Application Note

使用 MSPM0 MCU 优化 C 代码体积：技巧与诀窍

摘要

在选择微控制器 (MCU) 时，可用的代码空间和非易失性存储器往往是决策过程中的关键因素。为在目标器件的约束范围内实现功能最大化，开发者可以在软件开发和编译过程中采用特定策略，以实现代码体积效率。本应用笔记介绍了在使用 MSPM0 器件时，Code Composer Studio™ (CCS) 的各种优化配置，重点阐述了可显著减小代码占用的编译器设置。此外，本文档还提供了编码最佳实践与技巧，用于为 MSPM0 应用编写更紧凑、更注重体积优化的代码。

内容

1 简介.....2
2 C 编译器优化.....3
2.1 优化设置.....3
2.2 运行时模型选项.....6
3 编码技巧.....8
3.1 为变量和常量选用尽可能小的类型.....8
3.2 避免乘法和除法.....8
3.3 使用查找表代替手动计算.....8
3.4 高效使用函数.....8
3.5 实现递减计数循环.....8
3.6 使用 __even_in_range() 内建函数.....9
3.7 volatile 关键字.....9
4 示例.....10
4.1 优化级别.....10
4.2 LTO.....10
4.3 运行时模型选项.....10
5 总结.....11
6 参考资料.....12

商标

Code Composer Studio™ is a trademark of Texas Instruments.
所有商标均为其各自所有者的财产。

English Document: SDAA552
Copyright © 2026 Texas Instruments Incorporated

1 简介

战略性地选择编译器设置，并在编写代码时兼顾体积优化，能够对代码体积产生显著影响。本应用笔记中用作示例的代码体积，采用的是代码示例 *i2c_controller_target_dynamic_switching*，该示例可为 MSPM0C1104 MCU 构建，此 MCU 包含 16kB Flash 和 1kB SRAM。所列代码体积均使用 CCS 20.4 版及 TI Clang 4.0.4.LTS 版构建。

English Document: SDAA552
Copyright © 2026 Texas Instruments Incorporated

2 C 编译器优化

尽管在可读性方面 C 通常优于汇编，但在代码空间受限的器件上，C 可能会引入可观的开销。然而，通过精心的编译器优化，并策略性地运用全局变量等特性，C 能够接近汇编级别的效率，同时还能受益于编译器的自动优化，而无需手动调优汇编代码。

2.1 优化设置

在 CCS 中，优化级别设置可在 Project > Project Properties⋯ > Build > Arm Compiler > Optimization 中找到。

