User's Guide

C7000 主机仿真用户指南

TEXAS INSTRUMENTS

摘要

C7000™ 主机仿真支持您在 PC 或 Linux® 主机系统上使用 C7000 编译器内在函数和 TI 向量类型。因此，在使用 C7000 编译器之前，您可以使用不同的调试工具和编程环境来为 C7000 硬件设计原型程序。主机仿真包不会尝试仿真 C7000 CPU。

内容

1 关于本文档

本文档是使用 C7000 主机仿真编写 C7000 DSP 程序的用户指南。其中包含的示例概述了使用 C7000 编译器 (cl7x) 进行编程和在所需主机系统上使用主机仿真包进行编程之间的主要差异。本文档旨在提供 C7000 主机仿真包的主要特性和限制的参考。

1.1 相关文档

以下文档提供 C7000 的相关信息：

- *C7000 C/C++ 优化编译器用户指南* (SPRUIG8)
- *C7000 C/C++ 优化指南* (SPRUIV4)
- *C7000 嵌入式应用程序二进制接口 (EABI) 参考指南* (SPRUIG4)
- *C6000 至 C7000 迁移用户指南* (SPRUIG5)
- *VCOP Kernel-C 至 C7000 迁移工具用户指南* (SPRUIG3)
- *C7x 指令指南* (SPRUIU4，可通过 TI Field 应用工程师获得)
- *C71x DSP CPU、指令集和矩阵乘法加速器* (SPRUIP0，可通过 TI Field 应用工程师获得)
- *C71x DSP Corepac 技术参考手册* (SPRUIQ3，可通过 TI Field 应用工程师获得)

1.2 免责声明

C7000 主机仿真支持是一种实验性产品。建议用户阅读并理解本文档中披露的所有限制。可能存在其他未在本文档中披露的限制。

1.3 商标

2 主机仿真入门

C7000 主机仿真包包含 C++ 源代码和头文件，用于驱动 C7000 编译器提供的功能。

在编译 C7000 程序之前可能需要在主机上构建源文件，具体取决于所需的主机。以下各节提供了有关如何在不同主机上构建源代码的详细说明。

为了充分理解本指南中的内容并成功使用主机仿真，需要熟悉 *C7000 C/C++ 优化编译器用户指南* (SPRUIG8) 和 C7000 运行时支持库。

2.1 系统要求

通常，C7000 主机仿真的系统要求与安装 C7000 代码生成工具 (CGT) 所需的系统要求一致。

C7000 主机仿真包附带的预编译库需要安装以下编译器。建议您使用与构建和测试 C7000 主机仿真所用版本相匹配的编译器版本。

- **Linux**（x86-64 位）
  - GNU g++ 编译器。C7000 主机仿真基于 9.5.0 版本构建。
- **Microsoft Windows®**（x86-64 位）
  - Visual C++ 编译工具（单独提供或与相应的 Visual Studio® IDE 安装包一起提供）。C7000 主机仿真基于 C++ 工具集 v143、编辑器版本 14.37 构建。
  - GNU g++ 编译器。C7000 主机仿真基于 MSYS2 版本 9.2.0 构建。

使用主机仿真时无需 Boost C++ 库和头文件。

2.2 安装说明

C7000 主机仿真包将作为 C7000 CGT 的一部分进行分发。在所需的平台上安装 C7000 CGT 也会安装 C7000 主机仿真包。

不同平台和编译器的库可以在已安装工具的 `host_emulation` 目录下找到。与主机仿真相关的所有头文件都可以在已安装工具的 `host_emulation/include` 目录下找到。

对于 Visual C++， *<target>*`-host-emulation.lib` 库与静态运行时库的*发布*版本兼容。 *<target>*`-host-emulationd.lib` 库与静态运行时库的*调试*版本兼容。

2.3 差异总结：主机仿真编码与原生 C7000 编码

编写应用程序代码以使用 C7000 主机仿真运行时，应注意以下一般限制：

- 所有源文件必须包含 `c7x.h` 文件。（请参阅节 3.1。）
- 为了在日后实现可移植性，建议使用标准整数类型而非内置类型。（请参阅节 3.2。）
- 由于底层实现严重依赖 C++14 构造和功能，因此代码必须使用 C++14。（请参阅节 3.2。）
- 主机仿真不支持 C7000 pragma。（请参阅节 3.2。）
- 内在函数存在某些限制和差异。（请参阅节 4。）例如，直接在存储器和 L1D 高速缓存上运行的内在函数不能用于 C7000 主机仿真。（请参阅节 4.4。）

