User's Guide

到 C29 CPU 的应用软件迁移

摘要

本指南旨在提供将应用软件从 C28 CPU 迁移到 C29 CPU 以及从 CLA 迁移到 C29 CPU 时，需要重点处理的问题的相关信息。

内容

表格清单

商标

1 简介

本指南旨在提供将应用软件从 C28 CPU 迁移到 C29 CPU 以及从 CLA 迁移到 C29 CPU 时，需要重点处理的问题的相关信息。

2 C28 到 C29 的 CPU 迁移

2.1 用例

这适用于从 F28x 平台 (C28 + CLA) 迁移到 F29x 平台 (C29) 的用户。

2.2 主要差异

表 2-1 总结了 C28 和 C29 编程模型之间的主要差异

表 2-1. C28 与 C29 CPU 编程模型

项	C28 CPU	C29 CPU
每个时钟周期发出指令	单通道	VLIW - 多个（多达 8 个）
流水线级	8	9
存储器映射	分散	连续
8b 字节可寻址性	否	是
寄存器	8 x 64 位浮点 （具有双精度 FPU64） 8 x 32 位浮点（具有单精度 FPU） 3 x 32 位定点（ACC、P、XT） 8 x 32 位寻址（SP、DP）	16 x 64 位浮点 （32 位浮点寄存器对） 8 x 64 位定点 （32 位定点寄存器对） 16 x 32 位寻址
堆栈范围	16 位栈指针 (SP)	32 位栈指针 (A15)
指令宽度	16 位、32 位	16 位、32 位、48 位
总线宽度	1 x 32 位数据读取（32 位地址） 1 x 32 位数据写入（32 位地址） 1 x 32 位程序读取（22b 地址）	2 x 64 位数据读取（32 位地址） 1 x 64 位数据写入（32 位地址） 1 x 128 位程序读取（32 位地址）

2.3 源代码迁移

本节介绍了将源代码从 C28 迁移到 C29 CPU 时的关键问题。它还讨论了可帮助用户进行迁移的工具。

2.3.1 C/C++ 源代码

移植平台时的关键问题之一是维护存储器访问顺序，因为 C28 和 C29 CPU 流水线具有相同的步骤序列。但是，下面讨论了一些影响 C/C++ 源代码的主要差异。C28 编译器支持 C29 编译器所不支持的一些 C/C++ 扩展。

2.3.1.1 pragma 和属性

- clang 支持的目标无关 pragma 和属性很可能是可移植的，而许多特定于 C28 的 pragma 和属性则不可移植。有关详细信息，请参阅编译器用户指南。
- C29 编译器有一个名为 c29clang-tidy 的工具，用于检查 C28 pragma 的使用情况并提供替代方案建议（如有）。此处对该工具进行了讨论（在 c29migration-c28-pragmas 下）
- C29 编译器用户指南还分别在此处和此处讨论了 pragma 和属性

2.3.1.2 宏

- C29 编译器中不支持几个特定于 C28 的预定义宏。
- C29 编译器用户指南在此处和此处讨论了宏及其迁移。

2.3.1.3 内联函数

- C28 内联函数是映射到一条或多条 C28 汇编指令的内置函数。有些 C28 内联函数具有功能上等效的 C29 内联函数，但许多 C28 内联函数没有功能上等效的 C29 内联函数。

- C29 编译器有一个名为 c29clang-tidy 的工具，用于检查 C28 内联函数的使用情况并提供替代方案建议（如有），此处对该工具进行了讨论（在 c29migration-c28-builtins 下）
- C29 编译器用户指南还在此处和此处讨论了内联函数

备注

当编译器设置适当 (--fp_mode=relaxed) 时，C28 编译器会将多个 RTS 库调用映射到 TMU 指令。此优化尚未在 C29 编译器中实现。目前，建议用户使用与 TMU 指令相对应的 C29 内联函数来高效地实现此类运算。