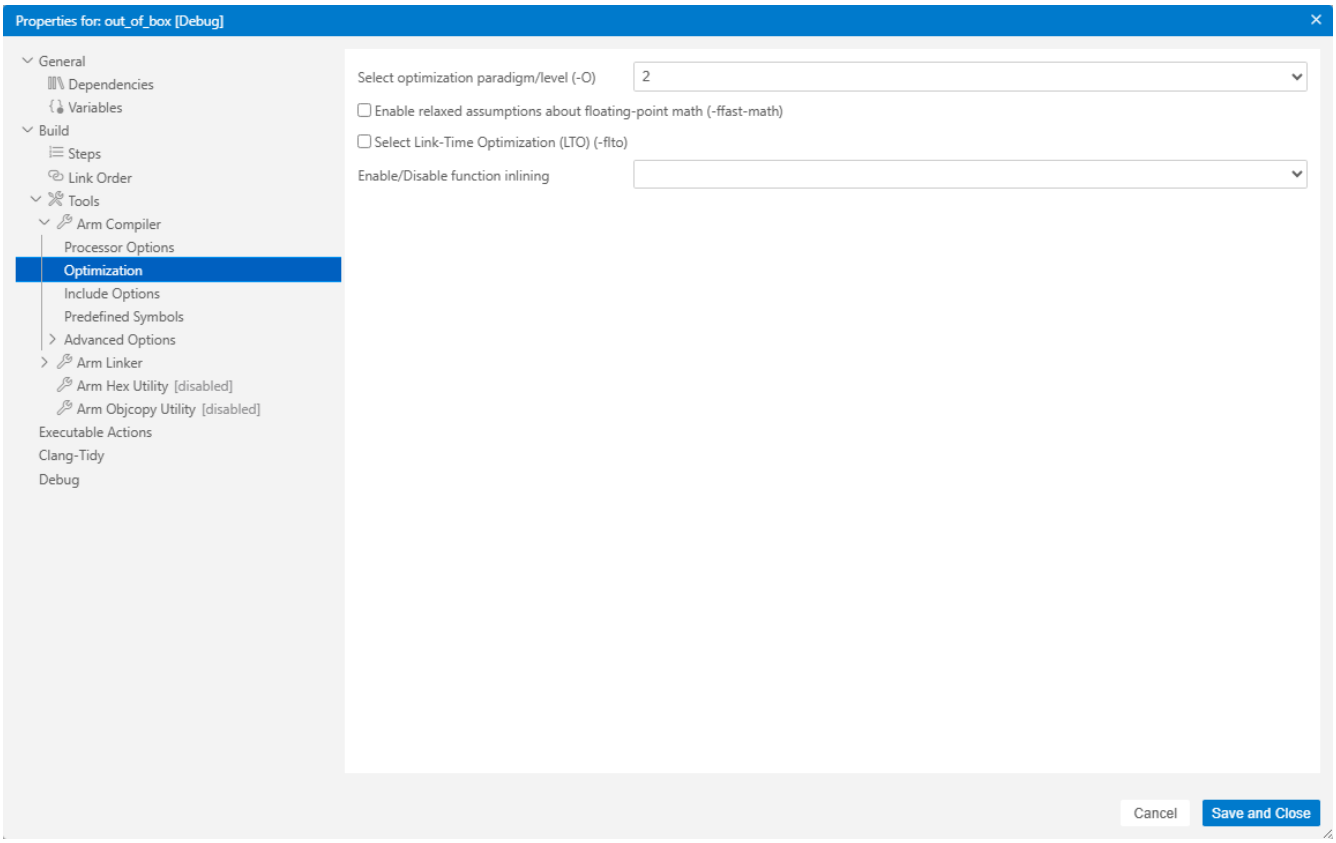


图 2-1. 优化设置

2.1.1 优化级别

优化级别标志可配置为：以牺牲性能为代价，偏向更小的编译器生成代码体积；或以牺牲代码体积为代价，偏向更佳性能。需要注意的是，由于优化算法和代码生成方法各不相同，编译器的选择会直接影响代码体积。

表 2-1. 优化级别

		代码优化级别							
IDE	Keil	O0	O1	O2	O3	Ofast	Os balanced	Oz image size	Og
	CCS	O0	O1	O2	O3	Ofast	Os	Oz	/
	IAR	None	Low	Medium	High	High-speed	High-Balanced	High-Size	/

表 2-2. 优化摘要

优化级别选项	优化摘要	执行的优化
-O0	不执行任何优化。*建议在需要最大限度可调试性时使用此选项。*	无

English Document: SDAA552
Copyright © 2026 Texas Instruments Incorporated

表 2-2. 优化摘要（续）

优化级别选项	优化摘要	执行的优化
-O1、-O	启用受限的优化。在代码体积与可调试性之间提供良好的折衷。	• 控制流简化 • 将连续的 icmp 合并为 memcmp • memcpy/memset/memcmp 内联 • 常量提升 • 部分内联对库函数的调用 • 对 always_inline 函数进行内联 • 全局变量合并 • 合并不相交的栈槽 • 循环强度削减 • 循环不变量代码运动 • 通用子表达式消除 • 死参数消除 • 机器码下沉 • 窥孔优化 • 尾部谓词化 • 尾部复制 • 加载/存储优化 • 简单寄存器合并 • 复制传播 • 条件常量传播 • 被调用值传播 • 控制流优化 • If 转换 • Thumb2 指令体积缩减 • 死代码消除 • 循环向量化 • printf 函数特化 • 小型 memcpy/memset 函数特化 • 有条件地消除死库调用 • 循环旋转 • 循环展开
-O2	启用大多数优化。禁用某些可能导致编译器生成代码体积增大的优化。	• 执行所有 (-O1) 优化，另加： • 函数集成/内联 • 指令推测 • 值传播 • 跳转线程化（非 DFA） • 尾调用消除 • 合并的加载/存储移动 • 全局值编号 • 内存依赖分析 • 死存储消除 • 超字级并行 (SLP) 向量化 • 合并冗余指令 • 死全局消除 • 全局重复常量合并 • 快速 memcpy/memset 函数特化 • 将循环目标边界对齐到 16 字节 (Cortex-R4/R5)

