Application Note

C7000 优化入门指南

摘要

从另一个 DSP 或处理器迁移到 C7000 的过程需要进行一些优化才能充分利用 C7x 架构的优势。本文档介绍了优化的一般顺序（从最简单到较困难的步骤）。

总之，启动时有几个基本步骤。

1. 在 C7x 内核上运行算法，并对初始性能进行基准测试，以开始测试。
2. 打开编译器优化。
3. 应用正确的内存高速缓存设置。
4. 使用 SDK 中提供的现有经优化的库。
5. 手动优化任何不存在的自定义模块。

内容

1 分析现有算法 2
1.1 使用 CPU 周期计数器测量周期数 2
2 编译器优化 3
2.1 使用生成的汇编代码验证编译器优化 3
3 仔细检查内存高速缓存设置 4
4 使用现有的经优化的库 6
5 优化自定义内核/模块 6
6 示例 7
6.1 示例系统结果汇总 7

商标

所有商标均为其各自所有者的财产。

English Document: SDAA248
Copyright © 2026 Texas Instruments Incorporated

1 分析现有算法

建议对相关算法进行性能分析，以确定是否需要进行优化，并帮助确定在每个步骤中所做的改进。分析算法本质上是确定循环迭代使用多少个时钟周期。为此，算法或代码必须能够在 C7x 内核上运行。在 FreeRTOS SDK/MCU Plus SDK 中，提供了几个示例。最快的方法是从 C7x 内核的空或 hello world 示例工程开始，然后复制现有的算法。如需示例，请参阅提供的软件示例代码。可以使用包含静态数据的数组来简单地分析算法，然后在稍后添加外部外设/实际数据。主要目标是获得用于评估性能指标的起点。

C7000 C/C++ 优化指南包含一个有关如何分析代码的部分，并提供了四种不同的方法。进行分析最简单的方法是使用 C7x 编译器内在指令来获取 CPU 计数器（TSC 寄存器）的当前值。CPU 周期计数器是一个简单的向上计数器，其增量为一个时钟周期。这是一个 64 位寄存器，因此溢出的风险非常小，当 DSP 核心运行在 1GHz 时，以 ns 为分辨率，它允许系统运行约 585 年而不会发生溢出。利用 CPU 周期计数器（TSC 寄存器）可以捕获用于算法各部分的时钟周期数，并提供有关优化改进的有用见解。

建议多次（至少三次）迭代运行该算法，以便让高速缓存预热，因为冷缓存命中是最慢的/最差情况下的性能表现，而在实际运行的系统中，经过几次迭代后，高速缓存就会处于预热状态。

1.1 使用 CPU 周期计数器测量周期数

C7x 提供一个 64 位 CPU 周期计数器，增量为一个时钟周期，该计数器可使用 __TSC 内在函数进行读取。要测量函数的周期数，请记录执行前后的计数器值并取差值。

```
#include <c7x.h>
#include <stdio.h>
uint64_t start, end, elapsed;
/* warm up cache */
for (int i = 0; i < 3; i++)
{
    myAlgorithm(input, output);
}
/* Measure one iteration */
start = __TSC;
myAlgorithm(input, output);
end = __TSC;
elapsed = end - start;
```

若要将周期数转换为实时值，请除以 C7x 内核频率。在 1GHz 时，一个周期等于一纳秒。实际频率取决于器件等级和时钟配置——请参阅 SBL 日志和器件数据表进行确认。

```
time_us = elapsed_cycles / C7x_freq_MHz;
time_ms = elapsed_cycles / (C7x_freq_MHz × 1000);
```

请注意，C7x_freq_MHz 不是实际变量，您需要提供实际值。这是测量算法执行时间并分析代码的最简单方法。可以进一步细化，测量算法中各个独立函数的执行时间，从而进一步分析算法将大部分计算周期花在了哪里。