2.3.1.4 内联汇编

- 内联汇编 - 如果用户 C 代码包含内联汇编，则用户需要手动更新此代码，因为 C28 和 C29 指令集是不同的。

2.3.1.5 关键字

- 关键字 - C28 编译器支持所有标准的 C89 关键字，包括 const、volatile 和 register。它支持所有标准的 C99 关键字，包括 inline 和 restrict。它支持所有标准的 C11 关键字。它还支持 TI 扩展关键字 __interrupt、__cregister 和 __asm。其中一些关键字（如 __interrupt、__cregister）不会移植到 C29。

  C29 编译器的 c29clang-tidy 工具不支持关键字检查。如果不支持某个关键字，C29 编译器会生成错误。用户需要手动处理此类关键字。如果适用，可通过 #ifdef 将其定义为空，如下所示。

```
#ifdef __c29__
# define CREGISTER
#elif defined(__TMS320C2000__)
# define CREGISTER __cregister
#endif
```

2.3.1.6 数据类型差异

数据类型差异 - 表 2-2 中总结并突出显示了此上下文中的主要差异。

表 2-2. 数据类型主要差异

类型	C28	CLA	C29	ARM
char	16	16	**8**	8
short	16			
int	16	32	**32**	32
long	32			
long long (COFF)	64	32	不适用	64
long long (EABI)	64			
float	32			
double (COFF)	32	32	不适用	64
double (EABI)	64			
long double (COFF)	64	32	不适用	64
long double (EABI)	64			
指针	32	16	32	32

用户需要特别注意数据类型：

1. int（C28 16 位与 C29 32 位）- 如果用户代码出现“int”，建议使用以下方法进行迁移：
   a. 将用户代码中出现的所有“int”更改为固定宽度类型 int16_t。这可确保代码中没有任何中断，并且保留了原始数据宽度。
      i. C29 编译器有一个名为 c29clang-tidy 的工具，用于检查是否出现“int”和“unsigned int”，此处对该工具进行了讨论（在 c29migration-c28-int-decls 下）
   b. 理想场景 - 用户代码中未出现任何“int”，仅包含可移植的固定宽度类型，如 int16_t、int32_t。在这种情况下，不进行任何更改即可移植代码。
2. char（C28 16 位与 C29 8 位）- 如果用户代码出现“char”，或者出现“int8_t”或“uint8_t”：
   a. 一开始，将所有出现的“char”更改为“int16_t”可能就像迁移方法那样，因为它会保留用户原始代码中存在的位宽。但这可能会导致意外的构建时问题，更糟糕的是，会导致难以检测运行时故障。这是因为，不同类型的指针混叠是不合法的。但是 char 比较特殊，可与其他指针类型混叠。换句话说，可以通过指向 char 的指针访问任何对象。如果现在将 char 更改为 int16_t，就会违反 char 独自拥有的这一“特权”。
   b. 因此，建议的迁移方法是不对代码进行任何更改，或者最好是将用户代码中出现的所有“char”都更改为“int8_t”。即便是这种方法，也会导致出现一些问题：
      i. 如果用户代码中出现“char”，但实际存在 16 位依赖关系，则需要将其更改为 int16_t。例如，分配的数据超出 8 位范围，需要 16 位范围。C29 编译器有一个名为 c29clang-tidy 的工具，用于检查对“char”类型表达式的运算是否超出了 8 位类型的范围，此处对该工具进行了讨论（在 c29migration-c28-char-range 下）
      ii. 如果用户代码包含指向“char”的指针和相关偏移，或针对此类指针的算术运算。C29 编译器有一个名为 c29clang-tidy 的工具，用于检查基于 char/int 的指针的指针算术运算，其位跨度在 C28 到 C29 之间变化，此处对该工具进行了讨论（在 c29migration-c28-types 和 c29migration-c28-suspicious-dereference 下）。