English Document: SDAA552
Copyright © 2026 Texas Instruments Incorporated

表 2-2. 优化摘要（续）

优化级别选项	优化摘要	执行的优化
-O3	启用 -O2 可用的所有优化，以及另一些可能增大编译器生成代码体积的优化。*建议在优化性能时使用此选项，但这很可能会增大代码体积。*	• 执行所有 (-O2) 针对速度调优的优化，另加： • 用受支持的内建函数替换函数 • 附加的别名分析和循环优化 • 激进函数内联 • 调用点拆分 • 将“按引用”传递的参数提升为标量 • 合并基于模式的表达式
-Og	启用受限的优化，同时保持可调试性。启用 -O1 可用的所有优化，并附加 -O2 中的部分优化。	• 执行所有 (-O2) 优化，但禁用以下项： • 不进行循环向量化 • 除 always_inline 函数外不进行函数内联 • 不进行指令推测 • 不进行跳转线程化 • 不进行值传播 • 不进行尾调用消除 • 不进行合并的加载/存储移动 • 不进行全局值编号 • 不进行内存依赖分析 • 不进行超字级并行 (SLP) 向量化 • 不进行死全局消除 • 不进行全局重复常量合并
-Os	启用 -O2 可用的所有优化，外加旨在减小代码体积的附加优化。与 -Oz 选项相比，-Os 选项会避免那些可能损害性能的代码体积优化。	• 执行所有 (-O2) 针对代码体积调优的优化，另加： • 先前已启用、针对代码体积调优的优化 • 小型 memcpy/memset 函数特化 • 最小化 memcpy/memset/memcmp 内联 • 不有条件地消除死库调用 • 不将循环目标边界对齐到 16 字节 (Cortex-R4/R5)
-Oz	启用 -O2 可用的所有优化，外加旨在减小代码体积的附加优化。与 -Os 选项相比，-Oz 选项会执行额外的代码体积优化，而这些优化可能损害性能。*若以减小代码体积为优先，建议使用此优化设置。*	• 执行所有 (-Os) 优化，另加： • 机器码外提 • 不进行循环向量化 • 减少对代码体积有影响的激进优化 • 不将循环目标边界对齐到 16 字节 (Cortex-R4/R5)
-Ofast	已弃用此选项。此选项启用 -O3 可用的所有优化，外加额外的激进优化，但这些优化不保证严格符合语言标准。若需要此行为，建议将 -O3 选项与 -fast-math 结合使用。	• 执行所有 (-O3) 优化 • 额外的激进优化

关于不同优化级别下汇编代码影响的示例，请参阅 *TI Arm Clang 编译器工具用户指南*。

2.1.2 Optnone

optnone 函数属性可施加于某个函数，指示编译器在为该函数生成代码时不进行任何优化。这对单个函数的效果，等同于 -O0 命令行选项对整个应用的效果。

语法

<返回类型> *符号* (<参数>) __attribute__((**optnone**));

示例

在下面的 C 示例中，*park_and_wait* 函数包含一个空的 while 循环。如果没有对 *park_and_wait* 施加 **optnone** 属性，编译器会将该循环优化掉，并移除对该函数的引用：

```
void check_peripheral();
__attribute__((optnone)) void park_and_wait();
```


Copyright © 2026 Texas Instruments Incorporated

```
void run_a_check_on_peripheral(void) {
check_peripheral();
}
void check_peripheral() {
if (*((unsigned long *)(268612608)) != 286529877) {
park_and_wait();
}
}
__attribute__((optnone)) void park_and_wait() {
while(1) {
;
}
}
```