English Document: SDAA248
Copyright © 2026 Texas Instruments Incorporated

2 编译器优化

提高性能的第一步也是减少所需的工作量，即让 C7x 编译器在可能的位置优化代码。在 Code Composer Studio (CCS) 工程中，打开工程设置，然后检查“工具”、“C7000 编译器”、“优化”下使用的优化级别。

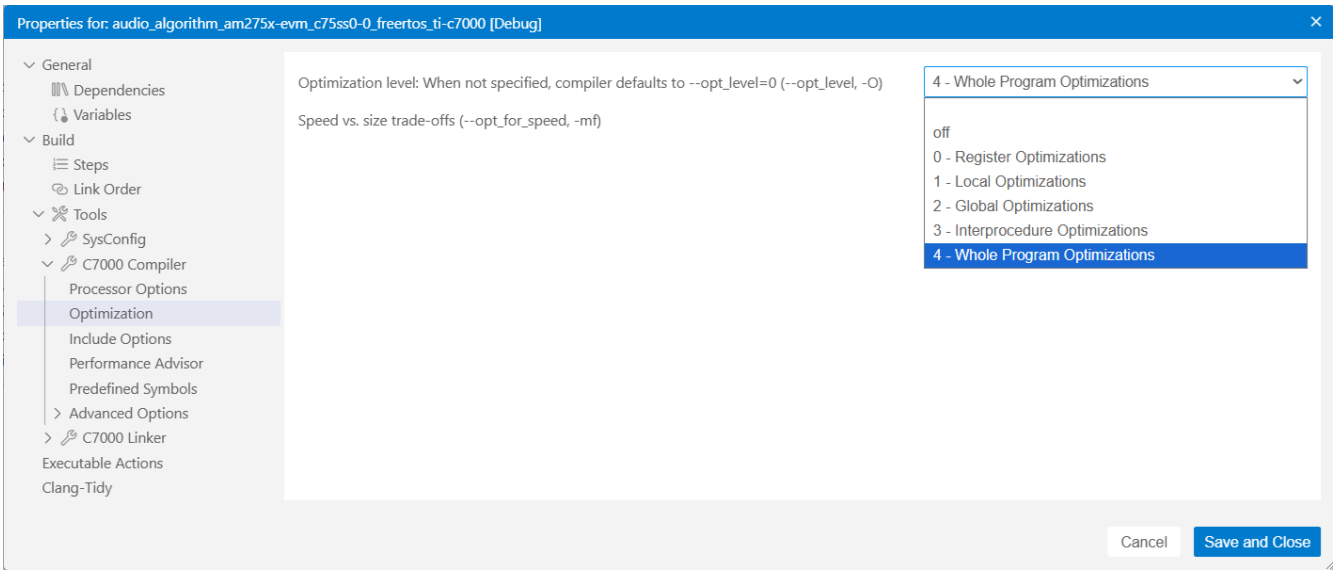


图 2-1. 设置优化级别

对于优化级别，该数字越高（0 到 4），应用的编译器优化就越多，但会使调试的容易度为代价。通常建议优化级别为 3 或更高，以便在不进行大量手动优化的情况下提供优化现有代码的理想机会。速度大小的权衡用于告诉编译器：是优先考虑应用程序代码的内存占用（大小，设为 0），还是优先考虑最快的运行时间（速度，设为 5），或者是介于两者之间。在调试期间，这些优化通常处于关闭状态，让用户可以轻松地调试程序，但性能会因此而降低。随着编译器对代码的优化程度越高，单步调试代码或查看变量就变得越困难，因为有时这些内容会被完全移除。有关这些选项的详细信息，请参阅 *C7000 C/C++ 优化指南*。

2.1 使用生成的汇编代码验证编译器优化

设置优化级别后，建议确认编译器是否真正对代码的内层循环进行了流水线和矢量化处理。设置高优化级别通常会告知编译器优化代码，但有时可能会失败。

软件流水线是 C7x 编译器的一种技术，用于重叠循环的连续迭代，以使 DSP 的多个功能单元同时处于繁忙状态。在没有流水线的情况下，处理器会在开始下一个循环迭代之前执行一次完整的循环迭代，在任何给定周期使大多数功能单元处于空闲状态。在流水线中，编译器将迭代 N+1 与迭代 N 重叠，以便不同迭代的不同部分同时执行。