有关特定编译器错误和警告的信息，以及 C7000 编译器和主机仿真编译器之间的语法解释差异，请参阅节 10。

3 一般编码要求

3.1 所需的头文件

无论选择何种主机，编写的每个程序都需要满足特定的先决条件，才能使用 C7000 主机仿真运行。

所有将 C7000 编译器功能与主机仿真一起使用的源文件都需要 #include `c7x.h` 或 `c6x_migration.h` 文件。这些文件依次包括所有其他所需的头文件。针对主机仿真进行编译时，请勿 #include C7000 运行时支持库中提供的任何其他头文件。

针对主机仿真进行编译时，请勿 #include C7000 运行时支持库中的任何头文件。其中包括 `c7x.h` 和 `c6x_migration.h` 文件。而是应使用预处理器符号来控制包含哪些头文件。

表 3-1. 主机仿真头文件

明确包含的文件	说明
c7x.h	主头文件。包括下面列出的所有其他文件（c6x_migration.h 除外）。
c6x_migration.h	传统内在函数和数据类型。包括下面列出的所有其他文件。
自动包括的文件	
c7x_cr.h	全局控制寄存器定义
c7x_ecr.h	全局扩展寄存器定义
c7x_luthist.h	查询表和直方图控制接口
c7x_strm.h	流引擎控制接口

`ti_he_impl` 文件夹包含用于实现的其他头文件；不应直接包含这些文件。

3.2 包依赖项

由于底层实现高度依赖于 C++14 结构和功能，因此为 C7000 主机仿真编写的程序必须使用 C++14 语言。

根据编译器的不同，编译命令中可能需要一个用于启用 C++14 支持的特殊标志。

虽然不是强制要求，但强烈建议在使用 C7000 主机仿真进行编程时使用标准整数类型（例如 `int32_t`）。使用内置数据类型可以编译和运行，但不能保证这些结果在所有平台上都是正确的。使用标准整数类型代替相应的内置类型将实现正确的结果，并且不会影响将程序转换到 C7000 编译器的能力。

使用主机仿真时，使用 C7000 编译器属性和指令将产生未定义的警告。这种行为是预期行为，无法纠正。如果程序在目标芯片上运行需要这些属性和指令，则通常可以在主机仿真编译器上抑制警告。

C7000 主机仿真包不仿真 C7000 编译器 pragma。因此，与 C7000 主机仿真一同运行的代码中使用时，C7000 编译器 pragma 将不起作用。

表 3-2 中提供了使用主机仿真时自动定义的 C7000 编译器符号的完整列表

表 3-2. C7000 预处理器符号

定义的预处理器符号	说明
__C7000__	如果针对 C7000 目标或任何类型的 C7000 主机仿真进行编译，则为已定义。
__C7100__	如果针对 C7100 主机仿真进行编译，则为已定义。
__C7120__	如果针对 C7120 主机仿真进行编译，则为已定义。
__C7504__	如果针对 C7504 主机仿真进行编译，则为已定义。
__C7X_HOSTEM__	如果针对主机仿真进行编译，则为已定义。在使用目标编译器 (cl7x) 时未定义此参数。
__little_endian__	默认情况下，已定义。

3.3 示例程序

以下是一个示例程序，可以交替使用主机仿真和 C7000 编译器进行编译，无需修改源代码。每种情况下都提供了示例编译器命令。

C7000 编译器 (cl7x) 命令行选项与主机仿真编译器不兼容。

```
/* Example Program test.cpp */
#include "c7x.h"
extern void test(int8 v);
int main()
{
    int8 vec1 = int8(1,2,3,4,5,6,7,8);
    int8 vec2 = (int8)5;
    test(vec1 + vec2);
}
```

C7100 主机仿真编译器命令 (Linux)：

```
g++ -c --std=c++14 -fno-strict-aliasing -I<cgt_install_path>/host_emulation/include/C7100
test.cpp -L<cgt_install_path>/host_emulation -lC7100-host-emulation
```