备注

请注意，将“char”更改为“int8_t”后，指针混叠不是问题。

---

3. 数据类型差异导致的 sizeof() 差异 - 表中对此进行了总结，这些差异会导致使用标准库函数调用时出现行为差异：

	sizeof(char)	sizeof(short)	sizeof(int)	sizeof(long)
C28	1	1	1	2
C29	1	2	4	4

   a. 如果使用硬编码大小，C29 编译器有一个名为 c29clang-tidy 的工具，用于检查没有 sizeof() 表达式的库调用，此处对该工具进行了讨论（在 c29migration-c28-stdlib 下）。
   b. 逐字节处理值的函数（如 memset）不会从 C28 移植到 C29。如果 memset 与 sizeof(int)、sizeof(int16_t)、sizeof(char) 或 sizeof(int8_t) 一起使用，则 C28 和 C29 的行为不同。请考虑以下 memset 示例。一些受影响的函数包括 memset、memccpy、memchr、strncmp。

```
memset(buf,5,2 * sizeof(char));
Byte address offset at buf: 0 8 16 24 32 40 48 56 64
C28:                        5    5
C29:                        5 5
```

```
memset(buf,5,2 * sizeof(short));
Byte address offset at buf: 0 8 16 24 32 40 48 56 64
C28:                        5    5
C29:                        5 5  5 5
```

```
memset(buf,5,2 * sizeof(int));
Byte address offset at buf: 0 8 16 24 32 40 48 56 64
C28:                        5    5
C29:                        5 5  5 5   5 5   5 5
```

其他函数（如 memcpy）也会出现行为差异，即使它们不是按字节操作也是如此。请考虑以下 memcpy 示例。

```
memcpy(dst,src,4 * sizeof(char));
Byte address offset at dst: 0 8 16 24 32 40 48 56 64
C28:                        1 2  3 4   5 6   7 8
C29:                        1 2  3 4
```

注意 memcpy 的使用方法，为了在 C28 和 C29 上获得相同的行为，可以使用 CHAR_BIT（在 limits.h 中定义）。

```
#if(CHAR_BIT == 16)
memcpy(dst,src,4 * sizeof(char));
#endif
#if(CHAR_BIT == 8)
memcpy(dst,src,4 * sizeof(char) * 2);
#endif
Byte address offset at dst: 0 8 16 24 32 40 48 56 64
C28:                        1 2  3 4   5 6   7 8
C29:                        1 2  3 4   5 6   7 8
```

备注

使用 C 头文件 stdint.h 可以移植固定宽度类型。

2.3.1.7 迁移工具支持

如上所述，C29 编译器有一个名为 c29clang-tidy 的实用程序，用于帮助将 C/C++ 源代码从 C28 迁移到 C29。此处对该实用程序进行了讨论。

2.3.2 汇编语言源代码

由于指令集不同，使用 C28 汇编语言编写的任何用户代码都不能移植到 C29。使用 C 编译器时，C29 VLIW 架构的性能保障是可以预期的，因此强烈建议用户不要编写 C29 汇编代码。对于已编写 C28 汇编器结构并正在 C29 工具链中寻找等效结构的用户，请参阅此处。

有关更多信息，请参阅此处的 *C29 Clang 编译器工具用户指南* 中的相关主题。

2.4 工具链迁移

本节介绍了 C28 和 C29 工具链之间的大致差异，并指向相关文档。

2.4.1 编译器

C28 编译器和 C29 编译器使用不同的底层基础设施和完全不同的源代码基础。

- C28 编译器是 TI 专有的，而 C29 编译器基于 LLVM-clang。编译器选项完全不同，需要进行更改。选项含义背后的许多概念也发生了变化，例如出于优化考虑。可在此处获取 *C29 Clang 编译器工具用户指南*。它包含一个详细介绍迁移的章节，请参阅此处。
- C29 编译器仅支持 EABI 输出格式，而 C28 编译器既支持 COFF，也支持 EABI。从 C28-COFF 迁移到 C29 的用户应首先从 COFF 迁移到 EABI，可参阅此处提供的文档。