矢量化利用 C7x 的宽矢量数据路径——AM27x 系列和 AM62D 上的 256 位。矢量化循环会同时处理多个值，而不是每条指令只操作一个值。例如，在 AM27x 系列上，一个处理 32 位浮点值的循环每个周期可以处理八个值。汇编中的矢量指令可通过 V 前缀（例如 `VFADDSP` 和 `VFMPYSP`）识别，而标量等效项没有前缀。

两个编译器选项可让您了解编译器的功能：

1. `--gen_opt_info=2`（或 `-on2`）会随编译的对象一起生成 .nfo 文件。此文件包含应用于每个循环的高级优化的可读摘要，以及无法应用优化时的建议。这是诊断问题的最快起点。
2. `--keep_assembly` 或使用 `--src_interlist` 会保留生成的 .asm 文件，其中包含每个循环的详细软件流水线信息注释块，显示达到的启动间隔、资源使用情况及未应用流水线的原因。添加 `--debug_software_pipeline` 会在这些注释块中提供更详细的信息。

这两个选项都可以在 CCS 中设置：依次进入工程属性 > 工具 > C7000 编译器 > 高级选项（`--debug_software_pipeline` 位于“运行时模型”下，`--gen_opt_info` 位于“补充信息”下），或者直接在

English Document: SDAA248
Copyright © 2026 Texas Instruments Incorporated

makefile 中添加编译器标志。典型的工作流程是从. nfo 开始创建高级视图，然后在需要时打开 .asm 以进行更深入的分析。启用 `--debug_software_pipeline` 后，会自动使用 `--keep_asm` 选项。

当循环无法进行流水线或矢量化时，最常见的原因和修复方法是：

1. 循环中的函数调用

   如果编译器无法内联，循环内部的函数调用会阻止软件流水线。若要解决此问题，请通过以下方式之一定义被调用的函数：

   - 在调用所在的源文件中使用“inline”（内联）关键字
   - 在通过 #include 包含的 .h 头文件中，使用静态内联功能

   注意：编译器在 `--opt_level=3` 和 `--opt_level=4` 执行自动内联，因此，一些调用可以在更高的优化级别进行内联而无需手动更改。
2. 未知行程计数

   向编译器提供有关循环执行次数的信息有助于确定矢量化和循环优化是否有益，以及冗余循环是否必要。

   软件流水线循环需要最少次数的迭代才能合法执行。如果编译器无法保证实际迭代计数始终满足此阈值，则会生成两个版本的循环：快速流水线版本和较慢的冗余回退。汇编注释块将在发生此情况时说明 `;* Redundant loop generated` 并指定最小安全迭代计数。

   使用 `#pragma MUST_ITERATE` 提供已知的交叉边界：

   ```
   #pragma MUST_ITERATE(128, 128, 8)
   for (int i = 0; i < n; i++) {
       ...
   }
   ```

   这个最小值取决于应用本身：如果算法总是处理固定的数据块大小，则该大小就是保证的迭代最小值。例如，处理每个块 128 个样本的音频算法将始终执行循环至少 128 次。添加 pragma 后，请检查汇编代码以确认 `;* Redundant loop generated` 不再存在。请注意，提供不正确的值将导致编译器生成可能产生错误行为的代码。
3. 指针别名/循环携带依赖

   如果编译器无法确定输入和输出数组占用单独的存储器区域,则必须保守地假设迭代 i 中的存储可能会影响迭代 i+1 中的加载。这会产生阻止软件流水线的循环携带依赖。若要解决此问题，请对保证不重叠的指针参数使用“restrict”（限制）关键字。

   ```
   void weighted_sum(int * restrict a, int * restrict b, int * restrict out, int weight_a, int
   weight_b, int n)
   ```

