUPHOOK                        TRUE    /*
    #define      CFG_ERRORHOOK                          TRUE
```

App_Tasks.c（任务实现）：

```
#include "Os.h"
#include "IfxPort.h"

#define LED_PORT    &MODULE_P00
#define LED_PIN     5u    /* TC334 LiteKit LED

static volatile uint32 g_Task10ms_Counter = 0U;

/**
 * @brief 10ms 周期任务
 * 每 10ms 被 Alarm_10ms 激活一次。
 * 业务逻辑：递增计数器 + 每 50 次（500ms）翻转 LED
 */
TASK(Task_10ms)
{
    g_Task10ms_Counter++;

    /* 每 500ms 翻转 LED（10ms * 50 = 500ms） */
    if ((g_Task10ms_Counter % 50U) == 0U)
    {
        IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_toggled);
    }

    TerminateTask();  /* 必须调用！否则 OS 进入 ErrorHook */
}
```

Os_UserInf.c（StartupHook 启动 Alarm）：

```
#include "Os.h"

void StartupHook(void)
{
    /* 启动 Alarm_10ms：
     * - increment = 10: 首次触发在 10 ticks（10ms）后
     * - cycle = 10: 之后每 10 ticks（10ms）循环触发
     * 注意：increment 必须 > 0（AUTOSAR 规范要求）
     */
    (void)SetRelAlarm(Alarm_10ms, 10U, 10U);
}
```

Cpu0_Main.c（系统启动）：

```
#include "Ifx_Types.h"
#include "IfxCpu.h"
#include "IfxScuWdt.h"
#include "Os.h"

IFX_ALIGN(4) IfxCpu_syncEvent g_cpuSyncEvent = 0;

void core0_main(void)
{
    IfxCpu_enableInterrupts();
    /* 关闭看门狗，避免开发阶段复位 */
    IfxScuWdt_disableCpuWatchdog(IfxScuWdt_getCpuWatchdogPassword());
    IfxScuWdt_disableSafetyWatchdog(IfxScuWdt_getSafetyWatchdogPassword());
    /* 多核同步（单核工程也需要） */
    IfxCpu_emitEvent(&g_cpuSyncEvent);
    IfxCpu_waitEvent(&g_cpuSyncEvent, 1);
    /* 启动 OS — 此函数不返回 */
    StartOS(OSDEFAULTAPPMODE);
    while(1) { }
}
```

17.1.3 Step 2：添加 50ms 任务（多优先级）

在 Step 1 基础上新增 Task_50ms，理解多任务优先级配置。

Os_Cfg.h 差异（新增配置）：

```
/* ===== Step 2: 新增 50ms 任务 ===== */
#define     CFG_TASK_MAX                            (3U)    /* 2-
#define     CFG_TASK_MAX_CORE0                      (3U)

#define     Task_10ms                               ((Os_TaskType)0x0000U)
#define     Task_50ms                               ((Os_TaskType)0x0001U)    /*
#define     OS_TASK_IDLE_CORE0                      ((Os_TaskType)0x0002U)

/* 优先级：0=Idle, 1=Task_50ms, 2=Task_10ms */
#define     CFG_PRIORITY_MAX_CORE0                  (3U)    /* 2-

/* Alarm：新增 Alarm_50ms */
#define     CFG_ALARM_MAX                           (2U)                      /* 1-
#define     CFG_ALARM_MAX_CORE0                     (2U)
#define     Alarm_10ms                              ((Os_AlarmType)0x0000U)
#define     Alarm_50ms                              ((Os_AlarmType)0x0001U)    /*
```

为什么 Task_10ms 优先级更高？10ms 任务的截止时间更紧迫，必须在 10ms 内完成；若 50ms 任务正在执行，10ms 任务到达时必须能抢占。这是 Rate-Monotonic Scheduling（RMS）原则的直接应用。

新增任务实现（App_Tasks.c）：

```
static volatile uint16 g_AdcResult = 0U;
static volatile uint32 g_Task50ms_Counter = 0U;


/**
 * @brief 50ms 周期任务 - 传感器采样
 * 优先级低于 10ms 任务，可被抢占
 */
TASK(Task_50ms)
{
    g_Task50ms_Counter++;
    /* 模拟 ADC 采样 */
    g_AdcResult = (uint16)(g_Task50ms_Counter & 0xFFFU);
    TerminateTask();
}
```

StartupHook 更新：

```
void StartupHook(void)
{
    (void)SetRelAlarm(Alarm_10ms, 10U, 10U);
    /* 启动 50ms Alarm, increment=50, cycle=50 */
    (void)SetRelAlarm(Alarm_50ms, 50U, 50U);
}
```

17.1.4 Step 3：添加 100ms 任务 + 完整配置

完整三级周期任务系统 Os_Cfg.h：

```
/* ===== Step 3: 完整三级周期任务系统 ===== */
#define     CFG_TASK_MAX                            (4U)    /* 10ms+50ms+100ms+Idle
*/
#define     CFG_TASK_MAX_CORE0                      (4U)
#define     CFG_EXTENDED_TASK_MAX                    (0U)

#define     Task_10ms                               ((Os_TaskType)0x0000U)
#define     Task_50ms                               ((Os_TaskType)0x0001U)
#define     Task_100ms                              ((Os_TaskType)0x0002U)
```

```c
#define     OS_TASK_IDLE_CORE0                          ((Os_TaskType)0x0003U)

/* 优先级层次：0=Idle < 1=100ms < 2=50ms < 3=10ms */
#define     CFG_PRIORITY_MAX_CORE0                      (4U)
#define     CFG_SCHED_POLICY                            OS_PREEMPTIVE_MIXED

/* Alarm 定义 */
#define     CFG_ALARM_MAX                               (3U)
#define     CFG_ALARM_MAX_CORE0                         (3U)
#define     Alarm_10ms                                  ((Os_AlarmType)0x0000U)
#define     Alarm_50ms                                  ((Os_AlarmType)0x0001U)
#define     Alarm_100ms                                 ((Os_AlarmType)0x0002U)
```

完整 App_Tasks.c：

```c
#include "Os.h"
#include "IfxPort.h"

#define LED_PORT    &MODULE_P00
#define LED_PIN     5u

static volatile uint32 g_Task10ms_Counter  = 0U;
static volatile uint32 g_Task50ms_Counter  = 0U;
static volatile uint32 g_Task100ms_Counter = 0U;
static volatile uint16 g_AdcResult         = 0U;
static volatile uint8  g_SystemStatus      = 0U;    /* 0=

/* ========== 10ms 高优先级控制任务 ========== */
TASK(Task_10ms)
{
    g_Task10ms_Counter++;
    /* PID 控制逻辑（实际工程中读编码器、算 PID、更新 PWM） */
    if ((g_Task10ms_Counter % 50U) == 0U)
        IfxPort_setPinState(LED_PORT, LED_PIN, IfxPort_State_toggled);
    TerminateTask();
}

/* ========== 50ms 中优先级采样任务 ========== */
TASK(Task_50ms)
{
    g_Task50ms_Counter++;
    uint16 rawValue = (uint16)(g_Task50ms_Counter & 0xFFFU);
    g_AdcResult = (uint16)((g_AdcResult * 7U + rawValue) / 8U);  /* 低通滤波 */
    TerminateTask();
}

/* ========== 100ms 低优先级监控任务 ========== */
TASK(Task_100ms)
{
    g_Task100ms_Counter++;
    if (g_AdcResult > 3500U)      g_SystemStatus = 2U;  /* 故障 */
    else if (g_AdcResult > 3000U) g_SystemStatus = 1U;  /* 警告 */
    else                          g_SystemStatus = 0U;  /* 正常 */
    TerminateTask();
}
```

完整 Os_UserInf.c（Alarm 偏移策略）：

```c
void StartupHook(void)
{
    /* 错开首次触发时间，避免多任务同时就绪 */
    (void)SetRelAlarm(Alarm_10ms,  1U, 10U);   /* 首次 1ms, 周期 10ms */
    (void)SetRelAlarm(Alarm_50ms,  3U, 50U);   /* 首次 3ms, 周期 50ms */
    (void)SetRelAlarm(Alarm_100ms, 7U, 100U);  /* 首次 7ms, 周期 100ms */
}
```

17.1.5 运行时序分析

关键设计点：

1. Alarm 偏移：首次触发时间错开（1/3/7ms），避免多任务同时进入就绪队列

2. 优先级分配：严格遵循 RMS 原则——频率越高优先级越高

3. 执行时间预算：每个任务必须在其周期内完成，否则下次激活将失败（osTaskActivation=1）

17.2 状态机任务模板（Event 驱动）

17.2.1 设计目标

Event 驱动的扩展任务（Extended Task）适用于"等待事件→处理→再等待"的模式，典型应用于通信协议状态机、按键处理等场景。

与 Basic Task 的关键区别：Extended Task 可调用 WaitEvent()，进入 WAITING 状态释放 CPU；Basic Task 只能 TerminateTask() 后等待下次激活。

17.2.2 Os_Cfg.h（Event 相关配置）

```
    /* ===== Event 驱动状态机配置 ===== */
    #define     CFG_EXTENDED_TASK_MAX                    (1U)
    #define     CFG_EXTENDED_TASK_MAX_CORE0              (1U)

    #define     CFG_TASK_MAX                            (3U)                    /*
StateMachine + Init + Idle */
    #define     Task_Init                               ((Os_TaskType)0x0000U)
    #define     Task_StateMachine                       ((Os_TaskType)0x0001U)    /*
Extended */
    #define     OS_TASK_IDLE_CORE0                      ((Os_TaskType)0x0002U)

    /* Event 定义（位掩码，每个 Event 占一个 bit） */
    #define     Evt_DataReady                           ((EventMaskType)0x0001U)
    #define     Evt_Timeout                             ((EventMaskType)0x0002U)
    #define     Evt_Error                               ((EventMaskType)0x0004U)
```

17.2.3 状态机任务实现（App_Tasks.c）

```
    #include "Os.h"
    #include "IfxPort.h"

    typedef enum {
        STATE_IDLE = 0,      /* 空闲：等待数据 */
        STATE_PROCESSING,    /* 处理中 */
        STATE_ERROR,         /* 错误：执行恢复流程 */
    } AppState_t;

    static volatile AppState_t g_CurrentState = STATE_IDLE;

    /**
     * @brief 状态机任务（Extended Task）
     * 执行流程：WaitEvent -> 事件到达 -> 状态转换 -> ClearEvent -> 循环
     * 注意：Extended Task 不调用 TerminateTask()，在内部循环等待
     */
    TASK(Task_StateMachine)
    {
```

```
    EventMaskType events;
    for (;;)
    {
        (void)WaitEvent(Evt_DataReady | Evt_Timeout | Evt_Error);
        (void)GetEvent(Task_StateMachine, &events);
        (void)ClearEvent(Evt_DataReady | Evt_Timeout | Evt_Error);

        if ((events & Evt_Error) != 0U) {
            g_CurrentState = STATE_ERROR;
            /* 错误恢复逻辑... */
            g_CurrentState = STATE_IDLE;
        } else if ((events & Evt_DataReady) != 0U) {
            g_CurrentState = STATE_PROCESSING;
            /* 处理数据... */
            IfxPort_setPinState(&MODULE_P00, 5u, IfxPort_State_toggled);
            g_CurrentState = STATE_IDLE;
        } else if ((events & Evt_Timeout) != 0U) {
            g_CurrentState = STATE_IDLE;
        }
    }
}
```

17.2.4 ISR 触发事件

在中断中使用 SetEvent() 通知状态机任务：

```
/* 中断服务程序（例如：CAN 接收中断） */
void Os_ISR_CanRx_Handler(void)
{
    OS_ARCH_ISR2_PROLOGUE(Os_GetObjLocalId(CFG_ISR_CAN_RX_ID));
    /* 读取数据... */
    (void)SetEvent(Task_StateMachine, Evt_DataReady);
    OS_ARCH_ISR2_EPILOGUE();
}
```

17.3 生产者-消费者模式（中断采集→事件通知→任务处理）

17.3.1 设计目标

构建经典的"ISR 采集数据 → Event 通知 → Task 处理"流水线。这种模式的优势：

1. 最短中断响应时间：ISR 只做最少工作（采集+通知）

2. 数据完整性：通过缓冲区解耦生产和消费速率差异

3. 实时性保证：Task 在 Event 驱动下及时响应

17.3.2 环形缓冲区 + 消费者任务

```
#include "Os.h"
#include "IfxPort.h"

#define BUFFER_SIZE     16U
#define BUFFER_MASK     (BUFFER_SIZE - 1U)

typedef struct {
    uint16 data[BUFFER_SIZE];
    volatile uint8 head;    /* ISR
    volatile uint8 tail;    /* Task
} RingBuffer_t;
```

```c
static RingBuffer_t g_SampleBuffer = { {0}, 0U, 0U };
static volatile uint16 g_FilteredValue = 0U;

/* 缓冲区写入（仅 ISR 调用，无需加锁） */
static inline boolean RingBuffer_Write(RingBuffer_t* buf, uint16 value)
{
    uint8 nextHead = (buf->head + 1U) & BUFFER_MASK;
    if (nextHead == buf->tail) return FALSE;  /* 满 */
    buf->data[buf->head] = value;
    buf->head = nextHead;
    return TRUE;
}

/* 缓冲区读取（仅 Task 调用） */
static inline boolean RingBuffer_Read(RingBuffer_t* buf, uint16* value)
{
    if (buf->head == buf->tail) return FALSE;  /* 空 */
    *value = buf->data[buf->tail];
    buf->tail = (buf->tail + 1U) & BUFFER_MASK;
    return TRUE;
}

#define Evt_NewData    ((EventMaskType)0x0001U)
#define Evt_BufferFull ((EventMaskType)0x0002U)

/* 消费者任务（Extended Task） */
TASK(Task_DataProc)
{
    uint16 sample;
    uint32 sum;
    uint8  count;
    for (;;)
    {
        (void)WaitEvent(Evt_NewData | Evt_BufferFull);
        (void)ClearEvent(Evt_NewData | Evt_BufferFull);
        sum = 0U; count = 0U;
        while (RingBuffer_Read(&g_SampleBuffer, &sample) == TRUE) {
            sum += sample;
            count++;
        }
        if (count > 0U)
            g_FilteredValue = (uint16)(sum / count);
        IfxPort_setPinState(&MODULE_P00, 5u, IfxPort_State_toggled);
    }
}
```

17.3.3 生产者（Alarm 回调）

```c
/* 在 Os_Cfg.c 中定义 Alarm 回调 */
static uint32 g_SampleCount = 0U;

static void AlarmCallback_Sample(void)
{
    static uint16 s_SimValue = 0U;
    s_SimValue = (s_SimValue + 17U) & 0xFFFU;
    g_SampleCount++;

    if (RingBuffer_Write(&g_SampleBuffer, s_SimValue) == FALSE) {
        (void)SetEvent(Task_DataProc, Evt_BufferFull);
    } else {
        /* 每 4 个样本通知一次（批量处理，减少调度开销） */
        if ((g_SampleCount % 4U) == 0U)
            (void)SetEvent(Task_DataProc, Evt_NewData);
```

```
    }
}
```

17.4 多优先级系统示例（安全关键/实时控制/后台诊断三层）

17.4.1 架构设计

实际汽车 ECU 通常采用三层优先级架构：

层级	优先级范围	任务类型	典型功能
安全层	6-7	Non-preemptive	看门狗喂狗、安全监控
控制层	3-5	Full-preemptive	PID 控制、PWM、通信
后台层	1-2	Full-preemptive	诊断、标定、日志

17.4.2 安全任务示例

```
/**
 * @brief 安全监控任务（最高优先级，不可抢占）
 * 功能：喂看门狗 + 检查关键参数 + 超限保护
 * 设为 NON-PREEMPTIVE：安全检查必须原子完成
 */
TASK(Task_Safety)
{
    /* 1. 喂看门狗 */
    /* IfxScuWdt_serviceCpuWatchdog(...); */

    /* 2. 温度超限保护 */
    if (g_SharedData.temperature > 1200U)  /* > 120 C */
    {
        g_SharedData.motorSpeed = 0U;
        g_SharedData.errorCode = 0x10U;  /* 过温故障 */
    }
    TerminateTask();
}
```

17.4.3 控制任务与 Resource 保护

```
/**
 * @brief 5ms 快速控制任务 - 电机 PID
 * 访问共享数据需要获取 Resource
 */
TASK(Task_FastCtrl)
{
    uint16 speedTarget = 1000U;
    uint16 speedActual;

    (void)GetResource(Res_SharedData);
    speedActual = g_SharedData.motorSpeed;
    /* PID 计算... */
    g_SharedData.motorSpeed = speedActual + (uint16)((sint16)(speedTarget -
speedActual) / 10);
    (void)ReleaseResource(Res_SharedData);