使用主机仿真时，-fno-strict-aliasing 命令行选项应始终与 g++ 一起使用。此选项可确保 g++ 编译器不会使用类型差异来做出别名决策。主机仿真实施方案使用不同类型来实现 TI 向量类型。因此，如果不使用此选项，g++ 可能会利用主机仿真功能错误地优化 TI 向量代码，这可能会导致意外的错误结果。

C7000 编译器命令：

```
cl7x test.cpp
```

4 内在函数

C7000 编译器提供的所有内在函数都可用于 C7000 主机仿真。以下各小节解决了在主机仿真中使用以下类型的内在函数时可能出现的问题：

- 类似 OpenCL 的内在函数（请参阅节 4.1）
- 用于为流引擎和流地址生成器编程的内在函数（请参阅节 4.2）
- 用于迁移为 C6000™ 编译器 (cl6x) 编写的旧代码的内在函数（请参阅节 4.3）
- 作用于存储器系统的内在函数（有关差异，请参阅节 4.4）

以下其他类型的内在函数对于 C7000 主机仿真和 C7000 编译器是相同的：用于向量和标量元素的特殊加载和存储的内在函数、低级别直接映射内在函数、作为寄存器接口向量谓词的一部分的内在函数，以及用于执行查询表和直方图运算的内在函数。

修改控制寄存器的内在函数在 C7000 主机仿真中执行此操作。C7000 主机仿真下可用的所有控制寄存器都可以随时作为无符号 64 位整数进行引用。

主机仿真并不能完全支持依赖于硬件信息的寄存器的读取和写入，例如执行模式和周期计数。虽然所有与这些寄存器关联的符号和内在函数都是为编译目的而定义，但在使用主机仿真时不能依赖它们的值以确保准确。

某些内在函数可能需要特殊处理才能正确使用。对于以下各小节中未提及的所有内在函数，其功能与 C7000 上的功能完全相同。可以在 `c7x.h` 文件和 C7000 运行时支持包中提供的其他头文件中找到可用于 C7000 编译器的内在函数完整列表。

指令执行可以尽可能接近地仿真硬件。

4.1 类似 OpenCL 的内在函数

C7000 编译器中可用的所有类似 OpenCL™ 的内部函数都可在 C7000 主机仿真中使用。内在函数接口保持不变；任何类似 OpenCL 的内在函数的合法使用在 C7000 主机仿真中也是合法的。

4.2 流地址生成器内在函数

C7000 编译器提供的所有流地址生成器内部函数也可在 C7000 主机仿真中使用。它们的接口与 C7000 编译器中的接口相同。

4.3 C6000 传统内在函数

`c6x_migration.h` 中定义的所有传统内在函数都可用于 C7000 主机仿真。它们的接口与 C7000 编译器中的接口相同。

节 8 讨论了有关旧数据类型的要求以及有关其 SIMD 使用方法的假设。由于这些限制，所有旧数据类型都必须被视为容器类型。也就是说，与旧数据类型的所有初始化和交互都必须通过内在函数进行。节 8 还包含有关如何在使用 C7000 主机仿真时使用旧数据类型和内在函数进行编程的示例。在 C7000 程序中使用 C6000 代码时，应随时使用 *C6000 到 C7000 迁移用户指南* (SPRUIG5) 和 `c6x_migration.h` 头文件作为参考。

4.4 存储器系统内在函数

与主机仿真搭配使用时，表 4-1 中列出的内在函数没有任何影响。这些内在函数在存储器和 L1D 高速缓存上运行，无法在主机系统上进行仿真。

表 4-1. 存储器系统内在函数

内在函数名称	实施说明
__memory_fence	成功执行且无任何影响
__memory_fence_store	成功执行且无任何影响
__prefetch	成功执行且无任何影响

5 TI 向量类型

C7000 主机仿真包通常允许按照与 C7000 编译器相同的方式使用 TI 向量类型（例如 `int16`），还支持布尔向量，例如 bool16。

但是，由于 C7000 主机仿真是用 C++ 编写的，因此存在一些限制。以下各节讨论并提供这些限制的示例。如果存在限制，可能需要更改用法和语法。

备注

如果本文未提及 TI 向量类型功能，但 C7000 编译器允许使用该功能，则 C7000 主机仿真也*允许*使用该功能。

5.1 构造函数

从 C7000 编译器 v3.0 开始，C7000 TI 编译器 (cl7x) 接受构造函数样式的语法来进行向量初始化。若要创建使用 cl7x 和主机仿真进行编译的可移植代码，请对向量初始化使用构造函数样式语法，而不是仅适用于 cl7x 的“cast/scalar-widening”初始化样式。