上述代码经编译器生成的汇编表明，即使指定了 -O2 优化选项，对 *park_and_wait* 的引用依然完整保留。如果未对 *park_and_wait* 函数施加 **optnone** 属性，优化器会检测到 *park_and_wait* 中的空 while 循环，并移除 *check_peripheral* 中的引用：

```
%> tiarmclang -mcpu=cortex-m4 -O2 -S optnone_function_attr.c
%> cat optnone_function_attr.s
...
.section .text.check_peripheral,"ax",%progbits
.hidden check_peripheral @ -- Begin function check_peripheral
.globl check_peripheral
.p2align 2
.type check_peripheral,%function
.code 16 @ @check_peripheral
.thumb_func
check_peripheral:
movw r0, #46080
movt r0, #4098
ldr r0, [r0]
movw r1, #6485
movt r1, #4372
cmp r0, r1
it eq
bxeq lr
.LBB1_1: @ %if.then
bl park_and_wait
...
```

2.1.3 链接时优化 - LTO

LTO 使编译器能够在链接阶段跨多个源文件进行优化，而不是逐个文件单独优化。LTO 可通过 -flto 编译器标志启用。

对于 Cortex-M0+ 处理器，在使用 -Oz 选项编译的 191 个 M0+ 驱动库应用上进行测试时，启用 LTO 实现了 23 – 25% 的代码体积缩减。代码体积缩减百分比可通过以下公式计算：

代码体积缩减 % = (1 – (LTO 代码体积 / 非 LTO 代码体积)) × 100

启用 LTO 还可带来显著的加速效果。对于 Cortex-M0+ 处理器，在使用 -O3 选项编译的 15 个 EEMBC AutoBench 应用上进行测试时，加速因子最高可达 1.20，时间/周期数减少 3.8%。可使用以下公式计算这些指标：

加速因子 = (非 LTO 周期数 / LTO 周期数)

周期数缩减 % = ((非 LTO 周期数 - LTO 周期数) / 非 LTO 周期数) × 100

2.2 运行时模型选项

在 CCS 中，运行时模型选项设置可在 Project > Project Properties⋯ > Build > Arm Compiler > Advanced Options > Runtime Model Options 中找到。