    TerminateTask();
}
```

Resource 优先级天花板配置：天花板优先级 = max(所有使用该 Resource 的任务优先级)。例

如 Task_FastCtrl(5) 和 Task_SlowCtrl(3) 都使用 Res_SharedData，则天花板优先级 = 5。

17.4.4 后台诊断任务

```
/**
 * @brief 诊断任务（Extended Task，Event 驱动）
 * 等待外部诊断请求
 */
TASK(Task_Diag)
{
    EventMaskType events;
    for (;;)
    {
        (void)WaitEvent(Evt_DiagRequest | Evt_DiagResponse);
        (void)GetEvent(Task_Diag, &events);
        (void)ClearEvent(Evt_DiagRequest | Evt_DiagResponse);
        if ((events & Evt_DiagRequest) != 0U) {
            (void)GetResource(Res_SharedData);
            /* 读取诊断信息... */
            (void)ReleaseResource(Res_SharedData);
        }
    }
}
```

17.5 从零搭建工程（基于 TC334 LiteKit）

17.5.1 工程目录结构

```
TC334_AUTOSAR_OS_Project/
+-- App/
|   +-- App_Tasks.c              <-- 用户任务实现
+-- Os_Cfg/
|   +-- Os_Cfg.h                 <-- OS 配置头文件（宏定义）
|   +-- Os_Cfg.c                 <-- OS 配置实现（数据结构）
|   +-- Os_CfgData.h             <-- OS 配置数据声明
|   +-- Os_CoreCfg.h/c           <-- 中断向量配置
|   +-- Os_Intvet.c              <-- ISR2 包装函数
|   +-- Os_UserInf.c             <-- 用户 Hook 和回调
+-- RTOS/
|   +-- Kernel/                  <-- AUTOSAR OS 内核（不修改）
|   +-- Portable/Mcu/Infineon/TC334/  <-- TC334 移植层
+-- Libraries/iLLD/TC33A/        <-- Infineon 底层驱动
+-- Cpu0_Main.c                  <-- Core0 入口
+-- Lcf_Gnuc_Tricore_Tc.lsl      <-- GCC 链接脚本
```

17.5.2 最小配置模板

以下是可直接用于新工程的最小 Os_Cfg.h 模板（单核、BCC1、1 个用户任务）：

```
#ifndef OS_CFG_H
#define OS_CFG_H
#include "Std_Types.h"

/* Core */
#define CFG_CORE_MAX                          (1U)
#define OS_CORE_ID_MASTER                     ((Os_CoreIdType)0U)
#define OS_AUTOSAR_CORES                      1U
#define CFG_CORE0_AUTOSAROS_ENABLE            TRUE

/* System */
```

```c
#define CFG_SC                                  OS_SC1
#define CFG_CC                                  OS_BCC1
#define CFG_STATUS                              OS_STATUS_STANDARD
#define OSDEFAULTAPPMODE                        ((Os_AppModeType)0x1U)
#define CFG_SYSTEM_TIMER_ENABLE                 TRUE
#define CFG_SYSTEM_TIMER_ENABLE_CORE0           TRUE
#define CFG_REG_OSTIMER_VALUE_CORE0             (100000U)    /* 1ms tick */

/* Task */
#define CFG_SCHED_POLICY                        OS_PREEMPTIVE_MIXED
#define CFG_PRIORITY_MAX_CORE0                  (2U)
#define CFG_PRIORITY_GROUP_CORE0                (1U)
#define CFG_TASK_MAX                            (2U)
#define CFG_TASK_MAX_CORE0                      (2U)
#define CFG_EXTENDED_TASK_MAX                    (0U)
#define Task_App              ((Os_TaskType)0x0000U)
#define OS_TASK_IDLE_CORE0    ((Os_TaskType)0x0001U)

/* ISR */
#define CFG_ISR_MAX                             (1U)
#define CFG_ISR2_MAX                            (1U)
#define CFG_ISR2_IPL_MAX_CORE0                  (2U)

/* Counter & Alarm */
#define CFG_COUNTER_MAX                         (1U)
#define SystemCounter         ((Os_CounterType)0x0000U)
#define CFG_ALARM_MAX                           (1U)
#define Alarm_App             ((Os_AlarmType)0x0000U)

/* Hooks */
#define CFG_STARTUPHOOK                         TRUE
#define CFG_ERRORHOOK                           TRUE
#define CFG_SHUTDOWNHOOK                        TRUE

#endif    /* OS_CFG_H */
```

17.5.3 常见编译错误及解决方法

错误信息	原因	解决方法
undefined reference to Os_TaskEntry_xxx	缺少 TASK(xxx) 定义	确保 App_Tasks.c 中有对应的 TASK() 宏
CFG_PRIORITY_MAX too small	优先级数量不足	增大 CFG_PRIORITY_MAX_CORE0
SetRelAlarm returns E_OS_ID	Alarm ID 越界	检查 CFG_ALARM_MAX 和 Alarm ID
TerminateTask missing	任务未正确终止	Basic Task 必须以 TerminateTask() 结束

17.6 CAN/LIN 通信集成示例

17.6.1 架构概述

将 CAN 通信与 AUTOSAR OS 集成的典型架构：ISR2 接收中断 → 读取报文存入队列 → SetEvent 通知处理任务 → Task 解析报文。

17.6.2 CAN 接收 ISR2

```c
/* CAN 接收中断 */
#define CAN_RX_QUEUE_SIZE   8U
typedef struct {
    uint32 id;
    uint8  data[8];
    uint8  dlc;
```

```c
} CanFrame_t;

static volatile CanFrame_t g_CanRxQueue[CAN_RX_QUEUE_SIZE];
static volatile uint8 g_CanRxHead = 0U;
static volatile uint8 g_CanRxTail = 0U;

void Os_ISR_CanRx_Handler(void)
{
    OS_ARCH_ISR2_PROLOGUE(Os_GetObjLocalId(CFG_ISR_CAN_RX_ID));

    IfxCan_Message rxMsg;
    uint32 rxData[2] = {0};
    IfxCan_Can_readMessage(&g_CanNode0, &rxMsg, rxData);

    uint8 nextHead = (g_CanRxHead + 1U) % CAN_RX_QUEUE_SIZE;
    if (nextHead != g_CanRxTail) {
        g_CanRxQueue[g_CanRxHead].id = rxMsg.messageId;
        g_CanRxQueue[g_CanRxHead].dlc = (uint8)rxMsg.dataLengthCode;
        /* 拷贝数据... */
        g_CanRxHead = nextHead;
    }
    (void)SetEvent(Task_CanProc, Evt_CanRxReady);

    OS_ARCH_ISR2_EPILOGUE();
}
```

17.6.3 CAN 处理任务

```c
/**
 * @brief CAN 接收处理任务（Extended Task）
 * ISR 将报文放入队列后通过 SetEvent 唤醒本任务
 */
TASK(Task_CanProc)
{
    CanFrame_t frame;
    for (;;)
    {
        (void)WaitEvent(Evt_CanRxReady | Evt_CanError);
        (void)ClearEvent(Evt_CanRxReady | Evt_CanError);

        while (g_CanRxHead != g_CanRxTail) {
            frame = g_CanRxQueue[g_CanRxTail];
            g_CanRxTail = (g_CanRxTail + 1U) % CAN_RX_QUEUE_SIZE;
            switch (frame.id) {
                case 0x100U: /* 电机控制命令 */ break;
                case 0x200U: /* 传感器数据 */   break;
                case 0x7DFU: /* UDS 诊断请求 */ break;
                default: break;
            }
        }
    }
}
```

17.6.4 CAN 周期发送任务

```c
/**
 * @brief CAN 周期发送任务（10ms，Basic Task）
 */
TASK(Task_CanTx)
{
    static uint8 s_TxCounter = 0U;
    uint32 txData[2] = {0};
```

```
        IfxCan_Message txMsg;

        txMsg.messageId = 0x180U;
        txMsg.dataLengthCode = IfxCan_DataLengthCode_8;
        txMsg.frameMode = IfxCan_FrameMode_standard;

        /* 填充数据 */
        txData[0] = ((uint32)g_SharedData.motorSpeed << 16U) | g_SharedData.temperature;
        txData[1] = ((uint32)g_SharedData.errorCode << 24U) | ((uint32)s_TxCounter <<
16U);

        IfxCan_Can_sendMessage(&g_CanNode0, &txMsg, txData);
        s_TxCounter++;
        TerminateTask();
    }
```

17.6.5 中断安装表配置

```
    /* Os_CoreCfg.c - 中断安装表 */
    static const Os_IntInstallType Os_IntInstallCore0[CFG_ISR_MAX_CORE0] =
    {
        { 2U, Os_ISR_OsTimerSourceCore_0_Handler0 },  /* 系统定时器 */
        { 4U, Os_ISR_CanRx_Handler },                 /* CAN 接收 */
    };
```

17.7 本章小结

本章通过六个逐步递进的实战案例，展示了 AUTOSAR OS 在 TC334 LiteKit 平台上的典型应用模式：

1. 周期任务模板（17.1）：Alarm 驱动的多优先级周期任务系统，RMS 优先级分配和 Alarm 偏移策略。

2. 状态机任务模板（17.2）：Extended Task + Event 机制实现事件驱动编程。

3. 生产者-消费者模式（17.3）：ISR 采集→缓冲→Task 处理的经典流水线。

4. 多优先级系统（17.4）：安全/控制/后台三层架构，Resource 保护共享数据。

5. 从零搭建工程（17.5）：完整的工程模板和配置步骤，可作为新项目起点。

6. CAN 通信集成（17.6）：iLLD CAN 驱动与 OS ISR2 机制结合的完整流程。

读者在实际项目中可以根据需求组合使用这些模板。建议从 17.5 的最简工程开始，确认编译运行无误后，再逐步添加 17.1~17.4 的功能。

第十八章 性能优化与最佳实践

嵌入式实时系统的性能优化与桌面应用有本质差异——它不仅关乎"快"，更关乎"确定性"。在 AUTOSAR OS + TC334 平台上，一个优化不当的系统可能在平均负载仅 30% 时就出现偶发的任务超时，而一个精心设计的系统可以在 70% 负载下仍保持稳定的实时响应。

本章系统性地讲解 AUTOSAR OS 在 TC334 平台上的性能优化策略，覆盖任务划分、中断响应、栈空间、CPU 负载、资源争用和内存布局六大维度，并在最后汇总常见陷阱与规避方法。每个优化策略均配合基于 TC334 的代码示例，提供"优化前 vs 优化后"的对比和量化判断标准。

18.1 任务划分策略

任务划分是 AUTOSAR OS 系统设计的第一步，也是影响最深远的一步。错误的任务划分会导致后续所有优化工作事倍功半。

18.1.1 按周期划分（时间触发型）

核心思想：将相同执行周期的功能聚合到同一个任务中，由 Alarm 按固定周期激活。

适用场景：控制类 ECU（如电机控制、底盘控制），功能执行时间确定、周期固定。

设计原则： - 同周期功能放入同一任务，减少任务切换开销 - 不同周期使用不同任务，由独立 Alarm 驱动 - 遵循 RMS（Rate Monotonic Scheduling）：周期越短 → 优先级越高

```
/* ===== 按周期划分的任务配置示例 ===== */

/* 任务定义（Os_Cfg.h） */
#define Task_1ms     ((Os_TaskType)0x0000U)    /*
#define Task_5ms     ((Os_TaskType)0x0001U)    /*
#define Task_10ms    ((Os_TaskType)0x0002U)    /*
#define Task_100ms   ((Os_TaskType)0x0003U)    /*

/* 优先级分配（数值越大越高） */
/* Task_1ms: Priority 5  — 1ms 周期，最高优先级 */
/* Task_5ms: Priority 4  — 5ms 周期 */
/* Task_10ms: Priority 3 — 10ms 周期 */
/* Task_100ms: Priority 2 — 100ms 周期 */
/* Idle: Priority 0       — 系统空闲任务 */

/* Alarm 配置（Os_Cfg.c）：各 Alarm 使用相同 Counter（SystemCounter, 1ms/tick） */
/* Alarm_1ms:   cycle=1,   offset=0  → 激活 Task_1ms */
/* Alarm_5ms:   cycle=5,   offset=1  → 激活 Task_5ms，偏移 1ms 避免同时激活 */
/* Alarm_10ms:  cycle=10,  offset=2  → 激活 Task_10ms */
/* Alarm_100ms: cycle=100, offset=5  → 激活 Task_100ms */
```

量化判断标准： - 单个周期任务的 WCET（最坏执行时间）不超过其周期的 60% - 所有任务的 CPU 利用率总和（按 RMS 理论）不超过 `n*(2^(1/n)-1)` ≈ 69%（4 个任务时） - Alarm 偏移量应大于最高优先级任务的 WCET

18.1.2 按优先级划分（事件触发型）

核心思想：将不同紧急程度的功能分配到不同优先级的任务，由事件（Event）或外部中断触发执行。

适用场景：通信类 ECU（如网关、HMI），功能执行时机不确定、响应时间要求各异。

```
/* ===== 按优先级划分的任务配置示例 ===== */

/* 高优先级：安全相关功能（必须在 5ms 内响应） */
TASK(Task_Safety)
```

```
{
    EventMaskType ev;
    while (1) {
        /* 等待安全相关事件（如过压、过流告警） */
        WaitEvent(EVT_OVERVOLTAGE | EVT_OVERCURRENT);
        GetEvent(Task_Safety, &ev);
        ClearEvent(ev);

        if (ev & EVT_OVERVOLTAGE) {
            Safety_HandleOvervoltage();   /* 紧急切断输出 */
        }
        if (ev & EVT_OVERCURRENT) {
            Safety_HandleOvercurrent();   /* 紧急限流 */
        }
    }
    TerminateTask();
}

/* 中优先级：通信处理（10ms 内响应即可） */
TASK(Task_Communication)
{
    EventMaskType ev;
    while (1) {
        WaitEvent(EVT_CAN_RX | EVT_LIN_RX);
        GetEvent(Task_Communication, &ev);
        ClearEvent(ev);

        if (ev & EVT_CAN_RX) {
            Com_ProcessCanMessage();     /* 解析 CAN 报文 */
        }
        if (ev & EVT_LIN_RX) {
            Com_ProcessLinFrame();       /* 解析 LIN 帧 */
        }
    }
    TerminateTask();
}

/* 低优先级：后台诊断（无严格时间约束） */
TASK(Task_Background)
{
    EventMaskType ev;
    while (1) {
        WaitEvent(EVT_DIAG_REQUEST);
        GetEvent(Task_Background, &ev);
        ClearEvent(ev);

        Diag_ProcessRequest();           /* 处理 UDS 诊断请求 */
    }
    TerminateTask();
}
```

设计要点：- 事件触发型任务必须使用 Extended Task（支持 WaitEvent）- 优先级层次建议不超过 4~5 层，过多层次增加分析复杂度 - 每层优先级之间应有明确的响应时间需求差异（至少 2 倍）

18.1.3 按功能模块划分

核心思想：每个软件功能模块（SWC）对应一个独立任务，模块间通过 OS 机制通信。

适用场景：大型 ECU 项目（多 SWC 集成）、需要模块化开发和独立测试。

```
/* ===== 按功能模块划分 ===== */
```

```
/* 模块 A：电机控制 SWC */
TASK(Task_MotorCtrl)
{
    MotorCtrl_MainFunction();   /* 10ms 周期：读取编码器、PI 计算、输出 PWM */
    TerminateTask();
}

/* 模块 B：热管理 SWC */
TASK(Task_ThermalMgmt)
{
    ThermalMgmt_MainFunction(); /* 100ms 周期：NTC 采样、风扇控制 */
    TerminateTask();
}

/* 模块 C：通信 SWC */
TASK(Task_ComStack)
{
    Com_MainFunctionRx();       /* 5ms 周期：CAN/LIN 接收处理 */
    Com_MainFunctionTx();       /* CAN/LIN 发送处理 */
    TerminateTask();
}

/* 优势：各 SWC 可独立开发、独立配置栈空间 */
/* 劣势：任务数量多，切换开销增大 */
```

权衡准则：

因素	按周期	按优先级	按功能
任务数量	少（3~5）	中（4~6）	多（8~15）
切换开销	低	中	高
模块化	差	中	优
时序分析	容易	中等	复杂
适用规模	小型 ECU	中型 ECU	大型 ECU

18.1.4 任务划分的反模式

反模式 1：巨型任务（God Task）