以下代码中显示了正确和错误的向量构造函数初始化语法示例。

```
/* Host Emulation vector constructor syntax examples */
// The following examples work for both cl7x and Host Emulation
long2 ex1 = long2(1);                  // -> (1,1)
long2 ex2 = long2(1,2);                // -> (1,2)
long8 ex3 = long8(long4(1), long4(2)); // -> (1,1,1,1,2,2,2,2)
long8 ex4 = long8(long4(1),2,3,4,5);   // -> (1,1,1,1,2,3,4,5)

// Do not use the following syntax for code that needs to compile
// with Host Emulation. This is valid C++ syntax, but results are
// not as expected when compiling with Host Emulation.
//long8 ex5 = (long8)(1,2,3,4,5,6,7,8); // -> (8,8,8,8,8,8,8,8) [for Host Emulation]
```

5.2 访问器

C7000 主机仿真为访问器提供以下支持：

- 支持单一元素访问器，如 .s0()。
- 支持半向量访问器，如 .lo() 和 .even()。
- 支持复杂访问器，如 .r()。
- 支持下标访问器，如 .s[0]。
- 多元素重排访问器（例如 .s0312()）*不受支持*。这是因为重排访问器的组合太多，并且不可能为所有这些组合都提供定义。权变措施涉及使用以下某个习语。

```
int2 my_new_vec = int2(my_int8_vec.x(), my_int8_vec.z()); // instead of my_vec.xz()

/* Swizzle accessor example workaround in Host Emulation code */
int16 ex = int16(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);

// Desired, but illegal when using Host Emulation:
// int8 swizzle = ex.s048c159d()
// Potential workaround:
int8 swizzle = int8(ex.even().even(), ex.odd().even());
```

5.3 向量运算符

主机仿真支持所有向量运算符，但向量三元运算符（即，条件表达式为向量时的运算符）除外。

所有其他运算符实现均遵循 *C7000 C/C++ 优化编译器用户指南* (SPRUIG8)。非法使用运算符会导致编译器错误。但是，收到的消息类型可能会有所不同。在少数情况下，非法使用部分运算符会导致在编译时出现断言错误，而不是传统的编译器错误。

嵌套子向量访问应使用函数调用语法指定。例如，针对主机仿真进行编译时，`vect.lo().lo()` 是合法的，但 `vect.lo.lo` 不是。从 C7000 编译器 v3.0 开始，针对主机仿真编译的子向量的嵌套深度没有限制。

5.4 打印调试函数

C7000 主机仿真提供了打印函数，可用于任何 TI 向量类型。此函数输出向量内容的格式化列表。此函数特定于 C7000 主机仿真，C7000 编译器不支持此函数。因此，为了使用 C7000 编译器进行编译，必须通过检查 __C7X_HOSTEM__ 预处理器符号来省略或保护对此函数的引用。以下示例演示了如何在向量的不同访问器级别使用打印函数。

```
/* Print function usage */
#ifndef __C7X_HOSTEM__
void print(int* ptr, int length)
{
    // Loop over elements and print
}
#endif

int8 example = int8(int4(0), int4(1));

#ifdef __C7X_HOSTEM__
example.print();                // Prints: (0,0,0,0,1,1,1,1)
example.lo().print();           // Prints: (0,0,0,0)
example.hi().lo().print();      // Prints: (1,1)
example.even().print();         // Prints: (0,0,1,1)
example.even().hi().print();    // Prints: (1,1)
//example.s0().print();         // Illegal, member .s0 is a scalar value

__vload_dup(&example).print();  // Prints (0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1)

#else
// Target implementation

// NOTE: Output depends on print() implementation
print((int*)(&example), 8);     // 0,0,0,0,1,1,1,1

// Error, can't take the address of a swizzle
//print((int*)(&example.hi()), 4);

// Option 1, preferred
int4 result_int4 = example.hi();
print((int*)(&result_int4), 4);  // 1,1,1,1

// Option 2
print((((int *)&example)+2), 4); // 0,0,1,1

int16 result_int16 = __vload_dup(&example);
print((int*)&result_int16, 16);  // 0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1
#endif
```