English Document: SDAA552
Copyright © 2026 Texas Instruments Incorporated

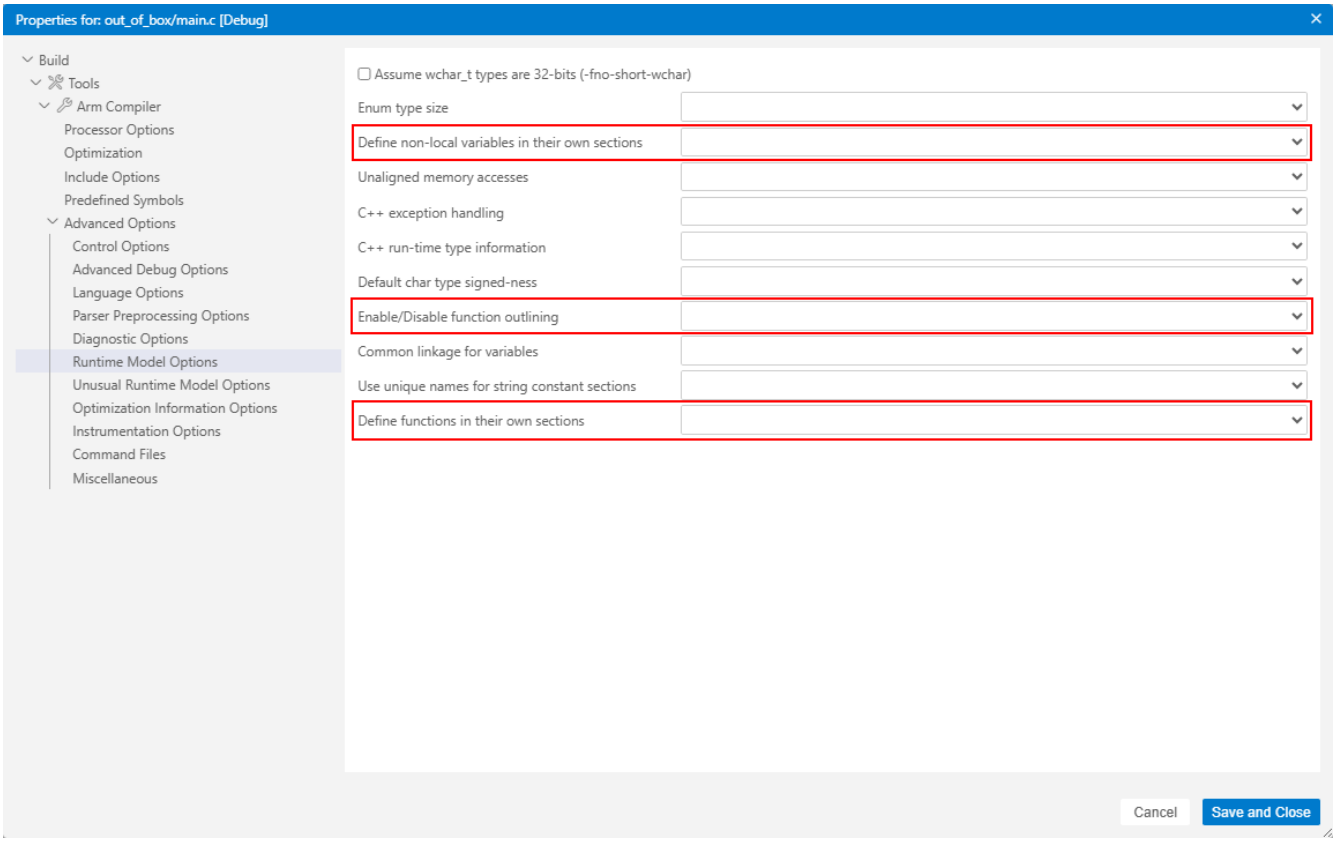


图 2-2. 运行时模型选项

2.2.1 数据段与函数段

-fdata-sections 和 -function-sections 选项默认启用。这些标志指示编译器将数据定义的代码和函数定义的代码分别生成到各自独立的段中。因此，这些标志能够实现更细粒度的死代码消除。

2.2.2 函数外提

函数外提（又称机器码外提）是一种优化技术，它识别出反复出现的机器码序列，并将每一处实例替换为对包含这些操作的新函数的调用。当选择 -Oz 选项时，函数外提即被启用。

表 2-3. 函数外提

函数外提选项	效果
-moutline	在函数内部执行机器码外提；仅当确保能减少净代码体积时才会应用。
-moutline-inter-function	更激进的函数外提，但并不总能保证整体代码体积缩减。可与 -Oz 选项结合使用，以启用函数间外提。
-mno-outline	在使用 -Oz 选项时，为给定的编译单元禁用函数外提。

English Document: SDAA552
Copyright © 2026 Texas Instruments Incorporated

3 编码技巧

3.1 为变量和常量选用尽可能小的类型

常量与代码一样，存储在非易失性存储器中。因此，在定义常量时使用尽可能小的类型，有助于减少浪费的代码空间。例如，若某个查找表仅包含 0 到 255 之间的值，则声明该常量表时使用无符号 8 位类型，相比 16 位 int 类型可将空间减半。这一理念同样适用于使用 PERSISTENT 关键字存储在 Flash 中的变量。如果变量存储在 Flash 而非 SRAM 中，它就会占用本可用作代码的 Flash 空间。因此，在这种情况下，选用最小的类型同样重要。

甚至可以在编写代码时让查找表使用更小的数值，前提是精度对应用而言仍然足够。一个例子是包含定时器 PWM 输出值的常量数组。如果定时器时钟源为 32768Hz，而定时器输出仅为 60Hz，则定时器周期或占空比的最大计数值可能仅为 32768 / 60 = 545。该值无法存入 8 位变量。但如果利用时钟模块或定时器内部的时钟分频器将定时器时钟源四分频，则占空比的最大计数值将变为 8192 / 60 = 136，小到足以存入 8 位值。只要这仍能为占空比设置提供足够的精度，这一简单的改动就能将包含定时器计数值设置的常量查找表大小减半。明智地选择数值的存储方式，可以对数组和查找表的大小产生巨大影响。