```
/* ✕ 错误示例：一个任务包含所有功能 */
TASK(Task_Main)
{
    /* 100+ 行代码，混合不同周期需求的功能 */
    Motor_Control();        /* 需要 1ms 周期 */
    Sensor_Sample();        /* 需要 10ms 周期 */
    Diagnosis_Update();     /* 需要 100ms 周期 */
    Com_MainFunction();     /* 需要 5ms 周期 */
    TerminateTask();
}
/* 问题：所有功能被迫以相同周期运行，浪费 CPU；无法针对性设置优先级 */
```

```
/* ✓ 正确做法：按周期拆分 */
TASK(Task_1ms)  { Motor_Control(); TerminateTask(); }
TASK(Task_5ms)  { Com_MainFunction(); TerminateTask(); }
TASK(Task_10ms) { Sensor_Sample(); TerminateTask(); }
TASK(Task_100ms){ Diagnosis_Update(); TerminateTask(); }
```

反模式 2：任务过度拆分

```
/* ✕ 错误示例：相同周期的功能拆成多个任务 */
TASK(Task_AdcSample)   { Adc_ReadChannel(); TerminateTask(); }   /* 10ms */
TASK(Task_AdcFilter)   { Filter_Apply(); TerminateTask(); }       /* 10ms */
TASK(Task_AdcConvert)  { Unit_Convert(); TerminateTask(); }       /* 10ms */
/* 问题：3 次任务切换 = 3 次 CSA 上下文保存/恢复 = 3×64 bytes CSA + 切换时间 */
```

```
/* ✓ 正确做法：同周期功能合并 */
TASK(Task_Sensor_10ms)
{
    Adc_ReadChannel();     /* 步骤 1：采样 */
    Filter_Apply();        /* 步骤 2：滤波 */
    Unit_Convert();        /* 步骤 3：单位转换 */
    TerminateTask();
}
/* 节省：2 次任务切换开销（约 2~4μs on TC334 @300MHz） */
```

反模式 3：不必要的高优先级

```
/* ✕ 错误示例：所有任务都设为高优先级 */
/* Task_A: Priority 10, Task_B: Priority 9, Task_C: Priority 8 */
/* 问题：优先级区分度不够，实际退化为 FIFO 调度，失去实时性保证 */

/* ✓ 正确做法：明确优先级层次，预留间隔 */
/* Safety:     Priority 10 */
/* Control:    Priority 6  — 预留 7~9 给未来扩展 */
/* Comm:       Priority 3  — 预留 4~5 */
/* Background: Priority 1 */
```

18.2 中断响应优化

在 AUTOSAR OS 中，中断是连接硬件事件与软件处理的桥梁。TC334 的中断系统（IR）支持最多 255 级优先级，合理利用 Cat1/Cat2 分类和优先级分配，是实现微秒级响应的关键。

18.2.1 Cat1 vs Cat2 选择策略

Cat1 中断：不经过 OS 管理，响应最快（无 OS 开销），但不能调用任何 OS API。

Cat2 中断：由 OS 管理，可调用受限的 OS API（如 ActivateTask、SetEvent），但有 OS 框架开销。

特性	Cat1 ISR	Cat2 ISR
OS API 调用	禁止	允许（受限）
响应延迟	极低（< 100ns @300MHz）	较低（~500ns~1μs）
上下文保存	仅硬件自动保存 CSA	OS 额外保存 + CSA
嵌套	由硬件优先级决定	OS 管理嵌套
典型用途	PWM 故障保护、硬件看门狗	CAN 接收、定时器

选择策略：

```
/* ===== Cat1 ISR 示例：PWM 紧急关断（响应时间 < 500ns） ===== */
/* Cat1 ISR 不使用 ISR() 宏，直接定义中断函数 */
/* 在 Os_Cfg.h 中配置优先级高于 CFG_ISR2_IPL_MAX_CORE0 */
/* TC334: 优先级 > ISR2 最高优先级 → 自动成为 Cat1 */

void __interrupt(0x30) __enable_ Pwm_FaultISR(void)
{
    /* 直接操作硬件寄存器，不调用任何 OS API */
    GTM_TOM0_CH0_CTRL.B.SL = 0;    /* 强制 PWM 输出低电平 */
```

```
    GTM_TOM0_CH1_CTRL.B.SL = 0;    /* 互补通道也关断 */

    /* 清除中断标志 */
    SRC_GTM_GTM0_TOM0_0.B.CLRR = 1;
}

/* ===== Cat2 ISR 示例：CAN 接收（允许激活处理任务） ===== */
ISR(ISR_CanRx)
{
    Can_MessageType msg;

    /* 从硬件 FIFO 读取报文（尽量快） */
    Can_Hw_ReadMessage(&msg);

    /* 写入软件队列 */
    RingBuffer_Write(&g_canRxQueue, &msg);

    /* 通知处理任务（调用 OS API，Cat2 允许） */
    SetEvent(Task_Communication, EVT_CAN_RX);
}
```

决策流程： 1. 是否需要调用 OS API？ → 是 → Cat2 2. 是否有极端响应时间要求（< 1μs）？ → 是 → Cat1 3. 是否涉及安全关断/硬件保护？ → 是 → Cat1 4. 其他情况 → Cat2（可管理性更好）

18.2.2 ISR 最小化原则

核心原则：ISR 中只做"必须立即做"的事，将复杂处理延迟到任务中执行。

```
/* ===== 优化前：ISR 中做了太多事情 ===== */
ISR(ISR_AdcComplete)  /* ❌ 执行时间过长 */
{
    uint16 rawValue = EVADC_G0_RES0.B.RESULT;   /* 读取 ADC 结果 */
    float voltage = rawValue * 3.3f / 4096.0f;   /* 浮点运算！ */
    float filtered = IIR_Filter(&g_filter, voltage); /* 滤波计算 */

    if (filtered > THRESHOLD_OVERVOLTAGE) {      /* 阈值判断 */
        Fault_SetDTC(DTC_OVERVOLTAGE);           /* DTC 设置（可能访问 NVM） */
        Motor_Shutdown();                         /* 关断电机（复杂操作） */
    }

    g_measuredVoltage = filtered;
}
/* 问题：浮点运算 + 滤波 + NVM + 电机控制 → ISR 执行可能超过 50μs */
/* 后果：阻塞其他中断，导致系统不确定性 */
```

```
/* ===== 优化后：ISR 只做最小必要操作 ===== */
ISR(ISR_AdcComplete)  /* ✅ 执行时间 < 2μs */
{
    /* 仅读取硬件结果并存储 */
    g_adcRawBuffer[g_adcWriteIdx] = EVADC_G0_RES0.B.RESULT;
    g_adcWriteIdx = (g_adcWriteIdx + 1) % ADC_BUFFER_SIZE;

    /* 激活处理任务 */
    ActivateTask(Task_AdcProcess);
}

/* 复杂处理在任务中进行 */
```

```
TASK(Task_AdcProcess)
{
    uint16 rawValue = g_adcRawBuffer[g_adcReadIdx];
    g_adcReadIdx = (g_adcReadIdx + 1) % ADC_BUFFER_SIZE;

    /* 浮点运算、滤波、判断 —— 可被更高优先级任务抢占 */
    float voltage = rawValue * 3.3f / 4096.0f;
    float filtered = IIR_Filter(&g_filter, voltage);

    if (filtered > THRESHOLD_OVERVOLTAGE) {
        Fault_SetDTC(DTC_OVERVOLTAGE);
        SetEvent(Task_Safety, EVT_OVERVOLTAGE);
    }

    g_measuredVoltage = filtered;
    TerminateTask();
}
```

量化标准：- Cat2 ISR 执行时间应 < 10μs（@300MHz 约 3000 条指令以内） - Cat1 ISR 执行时间应 < 2μs（@300MHz 约 600 条指令以内） - ISR 中禁止浮点运算（TC334 浮点单元上下文保存额外消耗 CSA）

18.2.3 中断到任务的响应延迟分析

从硬件中断触发到任务开始执行，延迟由以下部分组成：

```
总延迟 = T_hw + T_isr + T_os_api + T_schedule + T_context_switch
```

各阶段在 TC334 @300MHz 上的典型值：

阶段	描述	典型耗时
T_hw	中断请求→CPU 响应（流水线刷新+CSA 保存）	50~100ns
T_isr	ISR 执行体（按最小化原则）	200ns~2μs
T_os_api	ActivateTask/SetEvent OS 内核处理	300~800ns
T_schedule	调度器选择就绪最高优先级任务	100~300ns
T_context_switch	CSA 恢复目标任务上下文	50~100ns
总计		**0.7~3.3μs**

测量方法：使用 STM 时间戳精确测量

```
/* STM 时间戳测量中断到任务的延迟 */
volatile uint32 g_isrTimestamp = 0;
volatile uint32 g_taskTimestamp = 0;
volatile uint32 g_latencyTicks = 0;

ISR(ISR_Trigger)
{
    /* 进入 ISR 时立即记录 STM 时间戳 */
    g_isrTimestamp = STM0_TIM0.U;  /* STM Timer 0, 100MHz → 1 tick = 10ns */

    ActivateTask(Task_Response);
}

TASK(Task_Response)
{
    /* 任务开始执行时记录时间戳 */
    g_taskTimestamp = STM0_TIM0.U;

    /* 计算延迟（单位：STM ticks, 10ns/tick） */
    g_latencyTicks = g_taskTimestamp - g_isrTimestamp;
```

```
    /* 转换为微秒：latency_us = g_latencyTicks / 100 */
    /* 典型值：70~330 ticks = 0.7~3.3μs */

    TerminateTask();
}
```

18.2.4 中断优先级分配策略

TC334 中断路由器（IR）支持 1~255 优先级（0 表示禁用），数值越大优先级越高。

分配原则：

```
/* ===== 推荐的中断优先级分配方案 ===== */

/* 优先级分层（TC334，共 255 级可用） */
/* ┌─────────────────────────────────────────────┐ */
/* │ 200~255: Cat1 ISR（安全关断、硬件保护）      │ */
/* │ 100~199: 预留（未来扩展）                    │ */
/* │  50~99 : Cat2 ISR - 高优先级（通信接收等）   │ */
/* │  10~49 : Cat2 ISR - 普通优先级（定时器、ADC）│ */
/* │   1~9  : Cat2 ISR - 低优先级（后台采集）     │ */
/* │   0    : 禁用                                │ */
/* └─────────────────────────────────────────────┘ */

/* Os_Cfg.h 中定义 ISR2 最高优先级 */
#define CFG_ISR2_IPL_MAX_CORE0  (99U)
/* 含义：优先级 1~99 为 Cat2，100+ 为 Cat1 */

/* 具体分配示例 */
#define ISR_PRIO_SYSTEM_TIMER    (10U)     /* OS
#define ISR_PRIO_ADC_COMPLETE    (20U)     /* ADC
#define ISR_PRIO_CAN_RX          (50U)     /* CAN
#define ISR_PRIO_CAN_TX          (45U)     /* CAN
#define ISR_PRIO_LIN_RX          (40U)     /* LIN
#define ISR_PRIO_PWM_FAULT       (200U)    /* PWM
#define ISR_PRIO_WDG_TRIGGER     (210U)    /*
```

关键约束： - OS 系统定时器优先级不宜太高（10~20），否则影响通信中断及时性 - Cat2 ISR 之间优先级差距建议 ≥ 5，便于后续插入新中断 - 同一外设的多个中断应相邻分配（如 CAN_RX=50, CAN_TX=45, CAN_ERR=55）

18.3 栈空间优化

TC334（TriCore 架构）的栈管理与 ARM/x86 有本质区别——TriCore 使用 **CSA（Context Save Area）** 保存函数调用上下文，而非传统的栈帧压栈。理解这一差异是正确配置栈空间的前提。

18.3.1 CSA 消耗计算（TriCore 特有）

CSA 机制简述： - TriCore 每次函数调用/中断响应时，硬件自动将当前上下文（Upper/Lower Context）保存到一个 CSA 帧中 - 每个 CSA 帧固定 16 words = 64 bytes - CSA 帧通过链表（PCXI 寄存器）链接，不占用传统栈空间 - 任务栈（SP）仅用于局部变量和编译器临时数据

CSA 消耗计算公式：

```
CSA 消耗 = (最大调用深度 + 中断嵌套层数) × 64 bytes
```

示例计算：

```
/* 假设 Task_Control 的调用链：
 * Task_Control() → Motor_Run() → PID_Calculate() → Math_Sqrt()
 * 最大调用深度 = 4 层
 *
 * 可能被 2 层中断嵌套打断：
 * ISR_Timer (Cat2, 优先级 10) → ISR_CanRx (Cat2, 优先级 50)
 * 中断嵌套 = 2 层
 *
 * CSA 消耗 = (4 + 2) × 64 = 384 bytes
 *
 * 安全裕量（建议 ×1.5）：384 × 1.5 = 576 bytes ≈ 9 个 CSA 帧
 */

/* Os_Cfg.c 中配置 CSA 区域 */
/* TC334 DSPR0 大小为 96KB，预留 CSA 区域示例 */
#define CSA_REGION_SIZE    (4096U)    /* 4KB = 64
/* 对于整个系统（所有任务+ISR 共享 CSA 链表） */
/* 计算：假设最多 5 个任务同时在就绪/运行态，每任务平均 8 帧 */
/* 所需 CSA = 5 × 8 × 64 = 2560 bytes ≈ 3KB */
/* 加上 ISR 嵌套 2 层 × 64 = 128 bytes */
/* 总计约 3KB，配置 4KB 留有裕量 */
```

CSA 与栈的分工：

存储内容	存储位置	大小计算方式
寄存器上下文（A[2]~A[7], D[0]~D[7]等）	CSA 帧	固定 64 bytes/帧
局部变量（非寄存器分配的）	任务栈	取决于变量大小
函数参数（超过寄存器传递的）	任务栈	取决于参数大小
大型局部数组/结构体	任务栈	变量声明大小

18.3.2 Task 栈大小估算方法

方法 1：静态分析（推荐用于确定性评估）

```
/* 栈大小估算公式：
 * Stack_Size = Local_Vars_Max + Compiler_Overhead + Safety_Margin
 *
 * 其中：
 * Local_Vars_Max = 沿最深调用路径累加所有函数的局部变量
 * Compiler_Overhead = 对齐填充 + 临时变量（通常为 Local_Vars_Max 的 20%）
 * Safety_Margin = 30~50% 裕量
 */

/* 示例：Task_MotorCtrl 栈估算 */
/*
 * Motor_MainFunction():
 *   - float pid_output;          4 bytes
 *   - float error, integral;     8 bytes
 *   → 调用 PID_Calculate()
 *
 * PID_Calculate():
 *   - float p_term, i_term, d_term; 12 bytes
 *   - float dt;                      4 bytes
 *   → 调用 Filter_Apply()
 *
 * Filter_Apply():
 *   - float buffer[8];              32 bytes
 *   - uint32 i;                      4 bytes
```

```
 *
 * 最深路径局部变量总和 = 12 + 16 + 36 = 64 bytes
 * 加上编译器开销（20%）: 64 × 1.2 = 77 bytes
 * 加上安全裕量（50%）: 77 × 1.5 = 116 bytes
 * 对齐到 8 字节: 120 bytes
 * 推荐配置: 128 bytes（或 256 bytes 以获得更多裕量）
 */

/* Os_Cfg.c 中的栈配置 */
#define TASK_MOTOR_STACK_SIZE   (256U)    /* 256 bytes */
static uint8 Task_MotorCtrl_Stack[TASK_MOTOR_STACK_SIZE] __attribute__((aligned(8)));
```

方法 2：动态填充测量法（运行时验证）

```
/* 栈填充模式：初始化时用特定 pattern 填充整个栈区域 */
#define STACK_FILL_PATTERN  (0xDEADBEEFU)

void Stack_FillPattern(uint32* stackBase, uint32 stackSize)
{
    uint32 i;
    uint32 words = stackSize / 4;
    for (i = 0; i < words; i++) {
        stackBase[i] = STACK_FILL_PATTERN;
    }
}

/* 运行一段时间后，检查未被覆盖的 pattern 数量 → 得到实际使用峰值 */
uint32 Stack_GetUsage(uint32* stackBase, uint32 stackSize)
{
    uint32 i;
    uint32 words = stackSize / 4;
    uint32 usedWords = 0;

    /* 从栈底（低地址）向上扫描，直到找到第一个非 pattern 值 */
    for (i = 0; i < words; i++) {
        if (stackBase[i] != STACK_FILL_PATTERN) {
            usedWords = words - i;
            break;
        }
    }
    return usedWords * 4;  /* 返回字节数 */
}

/* 在 100ms 后台任务中周期性检查 */
TASK(Task_Monitor)
{
    uint32 motorStackUsage = Stack_GetUsage(
        (uint32*)Task_MotorCtrl_Stack, TASK_MOTOR_STACK_SIZE);