6 流引擎和流地址生成器

C7000 主机仿真流引擎 (SE) 和流地址生成器 (SA) 接口与 C7000 编译器相同。

7 查询表和直方图接口

C7000 主机仿真查询表 (LUT) 和直方图 (HIST) 接口与 C7000 编译器相同。`c7x_luthist.h` 中提到的任何内在函数或定义也在 C7000 主机仿真中定义和实现，并且可以以相同的方式使用。

7.1 查询表和直方图数据

使用 C7000 主机仿真时，会分配 32K 部分存储器来表示 C7000 的 L1D 高速缓存，用于 LUT 和 HIST 操作。在正常情况下，不应直接使用符号 `lut_sram`。直接访问 `lut_sram` 类似于直接访问 C7000 的 L1D 高速缓存，但这是禁止的。但是，该符号可用于调试目的。

8 C6000 迁移

c6x_migration.h 中定义的所有内在函数和数据类型均可在 C7000 主机仿真中用于迁移旧代码。映射到 C7000 指令或指令集的所有内在函数的用法与其在 C7000 编译器中的用法相同。但是，如节 4.4 中所述，在 C7000 主机仿真中使用传统类型时存在一些限制。

以下各部分仅重点介绍在 C7000 编译器中使用旧代码与使用 C7000 主机仿真之间的差异。*C6000 到 C7000 迁移用户指南* (SPRUIG5) 包含有关将 C6000 程序迁移到 C7000 的详细信息。

8.1 __float2_t 传统数据类型

对于 C7000 编译器，__float2_t 传统类型始终被视为 double。这对于 C7000 编译器有效，因为 double 是 64 位宽，可以容纳两个 32 位浮点元素以用于 SIMD 运算。

如果使用在 Intel x86 架构上执行的主机系统，则情况并非如此。在 Intel x86 计算机上执行双精度型的加载和存储时，会发生自动转换，将 64 位双精度型转换为 80 位“扩展实数”类型。当使用双精度型来存储两个不同的 32 位浮点值时，这会带来问题，因为 80 位“扩展实数”类型会进行规范化，从而改变存储器中的位。如果对表示两个 32 位浮点类型的 double 进行规范化的 80 位类型扩展，则无法再保证数据，并且期望两个浮点值的 SIMD 运算将产生不一致的结果。

为解决此问题，C7000 主机仿真针对 __float2_t 类型包含一个单独的类定义，该定义被视为不透明容器类型。容器类型只能使用特殊的内在函数进行修改、访问和初始化。虽然 __float2_t 类定义包含公共访问器方法，但建议仅使用内在函数来修改 __float2_t 类型，因为 C7000 主机仿真 __float2_t 类型的任何成员都不会在 C7000 编译器中定义。当传统内在函数中需要表示两个 32 位浮点值的单个数据结构时，应使用 __float2_t 类类型。在编写利用 C6000 旧版构造的 C7000 主机仿真代码时，double 类型只能用于表示一个双精度浮点值。

由于 __float2_t 类型有单独的定义，因此必须使用 _ftof2 内在函数来构造 __float2_t 类型。对于 C7000 编译器，此内在函数定义为 _ftod，它根据两个浮点指针参数创建一个 double 类型。__float2_t 的访问器方法以相同的方式定义。

表 8-1 列出了明显定义用于 C7000 主机仿真的内在函数。尽管此表中所列内在函数的定义有所不同，但为 C7000 主机仿真编写的旧代码无需更改即可传输到 C7000 编译器。

表 8-1. 主机仿真中具有不同定义的传统内在函数

内在函数名称	以前的定义	函数
_ftof2	_ftod	从 2 个浮点值构造 __float2_t 类型
_lltof2	_lltod	将 long long 值转换为 __float2_t 类型
_f2toll	_dtoll	将 __float2_t 类型转换为 long long
_hif2	_hif	从 _float2_t 类型访问高 32 位 float
_lof2	_lof	从 _float2_t 类型访问低 32 位 float
_fdmv_f2	_fdmv	使用 PACK 指令从 2 个 float 构造 __float2_type 的替代方法
_fdmvd_f2	_fdmvd	使用 PACKWDLY4 指令从 2 个 float 构造 __float2_type 的替代方法
_hif2_128	_hid128	从 __x128_t 类型访问高 __float2_t 类型
_lof2_128	_lod128	从 __x128_t 类型访问低 __float2_t 类型
_f2to128	_dto128	根据 2 个 __float2_t 类型构造 _x128_t 类型