备注

C29 CPU 采用 VLIW 架构，相较于 C28 和 CLA，其并行性显著提高。然而，要使 C29 编译器充分利用 C29 中的这些并行功能单元，用户必须使用更高级别的优化（例如 -o2、-o3）。

2.4.2 链接器

工具链的链接器仍然是 TI 链接器，而不是 LLVM 链接器。这意味着，通常情况下，链接器命令文件的语法不会改变。但是，编译期间指定链接器命令文件的方式发生了变化。本页包含有关 LLVM 中链接器标志的详细信息。有关迁移链接器命令文件以用于 C29 编译器的信息，请参阅此处。

以前:

```
cl2000 a.c -z -llnk.cmd
```

之后：

```
c29lang a.c -Wl,-llnk.cmd
```

- 在 C28 编译器中实现了一些特性，这些特性与 TI 链接器进行交互，并在链接器命令文件中包含独特的规范。如果这些特性适用于 C29，将在编译器的未来版本中添加这些特性，例如 C28 编译器支持的实时固件更新 (LFU)。

2.4.3 CCS 工程迁移

基于 C29 的器件仅在 CCS Theia IDE 上受支持，在 CCS IDE 上不受支持；而基于 C28 的器件在 CCS IDE 上受支持。有关从 C28 编译器选项迁移到 C29 编译器选项的信息，请参阅此处。有关工程迁移的其他信息，请参阅 C29 SDK 的 docs 文件夹。

3 CLA 到 C29 的 CPU 迁移

3.1 用例

这适用于从 C28+CLA 器件迁移到 C29 器件的用户。在这种情况下，需要从 CLA 迁移到 C29。

备注

本节基于不包含 CLA 的现有 F29x 器件。

3.2 主要差异

表 3-1 中总结了 CLA 和 C29 编程模型之间的主要差异。

表 3-1. CLA 与 C29 编程模型差异

项	CLA	C29
每个时钟周期发出指令	单通道	VLIW - 多个（多达 8 个）
流水线级	8	9
存储器访问	仅限 LSRAM	RAM 和闪存
8b 字节可寻址性	否	是
寄存器	4 x 32 位浮点 2 x 16 位辅助	16 x 64 位浮点 （32 位浮点寄存器对） 8 x 64 位定点 （32 位定点寄存器对） 16 x 32 位寻址
堆栈范围	16 位栈指针 (SP)	32 位栈指针 (A15)
指令宽度	32 位	16 位、32 位、48 位 位

表 3-1. CLA 与 C29 编程模型差异（续）

项	CLA	C29
总线宽度	1 x 32 位数据读取（32 位地址） 1 x 32 位数据写入（32 位地址） 1 x 32 位程序读取（16 位地址）	2 x 64 位数据读取（32 位地址） 1 x 64 位数据写入（32 位地址） 1 x 128 位程序读取（32 位地址）

3.3 源代码迁移

本节讨论了从 CLA 到 C29 的源代码迁移。

3.3.1 C/C++ 源代码

考虑以下事项有助于更好地理解一些关键问题

- 现有的 C28+CLA 开发，以及如何开发系统代码以在 C28 CPU 和 CLA 上运行。
- 是与 C28 代码迁移到同一 C29 CPU，还是迁移到不同的 C29 CPU。两种方案各有优势和挑战。