    /* 使用率 > 80% 发出告警 */
    if (motorStackUsage > TASK_MOTOR_STACK_SIZE * 80 / 100) {
        Dem_SetEventStatus(DTC_STACK_NEAR_OVERFLOW, DEM_EVENT_STATUS_FAILED);
    }

    TerminateTask();
}
```

18.3.3 栈监控实践（运行时检测）

OS 内建栈监控：

```
/* Os_Cfg.h 中启用栈监控 */
#define CFG_STACK_CHECK      FALSE     /*
#define CFG_STACK_MONITOR    TRUE      /*

/* 当 CFG_STACK_MONITOR = TRUE 时，OS 在任务切换时自动检测栈指针 */
/* 若检测到溢出，触发 ProtectionHook（需配置 SC3/SC4）或 ErrorHook */
```

自定义栈水印监控（更灵活）：

```
/* 高级栈监控：在 PostTaskHook 中统计 */
#define CFG_POSTTASKHOOK   TRUE

typedef struct {
    uint32 peakUsage;        /* 历史峰值使用量（bytes） */
    uint32 totalSize;        /* 总大小 */
    uint32 warningCount;     /* 告警次数 */
} StackMonitorInfo;

StackMonitorInfo g_stackMonitor[CFG_TASK_MAX];

void PostTaskHook(void)
{
    TaskType currentTask;
    uint32 currentUsage;

    GetTaskID(&currentTask);

    /* 获取当前任务栈使用量（平台相关实现） */
    currentUsage = Os_GetTaskStackUsage(currentTask);

    /* 更新峰值 */
    if (currentUsage > g_stackMonitor[currentTask].peakUsage) {
        g_stackMonitor[currentTask].peakUsage = currentUsage;
    }

    /* 检查是否超过 75% 阈值 */
    if (currentUsage > g_stackMonitor[currentTask].totalSize * 75 / 100) {
        g_stackMonitor[currentTask].warningCount++;
    }
}
```

18.3.4 栈溢出诊断与修复

现象：系统随机崩溃、变量莫名被修改、Hard Trap（TriCore Class 6 trap）。

诊断方法：

```
/* 诊断步骤 1：检查 CSA 链表完整性 */
/* TriCore: FCX (Free CSA List Head) 寄存器 */
/* 如果 FCX == 0，表示 CSA 耗尽 → Context Depletion Trap */

uint32 Os_CountFreeCsa(void)
{
    uint32 count = 0;
    uint32 fcx = __mfcr(CPU_FCX);  /* 读取 Free CSA Head */

    while (fcx != 0 && count < 1000) {  /* 防止死循环 */
        /* 将 CSA 链接字转换为物理地址 */
        uint32* csaAddr = (uint32*)((((fcx >> 16) & 0xF) << 28) |
```

```
                                      ((fcx & 0xFFFF) << 6));
        fcx = csaAddr[0];  /* 第一个 word 是 PCXI（指向下一个空闲帧） */
        count++;
    }
    return count;
}

/* 在 ErrorHook 中输出诊断信息 */
void ErrorHook(StatusType error)
{
    if (error == E_OS_STACKFAULT) {
        TaskType faultTask;
        GetTaskID(&faultTask);

        /* 记录故障信息 */
        g_diagInfo.faultTaskId = faultTask;
        g_diagInfo.freeCsaCount = Os_CountFreeCsa();
        g_diagInfo.timestamp = STM0_TIM0.U;

        /* 安全关断 */
        ShutdownOS(error);
    }
}
```

修复方案优先级： 1. 增大栈配置：最直接，但可能受 SRAM 限制（TC334 DSPR 仅 96KB） 2. 减少调用深度：内联小函数（`__inline`），展平调用链 3. 减少局部变量：大数组改为静态分配或全局变量 4. 减少中断嵌套：降低 `CFG_ISR2_IPL_MAX` 限制嵌套层数 5. 优化 CSA 区域：确保系统 CSA 池足够大

18.4 CPU 负载分析

CPU 负载是衡量实时系统健康度的核心指标。目标是在满足所有实时约束的前提下，保持足够的空闲时间应对突发事件。

18.4.1 执行时间测量方法（STM 时间戳）

TC334 的 STM（System Timer）提供 100MHz 的自由运行计数器，分辨率为 **10ns**，是精确测量执行时间的理想工具。

```
/* ===== STM 时间戳测量框架 ===== */

/* STM 配置说明:
 * TC334 STM0 时钟 = fSPB = 100MHz
 * 1 tick = 10ns
 * 32 位计数器范围: 0 ~ 4,294,967,295 = 约 42.9 秒溢出
 * 对于单次测量（< 42s），无需处理溢出
 */

typedef struct {
    uint32 startTick;        /* 开始时间戳 */
    uint32 endTick;          /* 结束时间戳 */
    uint32 execTicks;        /* 本次执行 tick 数 */
    uint32 maxExecTicks;     /* 历史最大值（WCET 近似） */
    uint32 minExecTicks;     /* 历史最小值（BCET） */
    uint32 avgAccumulator;   /* 累加器（用于计算平均值） */
    uint32 sampleCount;      /* 采样次数 */
} ExecTimeMeasure;
```

```
/* 为每个需要监控的任务创建测量结构 */
ExecTimeMeasure g_taskExecTime[CFG_TASK_MAX];

/* 在 PreTaskHook 中记录开始时间 */
#define CFG_PRETASKHOOK  TRUE

void PreTaskHook(void)
{
    TaskType taskId;
    GetTaskID(&taskId);
    g_taskExecTime[taskId].startTick = STM0_TIM0.U;
}

/* 在 PostTaskHook 中记录结束时间并统计 */
void PostTaskHook(void)
{
    TaskType taskId;
    uint32 elapsed;

    GetTaskID(&taskId);
    g_taskExecTime[taskId].endTick = STM0_TIM0.U;

    /* 计算执行时间（自动处理 32 位溢出） */
    elapsed = g_taskExecTime[taskId].endTick - g_taskExecTime[taskId].startTick;
    g_taskExecTime[taskId].execTicks = elapsed;

    /* 更新最大值 */
    if (elapsed > g_taskExecTime[taskId].maxExecTicks) {
        g_taskExecTime[taskId].maxExecTicks = elapsed;
    }

    /* 更新最小值 */
    if (elapsed < g_taskExecTime[taskId].minExecTicks ||
        g_taskExecTime[taskId].minExecTicks == 0) {
        g_taskExecTime[taskId].minExecTicks = elapsed;
    }

    /* 累加（用于滑动平均） */
    g_taskExecTime[taskId].avgAccumulator += elapsed;
    g_taskExecTime[taskId].sampleCount++;
}

/* 辅助函数：tick 转微秒 */
static inline float ExecTime_TicksToUs(uint32 ticks)
{
    return (float)ticks / 100.0f;  /* 100MHz → 100 ticks/μs */
}

/* 使用示例：读取 Task_10ms 的 WCET */
/* float wcet_us = ExecTime_TicksToUs(g_taskExecTime[Task_10ms].maxExecTicks); */
```

注意事项： - PreTaskHook/PostTaskHook 本身有执行开销（约 0.5~1μs），测量值包含此开销 - 生产版本应禁用 Hook（`CFG_PRETASKHOOK = FALSE`），仅在调试阶段使用 - 如需更精确测量，可在任务函数内部首尾直接读取 STM

18.4.2 Worst-Case 执行时间分析

WCET（Worst-Case Execution Time）分析对于证明系统满足实时约束至关重要。

方法对比：

方法	精度	工作量	适用阶段
静态分析（工具）	高（保守）	低	设计阶段
测量法（长时间运行）	中（统计）	中	集成测试
混合法（测量+安全裕量）	高（实用）	中	产品验证

实用的 WCET 评估方法（测量 + 裕量）：

```
/* WCET 评估策略：
 * 1. 长时间运行（>24h）收集 maxExecTicks
 * 2. 乘以安全裕量（通常 1.2~1.5）得到 WCET 估计值
 * 3. 验证：WCET < Deadline（周期任务的 deadline 通常等于周期）
 */

typedef struct {
    uint32 measuredWcet;    /* 测量到的最大执行时间（ticks） */
    float  safetyMargin;    /* 安全裕量因子 */
    uint32 estimatedWcet;   /* 估计 WCET = measuredWcet × safetyMargin */
    uint32 deadline;        /* 截止时间（ticks） */
    boolean deadlineMet;    /* 是否满足约束 */
} WcetAnalysis;

WcetAnalysis g_wcetTable[] = {
    /* Task_1ms:  measured=80μs, margin=1.3, deadline=1000μs(1ms) */
    { .measuredWcet = 8000, .safetyMargin = 1.3f,
      .estimatedWcet = 10400, .deadline = 100000, .deadlineMet = TRUE },

    /* Task_5ms:  measured=200μs, margin=1.3, deadline=5000μs(5ms) */
    { .measuredWcet = 20000, .safetyMargin = 1.3f,
      .estimatedWcet = 26000, .deadline = 500000, .deadlineMet = TRUE },

    /* Task_10ms: measured=500μs, margin=1.3, deadline=10000μs(10ms) */
    { .measuredWcet = 50000, .safetyMargin = 1.3f,
      .estimatedWcet = 65000, .deadline = 1000000, .deadlineMet = TRUE },
};

/* 运行时 Deadline 监控 */
void Task_CheckDeadline(TaskType taskId, uint32 execTicks)
{
    if (execTicks > g_wcetTable[taskId].deadline) {
        /* Deadline Miss! 严重问题 */
        g_wcetTable[taskId].deadlineMet = FALSE;

        /* 记录到诊断日志 */
        Diag_LogDeadlineMiss(taskId, execTicks,
                             g_wcetTable[taskId].deadline);

        /* 触发 ErrorHook（如果配置） */
        /* 注意：AUTOSAR OS 标准无直接 Deadline 监控 API，
         * 需在 Timing Protection (SC2/SC4) 或自定义 Hook 中实现 */
    }
}
```

18.4.3 空闲率监测（Idle Task 利用率）

CPU 负载率 = 1 - 空闲率。通过测量 Idle Task 执行时间占比，可精确得到系统负载。

```
/* ===== CPU 负载率计算方法 ===== */

/* 原理：在固定测量窗口内，统计 Idle Task 的累计执行时间 */
```

```c
/* CPU 负载率 = 1 - (Idle 执行时间 / 测量窗口时间) */

typedef struct {
    uint32 measureWindowTicks;   /* 测量窗口（如 100ms = 10,000,000 ticks） */
    uint32 idleAccumTicks;       /* 当前窗口内 Idle 累计执行时间 */
    uint32 windowStartTick;      /* 窗口起始时间 */
    uint32 lastIdleEntry;        /* Idle 最近一次进入时间 */
    uint8  cpuLoadPercent;       /* 计算结果：CPU 负载率 (0~100) */
    uint8  peakLoadPercent;      /* 历史峰值负载 */
} CpuLoadMonitor;

CpuLoadMonitor g_cpuLoad = {
    .measureWindowTicks = 10000000U,  /* 100ms 窗口，100MHz STM */
    .idleAccumTicks = 0,
    .cpuLoadPercent = 0,
    .peakLoadPercent = 0,
};

/* 在 Idle Task 中周期性计算 */
TASK(OS_TASK_IDLE_CORE0)
{
    while (1) {
        uint32 now = STM0_TIM0.U;

        /* 记录进入 Idle 时间 */
        g_cpuLoad.lastIdleEntry = now;

        /* 检查是否到达测量窗口边界 */
        if ((now - g_cpuLoad.windowStartTick) >= g_cpuLoad.measureWindowTicks) {
            /* 计算本窗口的 CPU 负载率 */
            uint32 idleRatio = (g_cpuLoad.idleAccumTicks * 100)
                               / g_cpuLoad.measureWindowTicks;
            g_cpuLoad.cpuLoadPercent = (uint8)(100 - idleRatio);

            /* 更新峰值 */
            if (g_cpuLoad.cpuLoadPercent > g_cpuLoad.peakLoadPercent) {
                g_cpuLoad.peakLoadPercent = g_cpuLoad.cpuLoadPercent;
            }

            /* 重置窗口 */
            g_cpuLoad.windowStartTick = now;
            g_cpuLoad.idleAccumTicks = 0;
        }

        /* Idle 空转（或 WFI 低功耗等待） */
        __nop();

        /* 被抢占后恢复时，累加本次 Idle 执行时间 */
        g_cpuLoad.idleAccumTicks += (STM0_TIM0.U - g_cpuLoad.lastIdleEntry);
    }

    TerminateTask();  /* 实际不会执行到这里 */
}
```

负载率判断标准：

CPU 负载率	**系统状态**	**建议措施**
0~40%	轻载	正常运行，有充足裕量
40~60%	中载	正常运行，可接受新功能

60~70%	较重	注意监控，谨慎新增功能
70~85%	重载	**需要优化**，突发事件可能导致 Deadline Miss
85~100%	过载	**严重问题**，系统不稳定

18.4.4 负载超标的优化手段

当 CPU 负载超过 70% 时，按以下优先级依次尝试优化：

```
/* ===== 优化手段 1：降低任务执行频率 ===== */

/* 优化前：100ms 诊断任务执行了不必要的高频操作 */
TASK(Task_Diag_100ms)  /* ✕ 每 100ms 全量扫描 */
{
    uint32 i;
    for (i = 0; i < 64; i++) {
        Dtc_Check(i);  /* 检查所有 64 个 DTC */
    }
    TerminateTask();
}

/* 优化后：分散到多个周期执行 */
static uint32 g_dtcScanIndex = 0;
TASK(Task_Diag_100ms)  /* ✓ 每 100ms 只检查 8 个 DTC */
{
    uint32 i;
    for (i = 0; i < 8; i++) {
        Dtc_Check(g_dtcScanIndex);
        g_dtcScanIndex = (g_dtcScanIndex + 1) % 64;
    }
    TerminateTask();
}
/* 效果：执行时间减少约 87.5%，完整扫描周期变为 800ms */
```

```
/* ===== 优化手段 2：算法优化（查表替代计算） ===== */

/* 优化前：实时计算三角函数（约 50μs @300MHz） */
float Motor_CalcAngle(uint16 encoderCount)
{
    float angle = (float)encoderCount * 2.0f * 3.14159f / 4096.0f;
    return sinf(angle);  /* 软浮点 sin 计算，非常耗时 */
}

/* 优化后：使用查找表（约 0.5μs @300MHz） */
/* 预计算 sin 表，256 点，Q15 定点格式 */
static const int16 g_sinTable[256] = {
    0, 804, 1608, 2410, 3212, 4011, 4808, 5602,
    /* ... 完整 256 点表格 ... */
};

int16 Motor_CalcAngle_Fast(uint16 encoderCount)
{
    /* 将 12bit 编码器值映射到 8bit 表索引 */
    uint8 index = (uint8)(encoderCount >> 4);
    return g_sinTable[index];
}
/* 效果：执行时间从 50μs 降到 0.5μs，速度提升 100 倍 */
```

```
/* ===== 优化手段 3：减少不必要的 OS API 调用 ===== */

/* 优化前：频繁获取/释放资源 */
```

```c
TASK(Task_Comm_5ms)  /* ✗ 每次循环都获取资源 */
{
    uint32 i;
    for (i = 0; i < msgCount; i++) {
        GetResource(RES_SharedData);
        ProcessMessage(i);
        ReleaseResource(RES_SharedData);
    }
    TerminateTask();
}

/* 优化后：批量处理，减少 API 调用次数 */
TASK(Task_Comm_5ms)  /* ✓ 一次获取，批量处理 */
{
    uint32 i;
    GetResource(RES_SharedData);
    for (i = 0; i < msgCount; i++) {
        ProcessMessage(i);
    }
    ReleaseResource(RES_SharedData);
    TerminateTask();
}
/* 效果：GetResource/ReleaseResource 各调用 1 次而非 N 次 */
/* 每次 OS API 调用约 0.3~0.5μs，N=10 时节省约 5~9μs */
```

18.5 资源争用优化

当多个任务访问共享资源时，AUTOSAR OS 通过优先级天花板协议（Priority Ceiling Protocol）避免死锁，但不当使用仍会导致严重的优先级反转和响应延迟。

18.5.1 减少资源持有时间

核心原则：资源持有时间越短，对其他任务的阻塞影响越小。

```c
/* ===== 优化前：在资源保护区内做了不必要的操作 ===== */
TASK(Task_DataLog)  /* ✗ 资源持有时间过长 */
{
    GetResource(RES_SharedBuffer);