   备注

   （在重叠的指针上）错误地使用“restrict”（限制）可能会导致编译器生成具有未定义行为和不正确结果的代码。
4. 循环未矢量化

   编译器可能无法完全对循环进行矢量化，或者可能只对其中的一些操作进行矢量化，从而导致低效混合循环，其中一些指令是矢量指令，而一些指令是标量指令。

   - 如果循环根本没有矢量化，请考虑直接在源中使用 C7x 矢量类型
   - 如果问题出在部分矢量化上，可能有必要使用矢量类型和内联函数手动将整个循环矢量化。

   有关更多信息，请参阅 C7000 C/C++ 优化指南 (SPRUIG8)。

3 仔细检查内存高速缓存设置

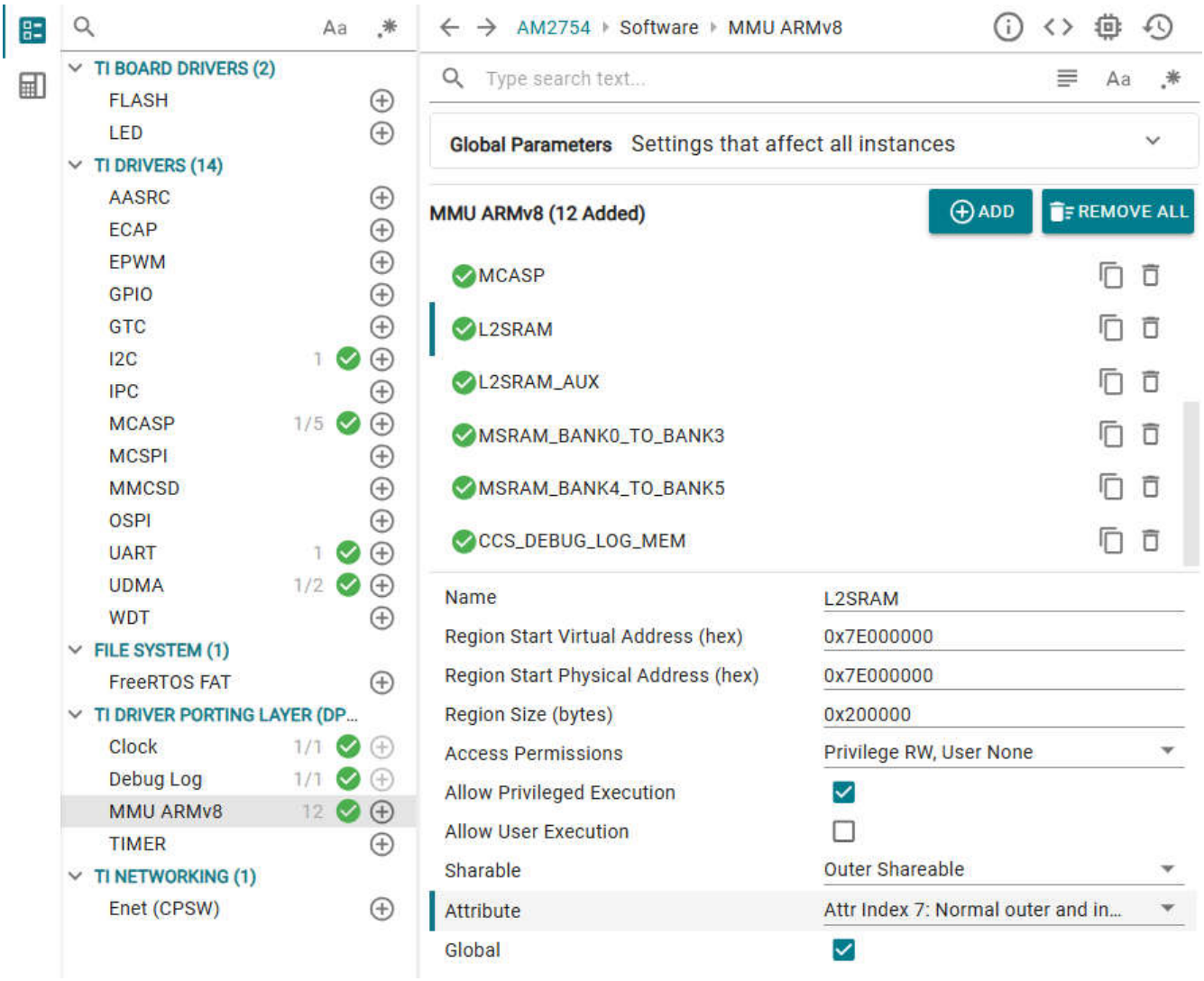
使用写回高速缓存设置而不是直写时，可实现最佳性能。示例工程默认就是按这种方式配置，但如果高速缓存配置错误，性能可能会发生显著变化。