1. 从 CLA 迁移到 C29 时：
   a. 需要将 .cla 文件转换为 .c 文件
   b. CLA 任务需要映射到 C29 CPU 上的中断。
      i. 如果可能，请使用 RTINT 而不是 INT 来保存硬件上下文并实现出色的性能
      ii. 由于 CLA 任务会运行至完成，而不会被抢占，因此为了实现类似的功能，可将中断分配到一个组中。给定组内的中断不会抢占或嵌套进该组内的其他中断。有关更多详细信息，请参阅 F29x 器件特定技术参考手册中的 *PIPE* 一章。
      iii. CLA 后台任务是可中断的，所以如果该任务已启用，它应位于比其余 CLA 任务相对应的组更低的嵌套组中。用户可以考虑的另一个选项是将后台任务功能移至 main() 中的空闲循环。
2. 将消除 LSRAM 中的代码和数据放置约束。对于 CLA，代码和数据都必须驻留在 LSRAM 中。在 C29 器件上，可以迁移 CLA 代码以在 LPAx RAM 上运行，并且数据可以驻留在 LDAx RAM 中。
3. CLA 编译器具有 C 语言的标准限制。*TMS320C28x 优化 C/C++ 编译器 v22.6.0.LTS 用户指南* 的编译器说明中提到了这些限制。例如，不支持定义和初始化全局/静态数据。不支持使用函数指针。当移至 C29 时，将会解除这些限制。
4. CLA 编译器支持 C28 pragma 和属性的子集。从 CLA 移植到 C29 时，需要解决这些问题。
5. 关键字 - 除了 C28 编译器支持的 2 个关键字（far 和 ioport）之外，CLA 编译器支持其他所有关键字。
6. 有关更多信息，请参阅编译器指南中讨论迁移 CLA 源代码的部分。

3.3.1.1 数据类型差异

1. 数据类型差异 - 表 3-2 中进行了总结和突出显示：红色表示 CLA-C29 差异，蓝色表示 CLA-C28 差异。
   a. 共享数据 - 在 C28 和 CLA 之间共享数据（共享结构）时，C28 和 CLA 之间的数据类型差异会产生影响。此处建议的方法是使用通过联合体的填充来求解指针或整数大小差异，如 CLA 软件开发指南中所述。例如，C2000Ware DigitalPower SDK 解决方案中针对 C28 和 CLA 之间共享的枚举（基于 int）使用了联合体。这些枚举定义了所使用的实验室、电路板状态等。由于 C29 器件不包含 CLA，不存在共享结构问题，因此从这个角度来看，上述数据类型差异不是问题。

b. 但是，将代码从 CLA 迁移到 C29 时，C29 和 CLA 之间的数据类型差异是相关的。这些问题及其解决方法与 C28-C29 数据类型差异中列出的项类似（请参阅节 2.3.1.6）。
   i. CLA（16 位）和 C29（32 位）之间存在指针大小差异。
   ii. C28 至 C29 源代码迁移部分介绍了“char”差异。C29 编译器的 c29clang-tidy 工具中关于 char 范围 (c29migration-c28-char-range) 和 char 指针算术运算 (c29migration-c28-types) 的检查器在此处也很有用。

表 3-2. 数据类型主要差异

类型	C28	CLA	C29	ARM
char	16	16	**8**	8
short	16			
int	16	32	32	32
long	32			
long long (COFF)	64	32	不适用	64
long long (EABI)	64			
float	32			
double (COFF)	32	32	不适用	64
double (EABI)	64			
long double (COFF)	64	32	不适用	64
long double (EABI)	64			
指针	32	16	**32**	32