    /* 以下操作在资源保护区内执行 */
    uint32 rawData = Adc_Read();          /* ADC 读取（可能等待转换） */
    float scaled = rawData * 0.00122f;    /* 缩放计算 */
    float filtered = LowPass(&g_filter, scaled);  /* 滤波 */
    g_sharedBuffer[g_writeIdx] = filtered;         /* 写入共享缓冲 */
    g_writeIdx++;

    ReleaseResource(RES_SharedBuffer);
    TerminateTask();
}
/* 问题：ADC 读取 + 浮点计算都在临界区内，持有时间可能 > 20μs */
```

```c
/* ===== 优化后：仅保护必要的共享数据访问 ===== */
TASK(Task_DataLog)  /* ✓ 资源持有时间最小化 */
{
    /* 在资源保护区外完成所有计算 */
    uint32 rawData = Adc_Read();
    float scaled = rawData * 0.00122f;
    float filtered = LowPass(&g_filter, scaled);
```

```
    /* 仅在写入共享数据时持有资源 */
    GetResource(RES_SharedBuffer);
    g_sharedBuffer[g_writeIdx] = filtered;   /* 单次赋值，约 10ns */
    g_writeIdx++;
    ReleaseResource(RES_SharedBuffer);

    TerminateTask();
}
/* 效果：资源持有时间从 20+μs 降到 < 0.1μs */
```

18.5.2 避免优先级反转

AUTOSAR OS 使用 OSEK 优先级天花板协议（Immediate Priority Ceiling），在 GetResource 时立即提升任务优先级到资源天花板值。

配置要点：

```
/* 资源天花板优先级 = 使用该资源的所有任务中的最高优先级 */

/* 示例：RES_SharedData 被 Task_A(pri=3) 和 Task_B(pri=5) 使用 */
/* 天花板优先级 = 5 */
/* 当 Task_A 获取资源时，其运行优先级临时提升到 5 */
/* 这防止了 Task_B 抢占 Task_A 时的优先级反转 */

/* Os_Cfg.c 中的资源配置 */
const Os_ResourceCfgType Os_ResourceCfg[CFG_RESOURCE_MAX] = {
    {
        .ceilingPriority = 5U,  /* 天花板 = max(Task_A.pri, Task_B.pri) */
        .coreId = OS_CORE_ID_0,
    },
};

/* 反模式：天花板优先级设置过高 */
/* ✕ 如果将天花板设为系统最高优先级，则任何任务持有资源时都不可被抢占 */
/* 这等效于全局关中断，破坏实时性 */
```

优先级反转诊断：

```
/* 检测优先级反转的自定义监控 */
typedef struct {
    TaskType  holdingTask;      /* 当前持有资源的任务 */
    uint32    acquireTimestamp; /* 获取时间 */
    uint32    maxHoldTicks;    /* 最长持有时间 */
} ResourceMonitor;

ResourceMonitor g_resMonitor[CFG_RESOURCE_MAX];

/* 如果一个高优先级任务等待资源的时间超过阈值，说明存在问题 */
#define RESOURCE_HOLD_THRESHOLD_TICKS  (10000U)    /* 100

void Resource_CheckHoldTime(ResourceType resId)
{
    uint32 holdTime = STM0_TIM0.U - g_resMonitor[resId].acquireTimestamp;
    if (holdTime > RESOURCE_HOLD_THRESHOLD_TICKS) {
        /* 资源持有时间超标！可能导致优先级反转效应 */
        Diag_LogResourceOverhold(resId,
                                 g_resMonitor[resId].holdingTask,
                                 holdTime);
```

```
    }
    if (holdTime > g_resMonitor[resId].maxHoldTicks) {
        g_resMonitor[resId].maxHoldTicks = holdTime;
    }
}
```

18.5.3 Spinlock 粒度控制（多核）

在多核 AUTOSAR OS 系统中（如 TC3xx 多核变体），跨核资源保护使用 Spinlock。

```
/* ===== Spinlock 使用原则 ===== */

/* 粒度过粗：✕ 一个 Spinlock 保护所有共享数据 */
void Core0_WriteData(void)
{
    GetSpinlock(SPINLOCK_GLOBAL);    /* 所有跨核操作共用一把锁 */
    g_motorData = newMotorData;
    g_sensorData = newSensorData;
    g_commData = newCommData;
    ReleaseSpinlock(SPINLOCK_GLOBAL);
}
/* 问题：Core1 任何跨核操作都被阻塞，即使访问的是不同数据 */

/* 粒度适当：✓ 按功能域划分 Spinlock */
void Core0_WriteMotorData(void)
{
    GetSpinlock(SPINLOCK_MOTOR);     /* 仅保护电机相关数据 */
    g_motorData = newMotorData;
    ReleaseSpinlock(SPINLOCK_MOTOR);
}

void Core0_WriteSensorData(void)
{
    GetSpinlock(SPINLOCK_SENSOR);    /* 仅保护传感器数据 */
    g_sensorData = newSensorData;
    ReleaseSpinlock(SPINLOCK_SENSOR);
}
/* 优势：不同功能域可并行访问，减少核间等待 */

/* 注意：Spinlock 在等待时 CPU 空转（busy-wait），绝不能长时间持有 */
/* 经验值：Spinlock 持有时间应 < 1μs（约 300 条指令 @300MHz） */
```

18.5.4 无锁设计模式

对于简单的生产者-消费者场景，可使用无锁环形缓冲区避免资源争用。

```
/* ===== 无锁环形缓冲区（单生产者-单消费者） ===== */

#define RING_BUFFER_SIZE  16    /*
#define RING_BUFFER_MASK  (RING_BUFFER_SIZE - 1)

typedef struct {
    volatile uint32 writeIdx;    /*
    volatile uint32 readIdx;     /*
    uint32 data[RING_BUFFER_SIZE];
} LockFreeRingBuffer;

LockFreeRingBuffer g_adcBuffer = { .writeIdx = 0, .readIdx = 0 };

/* 生产者（ISR 中调用，无需获取资源） */
```

```
static inline boolean RingBuffer_Write(LockFreeRingBuffer* rb, uint32 value)
{
    uint32 nextWrite = (rb->writeIdx + 1) & RING_BUFFER_MASK;
    if (nextWrite == rb->readIdx) {
        return FALSE;  /* 缓冲区满 */
    }
    rb->data[rb->writeIdx] = value;
    /* 内存屏障确保数据写入在索引更新之前完成 */
    __dsync();  /* TriCore 数据同步指令 */
    rb->writeIdx = nextWrite;
    return TRUE;
}

/* 消费者（Task 中调用，无需获取资源） */
static inline boolean RingBuffer_Read(LockFreeRingBuffer* rb, uint32* value)
{
    if (rb->readIdx == rb->writeIdx) {
        return FALSE;  /* 缓冲区空 */
    }
    *value = rb->data[rb->readIdx];
    __dsync();
    rb->readIdx = (rb->readIdx + 1) & RING_BUFFER_MASK;
    return TRUE;
}

/* 使用场景：ISR 采集 ADC 数据，Task 处理 */
ISR(ISR_AdcComplete)
{
    uint32 result = EVADC_G0_RES0.B.RESULT;
    RingBuffer_Write(&g_adcBuffer, result);  /* 无需 GetResource */
}

TASK(Task_AdcProcess)
{
    uint32 value;
    while (RingBuffer_Read(&g_adcBuffer, &value)) {
        Process_AdcValue(value);  /* 无需 GetResource */
    }
    TerminateTask();
}
```

无锁设计的适用条件： - 仅一个写入者和一个读取者（SPSC） - 数据元素为原子写入大小（≤ 32 位对齐访问在 TriCore 上原子） - 64 位数据或结构体需要额外同步机制

18.6 内存布局优化（TC334 特有）

TC334 拥有多级存储层次，不同内存区域的访问速度和特性各异。合理利用内存布局可显著提升性能。

18.6.1 DSPR / DLMU / LMU 内存区域特性

内存区域	大小	访问延迟	特性	典型用途
DSPR0	192 KB	0 wait states	CPU0 本地，最快	任务栈、关键变量、CSA
DLMU0	32 KB	0~1 wait states	本核低延迟	查找表、常用缓冲
LMU (SRAM)	32 KB	1~2 wait states	全局共享	核间通信数据
PFlash	2 MB	0~3 wait states	只读，有预取缓冲	代码、常量表
DFlash	128 KB	3~6 wait states	可擦写，最慢	NVM 数据、标定参数

访问速度实测对比（TC334 @300MHz，连续读取 1KB 数据）：

区域	读取耗时	相对速度
DSPR	3.4 µs	1.0x（基准）
DLMU	4.1 µs	0.83x
LMU	5.7 µs	0.60x
PFlash (cache hit)	3.4 µs	1.0x
PFlash (cache miss)	10.2 µs	0.33x

18.6.2 关键数据放置策略

```
/* ===== 利用 GCC section 属性将关键数据放入 DSPR ===== */

/* 策略 1：高频访问的全局变量放入 DSPR */
/* TC334 默认 .bss/.data 通常已在 DSPR，但需确认链接脚本 */

/* 关键控制变量：强制放入 DSPR */
__attribute__((section(".data_dspr")))
volatile float g_motorCurrent = 0.0f;     /* 1ms 周期读写 */

__attribute__((section(".data_dspr")))
volatile uint16 g_encoderPosition = 0;    /* 每次 ISR 更新 */

/* 策略 2：大型查找表放入 DLMU（DSPR 空间宝贵） */
__attribute__((section(".rodata_dlmu")))
static const int16 g_sinTable[1024] = { /* ... */ };

__attribute__((section(".rodata_dlmu")))
static const uint16 g_tempCurve[256] = { /* NTC 温度-阻值曲线 */ };

/* 策略 3：核间共享数据放入 LMU */
__attribute__((section(".data_lmu")))
volatile SharedDataType g_interCoreData;    /* 多核通信用 */

/* 策略 4：非关键大缓冲放入 LMU（节省 DSPR 空间） */
__attribute__((section(".bss_lmu")))
static uint8 g_diagBuffer[2048];           /* 诊断缓冲，非实时 */
```

18.6.3 代码段与数据段分离

原则：热代码（高频执行的函数）放入 PFlash 的连续区域，利用预取缓冲（Prefetch Buffer）加速。

```
/* ===== 热代码标记 ===== */

/* 将高频执行的函数放入特定 section，确保物理地址连续 */
__attribute__((section(".text_hot")))
void Motor_FocControl(void)
{
    /* FOC 算法核心，1ms 执行一次 */
    /* 连续放置可减少 PFlash 预取缓冲 miss */
}

__attribute__((section(".text_hot")))
void PID_Calculate(float setpoint, float feedback, PidState* state)
{
    /* PID 计算，被 Motor_FocControl 调用 */
}

/* 冷代码（低频执行）放入普通 section */
```

```c
__attribute__((section(".text_cold")))
void Diag_FullScan(void)
{
    /* 诊断全量扫描，1s 执行一次 */
}

/* 效果：热代码连续放置后，PFlash 预取命中率从 ~85% 提升到 ~98% */
/* 典型改善：1ms 任务执行时间减少 5~15% */
```

18.6.4 链接脚本配置示例

```
/* ===== Lcf_Gnuc_Tricore_Tc.lsl 链接脚本片段 ===== */
/* GCC TriCore 链接脚本配置内存区域 */

MEMORY
{
    /* Program Flash */
    pfls0 (rx):  org = 0x80000000, len = 1M

    /* Data SRAM (DSPR0) - CPU0 本地高速 SRAM */
    dsram0 (w!x): org = 0x70000000, len = 96K

    /* DLMU - 本地数据内存 */
    dlmu0 (w!x):  org = 0x90000000, len = 32K

    /* LMU SRAM - 全局共享 */
    lmu_sram (w!x): org = 0x90100000, len = 32K
}

SECTIONS
{
    /* 热代码段：放在 PFlash 起始位置，最大化预取效率 */
    .text_hot : { *(.text_hot) } > pfls0

    /* 普通代码段 */
    .text : { *(.text*) } > pfls0

    /* DSPR 数据段：关键变量和任务栈 */
    .data_dspr : { *(.data_dspr) } > dsram0
    .bss_dspr  : { *(.bss_dspr) } > dsram0

    /* CSA 区域：固定放在 DSPR */
    .csa : ALIGN(64) { *(.csa) } > dsram0

    /* 任务栈：放在 DSPR */
    .stacks : ALIGN(8) { *(.stack*) } > dsram0

    /* DLMU 段：查找表等 */
    .rodata_dlmu : { *(.rodata_dlmu) } > dlmu0
    .data_dlmu   : { *(.data_dlmu) } > dlmu0

    /* LMU 段：共享数据和大缓冲 */
    .data_lmu : { *(.data_lmu) } > lmu_sram
    .bss_lmu  : { *(.bss_lmu) } > lmu_sram
}
```

内存分配决策树：

```
数据访问频率高（每 ms 级）？
├ 是 → 数据大小 < 1KB？
│        ├ 是 → 放入 DSPR（最快）
│        └ 否 → 放入 DLMU（较快，空间较大）
└ 否 → 需要跨核共享？
         ├ 是 → 放入 LMU
         └ 否 → 放入 DLMU 或 LMU（按空间需求选择）
```

18.7 常见陷阱与规避

18.7.1 死锁场景与预防

AUTOSAR OS 保证：由于使用优先级天花板协议，单核系统中使用 Resource 不会死锁。但以下场景仍可能出现类似死锁的问题：

场景 1：多核 Spinlock 循环等待

```
/* ✕ 死锁场景：两个核以相反顺序获取 Spinlock */

/* Core 0 任务 */
TASK(Task_Core0)
{
    GetSpinlock(SPINLOCK_A);      /* 先获取 A */
    /* ... 此时 Core1 持有 B，等待 A ... */
    GetSpinlock(SPINLOCK_B);      /* 等待 B → 死锁！ */
    /* ... */
    ReleaseSpinlock(SPINLOCK_B);
    ReleaseSpinlock(SPINLOCK_A);
    TerminateTask();
}

/* Core 1 任务 */
TASK(Task_Core1)
{
    GetSpinlock(SPINLOCK_B);      /* 先获取 B */
    /* ... 此时 Core0 持有 A，等待 B ... */
    GetSpinlock(SPINLOCK_A);      /* 等待 A → 死锁！ */
    /* ... */
    ReleaseSpinlock(SPINLOCK_A);
    ReleaseSpinlock(SPINLOCK_B);
    TerminateTask();
}
```

```
/* ✓ 预防：统一 Spinlock 获取顺序 */

/* 规则：始终按 ID 从小到大获取 */
/* SPINLOCK_A (ID=0) → SPINLOCK_B (ID=1) → SPINLOCK_C (ID=2) */

/* Core 0 & Core 1 都按相同顺序 */
TASK(Task_Core0_Fixed)
{
    GetSpinlock(SPINLOCK_A);      /* 先 A */
    GetSpinlock(SPINLOCK_B);      /* 再 B */
    /* ... 安全操作 ... */
    ReleaseSpinlock(SPINLOCK_B); /* 释放顺序相反 */
    ReleaseSpinlock(SPINLOCK_A);
    TerminateTask();
}
```

场景 2：Event 等待循环依赖

```
/* ✕ 逻辑死锁：Task_A 等 Task_B 的事件，Task_B 等 Task_A 的事件 */

TASK(Task_A)  /* Extended Task */
{
    WaitEvent(EVT_FROM_B);    /* 等待 B 发来的事件 */
    ClearEvent(EVT_FROM_B);
    /* ... 处理 ... */
    SetEvent(Task_B, EVT_FROM_A);  /* 给 B 发事件 */
    TerminateTask();
}

TASK(Task_B)  /* Extended Task */
{
    WaitEvent(EVT_FROM_A);    /* 等待 A 发来的事件 */
    ClearEvent(EVT_FROM_A);
    /* ... 处理 ... */
    SetEvent(Task_A, EVT_FROM_B);  /* 给 A 发事件 */
    TerminateTask();
}
/* 如果两个任务同时进入 WaitEvent → 永远无法唤醒对方 */
```

```
/* ✓ 修复：引入初始触发者或使用 Alarm 打破循环 */
void StartupHook(void)
{
    /* 系统启动时，主动触发 Task_A 开始工作 */
    SetEvent(Task_A, EVT_FROM_B);  /* 初始种子事件 */
}
```

18.7.2 优先级反转诊断

现象：高优先级任务的响应时间偶尔异常增大（比正常值大几倍）。

根因：低优先级任务持有高优先级任务需要的资源，而中间优先级任务抢占了低优先级任务，间接延迟了高优先级任务。

注意：AUTOSAR OS 的优先级天花板协议已经从理论上避免了经典优先级反转。如果仍然观察到类似现象，原因通常是：

1. 资源天花板配置错误（天花板优先级低于实际使用者的优先级） 2. 使用了关中断（SuspendAllInterrupts）替代 Resource，但持有时间过长 3. 多核场景下 Spinlock 等待

诊断代码：