以下示例以注释中所示的有效和无效方式构造和设置 __float2_t 变量。

```
/* __float2_t type examples: Host Emulation Code */

#include <c7x.h>
#include <c6x_migration.h>

int main(){
    // Valid ways to construct a __float2_t
    __float2_t src1 = _ftof2(1.1022, 2.1010);
```

```
    __float2_t src2 = _ftof2(-1.1, 4.10101);

    // Invalid way to construct a __float2_t in Host Emulation
    // __float2_t from_double = (double)1.0;

    // Legal to set a __float2_t from other pre-constructed
    // __float2_t types (done using intrinsic)
    src1 = src2;

    // It is illegal to set a __float2_t type via a
    // constructor call. The following will not compile:
    // src1 = __float2_t(1.0, 2.0);

    // Correct way to access lo/hi
    float lo_correct = _lof2(src1);

    // Intrinsic use example
    __float2_t res = _daddsp(src1, src2);

    return 0;
}
```

9 矩阵乘法加速器 (MMA) 接口

C7000 主机仿真矩阵乘法加速器 (MMA) 接口与目标硬件上 C7000 编译器使用的接口相同，但有一个重要差异。`c7x_mma.h` 中提到的所有内在函数和定义也都是针对 C7000 主机仿真定义和实现的，并且可以以相同的方式使用。但是，程序必须通过调用所提供的 _HWAADV() 内在函数来显式指明 MMA 状态何时推进。这是因为与目标硬件不同，为主机仿真的 MMA 不能与 CPU 时钟的概念联系起来。

程序必须跟踪旨在并行执行的指令，并通过在每组“并行”指令之后调用 __HWAADV() 来显式推进 MMA 状态。

为了方便在主机模式和目标模式之间轻松移植，目标编译器将 __HWAADV() 内在函数定义为空宏。

10 编译器错误和警告

使用 C7000 主机仿真对 C7000 进行编程时，编译器错误和警告将与使用 C7000 编译器编译相同代码时所看到的错误和警告不同。由于 C7000 的某些功能在主机仿真中的实现很复杂，以下各节定义了一些必要的关键术语，以帮助辨别您可能会看到的某些主机仿真编译器错误。

本节还讨论了尝试使用 C7000 主机仿真特定语法和结构时可能发出的主机仿真编译器错误和警告，还描述了可能不会触发编译器错误或警告的情况。

10.1 编译器错误和警告中包含的关键术语

在处理 TI 向量构造函数时，编译器错误和警告可能会引用不同的类及其各自的成员。表 10-1 列出了这些关键术语及其用途。

这些类型的错误的唯一真正区别是，它将包含类似“_c70_he_detail::vtype<int, 16ul, **(_c70_he_detail::VTYPE_KIND)0**>”的内容，而非类似“_c70_he_detail::vtype<int, 16ul>”的内容。

表 10-1. 向量相关编译器错误和警告中的关键术语

术语	用途	示例错误/警告
_c70_he_detail	包含所有向量类和运算符的命名空间	“Error: could not convert ‘_c70_he_detail::vtype<long int, 8ul, (_c70_he_detail::vtype_kind)0>(0)’ …”
Vtype	高级向量类名称	“Error: conversion from ‘int2 {aka _c70_he_detail::vtype<int, 2ul, (_c70_he_detail::vtype_kind)0>]’ …”

10.2 主机仿真特定语法

C7000 主机仿真同时引入并省略了 C7000 编译器使用的一些语法。虽然本文档通篇详细说明了这些差异，但在所有这些情况下都不能依赖主机仿真编译器发出警告和错误。这是因为 C7000 编译器允许的一些原始语法构成了合法的 C++ 代码，而主机仿真编译器则没有理由警告用户。虽然在某些情况下使用原始 C7000 编译器语法在语法上可能是正确的，但无法始终保证结果。表 10-2 列出了在将原始 C7000 语法与 C7000 主机仿真搭配使用时可能出现的主机编译器错误和警告或缺失。