除了选择尽可能小的数据类型外，还可以通过使用位域来进一步优化——将多个布尔标志打包进单个字节，而不是使用各自独立的变量。对只读数据施加 const 限定符，以确保其存储在 Flash 存储器中，而不是占用 SRAM 空间。此外，应避免在代码中使用字符串字面量，因为它们会迅速耗尽可用内存。

3.2 避免乘法和除法

乘法与除法运算需要耗费大量周期来执行，并且需要更多代码来实现这些运算。因此，设法从代码中去除不必要的乘法或除法，是节省代码空间和执行时间的绝佳途径。

对于乘以或除以 2 的幂（例如 2、4、8、16、……）的情况，可以使用位移来代替。在 C 中进行位移，使用 >> 表示右移，<< 表示左移，后接要移位的位数。这比乘法或除法高效得多，因为 CPU 中有用于位移的汇编指令和硬件。

另一种方法是判断乘法或除法能否在编译时而非运行时计算。例如，当两个常量相乘且不涉及变量时，编译器可以在编译期间算出结果并直接使用该计算值，从而省去原本会同时消耗内存和执行时间的乘法代码。

如果乘法中涉及某个变量，因而无法全部提前计算，而该变量又具有已知的取值范围，就可以创建一张查找表来存放各种可能的结果。

3.3 使用查找表代替手动计算

如果必须针对不同数值反复执行某项复杂计算，不妨考虑查找表是否是更好的替代方案——对于浮点运算而言尤其如此。如果参与计算的变量输入具有已知的取值范围，就可以创建一张查找表来存放各种可能的结果。但请记住，查找表并不总能带来更小的体积，因为该表本身也必须驻留在器件的非易失性存储器中。

3.4 高效使用函数

函数可以显著提高代码的可读性。然而，函数也会在执行速度和代码空间方面带来额外开销。如果某个函数只被调用一次，尤其是对于小型函数而言，这些额外开销通常并不值得。为了兼顾可读性又不引入额外开销，可以通过在函数声明中使用 inline（或 __inline）关键字将该函数内联。这一声明指示编译器在每次调用该函数的位置插入函数体内容，而不是执行实际的函数调用。

然而，在某些情况下，某项操作需要在代码中多次执行。此时，函数反而能成为更节省代码空间的解决方案，因为它允许同一段代码在多个不同位置重复使用。有意识地编写可复用的代码，同样能对代码体积产生重大影响。

3.5 实现递减计数循环

对于需要重复执行操作或循环的应用而言，选择合适的循环结构会影响代码体积。递减计数循环通常比递增计数循环更高效，因为它利用了处理器内置的零标志位。当循环向下计数至零时，递减指令会自动置位零标志位，从而无需额外的比较指令即可检查循环终止条件。相比之下，递增计数循环则需要显式的比较指令，并且每次迭代都必须加载目标值。

English Document: SDAA552
Copyright © 2026 Texas Instruments Incorporated

3.6 使用 __even_in_range() 内建函数

__even_in_range(x,NUM) 内建函数为编译器提供了一条关于 switch 语句的提示：值 x 始终是 0 到 NUM（含）范围内的偶数值。这使得编译器能够对 x 的取值作出更多假设，从而进行比通常情况下更深入的优化。该内建函数通常用于中断服务程序 (ISR)，因为中断向量始终具有固定的取值范围，且始终为偶数值。

然而，__even_in_range() 并不限于 ISR 使用。只要代码能够保证 switch 的控制变量始终为偶数且其取值有上限，该内建函数也可用于代码中的其他 switch 语句。一个例子是某个字节计数器变量，它每次递增 2 而非 1，并且其最大值由代码控制，使其回绕至 0 或 2。