```
/* 响应时间抖动检测 */
typedef struct {
    uint32 activateTimestamp;    /* 任务被激活的时间 */
    uint32 startTimestamp;       /* 任务开始执行的时间 */
    uint32 responseTime;         /* 响应时间 = start - activate */
    uint32 maxResponseTime;      /* 历史最大响应时间 */
    uint32 expectedMaxResponse;  /* 期望的最大响应时间 */
} ResponseTimeMonitor;

ResponseTimeMonitor g_respMonitor[CFG_TASK_MAX];

/* 在 ActivateTask 前记录激活时间（需包装 API） */
StatusType ActivateTask_Monitored(TaskType taskId)
```

```c
{
    g_respMonitor[taskId].activateTimestamp = STM0_TIM0.U;
    return ActivateTask(taskId);
}

/* 在 PreTaskHook 中计算响应时间 */
void PreTaskHook(void)
{
    TaskType taskId;
    uint32 responseTime;

    GetTaskID(&taskId);
    responseTime = STM0_TIM0.U - g_respMonitor[taskId].activateTimestamp;
    g_respMonitor[taskId].responseTime = responseTime;

    if (responseTime > g_respMonitor[taskId].maxResponseTime) {
        g_respMonitor[taskId].maxResponseTime = responseTime;
    }

    /* 检查是否超出期望值（可能存在优先级反转） */
    if (responseTime > g_respMonitor[taskId].expectedMaxResponse) {
        Diag_LogPriorityInversion(taskId, responseTime);
    }
}
```

18.7.3 栈溢出的隐蔽表现

表现 1：变量被莫名修改（相邻内存被栈溢出覆盖）

```c
/* 场景：Task_A 的栈向下增长，溢出后覆盖了相邻的全局变量 */
/*
 * 内存布局：
 * 0x70001000 ┬─ Task_A_Stack[256]  (栈从高地址向低地址增长)
 *            │   ...
 * 0x70000F00 ┼─ Task_A 栈底
 * 0x70000EFC ┼─ g_importantFlag    ← 被溢出覆盖！
 * 0x70000EF8 ┼─ g_sensorCalibration ← 被溢出覆盖！
 */

/* 诊断：在可疑变量前后放置金丝雀值（Canary） */
volatile uint32 g_canary_before = 0xCAFEBABE;
volatile uint32 g_importantFlag = 0;
volatile uint32 g_canary_after = 0xDEADC0DE;

/* 周期性检查 */
void Stack_CheckCanaries(void)
{
    if (g_canary_before != 0xCAFEBABE || g_canary_after != 0xDEADC0DE) {
        /* 栈溢出已发生！ */
        Dem_SetEventStatus(DTC_STACK_CORRUPTION, DEM_EVENT_STATUS_FAILED);
    }
}
```

表现 2：Hard Trap（Class 6: Context Management Error）

```c
/* TriCore 特有：CSA 链表耗尽时触发 Trap 6 (Context Depletion) */
/* 原因：函数调用深度超出 CSA 池容量 */

/* 在 Trap 向量中添加诊断 */
void __trap(6) TrapClass6_Handler(void)
```

```
{
    /* TIN (Trap Identification Number) 区分具体原因 */
    uint32 tin = __mfcr(CPU_D15);  /* 读取 D[15] 获取 TIN */

    switch (tin) {
        case 1:  /* FCU: Free Context Unavailable */
            /* CSA 帧耗尽 → 增大 CSA 区域或减少调用深度 */
            break;
        case 2:  /* CDO: Context Depletion Overflow */
            /* 嵌套层数过深 */
            break;
    }

    /* 记录故障并安全关断 */
    g_trapInfo.trapClass = 6;
    g_trapInfo.tin = tin;
    g_trapInfo.pcxi = __mfcr(CPU_PCXI);
    ShutdownOS(E_OS_STACKFAULT);
}
```

表现 3：函数返回地址被破坏（系统跑飞到随机地址）

```
/* 诊断辅助：在 ErrorHook 中保存现场 */
void ErrorHook(StatusType error)
{
    /* 保存关键寄存器用于事后分析 */
    g_errorDiag.pc = __mfcr(CPU_PC);        /* 程序计数器 */
    g_errorDiag.psw = __mfcr(CPU_PSW);      /* 程序状态字 */
    g_errorDiag.pcxi = __mfcr(CPU_PCXI);    /* CSA 链接字 */
    g_errorDiag.sp = __mfcr(CPU_SP);        /* 栈指针 */
    g_errorDiag.errorCode = error;
    g_errorDiag.timestamp = STM0_TIM0.U;

    /* 可通过调试器查看 g_errorDiag 定位问题 */
}
```

18.7.4 事件丢失与竞态条件

场景：事件在 ClearEvent 和 WaitEvent 之间被设置，导致丢失

```
/* ✕ 事件丢失场景 */
TASK(Task_Handler)
{
    while (1) {
        WaitEvent(EVT_DATA_READY);
        ClearEvent(EVT_DATA_READY);
        Process_Data();
        /* ← 如果此时 ISR 再次 SetEvent(EVT_DATA_READY)... */
        /* ← 下次 WaitEvent 会立即返回，但之间可能丢失事件 */
        /* 注意：AUTOSAR OS 事件是二值信号量语义，不会累计计数 */
    }
    TerminateTask();
}
```

```
/* ✓ 修复方案 1：使用计数器配合事件 */
volatile uint32 g_dataReadyCount = 0;  /* ISR 中递增 */
volatile uint32 g_dataProcessedCount = 0;  /* Task 中递增 */

ISR(ISR_DataReady)
```

```
{
    g_dataReadyCount++;
    SetEvent(Task_Handler, EVT_DATA_READY);
}

TASK(Task_Handler)
{
    while (1) {
        WaitEvent(EVT_DATA_READY);
        ClearEvent(EVT_DATA_READY);

        /* 处理所有未处理的数据（弥补事件不计数的缺陷） */
        while (g_dataProcessedCount < g_dataReadyCount) {
            Process_Data();
            g_dataProcessedCount++;
        }
    }
    TerminateTask();
}
```

```
/* ✅ 修复方案 2：使用环形缓冲区（更健壮） */
/* ISR 将数据写入环形缓冲 + SetEvent */
/* Task 循环读取缓冲直到为空 */
/* 这样即使事件合并，数据也不会丢失（见 18.5.4 无锁环形缓冲） */
```

18.7.5 Alarm 精度与抖动

现象：Alarm 触发的任务执行间隔不精确，存在抖动（Jitter）。

根因分析：

抖动来源	**典型影响**	**控制方法**
高优先级任务/ISR 抢占	0~数 ms	提高目标任务优先级
Counter 分辨率	±1 tick	提高 Counter 频率
调度延迟	0.5~2μs	减少就绪任务数
中断关闭期间	取决于关闭时长	缩短 DisableAllInterrupts 时间

```
/* ===== 测量 Alarm 抖动的方法 ===== */

/* 期望每 10ms 触发一次的任务 */
volatile uint32 g_lastActivationTick = 0;
volatile int32  g_jitterTicks = 0;     /*
volatile int32  g_maxJitterTicks = 0;

#define EXPECTED_PERIOD_TICKS  (1000000U)    /* 10ms @100MHz STM */

TASK(Task_10ms)
{
    uint32 currentTick = STM0_TIM0.U;

    if (g_lastActivationTick != 0) {
        uint32 actualPeriod = currentTick - g_lastActivationTick;
        g_jitterTicks = (int32)(actualPeriod - EXPECTED_PERIOD_TICKS);

        /* 记录最大抖动 */
        int32 absJitter = (g_jitterTicks > 0) ? g_jitterTicks : -g_jitterTicks;
        if (absJitter > g_maxJitterTicks) {
            g_maxJitterTicks = absJitter;
        }
    }
    g_lastActivationTick = currentTick;
```

```
    /* 正常任务逻辑 */
    Sensor_Sample();

    TerminateTask();
}

/* 判断标准：
 * 抖动 < ±1% 周期 → 优秀（10ms 任务抖动 < ±100μs）
 * 抖动 < ±5% 周期 → 可接受
 * 抖动 > ±5% 周期 → 需要优化
 */
```

减少抖动的优化方法：

```
/* 方法 1：使用 Schedule Table 替代 Alarm（更精确） */
/* Schedule Table 基于绝对时间偏移，不受累积误差影响 */

/* 方法 2：在任务开始时补偿抖动 */
TASK(Task_Control_10ms)
{
    uint32 idealStartTick = g_lastIdealTick + EXPECTED_PERIOD_TICKS;
    g_lastIdealTick = idealStartTick;

    /* 基于理想时间点计算控制量，而非实际执行时间 */
    float dt = 0.01f;  /* 固定使用理想周期 10ms，不用实测间隔 */
    PID_Calculate(setpoint, feedback, dt);

    TerminateTask();
}
```

18.7.6 多核同步陷阱

陷阱 1：缓存一致性问题（TC334 无数据缓存，但有其他陷阱）

```
/* TC334 特性：TriCore 没有 L1 数据缓存，所以不存在经典的缓存一致性问题 */
/* 但存在以下等效问题： */

/* 陷阱：编译器优化导致的可见性问题 */
/* ❌ 不使用 volatile，编译器可能将变量缓存在寄存器中 */
uint32 g_sharedFlag = 0;  /* 编译器可能优化为寄存器访问 */

/* Core0: 设置标志 */
g_sharedFlag = 1;  /* 可能只写入寄存器，未写回内存 */

/* Core1: 检查标志 */
while (g_sharedFlag == 0);  /* 可能永远读取到旧值（寄存器中的 0） */

/* ✅ 修复：使用 volatile 确保每次访问都操作内存 */
volatile uint32 g_sharedFlag = 0;
```

陷阱 2：跨核启动时序

```
/* ❌ 错误：Core1 的任务访问 Core0 初始化的数据，但未同步 */
/* Core0 StartupHook */
void StartupHook(void)
{
```

```
    Init_SharedData();  /* 初始化共享数据 */
    /* Core1 可能在这之前就开始运行了！ */
}

/* ✅ 修复：使用同步屏障 */
volatile uint32 g_core0InitDone = 0;

/* Core0 StartupHook */
void StartupHook(void)
{
    Init_SharedData();
    __dsync();  /* 确保数据写入完成 */
    g_core0InitDone = 1;  /* 通知其他核 */
}

/* Core1 等待同步 */
void Core1_WaitSync(void)
{
    while (g_core0InitDone == 0) {
        __nop();  /* 等待 Core0 初始化完成 */
    }
    __dsync();  /* 确保看到 Core0 的所有写入 */
}
```

陷阱 3：Spinlock 内调用 OS API

```
/* ❌ 严重错误：Spinlock 区域内调用可能导致任务切换的 OS API */
GetSpinlock(SPINLOCK_A);
ActivateTask(Task_X);    /* 可能触发调度！ */
ReleaseSpinlock(SPINLOCK_A);
/* 如果调度器切换到其他任务，Spinlock 仍被持有 → 另一核永久等待 */