3.3.1.2 迁移 CLAmath.h 函数和内联函数

1. CLA 编译器不支持 math.h。我们使用了一个单独的文件 CLAmath.h，其中包含对 C2000Ware CLAmath 库中函数/变量的外部引用。这些是用于特定操作（trig、div、sqrt、isqrt、exp、log）的手动优化 CLA 汇编例程。它还包含可将代码从 C28 移植到 CLA 的函数重新定义/映射，例如：
   a. 将特定的 math.h 函数和 TMU 内联函数映射到上述 CLA 数学库函数。例如，如果 C28 代码包含 __cos（对应于 TMU 指令）或 cosf，则它将在不进行任何更新的情况下迁移到 CLA，因为 CLAmath.h 将 __cos 和 cosf 映射到 CLAcos（在 CLAMath 库中）或 CLAcos_inline（在 CLAmath.h 中）。
      i. 因此，如果用户 CLA 代码包含 __cos，那么迁移到 C29 的过程与将 C28 内联函数迁移到 C29 的过程类似，如 C28-C29 源代码迁移部分所述。
         1. C29 编译器有一个名为 c29clang-tidy 的工具，用于检查 C28 内联函数的使用情况并提供替代方案建议（如有），此处对该工具进行了讨论（在 c29migration-c28-builtins 下）。该工具在这里也适用。
      ii. 如果它包含 cosf，则迁移到 C29 不需要进行任何更改。
      iii. 但如果它包含 CLAcos 或 CLAcos_inline，则用户需要将所有这些调用更改为 cosf。**我们计划了一个映射头文件来帮助迁移 CLAMath 函数。**
   b. 将特定的 math.h 函数和 C28+FPU 内联函数映射到 CLA 内联函数。例如，如果 C28 代码包含 __fmax（对应于 FPU 指令）或 fmaxf，则它将在不进行任何更新的情况下迁移到 CLA，因为 CLAmath.h 将 __fmax 和 fmaxf 映射到 __mmaxf32（CLA 内联函数）。
      i. 因此，如果用户 CLA 代码包含 __fmax，那么迁移到 C29 的过程与将 C28 内联函数迁移到 C29 的过程类似，如 C28-C29 源代码迁移部分所述。
         1. C29 编译器有一个名为 c29clang-tidy 的工具，用于检查 C28 内联函数的使用情况并提供替代方案建议（如有），此处对该工具进行了讨论（在 c29migration-c28-builtins 下）。该工具在这里也适用。
      ii. 如果它包含 fmaxf，则迁移到 C29 不需要进行任何更改。
      iii. 但如果它包含 __mmaxf32，则用户需要将所有这些调用更改为 fmaxf。**计划使用 c29clang-tidy 扩展来检查 CLA 内联函数。**
   c. 如果用户代码包含的 CLA 内联函数不映射到对应的 C28 内联函数（例如 __mgeq、__mgequ、__mgt、__mgtu、__mleq、__mlequ、__mlt、__mltu、__mdebugstop），则需要手动更新用户代码来修复这些函数。**计划使用 c29clang-tidy 扩展来检查 CLA 内联函数。**

3.3.1.3 将 C28 和 CLA 迁移到相同的 C29 CPU

1. 在 C28+CLA 实现中，CLA 任务可由硬件或软件触发，并且在任务完成时，系统可能会向 C28 发送中断。
2. 要在这种情况下实现此目的，对于软件任务触发器，将通过 PIPE 触发软件中断以运行所需的 ISR。
3. 对于硬件任务触发器，将设置 C29 CPU 的 PIPE 以从所需的外设进行触发。
4. 对于任务完成中断，通过 PIPE 触发软件中断（用户代码需要写入 PIPE 寄存器）来运行所需的 ISR。
5. 如果 C28+CLA 实现中存在后台任务，则该任务可在 C29 CPU 中作为后台循环（空闲循环）轻松实现。
6. 此场景中的一个关键挑战是中断优先级分配，因为 C28 ISR 和 CLA 任务现在都映射到 C29 ISR，需要分配中断优先级。而在 C28+CLA 上，它们在独立内核上独立运行；在这里，它们在相同的 C29 CPU 上运行。CLA 任务会一直运行到完成，而不会被其他任务抢占。如上所述，这可通过将这些任务分组到同一个 C29 PIPE 中断组来实现。但是，无法确保 C28 ISR 不会抢占 CLA 任务。同样，无法确保 CLA 任务不会抢占 C28 ISR。因此，在这种情况下，用户必须从整体上分析 C28 ISR + CLA 任务，并确定适合应用的中断优先级。
7. 同样，如果 C28 ISR 是软件触发 CLA 任务，并且它们现在驻留在同一 C29 CPU 上，就会产生每个任务的相对中断优先级问题。如果 CLA 任务的优先级变得更高，它便可抢占 C28 ISR，并从 CLA 端运行和提供预期的功能。不过，这意味着 C28 ISR 将暂停，从 C28 的角度来看，这可能并非预期的功能。
8. 此外，如果完全相同的事件（如外设）触发 C28 ISR 以及 CLA 任务，现在它们在同一 C29 CPU 上运行，则用户可以将它们合并为单个 ISR。
9. CLA 寄存器为用户提供了许多功能，例如能够知道正在运行哪个任务，以及能够通过写入寄存器中的特定位来停止一个任务。迁移时，由于缺少这些寄存器和对应的功能，可能需要进行适当的源代码更新。