English Document: SDAA248
Copyright © 2026 Texas Instruments Incorporated

总结一下差异：高速缓存写回模式（有时称为延迟写入）是指向某地址写入数据时，仅写入高速缓存（使原始内存位置处于不同步状态），直到该数据被从高速缓存中逐出时，缓存的内容才被复制回原始内存位置。由于高速缓存通常比其他形式的内存更快，DSP 能够更快地写入和读取此数据。这种情况的风险是，如果内存在多个内核之间共享，则必须将数据写回内存位置，以便另一个内核可以看到有效数据。高速缓存写回调用只能用于在需要时强制执行此操作，以便仍然使用高速缓存的改进性能。

高速缓存直写模式不同，在这种模式下，对内存位置的任何写入都会同时在高速缓存和原始内存位置中更新。这会对写入性能产生负面影响，因为应用程序必须等待较慢的内存完成写入。从高速缓存中仍会进行读取，但写入速度较慢，因为每次写入时，高速缓存和内存位置都保持同步。

在 CCS 工程中，有一个包含所使用的硬件设置的 SysConfig 文件（以 .syscfg 结尾）。在 SysConfig 文件中，有一个名为 *MMU* 或 *MMU ARMv8* 的部分。此部分包含映射出的所有内存位置。



单击以选择一个部分（通常是 L2SRAM 或 MSRAM 部分用于存储数据），并检查属性字段是否设置为可写回高速缓存（属性索引 7）。在某些特殊应用中，可能需要进行其他设置。例如，使用写回时，单个内核使用的内存可缓存到处理器高速缓存中。对于可写回高速缓存，当数据更新时，数据只会在高速缓存中更新，而不会更新到原始内存位置，直到该高速缓存行被从缓存中清除。这可能会造成这样一种情况：处理器对某个缓冲器进行了一些修改，而这些修改仅存在于高速缓存中，并未写入原始内存位置。这可能会导致从原始内存位置拉取过时数据（例如，在算法更新数据后，用于 MCASP 事务）。当对内存位置的更新完成并且必须同步回另一个硬件外设或处理器的原始位置时，应用需要调用 `CacheP_inv()` 函数来强制进行高速缓存写回，以便使用高速缓存中的值更新原始内存位置。

English Document: SDAA248
Copyright © 2026 Texas Instruments Incorporated

4 使用现有的经优化的库

作为 FreeRTOS SDK/MCU Plus SDK 的一部分，提供了几个高度优化的库，可用于进一步提高性能。这些库通过已针对 C7x 架构进行优化的代码提供常用函数。有用于数学、数字信号处理、视觉处理和音频处理的库。

利用这些经过优化的函数可以为算法提供进一步改进的性能。请注意，C7x 架构基于矢量，允许同时对多个数据块进行操作。利用矢量架构，性能通常会随着数据大小（块大小）的增加而提高。

这些不同库的文档位置位于 SDK 文件夹内。例如 `<SDK base directory>/source/dsplib/docs/`。还为提供的内核提供了示例代码。通常建议尽可能使用这些优化的 DSP 函数，因为它们经过优化，可利用 C7x 架构的多种特性（例如流引擎和矢量），这些特性可以显著提高算法中执行实际计算的各部分的性能。