/* ✅ 正确做法：Spinlock 区域内禁止任何可能导致调度的操作 */
GetSpinlock(SPINLOCK_A);
g_pendingActivation = Task_X;  /* 仅设置标志 */
ReleaseSpinlock(SPINLOCK_A);
ActivateTask(g_pendingActivation);  /* 释放后再调用 */
```

18.8 本章小结

本章从六个维度系统讲解了 AUTOSAR OS 在 TC334 平台上的性能优化策略，核心要点回顾如下：

任务划分（18.1）： - 按周期划分适合控制类 ECU，遵循 RMS 原则 - 避免巨型任务和过度拆分，同周期功能应合并

中断优化（18.2）： - Cat1 用于极端响应（< 1μs），Cat2 用于需要 OS API 的场景 - ISR 执行时间目标：Cat2 < 10μs，Cat1 < 2μs - 中断到任务总延迟约 0.7~3.3μs @300MHz

栈空间（18.3）： - TriCore CSA 机制：每帧 64 bytes，与传统栈分离 - 栈大小 = 局部变量总量 + 编译器开销 + 50% 裕量 - 使用水印检测法验证运行时栈使用峰值

CPU 负载（18.4）： - 通过 Idle Task 时间占比计算负载率 - 目标：稳态负载 < 70%，峰值负载 < 85% - 优化手段：降频执行、查表替代计算、减少 API 调用

资源争用（18.5）： - 最小化资源持有时间（仅保护数据访问，不保护计算） - 多核 Spinlock 持有时间应 < 1μs - 单生产者-单消费者场景优先使用无锁环形缓冲

内存布局（18.6）： - DSPR 最快但最小（96KB），仅放置关键变量和栈 - 查找表放 DLMU，共享数据放 LMU - 热代码连续放置可提升 PFlash 预取命中率

性能优化 Checklist：

检查项	达标标准	检查方法
CPU 负载率	稳态 < 70%	Idle Task 时间占比
任务 WCET	< 60% 周期	STM 时间戳 + PreTaskHook/PostTaskHook
ISR 执行时间	Cat2 < 10µs	STM 测量
栈使用率	峰值 < 80% 配置	水印检测法
资源持有时间	< 100µs	自定义监控
Alarm 抖动	< ±5% 周期	连续周期差值统计
CSA 剩余帧数	> 总量 30%	FCX 链表遍历
中断响应延迟	< 5µs	时间戳测量

附录 A：AUTOSAR OS 全部 API 速查表

本附录按功能分类列出 AUTOSAR OS 全部 API（共 56 个），包含函数原型、所属标准和适用符合性类别。

A.1 任务管理（7 个 API）

API 名称	函数原型	标准	符合性类别	功能说明	调用上下文
ActivateTask	StatusType ActivateTask(TaskType TaskID)	OSEK	BCC1/BCC2/ECC1/ECC2	将指定任务从SUSPENDED 转为 READY，或增加激活计数	T, I2
ActivateTaskAsyn	StatusType ActivateTaskAsyn(TaskType TaskID)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	跨核异步激活任务（多核扩展）	T, I2
TerminateTask	StatusType TerminateTask(void)	OSEK	BCC1/BCC2/ECC1/ECC2	终止当前任务，释放资源回 SUSPENDED	T
ChainTask	StatusType ChainTask(TaskType TaskID)	OSEK	BCC1/BCC2/ECC1/ECC2	终止当前任务并激活目标任务（原子操作）	T
Schedule	StatusType Schedule(void)	OSEK	BCC1/BCC2/ECC1/ECC2	主动让出 CPU，触发重调度（非抢占任务用）	T
GetTaskID	StatusType GetTaskID(TaskRefType TaskID)	OSEK	BCC1/BCC2/ECC1/ECC2	获取当前运行任务的 ID	T, I2, EH, Pre/PostTH
GetTaskState	StatusType GetTaskState(TaskType TaskID, TaskStateRefType State)	OSEK	BCC1/BCC2/ECC1/ECC2	获取指定任务的当前状态	T, I2, EH, Pre/PostTH

A.2 事件管理（6 个 API）

API 名称	函数原型	标准	符合性类别	功能说明	调用上下文
SetEvent	StatusType SetEvent(TaskType TaskID, EventMaskType Mask)	OSEK	ECC1/ECC2	设置指定扩展任务的事件掩码	T, I2
SetEventAsyn	StatusType SetEventAsyn(TaskType	AUTOSAR	ECC1/ECC2	跨核异步设置	T, I2

	TaskID, EventMaskType Mask)			事件（多核扩展）	
ClearEvent	StatusType ClearEvent(EventMaskType Mask)	OSEK	ECC1/ECC2	清除当前任务的指定事件位	T
GetEvent	StatusType GetEvent(TaskType TaskID, EventMaskRefType Event)	OSEK	ECC1/ECC2	获取指定任务的当前事件掩码	T, I2, EH
WaitEvent	StatusType WaitEvent(EventMaskType Mask)	OSEK	ECC1/ECC2	等待指定事件，任务进入 WAITING 状态	T(Extended)
WaitAllEvents	StatusType WaitAllEvents(EventMaskType Mask)	AUTOSAR	ECC1/ECC2	等待所有指定事件位全部置位	T(Extended)

A.3 告警管理（5 个 API）

API 名称	函数原型	标准	符合性类别	功能说明	调用上下文
GetAlarm	StatusType GetAlarm(AlarmType AlarmID, TickRefType Tick)	OSEK	BCC1/BCC2/ECC1/ECC2	获取 Alarm 到期前的剩余 tick 数	T, I2
GetAlarmBase	StatusType GetAlarmBase(AlarmType AlarmID, AlarmBaseRefType Info)	OSEK	BCC1/BCC2/ECC1/ECC2	获取 Alarm 关联的 Counter 基本属性	T, I2, EH
SetRelAlarm	StatusType SetRelAlarm(AlarmType AlarmID, TickType increment, TickType cycle)	OSEK	BCC1/BCC2/ECC1/ECC2	设置相对 Alarm（相对当前 tick 偏移）	T, I2
SetAbsAlarm	StatusType SetAbsAlarm(AlarmType AlarmID, TickType start, TickType cycle)	OSEK	BCC1/BCC2/ECC1/ECC2	设置绝对 Alarm（绝对 tick 值触发）	T, I2
CancelAlarm	StatusType CancelAlarm(AlarmType AlarmID)	OSEK	BCC1/BCC2/ECC1/ECC2	取消已启动的 Alarm	T, I2

A.4 计数器管理（3 个 API）

API 名称	函数原型	标准	符合性类别	功能说明	调用上下文
IncrementCounter	StatusType IncrementCounter(CounterType CounterID)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	软件递增 Counter（软件	T, I2

				Counter专用）	
GetCounterValue	StatusType GetCounterValue(CounterType CounterID, TickRefType Value)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	获取Counter当前值	T, I2
GetElapsedValue	StatusType GetElapsedValue(CounterType CounterID, TickRefType Value, TickRefType ElapsedValue)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	获取自上次读取以来的经过tick 数	T, I2

A.5 资源管理（2 个 API）

API 名称	**函数原型**	**标准**	**符合性类别**	**功能说明**	**调用上下文**
GetResource	StatusType GetResource(ResourceType ResID)	OSEK	BCC1/BCC2/ECC1/ECC2	获取资源锁（提升优先级到天花板）	T, I2
ReleaseResource	StatusType ReleaseResource(ResourceType ResID)	OSEK	BCC1/BCC2/ECC1/ECC2	释放资源锁（恢复原优先级）	T, I2

A.6 中断管理（9 个 API）

API 名称	**函数原型**	**标准**	**符合性类别**	**功能说明**	**调用上下文**
DisableAllInterrupts	void DisableAllInterrupts(void)	OSEK	BCC1/BCC2/ECC1/ECC2	禁用所有中断（不可嵌套）	T, I2, SH, EH, All
EnableAllInterrupts	void EnableAllInterrupts(void)	OSEK	BCC1/BCC2/ECC1/ECC2	恢复所有中断（配对DisableAll）	T, I2, SH, EH, All
SuspendAllInterrupts	void SuspendAllInterrupts(void)	OSEK	BCC1/BCC2/ECC1/ECC2	挂起所有中断（可嵌套）	T, I2, SH,

					EH, All
ResumeAllInterrupts	void ResumeAllInterrupts(void)	OSEK	BCC1/BCC2/ECC1/ECC2	恢复所有中断（配对 SuspendAll）	T, I2, SH, EH, All
SuspendOSInterrupts	void SuspendOSInterrupts(void)	OSEK	BCC1/BCC2/ECC1/ECC2	挂起 OS 管理的中断（Cat2，可嵌套）	T, I2, SH, EH, All
ResumeOSInterrupts	void ResumeOSInterrupts(void)	OSEK	BCC1/BCC2/ECC1/ECC2	恢复 OS 管理的中断（配对 SuspendOS）	T, I2, SH, EH, All
EnableInterruptSource	StatusType EnableInterruptSource(ISRType ISRID, boolean ClearPending)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	使能指定 ISR 的中断源	T, I2
DisableInterruptSource	StatusType DisableInterruptSource(ISRType ISRID)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	禁用指定 ISR 的中断源	T, I2
ClearPendingInterrupt	StatusType ClearPendingInterrupt(ISRType ISRID)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	清除指定 ISR 的挂起标志	T, I2

A.7 调度表管理（6 个 API）

API 名称	函数原型	标准	符合性类别	功能说明	调用上下文
StartScheduleTableRel	StatusType StartScheduleTableRel(ScheduleTableType STID, TickType Offset)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	相对启动调度表	T, I2
StartScheduleTableAbs	StatusType StartScheduleTableAbs(ScheduleTableType STID, TickType Start)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	绝对启动	T, I2

				调度表	
StopScheduleTable	StatusType StopScheduleTable(ScheduleTableType STID)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	停止调度表	T, I2
NextScheduleTable	StatusType NextScheduleTable(ScheduleTableType STID_from, ScheduleTableType STID_to)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	切换到下一个调度表	T, I2
GetScheduleTableStatus	StatusType GetScheduleTableStatus(ScheduleTableType STID, ScheduleTableStatusRefType Status)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	获取调度表状态	T, I2, EH
StartScheduleTableSynchron	StatusType StartScheduleTableSynchron(ScheduleTableType STID)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	同步启动调度表（全局时间同步）	T, I2

A.8 系统管理（4 个 API）

API 名称	函数原型	标准	符合性类别	功能说明	调用上下文
StartOS	void StartOS(AppModeType Mode)	OSEK	BCC1/BCC2/ECC1/ECC2	启动OS，开始任务调度	main()

ShutdownOS	void ShutdownOS(StatusType Error)	OSEK	BCC1/BCC2/ECC1/ECC2	关闭OS，停止所有调度	T, I2, SuH, EH
GetActiveApplicationMode	AppModeType GetActiveApplicationMode(void)	OSEK	BCC1/BCC2/ECC1/ECC2	获取当前激活的应用模式	T, I2, SuH, SdH, EH, All
GetISRID	ISRType GetISRID(void)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	获取当前运行ISR的ID	T, I2, EH, Pre/PostTH

A.9 多核管理（5 个 API）

API 名称	函数原型	标准	符合性类别	功能说明	调用上下文
GetCoreID	CoreIdType GetCoreID(void)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	获取当前CPU核心ID	T, I2, SuH, SdH, EH, All
GetNumberOfActivatedCores	uint32 GetNumberOfActivatedCores(void)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	获取已激活的核心数量	T, I2
StartCore	void StartCore(CoreIdType CoreID, StatusType *Status)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	启动指定核心	main()
ControlIdle	StatusType	AUTOSA	BCC1/BCC2/ECC1/EC	控	T, I2

	ControlIdle(CoreIdType CoreID, IdleModeType IdleMode)	R	C2	制核心空闲行为	
ShutdownAllCores	void ShutdownAllCores(StatusType Error)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	关闭所有核心的OS	T, I2, EH

A.10 自旋锁（3 个 API）

API 名称	函数原型	标准	符合性类别	功能说明	调用上下文
GetSpinlock	StatusType GetSpinlock(SpinlockIdType SpinlockId)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	获取自旋锁（阻塞等待）	T, I2
ReleaseSpinlock	StatusType ReleaseSpinlock(SpinlockIdType SpinlockId)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	释放自旋锁	T, I2
TryToGetSpinlock	StatusType TryToGetSpinlock(SpinlockIdType SpinlockId, TryToGetSpinlockType *Success)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	非阻塞尝试获取自旋锁	T, I2

A.11 Hook 函数（6 个）

API 名称	函数原型	标准	符合性类别	功能说明	调用上下文
StartupHook	void StartupHook(void)	OSEK	BCC1/BCC2/ECC1/ECC2	OS 启动后、调度开始前调用（用户初始化）	OS内部调

					用
ShutdownHook	void ShutdownHook(StatusType Error)	OSEK	BCC1/BCC2/ECC1/ECC2	OS 关闭前调用（清理/记录原因）	OS内部调用
ErrorHook	void ErrorHook(StatusType Error)	OSEK	BCC1/BCC2/ECC1/ECC2	任何 API 返回错误时调用	OS内部调用
PreTaskHook	void PreTaskHook(void)	OSEK	BCC1/BCC2/ECC1/ECC2	任务切入前调用（调试/Trace）	OS内部调用
PostTaskHook	void PostTaskHook(void)	OSEK	BCC1/BCC2/ECC1/ECC2	任务切出后调用（调试/Trace）	OS内部调用
ProtectionHook	ProtectionReturnType ProtectionHook(StatusType Fault)	AUTOSAR	BCC1/BCC2/ECC1/ECC2	保护违规时调用（SC2/SC3/SC4）	OS内部调用

附录 B：常用 ARXML 配置模板

本附录提供 7 种常用 OS 对象的完整 ARXML 配置片段，可直接用于 AUTOSAR 配置工具导入。

B.1 Task 配置模板

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>Task_Blink</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
    /AUTOSAR/EcucDefs/Os/OsTask</DEFINITION-REF>
  <PARAMETER-VALUES>
    <ECUC-NUMERICAL-PARAM-VALUE>
      <DEFINITION-REF DEST="ECUC-INTEGER-PARAM-DEF">
        /AUTOSAR/EcucDefs/Os/OsTask/OsTaskPriority</DEFINITION-REF>
      <VALUE>2</VALUE>
    </ECUC-NUMERICAL-PARAM-VALUE>
    <ECUC-NUMERICAL-PARAM-VALUE>
      <DEFINITION-REF DEST="ECUC-INTEGER-PARAM-DEF">
        /AUTOSAR/EcucDefs/Os/OsTask/OsTaskActivation</DEFINITION-REF>
      <VALUE>1</VALUE>
    </ECUC-NUMERICAL-PARAM-VALUE>
    <ECUC-TEXTUAL-PARAM-VALUE>
      <DEFINITION-REF DEST="ECUC-ENUMERATION-PARAM-DEF">
        /AUTOSAR/EcucDefs/Os/OsTask/OsTaskSchedule</DEFINITION-REF>
      <VALUE>FULL</VALUE>
```

```
    </ECUC-TEXTUAL-PARAM-VALUE>
  </PARAMETER-VALUES>
</ECUC-CONTAINER-VALUE>
```

B.2 Alarm 配置模板

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>AlarmBlink</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
    /AUTOSAR/EcucDefs/Os/OsAlarm</DEFINITION-REF>
  <REFERENCE-VALUES>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">
        /AUTOSAR/EcucDefs/Os/OsAlarm/OsAlarmCounterRef</DEFINITION-REF>
      <VALUE-REF DEST="ECUC-CONTAINER-VALUE">
        /Os/OsCounter/SystemCounter</VALUE-REF>
    </ECUC-REFERENCE-VALUE>
  </REFERENCE-VALUES>
  <SUB-CONTAINERS>
    <ECUC-CONTAINER-VALUE>
      <SHORT-NAME>OsAlarmAction</SHORT-NAME>
      <SUB-CONTAINERS>
        <ECUC-CONTAINER-VALUE>
          <SHORT-NAME>OsAlarmActivateTask</SHORT-NAME>
          <REFERENCE-VALUES>
            <ECUC-REFERENCE-VALUE>
              <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">
                OsAlarmActivateTaskRef</DEFINITION-REF>
              <VALUE-REF DEST="ECUC-CONTAINER-VALUE">
                /Os/OsTask/Task_Blink</VALUE-REF>
            </ECUC-REFERENCE-VALUE>
          </REFERENCE-VALUES>
        </ECUC-CONTAINER-VALUE>
      </SUB-CONTAINERS>
    </ECUC-CONTAINER-VALUE>
  </SUB-CONTAINERS>
</ECUC-CONTAINER-VALUE>
```

B.3 Counter 配置模板

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>SystemCounter</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
    /AUTOSAR/EcucDefs/Os/OsCounter</DEFINITION-REF>
  <PARAMETER-VALUES>
    <ECUC-NUMERICAL-PARAM-VALUE>
      <DEFINITION-REF DEST="ECUC-INTEGER-PARAM-DEF">
        OsCounterMaxAllowedValue</DEFINITION-REF>
      <VALUE>65535</VALUE>
    </ECUC-NUMERICAL-PARAM-VALUE>
    <ECUC-NUMERICAL-PARAM-VALUE>
      <DEFINITION-REF DEST="ECUC-INTEGER-PARAM-DEF">
        OsCounterMinCycle</DEFINITION-REF>
      <VALUE>1</VALUE>
    </ECUC-NUMERICAL-PARAM-VALUE>
    <ECUC-NUMERICAL-PARAM-VALUE>
      <DEFINITION-REF DEST="ECUC-INTEGER-PARAM-DEF">
        OsCounterTicksPerBase</DEFINITION-REF>
      <VALUE>1</VALUE>
    </ECUC-NUMERICAL-PARAM-VALUE>
    <ECUC-TEXTUAL-PARAM-VALUE>
      <DEFINITION-REF DEST="ECUC-ENUMERATION-PARAM-DEF">
        OsCounterType</DEFINITION-REF>
```

```
      <VALUE>HARDWARE</VALUE>
    </ECUC-TEXTUAL-PARAM-VALUE>
  </PARAMETER-VALUES>
</ECUC-CONTAINER-VALUE>
```

B.4 Schedule Table 配置模板

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>SchedTable_10ms</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
    /AUTOSAR/EcucDefs/Os/OsScheduleTable</DEFINITION-REF>
  <PARAMETER-VALUES>
    <ECUC-NUMERICAL-PARAM-VALUE>
      <DEFINITION-REF>OsScheduleTableDuration</DEFINITION-REF>
      <VALUE>10</VALUE>
    </ECUC-NUMERICAL-PARAM-VALUE>
    <ECUC-NUMERICAL-PARAM-VALUE>
      <DEFINITION-REF>OsScheduleTableRepeating</DEFINITION-REF>
      <VALUE>1</VALUE>
    </ECUC-NUMERICAL-PARAM-VALUE>
  </PARAMETER-VALUES>
  <REFERENCE-VALUES>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF>OsScheduleTableCounterRef</DEFINITION-REF>
      <VALUE-REF>/Os/OsCounter/SystemCounter</VALUE-REF>
    </ECUC-REFERENCE-VALUE>
  </REFERENCE-VALUES>
  <SUB-CONTAINERS>
    <ECUC-CONTAINER-VALUE>
      <SHORT-NAME>EP_0</SHORT-NAME>
      <PARAMETER-VALUES>
        <ECUC-NUMERICAL-PARAM-VALUE>
          <DEFINITION-REF>OsScheduleTableExpiryPointOffset</DEFINITION-REF>
          <VALUE>0</VALUE>
        </ECUC-NUMERICAL-PARAM-VALUE>
      </PARAMETER-VALUES>
    </ECUC-CONTAINER-VALUE>
  </SUB-CONTAINERS>
</ECUC-CONTAINER-VALUE>
```

B.5 ISR 配置模板

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>ISR_SystemTimer</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
    /AUTOSAR/EcucDefs/Os/OsIsr</DEFINITION-REF>
  <PARAMETER-VALUES>
    <ECUC-TEXTUAL-PARAM-VALUE>
      <DEFINITION-REF>OsIsrCategory</DEFINITION-REF>
      <VALUE>CATEGORY_2</VALUE>
    </ECUC-TEXTUAL-PARAM-VALUE>
    <ECUC-NUMERICAL-PARAM-VALUE>
      <DEFINITION-REF>OsIsrInterruptPriority</DEFINITION-REF>
      <VALUE>2</VALUE>
    </ECUC-NUMERICAL-PARAM-VALUE>
    <ECUC-NUMERICAL-PARAM-VALUE>
      <DEFINITION-REF>OsIsrInterruptSource</DEFINITION-REF>
      <VALUE>OS_ISR_STM0_SR0</VALUE>
    </ECUC-NUMERICAL-PARAM-VALUE>
  </PARAMETER-VALUES>
</ECUC-CONTAINER-VALUE>
```

B.6 Resource 配置模板

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>ResShared</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
    /AUTOSAR/EcucDefs/Os/OsResource</DEFINITION-REF>
  <PARAMETER-VALUES>
    <ECUC-TEXTUAL-PARAM-VALUE>
      <DEFINITION-REF>OsResourceProperty</DEFINITION-REF>
      <VALUE>STANDARD</VALUE>
    </ECUC-TEXTUAL-PARAM-VALUE>
  </PARAMETER-VALUES>
</ECUC-CONTAINER-VALUE>
```

B.7 Event 配置模板

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>Ev_DataReady</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
    /AUTOSAR/EcucDefs/Os/OsEvent</DEFINITION-REF>
  <PARAMETER-VALUES>
    <ECUC-NUMERICAL-PARAM-VALUE>
      <DEFINITION-REF>OsEventMask</DEFINITION-REF>
      <VALUE>0x01</VALUE>
    </ECUC-NUMERICAL-PARAM-VALUE>
  </PARAMETER-VALUES>
</ECUC-CONTAINER-VALUE>
```