表 10-2. 语法更改相关的编译错误和警告

说明	示例	编译器输出

将 cl7x 转换样式构造函数语法与主机仿真配合使用	`// Works with cl7x but not Host Emulation (long8)(1,2,3,4,5,6,7,8)`
	与
	`// Recommended; works on Host Emulation and cl7x long8(1,2,3,4,5,6,7,8)`
		无错误或警告。使用转换样式构造函数语法时，结果不正确或产生意外结果。

以向量作为“布尔表达式”的三元运算符	`res = vec1 ? vec2 : vec3`	编译器错误：“无法将 vec_type 转换为 bool”。
将重排访问器与成员数据语法结合使用	`example.s0121`	编译器错误：“成员不存在”。
将重排访问器与函数数据语法结合使用	`example.a0121()`	编译器错误：“成员函数不存在”
不对访问器使用函数语法	`...= vect.s0;`	编译时错误。
在 SE/SA 参数中使用无效值	将 VECLEN 设置为负数	在运行时，主机仿真将指出哪个字段无效。

11 修订历史记录

注：以前版本的页码可能与当前版本的页码不同

Changes from DECEMBER 15, 2023 to MARCH 15, 2024 (from Revision J (December 2023) to Revision K (March 2024)) **Page**

- 更新了用于构建 C7000 主机仿真的编译器版本 3

Changes from AUGUST 5, 2022 to DECEMBER 15, 2023 (from Revision I (August 2022) to Revision J (December 2023)) **Page**

- 更正了多元素重排访问器权变措施的示例 7
- 更正了示例代码 8

Changes from OCTOBER 22, 2021 to AUGUST 5, 2022 (from Revision H (October 2021) to Revision I (August 2022)) **Page**

- 添加了 __C7X_HOSTEM__、__C7120__ 和 __C7504__ 预处理器符号 4
- 简化了示例程序语法（原因是 cl7x 编译器目前接受用于向量初始化的构造函数样式语法） 5
- 本机向量类型现在称为“TI 向量类型”。此外，与复数向量类型相关的一些限制也已得到解决。有关详细信息，请参阅 *C7000 C/C++ 优化编译器用户指南* 7
- 目前，cl7x 和主机仿真编译器均接受使用构造函数样式语法进行向量初始化 7
- 阐明了支持的访问器类型，并更新了重排访问器的权变措施 7
- 修改了示例代码 8
- 更新了与向量相关的错误和警告的关键术语表 14
- 更新了与语法更改相关的错误和警告表格 14

Changes from MARCH 15, 2021 to OCTOBER 22, 2021 (from Revision G (March 2021) to Revision H (October 2021)) **Page**

- 向向量和复数元素指针类型添加了指针比较运算符列表 7

Changes from DECEMBER 31, 2020 to MARCH 15, 2021 (from Revision F (December 2020) to Revision G (March 2021)) **Page**

- 允许 2017 年之后的 Visual C++ 编译工具版本 3
- 提供了与发布和调试兼容的静态运行时库，以便与 Microsoft Visual C++ 一起使用。 3
- 控制寄存器的初始值现在设置为硬件复位中使用的值，而不是将所有控制寄存器值设置为 0。 6

Changes from MAY 1, 2020 to DECEMBER 31, 2020 (from Revision E (May 2020) to Revision F (December 2020)) **Page**

- 更新了整个文档中的表格、图和交叉参考的编号格式。 2
- 请注意，主机仿真支持是一个实验性产品，应考虑其局限性。 2
- 在 g++ 编译器的命令行中添加了 -fno-strict-aliasing 选项。更正了示例代码中的错误 5
- 向量和复数元素指针类型不支持数组访问运算符 7
- C7000 主机仿真不支持向量三元运算符。 8
- 修改了示例代码 8

表 12-1. 将 2020 年 1 月 28 日更改为 2020 年 5 月 1 日 (从修订版本 D 更改为修订版本 E)

添加内容的版本	位置	注意事项
SPRUIG6E	节 1.1	向相关文档列表添加了 *C7x 指令指南* 和其他文档。
SPRUIG6E	--	为加载和存储内在函数的不兼容参数类型添加了信息和权变措施。
SPRUIG6E	节 5	使用主机仿真时，向量数据类型需要额外的存储器。
SPRUIG6E	节 5.3	使用主机仿真时，嵌套子向量的深度限制为 2。

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