3.7 volatile 关键字

volatile 关键字是 C 标准的一部分。tiarmclang 编译器在所有语言模式下均支持该关键字。

volatile 关键字指示编译器如何访问变量，这要求编译器不得投机取巧地优化涉及该变量的表达式。例如，外部程序、中断、另一个线程或外围设备也可访问该变量。

编译器会使用数据流分析来确定访问是否合法，从而尽可能消除冗余的存储器访问。不过，一些存储器访问可能在编译器未看到的方面比较特殊，在这类情况下，您应当使用 **volatile** 关键字来防止编译器优化掉某些重要内容。对于已声明为 volatile 的变量，编译器不会优化掉对该变量的任何访问。volatile 读取和写入的次数与 C/C++ 代码中的完全相同，不多不少而且顺序也完全相同。

任何可能由明显程序控制流程外部的事物（例如中断服务例程）进行修改的变量必须声明为 **volatile**。这会告知编译器，中断函数可能随时修改该变量的值，因此编译器不能执行会改变该变量访问次数或访问顺序的优化。这就是 volatile 关键字的主要用途。

Copyright © 2026 Texas Instruments Incorporated

4 示例

在使用优化级别 -O2 构建示例 i2c_controller_target_dynamic_switching 后，内存分配结果为：约 2.5kB 的 Flash 和 117B 的 SRAM 被占用。

图 4-1. 内存分配

4.1 优化级别

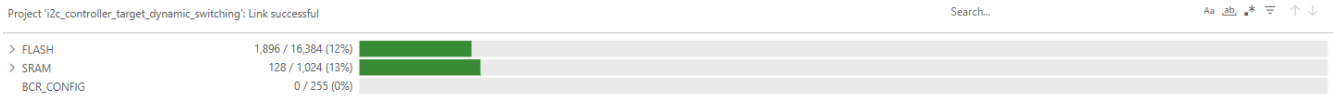
表 4-1 展示了示例在不同优化级别下的代码体积。

表 4-1. 代码体积

优化级别	代码体积
-O0	4,112B
-O1	2,208B
-O2	2,464B
-O3	2,456B
-Og	2,208B
-Os	2,160B
-Oz	2,128B

4.2 LTO

当在 -Oz 优化级别下启用 LTO 时，代码体积降至 1,896B，而 SRAM 用量则略微增至 128B。



4.3 运行时模型选项

以下测试均在优化级别 -O2 下进行。

表 4-2. 运行时模型选项

LTO	外提	数据/函数段	代码体积 Flash / SRAM
否	-moutline	默认（启用）	2,456B / 105B
否	-moutline-inter-function	默认	2,456B / 105B
否	-mno-outline	默认	2,464B / 105B
否	-moutline	-fno-data-sections -fno-function-sections	2,520B / 105B
是	-moutline	默认	2,328B / 105B

English Document: SDAA552
Copyright © 2026 Texas Instruments Incorporated

5 总结

尝试不同的方法通常是寻找最优编码解决方案的最佳途径。借助版本控制系统，你可以为项目创建独立的分支或副本，以尝试不同的优化方法。之后便可对比结果，看看哪种方法效果最佳。

English Document: SDAA552
Copyright © 2026 Texas Instruments Incorporated

6 参考资料

1. 德州仪器 (TI)，*TI Arm Clang 编译器工具用户指南 - 5.0.0.STS*
2. 德州仪器 (TI) *使用 MSP430™ MCU 优化 C 代码体积：技巧与诀窍* 应用报告

English Document: SDAA552
Copyright © 2026 Texas Instruments Incorporated

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、与某特定用途的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他安全、安保法规或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。对于因您对这些资源的使用而对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，您将全额赔偿，TI 对此概不负责。

TI 提供的产品受 TI 销售条款)、TI 通用质量指南 或 ti.com 上其他适用条款或 TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。 除非德州仪器 (TI) 明确将某产品指定为定制产品或客户特定产品，否则其产品均为按确定价格收入目录的标准通用器件。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

版权所有 © 2026，德州仪器 (TI) 公司

最后更新日期：2025 年 10 月