附录 C：各主流工具链对比

本附录对比三大 AUTOSAR 配置工具链的功能差异和使用流程。

C.1 工具链概览

特性	Vector DaVinci Configurator	EB tresos Studio	ETAS ISOLAR-A/B
厂商	Vector Informatik	Elektrobit (EB)	ETAS (Bosch 子公司)
定位	全栈 AUTOSAR 配置	BSW 配置专家工具	应用层+BSW 双向工具
支持 OS 版本	MICROSAR OS	EB tresos AutoCore OS	RTA-OS (ETAS)
配置格式	ARXML (ECUC Values)	ARXML (ECUC Values)	ARXML (ECUC Values)
代码生成	内置生成器	内置生成器	ISOLAR-B 生成器
GUI 类型	Eclipse 插件	Eclipse 独立 IDE	Eclipse 插件
多核支持	完整支持	完整支持	完整支持
验证功能	实时一致性检查	规则引擎验证	模型验证
价格区间	高（OEM 级）	中高	中高
学习曲线	中等	较陡	中等

C.2 典型使用流程对比

Vector DaVinci Configurator 流程

- 1. 导入 ECU Extract（.arxml）定义硬件平台

- 2. 添加 BSW 模块（Os, EcuM, BswM 等）
- 3. 在 GUI 中配置 OS 对象（Task, Alarm, ISR 等）
- 4. 运行一致性检查
- 5. 生成代码（Os_Cfg.h, Os_Cfg.c 等）
- 6. 集成到编译环境

EB tresos Studio 流程

- 1. 创建工程，选择目标 MCU 和 AUTOSAR 版本
- 2. 添加 OS 插件模块
- 3. 通过配置编辑器设置参数
- 4. 运行验证规则
- 5. 生成配置代码
- 6. 导出到构建系统

ETAS ISOLAR-A/B 流程

- 1. ISOLAR-A：定义 SWC 架构和接口
- 2. ISOLAR-B：配置 BSW 和 RTE
- 3. 映射 Runnable 到 OS Task
- 4. 配置 OS 对象参数
- 5. 生成 RTE 和 BSW 配置代码
- 6. 集成编译验证

C.3 选型建议

场景	推荐工具	理由
Vector MICROSAR 全套用户	DaVinci Configurator	原生集成，无兼容性问题
EB AutoCore OS 用户	EB tresos Studio	专用配置工具，支持深度定制
ETAS RTA-OS 用户	ISOLAR-A/B	应用层与 BSW 统一环境
学习/研究用途	任意（建议 EB tresos）	免费评估版功能较完整
自研 OS + 手动配置	文本编辑器 + 参考模板	本项目方式：直接编写 Os_Cfg.h/c

附录 D：FreeRTOS 与 AUTOSAR OS 概念映射总结表

本附录为从 FreeRTOS 过渡到 AUTOSAR OS 的开发者提供完整的概念映射参考。覆盖所有常用 FreeRTOS 概念。

FreeRTOS 概念	AUTOSAR OS 对应	差异说明
Task (xTaskCreate)	OsTask	AUTOSAR 需静态配置；FreeRTOS 可动态创

		建
Queue (xQueueCreate)	IOC (Inter-OS-Application Communication)	AUTOSAR IOC 支持跨核；FreeRTOS Queue 单核
Binary Semaphore	OsEvent (SetEvent/WaitEvent)	AUTOSAR 用事件位实现类似二值信号量
Counting Semaphore	OsResource + Counter 配合	AUTOSAR 无直接对应，需组合实现
Mutex (xSemaphoreCreateMutex)	OsResource (Priority Ceiling)	AUTOSAR 使用优先级天花板协议避免优先级反转
Software Timer (xTimerCreate)	OsAlarm + OsCounter	AUTOSAR Alarm 绑定 Counter 实现定时
Event Group (xEventGroupCreate)	OsEvent (事件掩码机制)	功能等价，AUTOSAR 事件掩码为 32/64 位
Task Notification	ActivateTask / SetEvent	FreeRTOS 轻量通知；AUTOSAR 通过激活/事件实现
Stream Buffer	IOC (队列模式)	AUTOSAR IOC 队列式通道提供类似功能
Message Buffer	IOC (变长队列)	AUTOSAR IOC 支持定长/变长数据传输
vTaskDelay	WaitEvent + Alarm 组合	AUTOSAR 无直接延时 API，通过事件等待实现
vTaskDelayUntil	Schedule Table Expiry Point	调度表在精确时刻激活任务
Idle Hook	IdleHook_Core0()	功能等价
Tick Hook	无直接对应（System Timer ISR）	可在 System Timer ISR 中添加用户代码
configASSERT	ErrorHook + ProtectionHook	AUTOSAR 更结构化的错误处理
Stack Overflow Hook	CFG_STACK_MONITOR / ErrorHook(E_OS_STACKFAULT)	功能等价
Critical Section (taskENTER_CRITICAL)	SuspendAllInterrupts / GetResource	AUTOSAR 提供两种粒度
ISR (中断服务函数)	Category 1 ISR / Category 2 ISR	AUTOSAR 区分 Cat1（快速）和 Cat2（OS 管理）
FreeRTOSConfig.h	Os_Cfg.h + ARXML	AUTOSAR 通常通过工具链生成配置
portYIELD	Schedule()	AUTOSAR 非抢占任务使用 Schedule() 让出 CPU
xTaskGetTickCount	GetCounterValue(SystemCounter, &val)	功能等价
pvPortMalloc / vPortFree	无（静态配置）	AUTOSAR OS 禁止动态内存分配
Task Priority (uxTaskPriorityGet)	静态配置 OsTaskPriority	AUTOSAR 不支持运行时修改优先级
vTaskSuspend / vTaskResume	无直接对应	AUTOSAR 使用事件等

		待实现挂起效果

D.1 关键差异总结

- 静态 vs 动态：AUTOSAR OS 所有对象在编译时静态配置，无运行时创建/删除
- 标准化：AUTOSAR OS 遵循 OSEK/VDX + AUTOSAR R19 标准，有明确的符合性类别
- 安全等级：AUTOSAR OS 支持 ASIL-D 功能安全（SC3/SC4），FreeRTOS 无原生安全认证
- 多核：AUTOSAR OS 原生支持多核（IOC、Spinlock），FreeRTOS 需 SMP 扩展
- 工具链：AUTOSAR OS 通常配合专业配置工具使用，FreeRTOS 手动配置为主
- 内存：AUTOSAR OS 禁止动态内存分配，FreeRTOS 支持多种内存分配方案
- 调度：AUTOSAR OS 支持 Non-Preemptive/Full Preemptive/Mixed，FreeRTOS 主要为抢占式
- 保护：AUTOSAR OS（SC2+）提供时间保护和内存保护，FreeRTOS 需 MPU 扩展

D.2 常见迁移场景代码示例

以下提供 5 个完整的 FreeRTOS → AUTOSAR OS 迁移代码案例，涵盖最常见的设计模式转换。

案例 1：FreeRTOS 周期任务 → AUTOSAR OS Alarm 驱动任务

FreeRTOS 版本：

```
void vTask_SensorRead(void *pvParameters) {
    TickType_t xLastWakeTime = xTaskGetTickCount();
    for (;;) {
        ReadSensor(&sensor_data);
        ProcessData(&sensor_data);
        vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(10));  // 10ms 周期
    }
}
// 创建: xTaskCreate(vTask_SensorRead, "Sensor", 256, NULL, 3, NULL);
```

AUTOSAR OS 版本：

```
// OIL 配置：
// ALARM AlarmSensor {
//   COUNTER = SystemCounter;
//   ACTION = ACTIVATETASK { TASK = Task_Sensor; };
// };
// 启动时: SetRelAlarm(AlarmSensor, 10, 10);  // 10ms 周期

TASK(Task_Sensor) {
    ReadSensor(&sensor_data);
    ProcessData(&sensor_data);
    TerminateTask();
```

```
}
```

案例 2：FreeRTOS Queue → AUTOSAR OS Event + 共享缓冲区

FreeRTOS 版本：

```
QueueHandle_t xQueue = xQueueCreate(10, sizeof(Message_t));

// 生产者
void vProducer(void *p) {
    Message_t msg = {.id = 1, .data = 42};
    xQueueSend(xQueue, &msg, portMAX_DELAY);
}

// 消费者
void vConsumer(void *p) {
    Message_t msg;
    for (;;) {
        xQueueReceive(xQueue, &msg, portMAX_DELAY);
        HandleMessage(&msg);
    }
}
```

AUTOSAR OS 版本：

```
// 共享缓冲区（需资源保护）
static Message_t msg_buffer[10];
static uint8 buf_head = 0, buf_tail = 0;

// 生产者（Basic Task，由外部事件激活）
TASK(Task_Producer) {
    Message_t msg = {.id = 1, .data = 42};
    GetResource(Res_Buffer);
      msg_buffer[buf_tail] = msg;
      buf_tail = (buf_tail + 1) % 10;
    ReleaseResource(Res_Buffer);
    SetEvent(Task_Consumer, EVT_MSG_READY);
    TerminateTask();
}

// 消费者（Extended Task，等待事件）
TASK(Task_Consumer) {
    Message_t msg;
    for (;;) {  // Extended Task 可以循环
        WaitEvent(EVT_MSG_READY);
        ClearEvent(EVT_MSG_READY);
        GetResource(Res_Buffer);
          msg = msg_buffer[buf_head];
          buf_head = (buf_head + 1) % 10;
        ReleaseResource(Res_Buffer);
        HandleMessage(&msg);
    }
}
```

案例 3：FreeRTOS Mutex → AUTOSAR OS Resource

FreeRTOS 版本：

```
SemaphoreHandle_t xMutex = xSemaphoreCreateMutex();

void vTask_A(void *p) {
```

```
    xSemaphoreTake(xMutex, portMAX_DELAY);
     UART_Send(data, len); // 保护共享外设
    xSemaphoreGive(xMutex);
}
```

AUTOSAR OS 版本：

```
// OIL: RESOURCE Res_UART { RESOURCEPROPERTY = STANDARD; };
// 天花板优先级自动计算为 max(所有使用该资源的 Task 优先级)

TASK(Task_A) {
    GetResource(Res_UART);
     UART_Send(data, len);
    ReleaseResource(Res_UART);
    TerminateTask();
}
// 优先级最大值势：PCP 协议天然防止死锁和优先级反转
// FreeRTOS Mutex 只有优先级继承，不能完全防死锁
```

案例 4：FreeRTOS Software Timer → AUTOSAR OS Alarm 回调

FreeRTOS 版本：

```
TimerHandle_t xTimer;
void vTimerCallback(TimerHandle_t xTimer) {
    watchdog_counter++;
    if (watchdog_counter > TIMEOUT) TriggerReset();
}
xTimer = xTimerCreate("WDG", pdMS_TO_TICKS(100), pdTRUE, NULL, vTimerCallback);
xTimerStart(xTimer, 0);
```

AUTOSAR OS 版本：

```
// OIL 配置：
// ALARM AlarmWDG {
//   COUNTER = SystemCounter;
//   ACTION = ALARMCALLBACK { ALARMCALLBACKNAME = "WDG_Callback"; };
//   AUTOSTART = TRUE {
//     ALARMTIME = 100; CYCLETIME = 100;
//     APPMODE = OSDEFAULTAPPMODE;
//   };
// };

ALARMCALLBACK(WDG_Callback) {
    watchdog_counter++;
    if (watchdog_counter > TIMEOUT) ShutdownOS(E_OS_STATE);
}
// 注意：Alarm 回调在 ISR 上下文执行，不能调用阻塞 API
```

案例 5：FreeRTOS Event Group → AUTOSAR OS Event

FreeRTOS 版本：

```
EventGroupHandle_t xEvents = xEventGroupCreate();
#define EVT_SENSOR  (1 << 0)
#define EVT_COMM    (1 << 1)
#define EVT_ALL     (EVT_SENSOR | EVT_COMM)

void vTask_Main(void *p) {
    for (;;) {
        // 等待两个事件都就绪
```

```
        EventBits_t bits = xEventGroupWaitBits(
            xEvents, EVT_ALL, pdTRUE, pdTRUE, portMAX_DELAY);
        if ((bits & EVT_ALL) == EVT_ALL) {
            ProcessAll();
        }
    }
}
```

AUTOSAR OS 版本：

```
// Extended Task 事件机制
#define EVT_SENSOR   ((EventMaskType)0x01)
#define EVT_COMM     ((EventMaskType)0x02)
```

```
TASK(Task_Main) {
    for (;;) {  // Extended Task
        WaitAllEvents(EVT_SENSOR | EVT_COMM);  // 等待全部事件(AND)
        ClearEvent(EVT_SENSOR | EVT_COMM);
        ProcessAll();
    }
}
// 区别：AUTOSAR Event 绑定到特定 Extended Task
// FreeRTOS EventGroup 是独立对象，任何任务可等待
```

D.3 迁移检查清单

以下是从 FreeRTOS 迁移到 AUTOSAR OS 的完整检查清单：

- □ 识别所有 vTaskDelay/vTaskDelayUntil → 改为 Alarm 周期激活
- □ 识别所有 Queue → 改为 Event + 共享缓冲区（Resource 保护）
- □ 识别所有 Mutex → 改为 Resource（PCP 自动防反转）
- □ 识别所有 Software Timer → 改为 Alarm（回调或激活 Task）
- □ 识别所有 Event Group → 改为 Event（注意仅 Extended Task 可用）
- □ 识别所有 Task Notification → 改为 Event 或 ActivateTask
- □ while(1) 循环任务 → Basic Task 单次执行 或 Extended Task + WaitEvent
- □ 动态创建对象 → 全部改为静态配置（OIL/ARXML）
- □ vTaskDelete → 不需要（Basic Task 自动回到 SUSPENDED）
- □ 优先级数字含义相同（越大越高），但范围不同

附录 E：API 调用上下文兼容性完整矩阵

本矩阵基于 AUTOSAR CP R25-11 SWS_OS 官方规范编制。

符号说明：

- ✓ = 允许调用
- ✗ = 禁止调用（返回 E_OS_CALLEVEL）
- △ = 有条件（仅特定 Hook 中允许）

上下文缩写：T=Task, I2=Cat2 ISR, SuH=StartupHook, SdH=ShutdownHook, EH=ErrorHook, Pre/PostTH=PreTaskHook/PostTaskHook

E.1 任务/事件/资源 API

API	Task	Cat2 ISR	StartupH	ShutdownH	ErrorH	Pre/PostTH
ActivateTask	√	√	X	X	X	X
TerminateTask	√	X	X	X	X	X
ChainTask	√	X	X	X	X	X
Schedule	√	X	X	X	X	X
GetTaskID	√	√	X	X	√	√
GetTaskState	√	√	X	X	√	√
SetEvent	√	√	X	X	X	X
ClearEvent	√	X	X	X	X	X
WaitEvent	√	X	X	X	X	X
GetEvent	√	√	X	X	√	X
GetResource	√	√	X	X	X	X
ReleaseResource	√	√	X	X	X	X

E.2 告警/计数器/调度表 API

API	Task	Cat2 ISR	StartupH	ShutdownH	ErrorH	Pre/PostTH
SetRelAlarm	√	√	X	X	X	X
SetAbsAlarm	√	√	X	X	X	X
CancelAlarm	√	√	X	X	X	X
GetAlarm	√	√	X	X	X	X
GetAlarmBase	√	√	X	X	√	X
GetCounterValue	√	√	X	X	X	X
GetElapsedValue	√	√	X	X	X	X
StartScheduleTableRel	√	√	X	X	X	X
StartScheduleTableAbs	√	√	X	X	X	X
StopScheduleTable	√	√	X	X	X	X
NextScheduleTable	√	√	X	X	X	X
GetScheduleTableStatus	√	√	X	X	√	X

E.3 系统/多核/Spinlock API

API	Task	Cat2 ISR	StartupH	ShutdownH	ErrorH	Pre/PostTH
GetActiveApplicationMode	√	√	√	√	√	√
ShutdownOS	√	√	√	X	√	X
GetISRID	√	√	X	X	√	√
GetSpinlock	√	√	X	X	X	X
ReleaseSpinlock	√	√	X	X	X	X
TryToGetSpinlock	√	√	X	X	X	X
GetCoreID	√	√	√	√	√	√
StartCore	X	X	X	X	X	X

E.4 中断管理 API

API	Task	Cat2 ISR	StartupH	ShutdownH	ErrorH	Pre/PostTH
DisableAllInterrupts	√	√	√	√	√	√
EnableAllInterrupts	√	√	√	√	√	√
SuspendAllInterrupts	√	√	√	√	√	√
ResumeAllInterrupts	√	√	√	√	√	√
SuspendOSInterrupts	√	√	√	√	√	√
ResumeOSInterrupts	√	√	√	√	√	√

注：StartCore 仅在 main() 函数中、StartOS() 之前调用。

Cat1 ISR 仅允许调用中断开关 API（Disable/Enable/Suspend/ResumeAllInterrupts）。

违反调用上下文约束将触发 E_OS_CALLEVEL 错误，进入 ErrorHook。