关于这些优化函数，需要注意的关键一点是，它们通常与 init() 和 exec() 函数配合使用。init 函数用于设置一次性操作，例如流引擎或地址生成器。exec 函数用于执行计算，应在循环中调用。init 函数只能在参数更改时调用一次。理想情况下，init 函数被调用一次，exec() 被循环，直到完全处理完数据块。这提供了最大的性能提升。

除了利用提供的库外，还必须正确使用它们以提高性能。例如，如果用于将两个数组相加的 ***add*** 函数一次仅处理一个元素，则不会提高性能。对于具有 256 位矢量宽度架构的 AM275，它能够在单个时钟周期内将多达八个 32 位值相加，但必须正确调用该函数来处理矢量宽度数据，而不是单标量数据。库中提供的示例展示了如何正确调用这些函数。

5 优化自定义内核/模块

在完成上述所有步骤后，最后的优化步骤之一是迁移任何目前尚未包含在已提供库中的自定义模块。为此，需要了解 C7x 架构和其他外设，以提高性能。如需更多详细信息，请参阅 *C7000 C/C++ 代码优化指南：基础代码优化策略与技术*，以及随后的*高级技术*。有几种在线资源可帮助您熟悉 C7x 架构，例如*视频培训系列*。

English Document: SDAA248
Copyright © 2026 Texas Instruments Incorporated

6 示例

使用前面概述的步骤，我们可以查看所提供的示例算法（参阅 GitHub 上提供的示例代码）并完成该过程。我们将使用一个简单的示例算法，该算法利用了几个内核。

示例算法是一种简单的音频算法。以下是系统参数：

1. 8 个音频通道 (48kHz)、16 位样本 (Q15)、交错数据
2. 块大小：每通道 128 个样本（总共 1,024 个样本）
3. 输入和输出样本将是有符号的 16 位样本 (Q15)

在此虚数系统示例中，将使用一个简单的算法，具体将执行以下操作

1. 将 16 位样本转换为浮点数
2. 三级级联双二阶低通滤波器
3. 应用 0.75 的全局增益
4. 转换回 16 位样本以与 DAC/放大器配合使用

该示例系统将从最基础的“原生 C”代码开始，这种代码没有任何针对 DSP 的优化，我们将使用 AM275 EVM 作为测试平台，逐步执行上述的每一个步骤，并观察 C7000 上的性能提升情况。

6.1 示例系统结果汇总

回顾每一步中的性能提升可以看出，花一些时间为 C7000 DSP 优化代码可以显著提高性能。

表 6-1. 影响编译器优化标志

代码类型	-O 标志	总计周期数	Q15 至悬空周期	级联双二阶周期数	增益周期数	悬空至 Q15 周期数	每个样本的周期数	备注
原生 C	0	522,142	18,459	446,010	14,364	43,238	509.9	
	1	249,780	16,412	178,789	13,337	41,171	243.9	
	2	180,145	2,443	150,692	284	26,670	175.9	
	3	180,146	2,443	150,592	284	26,671	175.9	
	4	177,840	241	150,694	176	26,667	173.7	
C7 DSPLib	0	3,602	185	2,981	184	137	3.5	
	1	3,591	185	2,981	184	137	3.5	
	4	3,579	185	2,981	184	137	3.5	

总结：从上述数据可以看出，编译器优化标志 (-O) 的效果显著提升了算法所有模块的性能，在某些情况下能够匹配优化的 DSPLib 内核的性能。这是因为编译器能够查找图形并设置矢量宽度计算，从而显著提高速度。

在所有内核模块都替换为手动优化的 DSPLib 内核后，性能显著提高，优于最佳原生 C 代码。使用优化后的 C7000 库时，性能从每个样本 173.7 个时钟周期提升到了每个样本 3.5 个时钟周期。

English Document: SDAA248
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