备注

此处可能并未确定所有极端情况。

3.3.1.4 将 C28 和 CLA 迁移到不同的 C29 CPU

1. 需要不同的 CCS 工程。
2. 在 C28+CLA 实现中，CLA 任务可由硬件或软件触发，并且在任务完成时，系统可能会向 C28 发送中断。
3. 要在这种情况下实现此目的，您需要在两个 C29 CPU 之间使用 IPC。不过，IPC 在任一方向上最多有四个中断，而我们可能有八个 CLA 任务。因此，可在任一方向上使用一个 IPC 中断，并使用命令说明符来指示需要运行的任务或已完成的任务。这需要额外的代码来解析 IPC 中断并触发 ISR。
4. 因此，对于软件任务触发器，您要将它们映射到 IPC 中断，然后接收 C29 CPU 将运行 IPC ISR，并通过其 PIPE 触发软件中断（通过写入 PIPE 寄存器）来运行所需的 ISR。
5. 对于硬件任务触发器，将设置 C29 CPU 的 PIPE 以从所需的外设进行触发。
6. 对于任务完成中断，您需要将其映射到 IPC 中断，然后接收 C29 CPU 将运行 IPC ISR，并通过其 PIPE 触发软件中断来运行所需的 ISR。
7. 如果 C28+CLA 实现中存在后台任务，则该任务可在 C29 CPU 中作为后台循环（空闲循环）轻松实现。
8. CLA 寄存器为用户提供了许多功能，例如能够知道正在运行哪个任务，以及能够通过写入寄存器中的特定位来停止一个任务。迁移时，由于缺少这些寄存器和对应的功能，可能需要进行适当的源代码更新。

3.3.2 汇编语言源代码

由于指令集不同，使用 CLA 汇编语言编写的任何用户代码都不能移植到 C29。使用 C 编译器时，C29 VLIW 架构的性能保障是可以预期的，因此强烈建议用户编写等效 C 代码，并利用 C29 架构的并行性以及编译器提供的性能保障。

3.4 工具链迁移

C29 器件不包含 CLA，因此 C29 编译器不需要实现对应的 CLA 编译器。用户的 C28+CLA 工程可能包含 C29 工具链中不相关的 CLA 特定编译器选项。

CLA 的现有存储器模型需要针对特定数据类型（.bss_cla、.const_cla、.scratchpad、无堆等）使用特定段。用户的 C28+CLA 工程的链接器命令文件包含这些在 C29 工具链中不相关的段和映射。用户需要修改这些命令。

基于 C29 的器件将仅在 CCS Theia 上受支持，而基于 C28 的器件在 CCS IDE 上受支持。有关工程迁移的其他信息，请参阅 C29 SDK 的 docs 文件夹。

4 参考资料

1. C29 编译器用户指南
2. 德州仪器 (TI)：*TMS320C28x 优化 C/C++ 编译器 v22.6.0.LTS 用户指南*
3. 德州仪器 (TI)：*TMS320C28x CPU 和指令集参考*
4. 德州仪器 (TI)：*C2000 C29x CPU 和指令集参考指南* (SPRUIY2)
5. 德州仪器 (TI)：*F29H85x 和 F29P58x 实时微控制器技术参考手册* (SPRUJ79)

重要声明和免责声明

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、某特定用途方面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功能安全、信息安全、监管或其他要求。

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 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/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。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265